

1 Homework 3

Compute the matrix-vector product

$$\mathbf{v} = \Phi \mathbf{u}, \quad \text{or} \quad v_j = \sum_{i=1}^N \Phi_{ji} u_i, \quad j = 1, \dots, M, \quad (1)$$

with absolute error $\epsilon < 10^{-6}$, where

$$\Phi = \begin{pmatrix} \Phi_{11} & \Phi_{12} & \dots & \Phi_{1N} \\ \Phi_{21} & \Phi_{22} & \dots & \Phi_{2N} \\ \dots & \dots & \dots & \dots \\ \Phi_{M1} & \Phi_{M2} & \dots & \Phi_{MN} \end{pmatrix}, \quad \mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \dots \\ u_N \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ \dots \\ v_M \end{pmatrix}, \quad (2)$$

$$\Phi_{ji} = \frac{1}{y_j - x_i}, \quad i = 1, \dots, N, \quad j = 1, \dots, M.$$

and x_1, \dots, x_N are random points uniformly distributed on $[0,1]$, $M = N - 1$, and each y_j is located between the closest x_i 's on each side, $j = 1, \dots, N - 1$ using optimized version of Pre-FMM that uses R-expansions near the centers of the target boxes.

1. Draw a sketch of the Pre-FMM algorithm. Identify supporting programs that you will need to use.
2. Write supporting programs, which allow you to determine necessary indices for the sources and targets for arbitrary N and K and box centers. (See appendix).
3. Evaluate the truncation number, $p(K, N)$, that guarantees the specified accuracy as a function of the number of boxes K and the size of the problem, N . N will vary in the tests between 10^2 and 10^4 and you can use some simplifications and find a p that should be sufficient to provide the necessary accuracy.
4. Evaluate theoretically the optimal number of boxes $K_{opt}(N)$ for space division based on the obtained evaluations of p for the specified accuracy.
5. Write and test for accuracy a program which provides you the R -expansion coefficients for a given target box (or target box center) and a source.
6. Write a program that implements both straightforward multiplication based on Eq. (1) and Pre-FMM that uses R-expansions.
7. Provide a graph of the absolute maximum error between the straightforward method and the Pre-FMM for $N = 10^3$, K varying between 10 and 100, and p from your theoretical evaluations. Compare the results with your evaluations of the accuracy. You may find that the theoretical p can be substantially reduced to stay within the specified error bounds. In this case you may (or may not) reduce p and use some experimental values to proceed further.
8. Provide a dependence of the CPU time required by the Pre-FMM as a function of K for $N = 10^3$ ($10 < K < 100$). Determine K_{opt} experimentally and compare with the theoretical evaluations (use actual p). Scale $K_{opt}(N)$ for computations with varying N . Plot your scaled function $K_{opt}(N)$.
9. Provide a graph of actual error (between the standard and the fast method with $K = K_{opt}(N)$) for N varying between 10^2 and 10^3 and the truncation number used.
10. Provide a graph that compares the CPU time required by the straightforward method and the Pre-FMM for N varying between 10^2 and 10^3 for straightforward and N varying between 10^2 and 10^4 for the optimized Pre-FMM. Compare results with theoretical complexities of the algorithms.

11. Find the “break-even” point (i.e. N at which the “Fast” method requires the same CPU time as the straightforward method) for your implementation.

1.1 Appendix: A way to create a data structure for the 1D-FMM

The way we organize points into boxes should have a cost consistent with the FMM. In the 1-D case it is very simple to do this since points can be naturally sorted. We have two sets of points \mathbb{X} and \mathbb{Y} (sources and evaluation points). Assume that these arrays are sorted. (Hint: Use Matlab *sort* function to sort \mathbb{X} after it is generated and find \mathbb{Y} as points between sources, so \mathbb{Y} is also sorted).

If we have a sorted list, it is easy to assign points to boxes by using an array of *bookmarks*. Using this you can determine all sources or evaluation points belonging to a particular box rapidly.

The idea is the following. Because \mathbb{X} is ordered you can create arrays *BookmarkLeft* and *BookmarkRight* (in the current problem you may find that one array is sufficient, but you may use two, for easier and faster search). These arrays contain bookmarks which show the bounds of \mathbb{X} indices as shown in the table below. To generate such a bookmark table you only need $O(N)$ operations (one pass through your ordered data) and it takes memory of order $O(K)$.

<i>BoxIndex</i>	1	2	...	k	...	K
<i>Set</i>	x_{n_0}, \dots, x_{n_1} ,	$x_{n_1+1}, \dots, x_{n_2}$,	...	$x_{n_{k-1}+1}, \dots, x_{n_k}$,	...	$x_{n_{K-1}+1}, \dots, x_{n_K}$,
<i>BookmarkLeft</i>	n_0	$n_1 + 1$...	$n_{k-1} + 1$...	$n_{K-1} + 1$
<i>BookmarkRight</i>	n_1	n_2	...	n_k	...	n_K

If it appears that the box with index k is empty you may put *BookmarkLeft(k)* and *BookmarkRight(k)* equal to zero. You also may create an array of *NonEmptyBoxes* where you store consequently the non-empty box indices k only, so it looks like $\{k_1, \dots, k_L\}$, where k_1, \dots, k_L are indexes of the nonempty-boxes. In this case you will skip possible empty boxes and your program will be faster (if such boxes exist). A similar bookmark table should be also created for \mathbb{Y} and non-empty evaluation box indexes can be stored in an array.

Write a Matlab function *SetDataStructure(Set, K)*, which returns you arrays *BookmarkLeft*, *BookmarkRight*, and *NonEmptyBoxes*.

Call this function twice with $Set = \mathbb{X}$ and $Set = \mathbb{Y}$.

Then the procedure of going over non-empty boxes with respect to evaluation points looks like

```
for k = EvaluationonNonEmptyBoxes
    for j = EvaluationBookmarkLeft(k) : EvaluationBookmarkRight(k)
        y = Y(j); % get the evaluation point coordinate in this box;
        [Required operations with y];
    end;
end;
```

Based on this idea you can also create your program which for given Evaluation Box runs through appropriate source boxes.