Python software suite of geometry-oblivious Fast Multipole Methods (GOFMM): its application in statistical plotting and high dimensional data classification

Tianyi Ge

April 18, 2020

Contents

1 Abstract

This guided research focuses on optimizing pre-existing python codes of GOFMM and incorporating new functionality into the embedded balanced tree data structure by GOFMM. The original codes rely heavily on nested loops, conditional checks and index tracking, making it difficult to apply GOFMM to new applications. The revised version implements a set-oriented interface on setting up the GOFMM data structure. Particularly, we formalize all sampling methods in set. As a result, the codes are much shorter, understandable and efficient as it involves basic mathematical operations on a fundamental data structure. Furthermore, we utilize **scipy.linalg.interpolative** package to simplify analysis in **skeleton**, such as interpolative decomposition and QR factorization. The resultant codes run much faster and are user-friendly in implementing related applications.

With our new GOFMM interface, we devised two examples that demonstrate high usability of GOFMM in statistical plotting and image recognition. Our first example takes in 2D position data and classifies points based on their density. By exploiting relative fast search and data structure in GOFMM, this user case accelerates underlying matrix-vector multiplication. In comparison to its raw data plot, our plot uses Gaussian kernel density estimation (Gaussian KDE) and displays a high level of accuracy. Our second example pipes over one thousand 8×8 images into GOFMM, utilizing its tree-like data structure for fast search and low storage. Then, we implement Gaussian KDE to learn over training dataset so that the updating model can accurately classify testing data. The result shows high accuracy of our model classification. With 30 training images, the accuracy of classifying 44 images is $\frac{43}{44}$.

2 Introduction and Theory

In numeric linear algebra, a dense matrix, whose entries are mostly nonzero, can be represented as a linear combination of matrices each of which keeps only one nonzero entry. Storing such a non-sparse matrix requires $O(N^2)$ space in memory, thus causing a stroage problem when N is large. We apply an approximation method to the dense matrix as a way to reduce the storage need by using hierarchical matrices. Such method costs only $O(Nlog(N))$ units of storage. GOFMM compresses a dense symmetric positive definite matrix (SPD) by creating a low-rank approximation [1]. Such compression requires $O(Nlog(N))$ storage where N is matrix size. It also reduces the cost of $N \times N$ matrix-vector multiplication from $O(N^2)$ to $O(Nlog(N))$.

2.1 Problem Statement

Dense SPD matrices appear often in scientific computing and data analysis. They are widely used particularly in areas such as kernel methods for statistical learning and image classification [2]. Suppose we are given a $m \times n$ dense SPD matrix K where each entry K_{ij} represents some data point. The storage costs as well as matrix-vector multiplication on such matrix take $O(N^2)$ where $N = \min(m, n)$. If N is large, such costs will be too expensive. Take *N-body* problem for example.

Given a set of N points $x_1, x_2, ..., x_N$ for $x_i \in \mathbb{R}^d$. Our goal is to calculate a potential and assign it to each point. Such potential is usually based on relation of the current point with all other points. As an example in Coulomb's Law, nearby particles contribute more than distant ones to the potential of current particle. This potential depends mainly on distance and charges. If $N \to +\infty$, calculating potential at every data point while taking all others into account is infeasible. The program needs to perform $T_{op} = O(N^2)$ calculations and the total computational costs would be $T_{op} \times T_{comp}$ where T_{comp} is the cost of computing potential between any two points. To tackle this issue, we introduce a statistical tool called *kernel density function* to sample all data points according to some metric. Then, potential at a point x_i can be computed as [3]

$$u_i = u(x_i) = \sum_{j=1}^{N} K(x_i, x_j) w_j \tag{1}$$

where w_j is the intrinsic weight of point j and $K(x_i, x_j)$ is the probability density at point (x_i, x_j). Although presampling in this method reduces some costs, the program still has to deal with $O(N^2)$ multiplications. We can reduce computational complexity by forming a hierarchical data structure on neighboring points. We rewrite Eq. (1) as [3]

$$u_i = \sum_{p \in \text{Near}_i} K_{ip} w_p + \sum_{p \in \text{Far}_i} K_{ip} w_p \tag{2}$$

where Near_i is a set of points relative to point i; and for these points their products $K_{ip}w_p$ is calculated one by one. On the other hand, Near_i is a set of points relative to point i; and for these points their products can be computed by *low-rank approximation* which reduces dimensionality and brings down storage costs to $O(N\log(N))$.

2.2 Hierarchical Low Rank Structure

Consider the kernel matrix K in Eq. (1). The *low-rank approximation* allows us to construct a hierarchically low-rank matrix $\tilde{K}$, also known as H-matrix, such that $\left\|\tilde{K} - K\right\| / \|K\|$ is small. Such H-matrix $\tilde{K}$ of K can be decomposed into

$$\tilde{K} = D + U + S \tag{3}$$

where D consists of diagonal blocks representing nearby points, U a dense block matrix and S a sparse matrix, both of which represent off diagonal entries. Notice, since U is a dense block matrix, it allows a block-wise low rank approximation [3]. According to theory, a matrix-vector application $\tilde{K}w$ takes $O(N\log N)$. In the following subsections, we introduced two major instances of *low-rank approximation* that we use in our codes.

2.3 Singular Value Decomposition

Consider a $m \times n$ matrix A where each row is a single sample of m-dimensional data point. Two datasets in our experiment both have $m >> n$. A can be decomposed into

$$A = U\Sigma V^* \tag{4}$$

where U is a $m \times m$ unitary matrix, Σ a $m \times n$ diagonal matrix with singular values and V a $n \times n$ unitary matrix. The columns of U and V are *left-singular vectors* and *right-singular vectors*, respectively. The diagonal entries in Σ are monotonically decreasing, namely,

$$\Sigma_{ii} \geq \Sigma_{jj} \quad \text{for} \quad i \leq j \tag{5}$$

In practice, we set up a cutoff line, say k, such that

$$\forall j > k,\ \Sigma_{jj} = 0 \tag{6}$$

As a result, we don't need to calculate some parts of matrix multiplication in Eq. (4). This scheme therefore accomplishes the goal of dimension reduction. In general, *singular value decomposition* takes $O(mn^2)$.

2.4 Interpolative Decomposition

An *interpolative decomposition(ID)* of a matrix A approximates it by using A's own columns. For a $m \times n$ matrix A of rank $k < \min(m, n)$,

$$AS = [AS_1\ AS_2] = AS_1[I\ T] \tag{7}$$

where S is a permutation matrix and S_1, S_2 are the unit vectors that select certain columns of S. We further write Eq. (7) as [4]

$$A = BP \quad \text{for} \quad B = AS_1, P = [I, T]S^T \tag{8}$$

where B and P are the *skeleton* and *interpolation* matrices of A, respectively. By design, B consists of a subset of columns of matrix A. Since we can reuse some columns of A in construction of A, ID yields a lower storage costs. Generally, it takes $O(kmn)$.

In the following section, we optimized sampling methods that generate matrix S in Eq. (7) and set up for SVD.

3 Code Optimization

This section focuses on improving two class methods in the **tree** library, **skeletonize** and **evaluate**. **tree** library contains both a BST tree data structure that effectively store all data points and class methods that compute SVD, ID and QR factorization. **skeletonize** computes the skeleton, interpolation matrix and SVD of the currently stored data points. **evaluate** recursively traverse the tree and conducts the matrix-vector multiplication.

3.1 Sampling Methods

One of the major drawbacks in the pre-existing **skeletonize** codes is its sampling methods. Sampling lays foundation for calculating the nearest neighbors of a node and later computing the skeleton. The original code used nested loops to find **self.selected_rows**, which is the S matrix in Eq. (8). As we can see in the simplest sample method in the code,

```
if (type=="uniform"):
    if verbose:
        print 'sampling rows uniformly..'
        for i in range(rows):
            l=-1
            while (l in self.idx_list or l in self.selected_rows or l<0):
                l=int(np.random.rand()*N)
                self.selected_rows[i]=l
```

uniform sampling essentially selects $M = len(rows)$ different integers among 0 to N. The problem with this nested loop implementation is that as the iteration number i gets bigger, it takes more rounds for the inner *while* loop to find a suitable l. l's generated by the previous outer rounds populate a large portion of random number range. Since a new l must be different from all previous l's, generation of $l's$ in the later rounds take more time. Moreover, as sampling methods get more complicated, such as neighboring-based search methods and skeleton methods in this code, more and more nested loops need to be used. It adds to difficulty of code readability and slows down running speed on generating S matrices.

We formulate the sampling problem in set notation for optimization. The uniform sampling is a selection of $M = len(rows)$ distinct elements without replacement from the random range. We set up sample scope, which spans from 1 to N, and complement, which is the pool from which l should not be picked. Then, we pick M distinct numbers from (sampleScope - complement) without replacement. The function of uniform sampling can be written as follows:

```
def uniform_sampling(sampleScope, complement, numSamples):
    sampleRange = np.setdiff1d(sampleScope, complement)
    return np.random.choice(sampleRange, numSamples, replace=False)

// Call on uniform sampling
self.selected_rows = np.append(self.selected_rows,
                               uniform_sampling(range(N),
                                                self.idx_list,
                                                rows))
```

It turns out that we can incorporate the uniform sampling routine further into neighboring based sampling method. This method first samples from a pre-assigned neighboring sets. If this subsample doesn't fulfill the requirement, then we must uniformly sample the rest. Furthermore, we rewrite the skeleton sampling method into a 3-stage sampling routine. The skeleton first samples the first part of its elements from the previous skeleton, the second part from the neighboring based method and the last part from the uniform sampling method.

```
elif (type == 'skeletons'):
    sampleScopeSets = (self.relevant_skeletons, self.neighbors, range(N))

    complementSets = (np.empty(shape=[1, 0], dtype=int),
                      np.union1d(self.idx_list, self.relevant_skeletons),
                      self.idx_list)

    rest = rows
    for i in range(3):
        sampleScope = sampleScopeSets[i]
        complement = np.union1d(self.selected_rows, complementSets[i])
        sampleRange = np.setdiff1d(sampleScope, complement)

        self.selected_rows = np.append(self.selected_rows,
                                       uniform_sampling(sampleScope,
                                                        complement,
                                                        min(len(sampleRange),
                                                            rest)))
        rest -= len(sampleRange)

        if (rest <= 0):
            break
```

3.2 Singular Value Decomposition

Another drawback of the pre-existing code is lots of raw coding on calculation of parameters in SVD. The old code spent almost 100 lines of code calculating accuracy, doing inverse mapping and finding optimal adaptivity. The accuracy calculation requires sophisticated calculation in a loop, while adaptivity calculation needs nested loops and many condition checks. So, both need to be optimized. Moreover, inverse mapping creates a dictionary with key and value, and later tries to find key according to some value. This type of operation consumes a large memory when matrix is big and also very costly as we need to go through all dictionary values to find corresponding keys.

We recognize much of the inefficiencies can be mitigated by applying **scipy.linalg.interpolative** package to Eq. (8). Instead of SVD, the package uses an efficient algorithm called *rank-revealing QR* to factorize the input matrix.

```
rows = len(self.selected_rows)
cols = len(self.selected_columns)
G = np.zeros((rows, cols))
for i in range(rows):
    G[i, :cols] = K[int(self.selected_rows[i]), :cols]

    [Q, R, PI] = linalg.qr(G, pivoting=True)

    self.s, idx, proj = sli.interp_decomp(G, rtol, rand=False)

    self.skeleton = G[:, idx[:self.s]]

    self.P = sli.reconstruct_interp_matrix(idx, proj)

    if (self.left is not None):
        self.Permute = PI
        self.selected_columns = self.selected_columns[PI[0:self.s]]
```

Our new implementation above takes only 17 lines to implement the original one with around 100 lines that are filled with nested loops, conditions and data structures. The code cleanup paves way for readability and usability.

3.3 Optimization Performance

We analyze the performance of old codes and improves ones in terms of accuracy and running speed. It turns out that most sampling methods plus their evaluation time have gained a speedup of 5% compared to the old one. Moreover, the mean and max error for most sampling method plus its evaluation drops by an order of magnitude from 10^{-15} to 10^{-16} and 10^{-13} to 10^{-14}, respectively.

	Mean Error	Max Error	Running Speed
old	1.1×10^{-15}	8.5×10^{-14}	0.83s
new	7.7×10^{-16}	3.08×10^{-14}	0.80s

Table 1: Performance comparison of neighbor based sampling

	Mean Error	Max Error
old	1.60×10^{-15}	6.4×10^{-13}
new	7.5×10^{-16}	3.08×10^{-14}

Table 2: Performance comparison of uniform sampling

	Mean Error	Max Error	Running Speed
old	9.80×10^{-16}	7.60×10^{-14}	2.15s
new	7.25×10^{-16}	3.08×10^{-14}	2.06s

Table 3: Performance comparison of skeleton sampling

4 GOFMM Applications

To demonstrate usability of revised GOFMM and the library in general, we devised two examples. The first example is about piping 2D position data sets into the GOFMM tree structure, computing density of data points and plotting. The second one is about using GOFMM to classify images. Both examples use *Gaussian Kernel Density Estimation (Gaussian KDE)*. The reason for using a continuous estimator like Gaussian KDE is that traditional bin estimator of statistical distribution changes dramatically when we modify bin size. Approximating groups of neighboring points using KDE in general smooth down fluctuation and gives a much more accurate distribution [6]. First, we will briefly go over some theoretical parts of KDE.

Let $X_1, ..., X_n \in \mathbb{R}^d$ be independently, identically distributed samples from some distribution with PDF $\hat{f}_h(x)$. Then, its *kernel density estimator* can be written as [5]

$$\hat{f}_n(x) = \frac{1}{nh^d} \sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right) \tag{9}$$

where h is the bandwidth, which characterizes flatness of the curve and $K : \mathbb{R}^d \to \mathbb{R}$ kernel function. The probability density at x sums up all local kernel density at x centering at each different X_i. The kernel function depends on which statistical distribution that we use to fit data. One of the most common examples is to use Gaussian distribution [5].

$$K(x) = \frac{\exp\left(-\|x\|^2/2\right)}{v_{1,d}}, \quad v_{1,d} = \int \exp\left(-\|x\|^2/2\right) \tag{10}$$

In practice, in order to calculate the PDF, we first need to fit Gaussian curves at every group of points that are delimited by grids. Then, we calculate the value of x at each locally fitted Gaussian curve. In the end, we sum all values of x up for the PDF.

4.1 2D Statistical Plotting

In this subsection, we classify 2D data points based on their local density. The input is coupled measurements that are randomly generated from the 2D normal distribution. Our goal is to reassemble these points and classify them according to their neighboring density. GOFMM accomplishes two tasks here. The first is to store the data in its tree structure using **skeletonize**. And the second is to fast calculate a matrix-vector multiplication which is required in Eq. (9) using **evaluate**. Here, we present the density plot along with their original data points in Fig. **1**.

This application takes advantage of GOFMM's capacity of efficiently storing and querying the data. Moreover, it accelerates the matrix-vector multiplication. This example is implemented in the code **gofmm_app.py**.

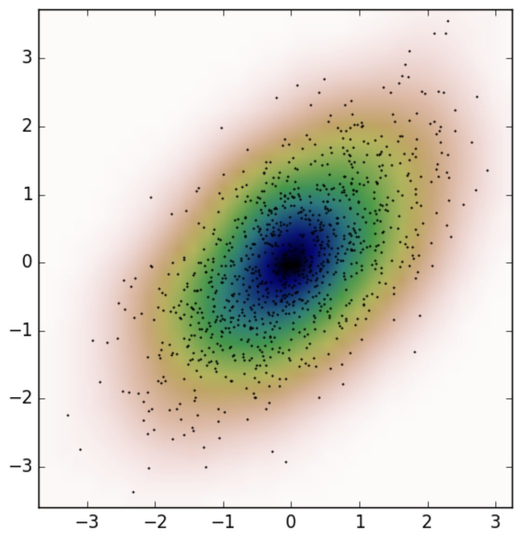

Figure 1: 2D density plot using Gassusian KDE and GOFMM

4.2 Image Classification

We can further expand GOFMM to include features on classifying images. We are interested in applying Bayes classification to testing data based on training data. Let x be a random datapoint from sample space and y a random label from all labels. According to Bayes' Rule

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)} \tag{11}$$

Assuming all samples are distinctive, then $P(x)$ is a constant. Therefore, $P(y|x) \propto P(x|y)P(y)$. Our task now is to calculate $P(x|y)$ and $P(y)$. We implement Bayes classification as follows [6]

1. Split the training data into sets according to their labels.
2. Perform KDE on each distinctive label (set from step 1). This step gives $P(x|y)$.
3. Calculate $P(y)$ based on occurrence of y relative to all samples
4. Given a testing point x, calculate $P(x|y)$ for each label y. Select the one with the largest probability and label x with this y.

In order to perform KDE in step 2, we first need to construct a Gaussian kernel matrix, K. An entry K_{ij} is calculated by Eq. (10). It corresponds to testing sample i with respect to training sample j. Then, we construct the KDE matrix, E, based on K by summing up probability of testing sample i to training samples of each label. E_{ij} corresponds to testing sample i with respect to j label in training data. The value E_{ij} is proportional to $P(x|y)$ in step 2.

We aim to run our self-implemented KDE on images. The dataset comes from **sklearn.datasets.load__digits** which consists of more than one thousand 8×8 black and white images [7]. Each image stores a hand-written digit from 0 to 9. We design an experiment on recognizing 44 testing images based on 30 training samples. Each of the training sample is manually labeled. The KDE is then applied to the testing sample with respect to training sample. The result is in Fig. **2**.

The KDE prediction on 44 testing images gives labels

```
$ python -i kde_handwriting.py
>>> predictionLabels.reshape(4,11)
array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0],
       [1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1],
       [2, 3, 4, 5, 6, 7, 8, 9, 0, 9, 5],
       [5, 6, 5, 0, 9, 8, 3, 8, 4, 1, 7]])
```

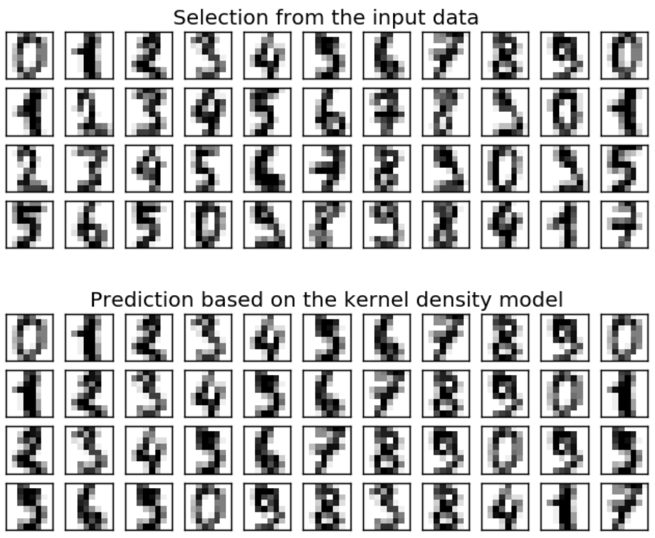
Selection from the input data
Prediction based on the kernel density model

Figure 2: Raw data and KDE prediction. The upper image displays the raw 44 handwriting digits. The lower image presents our KDE prediction on these 44 images. The prediction of a digit is demonstrated by an image that was manually sorted by label.

By comparing the predcitionLabels with raw image data, we find that the Gaussian KDE can successfully predict 43 out of 44 images. The success rate is 97.73%. We can futher improve the rate by adding our prediction result to training data. Then, we can predict another set of new images. And we can keep doing this to improve our training model. The code of this example is stored in **kde_handwriting.py**.

5 Conclusion

This guided research project focuses on code optimization and software package integration. We first implemented sampling methods in set and demonstrated an improvement of code readability and efficiency. Then, we implemented low-rank compression completely based on the python package **scipy.linalg**. This modification allows us to add more features to the general GOFMM package later on as the previous code did hard coding with nested loops and conditions. The new code achieves a 5% speed up and an order of magnitude improvement on error for a 1024×1024 matrix sample.

After GOFMM had been set in place, we wrote additional packages to implement Gaussian KDE and two user cases to demonstrate the usability of GOFMM. The first case is about generating statistical density plot of random 2D points. It takes advantage of GOFMM's tree data structure and matrix-vector multiplication. It reduces the storage space and query speed to $O\left(N \log(N)\right)$. The second example is about predicting image labels using Gaussian KDE based on training data. The result shows that our implementation not only is capable of processing high dimensional data, but also accurately predicts image label. To further improve the training model based on Gaussian KDE, we can further add testing data from the previous round of running to the training data in the current round. In this way, we assemble more reliable data for future classification.

6 Acknowledgment

I would like to give special thanks to my advisor, Severin Reiz, for his mentorship on this project. All data analysis and modeling are carried out in python. Software packages concerning this project are kept under https://gitlab.lrz.de/tum-i05/public/gofmm-python. In the original GOFMM code, **tree.py** is heavily modified especially with its two core functions, **skeletonize** and **evaluate**. In addition to **tree.py**, additional package that I wrote such as **handwriting_kde.py** are thoroughly commented. So, it is recommended to read this report along with code comments if confusion arises. The **README.md** file documents sources, algorithms and packages that I use to write package. Lastly,

continuous integration is also implemented for running scenarios, statistical plotting and handwriting recognition in **.gitlab-ci.yml**.

References

[1] C. Yu, S. Reiz & C. Biros, "Distributed O(N) Linear Solver for Dense Symmetric Hierarchical Semi-Separable Matrices" (2019).
https://mediatum.ub.tum.de/doc/1522878/1522878.pdf

[2] C. Yu, S. Reiz, J. Levitt & C. Biros, "Geometry-Oblivious FMM for Compressing Dense SPD Matrices" (2017).
https://arxiv.org/pdf/1707.00164.pdf

[3] S. Reiz, "Black Box Hierarchical Approximations for SPD Matrices" (2017).
https://www5.in.tum.de/pub/Reiz2017_Thesis.pdf

[4] The SciPy community, "Interpolative matrix decomposition (scipy.linalg.interpolative)" (2019).
https://docs.scipy.org/doc/scipy/reference/linalg.interpolative.html

[5] Y. Chen, "A Tutorial on Kernel Density Estimation and Recent Advances" (2017).
https://arxiv.org/pdf/1704.03924.pdf

[6] J. VanderPlas, "Python Data Science HandbookEssential Tools for Working with Data" (2016).
https://jakevdp.github.io/PythonDataScienceHandbook/05.13-kernel-density-estimation.html

[7] J. Boisberranger, J. Bossche und etc, "sklearn.datasets.load_digits" (2016).
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html#sklearn.datasets.load_digits