NUMERICAL PHYSICS

Many body interactions - FMM Algorithm

September 2024 - December 2024

ÉCOLE POLYTECHNIQUE

INSTITUT POLYTECHNIQUE DE PARIS

TABLE OF CONTENTS

INTRODUCTION

Computing the interactions in many-body systems is a common problem in various fields of physics (astrophysics and molecular physics for instance). As such, many mathematicians and physicists have been struggling with it for centuries. Analytical solutions do not exist in general when there are at least 3 bodies, which raises the interest of numerical simulations in order to describe more complex phenomena.

However, for an N-body system with pair-wise interactions, the most natural algorithm leads to a time complexity in $O(N^2)$ for updating the system once by an elementary time interval. This is, even with the recent progress in computational performance, detrimental for simulating large scale systems. However, there exists computation simplifications that could allow us to get a better complexity.

The main goal of this project is to understand and implement the Fast Multipole Method (FMM) as described in [1], [2], which yields an $O(N)$ time complexity for the same problem. We will first describe the algorithm before detailing the way we implemented it. Lastly, we will analyze our results and discuss possible improvements for our implementation.

1
ALGORITHMS FOR SOLVING CLASSICAL N-BODY PROBLEMS

1.1 - Generalities

Let us consider an N-body system (with $N \in \mathbb{N}^*$) within a cube of size R^3 where each pair of distinct bodies A and B interact with each other through gravity :

$$\vec{F}_{\mathrm{B}\to\mathrm{A}} = -\vec{F}_{\mathrm{A}\to\mathrm{B}} = Gm_{\mathrm{A}}m_{\mathrm{B}}\frac{\overrightarrow{\mathrm{AB}}}{\|\overrightarrow{\mathrm{AB}}\|^3} \tag{1}$$

Given a time interval $\mathrm{d}t$, the positions $(\vec{x}_1, ..., \vec{x}_N)$ and $(\vec{x}_1^-, ..., \vec{x}_N^-)$ respectively at time t and $t - \mathrm{d}t$, and masses $(m_1, ..., m_N)$ of the particles, we want to know their positions $(\vec{x}_1^+, ..., \vec{x}_N^+)$ at $t + \mathrm{d}t$. This is equivalent to finding, for each particle $\vec{x}_k$, the total force $\vec{F}_k = \sum_{1\le i\le n, i\neq k} \vec{F}_{i\to k}$. Indeed, using the Verlet integration scheme, the following ansatz provides a relatively good $O(\mathrm{d}t^4)$ accuracy :

$$\forall k \in [\![1, n]\!], \vec{x}_k^+ = 2\vec{x}_k - \vec{x}_k^- + \frac{\vec{F}_k}{m_k}\mathrm{d}t^2 \tag{2}$$

We will describe in this part the naive and the FMM algorithms to compute $\vec{F}_k$. Note that we can introduce the gravitational potential Φ such that $\vec{F}_{\mathrm{A}} = -m_{\mathrm{A}}\vec{\nabla}\Phi$ with $\Phi = \sum_{B\neq A} m_{\mathrm{B}}\frac{G}{\mathrm{AB}}$.

An issue with this setup is that the force and the potential diverge when A is close to B. In order to avoid numerical instabilities, we will smooth the potential function. Each particle, instead of being represented with a Dirac δ function in the mass density, will come as a "blob" of mass with a density shape function η of unit integral and a size ε, replacing $\delta(r)$ with $\frac{1}{\varepsilon}\eta\left(\frac{r}{\varepsilon}\right)$. This allows us to write its exerted gravitational potential as

$$\phi(\vec{r}) = -\frac{Gm}{\varepsilon}\varphi\left(\frac{r}{\varepsilon}\right) \tag{3}$$

ε is typically chosen as $\frac{4R}{\sqrt{N}}$ [1]. Then, we can obtain φ with Gauss' gravitational theorem:

$$4\pi\eta(\xi) = -\frac{1}{\xi^2}\frac{\mathrm{d}}{\mathrm{d}\xi}\left(\xi^2\frac{\mathrm{d}\varphi}{\mathrm{d}\xi}\right) \tag{4}$$

We choose here the Plummer smoothing: $\eta(\xi) = \frac{3}{4\pi}(1+\xi^2)^{-5/2}, \varphi(\xi) = (1+\xi^2)^{-1/2}$.

1.2 - The naive algorithm

The very simplest algorithm to compute $\vec{F}_k$ is the following:

Algorithm 1: Naive method to compute $\vec{F}_k$

```
1 F⃗ := 0⃗
2 for i in [[1, N]] do
3     if i = k
4         continue
5     end
6     F⃗ ← F⃗ + F⃗_{i→k}
7 end
8 return F⃗
```

Although it yields the most correct possible result for the force, Algorithm 1 is clearly executed in $O(N)$ operations. Running it for every particle is therefore a quadratic complexity algorithm, and thus not very efficient. We will use it for performance comparison and to test the accuracy of subsequent methods.

1.3 - The FMM algorithm

> 💡 **Idea**
>
> When one wants to model interactions between several galaxies, he or she will consider the objects as wholes, and not the billions of individual stars in them. This is because, if the masses are far enough, approximating their joint gravitational field with the field produced by a single point, at the center of mass of the ensemble and carrying all the mass, yields fairly good results.

The FMM algorithm generalizes this concept to approximate $\vec{F}_k$. The first step is to break our $R \times R \times R$ cube into a tree of sub-cubes, each parent cell containing $2^3 = 8$ children in 3D[1] (this becomes 4 in 2D). The particle k is then located in a certain leaf cell C. Then, given a particle i in a cell C', the idea is to compute the interaction between the centers of mass $\vec{z}_C$ and $\vec{z}_{C'}$, and to perform a Taylor expansion of the forces around the centers to approximate $\vec{F}_{i\to k}$. In fact, we not only get $\vec{F}_{i\to k}$ but the entire $\sum_{j\in C'} \vec{F}_{j\to k}$ vector in one go, which will save us a lot of work.

What's more, instead of taking the leaf cell containing i, if i and k are far enough, we can take an ancestor of C' to compute a contribution to $\vec{F}_{\vec{z}_C}$ without a huge loss in accuracy, and thus save even more computation workload.

[1] The figures below will represent the tree in a 2D space, for the sake of readability.

1.3.1 • Tree structure

To construct the tree, we will choose a depth d for its cells. That is, the tree will contain d floors, the first floor only holding the root (the entire simulation volume), and the $(n+1)$-th floor holding the 8^n children of the 8^{n-1} cells in the n-th floor. We thus have 8^{d-1} leave cells at the bottom, sharing the particles.

In order to construct the tree, we also could have defined a maximum number of particles per leaf cell, and subdivide a cell recursively as long as it contains too many particles. However, using a fixed depth for the tree, as introduced in [2], makes the structure constant in time and easier to represent in the code.

Each cell will be assigned a list of *direct neighbors*, the first layer of 27 cells around (the cell itself is included in this list), and an *interaction list*, all the cells of the same floor that are not direct neighbors and whose parents are direct neighbors of the current cell's parent (see Figure 1).

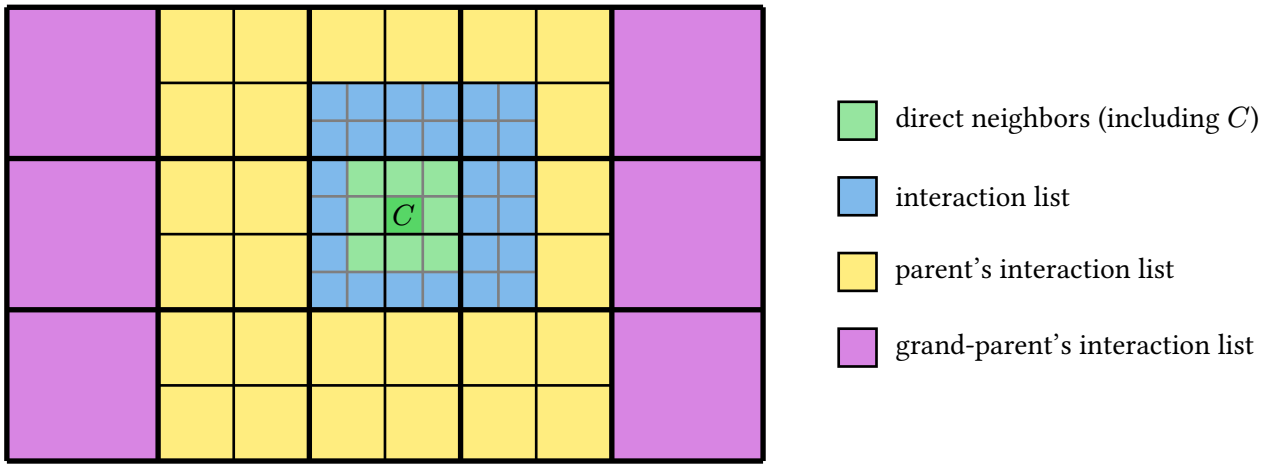


Figure 1: Representation of the interaction lists and direct neighbors for a cell C.

These lists are built once and for all at the beginning of the execution, in a single downward pass in the tree. This adds a $O(8^d)$ time complexity overhead for the first update only.

Then, the leaves hold an additional list of samples that they contain, which is updated at every step, and therefore contributes to the total time complexity. However, this only requires a single pass in the list of particles, and is therefore done in $O(N)$ operations in total, or $O(1)$ per particle.

With this setup, most cells are not "known" to C. Actually, it can be easily shown (as illustrated by Figure 1) that for every cell C' of C's floor that is not in C's interaction list or direct neighbors, there are some ancestors A of C and A' of C' on the same floor such that A' is in the interaction list of A (and reciprocally). This will allow us to later compute the field of A' onto A and propagate it to A's children, including C.

1.3.2 • Computing the forces

1.3.2.1 - Theory

Most results and notations used here are inspired by [1], although our final implementation might differ from the one detailed in it.

In the following equations, we introduce an "order vector" notation $\vec{n} \in \mathbb{N}^3$ to represent the order of a term along the three dimensions of space in Taylor expansions. In particular, for any real vector $\vec{u} \in \mathbb{R}^3$ and any function $f : \mathbb{R}^3 \to \mathbb{R}$ of class $\mathcal{C}^{\|\vec{n}\|}$,

$$\|\vec{n}\| = n_1 + n_2 + n_3 \qquad \vec{n}! = n_1! n_2! n_3! \qquad \vec{u}^{\vec{n}} = u_1^{n_1} u_2^{n_2} u_3^{n_3} \qquad \nabla^{\vec{n}} f = \frac{\partial^{\|\vec{n}\|} f}{\partial^{n_1} x_1 \partial^{n_2} x_2 \partial^{n_3} x_3} \tag{5}$$

> ⚠ **Defining the gravitational potential**
>
> In this part, we denote ϕ the gravitational potential and φ the potential per unit mass (basically, this new $\varphi(\vec{r})$ corresponds to the older $-\frac{G}{\varepsilon}\varphi\left(\frac{r}{\varepsilon}\right)$ in Equation 3).

Let us consider a particle $\vec{x}_a$ in a cell A with its center of mass $\vec{z}_\mathrm{A}$, and a particle $\vec{x}_b$ in a cell B with its center of mass $\vec{z}_\mathrm{B}$. We can expand the potential exerted by a onto b as

$$\phi(\vec{x}_b - \vec{x}_a) \approx \sum_{\|\vec{n}\| \leq p} (-1)^{\|\vec{n}\|} \frac{(\vec{x}_a - \vec{z}_\mathrm{A})^{\vec{n}}}{\vec{n}!} m_a \nabla^{\vec{n}} \varphi(\vec{x}_b - \vec{z}_\mathrm{A}) \tag{6}$$

There, we have one expansion with respect to $\vec{x}_a - \vec{z}_\mathrm{A}$. We can expand it further with respect to $\vec{x}_b - \vec{z}_\mathrm{B}$ to involve the cell-wise interaction:

$$\phi(\vec{x}_b - \vec{x}_a) \approx \sum_{\|\vec{n}\|+\|\vec{m}\| \leq p} (-1)^{\|\vec{n}\|} \frac{(\vec{x}_a - \vec{z}_\mathrm{A})^{\vec{n}} (\vec{x}_b - \vec{z}_\mathrm{B})^{\vec{m}}}{\vec{n}!\vec{m}!} m_a \nabla^{\vec{n}+\vec{m}} \varphi(\vec{z}_\mathrm{B} - \vec{z}_\mathrm{A}) \tag{7}$$

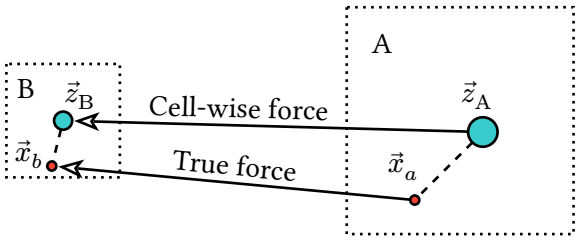


Figure 2: Illustration of the expansion of the force exterted by a onto b. The "true force" is computed by expanding the "cell-wise force" with respect to $\vec{x}_b - \vec{z}_\mathrm{B}$ and $\vec{x}_a - \vec{z}_\mathrm{A}$.

Then, we get the total potential estimation exerted by cell A on b by summing over a:

$$\Phi_{\mathrm{A}\to\mathrm{B}}(\vec{x}_b) = \sum_{a \in \mathrm{A}} \sum_{\|\vec{m}\| \leq p} \sum_{\|\vec{n}\| \leq p - \|\vec{m}\|} (-1)^{\|\vec{n}\|} \frac{(\vec{x}_a - \vec{z}_\mathrm{A})^{\vec{n}} (\vec{x}_b - \vec{z}_\mathrm{B})^{\vec{m}}}{\vec{n}!\vec{m}!} m_a \nabla^{\vec{n}+\vec{m}} \varphi(\vec{z}_\mathrm{B} - \vec{z}_\mathrm{A}) \tag{8}$$

Now, let us see how we can "prepare" this computation for every pair of particles by computing cell-wise interaction terms. First, organize the three Σ signs in $\Phi_{\mathrm{A}\to\mathrm{B}}$ to factorize the expression in a convenient form:

$$\Phi_{\mathrm{A}\to\mathrm{B}}(\vec{x}_b) = \sum_{\|\vec{m}\| \leq p} \frac{(\vec{x}_b - \vec{z}_\mathrm{B})^{\vec{m}}}{\vec{m}!} \underbrace{\sum_{\|\vec{n}\| \leq p - \|\vec{m}\|} \nabla^{\vec{n}+\vec{m}} \varphi(\vec{z}_\mathrm{B} - \vec{z}_\mathrm{A}) \underbrace{\frac{(-1)^{\|\vec{n}\|}}{\vec{n}!} \sum_{a \in \mathrm{A}} m_a (\vec{x}_a - \vec{z}_\mathrm{A})^{\vec{n}}}_{M_{\vec{n}}(\mathrm{A})}}_{F_{\vec{m}}(\mathrm{A}\to\mathrm{B})} \tag{9}$$

In Equation 9, we introduce the *multipoles* $M_{\vec{n}}(\mathrm{A})$ which only depend on the cell A, and the *field tensor* contributions $F_{\vec{m}}(\mathrm{A} \to \mathrm{B})$, representing the cell-wise interaction. We can also define the total field tensor $F_{\vec{m}}(\mathrm{B}) = \sum_{\mathrm{A\ in\ pertinent\ cells}} F_{\vec{m}}(\mathrm{A} \to \mathrm{B})$, where a pertinent cell is either in the interaction list or in the interaction list of an ancestor. Thus, the total potential from pertinent cells comes as

$$\Phi_B(\vec{x}_b) = \sum_{\|\vec{m}\| \leq p} \frac{(\vec{x}_b - \vec{z}_\mathrm{B})^{\vec{m}}}{\vec{m}!} F_{\vec{m}}(\mathrm{B}) \tag{10}$$

Now, consider B is a leaf cell. The total gravitational field to which b is subjected is the sum of $\vec{\mathcal{G}}_{\mathrm{B},f}(\vec{x}_b) = -\vec{\nabla}\Phi_B(\vec{x}_b)$ (*far-field*) and of the fields $\vec{\mathcal{G}}_{\mathrm{B},c}(\vec{x}_b)$ created by other particles in B's direct neighbors (*close-field*).

1.3.2.2 - In the algorithm

> **Expansion order**
>
> In this work, we only consider the cases $p = 1$ and $p = 2$. While this means order 1 and 2 expansions for the potential, the actual field (which is what we are interested in), is respectively expanded to order 0 and 1.

Note that $M_\mathrm{A} = M_{\vec{0}}(\mathrm{A})$ is simply the mass of the cell, i.e. the sum of the mass of its particles. Also, by definition of the center of mass, $\|\vec{n}\| = 1 \Rightarrow M_{\vec{n}}(\mathrm{A}) = 0$. Starting from this, we can give a definite expression for $\vec{\mathcal{G}}_{\mathrm{A}\to\mathrm{B},f}(\vec{x}_b)$, that is,

$$\vec{\mathcal{G}}_{\mathrm{A}\to\mathrm{B},f}(\vec{x}_b) = -M_\mathrm{A}\left(\underbrace{\vec{\nabla}\varphi(\vec{z}_\mathrm{B} - \vec{z}_\mathrm{A})}_{\text{order 0 term}} + \underbrace{\underline{\underline{H}}_\varphi(\vec{z}_\mathrm{B} - \vec{z}_\mathrm{A}) \cdot (\vec{x}_b - \vec{z}_\mathrm{B})}_{\text{order 1 term (ignored if } p=1)}\right) \tag{11}$$

Where $\underline{\underline{H}}_\varphi$ is the Hessian matrix of φ (also the opposite of the Jacobian of the gravitational field per unit mass).

Computing the far field will simply consist in, for each cell, holding its sole multipole (i.e. its mass), its center of mass and its full field tensor (the sum $\vec{\mathcal{G}}_0(\mathrm{B})$ over A of $-M_\mathrm{A}\vec{\nabla}\varphi(\vec{z}_\mathrm{B} - \vec{z}_\mathrm{A})$ and $\underline{\underline{\mathcal{G}}}_1(\mathrm{B})$ of $-M_\mathrm{A}\underline{\underline{H}}_\varphi(\vec{z}_\mathrm{B} - \vec{z}_\mathrm{A})$). Then, for a given particle, the computation of the far-field will go in $O(1)$ time!

Now, "holding" the multipoles and field tensors involves updating them at every $\mathrm{d}t$.

- The mass and center of mass of the leave cells are computed directly by summing the masses of the contained particles. Then, in an upward pass, the mass and the center of mass of a parent cell is computed by summing the masses of its children[2], and the center of mass is obtained with a weighted average of the children's centers. The time complexity is therefore $O\left(8^d + N\right)$.
- After computing the multipoles and centers of mass, each cell stores the sum of the field tensors exerted in its interaction list. Then, in a downward propagation, we can shift the parent's field tensor to the children. In our case, this shift happens as such:

[2] In fact, higher order multipoles can also be propagated upward with a little more analytical effort [1].

$$\begin{aligned}\vec{\mathcal{G}}_0(\text{Child}) &\leftarrow \vec{\mathcal{G}}_0(\text{Child}) + \vec{\mathcal{G}}_0(\text{Parent}) + \underbrace{\underline{\underline{\mathcal{G}}}_1(\text{Parent}) \cdot (\vec{z}_{\text{Child}} - \vec{z}_{\text{Parent}})}_{\text{Ignored if } p=1} \\ \underline{\underline{\mathcal{G}}}_1(\text{Child}) &\leftarrow \underline{\underline{\mathcal{G}}}_1(\text{Child}) + \underline{\underline{\mathcal{G}}}_1(\text{Parent})\end{aligned} \tag{12}$$

In total, we need an additional $O\left(N + 8^d\right)$ for the whole update, or $O\left(1 + \frac{8^d}{N}\right)$ per particle on average.

1.3.3 • Complete algorithm

As described in the previous sections, Algorithm 2 needs $O(1)$ operations on average per particle and Algorithm 3 needs $O\left(1 + \frac{8^d}{N}\right)$. While the far-field is clearly computed in $O(1)$ operations in Algorithm 4, the close-field involves a local usage of Algorithm 1, and therefore requires (for uniformly distributed particles) $O\left(\frac{N}{N_{\text{lc}}}\right)$ operations, where $N_{\text{lc}} = 8^{d-1}$ is the number of leave cells. In the end, $\vec{a}_k$ needs $O\left(\frac{N}{8^d} + \frac{8^d}{N}\right)$ operations to be computed. This last formula shows that a balance is to be found between d and N to optimize the performance.

If we choose $d = \lfloor \log_8 N \rceil$, this becomes $O(1)$ operations!

Algorithm 2: Populate cells and compute multipoles

```
!> Requires a tree built as described in Section 1.3.1
! Populate the leave cells
foreach leaf-cell do
    clear leaf-cell::particles
end
for i in [[1, N]] do
    locate the leaf-cell to which x_i belongs
    leaf-cell::particles::push(particle i)
end
! Compute the mass and center of mass of the leaves
foreach leaf-cell do
    leaf-cell::mass ← sum(particle::mass for particle in leaf-cell::particles)
    leaf-cell::CoM ←
        sum(particle::position×particle::mass for particle in leaf-cell::particles) / leaf-cell::mass
end
! Propagate upward
foreach tree-floor upward except lowest-floor do
    foreach cell in tree-floor do
        cell::mass ← sum(child::mass for child in cell::children)
        cell::CoM ← sum(child::CoM×child::mass for child in cell::children) / cell::mass
    end
end
```

Algorithm 3: Compute field tensors

```
!> Requires multipoles and centers of mass to be set
! Compute the field tensors from interaction lists
foreach cell in tree do
    cell::field-tensor ← sum(contributions from close-cell for close-cell in cell::interaction-list)
end
! Propagate downward
foreach tree-floor downward except root do
    foreach cell in tree-floor do
        find parent-cell
        cell::field-tensor::order0 ←
            cell::field-tensor::order0 + parent-cell::field-tensor::order0 +
            parent-cell::field-tensor::order1 · (cell::CoM - parent-cell::CoM)
        cell::field-tensor::order1 ← cell::field-tensor::order1 + parent-cell::field-tensor::order1
    end
end
```

Algorithm 4: Fast multipole method to compute $\vec{a}_k = \frac{\vec{F}_k}{m_k}$

```
!> Requires tensor fields to be computed
find cell containing particle k
a⃗_k := 0⃗
! Compute the close-field
foreach neighbor in cell::direct-neighbors do
    foreach particle in neighbor do
        if particle = particle k do
            continue
        end
        a⃗_k ← a⃗_k + G⃗(x⃗_i − particle::position)
    end
end
! Compute the far-field
a⃗_k ← a⃗_k + cell::field-tensor::order0 + cell::field-tensor::order1 · (x⃗_k − cell::CoM)
return a⃗_k
```

Conclusion

With the FMM algorithm (Algorithm 2, Algorithm 3, Algorithm 4), an update is done in $O(N)$ operations, instead of $O(N^2)$ for the naive algorithm.

2 ALGORITHM IMPLEMENTATION

2.1 - FMM algorithm implementation

2.1.1 • Representing the gravitational field

Taking back the notations of Section 1.1, the potential between two particles per unit mass writes as

$$\phi(\vec{r}) = -\frac{G}{\varepsilon}\varphi\left(\frac{r}{\varepsilon}\right) \qquad \varphi(\xi) = (1+\xi^2)^{-1/2} \tag{13}$$

Using $\vec{\xi} = \frac{\vec{r}}{\varepsilon}$, we obtain

$$\begin{aligned}
\phi(\vec{r}) &= -\frac{G}{r}\underbrace{\frac{\xi}{\sqrt{1+\xi^2}}}_{\text{Function phi}} \\
-\vec{\nabla}\phi(\vec{r}) &= \frac{G}{r^2} \times \underbrace{\left(-\frac{\xi^2}{(1+\xi^2)^{3/2}}\vec{\xi}\right)}_{\text{Function grad_phi}} \\
-\underline{\underline{H}}_{\phi}(\vec{r})_{i,j} &= \underbrace{\frac{G}{r^3}\xi^3\left(\frac{3\xi_i\xi_j}{(1+\xi^2)^{5/2}} - \frac{\delta_{ij}}{(1+\xi^2)^{3/2}}\right)}_{\text{Function hess_phi}}
\end{aligned} \tag{14}$$

In the equation above, we chose to pass the `phi`, `grad_phi` and `hess_phi` functions to the FMM algorithm as functions of $\vec{\xi}$ with a remaining factor only depending on G and r to have the actual field, hence the strange decomposition made above.

2.1.2 • Writing the code

After discussing ways to speed up the computation of the forces in an N-body system in Section 1, we will see how these ideas were translated into code.

Two implementations are proposed: one in Python (used for designing the method), and one in C++ (a faster version making more tests feasible). The codes in the two languages are written to be transparent one to another. A certain number of classes are defined in the code, namely:

1. A `MassSample` class, representing a particle in our space. Its fields are:
   - `position` (or `pos`) for the position of the particle.
   - `prev_pos` for the previous position.
   - `mass`.
2. An `FMMCell` class for a cell in the tree used in the FMM algorithm. Each cell is a cube described with the following fields:
   - `centroid` for the center of the cube.
   - `size` of the cube.
   - `interaction_list` as described in Section 1.3.1, to be computed when the tree is built.
   - `direct_neighbors`, same.
   - `samples`, the list of `MassSample`'s contained in the cell, updated with the tree at each simulation step. Always empty for non-leaf cells.
   - `barycenter` of the samples, updated with the tree.
   - the total `mass`, updated with the tree.
   - the `field_tensor`, updated with the tree. This consists of a $\mathbb{R}^3$ vector and a $\mathcal{M}_3(\mathbb{R})$ matrix for the 0 and 1-order terms of Equation 11.
3. An `FMMTree` class implementing the structure developed in Section 1.3.1. It has a unique field `data`, basically a list (`List` in Python, `std::vector` in C++) of 3D arrays of `FMMCell`'s (defined as a `List[List[List[FMMCell]]]` in Python, and a `boost::multi_array<FMMCell, 3>` in C++). The root is located at index 0, the leaves are on the last element of the list. The structure is constructed once and for all when the constructor of the class is called.
4. An `FMMSolver`, with the following fields:
   - `size` of the simulation volume.
   - the timestep `dt`.
   - the functions `phi`, `grad_phi` and `hess_phi` as in Equation 14. Note that `hess_phi` is eventually the zero function, resorting to the order-0 method.
   - the `FMMTree` instance named `tree`. Its constructor is called when `FMMSolver` is constructed.
   - the particles in a `samples` list.
   - the size `epsilon` of the smoothed potential.
   - the gravitational constant `G`.
5. A `NaiveSolver`, slightly different from `FMMSolver`, to use Algorithm 1 for speed and accuracy comparison tests.

The `FMMSolver` and `NaiveSolver` classes provide an `update` method to implement the simulation algorithms described above. We can now test simulations with varying number of particles and initial conditions, with the naive solver and the order 0 and 1 FMM solvers. Note that we can vary the depth of the FMM tree as well to see if $\lfloor \log_8(N) \rceil$ is indeed the right solution.

2.2 - Visualization

In order to do so, we first used python, as its simplicity and adaptability allows not only to compute our problem, but also to visualize the problem in 3 dimensions by using the VPython module.

2.2.1 • Using VPython

First of all, as VPython requires some time to generate frames of an animation, if we wanted to have a smooth enough animation, we had to create it in two steps. First we used the solver we defined previously and saved the positions of all particles at each step. Then only we generate the frames from the point clouds we saved.

However, we still had to tackle some other issues, as some of VPython classes can't be changed dynamically (at least not completely). As cloud points are not dynamically movable, we had, in order to set up animations, to save all positions and display one point cloud at a time. As a result, the longer the animation (i.e., the more frames it contains), the more data must be stored to display it.

In spite of these limitations, we managed to set up animations to represent the way our particles move.

2.2.2 • Simulation algorithm

In order to represent the different objects of our simulation, we used the following shapes/objects from VPython:

- Particles are represented as white spheres (point cloud).
- The cells of the tree are represented as blue translucent boxes.

By using boxes slightly smaller than the true cells, we were able to represent them without having to display separately all of the interfaces between cells.

When it comes to the simulation we made, we first set up N_{sample} samples with a uniform random distribution along our simulation space. However, we noticed that the movement then, as we defined no particular initial speed, was consisting in a mere linear oscillation of the samples close to the mass center of the system. As we wanted to display more complex movements, we decided to modify our initial conditions.

- First of all, we would place a high-mass sample (described in the implementation as a sun) at the center of our simulation volume.
- Then, in order to initiate a rotation movement of our particles around the "sun", we added, for all particles, an initial speed such that $\vec{v}_0 \propto \vec{e} \times \vec{x}_0$, $\vec{x}_0$ and $\vec{v}_0$ being the initial position and speed of the sample, and $\vec{e}$ being a unit vector in one of the directions of the simulation space.
- Finally, we added the possibility to set up random grouped clusters instead of samples.

3
RESULTS

3.1 - Performance comparison

Using the C++ implementation, several tests were carried out to see the change in execution time against the number of particles (which are placed randomly on the cube without initial speed, with identical masses), as well as other parameters. The results are presented in Figure 3. As expected, the FMM solvers quickly become more efficient than the naive solver.

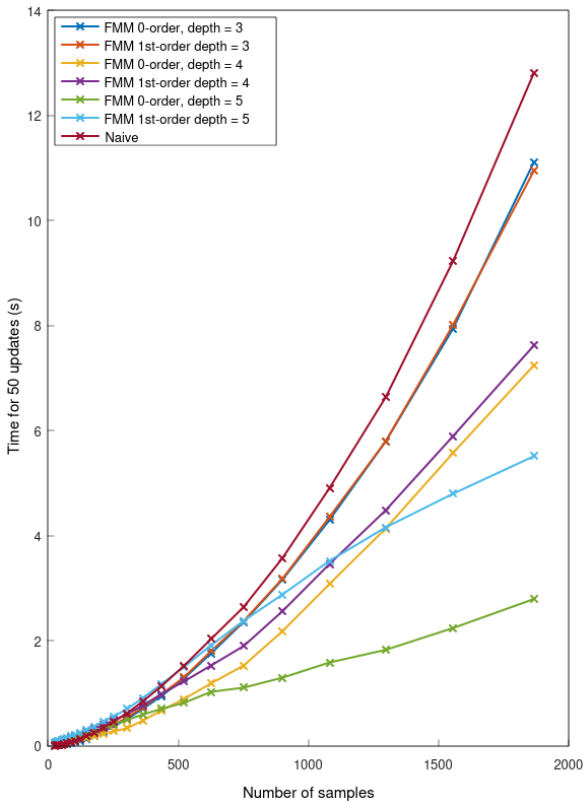
We can see on Figure 3b that $d = 3$ becomes preferable for $N \sim 60 \sim 8^2$. Then, $d = 4$ seems to win as soon as $N \sim 200$ which is close to the threshold between $\lfloor \log_8(N) \rceil = 2$ and $\lfloor \log_8(N) \rceil = 3$. This seems to tell that the best choice for d is $1 + \lfloor \log_8(N) \rceil$, which is close to what we anticipated in Section 1.3.3. Then, depending on the order of the solver, $d = 5$ starts winning the competition with N between 500 and 1500, knowing that $\lfloor \log_8(N) \rceil$ becomes 4 when $N \sim 1500$.

Now, let us focus on the shape of the curves, particularly on the slope of the log-scale curves. For the FMM solvers, we can observe a change in slope on Figure 3c, where a slower slope becomes a faster one. The most striking evidence is the curves for $d = 4$, which start parallel to $d = 5$ and slower than the naive solver, and in the end just have the same slope as the naive solver.

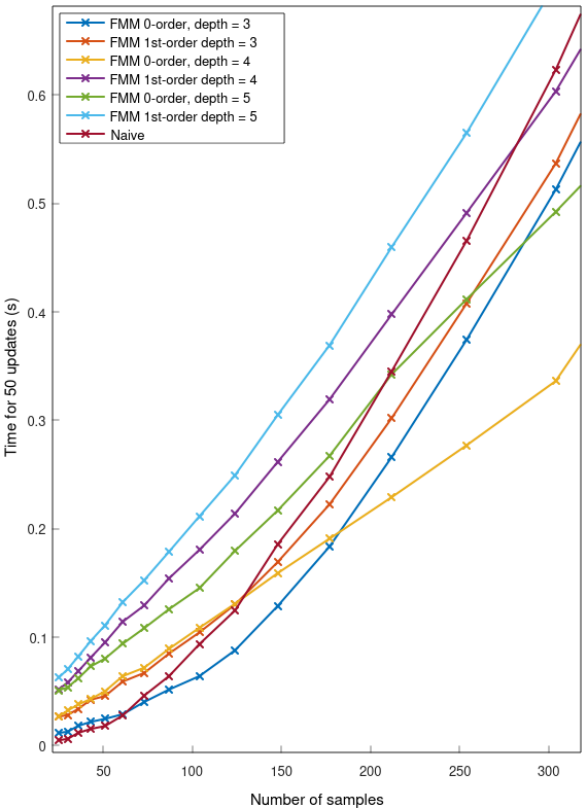
This seems to tell that the close-field complexity term identified in Section 1.3.3 raises the global complexity from $O(N)$ to $O(N^2)$, while the higher depth solver remain linear in complexity. Nevertheless, the other term $(\frac{8^d}{N})$ seems to slow down the computations on low values of N, although the solver finally catches up with the others as N rises.

Takeaways

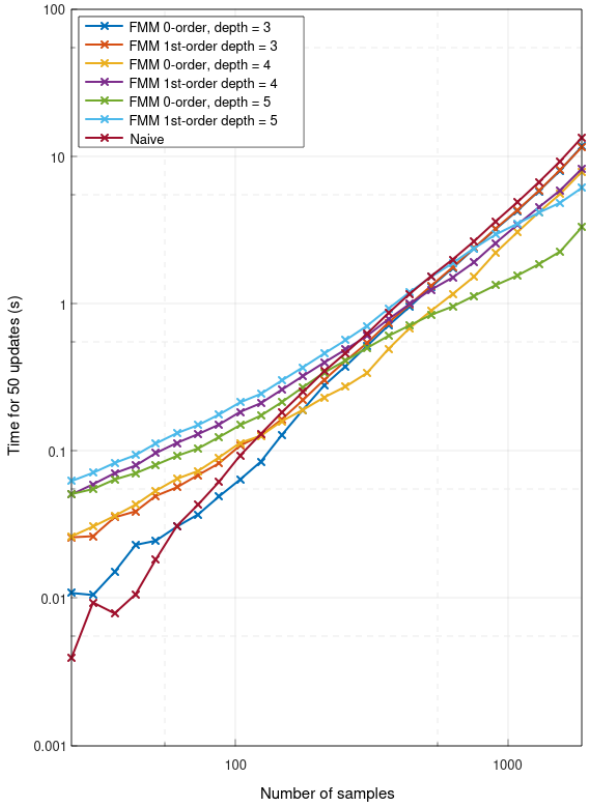
- While it is unclear which depth exactly should be chosen, the prediction of taking d around $\lfloor \log_8(N) \rceil$ seems to hold for $N \in [\![2, 2000]\!]$.
- Most features of the theoretical complexity are visible in these curves. Most importantly, when d and N are synchronized, the complexity is indeed linear.
- Each simulation was done on a single core, many parts of Algorithm 2, Algorithm 3 and Algorithm 4 are parallelizable, as they consist in iterating over independent cells/particles. This could lead to further performance gains.



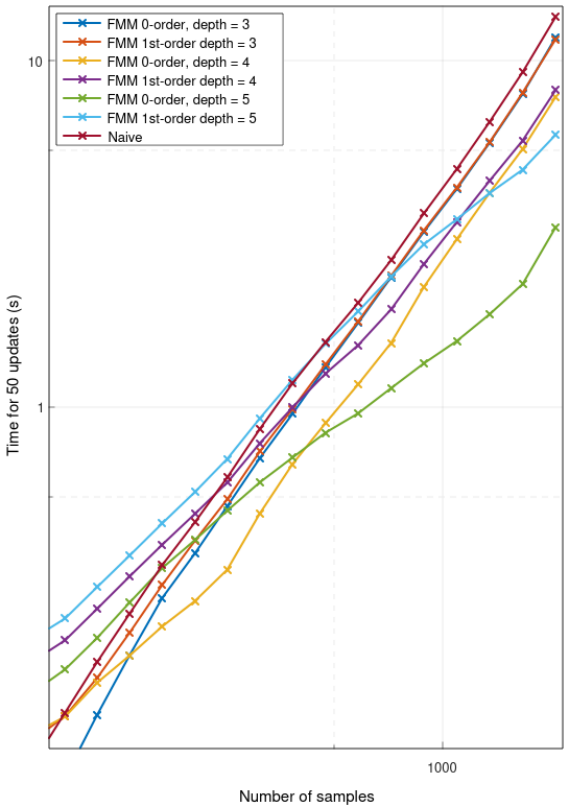
(a) All tests



(b) Zoom on the tests where $N < 350$



(c) All tests (log-scale)



(d) Zoom on the tests where $N < 350$ (log-scale)

Figure 3: Execution time for 50 updates. For each N, the different tests were ran in parallel (i.e. 7 threads).

3.2 - Accuracy tests

We define the square divergence between two solvers of the N-body system, giving two predictions $(\vec{x}_k), (\vec{y}_k)$ as

$$\mathrm{Div}((\vec{x}_k), (\vec{y}_k)) = \frac{1}{NR^2} \sum_{k=1}^{N} \|\vec{x}_k - \vec{y}_k\|^2 \tag{15}$$

With Div, we can assess the accuracy of a solver by comparing it to the naive (most correct) one. Using the same setup as in the previous part, and the same randomness seed, the results are presented in Figure 4. As expected, the accuracy decreases as N rises, and the order-1 solver is indeed much more precise than the order 0 one when $d = 3$. Disappointingly, this stops as soon as $d = 4$, where the difference becomes unseeable. Furthermore, the accuracy becomes overall pretty bad when $d > 3$.

This is either due to the order of expansion being unsufficient, or of a mistake in the code when propagating values through the tree, which time constraints prevented us from locating and correcting.

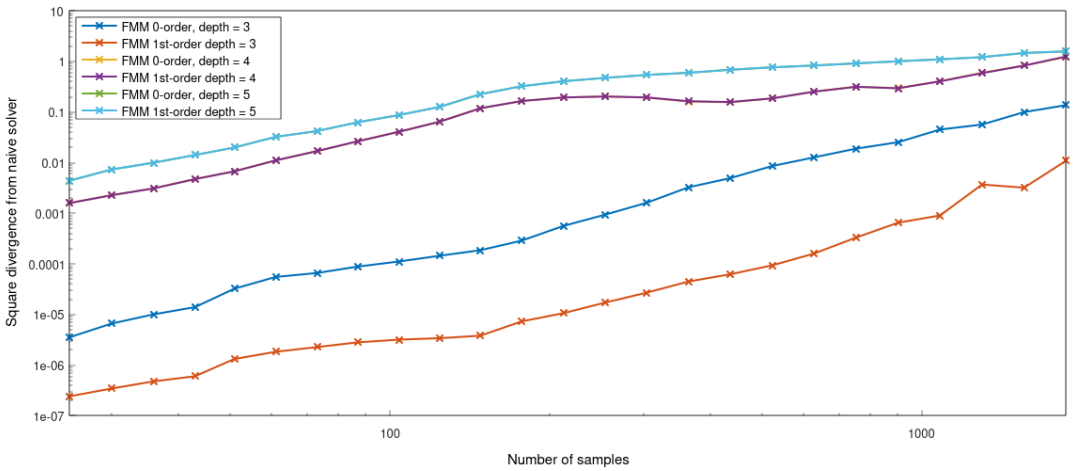


Figure 4: Accuracy of various solvers compared to the naive one (lower is better). The yellow and purple curves are almost identical, as well as the sky-blue and green ones.

Takeaway

Although we could see that the order 1 solver is clearly pore accurate when $d = 3$, the cases where $d > 4$ seem broken. A more thorough investigation of the code or an increase in the expansion order p would be necessary to fix these issues (which implies significant change to the algorithm, as new multipoles and element of the field tensor will become relevant).

This lack of precision might also be due to our chosen tree structure, too coarse to replicate the main interactions in a satisfying way.

3.3 - Visual rendering

By saving the different frames before rendering the animation, we were able to simulate various systems. As a result, we managed to simulate the way mass samples should orbit around a high-mass sample.

The parameters of the system are the following :

1. `N_cluster` and `N_sample` being respectively the number of clusters and the number of samples per cluster.
2. `mu`, the mass applied to all samples (the high-mass sample has a 3000 times higher mass).
3. `size`, which is the size of the simulation volume.
4. `dt`, the time step used by the solver to update the system.
5. `nb_images` and `frame_rate`, used to generate the animation, respectively the number of frames generated and the frame rate.

For example, by generating 20 clusters of 1 sample, we obtain this result:

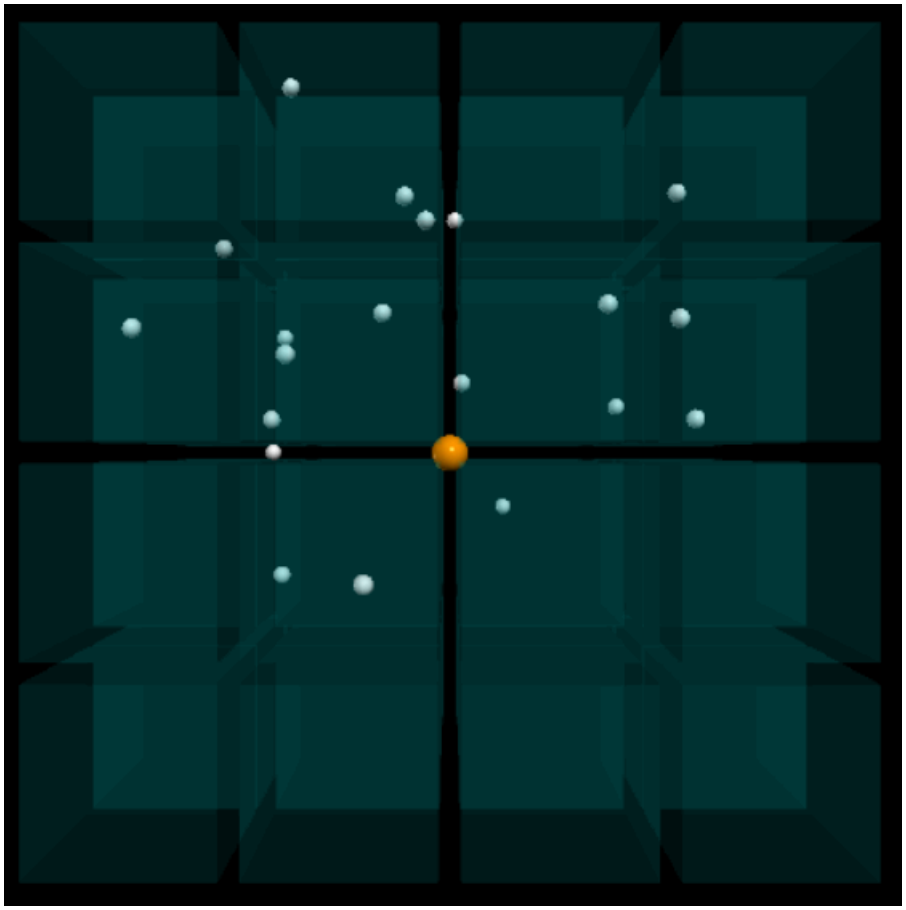


Figure 5: Example of generated frame. Cells are in blue, samples in white and the high-mass sample is at the center of the simulation volume.

By looking at the animation, we can notice that the samples tend to describe elliptic-like trajectories as expected from particles orbiting around a heavy center. By trying the same simulation with more samples,

we can even notice that the "sun" slightly move around the center of the simulation volume, in a fashion similar to which Jupiter enforces a slight movement of the sun around the solar system's center of mass.

However, by using Python (required to use VPython for animation), the animation still requires a lot of time to be generated (about 10 seconds for 500 frames of the previous illustrated example). By using a graphical engine that is compatible with, for instance, the C++ implementation, we could surely reduce the global rendering time.

To visualize the accuracy of the FMM algorithm, we also implemented a way to have a visual comparison between naive and FMM samples movements that can be seen on Figure 6.

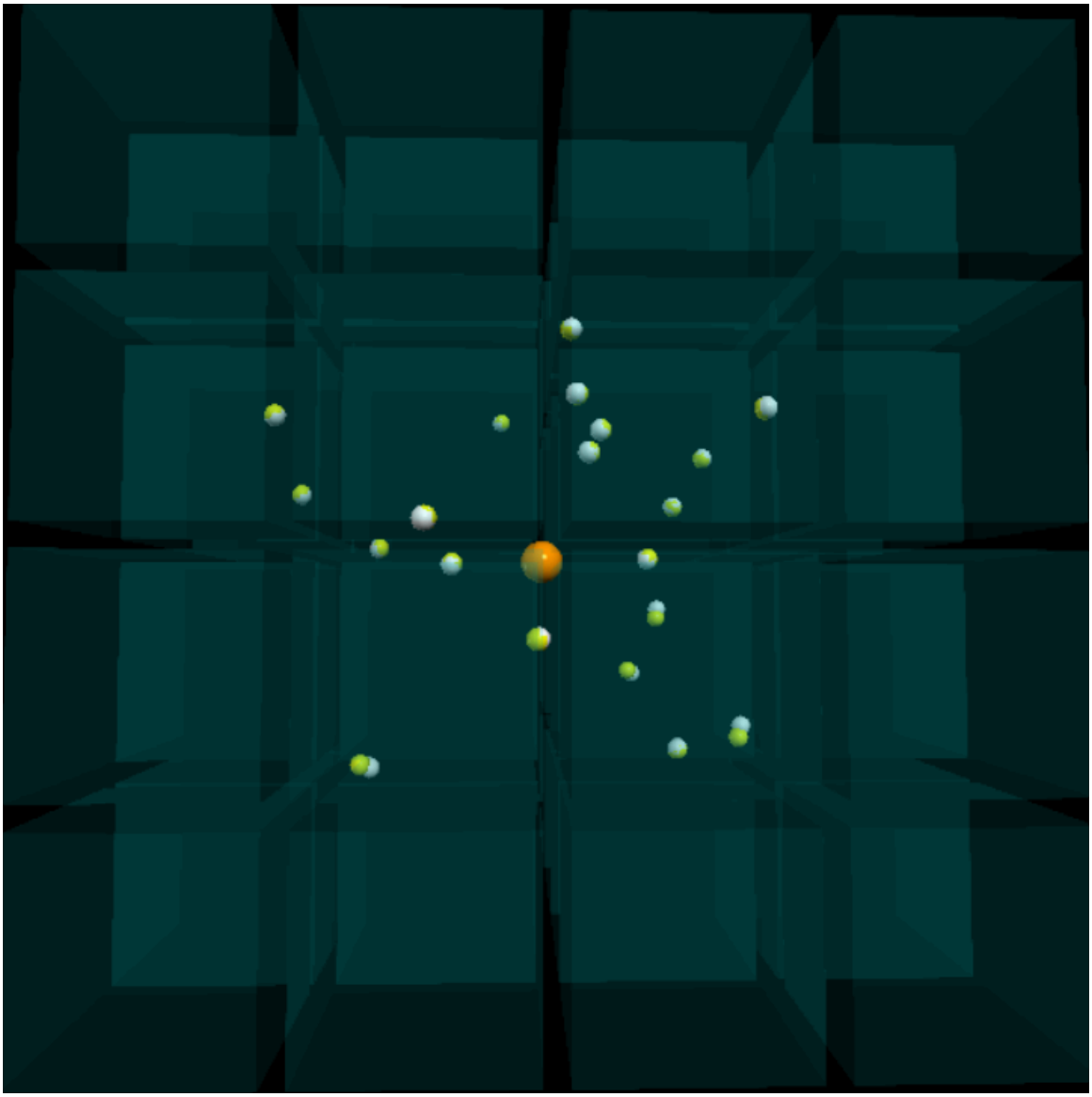


Figure 6: Comparison of the outputs of the FM method (white points) and naive method (yellow points).

CONCLUSION

As expected, the FMM algorithms gives a relatively precise depiction of the way numerous interacting masses are supposed to move, as long as the particles are not more than a few hundreds. According to Section 3.1, the FMM algorithm also has a noticeable effect on computation time compared to the naive solution. However, the FMM computation process is defined with various parameters, such as tree depth and expansion order, which require to be adjusted when simulating such systems.

More specifically, as shown by the curves obtained, the computation time can be optimized by choosing the tree depth accordingly with the number of samples. However, accuracy also tends to decrease as the tree depth increases. This can be compensated by computing the FMM algorithm with a larger expansion order, but it will also slow down the process : there is an equilibrium to be found.

BIBLIOGRAPHY

[1] W. Dehnen and J. I. Read, "N-body simulations of gravitational dynamics," *The European Physical Journal Plus*, vol. 126, no. 5, May 2011, doi: 10.1140/epjp/i2011-11055-3.

[2] L. Chen, "Introduction to fast multipole methods," 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:10209345