Structured Block Basis Factorization for Scalable Kernel Matrix Evaluation

Ruoxi Wang * Yingzhou Li * Michael W. Mahoney † Eric Darve ‡

Abstract

Kernel matrices are popular in machine learning and scientific computing, but they are limited by their quadratic complexity in both construction and storage. It is well-known that as one varies the kernel parameter, *e.g.*, the width parameter in radial basis function kernels, the kernel matrix changes from a smooth low-rank kernel to a diagonally-dominant and then fully-diagonal kernel. Low-rank approximation methods have been widely-studied, mostly in the first case, to reduce the memory storage and the cost of computing matrix-vector products. Here, we use ideas from scientific computing to propose an extension of these methods to situations where the matrix is not well-approximated by a low-rank matrix. In particular, we construct an efficient block low-rank approximation method—which we call the Block Basis Factorization—and we show that it has $\mathcal{O}(n)$ complexity in both time and memory. Our method works for a wide range of kernel parameters, extending the domain of applicability of low-rank approximation methods, and our empirical results demonstrate the stability (small standard deviation in error) and superiority over current state-of-art kernel approximation algorithms.

Keywords. kernel matrix, low-rank approximation, data-sparse representation, machine learning, high-dimensional data

1 Introduction

Kernel methods play an important role in a variety of applications in machine learning and scientific computing. They implicitly map any set of data to a high-dimensional feature space via a kernel function. Given n data points $x_1, \ldots, x_n \in \mathbb{R}^d$ and a kernel function $\mathcal{K} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, the corresponding kernel matrix $K \in \mathbb{R}^{n \times n}$ is defined as $K_{i,j} = \mathcal{K}(x_i, x_j)$. A limitation of these methods lies in their lack of scalability: $\mathcal{O}(n^2 d)$ time is required for their computation; $\mathcal{O}(n^2)$ time is required for a matrix-vector multiplication; and $\mathcal{O}(n^2)$ time is required even to write down the full kernel. This quadratic cost is prohibitive in many data-intensive applications.

In machine learning, it is popular to try to avoid this problem by considering low-rank matrix approximations such as those provided by the Nyström method [7, 10, 2, 1]. Often, however, the matrices are not well-approximated by a low-rank matrix. In scientific computing, one way to deal with this latter situation is to consider structured matrix factorizations, where one represents exactly or approximately an $n \times n$ full-rank matrix in such a way that it can be applied to an arbitrary vector in $\mathcal{O}(n)$ or $\mathcal{O}(n \log(n))$ time. While low-rank approximations do this, so do many other classes of structured approximations. Perhaps the most well-known of this class of methods is the fast multipole method (FMM) [13].

*Institute for Computational and Mathematical Engineering, Stanford University, Email: {ruoxi, ryanli}@stanford.edu

†International Computer Science Institute and Department of Statistics, University of California, Berkeley, Email: mmahoney@stat.berkeley.edu

‡Department of Mechanical Engineering, Stanford University, Email: edarve@stanford.edu

In this paper, we adopt some of the ideas from the FMM to obtain improved kernel approximations in machine learning. In particular, we propose a novel randomized procedure to compute a block low-rank approximation. Our method does not form the entire matrix, and—since it forms the approximation in a block-wise low-rank fashion—it generalizes low-rank methods that have been used recently in machine learning. (Informally, it efficiently captures both the "near-field" effects, *i.e.*, interactions between data points that are near each other in their original representation, as well as the "far-field" kernel interactions, *i.e.*, between data points that are far from each other in their original representation.)

Creating a single scheme that by design remains efficient for a wide range of kernel parameters is particularly important in light of recent results demonstrating the dependence of kernel matrix properties on the kernel parameter. For example, spectral properties such as spectral decay and leverage score uniformity (*i.e.*, coherence) depend strongly on kernel parameters such as the radial basis function width parameter and the degree of sparsity, and in many cases of interest these matrices are not particularly well approximated by low-rank matrices [10]. This is the case for the Gaussian kernel matrices $\exp(-\|x-y\|_2^2/h^2)$, which go from low-rank to diagonal as one changes h from very large to very small; while a traditional low-rank approximation will fail abruptly in the small-h regime, our algorithm is able to better represent the diagonal part, and thus better approximate the entire matrix.

In the remainder of this introduction, we will first (in Section 1.1) summarize our main results; and we will then (Section 1.2) describe related recent work.

1.1 Main Contributions

In this paper, we provide an accurate kernel matrix approximation algorithm that has a more consistent error (*i.e.*, lower standard deviation) than existing methods, that efficiently captures both near-field and far-field interactions between data points, and that has linear, *i.e.*, $\mathcal{O}(n)$, complexity. In more detail, here are our main contributions.

- We present a novel matrix format called the Block Basis Factorization (BBF) for machine learning applications. This is an efficient data-sparse representation of the kernel matrix with $\mathcal{O}(n)$ memory (see Section 2). This matrix format is efficient for a wide range of kernel parameters.
- We propose an $\mathcal{O}(n)$ algorithm to construct the BBF given a desired level of accuracy (see Section 3). Among other things, we provide a heuristic technique to compute near-optimal parameters for the method.
- We provide an empirical evaluation of our method on both synthetic and real datasets (see Section 4). In particular, this demonstrates the superiority of our algorithm over state-of-art low-rank methods and recent block factorizations.

Our algorithm first clusters the data into k distinct groups, and it permutes the matrix according to these clusters. Next, it computes the column basis (resp. row basis) for each row-submatrix (the interaction between one cluster and all data points) via a subsampling procedure and a randomized algorithm. Then, for every block, it uses the corresponding column and row basis to compress the block, *i.e.*, the "inner" block, also using a random sub-sampling algorithm. Consequently, our method computes a block-wise low-rank approximation for the k^2 blocks using a set of only k bases for a symmetric kernel matrix ($2k$ for a non-symmetric matrix). The resulting framework gives a rank-(rk) approximation with $\mathcal{O}(nr + (rk)^2)$ storage. This should be contrasted with a low-rank scheme that gives a rank-r approximation using $\mathcal{O}(nr)$ memory. For the latter case, r is typically much larger (for the same level of accuracy), making our BBF computationally more efficient. Note that, in particular, our algorithm takes as input the original feature vectors; and thus, by design, it can run without forming the full kernel matrix.

1.2 Related Research

There is a large body of research in scientific computing that aims to accelerate kernel methods. The FMM partitions points into different boxes and efficiently approximates interactions between boxes that are far apart. From a linear algebraic perspective, this involves using a low-rank approximation for off-diagonal blocks. The FMM was originally used for solving kernel summation of potential fields, and it then was extended to the solution of Helmholtz [5, 6] and Maxwell's [5, 4] equations. The FMM and similar methods work well for dimension up to three, but the complexity grows exponentially with the dimension, and thus for higher dimensions they fail due to lack of scalability. The Improved Fast Gauss Transform (IFGT) was proposed to alleviate the dimensionality issue for the Gaussian kernel [22]. It is a faster version of the Fast Gauss Transform (FGT) [14]. The FGT offers an $\mathcal{O}(m+n)$ algorithm for matrix-vector multiplications with Gaussian-kernel matrices of size m by n. The IFGT further reduces the growth rate of the constant factor from exponential to asymptotically polynomial order with respect to dimensionality d (the cost is $\mathcal{O}(d^p)$ for $d \to \infty$ and a moderate p). However when d increases, it still requires a very large number of points n to start exhibiting a cost with linear growth.

Another popular approach uses direct low-rank matrix factorizations: given a matrix $K \in \mathbb{R}^{n\times n}$, find a rank-$r$ approximation of K via two tall and skinny matrices $U, V \in \mathbb{R}^{n\times r}$: $K \approx UV^T$ [11]. The singular value decomposition (SVD) is the "gold-standard" method of low-rank factorization: it achieves the lowest reconstruction error, but has a cubic factorization cost; even computing the best rank-r approximation takes at least $\mathcal{O}(n^2 r)$ time with traditional methods. Recent work in Randomized Numerical Linear Algebra (RandNLA) has focused on using randomized matrix algorithms to speed up low-rank matrix approximations [17]. For example, one RandNLA algorithm is the randomized SVD algorithm of Halko et al. [15] that computes a low-dimensional approximation of the range space of K. Relatedly, Liberty et al. [16] and Sarlos [19] do a random projection of the matrix onto a p-dimensional ($p \gtrsim r$) subspace. These methods have an improved $\mathcal{O}(n^2 \log r)$ complexity, but they still take the matrix K as input and thus need to form the entire matrix.

A related algorithm is the Nyström method [7, 10], which randomly subsamples the columns of the matrix and uses them to construct a low-rank approximation. A naïve Nyström algorithm uses a uniform sampling, which is computationally inexpensive, but which works well only when the matrix has uniform leverage scores, *i.e.*, low coherence. Improved versions of Nyström have been proposed to provide more sophisticated ways of sampling the columns: for example, one can sample with probabilities proportional to the statistical leverage scores—which can be approximated with the algorithm of Drineas et al. [8] in essentially random projection time [10]—to obtain very strong results; or one can use the algorithm of Alaoui and Mahoney [1] to compute a variant of the leverage scores without even forming the entire matrix, thereby obtaining improved statistical bounds. The random sampling algorithm of Engquist et al. [9] gives an $\mathcal{O}(r^2(m+n))$ method to choose important rows/columns alternatively using the QR factorization. K-means Nyström [23] uses k-means clustering to choose the landmark points instead of sampling real points. A related line of research includes random feature maps: for example, Random Kitchen Sinks (RKS) [18] proposed a random feature map using the Fourier transform of the kernel function to map data to an Euclidean inner-product space.

There have also been several approaches to try to address problems that are not well-approximated by low-rank matrices. In particular, the idea of combining the low-rank approximation with a preclustering of the data points has been recently used in machine learning. For example, the clustered Low-Rank Approximation (CLRA) [20] performs a block-wise low-rank approximation of the kernel matrix, but it requires forming the entire matrix and has only been applied to social network data; and the Memory Efficient Kernel Approximation (MEKA) [21] improved upon CLRA by avoiding the formation of the entire matrix. (While interesting, we found that, in our benchmarks, this method is not stable, and produces errors with significant variations, *e.g.*, a large standard deviation over multiple trials.)

We briefly discuss the main differences between our algorithm and MEKA. We compute the low-rank basis vectors from a larger space, that is, from the entire row-submatrices instead of the diagonal blocks (resulting in a more accurate basis). We use a sampling procedure that is more sophisticated than uniform sampling, but still has linear cost. We provide a parameter selection method that computes the near-optimal number of clusters k, as well as the ranks for each cluster, given a desired accuracy. Therefore, for a given memory cost (which can be closely related to the computational cost of a fast matrix-vector product), our algorithm captures more information from the matrix while still keeping linear complexity.

2 Block Basis Factorization (BBF) of Matrix

In this section, we define the Block Basis Factorization (BBF) of a matrix. Given a matrix $M \in \mathbb{R}^{n\times n}$ partitioned into k by k blocks, let $M_{i,j}$ denote the (i,j)-th block for $i = 1,\ldots,k$. Then the BBF of M is defined as:

$$M = \widetilde{U}\,\widetilde{C}\,\widetilde{V}^T \tag{1}$$

where $\widetilde{U}$ is a block diagonal matrix with the i-th diagonal block U_i being the column basis of $M_{i,j}$ for all j, $\widetilde{V}$ is a block diagonal matrix with the j-th diagonal block V_j being the row basis of $M_{i,j}$ for all i, and $\widetilde{C}$ is a k by k block matrix with the (i,j)-th block denoted by $C_{i,j} = U_i^T M_{i,j} V_j$. In practice, $C_{i,j}$ is an approximation of $U_i^T M_{i,j} V_j$ up to a desired accuracy ϵ.

Based on the definition of the BBF, we have that *U_i is the column basis of $M_{i,:}$* and *V_j is the row basis of $M_{:,j}$*, where

$$\begin{aligned} M_{i,:} &= \begin{bmatrix} M_{i,1} & \cdots & M_{i,k} \end{bmatrix}, \\ \text{and } M_{:,j} &= \begin{bmatrix} M_{1,j}^T & \cdots & M_{k,j}^T \end{bmatrix}^T \end{aligned} \tag{2}$$

are the i-th row-submatrix and j-th column-submatrix, respectively. Hence, the width of U_i is the numerical rank of $M_{i,:}$ for a desired accuracy ϵ, and the width of V_j is the numerical rank of $M_{:,j}$. If the numerical ranks of all row- and column- submatrices are bounded by r, then the memory cost for the BBF is given by $\mathcal{O}(nr + (rk)^2)$. Further, if $k \leq \sqrt{n}$ and r is a constant independent of n, then the BBF gives a data-sparse representation of M with linear memory (see Figure 1). In this case, the complexity for both storing the BBF and applying it to a vector will be linear.

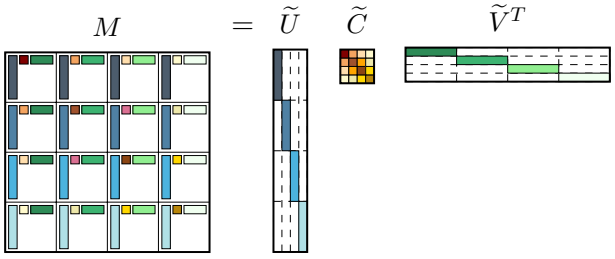


Figure 1: $M = \widetilde{U}\,\widetilde{C}\,\widetilde{V}^T$

3 Fast Algorithm for BBF

In this section, we provide a fast algorithm, with linear complexity, to obtain a BBF. We state our main theorem and the basic idea of our algorithm in Section 3.1. In Section 3.2 and Section 3.3 we give details about BBF construction and parameter selection, respectively. We consider the kernel matrix to be symmetric in this paper; the non-symmetric case can be derived in a similar fashion.

3.1 Main Theorem and Algorithm

We consider n data points $\mathcal{X} = \{x_i\}_{i=1}^n$, $x_i \in \mathbb{R}^d$, partitioned into k clusters $\mathcal{C} = \{\mathcal{C}_i\}_{i=1}^k$, and a shift-invariant kernel function $\mathcal{K} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$. saassWe start by stating a theorem which gives a bound on the reconstruction error of the best rank-r approximation of the interaction matrix $\mathcal{K}(\mathcal{C}_i, :)$.

Theorem 3.1. *Assume that the kernel function $\mathcal{K}$ is a shift-invariant kernel, i.e., $\mathcal{K}(\mathbf{x}, \mathbf{y}) = f(t(\mathbf{x} - \mathbf{y})) = g_{\mathbf{x}-\mathbf{y}}(t)$ where $g_{\mathbf{x}-\mathbf{y}}(t)$ is Lipschitz continuous with the Lipschitz constant κ. Given a k-cluster partition $\mathcal{C} = \{\mathcal{C}_i\}_{i=1}^k$ of the point set $\mathcal{X} = \{x_i\}_{i=1}^n, x_i \in \mathbb{R}^d$, if the radii of all clusters are bounded by a constant R independent of n and d, then, for any cluster $\mathcal{C}_i$,*

$$\|K(\mathcal{C}_i, :) - K_r(\mathcal{C}_i, :)\|_F \leq 4\kappa r^{-1/d}\sqrt{|\mathcal{C}_i|n}R$$

where $K_r(\mathcal{C}_i, :)$ is the best rank-r approximation to $K(\mathcal{C}_i, :)$.

Similarly, we also have

$$\|K(:, \mathcal{C}_i) - K_r(:, \mathcal{C}_i)\|_F \leq 4\kappa r^{-1/d}\sqrt{|\mathcal{C}_i|n}R.$$

The proof of Theorem 3.1 follows a similar strategy of Section 6.1 in Si et al. [21]. The error bound suggests that given r and d, a small radius R is preferred to lower this bound. To reduce R, we use k-means / k-center algorithm (see Appendix A) as the space partitioning method. They differ in the objective functions and thus give slightly different clusters. Empirically we find that neither of them is superior to the other for all datasets; k-means tends to cluster points more evenly while k-center tends to cluster points into compact groups.

Next, we state the basic idea of our algorithm. Suppose we are already given the partitions for data points, as well as the rank r_i used for cluster $\mathcal{C}_i$. We first permute the matrix according to the clusters, such that every block of the matrix represents the interaction between two clusters. The permuted matrix is in Equation 3.

$$M = PKP^T = \begin{array}{c} \\ \mathcal{C}_1 \\ \mathcal{C}_2 \\ \vdots \\ \mathcal{C}_k \end{array} \begin{array}{c} \begin{array}{cccc} \mathcal{C}_1 & \mathcal{C}_2 & \cdots & \mathcal{C}_k \end{array} \\ \begin{pmatrix} M_{1,1} & M_{1,2} & \cdots & M_{1,k} \\ M_{2,1} & M_{2,2} & \cdots & M_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ M_{k,1} & M_{k,2} & \cdots & M_{k,k} \end{pmatrix} \end{array} \tag{3}$$

where P is a permutation matrix, $M_{i,j}$ is the interaction matrix between cluster $\mathcal{C}_i$ and cluster $\mathcal{C}_j$. Then to get a BBF, we need to compute a set of k bases and use them to compute the inner matrices for all the blocks.

1. *Compute basis.* The most accurate way for computing the column basis U_i for the row-submatrix $M_{i,:}$ is to apply an SVD on $M_{i,:}$. But this is too slow, so we restrict ourselves to subsampled indices, *i.e.*, we sample from *all the columns* (denoted as Π), and apply the randomized SVD in Halko et al. [15] (see Appendix B) on the subsampled matrix $M_{i,\Pi}$ to obtain U_i.

2. *Compute inner matrices.* For block $M_{i,j}$, a matrix $C_{i,j}$ that can best approximate $M_{i,j}$ in the form of $U_i C_{i,j} U_j^T$ is given by $(U_i)^\dagger M_{i,j} (U_j^T)^\dagger$, where $(\cdot)^\dagger$ denotes the pseudo-inverse. However, forming each block M_{ij} will result in constructing the entire matrix, this $\mathcal{O}(n^2 d)$ cost renders the method impractical for massive datasets. To reduce the cost, we again restrict ourselves to subsampled indices. We sample columns $\mathcal{J}$ and rows $\mathcal{I}$ from $M_{i,j}$, and compute $C_{i,j}$ in the form of $(U_i(\mathcal{I}, :))^\dagger M_{i,j}(\mathcal{I}, \mathcal{J})(U_j(\mathcal{J}, :)^T)^\dagger$. We can further reduce the cost by skipping the computation for off-diagonal blocks with small entries.

   Our fast algorithm for kernel matrix approximation is presented in Algo. 1.

Algorithm 1 (Fast factorization for BBF)

Input: the number of clusters k, clusters $\mathcal{C}_i$, near-optimal rank r_i for $\mathcal{C}_i$ with $i = 1, \ldots, k$
Output:: $\widetilde{U}$ and $\widetilde{C}$
for $i = 1, \ldots, k$ **do**
  Sample $r_i + l$ columns (denoted Π) from $M_{i,:}$ using the randomized sampling algorithm in Appendix C
  Construct $M_{i,\Pi}$
  Apply a randomized SVD on $M_{i,\Pi}$ to compute U_i
end for
for $\mathcal{C}_i, \mathcal{C}_j$ $(i, j = 1, \ldots, k)$ **do**
  if cutoff criterion is satisfied **then**
    $C_{i,j} = 0$
  else
    Sample r_i rows Γ_r, r_j columns Γ_c uniformly at random, and let $\mathcal{I} = \Pi_r \cup \Gamma_r$, $\mathcal{J} = \Pi_c \cup \Gamma_c$, where Π_r/Π_c are the important rows/columns computed in step 4.
    $C_{i,j} = (U_i(\mathcal{I}, :))^\dagger M_{i,j}(\mathcal{I}, \mathcal{J})(U_j(\mathcal{J}, :)^T)^\dagger$
  end if
end for

3.2 BBF Construction

3.2.1 Fast computation of basis

In order to achieve linear complexity in computing the bases, we consider restricting ourselves to a subset of the dataset $\mathcal{X}$. This is to say, when computing $U_i \in \mathbb{R}^{n_i \times r_i}$, instead of working on the entire row-submatrix $M_{i,:} \in \mathbb{R}^{n_i \times n}$, we can work on a carefully selected set of sub-columns of $M_{i,:}$. These sub-columns will ideally preserve important information about the basis.

The procedure is then to sample from columns and compress the sampled matrix. We first sample $r_i + l$ (l is an over sampling parameter) columns Π from $M_{i,:}$ using the randomized sampling algorithm in Engquist et al. [9] (see Appendix C). It samples r columns by alternating between important columns given rows and important rows given columns, and has $\mathcal{O}(r^2(m+n))$ operations for a matrix of size m by n. After we have found the submatrix $M_{i,\Pi}$, we can then apply the randomized SVD on $M_{i,\Pi}$ to get U_i. With the sampling step, we avoid the construction of the entire matrix M.

3.2.2 Fast computation of inner matrices

For the interaction matrix $M_{i,j}$ between cluster $\mathcal{C}_i$ and $\mathcal{C}_j$, we seek a matrix $C_{i,j}$ that can best approximate $M_{i,j}$ in the form of $U_i C_{i,j} U_j^T$. Since for most machine learning applications, the Frobenius norm is often used, we chose to find a low-rank approximation $U_i C_{i,j} U_j^T$ that captures most of the Frobenius norm of $M_{i,j}$. This is equivalent to solving the following optimization problem:

$$\min_{C_{i,j}} \|M_{i,j} - U_i C_{i,j} U_j^T\|_F$$

The minimizer is given by $C_{i,j} = (U_i)^\dagger M_{i,j} (U_j^T)^\dagger$, where $\dagger$ denotes the pseudo-inverse.

Sample columns and rows. To avoid the quadratic cost in forming all blocks, we again restrict ourselves to subsampled indices.

Proposition 3.2. *If a matrix $M \in \mathbb{R}^{m \times n}$ can be written as $M = UCV^T$, where $U \in \mathbb{R}^{m \times r_1}$ and $V \in \mathbb{R}^{n \times r_2}$ have full column rank, then we subsample $l_1 > r_1$ rows from M, denoted as $\mathcal{I}$, $l_2 > r_2$ columns from*

M, *denoted as* $\mathcal{J}$. *If both* $U(\mathcal{I},:)$ *and* $V(\mathcal{J},:)$ *are full rank, then we have*

$$C = (U(\mathcal{I},:))^{\dagger}\ M(\mathcal{I},\mathcal{J})\ (V(\mathcal{J},:)^T)^{\dagger}$$

The proof can be found in Appendix D.

This suggests that to get $C_{i,j}$, instead of constructing $M_{i,j}$, we can sample columns and rows of $M_{i,j}$ and work on a smaller sub-space. That is, we sample rows $\mathcal{I}$ and columns $\mathcal{J}$ from $M_{i,j}$, and then compute the inner matrix by the form stated in Proposition 3.2. In fact, there is a way to likely satisfy the full rank requirement of $U(\mathcal{I},:)$ and $V(\mathcal{J},:)$ without extra cost. We can choose $\mathcal{I} = \Pi_r \cup \Gamma_r$, where Π_r is the set of important rows computed when sampling columns for computing U_i, and Γ_r is a set of uniformly sampled indices. The same method applies when choosing $\mathcal{J}$.

Skip computation for blocks with small entries. If an accuracy ϵ is desired for the approximation, we can skip the computation for blocks whose entries are sufficiently small. For two clusters that are far apart, the entries can become very small for a range of kernel parameters, in which case the computation of the inner matrix can be skipped and both memory and time savings can be gained.

Low-rank on inner matrices. When the rank of an inner matrix $C_{i,j} \in \mathbb{R}^{n_i \times n_j}$ is smaller than half of $\min\{n_i, n_j\}$, we can represent $C_{i,j}$ as a low-rank factorization. The improvement in memory will be significant when the number of clusters k is large. We can see this by noting that the storage of the BBF is $\sum_{i=1}^{k} n_i r_i + (\sum_{i=1}^{k} r_i)^2$, where the second term comes from inner matrices. When k is large, the second term is usually large. Thus if the memory of each inner matrix is reduced by half, we can reduce the second term by half. Empirically, we found that k is usually small for our test cases, therefore we did not see a significant gain by applying this step. Hence we will keep this as a theoretical interest but skip this step in our algorithm.

3.3 Optimal Parameter Selection

In this section we will provide a heuristic method to produce near-optimal parameters for BBF. We take n points $\mathcal{X}$, and a desired accuracy ϵ as input, and will compute the optimal number of clusters k, the index set for different clusters $\mathcal{I}$, and the estimated rank r_i for the column basis of row i. Our goal is to find the parameters that can satisfy the desired accuracy while achieving the lowest memory cost.

Compute near-optimal ranks. The diagonal block is the interaction of a cluster with itself, while the off-diagonal blocks are the interactions between clusters. It is natural to expect the rank of the off-diagonal blocks to be smaller than that of diagonal blocks. Therefore to achieve ϵ accuracy for all blocks, we only need to achieve that for all the diagonal blocks. If for each block $M_{i,j} \in \mathbb{R}^{n_i \times n_j}$, we have $\frac{\|\hat{M}_{i,j} - M_{i,j}\|_F^2}{\|M\|_F^2} \leq \frac{n_i n_j}{n^2}\epsilon^2$, then the reconstruction error will be bounded by ϵ:

$$\frac{\|\hat{M} - M\|_F^2}{\|M\|_F^2} \leq \frac{\sum_{i,j} n_i n_j}{n^2}\epsilon^2 = \epsilon^2$$

where n is the number of total data points. Therefore, the optimal rank r_i is given by $\min\{k \mid \sum_{j=k+1}^{n_i} \sigma_j^2 < \frac{n_i n_j}{n^2}\|M\|_F^2\ \epsilon^2\}$, where $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{n_i}$ are singular values for $M_{i,i}$.

Compute near-optimal number of clusters k**.** Our next goal is to look for k that achieves the lowest memory. We proceed by solving the following optimization problem:

$$\text{minimize}\ \ f(k) = \sum_{i=1}^{k} n_i r_i + (\sum_{i=1}^{k} r_i)^2,$$

$$\text{s.t.}\ \ r_i = \min\{k \mid \sum_{j=k+1}^{n_i} \sigma_j^2 < \frac{n_i n_j}{n^2}\|M\|_F^2\ \epsilon^2\},\ \ \forall i$$

where $f(k)$ is the memory requirement of the BBF when using k clusters. We observe that $f(k)$ is approximately convex in the domain $[1, \mathcal{O}(\sqrt{n})]$, which enables us to use an $\mathcal{O}(\log n)$ search algorithm for the minimizer.

3.4 Complexity Analysis

To simplify the analysis and ensure that the factorization and application time are linear, we further require the numerical ranks of $M_{i,:}$ $(i = 1, \ldots, k)$ to be bounded by $r_{\max}$, where $r_{\max}$ is a small constant independent of n. When we discuss the numerical rank of a matrix $M_{i,j} \in \mathbb{R}^{n_i \times n_j}$, we mean the smallest integer k such that

$$\sum_{j=k+1}^{\min\{n_i,n_j\}} \sigma_j^2 < \frac{n_i n_j}{n^2} \|M\|_F^2 \, \epsilon^2$$

where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min\{n_i,n_j\}} \geq 0$ are the singular values of $M_{i,j}$, and ϵ stands for the desired accuracy. The complexity for each step can be found in Table 1. The application refers to performing a matrix-vector multiplication. Note that when computing the inner matrices, k can reach up to $\mathcal{O}(\sqrt{n})$ while still maintaining a linear complexity for this step.

Table 1: Complexity table, where k is the number of clusters, n_i is the number of points in cluster $\mathcal{C}_i$, r_i is the estimated rank for the basis of row i, n is the total number of points, $r_{\max}$ is a small constant (independent of n).

Parameter selection		Factorization		Application
Each block	$3n_i r_{\max}^2$	Basis	$\mathcal{O}(nkr_{\max}^2)$	$\mathcal{O}(nr_{\max})$
Compute $f(k)$	$\mathcal{O}(nr_{\max}^2)$	Inner matrix	$\sum_{i=1}^{k}\sum_{j=1}^{k} 4r_i^2 r_j + 2r_i r_j^2$	
Total	$\mathcal{O}(r_{\max}^2 n \log n)$	Total	$\mathcal{O}(nkr_{\max}^2)$	

4 Experimental Results

In this section, we evaluate the performance of our proposed algorithm on synthetic and real datasets[1] listed in Table 2. We use BBF to denote our algorithm, which includes constructing the BBF with parameters computed from Section 3.3, and applying the BBF to a vector. We benchmark our method against other state-of-art kernel approximation methods. All experiments were run on a computer with 2.4 GHz CPU and 8 GB memory.

Table 2: Real datasets used in our experiments, where n is the number of data points, and d is the dimension for each point.

Dataset	**n**	**d**	**Dataset**	**n**	**d**
Abalone	4,177	8	Wine quality	4,898	11
Pageblock	5,473	11	Census house	22,748	16
Pendigits	10,992	16	Forest covertype	581,012	54

[1] All the datasets are downloaded from UCI repository [3].

4.1 Quality of BBF

We first investigate how well the factorization can approximate the exact kernel matrix as we vary *memory cost* and *kernel parameters*, and how consistently small the error is. The accuracy is computed via the relative error $\frac{\|\hat{K}-K\|_F}{\|K\|_F}$, where $\hat{K}$ is the approximated kernel matrix, K is the exact kernel matrix, and $\|\cdot\|_F$ stands for the Frobenius norm. The kernel functions used in the experiments are the Gaussian kernel $\mathcal{K}(x,y) = \exp(-\|x-y\|^2/h^2)$ and the Laplacian kernel $\mathcal{K}(x,y) = \exp(-\|x-y\|_1/h)$, where x and y are data points, and h is the kernel parameter. We now list the kernel approximation methods we are comparing against; details about parameter selection for each method are given in Appendix E:

– *The standard Nyström* (Nys) [7]
– *K-means Nyström* (kNys) [23]
– *Leverage score Nyström* (lsNys) [8]
– *Memory Efficient Kernel Approximation* (MEKA) [21]
– *Random Kitchen Sinks* (RKS) [18]

In our experiments, we normalized the data so that each dimension (feature) of data has zero mean and standard deviation one.

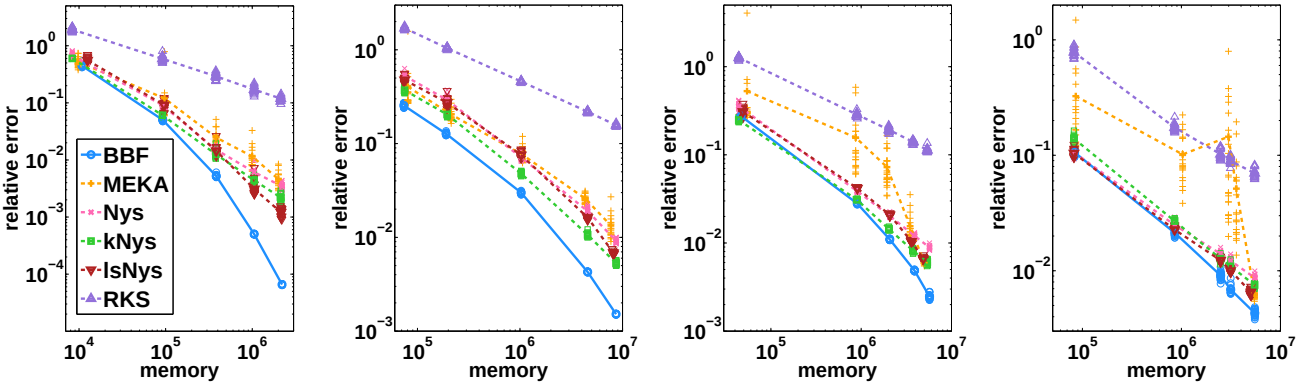
4.1.1 Quality of BBF for various memory costs

We will investigate how our method behaves as more memory is available. Note that the memory cost is a close approximation of the running time for a matrix-vector multiplication. In addition, computing memory is more accurate than running time, which is sensitive to the implementation and algorithmic details. In our experiments, we increase the memory cost by requiring a higher accuracy in BBF, and compute the input parameters accordingly for other methods to ensure all methods have roughly the same memory (the memory for a rank-r low-rank approximation is computed as nr). For each memory cost, the experiment is carried out 20 times to demonstrate the stability of our algorithm with respect to the random sampling errors. The results are shown in Figure 2.

As can be seen from Figure 2, the accuracy of all methods generally improves as memory increases. When memory is low, BBF is comparable with alternative methods; as more memory is available, BBF exhibits a significantly higher accuracy than the other methods. We also observe from the multiple trials that our method has a smaller standard deviation in error.

4.1.2 Quality of BBF for various kernel parameters

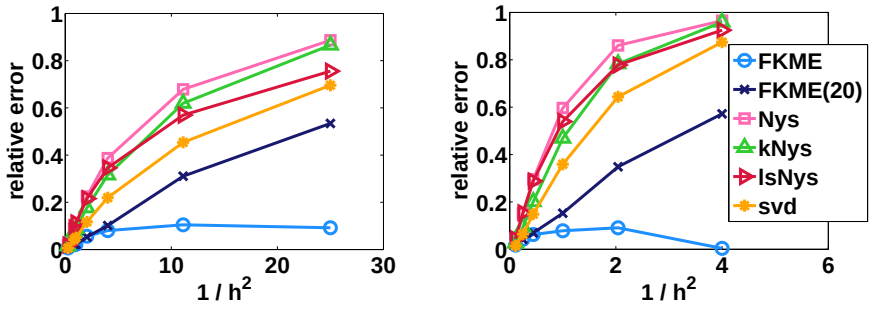
In this section, we will show that when the kernel matrix is not well approximated by low-rank methods, BBF still can approximate the matrix well. We will illustrate this point on two datasets, coupled with similar observations from Gittens and Mahoney [10]. In the experiments, we fix the memory for all methods and all kernel parameters to be roughly the same, and present the average accuracy vs $\frac{1}{h^2}$ in Figure 3. As we can see from Figure 3, when $\frac{1}{h^2}$ gets larger, all low-rank methods (including exact SVD) give worse relative error. The results are explained by the statistics in Table 3. This table shows that the property of kernel matrix changes with the kernel parameter: as the kernel parameter h gets smaller, the kernel matrix becomes less "nice" in terms of rank, that is the kernel matrix goes from low-rank to diagonally dominant. This can be seen from both the *stable rank* $\lceil\frac{\|M\|_F^2}{\|M\|_2^2}\rceil$, an underestimate of the rank of matrix M, and $\frac{\|M_r\|_F^2}{\|M\|_F^2}$, the portion of the Frobenius norm that is preserved in M_r. Therefore, we expect the behavior for all methods (for a given memory) to be worse as the rank of the kernel matrix becomes higher. This is what we see in Figure 3. However, BBF keeps a low error regardless of changes in the matrix property. This is because, as the matrix becomes more diagonal, we can increase the number of clusters k to better approximate the diagonal part, and the off-diagonal blocks can be sparsified due to their small entries. With the flexibility of varying the



2.1 Abalone, h = 2 2.2 Pendigits, h = 3 2.3 Wine quality, h = 3 2.4 Pageblock, h = 4

Figure 2: Comparisons (loglog scale) of BBF (our algorithm) with different state-of-art kernel approximation methods on various real datasets. All the plots have the same legend. For each method, we show the improvement in accuracy as the memory cost is increased. For each memory cost, we present the results of 20 runs of each algorithm. Each run is shown with a symbol, while the lines represent the average accuracy. h is the kernel parameter. The exact memories are 1.7e7, 1.2e8, 2.4e7 and 3.0e7 for subplots from left to right, respectively. For subplots 2.1, 2.2 and 2.3, the Gaussian kernel $\exp(-\|x-y\|_2^2/h^2)$ is used; for subplot 2.4, the Laplacian kernel $\exp(-\|x-y\|_1/h)$ is used. The exact leverage score is used for the leverage score Nyström. The input number of clusters k for MEKA was set to 5. Other values of k for MEKA were also tested, but they did not give better results. The results for MEKA (orange crosses) spread out from the mean (orange dotted line), indicating a large standard deviation in the error. Note that the speed-up depends on the choice of h. A smaller value of h increases the speed-up of BBF even further.

number of clusters, BBF can efficiently use the memory and is favorable in all cases, from low-rank to diagonal. Combined with the results from Section 4.1.1 where the rank of kernel matrix is low, we have demonstrated that our proposed scheme works for a large range of kernel parameters.



3.1 Abalone, $\frac{r}{n} \approx 2.5\%$ 3.2 Pendigits, $\frac{r}{n} \approx 3.4\%$

Figure 3: Plots for relative error vs $\frac{1}{h^2}$ for different kernel approximation methods. For the two BBF methods (our algorithm), the dark blue one fixed the number of clusters k to be 20, while for the one denoted by the light blue line the k was allowed to vary. The memory costs for all methods and all kernel parameters are fixed to be roughly the same. We use $\frac{r}{n}$ to describe the memory footprint, where r is the rank used for low-rank methods, n is the number of data points. The kernel function we use is the Gaussian kernel: $\exp(-\|x-y\|_2^2/h^2)$. On the x-axis, $\frac{1}{h^2}$ varies from small to large. As the rank of the matrix increases, the gap (in terms of accuracy) between low-rank approximation methods and BBF becomes larger.

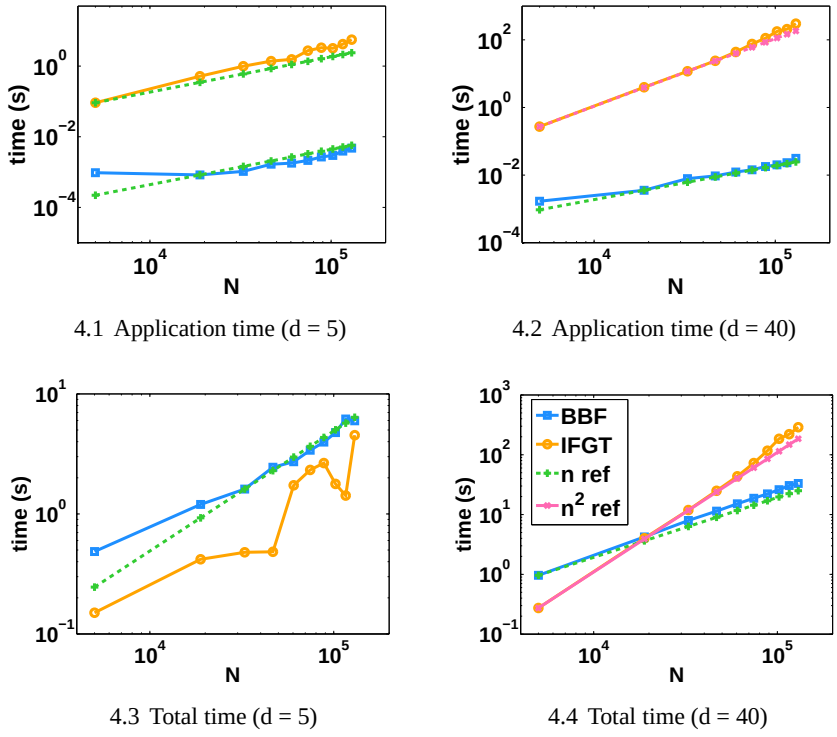
Table 3: Summary statistics for abalone and pendigits datasets with the Gaussian kernel $\exp(-\|x-y\|_2^2/h^2)$, where r is the rank, λ_{r+1} and λ_r are the r+1-th and r-th largest eigenvalues, M_r is the best rank-r approximation for M, and M is the exact matrix, l_r is the r-th largest leverage score scaled by $\frac{n}{r}$.

Dataset	r	$\frac{1}{h^2}$	$\left\lceil \frac{\|M\|_F^2}{\|M\|_2^2} \right\rceil$	$\frac{\lambda_{r+1}}{\lambda_r}$	$100\frac{\|M_r\|_F}{\|M\|_F}$	l_r
Abalone	100	0.25	2	0.9712	99.9991	4.34
		1	4	0.9939	99.8688	2.03
		4	5	0.9898	97.3333	1.94
		25	15	0.9979	72.0058	5.20
		100	175	0.9988	33.4007	12.60
		400	931	0.9991	19.4731	20.66
		1000	1155	1.0000	16.5210	20.88
Pendigits	252	0.1	3	0.9987	99.9937	2.39
		0.25	6	0.9956	99.7912	1.83
		0.44	8	0.9920	98.9864	1.72
		1	12	0.9945	93.6439	2.02
		2	33	0.9965	77.6317	2.90
		4	207	0.9989	49.6018	4.86
		25	2794	0.9998	19.8506	14.78

4.2 Complexity of BBF

We first evaluate the growth of running time on synthetic datasets, with a comparison against the IFGT; then we examine the complexity on two real datasets. The kernel function used in this section is the Gaussian kernel.

The IFGT has a linear complexity for both time and memory with a fixed dimension d. However, when d increases ($d \geq 10$), the number of data points n need to be large enough for the algorithm to exhibit a linear growth. BBF, however, does better than the IFGT in the sense that when d gets large, it does not require n to be very large to show this linear behavior. To demonstrate this, we investigate the performance of BBF and the IFGT on a synthetic dataset with different dimensions. Figure 4 shows that for a desired tolerance 10^{-3}, when $d = 5$, the IFGT behaves as $\mathcal{O}(n)$, but when $d = 40$, the behavior of the IFGT is quadratic. However, we can see that for all cases BBF grows linearly, and produced errors in range $[\mathcal{O}(10^{-4}),\ \mathcal{O}(10^{-3})]$. Note that the IFGT is in C++ while our algorithm is in MATLAB.



4.1 Application time (d = 5)

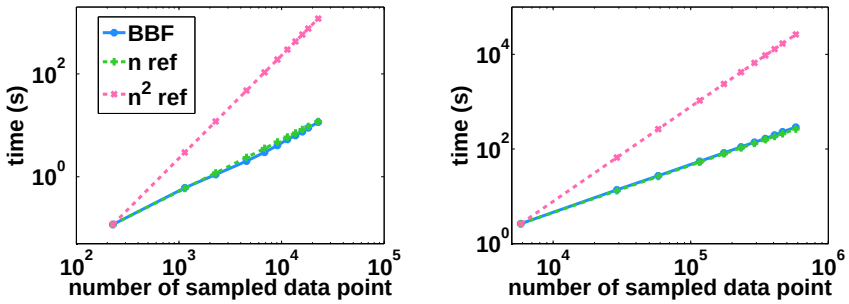
4.2 Application time (d = 40)

4.3 Total time (d = 5)

4.4 Total time (d = 40)

Figure 4: Timing for IFGT and BBF (loglog scale) on a synthetic dataset with dimension $d = 5$ and 40, where we generated 10 centers uniformly at random in a unit cube, and randomly generated data around the centers with standard deviation 0.1. The desired accuracy given for both algorithms is 10^{-3}, and the kernel parameter h was set to 0.5. All plots have the same legends. The plots at top are for application (matrix-vector product) time, and plots at bottom are for total time (factorization plus application). The timing for BBF is linear for all cases, while the timing for the IFGT behaves differently as d changes.

Next, we will examine the linear complexity on real datasets. The results are shown in Figure 5. These results are consistent with those shown on the synthetic datasets and conclusively demonstrate that our algorithm has a linear complexity.



5.1 Census housing, h = 2 5.2 Forest covertype, h = 2

Figure 5: Factorization time (loglog scale) for data from two real datasets. The left is census housing and the right is forest covertype. To illustrate the linear growth of the complexity, we generated datasets with a varying number of points with the following strategy. We first clustered census housing and forest covertype into groups $C_1, \ldots, C_k$ ($k = 15$), and sampled a portion $p\%$ from every group, then increased p. To avoid the other factors affect the timing, we fixed the input rank for each clusters. As we can see, the timing for both datasets grows linearly.

5 Conclusions

In this paper, we proposed a linear complexity algorithm that works for a large range of kernel parameters, and that achieves comparable and often orders of magnitude higher accuracy when comparing with other state-of-art kernel approximation methods, with the same memory cost. Although our algorithm makes use of randomized algorithms, the error in the method is very consistent and has a small standard deviation. This is in contrast with other randomized methods for which the error fluctuates much more significantly.

References

[1] A.E. Alaoui and M.W. Mahoney. Fast randomized kernel methods with statistical guarantees. *arXiv preprint arXiv:1411.0306*, 2014.

[2] F. Bach. Sharp analysis of low-rank kernel matrix approximations. In *Proceedings of the 26th Annual Conference on Learning Theory*, pages 185–209, 2013.

[3] K. Bache and M. Lichman. UCI machine learning repository, 2013. URL `http://archive.ics.uci.edu/ml`.

[4] W.C. Chew, E. Michielssen, J.M. Song, and J.M. Jin. *Fast and efficient algorithms in computational electromagnetics*. Artech House, Inc., 2001.

[5] E. Darve. The fast multipole method: numerical implementation. *Journal of Computational Physics*, 160(1):195–240, 2000.

[6] E. Darve. The fast multipole method I: error analysis and asymptotic complexity. *SIAM Journal on Numerical Analysis*, 38(1):98–128, 2000.

[7] P. Drineas and M.W. Mahoney. On the Nyström method for approximating a Gram matrix for improved kernel-based learning. *The Journal of Machine Learning Research*, 6:2153–2175, 2005.

[8] P. Drineas, M. Magdon-Ismail, M.W. Mahoney, and D.P. Woodruff. Fast approximation of matrix coherence and statistical leverage. *The Journal of Machine Learning Research*, 13(1):3475–3506, 2012.

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

[10] A. Gittens and M.W. Mahoney. Revisiting the Nyström method for improved large-scale machine learning. *arXiv preprint arXiv:1303.1849*, 2013.

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

[12] T.F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[13] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of computational physics*, 73(2):325–348, 1987.

[14] L. Greengard and J. Strain. The fast Gauss transform. *SIAM Journal on Scientific and Statistical Computing*, 12(1):79–94, 1991.

[15] 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.

[16] 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.

[17] M.W. Mahoney. Randomized algorithms for matrices and data. *Foundations and Trends® in Machine Learning*, 3(2):123–224, 2011.

[18] A. Rahimi and B. Recht. Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184, 2007.

[19] T. Sarlos. Improved approximation algorithms for large matrices via random projections. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 143–152. IEEE, 2006.

[20] B. Savas, I.S. Dhillon, et al. Clustered low rank approximation of graphs in information science applications. In *SDM*, pages 164–175. SIAM, 2011.

[21] S. Si, C.J. Hsieh, and I. Dhillon. Memory efficient kernel approximation. In *Proceedings of The 31st International Conference on Machine Learning*, pages 701–709, 2014.

[22] C. Yang, R. Duraiswami, N.A. Gumerov, and L. Davis. Improved fast Gauss transform and efficient kernel density estimation. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 664–671. IEEE, 2003.

[23] K. Zhang and J.T. Kwok. Clustered Nyström method for large scale manifold learning and dimension reduction. *Neural Networks, IEEE Transactions on*, 21(10):1576–1587, 2010.

Appendix

Appendix A Clustering algorithms

For a given set of data points, we need an efficient algorithm to partition the data points. Given n data points, and the number of clusters k, the k-center and k-means problems partition the points into k clusters. The difference between k-center and k-means is the objective function: k-center minimizes $\max_i \max_{v \in C_i} \|v - c_i\|_2$, while k-means minimizes $\sum_i \sum_{v \in C_i} \|v - c_i\|_2$, where c_i is the i-th cluster and $\mathcal{C}_i$ is the center/centroid of $\mathcal{C}_i$. Both of them are NP-hard problems, but there are efficient approximation algorithms that converge quickly to a local optimum. *Farthest-point clustering* [12] is a 2-approximation algorithm for k-center problem with a cost of $\mathcal{O}(nkd)$ for each iteration. *Standard K-means algorithm* alternates between best assignments given centroids, and best centroids given assignments; it also has a cost of $\mathcal{O}(nkd)$ for each iteration.

Appendix B Randomized SVD

To obtain a BBF, one step is to compute the column basis U_i of the row sub-matrix $M_{i,:}$ ($i = 1, \ldots, k$). This can be achieved via an exact SVD, but the cost is $\mathcal{O}(mn^2)$ for a matrix of size m by n ($m \geq n$). Here, we use the randomized SVD introduced in [15]. It reduces the cost of computing a rank-r approximation of a matrix of size m by n to $\mathcal{O}(mn(r + l))$, where l is an oversampling parameter. We briefly describe the algorithm for obtaining a rank-r approximation $U\Sigma V^*$ of a matrix $M \in \mathbb{R}^{m \times n}$.

Notations. Let r denote the desired rank, l denote the oversampling parameter, and q denote the iteration parameter.

1. *Preliminaries*. We randomly generate a Gaussian matrix $\Omega \in \mathbb{R}^{n \times (r+l)}$.
2. *Compute Basis*. Apply a QR decomposition on $M\Omega$ to get Q. Repeat the following steps for q times: do a QR decomposition of M^*Q to get an updated $\hat{Q}$, apply QR again on $M\hat{Q}$ to get Q.
3. *Construct SVD*. Apply an SVD on an r by n matrix $B = Q^*M$ and get $B = \hat{U}\hat{\Sigma}\hat{V}^*$. Then, the SVD of M is given by: $U = Q\hat{U}, \Sigma = \hat{\Sigma}, V = \hat{V}$.

Appendix C Randomized sampling method with linear cost

We can further reduce the cost for constructing a basis by working on a sub-sampled space. Hence, we need to seek an efficient and effective method of sampling columns from a matrix $M \in \mathbb{R}^{m \times n}$. We use a sampling algorithm [9] with complexity $\mathcal{O}(r^2(m + n))$, where r is the number of samples. We describe the algorithm below:

Notations. Let $M \in \mathbb{R}^{m \times n}$ be the matrix we sample columns from, let r denote the desired number of columns, l denote the oversampling parameter.

1. *Get important columns*. Uniformly sample r rows from M, and denote the index set as Γ_r. Apply a pivoted QR factorization on $M_{\Gamma_r,:}$ to get the top r important column indices, denoted as Π_c.
2. *Get important rows*. Uniformly sample l columns, and denote the index set as Γ_c. Update $\Pi_c = \Gamma_c \cup \Pi_c$. Apply a pivoted LQ factorization on $M_{:,\Pi_c}$ to get the top $r + l$ important row index set, denoted as Π_r.
3. *Update important columns.* Uniformly sample l rows, and denote the index set as Γ_r, update $\Pi_r = \Pi_r \cup \Gamma_r$. Apply a pivoted QR factorization on $M_{\Pi_r,:}$ to get the top r important columns.

Appendix D Proof of Theorem 3.2

Proof. For the simplification of notation, let us denote $\hat{U} = U(\mathcal{I}, :)$, $\hat{V} = V(\mathcal{J}, :)$, and $\hat{M} = M(\mathcal{I}, \mathcal{J})$. Directly from $M = UCV^T$, we get $\hat{M} = \hat{U}C\hat{V}^T$. By multiplying $\hat{U}^\dagger$ and $(\hat{V}^T)^\dagger$ on both sides, we have $\hat{U}^\dagger\hat{M}\hat{V}^\dagger = \hat{U}^\dagger\hat{U}C\hat{V}^T(\hat{V}^T)^\dagger$. Since $\hat{U}$ is a tall and skinny full rank matrix, we have $\hat{U}^\dagger\hat{U} = I$, the same goes for $\hat{V}^T(\hat{V}^T)^\dagger$. Therefore, $\hat{U}^\dagger\hat{M}(\hat{V}^T)^\dagger = C$. □

Appendix E Parameters and implementations used for other kernel approximation methods

The standard Nyström. We uniformly sample $2k$ columns without replacement for a rank k approximation.
K-means Nyström. We use the code provided by the author.
Leverage score Nyström. We compute *exact* leverage scores ($\mathcal{O}(n^2)$) and sample $2k$ columns with replacement for a rank k approximation.
Memory Efficient Kernel Approximation. We use the code provided by the author.
Improved Fast Gauss Transform. We use the code provided by the author. The code is implemented in C++ and we used its MATLAB interface.