Multidimensional Phase Recovery and Interpolative Decomposition Butterfly Factorization

Ze Chen
Department of Mathematics, National University of Singapore, Singapore

Juan Zhang
Department of Mathematics and Computational Science, Xiangtan University, China

Kenneth L. Ho
Center for Computational Mathematics, Flatiron Institute, USA

Haizhao Yang
Department of Mathematics, Purdue University, USA*
National University of Singapore, Singapore†

August 25, 2019

Abstract

This paper focuses on the fast evaluation of the matvec $g = Kf$ for $K \in \mathbb{C}^{N\times N}$, which is the discretization of a multidimensional oscillatory integral transform $g(x) = \int K(x,\xi)f(\xi)d\xi$ with a kernel function $K(x,\xi) = e^{2\pi i\Phi(x,\xi)}$, where $\Phi(x,\xi)$ is a piecewise smooth phase function with x and ξ in $\mathbb{R}^d$ for $d = 2$ or 3. A new framework is introduced to compute Kf with $O(N\log N)$ time and memory complexity in the case that only indirect access to the phase function Φ is available. This framework consists of two main steps: 1) an $O(N\log N)$ algorithm for recovering the multidimensional phase function Φ from indirect access is proposed; 2) a multidimensional interpolative decomposition butterfly factorization (MIDBF) is designed to evaluate the matvec Kf with an $O(N\log N)$ complexity once Φ is available. Numerical results are provided to demonstrate the effectiveness of the proposed framework.

Keywords. Data-sparse matrix, butterfly factorization, interpolative decomposition, operator compression, randomized algorithm, matrix completion.

1 Introduction

This paper is concerned with the efficient evaluation of multidimensional oscillatory integral transforms. After discretization with N grid points in each variable, the integral transform is reduced to a dense matrix-vector multiplication (matvec) as follows:

$$g(x) = \sum_{\xi\in\Omega} K(x,\xi)f(\xi) = \sum_{\xi\in\Omega} e^{2\pi i\Phi(x,\xi)}f(\xi), \quad x \in X, \tag{1}$$

where X and Ω are typically point sets in $\mathbb{R}^d$ for $d > 1$, $K(x,\xi) = e^{2\pi i\Phi(x,\xi)}$ is a kernel function, $\Phi(x,\xi)$ is a piecewise smooth phase function with $O(1)$ discontinuous points in x and ξ, $f(\xi)$ is a given function, and $g(x)$ is a target function.

When the explicit formula of the kernel function is known, the direct computation of matvec in (1) takes $O(N^2)$ operations and is prohibitive in large-scale computation. There has been an active research line

*Corresponding author and current address.
†The work was finished at the National University of Singapore from which H. Y. is currently on leave.

aiming at a nearly linear-scaling matvec for evaluating (1). In the case of uniformly distributed point sets X and Ω, the fast Fourier transform (FFT) [34] can evaluate (1) when $\Phi(x,\xi) = x \cdot \xi$ with $O(N\log(N))$ operations. When the point sets are non-uniform, the non-uniform FFT (NUFFT) algorithms in [12, 31] are able to evaluate (1) when $\Phi(x,\xi) = x \cdot \xi$ with $O(N\log(N))$ operations. For more general kernel functions, the butterfly factorization (BF) [20, 24, 26, 27] can factorize the dense matrix $e^{2\pi i\Phi(x,\xi)}$ as a product of $O(\log(N))$ sparse matrices, each of which has only $O(N)$ non-zero entries. Hence, storing and applying $e^{2\pi i\Phi(x,\xi)}$ via the BF for evaluating (1) take only $O(N\log(N))$ complexity.

However, for multidimensional kernel functions, existing algorithms are efficient only when the explicit formula of the phase function Φ is known [1, 31, 24, 6, 19, 20, 22, 27]. The computational challenge in the case of indirect access of the kernel function (see Table 1 for a list of different scenarios) motivates a series of new algorithms for multidimensional cases in this paper.

Scenario 1 :	There exists an algorithm for evaluating an arbitrary entry of the kernel matrix in $O(1)$ operations [3, 4, 20, 26].
Scenario 2 :	There exists an $O(N\log N)$ algorithm for applying K and its transpose to a vector [13, 20, 22, 30].
Scenario 3 :	The phase functions are solutions of partial differential equations (PDE's) [10]. $O(1)$ rows and columns of the phase matrices are available by solving PDE's.

Table 1: Three scenarios of the indirect access of the phase functions.

As the first main contribution of this paper, in the case of indirect access, a nearly linear scaling algorithm is proposed to recover multidimensional phase matrices in the form of low-rank matrix factorization. In scientific computing, several important problems require the construction of low-rank phase matrices [3, 4, 16, 29, 30, 7, 25, 33, 13]. Previously, a nearly linear scaling algorithm has been proposed in [36] to recover the low-rank phase matrix in 1D. However, the 1D algorithm in [36] is problematic in the case of high-dimensional nonuniform discretization grid points. In this paper, we address the problem in multidimensional cases via fast Delaunay triangulation (DT) and minimum spanning tree (MST) construction. Secondly, when low-rank phase matrices have been recovered, a new BF, multidimensional interpolative decomposition butterfly factorization (MIDBF), is proposed for the matvec Kf with $O(N\log N)$ complexity for both precomputation and application. The MIDBF is a generalization of the interpolative decomposition butterfly factorization (IDBF) [27] in multidimensional cases especially when the discretization grid points are non-uniform. These two contributions lead to the first framework for multidimensional fast oscillatory integral transforms in the case of indirect access with non-uniform grid points.

The rest of the paper is organized as follows. In Section 2, we revisit and generalize existing low-rank phase matrix factorization techniques, and propose a new low-rank matrix factorization in the case of indirect access. Next, the MIDBF will be introduced in Section 3. Finally, we provide several numerical examples to demonstrate the efficiency of the proposed framework in Section 4. For simplicity, we adopt MATLAB notations for the algorithm described in this paper.

2 Low-rank phase matrix factorization

This section introduces a new low-rank phase matrix factorization for indirect access, which is the first main step in the proposed framework. We begin with a brief review of existing techniques and introduce new algorithm afterward. These low-rank factorization methods will be repeatedly applied.

2.1 Low-rank approximation by randomized sampling

Let us revisit an existing low-rank matrix factorization with linear complexity. For $A \in \mathbb{C}^{m\times n}$, a rank-$r$ approximate singular value decomposition (SVD) of A is defined as

$$A \approx U\Sigma V^T, \tag{2}$$

where $U \in \mathbb{C}^{m\times r}$ is orthogonal, $\Sigma \in \mathbb{R}^{r\times r}$ is diagonal, and $V \in \mathbb{C}^{n\times r}$ is orthogonal, and $r = O(1)$ independent of the matrix size m and n with a prefactor depending only on the approximation error. Previously, [11, 14] have proposed efficient randomized tools to compute approximate SVDs for numerically low-rank matrices. The method in [11] is more attractive because it only requires $O(1)$ randomly sampled rows and columns of A for constructing (2) with $O(m+n)$ operation and memory complexity.

The method in [11] is denoted as Function `randomizedSVD` and is presented in Algorithm 1. Assuming the whole low-rank matrix A is known, the input of Function `randomizedSVD` is A, $O(1)$ randomly sampled row indices $\mathcal{R}$ and column indices $\mathcal{C}$, as well as a rank parameter r. Equivalently, it can also be assumed that $A(\mathcal{R},:)$ and $A(:,\mathcal{C})$ are known as the inputs. The outputs are three matrices $U \in \mathbb{C}^{m\times r}$, $\Sigma \in \mathbb{R}^{r\times r}$, and $V \in \mathbb{C}^{n\times r}$ satisfying (2). In Function `randomizedSVD`, for simplicity, given any matrix K, Function `qr` performs a pivoted QR decomposition $KP = QR$, which can return an index set Π corresponding to the r most important columns of K that span the column space of K, or return a set of orthonormal vectors Q that span the column space of K. Function `randperm(m,r)` denotes an algorithm that randomly selects r different samples in the set $\{1, 2, \ldots, m\}$. If necessary, we can add an over sampling parameter q such that we sample rq rows and columns and only generate a rank r truncated SVD in Line 10 in Algorithm 1. Larger q results in better stability of Algorithm 1.

1 **Function** $[U, \Sigma, V] \leftarrow$ `randomizedSVD(`$A, \mathcal{R}, \mathcal{C}, r$`)`
2 $[m, n] \leftarrow \mathrm{size}(A)$
3 $\Pi_{col} \leftarrow$ `qr(`$A(\mathcal{R},:), r$`)` `// `Π_{col} `: the first` r `important columns of` $A(\mathcal{R},:)$
4 $\Pi_{row} \leftarrow$ `qr(`$A(:,\mathcal{C})^T, r$`)` `// `Π_{row} `: the first` r `important columns of` $A(:,\mathcal{C})^T$
5 $Q \leftarrow$ `qr(`$A(:,\Pi_{col})$`)` ; $Q_{col} \leftarrow Q(:, 1:r)$ `// `$A(:,\Pi_{col})P = QR$
6 $Q \leftarrow$ `qr(`$A(\Pi_{row},:)^T$`)` ; $Q_{row} \leftarrow Q(:, 1:r)$ `// `$A(\Pi_{row},:)^T P = QR$
7 $S_{row} \leftarrow$ `randperm(`m, r`)` ; $I \leftarrow [\Pi_{row}, S_{row}]$
8 $S_{col} \leftarrow$ `randperm(`n, r`)` ; $J \leftarrow [\Pi_{col}, S_{col}]$
9 $M \leftarrow (Q_{col}(I,:))^\dagger A(I,J) \left(Q_{row}^T(:,J)\right)^\dagger$ `// `$(\cdot)^\dagger$ `: pseudo-inverse`
10 $[U_M, \Sigma_M, V_M] \leftarrow \mathrm{svd}(M)$
11 $U \leftarrow Q_{col}U_M$; $\Sigma \leftarrow \Sigma_M$; $V \leftarrow Q_{row}V_M$

Algorithm 1: Randomized sampling for a rank-r approximate SVD.

2.2 One-dimensional phase matrix factorization with indirect access

A nearly linear scaling algorithm for constructing the low-rank factorization of the phase matrix $\Phi \in \mathbb{R}^{N\times N}$ in (1) has been proposed in [36] when a 1D kernel matrix $K = e^{2\pi i\Phi}$ is available as in Scenarios 1 and 2 in Table 1. In this section, we revisit the algorithms in [36] as a motivation for the multidimensional one proposed in this paper. The introduction of the 1D algorithms also helps to clarify the difficulties in the multidimensional case.

The difficulty of reconstructing Φ from $K = e^{2\pi i\Phi}$ comes from the fact that

$$\frac{1}{2\pi}\Im\left(\log\left(K(i,j)\right)\right) = \frac{1}{2\pi}\Im\left(\log\left(e^{2\pi i\Phi(i,j)}\right)\right) = \frac{1}{2\pi}\arg\left(e^{2\pi i\Phi(i,j)}\right) = \mathrm{mod}(\Phi(i,j), 1),$$

where $\Im(\cdot)$ returns the imaginary part of the complex number, and $\arg(\cdot)$ returns the argument of a complex number. Thus, Φ is only known up to modular 1.

The main idea of [36] is to recover Φ by looking for the solution of the following combinatorial constrained TV^3-norm[1] minimization problem:

$$\begin{aligned} \min_{\Phi\in\mathbb{R}^{N\times N}} \quad & \sum_{i\in\mathcal{R}} \|\Phi(i,:)\|_{TV^3} + \sum_{j\in\mathcal{C}} \|\Phi(:,j)\|_{TV^3} \\ \text{subject to} \quad & \mathrm{mod}(\Phi(i,j),1) = \frac{1}{2\pi}\Im\left(\log\left(K(i,j)\right)\right) \text{ for } i\in\mathcal{R} \text{ or } j\in\mathcal{C}, \end{aligned} \tag{3}$$

[1]The TV^3-norm of a vector $v \in \mathbb{R}^N$ is defined as $\|v\|_{TV^3} := \sum_{i=4}^N |v_i - 3v_{i-1} + 3v_{i-2} - v_{i-3}|$ in this paper.

where $\mathcal{R}$ and $\mathcal{C}$ are row and column index sets with $O(1)$ randomly selected indices, respectively. The optimization problem above is appealing because it only requires the knowledge of $O(1)$ rows and columns of K and the computational cost in each iteration takes $O(N)$ operations and memory. If the optimization problem could be solved in $O(1)$ iterations, then the recovered rows and columns of Φ can be used to compute the low-rank factorization of Φ by Function `randomizedSVD` in Algorithm (1). The final computational cost is nearly linear in N. However, due to the non-convexity of (3), $O(1)$ iterations are almost impossible to give a good solution unless a very good initial guess is available. This motivates [36] to design an empirical $O(N)$ algorithm to provide a good initial guess to the optimization in (3).

The main algorithms of [36] are revisited and summarized in Algorithm 2 and Algorithm 3 in this paper for the preparation of higher dimensional cases. Algorithm 3 relies on the repeated application of Algorithm 2, which adjusts the values of phase vectors by minimizing the absolute value of the third-order derivative, to provide an empirical solution to (3). The functions in these two algorithms are denoted as `RecoveryVector1` and `RecoveryMatrix1`, respectively. In fact, the algorithms presented in this paper are slightly different from those in [36] for robustness against discontinuity detection, which relies on a class of vectors C_τ with a threshold τ defined via:

$$C_\tau = \{u \in \mathbb{R}^n : |u(i) - 3u(i-1) + 3u(i-2) - u(i-3)| < \tau, \forall i \in \{4, 5, \ldots, n\}\}. \tag{4}$$

Essentially, C_τ consists of vectors with a small absolute value of the third order derivative controlled by τ in the sense of finite difference. In our algorithms, if $|u(i) - 3u(i-1) + 3u(i-2) - u(i-3)| \geq \tau$, we will consider the original function that generates u is discontinuous at the location corresponding to $u(i)$. With this definition ready, we are able to explain our algorithms as follows.

For Function `RecoveryVector1` in Algorithm 2, input variables are a vector u of length N, a discontinuity detection parameter τ, and a parameter $flag$ which indicates whether u will be recovered from the first entry or the fourth entry. Then, the outputs are a smooth vector v satisfying $\mathrm{mod}(v, 1) = \mathrm{mod}(u, 1)$ and a vector of indices $\mathcal{D}$ for discontinuity locations.

```
Function [v, D] = RecoveryVector1(u, τ, flag)
    N ← length(u);   v ← u;   D ← [1];   n ← 1;   c ← 1
    while c ≤ n do
        st ← D(c)
        if flag ~= 1 or st ~= 1 then
            v(st + 1) ← u(st + 1) − round(u(st + 1) − v(st))
            v(st + 2) ← u(st + 2) − round(u(st + 2) − 2v(st + 1) + v(st))
        for a = st + 3 : N do
            v(a) ← u(a) − round(u(a) − 3v(a − 1) + 3v(a − 2) − v(a − 3))
            if |v(a) − 3v(a − 1) + 3v(a − 2) − v(a − 3)| ≥ τ and a ≤ N − 3 then
                D ← [D, a];   n ← n + 1                    // detect discontinuous locations
                v(a) ← u(a) − round(u(a) − v(a − 1))
                Break
        c ← c + 1
```

Algorithm 2: An $O(N)$ algorithm for recovering a vector v from the observation $u = \mathrm{mod}(v, 1)$. The locations of discontinuity in v are automatically detected. A vector v is identified via empirically minimizing the magnitude of the absolute value of its third-order derivative.

In Function `RecoveryMatrix1` in Algorithm 3, one of the input variables is a function handle Φ, which can evaluate an arbitrary row or column of the phase matrix. The other inputs are a vector $\mathcal{R}$ and a vector $\mathcal{C}$ as the row and column index sets indicating $O(1)$ randomly selected rows and columns of the phase matrix, as well as a discontinuity detection parameter τ.

Because it is more convenient to apply Algorithm 2 to recover a vector representing a continuous function, we first apply Algorithm 2 with τ to identify the sets of discontinuous points $\mathcal{D}_r$ and $\mathcal{D}_c$, each of which contains the first index 1. Next, the phase matrix is partitioned into $n_r \times n_c$ blocks, each of which is denoted as $\Phi.\mathcal{B}_s\mathcal{B}_t$ representing a continuous piece of the phase function, where n_r is the cardinality of $\mathcal{D}_r$, n_c is the cardinality of $\mathcal{D}_c$, $s = 1, 2, \ldots, n_r$, and $t = 1, 2, \ldots, n_c$. This procedure is referred to as the Function

`Partition1` in Line 6 in Algorithm 3. Similarly, $\mathcal{R}$ and $\mathcal{C}$ are partitioned into n_r and n_c parts by $\mathcal{D}_r$ and $\mathcal{D}_c$, and saved as $\mathcal{R}.\mathcal{B}_s$ and $\mathcal{C}.\mathcal{B}_t$ respectively. For example, Panel (a) in Figure 1 visualizes an example when the phase function contains only 4 continuous blocks: $\Phi.\mathcal{B}_1\mathcal{B}_1$, $\Phi.\mathcal{B}_1\mathcal{B}_2$, $\Phi.\mathcal{B}_2\mathcal{B}_1$, $\Phi.\mathcal{B}_2\mathcal{B}_2$. Panel (c) and (d) in Figure 1 visualize the randomly selected rows $\mathcal{R}.\mathcal{B}_1$ and columns $\mathcal{C}.\mathcal{B}_1$ in $\Phi.\mathcal{B}_1\mathcal{B}_1$.

Finally, the selected rows and columns are recovered by Algorithm 2 with a carefully designed order in Line 9 - 13 in Algorithm 3. The parameter for detecting discontinuous points is set to 1 since there is no need to detect discontinuity anymore. Note that there is no uniqueness for recovering a smooth vector from its values after mod 1. Hence, we introduce the specially designed order in Line 9 - 13 to guarantee that each recovered row and column at their intersection share the same value, as long as the discontinuous points in the phase function are well distinguished by a parameter τ from continuous points.

1 **Function** $[\Phi, \mathcal{R}, \mathcal{C}] =$ `RecoveryMatrix1`$(\Phi, \mathcal{R}, \mathcal{C}, \tau)$
2 $\quad \mathcal{D}_r \leftarrow$ `RecoveryVector1`$(\Phi(:, \mathcal{C}(1)), \tau, 0)$ `// ` $\mathcal{D}_r$ `: discontinuous point set`
3 $\quad \mathcal{D}_c \leftarrow$ `RecoveryVector1`$(\Phi(\mathcal{R}(1), :), \tau, 0)$ `// ` $\mathcal{D}_c$ `: discontinuous point set`
4 $\quad \mathcal{R} \leftarrow [\mathcal{R}, \mathcal{D}_r]$; $\quad \mathcal{C} \leftarrow [\mathcal{C}, \mathcal{D}_c]$
5 $\quad n_r \leftarrow \text{length}(\mathcal{D}_r)$; $\quad n_c \leftarrow \text{length}(\mathcal{D}_c)$
6 $\quad [\Phi, \mathcal{R}, \mathcal{C}] \leftarrow$ `Partition1`$(\Phi, \mathcal{R}, \mathcal{C}, \mathcal{D}_r, \mathcal{D}_c)$
7 $\quad$ **for** $s = 1 : n_r$ **do**
8 $\quad\quad$ **for** $t = 1 : n_c$ **do**
9 $\quad\quad\quad \Phi.\mathcal{B}_s\mathcal{B}_t(1, :) \leftarrow$ `RecoveryVector1`$(\Phi.\mathcal{B}_s\mathcal{B}_t(1, :), 1, 0)$
10 $\quad\quad\quad \Phi.\mathcal{B}_s\mathcal{B}_t(:, k) \leftarrow$ `RecoveryVector1`$(\Phi.\mathcal{B}_s\mathcal{B}_t(:, k), 1, 0)$ for $k = 1, 2, 3$
11 $\quad\quad\quad \Phi.\mathcal{B}_s\mathcal{B}_t(k, :) \leftarrow$ `RecoveryVector1`$(\Phi.\mathcal{B}_s\mathcal{B}_t(k, :), 1, 1)$ for $k = 2, 3$
12 $\quad\quad\quad \Phi.\mathcal{B}_s\mathcal{B}_t(\mathcal{R}.\mathcal{B}_s(k), :) \leftarrow$ `RecoveryVector1`$(\Phi.\mathcal{B}_s\mathcal{B}_t(\mathcal{R}.\mathcal{B}_s(k), :), 1, 1)$ for all k
13 $\quad\quad\quad \Phi.\mathcal{B}_s\mathcal{B}_t(:, \mathcal{C}.\mathcal{B}_t(k)) \leftarrow$ `RecoveryVector1`$(\Phi.\mathcal{B}_s\mathcal{B}_t(:, \mathcal{C}.\mathcal{B}_t(k)), 1, 1)$ for all k

Algorithm 3: An $O(N)$ algorithm for the approximate solution of the TV^3-norm minimization when the phase function $\Phi(x, \xi)$ is defined on $\mathbb{R} \times \mathbb{R}$.

Once the phase function recovery algorithm in Algorithm 3 is ready, following the idea of low-rank matrix factorization via randomized sampling in Algorithm 1, we can obtain a nearly linear scaling algorithm to construct the low-rank factorization of the phase matrix.

(a) (b) (c) (d)

Figure 1: An illustration of the low-rank matrix recovery for a 1D phase matrix in Algorithm 3. (a) Line 6 partitions the phase matrix into submatrices such that there is no discontinuity along rows and columns in each submatrix. Then, Line 9 - 10 recovers the first row and column of each submatrix. (b) Next, Line 10 recovers the second and the third columns for each submatrix. (c) Next, Line 11 - 12 recovers $O(1)$ rows (including the second row and the third row) of each submatrix. (d) Finally, Line 13 recovers $O(1)$ columns of each submatrix.

2.3 Multidimensional phase matrix factorization with indirect access

2.3.1 Overview

In this section, a nearly linear scaling algorithm for constructing the low-rank factorization of multidimensional phase matrix $\Phi \in \mathbb{R}^{N \times N}$ will be introduced when we only know the kernel matrix $K = e^{2\pi i \Phi}$ through Scenarios 1 and 2 in Table 1. $N = n^d$ is the number of points in a d-dimensional domain, where $d = 2$ or 3, n is the number of points in each dimension. Our algorithm aims to use $O(1)$ randomly selected rows and

columns of $K = e^{2\pi i\Phi}$ to recover the corresponding rows and columns of Φ. Then, the low-rank factorization of Φ can be constructed by the randomized SVD in Section 2.1.

In Scenario 1, $O(1)$ randomly selected rows and columns of K can be directly evaluated. In Scenario 2, applying the kernel matrix K and its transpose to $O(1)$ randomly chosen natural basis vectors in $\mathbb{R}^N$ can obtain the rows and columns of K. Similar to the 1D case, instead of recovering the exact Φ that generates K, our primary purpose is to find a low-rank matrix Ψ such that

$$\operatorname{mod}(\Psi, 1) = \frac{1}{2\pi}\Im(\log(K)). \tag{5}$$

Based on the piecewise smoothness of the multidimensional phase function, a recovery algorithm similar to the 1D case can be proposed to recover the rows and columns of Φ up to an additive error matrix E that is numerically low-rank, i.e., the method returns a matrix $\Psi = \Phi + E$ such that $e^{2\pi i\Psi} = e^{2\pi i\Phi}$ and E is numerically low-rank. However, the discretization of the integral operator especially in the case of non-uniform grid point can introduce "artificial" discontinuity along the row and column of the phase matrix. Hence, it is impossible to apply the vector class C_τ and the algorithms in the 1D case. Although informally the recovery problem can be stated as

$$\begin{aligned} &\text{Find piecewise smooth } \Psi(i,:) \text{ and } \Psi(:,j) \text{ for } i \in \mathcal{R} \text{ and } j \in \mathcal{C} \\ \text{subject to} \quad &\operatorname{mod}(\Psi(i,j),1) = \frac{1}{2\pi}\Im(\log(K(i,j))) \text{ for } i \in \mathcal{R} \text{ or } j \in \mathcal{C}, \end{aligned} \tag{6}$$

where $\mathcal{R}$ and $\mathcal{C}$ are row and column index sets with $O(1)$ randomly selected indices, respectively, the vectors $\Psi(i,:)$ and $\Psi(:,j)$ are not "smooth" at the location when adjacent entries are corresponding to non-adjacent points in the high-dimensional spatial domain in $\mathbb{R}^d$. In other words, the definition of the smoothness of these vectors should rely on the smoothness of the phase function in the original domain in $\mathbb{R}^d$ instead of the difference of adjacent entries as in (4).

2.3.2 Vector recovering in the high-dimensional case

Let us use the example of a vector recovery in the high-dimensional case to illustrate the ideas to conquer the difficulty mentioned above. Suppose v is the discretization of a piecewise smooth function $\phi(x)$ with N (possibly nonuniform) grid points in $[0,1]^d$ and $O(1)$ pieces of domains in which $\phi(x)$ is smooth. The spatial locations of the N grid points are stored in a matrix $\mathcal{X} \in \mathbb{R}^{N\times d}$, i.e., $\mathcal{X}(i,:)$ is the location of the i-th entry of v. Assume that k is a vector representing $e^{2\pi i\phi(x)}$ using the same discretization. Informally, the vector recovery problem is to find a "piecewise smooth" vector v subject to $\operatorname{mod}(v,1) = \frac{1}{2\pi}\Im(\log(k))$.

To conquer the difficulty of artificial discontinuity, the entry values of v are identified via minimizing the variation of $\phi(x)$ using physically adjacent locations in $\mathbb{R}^d$. For this purpose, we introduce a special recovery path matrix $P \in \mathbb{Z}^{(N-1)\times 2}$ such that $P(:,2)$ is a permutation of $\{2,3,\ldots,N\}$, and $(P(i,1),P(i,2))$ is a pair of indices of v with corresponding spatial locations adjacent to each other in $\mathbb{R}^d$, i.e., $\mathcal{X}(P(i,1),:)$ is an adjacent grid point of $\mathcal{X}(P(i,2),:)$ in $\mathbb{R}^d$.

If the recovery path matrix P and a set of indices for discontinuous locations $\mathcal{D}$ are given, the recovery of v could be solved via the optimization problem:

$$\begin{aligned} \min_{v\in\mathbb{R}^N} \quad &\sum_{i\in\{1,\ldots,N-1\}\setminus\mathcal{D}} |v(P(i,2)) - v(P(i,1))| \\ \text{subject to} \quad &\operatorname{mod}(v,1) = \frac{1}{2\pi}\Im(\log(k)). \end{aligned} \tag{7}$$

We will revisit the construction of P later and focus on the construction of $\mathcal{D}$ and a nearly linear scaling empirical solution to (7) first. Similarly to the 1D case, to automatically detect discontinuity of the piecewise smooth function, we define a class of vectors $C_{\tau,P}$ for a threshold τ and a recovery path P via:

$$C_{\tau,P} = \{v \in \mathbb{R}^n : |v(P(i,2)) - v(P(i,1))| < \tau, \forall i \in \{1,2,\ldots,n-1\}\}.$$

$C_{\tau,P}$ consists of vectors with a small absolute value of the first order derivative controlled by τ in the sense of finite difference. In our assumption, if $|v(P(i,2)) - v(P(i,1))| \geq \tau$, we will consider the original function that generates v is discontinuous at the location $\mathcal{X}(P(i,2),:)$.

Function `RecoveryVector2` in Algorithm 4 below identifies a piecewise smooth vector v from a given vector $u = \frac{1}{2\pi}\Im(\log(k))$ via empirically minimizing $|v(P(i,2)) - v(P(i,1))|$ such that $\mathrm{mod}(v(P(i,2)),1) = u(P(i,2))$, for each $i = 1, 2, \ldots, N$ (corresponding to Line 5 in Algorithm 4). Each smooth piece of v belongs to $C_{\tau,P}$. The discontinuity location i will be detected and assigned to the discontinuity location set $\mathcal{D}$ if $|v(P(i,2)) - v(P(i,1))| \geq \tau$. It is clear that the complexity of Algorithm 4 to empirically solve (7) and detect discontinuity is $O(N)$. Note that Function `RecoveryVector2` in Algorithm 4 is based on the first-order derivative of the phase function while Function `RecoveryVector1` in Algorithm 2 is based on the third-order derivative. It is an simple extension to apply higher order derivative in Algorithm 4 using the high-order finite difference schemes in [17, 35], which is left as future work if necessary.

```
1 Function [v, D] = RecoveryVector2(u, τ, P)
2     N ← length(u);   D ← [1];   v ← u
3     for c = 1 : N − 1 do
4         bg ← P(c, 1);   ed ← P(c, 2)
5         v(ed) ← u(ed) − round(u(ed) − v(bg))
6         if |v(ed) − v(bg)| ≥ τ then
7             D ← [D, ed]                          // detect discontinuous locations
```

Algorithm 4: An $O(N)$ algorithm for recovering a vector v from the observation $u = \mathrm{mod}(v, 1)$ and detecting discontinuity using the recovery path matrix P.

The rest of the problem for vector recovery is to identify a recovery matrix P efficiently. A naive algorithm to identify an adjacent point of a given location is to traverse all other points, compute distances, and pick up the smallest one. However, this takes $O(N^2)$ operations to construct P for N points. This is the main difficulty of the extension of the 1D algorithm to high-dimensional cases. To solve this difficulty, we propose a new algorithm based on the Delaunay triangulation (DT) and the minimum spanning tree.

Definition 2.1. *For a set of points in the d-dimensional Euclidean space with locations $\mathcal{X} \in \mathbb{R}^{N\times d}$, a* **Delaunay triangulation** *is a triangulation DT($\mathcal{X}$) such that no point in this set is inside the circum-hypersphere of any d-simplex in DT($\mathcal{X}$).*

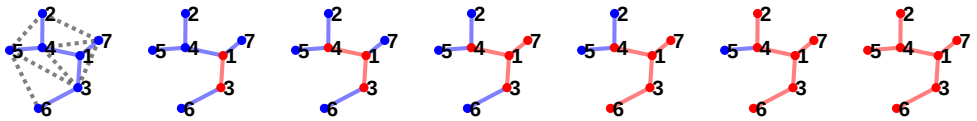
$\mathrm{DT}(\mathcal{X})$ can be treated as a fully connected undirected graph $\mathcal{G}$ with edges weighted by the Euclidean distance of two connected points and hence we can define the minimum spanning tree $\mathcal{T}(\mathcal{X})$ of $\mathrm{DT}(\mathcal{X})$ below.

Definition 2.2. *A* **minimum spanning tree** *(MST) $\mathcal{T}$ is a subset of the edges of a connected, edge-weighted undirected graph $\mathcal{G}$ that connects all the vertices, without any cycle and with the minimum possible total edge weight.*

An MST $\mathcal{T}(\mathcal{X})$ is an efficient representation of a graph $\mathcal{G} = \mathrm{DT}(\mathcal{X})$ to eliminate useless edges, based on the fact in [9] that the set of edges of $\mathrm{DT}(\mathcal{X})$ contains an MST for $\mathcal{X}$. A recovery path matrix P can be efficiently identified following the order of nodes in $\mathcal{T}(\mathcal{X})$ as follows.

Definition 2.3. *Given an MST $\mathcal{T}$ with N nodes and the root at Node q, a* **recovery path matrix** $P \in \mathbb{Z}^{(N-1)\times 2}$ *associated to $\mathcal{T}$ is a matrix such that 1) $P(:,2)$ is a permutation vector of $\{1, 2, \ldots, N\} \setminus q$; 2) the depth of Node $P(i,2)$ is less than or equal to that of Node $P(j,2)$ if $i \leq j$; 3) Node $P(i,1)$ is the predecessor node of Node $P(i,2)$ in $\mathcal{T}$ for all $i = 1, 2, \ldots, N-1$.*

Given the location matrix $\mathcal{X} \in \mathbb{R}^{N\times d}$ of N points in $\mathbb{R}^d$, it has been a standard routine to identify $\mathrm{DT}(\mathcal{X})$ with an expected runtime bounded by $O(N \log N)$ for $d = 2$ or 3 (e.g., see [5, 23, 32]). Next, the Prim's algorithm [28] is able to find $\mathcal{T}(\mathcal{X})$ with an $O(N \log N)$ complexity. Finally, the Breadth-First search algorithm [18] can be applied to $\mathcal{T}(\mathcal{X})$ and return a recovery matrix P with an $O(N)$ complexity. Figure 2 below visualizes an example of $\mathrm{DT}(\mathcal{X})$, $\mathcal{T}(\mathcal{X})$, and the corresponding P for $\mathcal{X} \in \mathbb{R}^{7\times 2}$. The whole algorithm



(a) (b) (c) (d) (e) (f) (g)

Figure 2: An illustration of DT($\mathcal{X}$), $\mathcal{T}(\mathcal{X})$, and the corresponding P for $\mathcal{X} \in \mathbb{R}^{7\times 2}$. (a) DT($\mathcal{X}$) (black dash line) and $\mathcal{T}(\mathcal{X})$ (in blue). (b) Starting from the root (Node 1), find the first undiscovered node, e.g., Node 3 with depth 1, then let $P = [1, 3]$. (c) Add $[1, 4]$ to P. (d) Add $[1, 7]$ to P. (e) find the first undiscovered node, e.g., Node 6 with depth 2, then add $[3, 6]$ to P. (f) Add $[4, 2]$ to P. (g) Add $[4, 5]$ to P. Finally, a recovery path matrix $P \in \mathbb{R}^{6\times 2}$ is set to be $P = [1, 3; 1, 4; 1, 7; 3, 6; 4, 2; 4, 5]$.

is summarized in Algorithm 5.

```
1 Function P = RecoveryPath(X)
2     G ← delaunayTriangulation(X);
3     T ← minspantree(G);
4     P ← bfsearch(T);
```

Algorithm 5: An $O(N \log N)$ algorithm for generating a recovery path matrix P.

2.3.3 Matrix recovery in the high-dimensional case

When the vector recovery algorithms in Algorithm 4 and Algorithm 5 are ready, we apply them to design a matrix recovery algorithm. Recall that the main idea is to identify piecewise smooth rows and columns of Ψ satisfying (5) as summarized in an informal problem statement in (6). Let $\mathcal{X}_1$ and $\mathcal{X}_2 \in \mathbb{R}^{N\times d}$ store the spatial locations of the N grid points for the discretization of $\Phi(x, \xi)$ in x and ξ, respectively.

First, Algorithm 5 is applied to construct the recovery path matrix P_1 and P_2 corresponding to $\mathcal{X}_1$ and $\mathcal{X}_2$, respectively. Then the matrix recovery problem can be formally stated as

$$\begin{aligned} \min_{\Phi\in\mathbb{R}^{N\times N}} \quad & \sum_{i\in\mathcal{R}} \sum_{s\in\{1,\ldots,N-1\}\setminus\mathcal{D}_c} |\Phi(i, P_2(s, 2)) - \Phi(i, P_2(s, 1))| \\ + \quad & \sum_{j\in\mathcal{C}} \sum_{t\in\{1,\ldots,N-1\}\setminus\mathcal{D}_r} |\Phi(P_1(t, 2), j) - \Phi(P_1(t, 1), j)| \\ \text{subject to} \quad & \mathrm{mod}(\Phi(i, j), 1) = \frac{1}{2\pi}\Im\left(\log\left(K(i, j)\right)\right) \text{ for } i \in \mathcal{R} \text{ or } j \in \mathcal{C}, \end{aligned} \tag{8}$$

where $\mathcal{D}_c$ and $\mathcal{D}_r$ are index sets indicating the discontinuous locations of Φ along columns and rows, $\mathcal{R}$ and $\mathcal{C}$ are row and column index sets with $O(1)$ randomly selected indices, respectively.

Next, Algorithm 4 is applied with τ to identify the sets of discontinuous points $\mathcal{D}_r$ and $\mathcal{D}_c$ to make (8) self-contained. Similarly to the 1D case, we can partition the phase matrix into (usually non-contiguous) submatrices corresponding to the domains in which the phase matrix is continuous, which is equivalent to dividing the MST $\mathcal{T}(\mathcal{X})$ into subtrees whenever an edge connects a predecessor node considered as a discontinuous point. Correspondingly, the recovery path matrix is partitioned into submatrices associated with these subtrees. Figure 3 below visualizes an example when an MST $\mathcal{T}(\mathcal{X})$ is partitioned into two MSTs at the discontinuity location at Node 4.

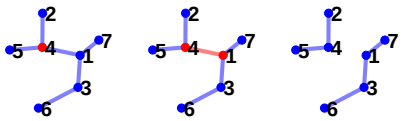
The partition procedure is denoted as Function `Partition2` in Algorithm 6, resulting in $n_r \times n_c$ submatrices of the phase matrix denoted as $\Phi.\mathcal{B}_s\mathcal{B}_t$, n_r submatrices of the recovery path matrix P_1, and n_s submatrices of the recovery path matrix P_2, for $s = 1, 2, \ldots, n_r$, and $t = 1, 2, \ldots, n_c$. The random samples of the row and column indices in the submatrices are denoted as $\mathcal{R}.\mathcal{B}_s$ and $\mathcal{C}.\mathcal{B}_t$, respectively. For example, Panel (a) in Figure 4 visualizes an example when the phase function contains only 4 continuous submatrices (from light color to dark color): $\Phi.\mathcal{B}_1\mathcal{B}_1$, $\Phi.\mathcal{B}_1\mathcal{B}_2$, $\Phi.\mathcal{B}_2\mathcal{B}_1$, $\Phi.\mathcal{B}_2\mathcal{B}_2$. Panel (b) in Figure 4 visualizes the root row and the root column of each submatrix. Panel (c) and (d) in Figure 4 visualize the randomly selected rows $\mathcal{R}.\mathcal{B}_1$ and columns $\mathcal{C}.\mathcal{B}_1$ in $\Phi.\mathcal{B}_1\mathcal{B}_1$.

Finally, we apply Algorithm 4 again to recover each submatrix. The parameter for detecting discontinuity is set to 1 since there is no need to detect discontinuity. The specially designed order also guarantees that each recovered row and column at their intersection share the same value, as long as the discontinuous points in the phase function have already been well distinguished.

Algorithm 6 below summarizes the above steps and the whole process is illustrated in Figure 4.

1 **Function** $\Phi = \texttt{RecoveryMatrix2}(\Phi, \mathcal{R}, \mathcal{C}, \mathcal{X}_1, \mathcal{X}_2, \tau)$
2 $P_1 \leftarrow \texttt{RecoveryPath}(\mathcal{X}_1)$; $P_2 \leftarrow \texttt{RecoveryPath}(\mathcal{X}_2)$
3 $\mathcal{D}_r \leftarrow \texttt{RecoveryVector2}(\Phi(:,1), \tau, P_1)$ // $\mathcal{D}_r$: discontinuous point set
4 $\mathcal{D}_c \leftarrow \texttt{RecoveryVector2}(\Phi(1,:), \tau, P_2)$ // $\mathcal{D}_c$: discontinuous point set
5 $\mathcal{R} \leftarrow [\mathcal{R}, \mathcal{D}_r]$; $\mathcal{C} \leftarrow [\mathcal{C}, \mathcal{D}_c]$
6 $n_r \leftarrow \text{length}(\mathcal{D}_r)$; $n_c \leftarrow \text{length}(\mathcal{D}_c)$
7 $[\Phi, \mathcal{R}, \mathcal{C}, P_1, P_2] \leftarrow \texttt{Partition2}(\Phi, \mathcal{R}, \mathcal{C}, P_1, P_2, \mathcal{D}_r, \mathcal{D}_c)$
8 **for** $s = 1 : n_r$ **do**
9 **for** $t = 1 : n_c$ **do**
10 $\Phi.\mathcal{B}_s\mathcal{B}_t(1,:) \leftarrow \texttt{RecoveryVector2}(\Phi.\mathcal{B}_s\mathcal{B}_t(1,:), 1, P_2.\mathcal{B}_t)$
11 $\Phi.\mathcal{B}_s\mathcal{B}_t(:,1) \leftarrow \texttt{RecoveryVector2}(\Phi.\mathcal{B}_s\mathcal{B}_t(:,1), 1, P_1.\mathcal{B}_s)$
12 $\Phi.\mathcal{B}_s\mathcal{B}_t(\mathcal{R}.\mathcal{B}_s(k),:) \leftarrow \texttt{RecoveryVector2}(\Phi(\mathcal{R}.\mathcal{B}_s(k),:), 1, P_2.\mathcal{B}_t)$ for all k
13 $\Phi.\mathcal{B}_s\mathcal{B}_t(:,\mathcal{C}.\mathcal{B}_t(k)) \leftarrow \texttt{RecoveryVector2}(\Phi(:,\mathcal{C}.\mathcal{B}_t(k)), 1, P_1.\mathcal{B}_s)$ for all k

Algorithm 6: An $O(N \log N)$ algorithm for the solution of matrix recovery problem 6 when the phase function $\Phi(x, \xi)$ is defined on $\mathbb{R}^d \times \mathbb{R}^d$.



(a) (b) (c)

Figure 3: (a) An MST $\mathcal{T}(\mathcal{X})$ with a discontinuity location at Node 4. (b) Separate $\mathcal{T}(\mathcal{X})$ at the edge between Node 4 and its predecessor Node 1. (c) Two resulting subtrees and the corresponding recovery path matrices $[1, 3; 1, 7; 3, 6]$ and $[4, 2; 4, 5]$.

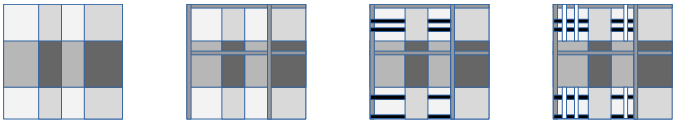

(a) (b) (c) (d)

Figure 4: An illustration of the low-rank matrix recovery for multidimensional phase matrix in Algorithm 6. (a) Line 7 partitions the phase matrix into 4 submatrices in 4 kinds of color such that there is no discontinuity along rows and columns in each submatrix. (b) Line 10 - 11 recovers the row and column of each submatrix corresponding to the root node of each sub-MST. (c) Line 12 recovers $O(1)$ rows of each submatrix. (d) Line 13 3 recovers $O(1)$ columns of each submatrix.

2.3.4 Phase matrix factorization

Once the phase function recovery algorithm in Algorithm 6 is ready, following the idea of low-rank matrix factorization via randomized sampling in Algorithm 1, we can introduce a nearly linear scaling algorithm to construct the low-rank factorization of the phase matrix as summarized in Algorithm 7. In particular, Algorithm 7 constructs a low-rank factorization UV^T, where $U \in \mathbb{C}^{N\times r}$ and $V \in \mathbb{C}^{N\times r}$, such that $e^{2\pi i UV^T} \approx e^{2\pi i\Phi}$ when we only know the kernel matrix $K = e^{2\pi i\Phi}$ through Scenarios 1 and 2 in Table 1.

In Algorithm 7, K (and Φ) is a function handle for evaluating an arbitrary entry of the kernel matrix, or evaluating an arbitrary row or column of K (and Φ). Two coordinate matrices $\mathcal{X}_1, \mathcal{X}_2 \in \mathbb{R}^{N\times d}$, a rank parameter r, an over-sampling parameter q, and the matrix size N are also inputs. We randomly select rq rows and columns of the kernel matrix and use `RecoveryMatrix2` to obtain the corresponding rows and columns of Ψ such that $e^{2\pi i\Psi} \approx K$. Finally, apply Function `randomizedSVD` in Algorithm 1 in Section 2.1 to evaluate the low-rank factorization of $\Psi \approx UV^T$ such that $e^{2\pi i UV^T} \approx K = e^{2\pi i\Phi}$. The reconstructed phase matrix can be set as an initial guess to the optimization problem in (8) and it takes $O(1)$ iterations

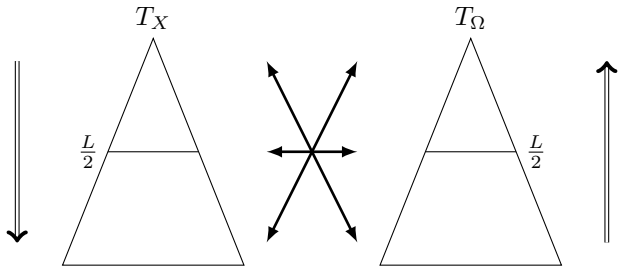


Figure 5: Trees of the row and column indices. Left: T_X for the row indices X. Right: T_Ω for the column indices Ω. The interaction between $A \in T_X$ and $B \in T_\Omega$ starts at the root of T_X and the leaves of T_Ω.

for sub-gradient descent methods to converge.

Function $[U, V] =$ `LowRankFactorization`$(K, \mathcal{X}_1, \mathcal{X}_2, r, q, N)$
  $\mathcal{R} \leftarrow$ `randperm(`N`,`rq`)` ; $\mathcal{C} \leftarrow$ `randperm(`N`,`rq`)`
  $\Phi \leftarrow \frac{1}{2\pi}\Im(\log(K))$ `// generate a function handle for the evaluation of` Φ
  $\Psi \leftarrow$ `RecoveryMatrix2(`$\Phi, \mathcal{R}, \mathcal{C}, \mathcal{X}_1, \mathcal{X}_2$`)`
  `// generate a function handle for the evaluation of` Ψ
  $[U, \Sigma, V] \leftarrow$ `randomizedSVD(`$\Psi, \mathcal{R}, \mathcal{C}, r$`)`
  $V \leftarrow V\Sigma$

Algorithm 7: Low-rank matrix factorization of phase functions in the case of indirect access. The computational complexity is bounded by $O(N \log N)$.

3 Multidimensional Interpolative Decomposition Butterfly Factorization (MIDBF)

This section will introduce the multidimensional interpolative decomposition butterfly factorization for a matrix $K = (K(x, \xi))_{x \in X, \xi \in \Omega}$ satisfying a complementary low-rank property [20], where X and Ω contain $O(N)$ points possibly uniformly distributed in $[0, 1)^d$ and d is the dimension of domain. As a special example, the kernel matrix $K(x, \xi) = e^{2\pi i \Phi(x,\xi)}$ satisfies the complementary low-rank property. Hence, once the phase function Φ has been recovered by Algorithm 7 in Section 2.1 in the form of low-rank factorization, we can construct a function handle to evaluate an arbitrary entry of K in $O(1)$ operations. Given this function handle, the MIDBF can construct the butterfly factorization of K for nearly linear scaling fast matvec.

Let us recall the definition of complementary low-rank matrices in [20]. For such a matrix, we construct two trees T_X and T_Ω for point sets X and Ω, respectively, assuming that both trees have the same depth $L = O(\log N)$, with the top-level being level 0 and the bottom one being level L (see Figure 5 for an illustration). Such a matrix K of size $N \times N$ is said to satisfy the **complementary low-rank property** if for any level ℓ, any node A in T_X at level ℓ, and any node B in T_Ω at level $L - \ell$, the submatrix $K(A, B)$, obtained by restricting K to the rows indexed by the points in A and the columns indexed by the points in B, is numerically low-rank.

3.1 Notations and overall structure

The notation of the 1D IDBF introduced in [27] will be adopted and adjusted to the multidimensional case in this paper. With no loss of generality, we focus on the 2D case with uniform point distributions first. The notations and overall structure discussed below are similar to that in [21, 27].

Recall that n is the number of grid points on each dimension, $N = n^2 = 4^L n_0$ is the total number of points, $n_0 = O(1)$ is the number of row or column indices in a leaf in the quadtrees of row and column spaces and, without loss of generality, L is an even integer, i.e. T_X and T_Ω with L levels. For a fixed level ℓ between 0 and L, the quadtree T_X has 4^ℓ nodes at level ℓ. By defining $\mathcal{I}^\ell = \{0, 1, \ldots, 4^\ell - 1\}$, we denote

these nodes by A_i^ℓ with $i \in \mathcal{I}^\ell$. These 4^ℓ nodes at level ℓ are further ordered according to a Z-order curve (or Morton order) as illustrated in Figure 6. Based on this Z-ordering, the node A_i^ℓ at level ℓ has four child nodes denoted by $A_{4i+t}^{\ell+1}$ with $t = 0, \ldots, 3$. The nodes plotted in Figure 6 for $\ell = 1$ (middle) and $\ell = 2$ (right) illustrate the relationship between the parent node and its child nodes. Similarly, in the quadtree T_Ω, the nodes at level $L - \ell$ are denoted as $B_j^{L-\ell}$ for $j \in \mathcal{I}^{L-\ell}$.

For any level ℓ between 0 and L, the kernel matrix K can be partitioned into $O(N)$ submatrices $K(A_i^\ell, B_j^{L-\ell}) := (K(x,\xi))_{x \in A_i^\ell, \xi \in B_j^{L-\ell}}$ for $i \in \mathcal{I}^\ell$ and $j \in \mathcal{I}^{L-\ell}$. For simplicity, we shall denote $K(A_i^\ell, B_j^{L-\ell})$ as $K_{i,j}^\ell$, where the superscript ℓ denotes the level in the quadtree T_X. Because of the complementary low-rank property, every submatrix $K_{i,j}^\ell$ is numerically low-rank with the rank bounded by a uniform constant r independent of N.

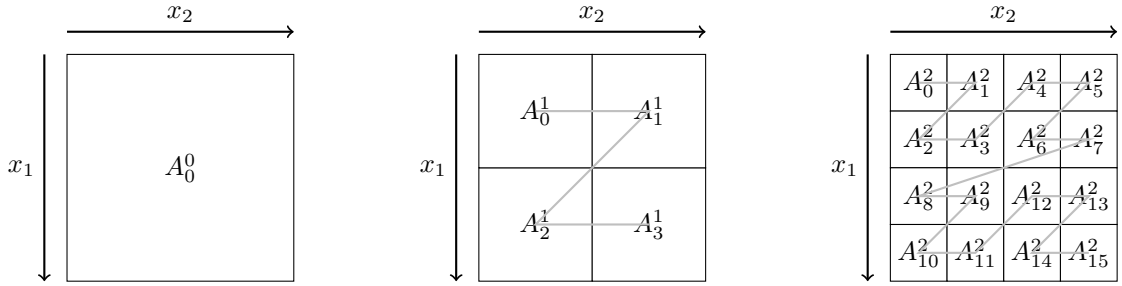


Figure 6: An illustration of Z-order curve cross levels. The superscripts indicate the different levels while the subscripts indicate the index in the Z-ordering. The light gray lines show the ordering among the subdomains on the same level. Left: The root at level 0. Middle: At level 1, the domain A_0^0 is divided into 2×2 subdomains A_i^1 with $i \in \mathcal{I}^1 = \{0, 1, 2, 3\}$. These 4 subdomains are ordered according to the Z-ordering. Right: At level 2, the domain A_0^0 is divided into 4×4 subdomains A_i^2 with $i \in \mathcal{I}^2 = \{0, 1, \ldots, 15\}$. These 16 subdomains are ordered similarly.

The multidimensional interpolative decomposition butterfly factorization for K is a product of $O(\log N)$ sparse matrices, each of which contains $O(\frac{k^2}{n_0} N)$ nonzero entries as follows:

$$K \approx U^L U^{L-1} \cdots U^h S^h V^h \cdots V^{L-1} V^L, \tag{9}$$

where k is a local rank parameter, $h = \frac{L}{2}$, and the level L is assumed to be even.

3.2 Linear scaling Interpolative Decompositions

This section introduces the linear scaling interpolative decomposition (ID) method in [27]. Suppose $K \in \mathbb{C}^{m \times n}$ has a numerical rank $k_\epsilon \ll \min\{m, n\}$, i.e., K admits a rank k_ϵ factorization with ϵ relative approximation accuracy. Let s be an index set containing tk rows of K chosen from the Mock-Chebyshev grids as in [36, 15, 2], t is an oversampling parameter, and k is an empirical estimation of k_ϵ. s is empirically selected and gradually increased if not large enough. We apply the rank revealing thin QR to $K(s, :)$:

$$K(s,:)\Lambda = QR = Q[R_1 \ R_2] \qquad \text{with} \qquad R_1 \in \mathbb{C}^{tk \times tk} \text{ and } R_2 \in \mathbb{C}^{tk \times (n-tk)}.$$

Define

$$T = (R_1(1:k, 1:k))^{-1}[R_1(1:k, k+1:kt) \ R_2(1:k,:)] \in \mathbb{C}^{k \times (n-k)},$$

and $V = [I \ T]\Lambda^* \in \mathbb{C}^{k \times n}$. Let q be the index set with $|q| = k$ such that

$$K(s, q) = QR_1(1:k, 1:k),$$

then q and V will satisfy

$$K(s,:) \approx K(s,q)V \tag{10}$$

with an approximation error by the QR truncation. By the approximation power of Lagrange interpolation with Mock-Chebyshev points if K is the discretization of a smooth function, we have

$$K \approx K(:, q)V \tag{11}$$

with an approximation error coming from the QR truncation and the Lagrange interpolation. Hence, $K(:, q)$ are important columns of K such that they can be "interpolated" back to K via a *column interpolation matrix* V. In this sense, q is called the *skeleton* index set, and the rest of indices are called *redundant* indices. This column ID requires only $O(nk^2)$ operations and $O(nk)$ memory and is denoted as *cID* for short.

Similarly, a row ID with $O(mk^2)$ operations and $O(mk)$ memory, denoted as *rID*, can be constructed via

$$K \approx \Lambda[I\ T]^* K(q, :) := UK(q, :) \tag{12}$$

with a *row interpolation matrix* U.

3.3 Leaf-root complementary skeletonization (LRCS)

This section introduces the LRCS of a 2D complementary low-rank kernel matrix K, $K \approx USV$, via *cID*s of the submatrices corresponding to the leaf-root levels of the column-row quadtrees (e.g., see the associated matrix partition in Figure 7 (right)), and *rID*s of the submatrices corresponding to the root-leaf levels of the column-row quadtrees (e.g., see the associated matrix partition in Figure 7 (middle)). Assume k_ϵ is constant in all IDs for low-rank approximations and denote it by k for simplicity.

Assume that the row index set r and the column index set c of K could be partitioned into leaves $\{r_i\}_{i\in\mathcal{I}^L}$ and $\{c_j\}_{j\in\mathcal{I}^L}$ at the leaf level of the row and column quadtrees as follows

$$r = [r_0, r_1, \cdots, r_{m-1}] \qquad (\text{and } c = [c_0, c_1, \cdots, c_{m-1}]), \tag{13}$$

with $|r_i| = n_0$ (and $|c_j| = n_0$) for all $0 \le i, j \le m-1$, where $m = 4^L = \frac{N}{n_0}$, $L = \log_4 N - \log_4 n_0$, and $L+1$ is the depth of quadtrees T_X and T_Ω. See an example of row and column quadtrees with $m = 16$ in Figure 7.

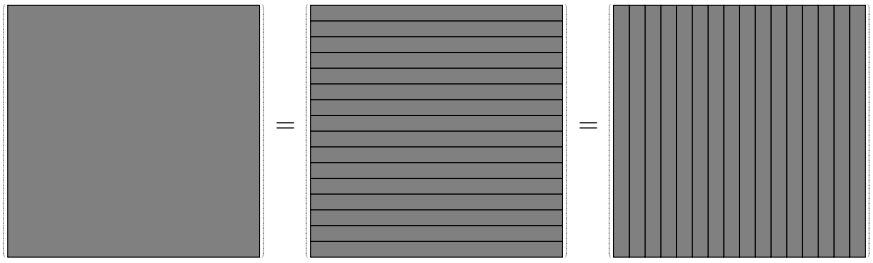


Figure 7: The left figure is a complementary two-dimensional low-rank kernel matrix K. Assume that the depth of the quadtrees of column and row spaces is 3. The middle figure illustrates the root-leaf partitioning that divides the row index set into 16 subsets as 16 leaves. The right one is for the leaf-root partitioning that divides the column index set into 16 subsets as 16 leaves.

Apply *rID* to each $K(r_i, :)$ to obtain the row interpolation matrix U_i and the associated skeleton indices $\hat{r}_i \subset r_i$ for all $0 \le i \le m-1$. Then, after denoting $K(\hat{r}, :)$ as the important skeleton of K, where

$$\hat{r} = [\hat{r}_0, \hat{r}_1, \cdots, \hat{r}_{m-1}], \tag{14}$$

we have

$$K \approx \begin{pmatrix} U_1 & & & \\ & U_2 & & \\ & & \ddots & \\ & & & U_m \end{pmatrix} \begin{pmatrix} K(\hat{r}_0, c_0) & K(\hat{r}_0, c_1) & \dots & K(\hat{r}_0, c_{m-1}) \\ K(\hat{r}_1, c_0) & K(\hat{r}_1, c_1) & \dots & K(\hat{r}_1, c_{m-1}) \\ \vdots & \vdots & \ddots & \vdots \\ K(\hat{r}_{m-1}, c_0) & K(\hat{r}_{m-1}, c_1) & \dots & K(\hat{r}_{m-1}, c_{m-1}) \end{pmatrix} := UM.$$

Similarly, apply *cID* to each $K(\hat{r}, c_j)$ to obtain the column interpolation matrix V_j and the skeleton indices $\hat{c}_j \subset c_j$ for all $0 \le j \le m-1$. Then, the LRCS of K will be formed as

$$K \approx \begin{pmatrix} U_1 & & & \\ & U_2 & & \\ & & \ddots & \\ & & & U_m \end{pmatrix} \begin{pmatrix} K(\hat{r}_0, \hat{c}_0) & K(\hat{r}_0, \hat{c}_1) & \dots & K(\hat{r}_0, \hat{c}_{m-1}) \\ K(\hat{r}_1, \hat{c}_0) & K(\hat{r}_1, \hat{c}_1) & \dots & K(\hat{r}_1, \hat{c}_{m-1}) \\ \vdots & \vdots & \ddots & \vdots \\ K(\hat{r}_{m-1}, \hat{c}_0) & K(\hat{r}_{m-1}, \hat{c}_1) & \dots & K(\hat{r}_{m-1}, \hat{c}_{m-1}) \end{pmatrix} \begin{pmatrix} V_1 & & & \\ & V_2 & & \\ & & \ddots & \\ & & & V_m \end{pmatrix} := USV. \tag{15}$$

For a concrete example, Figure 8 illustrates the non-zero pattern of the LRCS in (15) of K in Figure 7.

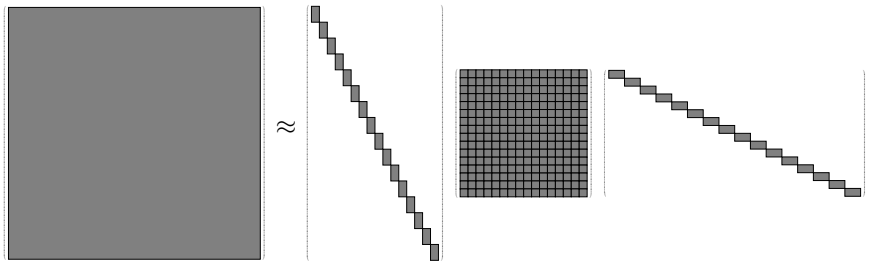


Figure 8: An example of the LRCS in (15) of the complementary two-dimensional low-rank kernel matrix K in Figure 7. Non-zero submatrices in (15) are shown in gray areas.

The main contribution of the LRCS is that M and S are only required to be generated and stored via the skeleton of row and column index sets with $O(\frac{k^3}{n_0}N)$ operations and $O(\frac{k^2}{n_0}N)$ memory, instead of being computed explicitly, since there are only $2m = \frac{2N}{n_0}$ IDs in total. Notice that the matrix S in $K \approx USV$ is also a complementary low-rank matrix. The row and column quadtrees $\hat{T}_X$ and $\hat{T}_\Omega$ of S are the compressed version of the row and column quadtrees T_X and T_Ω of K. If we consider $\hat{T}_X$ and $\hat{T}_\Omega$ as quadtrees with one depth less than the leaf level of T_X and T_Ω, they will be compressible.

3.4 Matrix splitting with complementary skeletonization (MSCS)

Now we introduce another key idea repeatedly applied in 2D IDBF, the MSCS. According to the nodes of the second level of the row and column quadtrees T_X and T_Ω (with $m = 4^L$ leaves), the complementary 2D low-rank kernel matrix K can be split into a 4×4 block matrix

$$K = \begin{pmatrix} K_{11} & K_{12} & K_{13} & K_{14} \\ K_{21} & K_{22} & K_{23} & K_{24} \\ K_{31} & K_{32} & K_{33} & K_{34} \\ K_{41} & K_{42} & K_{43} & K_{44} \end{pmatrix}. \tag{16}$$

It is obvious that K_{ij} is complementary low-rank for all $1 \le i, j \le 4$, with row and column quadtrees $T_{X,ij}$ and $T_{\Omega,ij}$ of depth $L-1$ and with $\frac{m}{4}$ leaves.

Suppose that the LRCS of each K_{ij} is $K_{ij} \approx U_{ij} S_{ij} V_{ij}$. Then, according to the LRCS of K_{ij}, the matrix splitting with complementary skeletonization (MSCS) of the kernel matrix K can be proposed as:

$$K \approx USV, \tag{17}$$

where

$$U = \begin{pmatrix} U_1 & U_2 & U_3 & U_4 \end{pmatrix} \quad \text{with} \quad U_k = \begin{pmatrix} U_{1k} & & & \\ & U_{2k} & & \\ & & U_{3k} & \\ & & & U_{4k} \end{pmatrix}, \tag{18}$$

$$S = \begin{pmatrix} \bar{S}_{11} & \bar{S}_{12} & \bar{S}_{13} & \bar{S}_{14} \\ \bar{S}_{21} & \bar{S}_{22} & \bar{S}_{23} & \bar{S}_{24} \\ \bar{S}_{31} & \bar{S}_{32} & \bar{S}_{33} & \bar{S}_{34} \\ \bar{S}_{41} & \bar{S}_{42} & \bar{S}_{43} & \bar{S}_{44} \end{pmatrix} \quad \text{with} \quad \bar{S}_{ij} \quad \text{as a 4 by 4 block matrix with the } (j,i)\text{-th block as } S_{ji}, \tag{19}$$

$$V = \begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{pmatrix} \quad \text{with} \quad V_k = \begin{pmatrix} V_{k1} & & & \\ & V_{k2} & & \\ & & V_{k3} & \\ & & & V_{k4} \end{pmatrix}. \tag{20}$$

Recall that the middle factor S is only required to be generated by some entries of the original kernel matrix, forming (17)-(20) will be a linear scaling algorithm as well. Figure 9 illustrates the MSCS of a complementary 2D low-rank kernel matrix K with quadtrees of depth 3 and 16 leaf nodes in Figure 7.

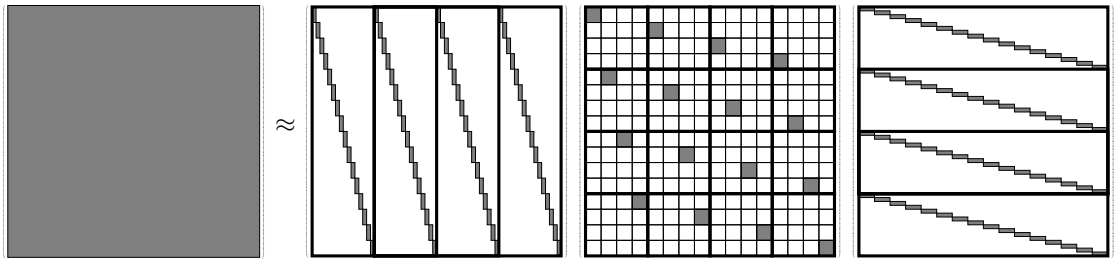


Figure 9: The illustration of an MSCS of a complementary 2D low-rank kernel matrix $K \approx USV$ with quadtrees of depth 3 and 16 leaf nodes in Figure 7. Non-zero blocks in (18)-(20) are shown in gray areas. $\{U_i\}_{1\le i\le 4}$, $\{\bar{S}_{ij}\}_{1\le i\le j\le 4}$, and $\{V_i\}_{1\le i\le 4}$ are visualized by large submatrices with wide edges in the middle left, middle right, and right figures, respectively.

3.5 Recursive MSCS

This section applies MSCS recursively to obtain the full 2D IDBF of a complementary 2D low-rank kernel matrix K.

First, we denote the first level of MSCS of K in (17) as

$$K \approx U^L S^L V^L, \tag{21}$$

where U^L, S^L, V^L maintain the same structures as (18)-(20). Then, the index set r and the column index set c of K could be partitioned into leaves $\{r_i\}_{0\le i\le m-1}$ and $\{c_j\}_{0\le j\le m-1}$ at the leaf level of the row and column quadtrees as (13). In addition, the skeleton index sets $\hat{r}_i \subset r_i$ and $\hat{c}_j \subset c_j$ will be obtained by applying the *rID*s and *cID*s to the construction of (21), and the middle factor S^L will be constructed by the non-zero submatrices S^L_{ij} for all $1 \le i, j \le 4$ as follows:

$$S^L_{ij} = \begin{pmatrix} K(\hat{r}_{(i-1)(m-1)/4+1}, \hat{c}_{(j-1)(m-1)/4+1}) & \cdots & K(\hat{r}_{(i-1)(m-1)/4+1}, \hat{c}_{j(m-1)/4}) \\ \vdots & \ddots & \vdots \\ K(\hat{r}_{i(m-1)/4}, \hat{c}_{(j-1)(m-1)/4+1}) & \cdots & K(\hat{r}_{i(m-1)/4}, \hat{c}_{j(m-1)/4}) \end{pmatrix}. \tag{22}$$

Since S^L_{ij} consists of the important rows and columns of K_{ij} for all $1 \le i, j \le 4$, it will inherit the complementary low-rank property of K_{ij}. Suppose that $T_{X,ij}$ and $T_{\Omega,ij}$ are the quadtrees of the row and column spaces of K_{ij} with $\frac{m}{4}$ leaves and $L-1$ depth. Then, S^L_{ij} has compressible row and column quadtrees $\hat{T}_{X,ij}$ and $\hat{T}_{\Omega,ij}$ with $\frac{m}{16}$ leaves and $L-2$ depth according to Section 3.3.

Next, a recursive MSCS will be applied to each S^L_{ij}. The first step is similar to that of MSCS, we divide each S^L_{ij} into a 4×4 block matrix according to the nodes at the second level of its row and column quadtrees:

$$S^L_{ij} = \begin{pmatrix} (S^L_{ij})_{11} & (S^L_{ij})_{12} & (S^L_{ij})_{13} & (S^L_{ij})_{14} \\ (S^L_{ij})_{21} & (S^L_{ij})_{22} & (S^L_{ij})_{23} & (S^L_{ij})_{24} \\ (S^L_{ij})_{31} & (S^L_{ij})_{32} & (S^L_{ij})_{33} & (S^L_{ij})_{34} \\ (S^L_{ij})_{41} & (S^L_{ij})_{42} & (S^L_{ij})_{43} & (S^L_{ij})_{44} \end{pmatrix}. \tag{23}$$

For each block $(S^L_{ij})_{k\ell}$, the LRCS could be constructed as $(S^L_{ij})_{k\ell} \approx (U^{L-1}_{ij})_{k\ell}(S^{L-1}_{ij})_{k\ell}(V^{L-1}_{ij})_{k\ell}$ for all $1 \le k, \ell \le 4$. After that, the MSCS of S^L_{ij} will be obtained as follows:

$$S^L_{ij} \approx U^{L-1}_{ij} S^{L-1}_{ij} V^{L-1}_{ij}, \tag{24}$$

where $U_{ij}^{L-1}, S_{ij}^{L-1}, V_{ij}^{L-1}$ are constructed by $(U_{ij}^{L-1})_{k\ell}(S_{ij}^{L-1})_{k\ell}(V_{ij}^{L-1})_{k\ell}$ for all $1 \le k, \ell \le 4$ as in (18)-(20).

Eventually, the factorizations in (24) for all $1 \le i, j \le 4$ will be combined to form a factorization of S^L:

$$S^L \approx U^{L-1} S^{L-1} V^{L-1}, \tag{25}$$

where

$$U^{L-1} = \begin{pmatrix} U_1^{L-1} & & & \\ & U_2^{L-1} & & \\ & & U_3^{L-1} & \\ & & & U_4^{L-1} \end{pmatrix} \quad \text{with} \quad U_k^{L-1} = \begin{pmatrix} U_{1k}^{L-1} & & & \\ & U_{2k}^{L-1} & & \\ & & U_{3k}^{L-1} & \\ & & & U_{4k}^{L-1} \end{pmatrix}, \tag{26}$$

$$S^{L-1} = \begin{pmatrix} \bar{S}_{11}^{L-1} & \bar{S}_{12}^{L-1} & \bar{S}_{13}^{L-1} & \bar{S}_{14}^{L-1} \\ \bar{S}_{21}^{L-1} & \bar{S}_{22}^{L-1} & \bar{S}_{23}^{L-1} & \bar{S}_{24}^{L-1} \\ \bar{S}_{31}^{L-1} & \bar{S}_{32}^{L-1} & \bar{S}_{33}^{L-1} & \bar{S}_{34}^{L-1} \\ \bar{S}_{41}^{L-1} & \bar{S}_{42}^{L-1} & \bar{S}_{43}^{L-1} & \bar{S}_{44}^{L-1} \end{pmatrix} \tag{27}$$

with $\bar{S}_{ij}^{L-1}$ as a 4×4 block matrix with the (j, i)-th block as S_{ji}^{L-1},

$$V^{L-1} = \begin{pmatrix} V_1^{L-1} & & & \\ & V_2^{L-1} & & \\ & & V_3^{L-1} & \\ & & & V_4^{L-1} \end{pmatrix} \quad \text{with} \quad V_k^{L-1} = \begin{pmatrix} V_{k1}^{L-1} & & & \\ & V_{k2}^{L-1} & & \\ & & V_{k3}^{L-1} & \\ & & & V_{k4}^{L-1} \end{pmatrix}. \tag{28}$$

Hence, the second level factorization of K could be constructed as follows:

$$K \approx U^L U^{L-1} S^{L-1} V^{L-1} V^L.$$

Comparing (21) and (25), a fractal structure could be found in each level of the middle factor S^L and S^{L-1}. For example, S^L and S^{L-1} have the same structure consisting of 16 submatrices as shown in (19) and (27). Besides, submatrices S_{ij}^{L-1} can factorized into a product of three matrices U_{ij}^{L-2}, S_{ij}^{L-2}, V_{ij}^{L-2} with the same sparsity structure as that of S^L in (25)-(28). Thus, the recursive MSCS could be applied repeatedly to each S^ℓ for $\ell = L, L-1, \ldots, \frac{L}{2}$ and the matrix factors could be assembled hierarchically as follows:

$$K \approx U^L U^{L-1} \cdots U^h S^h V^h \cdots V^{L-1} V^L, \tag{29}$$

where $h = \frac{L}{2}$.

In the ℓ-th recursive MSCS, there are $4^{2(L-\ell+1)}$ dense submatrices with compressible row and column quadtrees, which consist $\frac{m}{4^{2(L-\ell+1)}}$ leaves and depth $L - 2(L - \ell + 1)$, in S^ℓ. Thus, after $h = \frac{L}{2}$ iterations, the recursive MSCS will stop, since there is not any compressible submatrix in S^h. Otherwise, when S^ℓ is still compressible, there are $4^{2(L-\ell+1)} \frac{m}{4^{2(L-\ell+1)}} = \frac{N}{n_0}$ low-rank submatrices to be factorized. Linear IDs only require $O(k^3)$ operations for each low-rank submatrix, and hence at most $O(\frac{k^3}{n_0} N)$ for each level of factorization, and $O(\frac{k^3}{n_0} N \log N)$ for the whole 2D IDBF.

3.6 Extensions

We have introduced the 2D IDBF for a complementary low-rank kernel matrix K in the entire domain $X \times \Omega$. Although we have assumed the uniform grid in X and Ω, the butterfly factorization extends naturally to more general settings. In the case with non-uniform point sets X or Ω, one can still construct a butterfly factorization for K following the same procedure. More specifically, we construct two trees T_X and T_Ω adaptively via hierarchically partitioning the square domains covering X and Ω. For non-uniform point sets X and Ω, the numbers of points in A_i^ℓ and B_j^ℓ are different. If a node does not contain any point inside it, it is simply discarded from the quadtree. We can also extend the 2D IDBF to the 3D case by constructing two octrees T_X and T_Ω via hierarchically partitioning the cube domains covering X and Ω. Lastly, the numerical rank in all low-rank approximations in the IDBF presented is fixed. It's easy to extend the current version to an adaptive one with an adaptive rank k_ϵ in IDs depending on a target accuracy ϵ. For example, choose $k_\epsilon = \min\{k : R_1(k, k) \le \epsilon R_1(1, 1)\}$ and update $k \leftarrow k_\epsilon$ after the QR in IDs. An adaptive rank leads to a more compressed IDBF while a fixed rank results in a more predictable sparsity pattern in IDBF.

4 Numerical results

This section presents several numerical examples to demonstrate the efficiency of the proposed framework. All implementations are in MATLAB® on a server computer with a single thread and 3.2 GHz CPU, and are available in the ButterflyLab (`https://github.com/ButterflyLab/ButterflyLab`). Let $\{g^d(x), x \in X\}$ and $\{g^b(x), x \in X\}$ denote the results given by the direct matrix-vector multiplication and MIDBF, respectively. The accuracy of applying fast algorithms is estimated by the relative error defined as follows:

$$\epsilon^b = \sqrt{\frac{\sum_{x\in S} |g^b(x) - g^d(x)|^2}{\sum_{x\in S} |g^d(x)|^2}}, \tag{30}$$

where S is an index set containing 256 randomly sampled row indices of the kernel matrix K. The error for recovering the kernel function is defined as

$$\epsilon^K = \frac{\|e^{2\pi i\Phi(S,S)} - e^{2\pi iU(S,:)V(:,S)^T}\|_2}{\|e^{2\pi i\Phi(S,S)}\|_2}, \tag{31}$$

where Φ is the phase matrix and UV^T is its low-rank recovery. In all of our examples, the tolerance parameter ϵ is set to 10^{-9}, the over-sampling parameter q in low-rank phase matrix factorization is set to 2, the number of points in a leave node n_0 in the MIDBF is set to 8^d, and the over-sampling parameter t in ID in MIDBF is set to 5. We applied IDs with an adaptive rank and k denotes our empirically estimated rank.

4.1 Accuracy and scaling of low-rank matrix recovery and MIDBF

In this part, we present numerical results of several examples to demonstrate the accuracy and asymptotic scaling of the proposed low-rank matrix recovery for phase functions, and MIDBF. With no loss of generality, we only focus on Scenario 1 of indirect access.

Example 1. Our first example is to evaluate a 2D generalized Radon transform which is a Fourier integral operators (FIO) [36] defined as follows:

$$g(x) = \int_{\mathbb{R}} e^{2\pi i\Phi(x,\xi)} \widehat{f}(\xi) d\xi, \tag{32}$$

where $\widehat{f}$ is the Fourier transform of f, and $\Phi(x,\xi)$ is a phase function given by

$$\begin{gathered}\Phi(x,\xi) = x\cdot\xi' + \sqrt{c_1^2(x)\cdot\xi_1^2 + c_2^2(x)\cdot\xi_2^2}, \\ c_1(x) = (2+\sin(2\pi x_1)\sin(2\pi x_2))/16, \quad \text{and} \quad c_2(x) = (2+\cos(2\pi x_1)\cos(2\pi x_2))/16.\end{gathered} \tag{33}$$

The discretization of (32) is

$$g(x) = \sum_{\xi\in\Omega} e^{2\pi i\Phi(x,\xi)} \widehat{f}(\xi), \quad x \in X, \tag{34}$$

where X and Ω are the sets of $O(N)$ points uniformly distributed in $[0,1)\times[0,1)$. The computation in (34) approximately integrates over spatially varying ellipses, for which $c_1(x)$ and $c_2(x)$ are the axis lengths of the ellipse centered at the point $x \in X$. The corresponding matrix form of (34) is simply

$$u = Kg, \quad K = (e^{2\pi i\Phi(x,\xi)})_{x\in X,\xi\in\Omega}. \tag{35}$$

The framework is applied to recover the phase functions in the form of low-rank matrix factorization, compute the MIDBF of the kernel function, and apply it to a randomly generated f in (32) to obtain g. Table 2 summarizes the results of this example for different grid sizes $N = n^2$ and different rank parameters r, k.

Table 2 shows that the accuracy of the low-rank matrix recovery and the MIDBF stay almost of the same order, though the accuracy becomes slightly worse as the problem size increases. The slightly increasing error is due to the randomness of the proposed algorithm. As the problem size increases, the probability for capturing the low-rank matrix with a fixed rank parameter becomes smaller. Otherwise, when the rank parameter r or k increases, the accuracy of results will increase as well. In Figure 10 (left), we see that the reconstruction time of the phase functions scales nearly linearly, the factorization time and the application time of the MIDBF scale nearly linearly, e.g. when $r = 20$ and $k = 30$.

n, r, k	ϵ^b	ϵ^K	$T_{rec}(min)$	$T_{fac}(min)$	$T_{app}(sec)$	T_d/T_{app}
16, 10, 30	2.04e-06	2.59e-08	1.48e-02	1.63e-02	1.83e-04	2.56e+01
16, 20, 20	4.75e-05	2.04e-09	1.80e-02	1.33e-02	1.73e-04	2.97e+01
16, 20, 30	2.03e-06	2.04e-09	1.69e-02	1.63e-02	1.92e-04	2.46e+01
64, 10, 30	2.34e-07	2.53e-08	1.66e-01	2.61e-01	3.61e-03	2.28e+02
64, 20, 20	9.27e-06	1.84e-09	2.54e-01	2.09e-01	3.11e-03	3.11e+02
64, 20, 30	2.04e-07	1.84e-09	2.02e-01	2.59e-01	3.60e-03	2.28e+02
256, 10, 30	3.47e-08	2.53e-08	2.80e+00	5.22e+00	4.80e-02	4.34e+03
256, 20, 20	5.14e-07	1.91e-09	4.61e+00	4.90e+00	5.97e-02	4.65e+03
256, 20, 30	1.94e-08	1.91e-09	3.47e+00	5.24e+00	5.09e-02	4.09e+03
1024, 10, 30	2.99e-08	2.48e-08	5.63e+01	9.60e+01	8.93e-01	7.79e+04
1024, 20, 20	8.50e-08	1.91e-09	9.56e+01	9.56e+01	9.83e-01	9.08e+04
1024, 20, 30	9.11e-09	1.91e-09	8.32e+01	1.01e+02	8.88e-01	7.86e+04
4096, 10, 30	3.30e-08	2.46e-08	1.16e+03	1.81e+03	1.72e+01	1.69e+06
4096, 20, 20	2.20e-08	1.81e-09	1.58e+03	1.54e+03	1.71e+01	1.83e+06
4096, 20, 30	5.65e-09	1.81e-09	1.57e+03	1.76e+03	1.76e+01	1.66e+06

Table 2: Numerical results for the 2D uniform FIO given in (34). r is the rank parameter of the low-rank approximation of the phase function. k is the rank parameter of the MIDBF. T_{rec} is the time for ordering and recovering the phase functions, T_{fac} is the time for computing the MIDBF, T_{app} is the time for applying the MIDBF, and T_d is the time for a direct summation in (34).

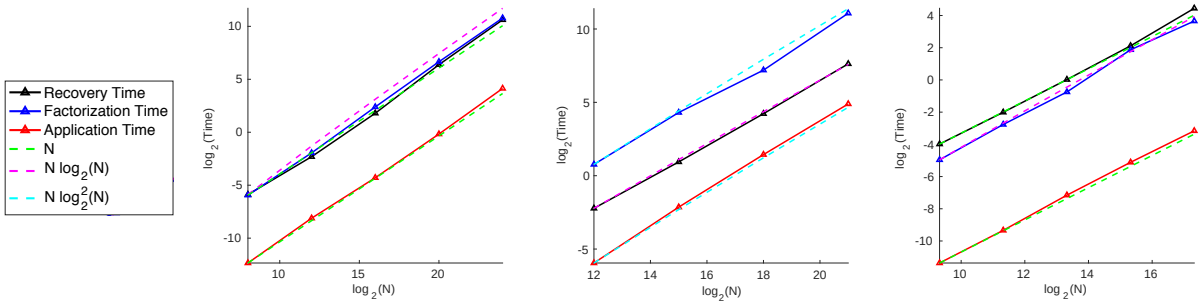


Figure 10: The visualization of the computational complexity. N is the size of the matrix. Left: the 2D uniform FIO given in (34). Middle: the 3D Fourier transform given in (36). Right: the example in (37).

n, r	ϵ^b	ϵ^K	$T_{rec}(min)$	$T_{fac}(min)$	$T_{app}(sec)$	T_d/T_{app}
16, 3	2.15e-01	2.48e-01	1.98e-01	1.45e+00	1.41e-02	6.33e+01
16, 4	3.84e-06	7.02e-15	2.05e-01	1.73e+00	1.61e-02	5.31e+01
16, 5	3.84e-06	4.23e-15	2.14e-01	1.70e+00	1.62e-02	5.30e+01
32, 3	2.94e-01	3.29e-01	1.77e+00	1.52e+01	1.39e-01	4.48e+02
32, 4	1.70e-07	8.33e-15	1.83e+00	2.00e+01	2.10e-01	2.58e+02
32, 5	1.70e-07	7.35e-15	1.92e+00	1.98e+01	2.27e-01	2.42e+02
64, 3	3.49e-01	3.21e-01	1.74e+01	1.18e+02	1.70e+00	2.25e+03
64, 4	2.46e-07	1.84e-14	1.79e+01	1.49e+02	2.80e+00	1.27e+03
64, 5	2.46e-07	1.74e-14	1.87e+01	1.47e+02	2.71e+00	1.30e+03
128, 3	2.60e-01	3.24e-01	1.86e+02	1.45e+03	1.80e+01	2.23e+04
128, 4	8.27e-09	3.37e-14	1.92e+02	2.13e+03	3.14e+01	9.57e+03
128, 5	8.27e-09	2.79e-14	1.97e+02	2.16e+03	2.97e+01	1.02e+04

Table 3: Numerical results for the 3D Fourier transform given in (36). T_{rec} is the time for ordering and recovering the phase functions, T_{fac} is the time for computing the MIDBF, T_{app} is the time for applying the MIDBF, and T_d is the time for a direct summation in (36).

Example 2. In this example, we evaluate a 3D non-uniform Fourier transform:

$$g(x) = \sum_{\xi \in \Omega} e^{2\pi i x^T \xi} \widehat{f}(\xi), \tag{36}$$

where X and Ω are the sets of N points randomly selected in $[0, 1)^3$.

Table 3 summarizes the results of this example for different grid sizes $N = n^3$ and different rank parameters r in the low-rank approximation of the phase function. In the MIDBF, the rank parameter k is 80. The accuracy of the low-rank matrix recovery and the MIDBF stay almost of the same order in Table 3. In Figure 10 (middle), we see that the reconstruction time of the phase function scales in $O(N \log N)$, the factorization time and the application time of the MIDBF scale nearly linearly, e.g., when $r = 5$.

Example 3. The final example is the oscillatory part of the Green's function of a Helmholtz equation [8]:

$$g(x) = \sum_{\xi \in \Omega} e^{2\pi i \Phi(x,\xi)} \widehat{f}(\xi), \quad x \in X, \tag{37}$$

where $\Phi(x, \xi) = h \cdot \|x - \xi\|_2$ and $h = \frac{\sqrt{N}}{6} \sim O(n)$. X and Ω are the sets of N points generated via a triangular mesh to discretize the surface of a unit sphere. The triangular mesh is generated by uniformly refining an icosahedron and projecting the new mesh nodes, which are the old mesh edge center, onto the sphere. The submatrix of the oscillatory part of the Green's function corresponding to one half of the sphere in X and the other half of the sphere in Ω is chosen as the matrix to be reconstructed, factorized, and applied to a random vector.

In this example, rank parameters $r = 40$ and $k = 30$. As shown in Table 4, the accuracy of the low-rank matrix recovery and the MIDBF stay almost of the same order. Figure 10 (right) shows that the reconstruction time of the phase functions scales nearly linearly, the factorization time and the application time of the MIDBF scale nearly linearly.

5 Conclusion

This paper introduced a framework for $O(N \log N)$ evaluation of the multidimensional oscillatory integral transform $g(x) = \int e^{2\pi i \Phi(x,\xi)} f(\xi) d\xi$. In the case of indirect access of the phase functions, this paper proposed a novel fast algorithm for recovering the phase functions in $O(N \log N)$ operations. Second, a new BF, the multidimensional interpolative decomposition butterfly factorization (MIDBF), for multidimensional kernel

N	ϵ^b	ϵ^K	$T_{rec}(min)$	$T_{fac}(min)$	$T_{app}(sec)$	T_d/T_{app}
640	1.02e-08	5.29e-09	6.32e-02	3.21e-02	3.78e-04	1.69e+02
2560	2.26e-08	1.18e-08	2.49e-01	1.47e-01	1.54e-03	3.70e+02
10240	9.26e-08	2.48e-08	1.02e+00	5.95e-01	7.02e-03	1.02e+03
40960	5.01e-08	5.25e-08	4.34e+00	3.66e+00	2.89e-02	3.67e+03
163840	1.24e-07	1.14e-07	2.18e+01	1.26e+01	1.12e-01	1.47e+04

Table 4: Numerical results for the case given in (37). T_{rec} is the time for ordering and recovering the phase functions, T_{fac} is the time for computing the MIDBF, T_{app} is the time for applying the MIDBF, and T_d is the time for a direct summation in (37).

matrices in the form of a low-rank factorization is proposed, and it requires only $O(N \log N)$ operations to evaluate the oscillatory integral transform.

Acknowledgments. H. Y. was partially supported by Grant R-146-000-251-133 in the Department of Mathematics at the National University of Singapore, by the Ministry of Education in Singapore under the grant MOE2018-T2-2-147, and the start-up grant of the Department of Mathematics at Purdue University.

References

[1] G. Bao and W. W. Symes. Computation of pseudo-differential operators. *SIAM Journal on Scientific Computing*, 17(2):416–429, 1996.

[2] J. P. Boyd and F. Xu. Divergence (Runge Phenomenon) for least-squares polynomial approximation on an equispaced grid and Mock Chebyshev subset interpolation. *Applied Mathematics and Computation*, 210(1):158 – 168, 2009.

[3] J. Bremer. An algorithm for the rapid numerical evaluation of Bessel functions of real orders and arguments. *arXiv:1705.07820 [math.NA]*, 2017.

[4] J. Bremer. An algorithm for the numerical evaluation of the associated Legendre functions that runs in time independent of degree and order. *Journal of Computational Physics*, 360:15 – 38, 2018.

[5] K. Buchin and W. Mulzer. Delaunay triangulations in o(sort(n)) time and more. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 139–148, Oct 2009.

[6] E. J. Candès, L. Demanet, and L. Ying. A fast butterfly algorithm for the computation of Fourier integral operators. *Multiscale Modeling and Simulation*, 7(4):1727–1750, 2009.

[7] M. Costantin, A. Farina, and F. Zirilli. A fast phase unwrapping algorithm for sar interferometry. *IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING*, 37(1), 1999.

[8] B. Davies. *Green's Functions*, pages 163–179. Springer New York, New York, NY, 2002.

[9] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Delaunay Triangulations*, pages 191–218. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[10] L. Demanet and L. Ying. Fast wave computation via Fourier integral operators. *Math. Comput.*, 81(279), 2012.

[11] B. Engquist and L. Ying. A fast directional algorithm for high frequency acoustic scattering in two dimensions. *Communications in Mathematical Sciences*, 7(2):327–345, 06 2009.

[12] L. Greengard and J.-Y. Lee. Accelerating the Nonuniform Fast Fourier Transform. *SIAM Review*, 46(3):443–454, 2004.

[13] H. Guo, Y. Liu, J. Hu, and E. Michielssen. A butterfly-based direct integral-equation solver using hierarchical LU factorization for analyzing scattering from electrically large conducting objects. *IEEE Transactions on Antennas and Propagation*, 65(9):4742–4750, Sept 2017.

[14] 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.

[15] P. Hoffman and K. Reddy. Numerical differentiation by high order interpolation. *SIAM Journal on Scientific and Statistical Computing*, 8(6):979–987, 1987.

[16] H. Isozaki and J. L. Rousseau. Pseudodifferential multi-product representation of the solution operator of a parabolic equation. *Communications in Partial Differential Equations*, 34(7):625–655, 2009.

[17] L. Jianchun, G. A. Pope, and K. Sepehrnoori. A high-resolution finite-difference scheme for nonuniform grids. *Applied Mathematical Modelling*, 19(3):162 – 172, 1995.

[18] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, Sep. 1961.

[19] Y. Li and H. Yang. Interpolative butterfly factorization. *SIAM Journal on Scientific Computing*, 39(2):A503–A531, 2017.

[20] Y. Li, H. Yang, E. R. Martin, K. L. Ho, and L. Ying. Butterfly Factorization. *Multiscale Modeling & Simulation*, 13(2):714–732, 2015.

[21] Y. Li, H. Yang, and L. Ying. Multidimensional butterfly factorization. *Applied and Computational Harmonic Analysis*, 2017.

[22] Y. Liu, H. Guo, and E. Michielssen. An HSS matrix-inspired butterfly-based direct solver for analyzing scattering from two-dimensional objects. *IEEE Antennas and Wireless Propagation Letters*, 16:1179–1183, 2017.

[23] S. Lo. Parallel delaunay triangulation in three dimensions. *Computer Methods in Applied Mechanics and Engineering*, 237-240:88 – 106, 2012.

[24] E. Michielssen and A. Boag. A multilevel matrix decomposition algorithm for analyzing scattering from large structures. *Antennas and Propagation, IEEE Transactions on*, 44(8):1086–1093, Aug 1996.

[25] G. Nico, G. Palubinskas, and M. Datcu. Bayesian approaches to phase unwrapping: Theoretical study. *IEEE TRANSACTIONS ON SIGNAL PROCESSING*, 48(9), 2000.

[26] M. O'Neil, F. Woolfe, and V. Rokhlin. An algorithm for the rapid evaluation of special function transforms. *Appl. Comput. Harmon. Anal.*, 28(2):203–226, 2010.

[27] Q. Pang, K. L. Ho, and H. Yang. Interpolative decomposition butterfly factorization. *arXiv:1809.10573 [math.NA]*, 2018.

[28] R. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 11 1957.

[29] J. L. Rousseau. Fourier-integral-operator approximation of solutions to first-order hyperbolic pseudodifferential equations I: Convergence in sobolev spaces. *Communications in Partial Differential Equations*, 31(6):867–906, 2006.

[30] J. L. Rousseau and G. Hörmann. Fourier-integral-operator approximation of solutions to first-order hyperbolic pseudodifferential equations II: Microlocal analysis. *Journal de Mathmatiques Pures et Appliquées*, 86(5):403 – 426, 2006.

[31] D. Ruiz-Antoln and A. Townsend. A nonuniform fast fourier transform based on low rank approximation. *SIAM Journal on Scientific Computing*, 40(1):A529–A547, 2018.

[32] M. Smid. The well-separated pair decomposition and its applications. In *Handbook of Approximation Algorithms and Metaheuristics*, 2007.

[33] E. Trouvé, J.-M. Nicolas, and H. Maître. Improving phase unwrapping techniques by the use of local frequency estimates. *IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING*, 36(6), 1998.

[34] C. Van Loan. *Computational Frameworks for the Fast Fourier Transform.* Society for Industrial and Applied Mathematics, 1992.

[35] O. V. Vasilyev. High order finite difference schemes on non-uniform meshes with good conservation properties. *Journal of Computational Physics*, 157(2):746 – 761, 2000.

[36] H. Yang. A unified framework for oscillatory integral transforms: When to use NUFFT or butterfly factorization? *Journal of Computational Physics*, 388:103–122, Jul 2019.