

AN $O(N)$ DIRECT SOLVER FOR INTEGRAL EQUATIONS ON THE PLANE

EDUARDO CORONA [†], PER-GUNNAR MARTINSSON [‡], AND DENIS ZORIN [†]

Abstract. An efficient direct solver for volume integral equations with $O(N)$ complexity for a broad range of problems is presented. The solver relies on hierarchical compression of the discretized integral operator, and exploits that off-diagonal blocks of certain dense matrices have numerically low rank. Technically, the solver is inspired by previously developed direct solvers for integral equations based on “recursive skeletonization” and “Hierarchically Semi-Separable” (HSS) matrices, but it improves on the asymptotic complexity of existing solvers by incorporating an additional level of compression. The resulting solver has optimal $O(N)$ complexity for all stages of the computation, as demonstrated by both theoretical analysis and numerical examples. The computational examples further display good practical performance in terms of both speed and memory usage. In particular, it is demonstrated that even problems involving 10^7 unknowns can be solved to precision 10^{-10} using a simple Matlab implementation of the algorithm executed on a single core.

1. Introduction. Many boundary value problems from classical physics have a boundary integral representation of the form

$$(1.1) \quad a(x)\sigma(x) + \int_{\Gamma} \mathcal{K}(x, y)\sigma(y)dS(y) = f(x), \quad \forall x \in \Gamma,$$

where Γ is a domain in either \mathbb{R}^2 or \mathbb{R}^3 (either a boundary or a volume) and $\mathcal{K}(x, y)$ is a kernel function derived from the fundamental solution associated with the elliptic PDE (e.g. the Laplace or Helmholtz equation, the Stokes equations etc.). The kernel function is locally singular near the diagonal (as y approaches x) but is otherwise smooth.

Discretizing the integral in (1.1) using, e.g., the Nyström method [1] results in a linear system of the form

$$(1.2) \quad A\sigma = f,$$

where A is a dense $N \times N$ matrix. For many problems it has been known since the 1980's show analytically that the matrix-vector product $\sigma \mapsto A\sigma$ using $O(N)$ operations (see, e.g., [21]). Such matrix-vector multiplication techniques can be employed in an iterative solver (e.g. GMRES [34] or Bi-CGSTAB [37]) to obtain a solution for (1.1) that can be as effective as convergence is rapid. More recently *direct solvers* in linear complexity have been developed for (1.2) for the case when Γ is a contour in the plane [31]. When the method of [31] is applied to a volume domain in the plane, a reasonably efficient direct solver in asymptotic complexity $O(N^{3/2})$ is [17, 18, 26]. This paper describes how the algorithm of [31] can be extended to construct a direct solver for volume integral equations in two dimensions that has optimal $O(N)$ complexity and high practical efficiency (even at high accuracies such as 10^{-10}). We believe that this solver will also (with some additional work) be capable of directly solving boundary integral equations for problems in 3D in $O(N)$ or $O(N \log N)$ operations and will be well suited for implementation on multicore and parallel computers.

1.1. Previous Work. The key observation enabling the construction of $O(N)$ algorithms for solving linear systems arising from the discretization of integral equations is that the off-diagonal blocks of the coefficient matrix can be well approximated by matrices of low numerical rank. In this section, we briefly describe some key observations.

Optimal complexity iterative solvers for integral equations. By combining the observation that off-diagonal blocks of the matrix have low rank with a hierarchical partitioning of the physical domain into a tree-structure, algorithms of $O(N)$ or $O(N \log N)$ complexity for matrix-vector multiplication were developed in the 1980's. Perhaps the most prominent is the Fast Multipole Method [21, 22, 33], but the Panel Clustering [23] and Barnes-Hut [2] methods are also well known. When a fast algorithm for the matrix-vector multiplication such as the FMM is included in an iterative method such as e.g., GMRES [34] or Bi-CGSTAB [37], the result is a solver for the linear system arising upon discretization

[†]Courant Institute of Mathematical Sciences, New York University

[‡]Department of Applied Mathematics, University of Colorado at Boulder

of (1.1) in overall complexity $O(MN)$, here M is the number of strips by the iterative solver. In many situations M is independent of N and convergence can be rapid.

The first FMMs were specifically designed for elliptic equations (e.g. Laplace, Helmholtz Stokes), but it is later realized that *kernel-independent methods* have taken for broad classes of problems could be developed [19,42]. They directly inspired the work described in this paper and the new direct solvers can also be said to be “kernel-independent” in the sense that the same algorithm, and the same code, can be applied to several different types of physical problems such as electrostatics Stokes flow and low frequency scattering.

While iterative solvers accelerated by fast methods format *low-complexity* can be very effective in many contexts their performance is held hostage to the convergence rate of the iteration. If the equation is not well-conditioned, the complexity of an iterative solver may increase. Pre-conditioners can sometimes be considered to accelerate convergence, but these tend to be *problem-specific* and do not readily lend themselves to the construction of general purpose codes. Examples of ill-conditioned problems that can be challenging to iterative methods include Fredholm equations of the first kind, elasticity problems on thin domains and scattering problems near resonances.

Direct solvers for integral equations. In the last year a number of efficient *direct solvers* for linear systems associated with integral equations have been considered. These solvers tend to address the challenges related to convergence speed of iterative solvers. They can also lead to dramatic improvements in speed, in particular in situations where several equations with the same coefficient matrix but different right hand sides need to be solved.

The work presented here draws heavily on [31], (based on on [36]), which describes a direct solver that was originally developed for boundary integral equations defined on curves in the plane, and has optimal $O(N)$ complexity for this case. The observation that this algorithm can also be applied to volume integral equations in the plane, and to boundary integral equations in 3D was made in [20] and later elaborated in [17]. For these cases the direct solvers require $O(N^{3/2})$ flops to build an approximation to the inverse of the matrix and $O(N \log N)$ flops for the “*solve stage*” once the inverse has been considered. Similar work was done in [26], where it is also demonstrated that the direct solver can, from a practical point of view be implemented using standard direct solvers for large *sparse matrices*. This impossibility and greatly simplifies the practical implementation due to the availability of standard packages such as UMFPACK.

The direct solver of [31] relies on the fact that the matrices arising from the discretization of integral equations can be efficiently represented in a data-*po* format often referred to as “Hierarchically Semi-Separable (HSS)” matrices. This matrix format was also proposed in [9,11], in a more recent efficient version presented in [40]. This work compares ULV and Choleky factorizations of HSS matrices if these techniques are applied to volume integral equations in 2D, the complexity would be $O(N^{3/2})$, in complete agreement with [20] and [17]. The paper [39] presents a general complexity analysis of HSS algorithms under different rank growth patterns, it presents an optimal-complexity HSS recomposition method which was adapted to constructing a part of the overall algorithm.

An important class of related algorithms is \mathcal{H} and \mathcal{H}^2 -matrix methods of Hackbusch and co-workers (see [3,6,7] for review). These techniques are based on variations of the *cos* approximation method for low-rank compression, and have been applied both to integral equations and problems derived from PDEs. The matrix factorization for two and three-dimensional problems algorithms are formulated explicitly and a full set of compressed operations for low-dimensional problems needs to be available. In [4,5], algorithms for \mathcal{H}^2 matrix arithmetic are described; Observed behavior for integral equation operations on the cube and on the sphere in chapter 10 of [7] is $O(N \log^4 N)$ for matrix compression, $O(N \log^3 N)$ for inversion and $O(N \log^2 N)$ for solve time and memory.

Direct solvers for sparse systems. Our direct solver is conceptually related to direct solvers for sparse matrices such as the classical *nested dissection* and *multifrontal* methods [15,16,27]. These solvers do not have optimal complexity (they are typically $O(N^{3/2})$ for the factorization stage in 2D, and $O(N^2)$ in 3D), but are nevertheless superior (especially in 2D) due to their robustness and in particular the mixed precision that can be attained for problems involving multiple right hand sides. Very recently it has been demonstrated that by exploiting fast matrix algebra (such as e.g., \mathcal{H} -matrices or HSS matrices), to manipulate the dense matrices that arise due to fill-in, direct

solves linear or close to linear complexity can be constructed [17, 28, 30, 35, 41]. The direct solver described in this paper is particularly similar to these algorithms in that they all rely on hierarchical domain decompositions and efficient representations of operators that live on the interfaces between sub-domains

1.2. Overview of new results. We present a direct solver that achieves optimal $O(N)$ complexity for two-dimensional problems defined from integral equations in non-oscillatory kernels or kernels in low-frequency mode. Similarly to other HSS or \mathcal{H} -matrix methods we rely on the fact that the problem is defined from an integral equation only weakly if the kernel is of a different nature, the solvability of the problem may deteriorate, but can still profit from accurate calculations

The main features of our solver include:

- Observed $O(N)$ complexity both in time and storage for all stages of the computation (in particular for both the “build” and the “solve” stages).
- The algorithm has high accuracy (up to $10^{-10} - 10^{-12}$ is practical), while maintaining reasonable efficiency both in time and memory
- The algorithm can take direct advantage of translation invariance and symmetry of underlying kernels achieving considerable speed and reduction of memory costs

The main aspects of the algorithm that allow us to achieve high performance are:

- Two levels of hierarchical structure. The matrix A is stored in the HSS format and certain blocks within the HSS structure are themselves represented in the HSS format
- Direct construction of the inverse: unlike many previous algorithms we directly build a compressed representation of the inverse, rather than computing the matrix itself and then inverting.
- Our direct solver needs only a list of a few HSS matrix arithmetic in particular the relatively expensive matrix-matrix multiplication is avoided.

We achieve significant gains in speed and memory efficiency in respect to the existing $O(N^{3/2})$ approach. For non-oscillatory kernels our algorithm performs the $O(N^{3/2})$ algorithm around $N \sim 10^5$. Sizes of up to $N \sim 10^7$ are practical on a desktop gaining one order of magnitude in inverse computation time and storage. For example, for non-translation-invariant kernels a problem of size $N = 3 \times 10^6$ and target accuracy $\epsilon = 10^{-10}$ takes 1 hour and ~ 50 GB of memory. Each solve takes 10 seconds. For translation-invariant kernels such as Laplace, a problem of size $N = 1 \times 10^7$ and target accuracy $\epsilon = 10^{-10}$ takes half an hour and 5 GB of memory in 20-second solves. Reducing target accuracy to $\epsilon = 10^{-5}$, inverse costs for the latter problem go down to 5 minutes and 1 GB of storage, and each solve takes 10 seconds

Our accelerated approach to build the HSS binary tree also yields a fast $O(N)$ matrix-compression algorithm. As is noted in [26], given that matrix-compression is orders of magnitude faster than one round of FMM, this algorithm could be preferable to an iterative method coded in FMM for problems that require more than a few iterations. To give an example, for $N = 10^7$ and $\epsilon = 10^{-10}$, matrix-compression takes 5 minutes and 1 GB of storage, and each matrix-compression takes less than 10 seconds

Finally we are able to apply our method in some minor modifications to oscillatory kernels in low-frequency mode, and apply it to the solution of the corresponding 2D volume scattering problems. Although the cost is considerably higher we still observe optimal scaling and similar performance gains

2. Background. Our approach builds on the fast direct solver in [18], which consists of sparse hierarchical compression, inverse and inverse application algorithms all of which achieve linear complexity when applied to integral operators on one dimensional curves. These algorithms can be applied to volume integral equations in 2D and to boundary integral equations defined on surfaces in 3D but compression and inversion then have $O(N^{3/2})$ complexity while application of the inverse is $O(N \log N)$. The $O(N)$ linear complexity under the technique is practical for large problems if it is possible to fully form small and moderate size problems see [17, 26].

In this section we review the algorithms for compression and inversion described in [18], as they are needed to describe our algorithms and discuss their limitations for problems on 2D domains

2.1. Notation and Preliminaries. We view an $N \times N$ matrix A as a kernel function $K = K(p, q)$ evaluated at pairs of sample points. As it usually comes from an integral formulation of an elliptic PDE, it obeys a low-rank block structure and exact Green's identities hold for K ; however, it is not a fundamental limitation of our algorithm: the use of Green's identities is restricted to one small part of the algorithm (Section 2.2) relying on eigenvalue separation and can be replaced by any other technique of similar nature.

We use a Matlab-like notation $A(I, J)$ where I and J are ordered sets of indices to denote blocks of a matrix A . Again, following Matlab conventions $A(:, J)$ and $A(I, :)$ indicate blocks of columns and rows respectively.

An essential building block for our algorithm is the interpolative low-rank decomposition. This decomposition factors an $m \times n$ matrix A into a narrow *skeleton* matrix $A^{\text{sk}} = A(:, I^{\text{sk}})$ of size $m \times k$, consisting of a list of columns of A indexed by the set of indices I^{sk} , and the interpolation matrix T of size $k \times (n - k)$, expressing the remaining columns of A as linear combinations of columns of A^{sk} : The set of indices I^{sk} is called the *column skeleton* of A . If Π^{sk} is the permutation matrix exchanging entries in the indices from I^{sk} to $\{1, \dots, k\}$

$$(2.1) \quad A = A^{\text{sk}} [I_{k \times k} \ T] \Pi^{\text{sk}} + E$$

where $\|E\|_2 \sim \sigma_{k+1}$ which decreases as k increases, and $R = [I_{k \times k} \ T] \Pi^{\text{sk}}$ is a down-sampling interpolation matrix. We denote this composition operation by $[A, I^{\text{sk}}] = ID(A, \varepsilon)$, where ε controls the norm of E . To obtain a similar composition for rowwise approximation to A^T ; in this case, we obtain a factorization $A = (\Pi_r^{\text{sk}})^T \begin{bmatrix} I_{k \times k} & T_r \\ & T_r^T \end{bmatrix} A_r^{\text{sk}} + E_r$, where $L = (\Pi_r^{\text{sk}})^T \begin{bmatrix} I_{k \times k} & T_r \\ & T_r^T \end{bmatrix}$ is an up-sampling interpolation matrix.

2.2. Constructing hierarchically semi-separable matrices. We assume that the domain of interest is contained in a rectangle Ω , in a regular grid of samples (if the input data is given in a different representation, we sample it first). A quadtree is constructed by recursively dividing Ω into cells (boxes B_i) by bisection, corresponding to the nodes of a binary tree \mathcal{T} . The *children* of a box are denoted $c_1(i)$ and $c_2(i)$. Subdivision can be done adaptively, where significant changes to the algorithm, but to simplify the exposition we focus on the uniform refinement case. Let I_i denote the index set comprising all discretization points in box B_i , and let \mathcal{L} denote a list of the leaf boxes. Then $\{I_i\}_{i \in \mathcal{L}}$ forms a disjoint partition of the full index set

$$(2.2) \quad \{1, 2, 3, \dots, N\} = \bigcup_{i \in \mathcal{L}} I_i.$$

Let m_i denote the number of points in B_i . The partition (2.2) corresponds to a blocking

$$(2.3) \quad \sum_{j \in \mathcal{L}} A_{ij} \sigma_j = f_i, \quad i \in \mathcal{L},$$

of the linear system (1.2), where $A_{ij} = A(I_i, I_j)$, and the vectors σ and f are partitioned accordingly. For a linear system such as (1.2) arising from the discretization of an integral equation with a smooth kernel, the off-diagonal blocks of (2.3) usually have low numerical rank. Such matrices can be represented in an efficient data-parallel format called *hierarchically semi-separable (HSS)*. In order to rigorously describe this format we first define the concept of a *block-separable matrix*.

DEFINITION 2.1 (Block-separable Matrices). We say A is *block-separable* if there exist matrices $\{L_i, R_i\}_{i \in \mathcal{L}}$ such that each off-diagonal block $A_{i,j}$ in (2.3) admits the factorization

$$(2.4) \quad A_{i,j} = \begin{matrix} L_i & M_{i,j} & R_j \\ m_i \times k_i & k_i \times k_j & k_j \times m_j \end{matrix},$$

where the block ranks k_i satisfy $k_i < m_i$.

In order to construct the matrices L_i and R_i in (2.4) it is helpful to introduce *block rows* and *block columns* of A : For $i \in \mathcal{L}$, we define the i th off-diagonal block row of A as $A_i^{\text{row}} = A(I_i, I \setminus I_i) =$

$[A_{i,1} \dots A_{i,i-1} A_{i,i+1} \dots A_{i,p}]$. The j th off-diagonal block column A_j^{col} is defined analogously. Given a prescribed accuracy ε , we denote by k_i^r and k_i^c the ε -rank of A_i^{row} and A_i^{col} , respectively.

Now what we have defined A_i^{row} and A_i^{col} , we can then obtain the factorization (2.4) as follows. For each $i \in \mathcal{L}$, form the interleaved decomposition of A_i^{row} and A_i^{col} :

$$(2.5) \quad A_i^{row} = \begin{matrix} L_i & A(I_i^{rsk}, :) \\ m_i \times k_i & k_i \times (N-m_i) \end{matrix} \quad \text{and} \quad A_i^{col} = \begin{matrix} A(I_i^{csc}, :) & R_i \\ (N-m_i) \times k_i & k_i \times m_i \end{matrix},$$

where the index vectors I_i^{rsk} and I_i^{csc} are the row-skeleton and column-skeleton of block i , respectively. Note that the column vector L_i is a column basis for A_i^{row} , and the row vector R_i is a row basis for A_i^{col} . Setting

$$M_{i,j} = A(I_i^{rsk}, I_j^{csc}),$$

we then find that (2.4) necessarily holds. Observe that each matrix $M_{i,j}$ is a submatrix of A . Set $D_i = A_{i,i}$. This yields a block factorization for A :

$$(2.6) \quad A = D^d + L^d A^{d-1} R^d$$

where D^d , L^d , and R^d are the block diagonal matrices whose diagonal blocks are given by $\{D_i\}_{i \in \mathcal{L}}$, $\{L_i\}_{i \in \mathcal{L}}$, and $\{R_i\}_{i \in \mathcal{L}}$, respectively. The matrix A^{d-1} is the sum of a block-diagonal matrix D^d and off-diagonal blocks $M_{i,j}$.

REMARK 2.2. Row and column skeleton sets need not coincide, although for all purposes, we will assume they are augmented so that they are the same size (so D_i blocks are square). If the system matrix is symmetric, as is the case for all matrices considered in this paper, these sets are indeed the same and further $R_i = L_i^T$. For this reason, as well as for the sake of simplicity, we make no further distinction between them unless it is necessary.

Hierarchical compression of A . The key property that allows matrix operations to be performed in less than $O(N^2)$ complexity is that the low-rank matrix in definition 2.1 can be exploited efficiently in the sense that the matrix A^{d-1} in (2.6) is block-separable. To be precise, we partition the matrix A^{d-1} by merging 2×2 sets of blocks into new larger blocks. Each larger block is associated with a box i on level $d-1$ corresponding to the index vector $I_{c_1(i)}^{rsk} \sqcup I_{c_2(i)}^{rsk}$ (the new index vector holds $m_i = k_{c_1} + k_{c_2}$ nodes). The resulting matrix with larger blocks is then itself block-separable and admits a factorization, cf. Fig 2.1,

$$(2.7) \quad A = D^d + L^d (D^{d-1} + L^{d-1} A^{d-2} R^{d-1}) R^d$$

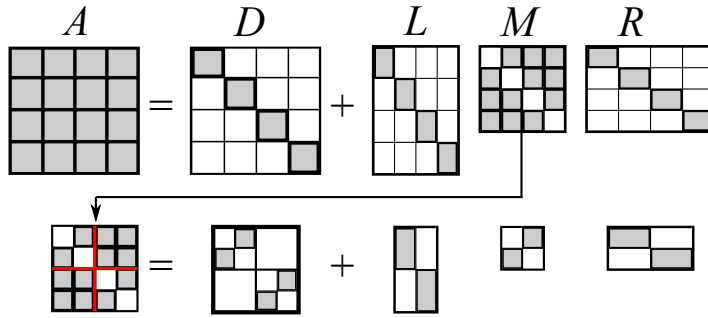


FIG. 2.1. Two levels of block-separable compression: blocks of M corresponding to children are merged and then off-diagonal interactions are further compressed.

We say that A is hierarchically semiseparable (HSS) if the process of blocking and compression can be continued through all levels of the tree. In other words, we say that $A^\ell = D^\ell + L^\ell A^{\ell-1} R^\ell$ for $\ell = d \dots 1$, or more explicitly

$$(2.8) \quad A^d = D^d + L^d (D^{d-1} + L^{d-1} (D^{d-1} + \dots (D^1 + L^1 D^0 R^1) \dots) R^{d-1}) R^d$$

in $A = A^d$, $A^0 = D^0$, and D^ℓ , L^ℓ and R^ℓ block-diagonal, in blocks in matrices in index ℓ corresponding to boxes at level ℓ . For non-leaf boxes blocks D_i^ℓ account for “ sibling interactions ” in other words interactions between the children $c_1(i)$ and $c_2(i)$ of B_i ,

$$(2.9) \quad D_i = \begin{bmatrix} & M_{c_1(i), c_2(i)} \\ M_{c_2(i), c_1(i)} & \end{bmatrix}$$

We call (2.8) the telescoping factorization of A . The matrices under consideration in this manuscript are (like most matrices arising from the discretization of integral operators) all HSS.

REMARK 2.3. In this paper we use the term hierarchically semi-separable (HSS) to conform with standard use in the literature, see [9–11, 40]. In [18, 26, 31] the term hierarchically block-separable (HBS) is alternatively used to refer to this hierarchical version of block-separability, consistent with definition 2.1.

Using equivalent densities to accelerate compression. A matrix is block-separable as long as all k -matrices A_i^{row} and A_i^{col} are rank-deficient. However, these matrices are large, so directly computing the IDs in (2.5) is expensive ($O(N^2)$ cost [39]). In this section, we describe how to exploit the fact that the matrix can be compressed as associated in an elliptic PDE to reduce the asymptotic cost. Related techniques are previously described in [42] and [31].

Let B_i denote a leaf box with associated index vector I_i . We will describe the accelerated technique for computing a matrix L_i and an index vector $I_i^{\text{rsk}} \subset I_i$ such that

$$(2.10) \quad A_i^{\text{row}} = L_i A(I_i^{\text{rsk}}, :)$$

holds to high precision. (The technique for finding R_i and I_i^{csk} such that (2.5) holds is an analogous). For concreteness we temporarily take the kernel \mathcal{K} is the fundamental solution of the Laplace equation, $\mathcal{K}(x, y) = \frac{1}{2\pi} \log|x - y|$. The idea is to construct a small matrix \tilde{A}_i^{row} in the proxy that

$$(2.11) \quad \text{Ran}(A_i^{\text{row}}) \subseteq \text{Ran}(\tilde{A}_i^{\text{row}}).$$

In other words the columns of \tilde{A}_i^{row} need to span the columns of A_i^{row} . Then compute an ID of the small matrix \tilde{A}_i^{row} ,

$$(2.12) \quad \tilde{A}_i^{\text{row}} = L_i \tilde{A}(I_i^{\text{rsk}}, :).$$

Now (2.11) and (2.12) together imply that (2.10) holds.

It remains to construct a small matrix \tilde{A}_i^{row} whose columns span the range of A_i^{row} . To do this we note that $v \in \text{Ran}(A_i^{\text{row}})$, so that for some vector $q \in \mathbb{R}^{N-m_i}$ $v = A_i^{\text{row}} q$. Physically this means that the values of v represent values of a harmonic function generated by sources q located inside the box B_i . We now know from potential theory that any harmonic function in B_i can be replicated by a source density on the boundary ∂B_i . The discrete analog of this statement is that to high precision, we can replicate the harmonic function in B_i by placing point charges in a thin layer of discretization nodes surrounding B_i (drawn as solid diamonds in Figure 2.2(b)). Let $\{z_j\}_{j=1}^{p_i}$ denote the locations of these pins. The claim is then that v can be replicated by placing some “ equivalent charges ” at these pins. In other words we form \tilde{A}_i^{row} as the matrix of size $m_i \times p_i$ whose entries take the form $\mathcal{K}(x_r, z_j)$ for $r \in I_i$, and $j = 1, 2, \dots, p_i$.

REMARK 2.4. Figure 2.2 shows an example of how accelerated compression works. Figure 2.2(a) illustrates a domain Ω with a sub-domain B_i (the dotted box). Suppose that φ is a harmonic function on B_i . Then potential theory assures us that φ can be generated by sources on ∂B_i , in other words $\varphi(x) = \int_{\Gamma} \mathcal{K}(x, y) \sigma(y) ds(y)$ for some density σ . The discrete analog of this statement is that to high precision, the harmonic function φ can be generated by placing point charges on the proxy points $\{z_j\}_{j=1}^{p_i}$ marked with solid diamonds in Figure 2.2(b). The practical consequence is that instead of factoring the big matrix A_i^{row} which represents interactions between all target points in B_i (circles) and all source points (diamonds), it is enough to factor the small matrix \tilde{A}_i^{row} representing interactions between target points (circles) and proxy points (solid diamonds).

REMARK 2.5. The width of the layer of proxy points depends on the accuracy requested. We found that for the Laplace kernel, a layer of width 1 leads to relative accuracy about 10^{-5} , and width 2 leads to

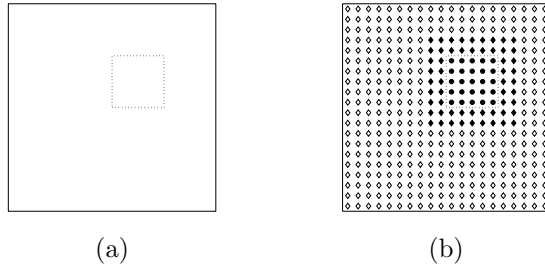


FIG. 2.2. (a) A domain Ω (solid) with a sub-domain B_i (dotted). (b) Target points in B_i are circles, source points are diamonds, and among the source points, the proxy points are solid.

relative accuracy 10^{-10} . For the Helmholtz kernel $\mathcal{K}(x, y) = H_0^{(1)}(\kappa|x - y|)$, similar accuracy is typically observed, but in this case, thicker skeleton layers are recommended to avoid problems associated with resonances. (Recall that in classical potential theory, a solution to the Helmholtz equation may require both monopole and dipole charges to be placed on ∂B_i .)

2.3. HSS matrix-vector multiplication. To describe the process of computing the inverse of an HSS matrix composed from its coefficients, we explain how matrix-vector products can be computed. The telescoping factorization (2.8) yields a fast matrix-vector algorithm evaluating $u = A\sigma$. The structure of this algorithm is similar to FMM (but simpler as we do not take the near field spatially approximating all internal interactions in a single step of coefficients). To emphasize the underlying physical intuition, we refer to the charges σ at pins as charges and to the values u at pins as potentials. On non-leaf boxes, we use the notation ϕ_i^ℓ for the charges assigned to the skeleton pins of the box and $u_i^{up, \ell}$ for computed potentials. The vector $\phi^\ell (u^\ell)$ is the concatenation of all charges (potentials) of the boxes at level ℓ . At the finest level, we define $\phi^d := \sigma$.

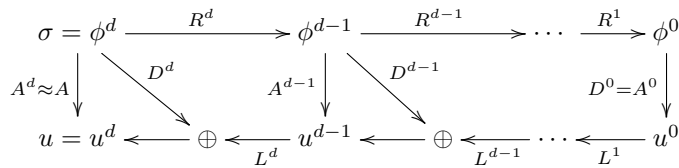
Upward pass. The upward pass is the rectangular block-diagonal matrix to compute ϕ^ℓ : $\phi^\ell = R^{\ell+1} \phi^{\ell+1}$. Each block $R_i^{\ell+1}$ acts on the vector of charges corresponding to the children of B_i at level ℓ .

Downward pass. We compute a potential u_i for each box due to all outside charges using fast multipole methods. For the top level boxes B_1 and B_2 , the values are computed directly using the sibling interaction matrix forming D^0 as defined by (2.9), in other words $u^1 = D^0 \phi^1$.

For boxes at the level $\ell > 1$, the field on the boxes is obtained as the sum of the fields interpolated to the boxes at level ℓ using L^ℓ ("all" rectangular block-diagonal) and contributions of the siblings through the block diagonal matrix D^ℓ :

$$u^\ell = D^\ell \phi^\ell + L^\ell u^{\ell-1}$$

At the leaf level, the leaves are inserted into the simple pins and add the field due to boxes themselves (self-interactions seen in the diagonal blocks $A_{i,i}$ of D^d). The actions of different transformations are summarized in the following computational flow diagram:



2.4. Computing the HSS form of A^{-1} . If A is non-singular, the telescoping factorization (2.8) can be inverted directly yielding an HSS representation of A^{-1} , which can be applied efficiently using the algorithm described in Section 2.3. The inversion process is also based on terms of the variables introduced in Section 2.3.

We derive a formula for inversion of matrices using single-level telescoping factorization $Z = F + LMR$. (The matrix Z on the left coincides with $A = A^d$, and $F = D^d$, but both Z and F are different from A^ℓ and D^ℓ on subsequent

We consider the system $Z\sigma = f$, and define $\phi = \phi^{d-1} = R\sigma$, $u = u^{d-1} = M\phi$. We then perform block-Gaussian elimination on the enlarged block system:

$$(2.13) \quad \begin{bmatrix} F & L & 0 \\ -R & 0 & I \\ 0 & -I & M \end{bmatrix} \begin{bmatrix} \sigma \\ u \\ \phi \end{bmatrix} = \begin{bmatrix} f \\ 0 \\ 0 \end{bmatrix}$$

We form the auxiliary matrices $E = RF^{-1}L$, and $G = E + M$. Then

$$(2.14) \quad \begin{bmatrix} F & 0 & 0 \\ 0 & E^{-1} & 0 \\ 0 & 0 & G \end{bmatrix} \begin{bmatrix} \sigma \\ u \\ \phi \end{bmatrix} = \begin{bmatrix} [I - LERF^{-1} + LEG^{-1}ERF^{-1}]f \\ [RF^{-1} - G^{-1}ERF^{-1}]f \\ ERF^{-1}f \end{bmatrix}$$

which yields the inverse of Z by inverting the block-diagonal system in the first line. Denoting $\tilde{R} = ERF^{-1}$, $\tilde{D} = F^{-1}(I - L\tilde{R})$ and $\tilde{L} = F^{-1}LE$, we obtain:

$$(2.15) \quad Z^{-1} = \tilde{D} + \tilde{L}(E + M)^{-1}\tilde{R}$$

We make a few observations

- If D, L and R are block diagonal, then so are E, \tilde{R}, \tilde{L} and \tilde{D} . This means that these matrices can be computed independently via independent computations that are local to each box
- The facts in the inverse can be interpreted as follows
 - $E^{-1} = RF^{-1}L$ can be viewed as a “local solution operator” defined to the skeleton points for each box in a field to change densities on the box
 - $E + M$ maps change densities on the union of skeleton points to fields adding diagonal $E\sigma$ and off-diagonal $M\sigma$ contributions

This system procedure can be applied recursively. To obtain a recursion formula, we define an auxiliary matrix $\tilde{A}^\ell = A^\ell + E^{\ell+1}$, for $\ell < d$, and $\tilde{A}^d = A^d = A$; $F^\ell = D^\ell + E^{\ell+1}$, and $F^d = D^d$. Then \tilde{A}^ℓ satisfies $\tilde{A}^\ell = F^\ell + L^\ell A^{\ell-1} R^\ell$, and is inverse by (2.15) satisfies

$$(2.16) \quad (\tilde{A}^\ell)^{-1} = \tilde{D}^\ell + \tilde{L}^\ell (E^\ell + A^{\ell-1})^{-1} \tilde{R}^\ell = \tilde{D}^\ell + \tilde{L}^\ell (\tilde{A}^{\ell-1})^{-1} \tilde{R}^\ell$$

A fine-to-coarse procedure for computing the blocks of the inverse immediately follows from (2.16): for each layer ℓ , we first compute F^ℓ , using $E^{\ell+1}$ for the finer layer (or for the finest). F^ℓ determines \tilde{D}^ℓ , \tilde{R}^ℓ and \tilde{L}^ℓ , and E^ℓ , to be used at the next layer.

Algorithm 1 summarizes the HSS inverse algorithm; it takes as input the \mathcal{T} in indices I_i defined for leaf nodes and matrices D_i, R_i and L_i for each box B_i , and computes components \tilde{D}_i, \tilde{R}_i and \tilde{L}_i of the HSS form of A^{-1} . This algorithm has complexity $O(N^{3/2})$ for a volume integral equation in 2D. It forms the starting point from which we derive an $O(N)$ algorithm in Section 3.

The hierarchical structure of this algorithm compares to the same as that of matrix A : \tilde{L}, \tilde{R} are not triangular matrices and \tilde{D} has non-zero diagonal blocks. However, it can be converted to a standard HSS format if needed in the simple re-formatting algorithm in [17, 18].

3. $O(N)$ Inverse Compression Algorithm. When applied to boundary integral equations (BIEs) in 2D, Algorithm 1 has optimal $O(N)$ asymptotic complexity for non-oscillatory kernels and a broad range of geometries. However, for volume integral equations in 2D, the local complexity is $O(N^{3/2})$.

In this section, we first show why the algorithm has higher complexity for volume problems and then develop a fast algorithm in $O(N)$ by first modifying the inverse algorithm to operate on the essential blocks of A^{-1} , which can in turn be viewed as a compressed operator acting on 1D-like sets of points and avoid the need to store a non-sparse matrix in the algorithm. Then we demonstrate how these operators can be compressed into one-dimensional HSS structures.

3.1. Efficient skeleton construction in 2D.

Algorithm 1 HSS matrix-vector

```

1: for each box  $B_i$  in fine-to-coarse order do
2:   if  $B_i$  is a leaf then
3:      $F_i = D_i$ 
4:   else
5:      $F_i = D_i + \begin{bmatrix} E_{c_1(i)} & \\ & E_{c_2(i)} \end{bmatrix}$ 
6:   end if
7:   if  $B_i$  is a level  $\ell$  then
8:      $\tilde{F}_i = F_i^{-1} \{ \text{Direction at the level } \ell \}$ 
9:   else
10:    Compute  $F_i^{-1}$ 
11:     $E_i = (R_i F_i^{-1} L_i)^{-1}$ 
12:     $\tilde{R}_i = E_i R_i F_i^{-1}$ 
13:     $\tilde{D}_i = F_i^{-1} (I - L_i \tilde{R}_i)$ 
14:     $\tilde{L}_i = F_i^{-1} L_i E_i$ 
15:   end if
16: end for

```

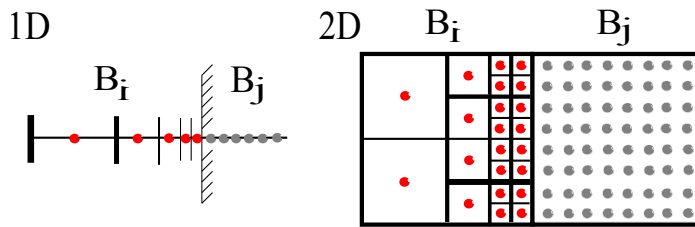


FIG. 3.1. Interaction ranks in 1D and 2D. Source box B_i is recursively subdivided into well-separated sets, whose interaction with B_j is constant rank. This provides an upper bound for the overall interaction rank.

Skeleton size scaling. Let us consider a non-oscillatory kernel (e.g. Laplace) the rank of the interaction between two neighboring boxes in D dimensions is $D = 1, 2$. This example captures essential behavior and is of skeleton sets in different dimensions (Figure 3.1)

Let B_i be a box at level ℓ of the tree in D dimensions with $m_i \sim n_\ell = N/2^\ell$ points. We estimate the ε -rank k_i of the interaction of B_i with a box of the same size adjacent to it. If sources and targets are well-separated (the distance between the standard points is at least the standard deviation), the rank of their interaction matrix can be bounded by a constant p for a given ε . We partition a given subdivision of box B_i into well-separated sets which contain p points of less. Then

$$k_i \sim p \sum_{s=0}^{\log_2(n_\ell/p)/D} 2^{(D-1)s}$$

For $D = 1$, we have $\log_2(n_\ell/p)$ intervals and so $k_i \sim O(\log_2(n_\ell))$. For $D = 2$, we divide $\log_4(n_\ell/p)$ times in the direction normal to the hard edge of the boxes getting 2 well-separated boxes. Then, $k_i \sim 2^{\log_2(n_\ell/p)/2} = O(n_\ell^{1/2})$. A simple calculation shows this yields an estimate $O(N)$ for the complexity of the 1D inverse algorithm, (the logarithmic rank growth in boxes does not affect the complexity because the number of boxes grows exponentially and $O(N^{3/2})$ in the D -dimensional case.

Structure of skeleton sets in 2D. The HSS hierarchical compression procedure constructs skeleton sets for boxes at all levels

The algorithm of Section 2.2 with constant skeletons for leaf boxes using interleaved decomposition of block-orthogonal matrices for acceleration. From leaf boxes at level

ℓ , indices

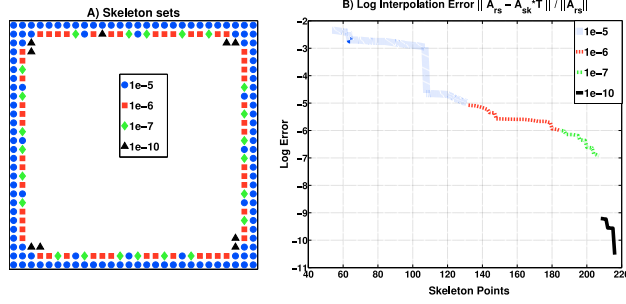


FIG. 3.2. A) Skeleton sets picked by the interpolative decomposition for different relative interpolation accuracies B) Log plot of relative interpolation error depending on skeleton set size

are obtained by merging skeleton sets of children and performing another interpolative decomposition on the corresponding block of A^ℓ to obtain the skeleton.

This approach works for one-dimensional problems but for two-dimensional ones using interpolative decomposition at all levels of the hierarchy is prohibitive even in 2D. A densification acceleration: the complexity of each decomposition for a box B_i is proportional to $k_i^3 \sim n_\ell^{3/2}$ in the d -dimensional case. As a consequence, the overall complexity cannot be lower than $O(N^{3/2})$.

Our algorithm for constructing the skeleton set at all levels is based on the following crucial observation: *It is always possible to find an accurate set of skeleton points for a box by searching exclusively within a thin layer of points along the boundary of the box.* This observation can be justified using separation estimates from potential theory, cf. Section 2.2. It has also been validated by extensive numerical experiments. Increasing accuracy adds more points in deeper layers but the depth never goes too large: for the kernels we have considered, the factorization selects one boundary layer for $\varepsilon \sim 10^{-5}$, and two layers for $\varepsilon \sim 10^{-10}$. (Figure 3.2).

This observation allows to make two modifications to the skeleton selection algorithm. First, pick the set of points from which the skeletons are selected *a priori* to m boundary layers. Second, rather than selecting skeleton points for a parent box from the union of skeletons of child boxes, an efficient interpolative decomposition, we simply take all points in the boundary layers of the parent box.

More specifically $I_i = I_{c_1(i)}^{sk} \sqcup I_{c_2(i)}^{sk}$ is a union of I_i^{sk} , the skeleton of B_i , consisting of all points of I_i within m layers of the boundary of B_i , and $I_i^{rs} = I_i \setminus I_i^{sk}$ (interior index), consisting of points at the interface of two child boxes. To obtain the interpolation operator $T_i : I_i^{rs} \rightarrow I_i^{sk}$ provided by the interpolative decomposition in the two-branch, we use a proxy $Z^{proxy} = \{z_j\}_{j=1}^{p_i}$ (as described in Section 2.2), and compute T_i from the following equation:

$$K(Z^{proxy}, X_i^{sk})T = K(Z^{proxy}, X_i^{rs}).$$

REMARK 3.1. . The set from which the skeleton set is picked has fixed width. This means that matrices acting on the skeleton set are compressible (in the HSS sense) in a manner analogous to boundary integral operators and admit linear complexity matrix algebra. These matrices in essence act like boundary-to-boundary operators on the box.

3.2. Overview of the modified inversion algorithm. Construction of a compressed inverse of an HSS matrix described in Section 2.4 proceeds in two stages: first the HSS form of the matrix is constructed, followed by HSS inversion, which is performed in two additional components. In the unmodified algorithm, significant changes are made to the second stage, including compression of all blocks. Not all blocks are compressed at the first stage (compression of A) as needed for the inverse construction, so we reduce the first stage algorithm to building the tree and interpolation operators L_i, R_i only.

Examining Algorithm 1, we observe that all blocks formed for each box B_i involve the factors E_i, F_i^{-1} and the interpolation matrices L_i, R_i . One first step in compression is to compute $\{E_i, F_i^{-1}\}$ only. Matrix-vector multiplication for blocks \tilde{D}, \tilde{L} and \tilde{R} needed for the inverse matrix algorithm can be implemented as a sequence of matrix-vector products for L_i, R_i, E_i and F_i^{-1} .

The main operations in the algorithm are the U -dense block inversion in lines 10–11. Since F_i is of size $m_i = k_{c_1(i)} + k_{c_2(i)}$ (merge of U -kelebon U) and E_i is of size k_i , the space of these blocks is $O(k_i^2) = O(n_\ell)$ and inverting them costs $O(k_i^3) = O(n_\ell^{3/2})$ floating point operations. This observation shows that it is impossible to obtain linear complexity for HSS inversion if we use and invert E_i and F_i blocks densely to form the replacement matrices that form L_i and R_i .

We first present a reformulation of the HSS inversion algorithm that partitions completion of E_i and F_i^{-1} into block operations allowing us to compute blocks as low-rank or one-dimensional HSS forms. As is typically the case, it is essential to avoid explicit construction of the blocks that we want to compute, i.e. they are computed in a compact form from the start.

Building and inverting F_i . For a non-leaf box B_i , F_i is a linear operator defined on $I_i = I_{c_1(i)}^{sk} \sqcup I_{c_2(i)}^{sk}$, that is on the merge of kelebon pins from its children. Using physical interpretation of Section 2.4, it maps charge distributions to fields on B_i , adding contributions from local operators $E_{c_1(i)}, E_{c_2(i)}$ and sibling interactions. We expect F_i to have a rank that is similar to that of $K[I_i, I_i]$.

Aside from needing a compact form and an efficient matrix for F_i^{-1} to be used in the inverse HSS matrix algorithm, we also want F_i^{-1} to consist of $E_i^{-1} = R_i F_i^{-1} L_i$. Let Π_i be the permutation matrix that places kelebon pins in its Matrices $R_i = [I \ T_i^{up}] \Pi_i^T$ and $L_i = \Pi_i \begin{bmatrix} I \\ (T_i^{dn})^T \end{bmatrix}$ have a block form, in b -blocks T_i^{up} and T_i^{dn} , which will have low-rank (Section 3.3.2). To construct E_i efficiently, we need an explicit partition of F_i into blocks matching blocks of R_i and L_i , i.e. corresponding to kelebon indices I_i^{sk} and I_i^{rs} and I_i^{rs} of B_i . We use the following notation for the blocks of F_i :

$$\Pi_i F_i \Pi_i^T = \begin{bmatrix} F_i[I_i^{sk}, I_i^{sk}] & F_i[I_i^{sk}, I_i^{rs}] \\ F_i[I_i^{rs}, I_i^{sk}] & F_i[I_i^{rs}, I_i^{rs}] \end{bmatrix} = \begin{bmatrix} F_i^{sk} & F_i^{s \leftarrow r} \\ F_i^{r \leftarrow s} & F_i^{rs} \end{bmatrix}$$

here we use s to refer to the kelebon set of the parent and r to the “child” set (the proof of the notion of the kelebon set of the children is contained in the appendix).

To represent F_i^{-1} we use the blocks and perform block-Gaussian elimination. If $\Phi = \Pi_i F_i^{-1} \Pi_i^T$. Then:

$$\Phi_i = \begin{bmatrix} \phi_i^{sk} & \phi_i^{s \leftarrow r} \\ \phi_i^{r \leftarrow s} & \phi_i^{rs} \end{bmatrix}$$

We note that by eliminating child pins ϕ_i^{sk} is the inverse of the Schur complement matrix $(S_i^{rs})^{-1} = [F_i^{sk} - F_i^{s \leftarrow r} (F_i^{rs})^{-1} F_i^{r \leftarrow s}]^{-1}$, and that only $\{ (F_i^{rs})^{-1}, F_i^{r \leftarrow s}, F_i^{s \leftarrow r}, (S_i^{rs})^{-1} \}$ are needed to compute the blocks for the inverse Φ_i . The core in our new compact algorithm that builds F_i^{-1} (Section 3.3.3) consists of blocks in compact form.

Building E_i . Following the definition of E_i and the block structure of F_i^{-1} we obtain the expression:

$$E_i^{-1} = [I \ T_i^{up}] \begin{bmatrix} \phi_i^{sk} & \phi_i^{s \leftarrow r} \\ \phi_i^{r \leftarrow s} & \phi_i^{rs} \end{bmatrix} \begin{bmatrix} I \\ (T_i^{dn})^T \end{bmatrix} = \phi_i^{sk} + T_i^{up} \phi_i^{s \leftarrow r} + \phi_i^{r \leftarrow s} (T_i^{dn})^T + T_i^{up} \phi_i^{rs} (T_i^{dn})^T$$

As T_i^{up}, T_i^{dn} are low-rank, the last three terms in this sum are low-rank, too. Hence, E_i can be computed as a low-rank piece of ϕ_i^{sk} , the inverse of S_i^{rs} .

We summarize the modified algorithm below. This is just a physical algebraic transformation, we explain in greater detail how the new algorithm can be used to compute sparse matrices. Next to each block we give the new backends of computation and in the complexity analysis $O(N)$ algorithm: [LR] (low-rank) or [HSS1D]. The operation of the new form consists of:

1. The blocks of F_i^{-1} : $\{ (F_i^{rs})^{-1}, F_i^{r \leftarrow s}, F_i^{s \leftarrow r}, (S_i^{rs})^{-1} \}$
2. The matrix E_i

The entries of blocks of F_i are evaluated using the formulas in Algorithm 1.

In the remaining part of this section we elaborate the details of efficient construction of all blocks using HSS and low-rank operations.

Algorithm 2 Modified HSS inversion algorithm

```
1: for each box  $B_i$  in fine-to-coarse order do
2:   if  $B_i$  is a leaf then
3:      $F_i^{-1} = D_i^{-1} = K[I_i, I_i]^{-1}$ 
4:   else
5:     Comp blocks  $F_i^{sk}, F_i^{rs}$  from  $(E_{c_1(i)}, E_{c_2(i)})$ 
      In computing and inverting the matrix  $F_i$  for non-leaf boxes we use block inversion formulae as explained above:
6:      $(F_i^{rs})^{-1} = (F[I_i^{rs}, I_i^{rs}])^{-1}$  [HSS1D]
7:     Comp blocks  $F_i^{r\leftarrow s}, F_i^{s\leftarrow r}$  [LR]
8:      $(S_i^{rs})^{-1} = (F_i^{sk} - F_i^{s\leftarrow r}(F_i^{rs})^{-1}F_i^{r\leftarrow s})^{-1}$  [HSS1D]
9:   end if
10:   $E_i = ((S_i^{rs})^{-1} + T_i^{up}\phi_i^{s\leftarrow r} + \phi_i^{r\leftarrow s}(T_i^{dn})^T + T_i^{up}\phi_i^{rs}(T_i^{dn})^T)^{-1}$  [HSS1D]
11: end for
```

3.3. Compressed two-dimensional HSS inversion. We now compress formulas for linear operations in the algorithm: low-rank and one-dimensional HSS (dense-block HSS) described in Section 2.

Operator Notation. To distinguish between linear operations compressed in different ways we use different fonts

- X (in normal font) refers to an abstract linear operator; its representation is not specified.
- \mathcal{X} refers to a dense-block HSS representation of X ;
- \mathcal{X} refers to a low-rank representation of X .

The font used for the representation operators like $R(:, J) = [I \ T]$ refers to the representation of T . Operations like matrix multiplication should be understood accordingly e.g., Xv is evaluated using a low-rank factorization of X ; if the rank is q , and the vector is n , the complexity is $O(qn)$. Similarly $\mathcal{X}v$ is an $O(n)$ application of a dense-block HSS matrix.

We analyze the algorithms operating on sparse matrices as fast if all operations involved have cost and storage proportional to the block size $O(n^{1/2})$ or at times a logarithmic factor $O(n^{1/2} \log^q(n))$. Our algorithm includes the main fast routines

1. **INTER_LOW_RANK:** Inversion matrices T_i are built in low-rank form.
2. **BUILD_Finv:** Given dense-block HSS matrices $\{\mathcal{E}_{c_1(i)}, \mathcal{E}_{c_2(i)}\}$, it computes $\{(F_i^{rs})^{-1}, F_i^{s\leftarrow r}, F_i^{r\leftarrow s}, (S_i^{rs})^{-1}\}$ in hierarchical compressed forms
3. **BUILD_E:** Given $\{(F_i^{rs})^{-1}, F_i^{s\leftarrow r}, F_i^{r\leftarrow s}, (S_i^{rs})^{-1}\}$ and T_i^{up}, T_i^{dn} , it computes E_i as a dense-block HSS matrix

3.3.1. Fast Arithmetic. There is a number of operations in low-rank and HSS matrices which we must be able to perform efficiently

- **Dense-block HSS1D compression, inversion and matvec:** These are the algorithms in [18], which were outlined in Section 2, and we denote the corresponding routines as **HSS1D_Compress** and **HSS1D_Invert**.
- **Fast addition and manipulation of HSS1D matrices:**
 - **HSS1D_Sum:** Given two matrices \mathcal{A} and \mathcal{B} in HSS form, return an HSS form for $\mathcal{C} = \mathcal{A} + \mathcal{B}$
 - **HSS1D_Split:** Given a matrix \mathcal{A} defined on $I_1 \cup I_2$, it produces the diagonal blocks \mathcal{A}_1 and \mathcal{A}_2 in HSS form.
 - **HSS1D_Merge:** Given matrices \mathcal{A}_1 and \mathcal{A}_2 , it concatenates them to produce the block diagonal HSS matrix \mathcal{A} , and stores it accordingly
 - **Additional HSS compression routines:**
 - * **LR_to_HSS1D:** Convert a low-rank operator to dense-block HSS form.
 - * **HSS1D_Recompress:** Using the algorithm by Xia [39], we compress an HSS 1D form to obtain optimal ranks. It is crucial to do so after performing fast arithmetic (e.g. a sum) of HSS matrices

REMARK 3.2. **Matrix-matrix products.** As we have mentioned before, a feature of our algorithm is that it avoids using the linear yet expensive matrix-matrix product algorithm for structured matrices.

We arrange the computations so that all matrix-matrix products are between a dense-block HSS \mathcal{H} of size $k \times k$ and a low rank matrix $K = UV^T$ of rank q . Since $\mathcal{H}U$ requires only q $O(k)$ fast matvecs, these products may be computed in $O(qk)$ work.

REMARK 3.3. For every call to **HSS1D-Sum** in the algorithms below, it should be assumed that it is followed by a recompression step (a call to **HSS1D-Recompress**). Due to the theorem 5.3 of [39], this implies all dense-block HSS matrices presented are compact, that is, their blocks have a near-optimal size.

Randomized interpolative decomposition RAND-ID. The randomized sampling technique in [24, 29, 38] speeding up the interpolative decomposition are critical to obtain the right complexity for the compression of low-rank operators. For an $m \times m$ dense-block HSS matrix A of rank q , the matrix has complexity $O(m)$ time. Sampling that level ℓ of the binary tree \mathcal{T} , $m = O(n_\ell^{1/2})$ and $q = O(\log(n_\ell))$, the complexity of the randomized IDs is

$$O(mq^2 + q^3) = O(n_\ell^{1/2} \log^2(n_\ell))$$

3.3.2. Interpolation Operators in low-rank form. We recall that interpolation operators T_i are built in the binary tree \mathcal{T} using an interpolative decomposition, and that they are inserted in the dense-block HSS in our algorithms. They encode interactions between the residual points indexed as I_i^{rs} and the exterior X_i^{ext} as a linear combination of interactions in the skeleton points indexed by I_i^{sk} . In other words, they are solutions of the linear equation:

$$K[X_i^{ext}, X_i^{sk}]T_i = K[X_i^{ext}, X_i^{rs}]$$

The acceleration proposed in Section 2.2 implies that interaction in the points may be represented in a proxy Z_i^{proxy} of chargesight to the box boundary. We employ as many layers as are kept for I_i^{sk} , and remove points $|Z_i^{proxy}| = |X_i^{sk}| = k_i$. This yields an equation for T_i :

$$K[Z_i^{proxy}, X_i^{sk}]T_i = K[Z_i^{proxy}, X_i^{rs}]$$

$K[Z_i^{proxy}, X_i^{sk}]$ encodes interactions between ϵ close boundaries across the interface being equal to the grid spacing h . It is, in fact, ill-conditioned, and most importantly, it has dense-block HSS structure.

$K[Z_i^{proxy}, X_i^{rs}]$ encodes interactions in the interface points. Except for the closest layer, the interface is well-separated from the proxy and so it is easy to see (in a multiple argument manner) that this matrix is of logarithmic rank:

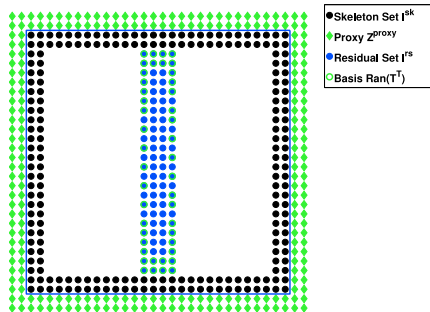


FIG. 3.3. Skeleton points are in black, residual points in blue and proxy points (diamonds) in green. We apply an ID with $\epsilon = 1e - 10$, and label subselected interface points also in green.

As a result, $T_i = K[Z_i^{proxy}, X_i^{sk}]^{-1}K[Z_i^{proxy}, X_i^{rs}]$ is also a low-rank operator. We describe a fast algorithm to compute this operator. We note that while the description above follows the case of boxes

in the plane, these observations hold in general, along as most of the pins are all spaced from the px

Algorithm 3 Algorithm INTER LOWRANK

Input: Box B_i information, indices I_i^{sk} and I_i^{rs} ; **Output:** $\{U_i, V_i\}$ such that $T_i = U_i V_i^T$.

- 1: (i) Compress and Invert HSS 1D operator
- 2: $\mathcal{K}^{s \rightarrow p} = \text{HSS1D_Compress}(K, Z_i^{\text{proxy}}, X_i^{sk}, \varepsilon)$
- 3: $(\mathcal{K}^{s \rightarrow p})^{-1} = \text{HSS1D_Invert}(\mathcal{K}^{s \rightarrow p}, \varepsilon)$
- 4: (ii) Randomized interpolatory decomposition
- 5: $\mathcal{K}^{r \rightarrow p} = \text{HSS1D_Compress}(K, Z_i^{\text{proxy}}, X_i^{rs}, \varepsilon)$
- 6: $[T^{r \rightarrow p}, J^{r \rightarrow p}] = \text{RAND_ID}(\mathcal{K}^{r \rightarrow p}, \varepsilon)$;
- 7: The ID gives a low-rank decomposition of $K^{r \rightarrow p}$ (of ε -rank q);
- 8: $U_i = (\mathcal{K}^{s \rightarrow p})^{-1} K^{r \rightarrow p}(:, J^{r \rightarrow p}(1 : q))$;
- 9: $V_i^T(:, J^{r \rightarrow p}) = [I \quad T^{r \rightarrow p}]$

REMARK 3.4. In general, Z_i^{proxy} will have slightly more points than X_i^{sk} , making $\mathcal{K}^{s \rightarrow p}$ a rectangular matrix. A fast HSS least squares algorithm such as in [13, 25] replaces the inversion and inverse apply in lines 3 and 8 of Algorithm 3, with no impact in complexity.

3.3.3. Compressed F^{-1} . We define some auxiliary indices for the merge of two children boxes $I_{i,c(i)}^{sk}$ as keleon pins shared in child $c(i)$ (on boundary) and $I_{i,c(i)}^{rs}$ as ridal children's keleon pins (in the middle interface). We define $I_i^{sk} = I_{i,c_1(i)}^{sk} \cup I_{i,c_2(i)}^{sk}$, and $I_i^{rs} = I_{i,c_1(i)}^{rs} \cup I_{i,c_2(i)}^{rs}$

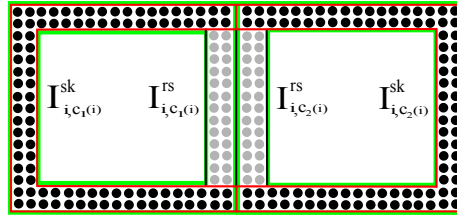


FIG. 3.4. Merge of two children's skeleton indices.

We recall that this matrix F is built and spaced into blocks corresponding to keleon I_i^{sk} and ridal I_i^{rs} sets. These can be seen as arranged along one-dimensional cuts (boundary of the pentbox and interface between siblings). We order I_i^{rs} in the direction of the interface, and I_i^{sk} cyclically around the box boundary.

Then $\mathcal{F}_i^{rs} = F(I_i^{rs}, I_i^{rs})$ and $\mathcal{F}_i^{sk} = F(I_i^{sk}, I_i^{sk})$ can be computed in dense-block HSS form.

The off-diagonal blocks $\{F_i^{s \leftarrow r}, F_i^{r \leftarrow s}\}$ encode interaction between sets that are close only at few pins and can thus be compressed as low-rank operators.

BUILD_Finv routine. To clarify the structure of the construction of F^{-1} , in Algorithm 4 we show the routine in two ways. On the left we indicate the original dense computation, and on the right line of the reformulated algorithm in Section 2.5 it occurs. On the right we indicate the fast operations that replace the linear complexity algorithm.

Applying F^{-1} to a vector. Once we have the four compressed blocks $\{(\mathcal{F}_i^{rs})^{-1}, F_i^{s \leftarrow r}, F_i^{r \leftarrow s}, (\mathcal{F}_i^{sk})^{-1}\}$, the routine APPLY_Finv can be used to compute the fast product $\sigma = F_i^{-1}u$. This is done in a straightforward way using u into $[u[I_i^{sk}], u[I_i^{rs}]]$ and using fast matrix-vector products of the blocks of F^{-1} .

3.3.4. Compressed \mathcal{E} . The last piece that is required is a routine that constructs \mathcal{E} as a dense-block HSS matrix from F^{-1} in compressed form. For a box B_i , E_i is a linear operator defined on the keleon set I_i^{sk} . It is now obtained by applying F_i^{-1} to the HSS interpolator operators

$$E_i^{-1} = L_i F_i^{-1} R_i$$

Algorithm 4 BUILD_Finv

Input. Children's matrices $\mathcal{E}_{c_1(i)}$ and $\mathcal{E}_{c_2(i)}$ in HSS fom (each defined on keletons $I_{c_j(i)}^{sk}$).

Output.

1. $(\mathcal{F}_i^{rs})^{-1}$ in dens-block HSS fom on he eidal st I_i^{rs} .
2. $F_i^{s\leftarrow r}, F_i^{r\leftarrow s}$ as low-rank ops
3. Inverse of Schur complement matrix $(S_i^{rs})^{-1}$ in dens-block HSS fom on I_i^{sk} .

(i) Obtain diagonal blocks of E by splitting into boundary and interface	
$E_{c_j(i)}^{sk} = E_{c_j(i)}[I_{i,c_j(i)}^{sk}, I_{i,c_j(i)}^{sk}]$ $E_{c_j(i)}^{rs} = E_{c_j(i)}[I_{i,c_j(i)}^{rs}, I_{i,c_j(i)}^{rs}]$	$[\mathcal{E}_{c_j(i)}^{sk}, \mathcal{E}_{c_j(i)}^{rs}] = \text{HSS1D_Split}(\mathcal{E}_{c_j(i)}, I_{i,c_j(i)}^{sk})$
(ii) Bild diagonal blocks of F: merge diagonal blocks of E, compress off-diagonal blocks	
Line 5: $F^{sk} = \begin{bmatrix} E_{c_1}^{sk} & 0 \\ 0 & E_{c_2}^{sk} \end{bmatrix} + \begin{bmatrix} 0 & K[I_{c_1}^{sk}, I_{c_2}^{sk}] \\ K[I_{c_2}^{sk}, I_{c_1}^{sk}] & 0 \end{bmatrix}$	$\mathcal{E}_i^{sk,dg} = \text{HSS1D_Merge}(\mathcal{E}_{c_1(i)}^{sk}, \mathcal{E}_{c_2(i)}^{sk});$ $\mathcal{K}_i^{sk,off} = \text{HSS1D_Compress}(K, I_i^{sk});$ $\mathcal{F}_i^{sk} = \text{HSS1D_Sum}(\mathcal{E}_i^{sk,dg}, \mathcal{K}_i^{sk,off,d});$
Line 5: $F^{rs} = \begin{bmatrix} E_{c_1}^{rs} & 0 \\ 0 & E_{c_2}^{rs} \end{bmatrix} + \begin{bmatrix} 0 & K[I_{c_1}^{rs}, I_{c_2}^{rs}] \\ K[I_{c_2}^{rs}, I_{c_1}^{rs}] & 0 \end{bmatrix}$	$\mathcal{E}_i^{rs,dg} = \text{HSS1D_Merge}(\mathcal{E}_{c_1(i)}^{rs}, \mathcal{E}_{c_2(i)}^{rs});$ $\mathcal{K}_i^{rs,off} = \text{HSS1D_Compress}(K, I_i^{rs});$ $\mathcal{F}_i^{rs} = \text{HSS1D_Sum}(\mathcal{E}_i^{rs,dg}, \mathcal{K}_i^{rs,off,d});$
(iii) Inverse of F_i^{rs} :	
Line 6: $(F_i^{rs})^{-1} = F[I_i^{rs}, I_i^{rs}]^{-1}$	$(\mathcal{F}_i^{rs})^{-1} = \text{HSS1D_Invert}(\mathcal{F}_i^{rs});$
(iv) Low-rank decompositions for $F_i^{s\leftarrow r}$ and $F_i^{r\leftarrow s}$ using Randomized IDs	
Line 7: $F_i^{s\leftarrow r} = F[I_i^{sk}, I_i^{rs}]$ $F_i^{r\leftarrow s} = F[I_i^{rs}, I_i^{sk}]$	$[T_i^r, J_i^r] = \text{RAND_ID}(F_i^{s\leftarrow r}, \varepsilon);$ $[T_i^s, J_i^s] = \text{RAND_ID}(F_i^{r\leftarrow s}, \varepsilon);$
(v) Schur complement as a low-rank perturbation of \mathcal{F}_i^{sk} :	
Line 8: $(S_i^{rs})^{-1} = [F[I_i^{sk}, I_i^{sk}] - F^{s\leftarrow r}(F_i^{rs})^{-1}F^{r\leftarrow s}]^{-1}$	$\mathcal{P}_i^{sk} = \text{LR_to_HSS1D}(-F_i^{s\leftarrow r}(\mathcal{F}_i^{rs})^{-1}F_i^{r\leftarrow s});$ $S_i^{rs} = \text{HSS1D_Sum}(\mathcal{F}_i^{sk}, \mathcal{P}_i^{sk});$ $(S_i^{rs})^{-1} = \text{HSS1D_Invert}(S_i^{rs});$

Hence, he pical intion again is that E_i has a rank k_i similar to that of $K[I_i^{sk}, I_i^{sk}]$. In [8, 12, 32] this is called a *reduced scattering matrix*. From a polyalgebraic point of view we observe that \mathcal{E} is a low-rank perturbation of S^{rs} . Hence, if S^{rs} has dens-block HSS, so does \mathcal{E} .

BUILD_E routine. Recalling the definition of E_i in Algorithm 2,

$$E_i^{-1} = R_i F_i^{-1} L_i = [I \ T_i^{up}] \begin{bmatrix} \phi_i^{sk} & \phi_i^{s\leftarrow r} \\ \phi_i^{r\leftarrow s} & \phi_i^{rs} \end{bmatrix} \begin{bmatrix} I \\ (\mathbf{T}_i^{dn})^T \end{bmatrix} = \phi_i^{sk} + \mathbf{T}_i^{up} \phi_i^{s\leftarrow r} + \phi_i^{r\leftarrow s} (\mathbf{T}_i^{dn})^T + \mathbf{T}_i^{up} \phi_i^{rs} (\mathbf{T}_i^{dn})^T$$

we observe the last three matrices are low-rank. Using the Schur complement formulae we obtain explicit factorizations for each one, recompose a single low-rank matrix and then convert to the dens-block HSS form using **LR_to_HSS1D**.

We can perform the low-rank matrix products (of rank q_i) in Algorithm 5 in $O(m_i q_i)$ or $O(m_i q_i^2)$ operations since all of the products involved are fast ($O(k_i)$ or $O(m_i - k_i)$, here $|I_i| = m_i$ and $|I_i^{sk}| = k_i$).

3.4. Inverse matrix-vector multiplication. Once the compressed form of the inverse is obtained, it can be efficiently applied to a righthand side vector using the algorithm of Section 2.3, but using fast algorithms for compressed representations of blocks. We present these for completeness (Algorithm 6).

We notice that a product of \tilde{R}_i, \tilde{D}_i and \tilde{L}_i is obtained by fast matrix-vector multiplication of L_i, R_i, \mathcal{E}_i and the blocks comprising F_i^{-1} . All of these have complexity $O(n_\ell^{1/2})$ or $O(n_\ell^{1/2} \log(n_\ell))$.

Algorithm 5 BUILD_E

Input. F^{-1} asblocks (\mathcal{F}_i^{rs})⁻¹, $\mathbf{F}_i^{s\leftarrow r}$, $\mathbf{F}_i^{r\leftarrow s}$, $(\mathcal{S}_i^{rs})^{-1}$; L_i and R_i in low rank fom: \mathbf{T}_i^{up} , J_i^{up} , \mathbf{T}_i^{dn} , and J_i^{dn} , $\mathbf{T}_i^{up} = U_i^{up}V_i^{up}$, $(\mathbf{T}_i^{dn})^T = U_i^{dn}V_i^{dn}$. **Output.** \mathcal{E}_i in HSS fom (on the cu I_i^{sk})

- 1: $V_{i,1}^E = V_i^{up}\phi^{s\leftarrow r}$; $U_{i,1}^E = U_i^{up}$
- 2: $U_{i,2}^E = \phi^{r\leftarrow s}U_i^{dn}$; $V_{i,2}^E = V_i^{dn}$
- 3: $U_{i,3}^E = U_i^{up}V_i^{up}\phi^{rs}U_i^{dn}$; $V_{i,3}^E = V_i^{dn}$
- 4: **Recomps** $U_i^E(V_i^E)^T = \sum_{p=1}^3 U_{i,p}^E(V_{i,p}^E)^T$
- 5: $\mathcal{M}_i = \mathbf{LR_to_HSS1D}(U_i^E, V_i^E)$
- 6: $\mathcal{E}_i^{-1} = \mathbf{HSS1D_Sum}((\mathcal{S}_i^{rs})^{-1}, \mathcal{M}_i)$
- 7: $\mathcal{E}_i = \mathbf{HSS1D_Invert}(\mathcal{E}_i^{-1})$

Algorithm 6 HSS inverse multiplication.

Input

1. The binary tree \mathcal{T} including leaf nodes and R_i, L_i associated in boxes
2. HSS-comps in boxes forming F_i and \mathcal{E}_i .
3. the vector f of field elements defined at the pins

Output. The output $\sigma = A^{-1}f$, here A^{-1} is the compd inv.

{**Upward Pass comp** u_i^{up} }

for each box B_i in fine-to-coarse order **do**

- 1: **if** B_i is a leaf **then**
- 3: $u = f(I_i^{up})$
- 4: **else**
- 5: $u = \begin{bmatrix} u_{c_1(i)}^{up} \\ u_{c_2(i)}^{up} \end{bmatrix}$
- 6: **end if**
- 7: **Multiplication by** $\tilde{R}_i = E_i R_i F_i^{-1}$:
- 8: $\varphi_i = \mathbf{APPLY_Finv}(u)$
- 9: $u_i^{up} = \mathcal{E}_i R_i \varphi_i$
- 10: **end for**
- 11: {**Downward Pass comp** ϕ_i^{dn} ; here $A^{-1}f$ on leaf B_i is ϕ_i^{dn} }
- 12: $\phi_{top}^{dn} = 0$, $\nu_{top}^{dn} = u$
- 13: **for** each B_i in coarse-to-fine order **do**
- 14: **Multiplication by** $\tilde{D}_i = F_i^{-1}[I - L_i \tilde{R}_i]$
- 15: **Define** u as above
- 16: $\nu_i^{dn} = u - L_i u_i^{up}$
- 17: **Multiplication by** $\tilde{L}_i = F_i^{-1} L_i E_i$: $\rho_i^{dn} = L_i \mathcal{E}_i \phi_i^{dn}$
- 18: **Add both contributions** by the common factor F_i^{-1} :
- 19: **if** B_i is a leaf **then**
- 20: $\sigma(I_i^{dn}) = \mathbf{APPLY_Finv}(\nu_i^{dn} + \rho_i^{dn})$
- 21: **else**
- 22: $\begin{bmatrix} \phi_{c_1(i)}^{dn} \\ \phi_{c_2(i)}^{dn} \end{bmatrix} = \mathbf{APPLY_Finv}(\nu_i^{dn} + \rho_i^{dn})$
- 23: **end if**
- 24: **end for**

4. Complexity Estimates. In this section we estimate the computational complexity of the dense-block and the compressed-block algorithms presented in Section 3, under a number of assumptions on the rank m of blocks of A .

We first define a framework of what we mean by algorithms defined on a binary tree \mathcal{T} , and use it to analyze all the algorithms described in this paper. We consider both non- m -rank-invariant (NTI), and m -rank-invariant kernels (TI) for which significant performance gains can be

obtained.

4.1. Complexity of algorithms on binary tries. All algorithms can be matrix blocks associated in a binary tree \mathcal{T} . To produce complexity estimates for a given accuracy ε , we divide boxes for $W_\ell(n_\ell, \varepsilon)$ and $M_\ell(n_\ell, \varepsilon)$ at each level ℓ of the tree, here $n_\ell = 2^{-\ell}N$ is the maximum number of pins in a box at this level (where the box or box on a given level has small variance).

LEMMA 4.1. *Let $n_\ell = 2^{-\ell}N$, and $d = \log_2(N/n_{max})$, and exponents $p, q \geq 0$. Then if $W_\ell(n_\ell, \varepsilon)$ has the form $C_\varepsilon n_\ell^p \log_2^q(n_\ell)$, the total work has complexity*

$$NTI: \sum_{\ell=0}^d 2^\ell W_\ell(n_\ell, \varepsilon) = \begin{cases} O(N) & : 0 \leq p < 1 \\ O(N \log_2^{q+1} N) & : p = 1 \\ O(N^p \log_2^q N) & : p > 1 \end{cases}$$

$$TI: \sum_{\ell=0}^d W_\ell(n_\ell, \varepsilon) = O(N^p \log^q N)$$

For NTI algorithms on each level ℓ we obtain an estimate by adding the bond for box for the $2^{-\ell}$ boxes. The polynomial growth in W_ℓ or M_ℓ is compensated by the fact that the number of boxes decreases exponentially going up the tree.

If the rate of growth of W_ℓ is slower than linear ($p < 1$), the overall complexity is linear. If W_ℓ grows linearly we accumulate a $\log N$ factor going up the tree. If the growth of W_ℓ is linear ($p > 1$), the work performed on the top boxes dominates and we obtain the same complexity as W_ℓ for the overall algorithm.

In the TI case, work done on the top boxes dominates the calculation, since only one set of matrices needs to be computed and used per level. Hence, the interpolation is simpler the single-box bond for complexity, the top level reflects the overall complexity.

4.2. Assumptions on matrix structure. Complexity estimates for the work performed on a pin on a matrix. Let us state some assumptions already made in Sections 2 and 3:

- 1. Skeleton size scaling:** The maximum size of skeleton sets for boxes at level ℓ grows as $O(n_\ell^{1/2})$. This determines the size of blocks in the HSS set, and can be proved for non-oscillatory PDE kernels in 2D.
- 2. Localization:** Element densities may be assumed to represent long range interactions in any confined accuracy. Section 2.2.
- 3. Skeleton structure:** The skeleton set for any box may be chosen from within a thin layer of pins close to the boundary of the box. Section 3.1.
- 4. Compressed block structure:** Empirical evidence and physical intuition from solving problems allows us to assume that the blocks of F and E discussed in Section 3 have one-dimensional HSS or low-rank structure, in logarithmic rank growth ($O(\log(n_\ell))$).

These assumptions are naturally in the context of solving integral equations in non-oscillatory PDE kernels in 2D. All assumptions regarding the lattice are relevant for both dense and compressed block algorithms. The lattice is needed only for the compressed-block algorithms.

We note assumption 3 implies: being able to pick skeletons from a thin boundary layer determines how the size scale. We mention them separately to distinguish them from the design and complexity analysis of algorithms while 1 mainly impacts block size in the other HSS set, 3 is much more specific and refers to a prior knowledge of skeleton structure which we put into the compressed-block algorithm.

4.3. Estimates. We analyze work and range for the algorithms of Section 3. Since they all share the same set of fast operations (see Section 3.3.1), we can make unifying observations

Work.

- Assumptions 1 and 3 imply that fast blockwise operations on HSS blocks of size $O(n_\ell^{1/2})$. The Assumption 4 states that they behave like HSS matrices opening boundary integral operators for which all one-dimensional HSS operations are known to be linear in size. Thus all **HSS1D** operations are $O(n_\ell^{1/2})$, including matrix-vector multiplication.
- As indicated in Remark 3.2, for an HSS matrix of size $k \times k$, product of HSS and low-rank matrices is $O(kq)$. Assumption 4 implies all such products are $O(n_\ell^{1/2} \log_2(n_\ell))$. Low-rank matrix-vector multiplication has the same complexity.
- Finally both **LR_to_HSS1D** and **Rand_ID** involve interpolative decompositions of a matrix of size $O(n_\ell^{1/2}) \times O(\log_2(n_\ell))$, and, hence have complexity $O(n_\ell^{1/2} \log_2^2(n_\ell))$. Products between low-rank matrices are also of this complexity.

Storage.

- Again, since all HSS blocks behave as operators acting on one-dimensional box boundaries, storage is linear in respect to the number of nodes along the boundary of the box $O(n_\ell^{1/2})$.
- A low-rank matrix of size $m \times n$ and rank q occupies $O((m+n)q)$ space in storage. By Assumption 4, storage of low-rank blocks (L, R and off-diagonal blocks of F) is $O(n_\ell^{1/2} \log_2(n_\ell))$.

Algorithms 3, 4, and 5, use only the operations listed above. We observe that all algorithms contain at least one $O(n_\ell^{1/2} \log_2^2(n_\ell))$ operation. In terms of storage, **INTER_LOW_RANK** and **BUILD_Finv** use both HSS and low-rank blocks ($O(n_\ell^{1/2} \log_2(n_\ell))$), and **BUILD_E** uses one HSS block ($O(n_\ell^{1/2})$). Hence, the compressed-block interpolation operator build and inverse compression algorithms perform $W_\ell^{CB} = O(n_\ell^{1/2} \log_2^2(n_\ell))$ work per box and use $M_\ell^{CB} = O(n_\ell^{1/2} \log_2(n_\ell))$ storage for each of the matrices computed at level ℓ . As a consequence their dense-block counterparts use $W_\ell^{DB} = O(n_\ell^{3/2})$ and $M_\ell^{DB} = O(n_\ell)$.

We note that a more detailed complexity analysis may be performed to obtain constants for each problem, given the necessary experimental data about the kernel for a given accuracy ϵ . The specific dependence of these constants on accuracy is briefly discussed and listed in Section 5.2.

We summarize the complexity results in the following proposition

PROPOSITION 4.2.

Dense-Block Algorithms Let A be an $N \times N$ system matrix such that assumptions 1-3 hold. Then, the dense-block tree build and inverse compression algorithms perform $O(N^{3/2})$ work. For NTI kernels, storage requirements and matrix-vector products are both $O(N \log N)$. For TI kernels, storage is $O(N)$, and matrix-vector products are $O(N \log N)$.

NTI Compressed-Block Algorithms Let A be an $N \times N$, non translation invariant system matrix such that assumptions 1-4 hold. Then compressed-block tree build, inverse compression and HSS apply all perform $O(N)$ work, and require $O(N)$ storage.

TI Compressed-Block Algorithms Let A be an $N \times N$, translation invariant system matrix such that assumptions 1-4 hold. Then compressed-block tree build, inverse compression and HSS apply all perform $O(N)$ work, and require $O(N)$ storage. In fact, inverse compression work and storage are sublinear: $O(N^{1/2} \log_2^2 N)$ and $O(N^{1/2} \log_2 N)$, respectively.

The limitations of dense-block algorithms now become clear. With notation as in Lemma 4.1, dense-block algorithms depend on $(p, q) = (3/2, 0)$ for work and $(p, q) = (1, 0)$ for storage, which produces all linear complexity. For the compressed block algorithms on the other hand, which use $(p, q) = (1/2, 2)$ for work and $(p, q) = (1/2, 1)$ for storage, which does yield linear complexity.

REMARK 4.3. As we have previously observed, the compressed-block algorithm may be generalized to system matrices with other rank growth and behavior. More specifically, we observe that a sufficient condition to maintain optimal complexity could be that the outer HSS structure ranks grow as $O(n_\ell^p)$, for $p < 1$, and that the most expensive operations such as the randomized interpolative decomposition also remain being sublinear. This would require all low-rank blocks to be of rank $q_\ell \sim n_\ell^{\min\{1/3, (p-1)/2p\}}$.

REMARK 4.4. In all our practical implementations of the compressed-block algorithms, we perform dense computations for blocks up to a fixed threshold skeleton size k_{cut} , after which we switch to the fast routines. This may then be tuned as a parameter to further speed up these algorithms, and a straight-forward computation shows it does not alter their computational complexity.

5. Numerical Results. In this section, we describe a series of numerical experiments that test an improved convolution algorithm on both translation-invariant (TI) and non-translation-invariant (NTI) kernels.

All results are obtained by running a Matlab implementation of our algorithm on a server with Intel Xeon X5650, 2.67 GHz processor in no parallelization.

General formulation. As noted in the introduction, a considerable number of physical problems involve solving one or a sum of integral equations of the form

$$\mathcal{A}[\sigma](x) = a(x)\sigma(x) + \int_{\Omega} b(x)\mathcal{K}(\|x-y\|)c(y)\sigma(y)dy = f(x),$$

where a , b and c are given smooth functions and $\mathcal{K}(r)$ is related to a free-space Green's function. Then, given $\{x_i\}_{i=1}^N \in \Omega = [-1, 1]^2$ partitioned into a regular grid with spacing h , we can perform a Nyquist discretization, obtaining a linear system $A\sigma = f$, with A an $N \times N$ matrix with entries

$$A_{i,j} = a(x_i)\delta_{i,j} + h^2 b(x_i)\mathcal{K}(\|x_i - x_j\|)c(x_j)$$

5.1. High accuracy: performance and scaling. Most of our work is done for a target accuracy of $\epsilon = 10^{-10}$ (referring to the local truncation error of all operations); this is the “best” for our algorithm, as high accuracy requires larger kernels. We measure the all-clock time and memory usage for the following parts of the solution process:

1. building the tree and interpolation operators
2. inverse convolution and composition;
3. inverse matrix-vector multiplication (slow, memory only).

We compare against the dense-block HSS inversion algorithm. We take $\mathcal{K}(r)$ to be the 2D Laplace free-space Green's function: $\mathcal{K}(r) = \frac{1}{2\pi} \log(r)$ and $a \equiv 1$.

- If $b \equiv c$, A is symmetric, which leads to some computational savings in the HSS algorithms
- The case $b \neq 1$ is an example of a **non translation invariant (NTI)** kernel. Such an equation arises in forward scattering problems (e.g. low-frequency Lipman-Schinger equation).
- The case $b \equiv 1$ is one of our examples of a **translation invariant (TI)** kernel. We recall that significant computational savings may be achieved in this case, given that we only build one set of matrix blocks per level.

We note that these results are typical for non-oscillatory Green's function kernels which are performed the same way for the 3D Laplace single layer potential and the 2D and 3D Yukawa Green's functions and found behavior analogous to the one described below.

5.1.1. Non-translation-invariant kernel. We take $b(x)$ to be a smooth function in moderate variation:

$$b(x) = 1 + 0.5e^{-(x_1-0.3)^2 - (x_2-0.6)^2}$$

Although this may seem like a simple choice, it does not turn out to be a particularly good performance unless the problem is understood or b is 0 on a large subdomain.

For the NTI case, ranks of blocks in dense-block HSS matrices and ranks of low-rank blocks vary both by level and inside each level. However, we observe that the rank growth for the blocks matches our complexity analysis.

We startleak at $n_{max} = 7^2$, and for problem sizes $N = 784$ to $N = 3211264$, we compare total inversion time (tree build + inverse composition) and total memory usage for the dense-block (HSS-D) and compressed-block (HSS-C) inverse composition algorithms.

We observe a very close match between the empirical scaling and the complexity estimates in Proposition 4.2, which we recall on the second row of Table 5.1.

The slope in a log-log plot (Figure 5.1.A) shows that both inversion and tree build times are $O(N^{3/2})$ for the dense block inversion and $O(N)$ for the accelerated algorithm. The break-even point is around $N = 10^5$, for which both methods take about 0.5 minutes. In the subplot in 5.1.B, we can clearly observe performance gains for $N > 10^5$ due to difference in scaling. By $N = 10^7$ our algorithm gains

N	HSS-D Time $O(N^{3/2})$	HSS-C Time $O(N)$	HSS-D Memory $O(N \log N)$	HSS-C Memory $O(N)$
784	0.11 s	0.17 s	4.68 MB	4.48 MB
3136	0.67 s	1.70 s	29.09 MB	25.24 MB
12544	4.50 s	8.32 s	159.59 MB	123.07 MB
50176	31.45 s	40.43 s	819.58 MB	538.51 MB
200704	3.79 m	3.23 m	3.72 GB	2.23 GB
802816	28.35 m	13.66 m	17.27 GB	9.23 GB
3211264	3.58 hr	54.795 m	70.99 GB	34.09 GB

TABLE 5.1

Total inversion time and storage for the NTI inverse compression algorithms

one order of magnitude per additional node. Additional gains may be obtained by adjusting the size of the fine-scale boxes.

The slope of the log-log plot (Fig. 5.1.C) confirms that memory for these algorithms behaves like $O(N \log N)$ and $O(N)$, respectively. Our algorithm uses less memory in all cases (Fig. 5.1.D). By $N = 10^7$, it uses $\sim 0.5 \times$ less memory (150 GB, which amounts to approximately 1800 double-precision degrees of freedom).

Matrix compression. Fig. 5.1.A, shows that the tree build time is about 18% of the total inversion time. The constant term can also be used to compare the original matrix \mathcal{A} in $O(N \log N)$ work, and even in $O(N)$ work by incurring the small additional cost of computing sibling interaction matrices as HSS 1D. Even though our method for building a fast direct solver is accelerated, the method to compute interaction matrices L and R readily yields $O(N)$ matrix compression. The compressed matrix can be used as a kernel-independent FMM, in a significantly simplified algorithm [54].

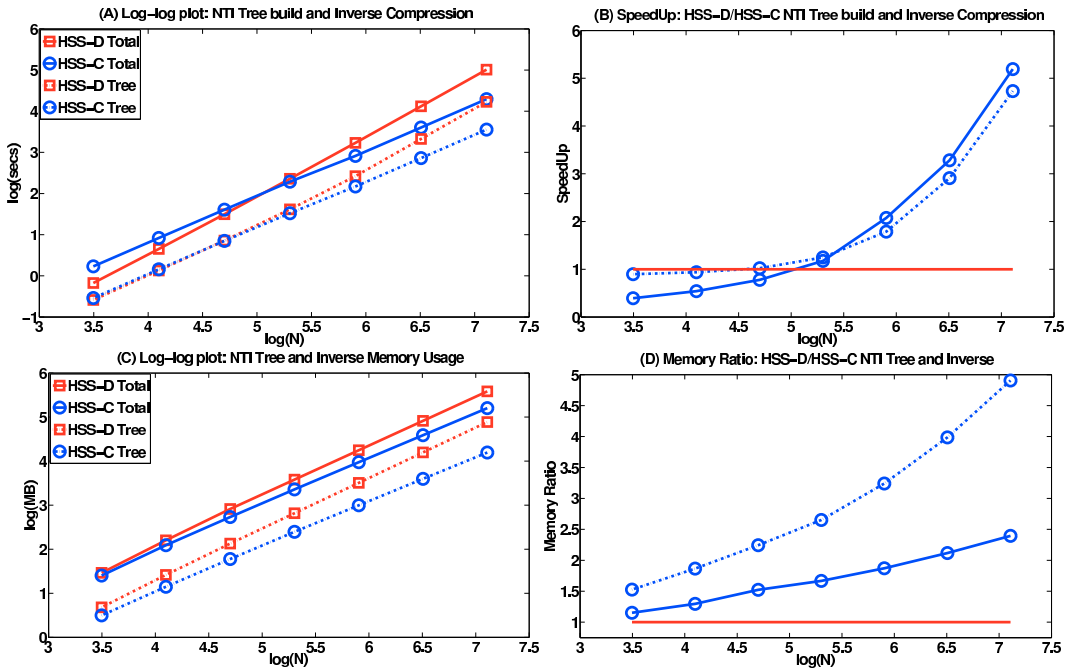


FIG. 5.1. NTI 2D Laplace Kernel: Inverse compression timings and memory usage. Total times and storage are represented in solid lines, Tree build and storage in dashed lines.

5.1.2. Translation-invariant kernel. Exploiting translation invariance yields a substantial improvement in performance, which stems from the fact that only one set of matrices per level of

\mathcal{T} needs

to be computed and stored. The full-block HSS inverse compression algorithm scales as $O(N^{3/2})$, but is \times faster for all sized problems and memory usage becomes more efficient as N grows. The lack of improvement in the asymptotic behavior of inverse compression when compared to the non-invariant case stems from the fact that $O(N^{3/2})$ work is performed on the top block of the tree in both cases.

As one could expect, our compressed-block algorithm displays linear scaling for both inverse compression time and storage, for the largest problems gaining an order of magnitude in both when compared to the NTI case. For $N = 10^7$, it has become about $11 \times$ faster (29 minutes, using 30 GB of memory (5 GB)). Building the binary tree, which as mentioned amounts to computing \mathcal{A} , displays similar performance gains. For $N = 10^7$, it takes only 4.5 minutes to build and storage is only 1.3 GB.

N	HSS-D Time $O(N^{3/2})$	HSS-C Time $O(N)$	HSS-D Memory $O(N)$	HSS-C Memory $O(N)$
784	0.05 s	0.13 s	1.94 MB	1.75 MB
3136	0.21 s	0.98 s	9.04 MB	6.19 MB
12544	1.40 s	3.41 s	39.16 MB	19.03 MB
50176	9.68 s	10.76 s	163.19 MB	52.09 MB
200704	1.21 m	30.89 s	666.39 MB	151.41 MB
802816	9.20 m	1.59 m	2.61 GB	474.74 MB
3211264	1.19 hr	6.68 m	9.9359 GB	1.56 GB
12845056	9.28 hr	29.22 m	39.74 GB	5.29 GB

TABLE 5.2
Total inversion time and storage for the TI inverse compression algorithms

The algorithm achieves parity with the dense-block HSS for lower values of N (and $N = 50000$), for which both methods take about 10 seconds to produce the HSS inverse. By $N = 10^7$, our algorithm is about $20 \times$ faster than the dense-block approach (Figure 5.2). We again observe that the build time is only about 15%-20% of the total inversion time, and so construction of the matrix compression holds Figure 5.2.C as a linear scaling for memory usage. It remains that our algorithm scales memory in all cases. For $N = 10^7$, it uses $8 \times$ less memory (5 GB, which amounts to 50 degrees of freedom).

5.1.3. Inverse matrix-vector multiplication. In order to solve the inverse matrix algorithm, we utilize an random ighthand sides and compute time profile (in seconds) for both NTI and TI examples. We note that given the same matrix blocks are applied to the same block of each ighthand side, code can be easily vectorized leading to performance gains if multiple solves are to be performed simultaneously. If our matrix is non-invariant additional speedup from the fact that the same set of matrices are applied to corresponding to all blocks in a given level.

We note that for all cases the inverse display scaling linearly and remaining well under a minute for $N \sim 10^7$. As expected, the differences between the standard and compressed block appear small, the latter becoming marginally faster around $N \sim 3 \times 10^6$.

For all cases if \mathcal{A} and \mathcal{A}^{-1} are computed, a point matrix and inverse, via the following error measure:

$$E = \frac{\|v - \mathcal{A}\mathcal{A}^{-1}v\|}{\|v\|}$$

Taking the maximum over a number of randomly generated ighthand sides v . In his means, which can be thought of as an approximate residual, both algorithms achieve the desired accuracy for the inverse apply. We may then bound the exact residual by

$$\|v - \mathcal{A}\mathcal{A}^{-1}v\| \leq E + \|\mathcal{A} - A\| \|\mathcal{A}^{-1}v\|$$

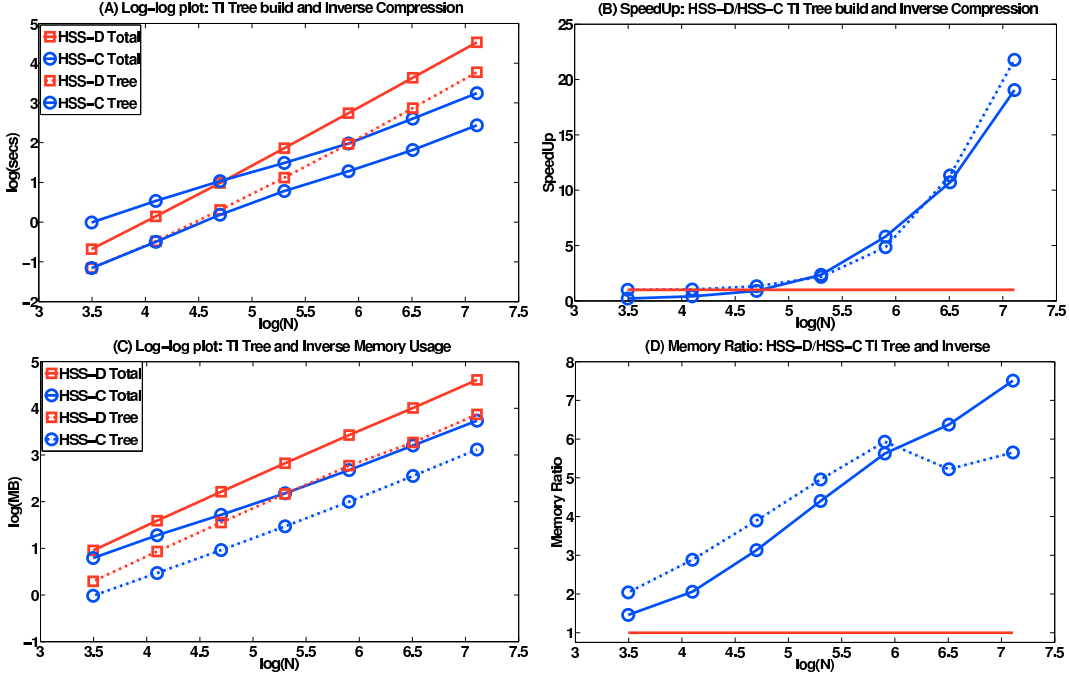


FIG. 5.2. TI 2D Laplace kernel: Inverse compression timings and memory usage. Total times and storage are represented in solid lines, Tree build and storage in dashed lines.

N	NTI HSS-D $O(N \log N)$	NTI HSS-C $O(N)$	TI HSS-D $O(N \log N)$	TI HSS-C $O(N)$	NTI HSS-C Error	TI HSS-C Error
784	0.0014	0.0018	0.0007	0.0011	1.6e-14	9.2e-15
3136	0.0064	0.0090	0.0031	0.0046	1.8e-14	1.8e-14
12544	0.0292	0.0362	0.0137	0.0162	8.6e-11	5.7e-11
50176	0.1320	0.1546	0.0590	0.0600	1.6e-10	1.7e-10
200704	0.5993	0.6772	0.2819	0.2512	2.3e-10	1.6e-10
802816	2.6611	2.8193	1.2709	1.0763	4.0e-10	3.8e-10
3211264	11.816	11.737	5.77296	4.5650	5.1e-9	1.6e-9
12845056	52.468	48.8641	25.8312	19.3619	-	3.2e-9

TABLE 5.3

Inverse apply timings (in seconds) for both NTI and TI algorithms. Numbers in red are extrapolated from previous data.

5.2. The effect of varying accuracy. If we want a lower accuracy, this can significantly speed up both algorithms presented above. The number of boundary layers needed, the size of kernel \mathcal{T} , and the number of HSS 1D blocks in the compressed-block algorithm are all low and low-rank factorization can be performed faster.

In the case of using the Laplace single layer approximation in 2D, we compare their behavior for target accuracies $\varepsilon = 10^{-5}$ and 10^{-10} .

Dense-block algorithm. In this case, only the change in kernel size is relevant. As in Section 4, we can bound $k_i \leq k_\ell = C_\varepsilon n_\ell^{1/2}$. In particular, $\varepsilon = 10^{-5}$ requires one boundary layer and $\varepsilon = 10^{-10}$ requires two, and so $C_{10^{-10}} \sim 2C_{10^{-5}}$, as expected from standard multiple estimates ($C_\varepsilon \sim \log(1/\varepsilon)$). Following our complexity analysis, we observe both the build and inverse compression algorithms perform $O(k_i^3)$ work per box and so the constant in the leading term is of the form $\hat{C} \log^3(1/\varepsilon)$. This would imply a factor of 8 between $\varepsilon = 10^{-5}$ and $\varepsilon = 10^{-10}$. Analogously, inverse matrix-vector and memory usage are $O(k_i^2)$ per box and so we expect a factor of 4 between these two cases.

Compressed-block algorithm. Performance of the compressed block algorithm depends on all the quantities mentioned above: the size of 2D HSS skeletons is again $O(\log(n_\ell/\varepsilon))$, but operations are performed on these. Hence, we expect a factor of $\log(1/\varepsilon)$ to appear at all stages. Assuming $O(\log(n_\ell/\varepsilon))$, the most expensive one-dimensional dense-block HSS operations are linear in a constant $O(\log^3(1/\varepsilon))$, and again matrix and vector operations are $O(\log^2(1/\varepsilon))$.

Hence, we expect the factor to behave as $O(\log^4(1/\varepsilon))$ ($16\times$) for the build and inverse compression, and as $O(\log^3(1/\varepsilon))$ ($8\times$) for inner products and memory usage.

N	HSS-D Time	Ratio	HSS-C Time	Ratio	HSS-D Memory	Ratio	HSS-C Memory	Ratio
784	0.01 s	3.1	0.03 s	6.7	0.45 MB	3.3	0.44 MB	4.1
3136	0.03 s	5.2	0.07 s	15	1.92 MB	3.7	1.75 MB	4.2
12544	0.20 s	6.4	0.75 s	6	7.98 MB	3.8	5.40 MB	4.3
50176	1.35 s	6.9	2.83 s	6	32.53 MB	3.9	13.67 MB	4.4
200704	9.77 m	7.4	7.87 s	6.4	131.39 MB	4	36.59 MB	4.4
802816	1.22 m	7.6	20.88 s	7.3	528.19 GB	4	107.38 MB	4.4
3211264	9.13 m	7.9	1.19 m	8	2.07 GB	4	349.50 MB	4.2
12845056	1.14 hr	8.1	4.56 m	8.9	8.34 GB	4	1.21 GB	3.8

TABLE 5.4

Inverse compression time and storage of the TI algorithms for $\varepsilon = 10^{-5}$, and the ratio of the quantities corresponding to 10^{-10} and 10^{-5} target accuracy

Experimental observations indicate that the dense block algorithm (HSS-D) is close to N operations, while the ratio between compression times is consistent with 8 , and that for memory usage is 4 , as expected. However, although the memory ratio in the compressed-block case is generally observed to be consistent, since the ratios are fairly similar to the dense-block case.

Only essential parts can be considered for the inverse algorithms as outlined in Section 2 and 3 for all cases.

We omit the analysis for the non-invariant cases, since the compression and inversion are both expected to be similar. Overall, this implies that the analysis and observations in Section 5.1 can be applied in the modification to the case $\varepsilon = 10^{-5}$: for both algorithms a factor of 8 times faster build and 4 times less memory and can be applied to the inverse than the high accuracy case.

5.3. Low Frequency Oscillatory Kernels. Finally, we perform the same set of experiments as in Section 5.1, but this time taking $\mathcal{K}(r)$ in its original formulation to be the 2D Helmholtz free-space Green's function for wave number k :

$$(5.1) \quad \mathcal{G}_k(r) = -\frac{i}{4} H_0^1(kr)$$

where H_0^1 is the Hankel function of the first kind. As will be seen in Section 5.4, this is akin to solving the Lipmann-Schwinger equation for a given frequency k .

As mentioned before, in some minor modifications are necessary to handle oscillatory kernels for low frequencies. We briefly note the main differences in non-oscillatory problems

- **Skeleton Structure:** For high accuracy skeletons consisting of more layers of pins are needed in compression in non-oscillatory kernels such as Laplace. Besides this, as the wave number k is small, it is necessary to keep pins inside of the box in the skeleton structure. Experimentally, we determine that keeping several pins per wavelength is enough to maintain accuracy and inverse compression.
- **Conditioning:** The Lipmann-Schwinger equation is moderately ill-conditioned, in condition number depending on k . This impacts the main issues in both levels of compression: the singular values of accuracy. We note that this is mild for the wave numbers used, and

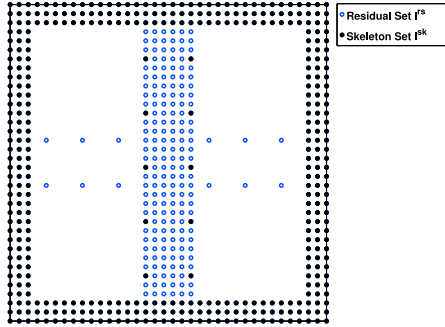


FIG. 5.3. 2D Helmholtz kernel skeleton set structure: Skeleton points are in black and residual points in blue. A few points per wavelength are kept on the interface between the two children boxes.

that we can always gain them back by one round of iterative refinement (matrix and inverse) and they remain in fact.

- **Rank growth assumptions:** Ultimately as k grows the assumptions stated in Section 4 will fail: The size of skeletons in compressed blocks will depend on k , as well as the number of points needed in order to discretize.

This points to the fact that a different approach is needed to build a fast direct solver that can handle moderate and high frequency regimes.

5.3.1. Inverse compression results: TI and NTI cases. We present experimental results for both the non-ankalton-invariant and the ankalton-invariant cases. Let $\kappa = k/2\pi$ be the number of wavelengths that fit in our domain, the unit box. For $\kappa = 4, 8, 16, 32$, we set a leaf box size at $n_{max} = 9^2$ and test for increasing problem size N . Here we compare total inverse time and memory usage for the HSS-D and HSS-C inverse compression algorithms.

We observe that while the cases for $\kappa = 4, 8$ behave similarly to their Laplace counterparts (kernels consist of 2 boundary layers), for $\kappa = 16, 32$ and higher at least 3 layers are needed, and accuracy in matrix compression deteriorates unless we keep a few extra points per wavelength inside the box. As will see, this change in kernel structure is the main cause for differences in behavior between these two sets of examples.

We note that given that the Helmholtz kernel is complex-valued, we double the speed for each matrix entry. This implies that memory usage can be expected to be at least 2 times bigger than that of real-valued kernels.

We first observe that for all cases empirical scaling again coincides with the expected complexity (the $O(N^{3/2})$ HSS-D algorithm is not plotted in figure 5.4 to avoid clutter). For the HSS-C algorithm, look at 5.4.A if you wish to reach a speedup of 1 as N grows.

Break-even points and speedup: on the speed plot in figure 5.4.B, we can observe that the point where the compressed-block algorithm overcomes its dense-block counterpart goes slightly in κ . While it is around 10^5 for $\kappa = 4, 8$, it goes closer to 10^6 for $\kappa = 16, 32$. We can also see a moderate speedup gained after the break-even point due to better scaling, similarly to the case for Laplace.

Memory Usage: In figures 5.4.C and 5.4.D we again can observe the HSS-C algorithm always provides better compression in terms of storage, and this improves as N grows. By $N \sim 10^6$, it is 3 – 4 times less memory than HSS-D.

For $N \sim 10^6$ and $\kappa = 4, 8$, the HSS-C algorithm takes about 0.3 hours to produce the HSS inverse, requiring 27 GB of storage. For $\kappa = 16, 32$, it takes 7 and 10 hours to build the inverse, respectively, and storage requirements go up to $\sim 60GB$. Comparing its performance with the Laplace NTI kernel in Section 5.1 (which takes 0.5 hours and 10 GB for this size), we see that as the wavenumber increases it takes considerably more time and storage to compress the inverse. The most dramatic change can be observed for $\kappa = 16, 32$, since kernel sizes become much bigger in size.

Exploiting ankalton invariance yields significant performance gains for all tested wavenumbers. The HSS-D algorithm is again $O(N^{3/2})$, and about 3 times faster than the NTI case. We also observe that linear scaling for inverse compression and storage for the HSS-C version, although a bit less dramatic

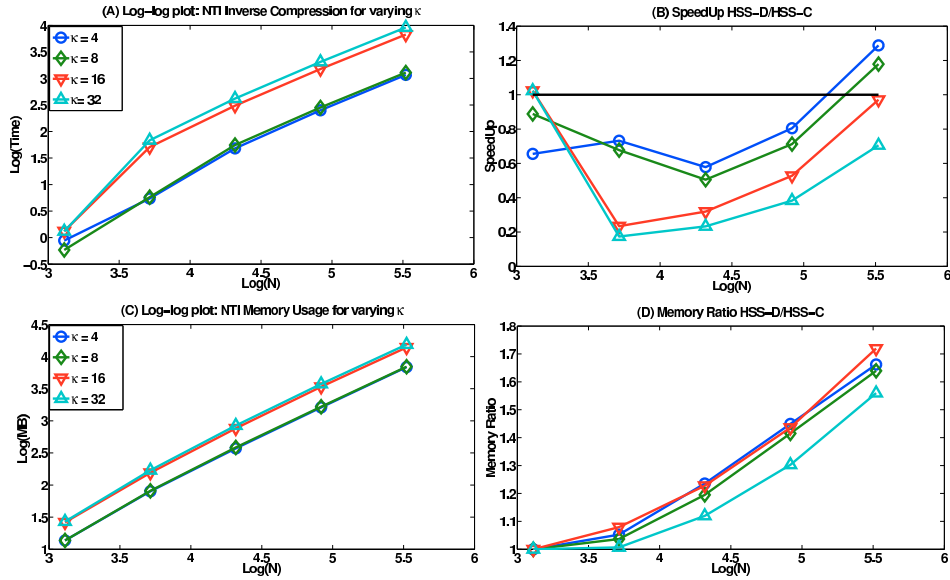


FIG. 5.4. NTI 2D Helmholtz kernel: Inverse compression timings and memory usage for increasing values of κ

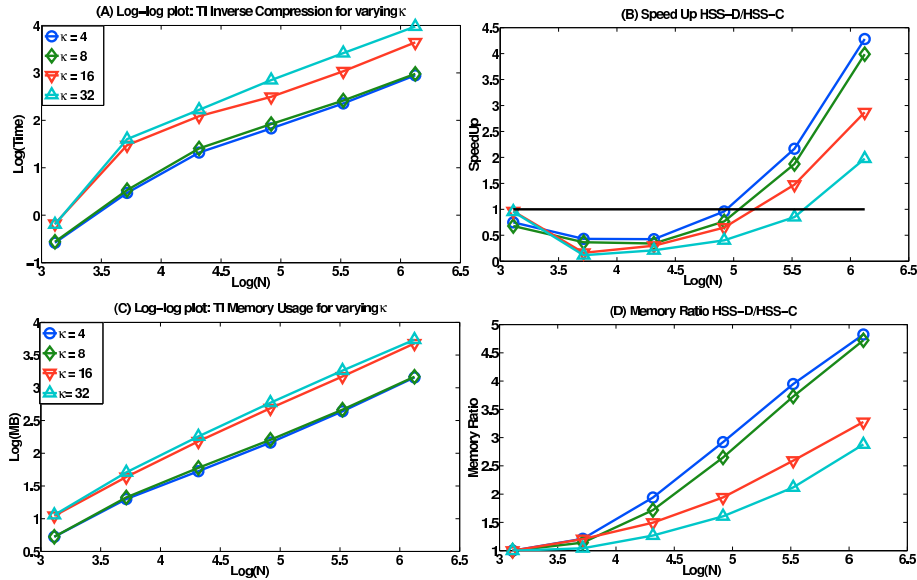


FIG. 5.5. TI 2D Helmholtz kernel: Inverse compression timings and memory usage for increasing values of κ

han in he cas for he Lapace kernel. By $N \sim 10^6$, it has become $5 - 6 \times$ faster in $15 \times$ less memory

Since only one set of matrices is computed for each level in \mathcal{T} , we can observe more clearly the impact of adding an extra layer of kernel points in inverse compression time and usage become about 1 order of magnitude higher. Also, here is a more pronounced difference between $\kappa = 16$ and $\kappa = 32$ for bigger problem sizes.

Break-even points and speedup: On the plot in figure 5.5.B, we have break-even points a bit smaller than computed in the NTI case, and that they gain growth with κ . For $\kappa = 4$ it is below 10^5 , for $\kappa = 32$, it is around 3×10^5 . Speedup again considerably better and far going due to bilinear scaling, especially for small numbers.

Memory Usage: The gain in compression is more rapid, especially for lower numbers. By $N \sim 10^6$, it is $3 - 5$ times less memory than HSS-D. Also, in 5.5.D, we see the slope for the memory ratio is higher than in the NTI case.

For $N \sim 10^6$ and $\kappa = 4, 8$, the HSS-C algorithm takes about 15 minutes to solve the HSS inverse, using 1.5 GB of memory. For $\kappa = 16, 32$, they go to 1.2 and 2.6 hours to build the inverse, taking 4.6 and 5 GB of memory. Comparing its performance in the Laplace TI kernel (2 minutes and 0.5 GB for this), we observe similar differences in memory as in the NTI case, but much more data ones in terms of inverse computation time.

5.3.2. Inverse matrix-vector multiplication. Finally, we test the inverse apply algorithm as in Section 5.1.3. Since the differences between dense and compressed block are small, we report only for the HSS-C algorithm for $\kappa = 8$ and $\kappa = 32$. The inverse apply is still considerably faster and in optimal scaling.

N	NTI HSS-C $\kappa = 8$	NTI HSS-C $\kappa = 32$	TI HSS-C $\kappa = 8$	TI HSS-C $\kappa = 32$	TI HSS-C Eor($\kappa = 8$)	TI HSS-C Eor($\kappa = 32$)
1296	0.0036	0.0074	0.0022	0.0048	1e-13	1e-12
5184	0.0205	0.0596	0.0126	0.0365	5.5e-10	2.6e-8
20736	0.1102	0.3046	0.0587	0.1663	1.6e-10	8.7e-9
82944	0.5197	1.4962	0.2465	0.8148	2.3e-9	3.6e-8
331776	2.5431	6.7812	1.0350	3.4867	1.0e-10	2.0e-8
1327104	10.1684	27.1248	4.3536	15.4991	1.2e-9	4.9e-8

TABLE 5.5

Inverse apply timings (in seconds) for both NTI and TI algorithms, for different values of κ . Numbers in red are extrapolated from previous data.

For $\kappa = 16, 32$, we allow 6 digits in accuracy means ($\sim 10^{-8}$), then compressed to 10^{-10} . As mentioned above, this is an effect of conditioning and can be addressed if necessary by banking accuracy by one order of iterative refinement.

5.4. 2D scattering problem: Lippmann-Schwinger equation.

Background on scattering. Consider an acoustic scattering problem in \mathbb{R}^2 involving a “soft scatterer” contained in a domain Ω . Formally, we assume that the “incoming field” is generated from a point source at the point $x_s \in \Omega^c$. A local mathematical model could take the form:

$$(5.2) \quad -\Delta u(x) - \frac{\omega^2}{v(x)^2} u(x) = \delta(x - x_s), \quad x \in \mathbb{R}^2$$

where ω is the frequency of the incoming wave, and here $v(x)$ is the wave speed at x . We assume that the wave speed is constant in Ω , so that $v(x) = v_0$ for $x \in \Omega^c$. To make the equation well-posed, we assume that $u(x)$ satisfies the natural “radiation condition” at infinity. We define the “wave number” k as

$$k = \frac{\omega}{v_0},$$

and a function $b = b(x)$ that quantifies the “deviation” in the wave number from the free-space wave number k

$$b(x) = k^2 - \frac{\omega^2}{v(x)^2}.$$

Observe that $b(x) = 0$ outside Ω . Equation (5.2) then takes the form

$$(5.3) \quad -\Delta u(x) - k^2 u(x) + b(x) u(x) = \delta(x - x_s), \quad x \in \mathbb{R}^2.$$

Finally, let G_k denote the Helmholtz fundamental solution as in equation 5.1, and let the solution of (5.3) into “incoming” and “outgoing” fields so that $u = u_{in} + u_{out}$, where

$$u_{in}(x) = G_k(x, x_s).$$

Then u_{out} satisfies

$$(5.4) \quad -\Delta u_{out}(x) - k^2 u_{out}(x) + b(x) u_{out}(x) = -b(x) u_{in}(x), \quad x \in \Omega.$$

When converting (5.4) as an integral equation, we assume that $b(x)$ is non-negative, and look for a solution of the form

$$(5.5) \quad u_{out}(x) = [S_k(\sqrt{b}\tau)](x) = \int_{\Omega} G_k(x, y) \sqrt{b(y)} \tau(y) dy.$$

Inserting (5.5) into (5.4), and dividing the result by $\sqrt{b(x)}$, we find the equation

$$(5.6) \quad \tau(x) + \int_{\Omega} \sqrt{b(x)} G_k(x, y) \sqrt{b(y)} \tau(y) dy = -\sqrt{b(x)} u_{in}(x), \quad x \in \Omega$$

Test problem. For numerical tests we take:

$$b(x_1, x_2) = 0.25k^2(1 + \tanh D(1 - \epsilon - |x_1|))(1 + \tanh D(1 - \epsilon - |x_2|))$$

Note that b is equal to 1 in a subdomain of $[-1, 1]^2$ and vanishes exponentially to 0 close to the boundary of the square. Parameters D and ϵ can be manipulated to change his localization and width. We note that if b close to 0 in machine precision, the test problem may be modified to check the behavior along the profile b .

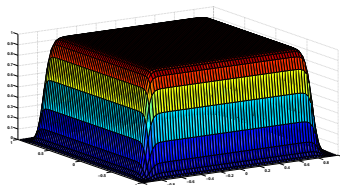


FIG. 5.6. Plot of deviation b for the scatterer in our Lippman-Schwinger equation

We implement an $O(h^4)$ accurate corrected approach in [14] by adding a diagonal correction to the kernel. Since our algorithm compresses off-diagonal blocks this has no bearing on the performance of our direct solver. We note that higher order (up to $O(h^{10})$) quadratures of this form can be applied if needed.

Numerical results. For $\kappa = 4, 8$ and increasing problem size N , we solve the Lippman-Schwinger equation (5.6) using our reformulation-invariant symmetric operators

For both for the approximate matrix \mathcal{A} and its inverse, we fit means the empirical order of convergence when applying them to the same right hand side while refining grid size h . We confirm that the order is approximately $O(h^4) = O(N^{-2})$ until the error is comparable to the target accuracy

Since here is a much higher condition in the kernel entries and they get close to 0 at the corners and columns corresponding to the box boundary high precision is needed. We observe that 2 layers yield moderate accuracy ($\sim 10^{-6}$), and 3 are needed for high accuracy ($\sim 10^{-10}$). We note that for cases where the small variation within the domain of interest Ω , his approach could be optimized by giving special treatment to those which intersect the boundary.

Also, we observe that the inverse of the localization-invariant operator in Section 5.3 may be used as a right preconditioner for this problem. Using BiCGstab, we find that for a given wave number κ , the number of iterations to solve the preconditioned system is moderate and independent of problem size N .

We show the direct solver and the preconditioned iterative solvers speed above (Table 5.6) For the latter, the comparison includes the test-build (matrix comparison) for the Lippman-Schwinger kernel, and inverse comparison for the corresponding localization-invariant Helmholtz kernel (operator-conditioned).

N	HSS-D Time	HSS-C Time	IterTime	HSS-D Memory	HSS-C Memory	IterMemory
784	1.92 s	1.86 s	1.72 s	18.74 MB	18.74 MB	9.63 MB
3136	11.81 s	24.93 s	9.30 s	125.87 MB	92.30 MB	54.37 MB
12544	1.31 m	2.68 m	1.13 m	720.90 MB	464.81 MB	258.97 MB
50176	8.03 m	14.20 m	5.65 m	3.69 GB	5.56 GB	1.19 GB
200704	47.13 m	1.17 h	25.71 m	18.34 GB	13.22 GB	5.53 GB
802816	4.78 h	5.70 h	1.93 h	90.36 GB	47.16 GB	25.38 GB

TABLE 5.6

Total inverse compression time and storage for the Lippmann-Schwinger equation for $\kappa = 8$ and $\varepsilon = 10^{-10}$ target accuracy

N	HSS-D Solv	HSS-C Solv	HSS Err	IterSolv	Iter#	Err
784	0.03 s	0.02 s	2.7e-11	0.59 s	20	8.0e-9
3136	0.08 s	0.17 s	5.2e-9	3.13 s	30	9.2e-9
12544	0.36 s	0.82 s	7.1e-8	14.34 s	28	4.4e-9
50176	1.74 s	3.73 s	3.6e-11	56.12 s	28	3.6e-9
200704	9.54 s	20.21 s	9.3e-11	236.23 s	29.5	7.7e-9
802816	52.50 s	109.32 s	1.2e-10	1010.1 s	29.5	4.4e-9

TABLE 5.7

Inverse apply timings for the Lippmann-Schwinger equation for $\kappa = 8$ and $\varepsilon = 10^{-10}$ target accuracy

Finally on Table 5.7 we compare inverse apply (solv) time for both algorithms and for the iterative approach.

We note that for the HSS-C algorithm, one round of adaptive refinement is needed in order to obtain the target accuracy for the approximate solver. This means two and one matrix solves are used in the solver.

6. Conclusions and Future Work. We have described a direct solver for volume integral equations in the plane that attains optimal $O(N)$ complexity and high practical efficiency for problems with non-oscillatory (or moderately oscillatory) kernels. The solver displays high performance for large problems even for high target accuracies. It gains a significant additional advantage when dealing with translation invariant kernels since only one set of fast matrix solves needs to be computed for each level of the tree, leading to a linear scaling of inverse compression time and storage.

The solver is based on the recursive skeletonization scheme described in [31], which would have $O(N^{3/2})$ complexity if applied to the volume integral equations considered here. We attained the acceleration to optimal $O(N)$ complexity by using fast matrix algebra to manipulate certain large dense matrices that arise in the computation. Specifically we used the so-called *Hierarchically Semi-Separable (HSS)*, format which exploits rank deficiencies in the off-diagonal blocks of the matrix. As noted in Section 5, the accelerated direct solver provides better compression than the dense-block algorithm of [31] in all tested environments and it becomes faster for sizes greater than $N \sim 10^5$. Because of optimal scaling, this implies a one order of magnitude speedup for very large N in our order n problem.

We also note that the increase in memory efficiency of our algorithm is of significant importance: given that all fast-matrix and FMM-like methods are memory intensive, the lack of efficient compression can render them prohibitive for large enough problems.

The fast solver described can be modified into an $O(N)$ matrix compression algorithm which is competitive with the FMM and maybe used in iterative methods such as GMRES, see Section 5.

Future work directions. There are several directions we plan to explore:

- **Extension to surfaces in 3D:** we expect that the compressed-block HSS algorithms presented in this paper can be ported with no major structural differences to the case of solving boundary integral equations on surfaces in 3D, and ongoing work features this extension. The main challenges will most likely come from a potentially more complex skeletonization and

from implementation considerations in more complex geometries
 Having such fast direct solvers available is of more relevance to solve large scale boundary
 value and evolution PDE problems in applications such as fluid-structure interaction and
 3D scattering problems

- **Parallel implementation:** as we have noted, the algorithms and implementation presented are serial. However, direct solvers are ideally suited for parallel implementation: intensive linear algebra operations are performed for each node on a binary tree, and the most expensive computations (in-core compression) are only done once and as
- **Direct solvers for oscillatory problems:** limited fast direct solvers such as those needed to tackle oscillatory problems arising from acoustics and electromagnetics (e.g. Helmholtz equation and Maxwell equations) in moderate and high frequency regimes

Acknowledgments: We would like to thank Mak Tytgert and Leslie Greengard's group for many helpful insights during the design and implementation of our algorithm.

REFERENCES

- [1] K. E. Atkinson. *The numerical solution of integral equations of the second kind*. Cambridge University Press, Cambridge, 1997.
- [2] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:4, 1986.
- [3] Mario Bebendorf. *Hierarchical matrices*, volume 63 of *Lecture Notes in Computational Science and Engineering*. Springer-Verlag, Berlin, 2008. A means to efficiently solve elliptic boundary value problems.
- [4] S. Börm. \mathcal{H}^2 -matrix arithmetics in linear complexity. *Computing*, 77(1):1–28, 2006.
- [5] S. Börm. Construction of data-sparse \mathcal{H}^2 -matrices by hierarchical compression. *SIAM Journal on Scientific Computing*, 31(3):1820–1839, 2009.
- [6] S. Börm, L. Grasedyck, and W. Hackbusch. Hierarchical matrices. *Lecture notes*, 21, 2003.
- [7] Steffen Börm. *Efficient numerical methods for non-local operators*, volume 14 of *EMS Tracts in Mathematics*. European Mathematical Society (EMS), Zürich, 2010. \mathcal{H}^2 -matrix compression, algorithms and analysis.
- [8] J. Bremer. A fast direct solver for the integral equations of scattering theory on planar curves with corners. *Journal of Computational Physics*, 2011.
- [9] S. Chandrasekaran, P. Dewilde, M. Gu, W. Lyons, and T. Pals. A fast solver for HSS representations via sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 29(1):67–81, 2006.
- [10] S. Chandrasekaran, P. Dewilde, M. Gu, and N. Somasunderam. On the numerical rank of the off-diagonal blocks of Schur complements of discretized elliptic PDEs. *SIAM Journal on Matrix Analysis and Applications*, 31(5):2261–2290, 2010.
- [11] S. Chandrasekaran, M. Gu, and T. Pals. A fast ULV decomposition solver for hierarchically semiseparable representations. *SIAM Journal on Matrix Analysis and Applications*, 28(3):603–622, 2006.
- [12] Y. Chen. A fast, direct algorithm for the Lippmann–Schwinger integral equation in two dimensions. *Advances in Computational Mathematics*, 16(2):175–190, 2002.
- [13] Patrick Dewilde and Shivkumar Chandrasekaran. A hierarchical semi-separable Moore-Penrose equation solver. *Wavelets, Multiscale Systems and Hypercomplex Analysis*, page 69.
- [14] R. Duan and V. Rokhlin. High-order quadratures for the solution of scattering problems in two dimensions. *Journal of Computational Physics*, 228(6):2152–2174, 2009.
- [15] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford, 1989.
- [16] A. George. Nested dissection of a regular finite element mesh. *SIAM J. on Numerical Analysis*, 10:345–363, 1973.
- [17] A. Gillman. *Fast direct solvers for elliptic partial differential equations*. PhD thesis, University of Colorado, 2011.
- [18] A. Gillman, P.M. Young, and P.G. Martinsson. A direct solver with $O(N)$ complexity for integral equations on one-dimensional domains. *Frontiers of Mathematics in China*, pages 1–31, 2012.
- [19] Zydrunas Gimbutas and Vladimir Rokhlin. A generalized fast multipole method for nonoscillatory kernels. *SIAM J. Sci. Comput.*, 24(3):796–817 (electronic), 2002.
- [20] L. Greengard, D. Gueyffier, P.G. Martinsson, and V. Rokhlin. Fast direct solvers for integral equations in complex three-dimensional domains. *Acta Numerica*, 18(1):243–275, 2009.
- [21] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, 1987.
- [22] Leslie Greengard and Vladimir Rokhlin. A new version of the fast multipole method for the Laplace equation in three dimensions. In *Acta numerica, 1997*, volume 6 of *Acta Numer.*, pages 229–269. Cambridge Univ. Press, Cambridge, 1997.
- [23] W. Hackbusch and Z.P. Nowak. On the fast matrix multiplication in the boundary element method by panel clustering. *Numerische Mathematik*, 54(4):463–491, 1989.
- [24] N. Halko, P.G. Martinsson, and J.A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.
- [25] Kenneth L Ho and Leslie Greengard. A fast direct least squares algorithm for hierarchically block separable matrices. *arXiv preprint arXiv:1212.3521*, 2012.
- [26] K.L. Ho and L. Greengard. A fast direct solver for structured linear systems by recursive skeletonization. *SIAM Journal on Scientific Computing*, 34(5):2507–2532, 2012.

- [27] A. J. Hoffman, M. S. Martin, and D. J. Rose. Complexity bounds for regular finite difference and finite element grids. *SIAM J. Numer. Anal.*, 10:364–369, 1973.
- [28] Sabine Le Borne, Lars Grasedyck, and Ronald Kriemann. Domain-decomposition based \mathcal{H} -LU preconditioners. In *Domain decomposition methods in science and engineering XVI*, volume 55 of *Lect. Notes Comput. Sci. Eng.*, pages 667–674. Springer, Berlin, 2007.
- [29] E. Liberty, F. Woolfe, P.G. Martinsson, V. Rokhlin, and M. Tygert. Randomized algorithms for the low-rank approximation of matrices. *Proceedings of the National Academy of Sciences*, 104(51):20167–20172, 2007.
- [30] Per-Gunnar Martinsson. A fast direct solver for a class of elliptic partial differential equations. *J. Sci. Comput.*, 38(3):316–330, 2009.
- [31] P.G. Martinsson and V. Rokhlin. A fast direct solver for boundary integral equations in two dimensions. *Journal of Computational Physics*, 205(1):1–23, 2005.
- [32] P.G. Martinsson and V. Rokhlin. A fast direct solver for scattering problems involving elongated structures. *Journal of Computational Physics*, 221(1):288–302, 2007.
- [33] V. Rokhlin. Rapid solution of integral equations of classical potential theory. *J. Comput. Phys.*, 60(2):187–207, 1985.
- [34] Y. Saad and M.H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.
- [35] P.G. Schmitz and L. Ying. A fast direct solver for elliptic problems on general meshes in 2d. *Journal of Computational Physics*, 231(4):1314 – 1338, 2012.
- [36] Page Starr and Vladimir Rokhlin. On the numerical solution of two-point boundary value problems. II. *Comm. Pure Appl. Math.*, 47(8):1117–1159, 1994.
- [37] H.A. Van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.
- [38] F. Woolfe, E. Liberty, V. Rokhlin, and M. Tygert. A fast randomized algorithm for the approximation of matrices. *Applied and Computational Harmonic Analysis*, 25(3):335–366, 2008.
- [39] J. Xia. On the complexity of some hierarchical structured matrix algorithms. *SIAM Journal on Matrix Analysis and Applications*, 33(2):388–410, 2012.
- [40] J. Xia, S. Chandrasekaran, M. Gu, and X.S. Li. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications*, 17(6):953–976, 2010.
- [41] Jianlin Xia, Shivkumar Chandrasekaran, Ming Gu, and Xiaoye S. Li. Superfast multifrontal method for large structured linear systems of equations. *SIAM J. Matrix Anal. Appl.*, 31(3):1382–1411, 2009.
- [42] L. Ying, G. Biros, and D. Zorin. A kernel independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics*, 196(2):591–626, 2004.