A Python Implementation of the Fast Multipole Method

MA 221 - UC Berkeley

Luis Barroso-Luque

May 3, 2018

1. The algorithm and its implementation

The Fast Multipole Method (FMM) reduces the amount of work necessary to compute n-body interactions. Greengard and Rohklin first proposed the algorithm for 2D electrostatic interactions [1]. I have implemented it here in the same spirit.

1.1 Math preliminaries

The current implementation of the FMM deals with 2D Coulomb potentials of the form,

$$\phi_{\mathbf{x_0}}(x,y) = -\log(|\mathbf{x}-\mathbf{x_0}|) \tag{1.1}$$

Using complex variables $(x,y) \in \mathbb{R}^2 \rightarrow x + iy = z \in \mathbb{C}$, the potential is expressed as,

$$\phi_{\mathbf{x_0}}(x,y) = \Re(-\log(z-z_0)) \tag{1.2}$$

The key idea the multipole method, is to express the potential as a multipole expansion, and subsequently shift these expansions appropriately and convert them to local expansions for evaluation of the potential at each particle location.

The main ingredient for the multipole expansion is the following expansion of the potential in 1.2,

Lemma 1 For a single point charge q located at z_0, the potential at $|z| > |z_0|$ can be expanded as,

$$\phi_{z_0}(z) = q\log(z-z_0) = q\left(\log(z) - \sum_{k=1}^{\infty}\frac{1}{k}\left(\frac{z_0}{z}\right)^k\right) \tag{1.3}$$

The multipole expansion is obtained by putting together the potential from many particles as expressed in 1.3.

Theorem 1 (multipole expansion) Given n point charges of charge $\{q_i, i = 1\ldots n\}$ located at $\{z_i, i\ldots n\}$, with $|z_i| < r$. Then the potential $\phi(z)$ for any $z \in \mathbb{C}$, $|z| > r$ is given by

$$\phi(z) = Q\log(z) + \sum_{k=1}^{\infty}\frac{a_k}{z^k} \tag{1.4}$$

where,

$$Q = \sum_{i=1}^{n} q_i \quad \text{and} \quad a_k = \sum_{i=1}^{n}\frac{-q_i z_i^k}{k}$$

The FMM uses the multipole expansion efficiently by shifting multipole expansions to centers of appropriate cells in the computational domain and by converting the multipole expansions to local (Taylor) expansions to evaluate them for particles inside cells. The required mathematical tools are given in the lemmas that follow.

Lemma 2 (multipole shift) For a given multipole expansion for a given set of point charges located inside a circle D_0 of radius R centered at z_0,

$$\phi(z) = a_0 \log(z - z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z - z_0)^k}$$

For z outside the circle D_1 of radius $(R + |z_0|)$ and center at the origin the potential is,

$$\phi(z) = a_0 \log(z) + \sum_{l=1}^{\infty} \frac{b_l}{z^l} \tag{1.5}$$

where

$$b_l = \sum_{k=1}^{l} a_k z_0^{l-k} \binom{l-1}{k-1} - \frac{a_0 z_0^l}{l} \tag{1.6}$$

Lemma 3 (multipole to local) Given a set of charges located inside a circle D_1 of radius R and centered at z_0 and such that $|z_0| > (c+1)R$ with $c > 1$. Then the multipole expansion about z_0 converges inside a circle D_2 of radius R centered at the origin to the following,

$$\phi(z) = \sum_{l=0}^{\infty} b_l z^l \tag{1.7}$$

where

$$\begin{aligned} b_0 &= \sum_{k=1}^{\infty} (-1)^k \frac{a_k}{z_0^k} + a_0 \log(-z_0) \\ b_l &= \frac{1}{z_0^l} \sum_{k=1}^{\infty} \frac{(-1)^k a_k}{z_0^k} \binom{l+k-1}{k-1} - \frac{a_0}{l z_0^l} \end{aligned} \tag{1.8}$$

Lemma 4 (local shift) For any $z, z_0 \in \mathbb{C}$,

$$\sum_{k=0}^{n} a_k (z - z_0)^k = \sum_{l=0}^{n} \left(\sum_{k=l}^{n} a_k \binom{k}{l} (-z_0)^{k-l} \right) z^l \tag{1.9}$$

For proofs and convergence properties of the above theorems and lemmas see the original paper [1].

1.2 The quadtree data structure

The efficient application of the FMM relies on using the mathematical tools previously described in a clever manner. To do so, the 2D computational domain is decomposed into a quadtree [1]. As show in figure 1, a quadtree is a recursive subdivision of a rectangular

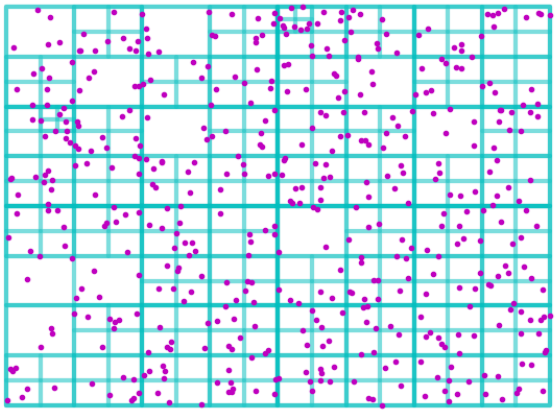


Figure 1: 4 level adaptive quadtree based division of computational domain.

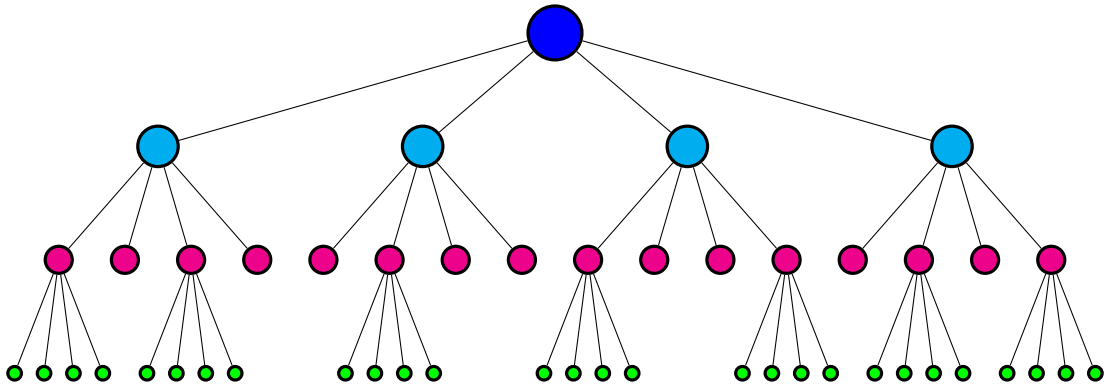


Figure 2: Schematic of a 4 level adaptive quadtree data structure. Nodes without children, or leaves contain a number of particles $n \leq$ threshold, for some threshold value.

domain into 4 equal sized rectangles up to a certain level of refinement. The actual data structure can be implemented as a tree, where nodes have 4 children each[2] as shown in 2.

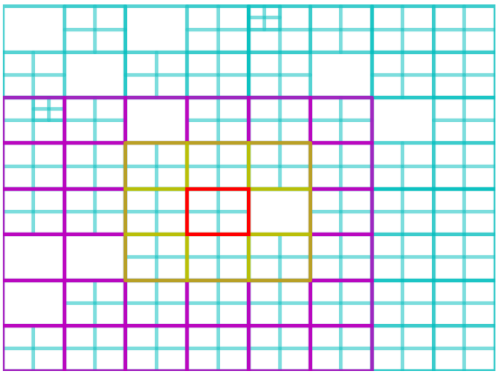
The quadtree I wrote for the FMM is an adaptive[3] quadtree in which the recursive subdivision goes on until a threshold for the maximum number of particles a cell can hold is reached. In this case, cells will be of different sizes depending on the distribution of particles in the computational domain (see figures 1 and 2).

The quadtree is implemented as a `class QuadTree`, which holds a single root node of

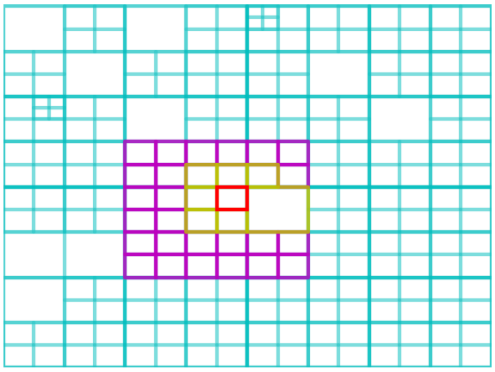
[1]For the 1D and 3D case the analogous structure is a binary tree and octtree respectively.

[2]A quadtree can also be implemented in a linear array—known as a linear quadtree—where relationships between nodes are accounted for with locational codes. In this implementation I used the tree form.

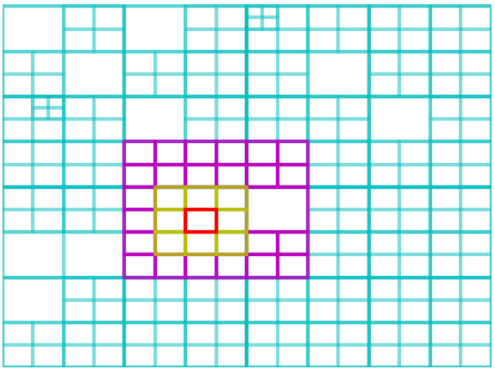
[3]An adaptive quadtree is better at handling nonuniform distributions of particles. It would be interesting to compare performance with a regular quadtree for both uniform and non-uniform distributions.



(a) Level 3



(b) Level 4



(c) Level 4

Figure 3: Examples of nearest neighbor and interaction sets for nodes at level 3 and 4. Nearest neighbor sets are colored yellow and interaction sets are colored purple.

`class Node`. The `class QuadTree` has apart from simple descriptive member variables and methods, methods to recursively traverse itself. The `class Node` has the member methods necessary to split, find neighbors and find children recursively. Each `Node` has references to its parent and children.

In addition to subdividing space, the quadtree is useful to determine a cells nearest neighbor set and its interaction set; both of which are necessary for implementing the FMM.

The nearest neighbor set of a given cell is defined all the cells of minimum size but not smaller than the given cell, that share a border or vertex with the given cell. The interaction set of a cell is the minimum size cells not smaller than the given cell, that are nearest neighbors of that cells parent or children of those nearest neighbors. Figure 3 shows a few examples of nearest neighbor and interaction sets for different levels.

1.3 Method implementation

Using the previously described quadtree data structure, the 2D FMM method was implemented following the steps given by [1], but using recursion when possible. The main steps of the algorithm are,

Upward pass Compute multipole expansions (Theorem 1 starting from leaf cells and recursively computing multipole expansions for parent cells by using the shifting Lemma 4.

Downward pass Starting from the coarsest level, convert multipole expansions using Lemma 3 from a cells interaction set into local expansions at the given cell and recursively compute the same for children cells and add its parents local expansion by shifting it according to Lemma 4.

Potential evaluation For all particles in each leaf cell, evaluate the cells inner expansion at the particles' locations, and compute interactions with all other particles in cell and nearest neighbor set directly. Add the inner expansion and direct interactions for each cell to obtain the final value of potential.

1.3.1 Upward pass

As previously described, the upward pass computes a multipole expansion for all particles in each cell. To do so efficiently, in the FMM, the multipole expansions according to Theorem 1 are computed directly for all leaf nodes. The Python function `multipole` which returns the coefficients for the multipole expansion of a set of particles is shown below.

```
def multipole(particles, center=(0,0), nterms=5):
    """Compute a multiple expansion up to nterms terms"""

    coeffs = np.empty(nterms + 1, dtype=complex)
    coeffs[0] = sum(p.q for p in particles)
    coeffs[1:] = [sum([-p.q*complex(p.x - center[0], p.y - center[1])**k/k
                for p in particles]) for k in range(1, nterms+1)]

    return coeffs
```

To compute the expansion for parent nodes, the expansions from its 4 children are shifted to the parents center using Lemma 2 and added. The Python function `_outer_mexp` does so recursively by using the function `_shift_mexp` based on Lemma 2 to shift children cell expansions. Both functions are shown below, calling `_outer_mexp` will take care of doing the upward pass as can be seen in the main `potential` function for the FMM.

```
def _shift_mpexp(coeffs, z0):
    """Update multipole expansion coefficients according for a center shift"""
    shift = np.empty_like(coeffs)
    shift[0] = coeffs[0]
    shift[1:] = [sum([coeffs[k]*z0**(l - k)*binom(l-1, k-1) - (coeffs[0]*z0**l)/
        l
                  for k in range(1, l)]) for l in range(1, len(coeffs))]

    return shift


def _outer_mpexp(tnode, nterms):
    """Compute outer multipole expansion recursively"""

    if tnode.is_leaf():
        tnode.outer = multipole(tnode.get_points(), center=tnode.center, nterms=
            nterms)
    else:
        tnode.outer = np.zeros((nterms + 1), dtype=complex)
        for child in tnode:
            _outer_mpexp(child, nterms)
            z0 = complex(*child.center) - complex(*tnode.center)
            tnode.outer += _shift_mpexp(child.outer, z0)
```

1.3.2 Downward pass

The downward pass involves converting multipole expansions to inner expansions by means of Lemma 3, in order to account for the interactions from groups of particles *far enough away*[4] from the current cell. The function `_convert_oi` compute the appropriate inner expansion coefficients from the multipole expansion coefficients.

```
def _convert_oi(coeffs, z0):
    """Convert outer to inner expansion about z0"""

    inner = np.empty_like(coeffs)
    inner[0] = (sum([(coeffs[k]/z0**k)*(-1)**k for k in range(1, len(coeffs))])
        +
          coeffs[0]*np.log(-z0))
    inner[1:] = [(1/z0**l)*sum([(coeffs[k]/z0**k)*binom(l+k-1, k-1)*(-1)**k
                for k in range(1, len(coeffs))]) - coeffs[0]/((z0**l)*l)
                for l in range(1, len(coeffs))]
    return inner
```

[4] Far enough away means at a distance greater than or equal to the radius of the circle that holds the particles included in the multipole expansion. This is an essential part of Lemma 3. The proof for the convergence of the shifted expansion is laid out in GR

Along the same lines as in the upward pass, in the downward pass the inner expansions are computed for all cells starting with the coarsest level (the children of the root node) from their corresponding interaction sets. Afterwards, the inner for deeper children are computed recursively and their parents inner expansion shifted by means of Lemma 4 are added in order to account for particles *farther away* meaning those farther than their interaction sets. I implemented this as part the function `_inner`, and using the function `_shift_texp` to shift inner expansions. After the dust clears, up to this step all cells will have an inner expansion accounting for the potential of all particles excluding their nearest neighbors and interaction sets—those which aren't *far enough away*.

```
def _shift_texp(coeffs, z0):
    """Shift inner expasions (Taylor) to new center"""
    shift = np.empty_like(coeffs)
    shift = [sum([coeffs[k]*binom(k,l)*(-z0)**(k-l)
             for k in range(l,len(coeffs))])
             for l in range(len(coeffs))]
    return shift


def _inner(tnode):
    """Compute the inner expansions for all cells recursively and potential
    for all particles"""

    z0 = complex(*tnode.parent.center) - complex(*tnode.center) # check sign
    tnode.inner = _shift_texp(tnode.parent.inner, z0)
    for tin in tnode.interaction_set():
        z0 = complex(*tin.center) - complex(*tnode.center)
        tnode.inner += _convert_oi(tin.outer, z0)

    if tnode.is_leaf():
        # Compute potential due to all far enough particles
        z0, coeffs = complex(*tnode.center), tnode.inner
        for p in tnode.get_points():
            z = complex(*p.pos)
            #p.phi += np.real(sum([a*(z-z0)**k for k, a in enumerate(coeffs)]))
            p.phi -= np.real(np.polyval(coeffs[::-1], z-z0))
        # Compute potential directly from particles in interaction set
        for nn in tnode.nearest_neighbors:
            potentialDDS(tnode.get_points(), nn.get_points())

        # Compute all-to-all potential from all particles in leaf cell
        _ = potentialDS(tnode.get_points())
    else:
        for child in tnode:
            _inner(child)
```

1.3.3 Potential Evaluation

The final step in the FMM involves computing the interactions of all particles in all leaf cells with all other particles in their nearest neighbor and their interaction set cells. In addition the interactions among all particles contained in the same leaf cell must also be computed. These interactions are done directly with the double summation and the pair interaction potential in equation 1.1. For each particle, the inner expansion is evaluated at its location and added to the direct interactions computed to give the final value of the potential from all other particles in the domain. This step is done inside the `_inner`.

1.3.4 Main FMM function

The main fmm function `potential` takes a list of `class Particle` instances and simply calls the previously described helper functions in the correct order to complete the full FMM.

```
def potential(particles, bbox=None, tree_thresh=None, nterms=5, boundary='wall'
    ):
    """Fast Mulitipole Method evaluation of all-to-all potential"""

    tree = build_tree(particles, tree_thresh, bbox=bbox, boundary=boundary)
    _outer_mpexp(tree.root, nterms)
    tree.root.inner = np.zeros((nterms + 1), dtype=complex)
    any(_inner(child) for child in tree.root)
```

2. Some results regarding complexity, scaling, and accuracy

The whole idea of the FMM is to reduce the time complexity needed to compute pair potentials for many particle systems. Lets have a look at how the FMM reduces the amount of work, and if it can do so with enough accuracy.

2.1 Complexity and scaling

2.1.1 Time scaling

Figure 4 shows the time complexity of the direct double sum calculation and that of the FMM. It clear that for any formidable sized system (< 100 particles), the FMM is worth using, and quickly becomes necessary for large systems. A note here is that altough the complexity of the FMM is approximately linear, the total runtime depends on the number of terms and quadtree grid refinement, hence for small systems the FMM with high refinement and many terms is slower than a direct calculation—in other words it is overkill.

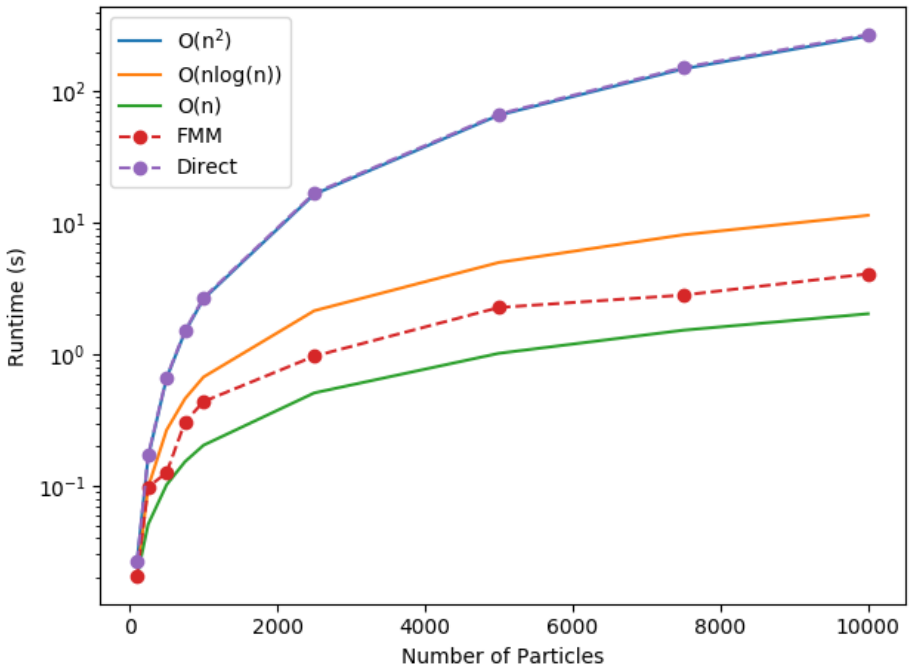


Figure 4: Time complexity scaling for FMM and direct calculation of potential.

2.2 Accuracy

Apart from reducing the amount of work, the FMM should have good accuracy. And for enough terms, such that the error is of the order of the machine precision, the FMM and the direct calculation are just as accurate[5]. Figure 5 shows the results for a calculation with 100 particles and 10 terms in the expansion. With 10 terms the results are almost indistinguishable. Furthermore, as can be seen in Figure 6 the decrease in error drops substantially even after 3 terms, and at about 30 terms the error starts reaching floating point precision.

3. Concluding remarks

The FMM provides a very clever and efficient means to improve the time complexity of n-body all-to-all interactions. In the current effort, I implemented the FMM for a Coulomb potential for electrostatics, however n-body interactions are ubiquitous in many other systems. My current implementation still needs work to get the full scaling and accuracy results of the original method, in addition to optimal run-time performance. Furthermore, additional code is required to extend it in order compute forces and update particle positions to use the code for particles simulations. Nevertheless, it is a good start,

[5]The FMM using an infinite number of terms is technically exact.

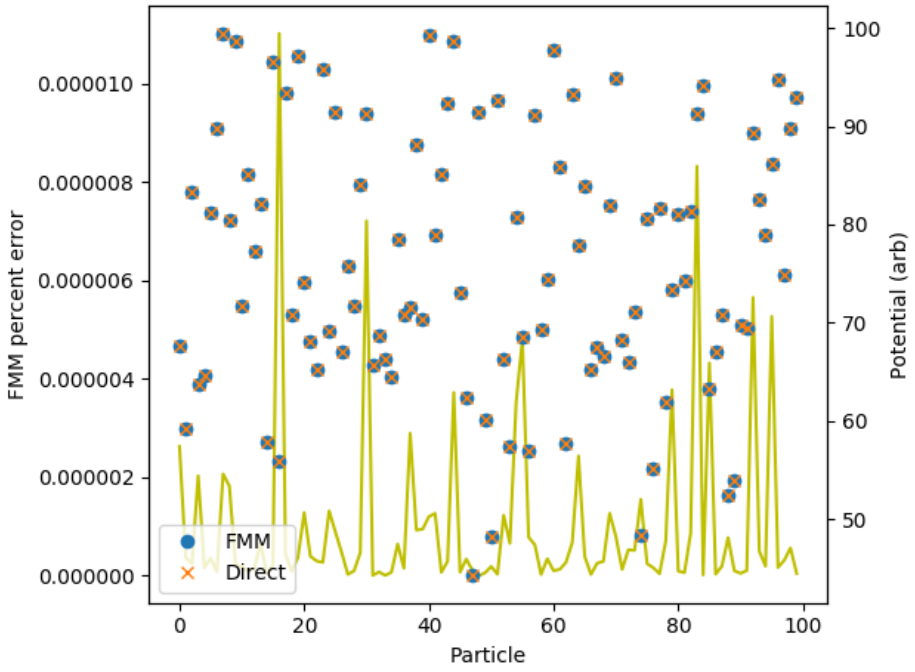
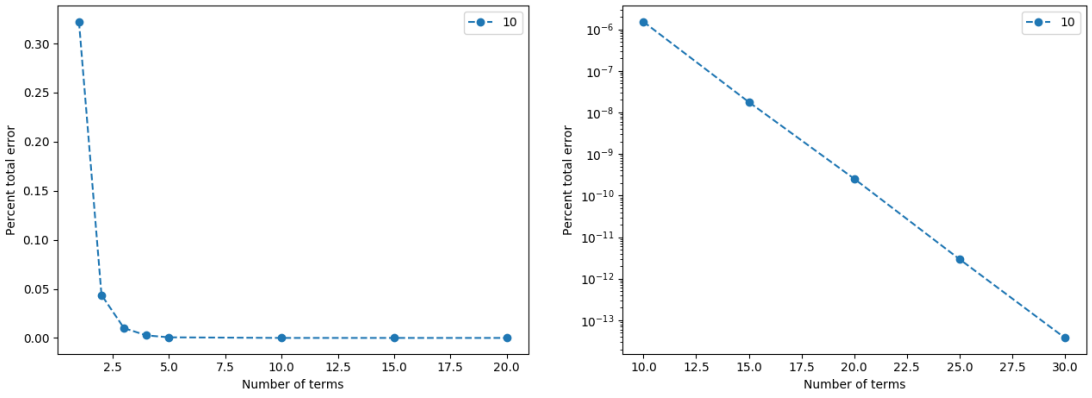


Figure 5: Percent error of FMM vs direct calculation for potential of each particle



(a) (b)

Figure 6: Error of FMM vs direct for different number of terms in multipole expansion.

and I may eventually extend this for fun...

Finally, although I did not directly include any external libraries for the quadtree data structure and FMM calculations, I did get ample insight and inspiration from [2, 3, 4].

Note: For further reference, I have uploaded the full source code to GitHub.

References

[1] Greengard, L., & V. Rokhlin. "A Fast Algorithm for Particle Simulations." Journal of Computational Physics, vol. 135, no. 2, 1987, pp. 280-292., doi:10.1006/jcph.1997.5706.

[2] James Demmel, CS267 Lecture Slides (2015) https://people.eecs.berkeley.edu/~demmel/cs267_Spr15/Lectures/lecture21_NBody_jwd15_4pp.pdf

[3] Lorena Barba Group, FMM Tutorial, (2016), GitHub Repository, https://github.com/barbagroup/FMM_tutorial

[4] Karim Bahgat, Pyqtree, (2015), GitHub Repository. https://github.com/karimbahgat/Pyqtree