Journal of Computational Mathematics
Vol.41, No.6, 2023, 1093–1116.

http://www.global-sci.org/jcm
doi:10.4208/jcm.2202-m2021-0324

A FAST FREE MEMORY METHOD FOR AN EFFICIENT COMPUTATION OF CONVOLUTION KERNELS*

Matthieu Aussal and Marc Bakry[1)]
Ecole Polytechnique (CMAP), INRIA, Institut Polytechnique Paris, Route de Saclay 91128, Palaiseau, France
Email: matthieu.aussal@polytechnique.edu, marc.bakry@polytechnique.edu

Abstract

We introduce the Fast Free Memory method (FFM), a new implementation of the Fast Multipole Method (FMM) for the evaluation of convolution products. The FFM aims at being easier to implement while maintaining a high level of performance, capable of handling industrially-sized problems. The FFM avoids the implementation of a recursive tree and is a kernel independent algorithm. We give the algorithm and the relevant complexity estimates. The quasi-linear complexity enables the evaluation of convolution products with up to one billion entries. We illustrate numerically the capacities of the FFM by solving Boundary Integral Equations problems featuring dozen of millions of unknowns. Our implementation is made freely available under the GPL 3.0 license within the Gypsilab framework.

Mathematics subject classification: 65T50, 65Z05, 65R20.
Key words: Convolution product, Fast multipole method, Boundary integral equations, Open-source.

1. Introduction

The numerical computation of convolution products is a crucial issue arising in many domains like the filtering, the computation of boundary integral operators, optimal control, etc. In a continuous framework, a convolution product is of the form

$$v(x) = \int_\Omega G(x,y)\, u(y)\, d\Omega_y, \tag{1.1}$$

where Ω is some domain of integration in $\mathbb{R}^d, d \in \mathbb{N}^\star$, u some function. The bivariate function $G(.,.)$ is some convolution kernel of the form

$$G(x,y) = G(x-y, |x-y|), \tag{1.2}$$

where $|\,.\,|$ is the euclidean distance. Of course, Eq. (1.1) does not admit an analytical expression in the general case and the integral is computed numerically using, for instance, a quadrature rule. Assuming we want to evaluate v on a finite set of nodes $X = (x_i)_{i\in[\![1,N_X]\!]}$, we have

$$v(x_i) \approx \sum_{j=1}^{N_Y} \omega_j\, G(x_i,y_j)\, u(y_j), \tag{1.3}$$

* Received November 16, 2021 / Accepted February 16, 2022 /
Published online July 26, 2022 /
1) Corresponding author

where $(\omega_j)_{j\in[\![1,N_Y]\!]}$ and $Y = (y_j)_{j\in[\![1,N_Y]\!]}$ are respectively the weights and nodes of such a quadrature. The *discrete convolution* may be recast as a simple matrix-vector product

$$\mathbf{v} = \mathbf{G}\cdot\mathbf{W}\cdot\mathbf{u}, \tag{1.4}$$

where $\mathbf{u} = (u_j)_j = (u(y_j))_j$, $\mathbf{v} = (v_i)_i = (v(x_i))_i$, $\mathbf{G} = G(x_i, y_j))_{\{i,j\}}$ and $\mathbf{W} = \mathrm{diag}((\omega_j)_j)$ (we omit $\mathbf{W}$ in the following). Obviously, the matrix $\mathbf{G}$ is *dense*. Therefore, the memory footprint and computational cost grow quadratically. The computation of (1.4) is constrained to smaller problems ($N_X, N_Y \approx$ a few thousand) on personal computers and smaller servers, and to $N_X, N_Y \approx 10^6$ for industrial servers.

The current approach is to perform the computation approximately up to a given tolerance ε (accuracy). In the past thirty years, multiple so-called acceleration methods have been proposed. The entries of $\mathbf{G}$ can be seen as the description of an interaction between a source set of nodes Y and a target set of nodes X. Thus all blocks of $\mathbf{G}$ describe the interaction between a source subset of Y and a target subset of X. Theses interactions may be compressible, i.e. it admits a low-rank representation. This is the case, for example, when two subsets are far enough following an admissibility criterion. The methods mentioned in the following propose different alternatives on the way the interactions are characterized and computed. The standard way is probably the Fast Multipole Method (FMM) developed by L. Greengard and V. Rokhlin (see [13]), initially introduced for the computation of the gravitational potential of a cloud of particles. Later versions feature the support of oscillatory kernels like the Helmholtz Green kernel. One major drawback is that the implementations are mostly kernel-specific despite recent advances in the domain. We refer to [14] for more details. In 1999, a new approach named Hierarchical matrices ($\mathcal{H}$-matrices) was introduced by S. Börm, L. Grasedyck and W. Hackbusch. This method is based on the representation of the matrix by a quadtree whose leaves are low-rank or full-rank submatrices. A strong advantage in favor of hierarchical matrices is that a complete algebra has been created: addition, multiplication, LU-decomposition, etc. Unfortunately, $\mathcal{H}$-matrices become less effective for strongly oscillating kernels because the rank of the compressible blocks increases with the frequency of the oscillations. For more details and a complete mathematical analysis, we refer to [1, 6]. A recent compression method is the Sparse Cardinal Sine Decomposition (SCSD) proposed by F. Alouges and M. Aussal in 2015 [15]. It is based on a representation of the Green kernel in the Fourier domain using the integral representation of the cardinal sine function. One major advantage is that the matrix-vector product is performed without partitioning of the space. However, there is no corresponding algebra. More recently, we find skeletonization techniques like the Hierarchical Interpolative Factorization [30]. All the aforementioned methods aim at having a quasi-linear memory footprint and computational complexity.

In this paper, we present a simplified implementation of the well-known FMM algorithm which we call the Fast Free Memory method (FFM). The FFM aims at being easier to implement and enabling a high level of versatility without compromising too much on the performance. The purpose of this method is not to compete with the high-performance industrial FMM but rather to provide a good and well-performing academic tool still capable of dealing with very large problems with dozen of millions of entries. Unlike the FMM, there are no communications between levels of the octree (there are no L2L or M2M operators): only a downward “implicit” tree traversal is performed. It makes use of the Non Uniform Fast Fourier Transform algorithm when the convolution kernel is oscillating, and the Lagrange interpolation in the other case. Consequently, it is easy to support a new kernel in an already-existing implementation. The complexity remains quasi-linear for non-oscillatory kernels or for oscillatory kernels when the nodes are scattered in a volume. In the case of nodes scattered over a surface, we show that the oscillating FFM has a $\mathcal{O}(N^{3/2} \cdot \log^2(N)$ complexity. In practice, linear or quasi-linear complexity is achieved. Moreover, computations with hundred of millions of nodes are performed on laboratory-sized servers. A reference implementation written in the Matlab programming language is provided under the GPL 3.0 license within the Gypsilab framework. We divide this paper in three parts. In the first part, we develop the FFM algorithm for standard kernels. We take as examples the Laplace and Helmholtz kernels. We point out the differences with a reference implementation of the FMM. In the second part, we prove the memory and computational complexity estimates. In the last part, we illustrate the functioning of the FFM on various examples including academic and industrially-sized problems with millions of nodes, up to one billion.

2. The FFM Algorithm

Like the FMM, the FFM is a divide-and-conquer method. A typical example of the main loop of the algorithm is given on Algorithm 2.1 and reflects the implementation which can be found in the file ffmProduct.m in the subfolder openFfm/ at [21]. It is based on two partitioning trees (one for the source and one for the target set) just like the one used in the FMM (see for example [29] and Fig. 2.1).

The basic idea is that the interaction between subsets of nodes sufficiently far one from another admit a low-rank representation. The space is therefore partitioned using two octrees (see Fig. 2.1) obtained by successive refinements of the bounding boxes of the initial source and target set of nodes. At each level of refinement, boxes far from each other (following a given criterion) correspond to compressible interactions. The other boxes are further subdivided and yield in turn low-rank and non-compressible interactions. When the matrices corresponding to the non-compressible interactions are small-enough, a full computation is performed. For more details, we refer to the bibliography.

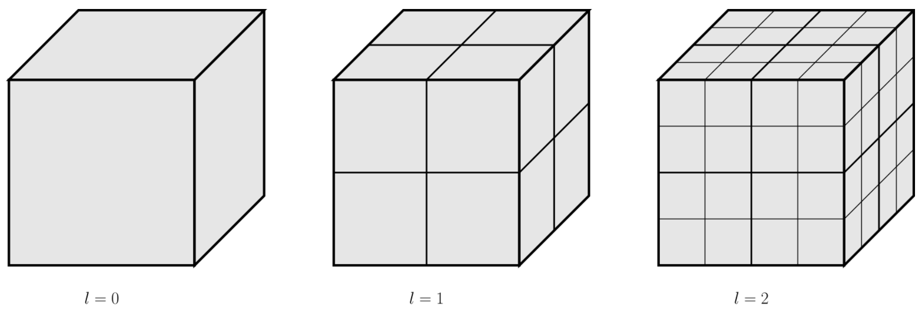

Fig. 2.1. Representation of the octree used in the FFM with three levels of refinement.

Remark 2.1. Regarding the implementation of the algorithm described thereafter, it is important to notice that everything is computed inline. There is absolutely no pre-computation since the purpose is to spare as much memory as possible.

2.1. The kernel-independent FFM

In this subsection, we describe the compression method for the kernel-independent matrix-vector product. We first describe the initialization. Then, we explain how the kernel-independent matrix-vector product is computed using a well-known Lagrange-interpolation method. We show how the user can choose the compression method when dealing with the particular case of the Helmholtz Green kernel. Finally, a stopping criterion is proposed. In the following, we reuse the previous notations and we introduce l as the depth of the octree. The case $l = 0$ corresponds to the root and $l = l^{\max}$ is the maximum allowed depth.

The initialization ($l = 0$) is performed easily by computing the bounding boxes for X and Y. Let $d_{X,\max}$ and $d_{Y,\max}$ be the maximum edge length of X, resp. Y, then the initial length for both bounding boxes is

$$d_0 = \max(d_{X,\max}, d_{Y,\max}). \tag{2.1}$$

The initial bounding box for each set is simply a cube enclosing X, resp. Y, with edge length d_0. At the depth l in the octree, the edge length of the bounding boxes is simply $d_l = d_0/2^l$.

The kernel-independent FFM may be seen as a $\mathcal{H}^2$-matrix-vector product on-the-fly where the binary tree (see [31], chapter 3) has been replaced by an octree, see for example [23, 29]. We suppose that the current refinement level is $l \in [\![1, l^{\max}]\!]$. We consider only one interaction between a box of the source tree and a box of the target tree. We try to perform a low-rank approximation if the distance between their centers is greater than two-times the edge length of a box. Let m be the number of nodes in the target box and n be the number of nodes in the source box, the compressed product is performed using a bivariate Lagrange interpolation, see for example [19] or the chapters related to the $\mathcal{H}^2$-matrices in [1]. The principle of the Lagrange interpolation is to approximate the convolution kernel like

$$G(x,y) \approx \sum_{i=1}^{r_X} \mathcal{L}_i(x) \sum_{j=1}^{r_Y} G(x_{c,i}, y_{c,j})\, \mathcal{L}_j(y), \tag{2.2}$$

where

- $\{x_{c,i}\}_i$ and $\{y_{c,j}\}_j$ are the *target* and *source control* nodes for the Lagrange polynomials. Following [19, 22], the best interpolation nodes are the Chebyshev nodes.
- $\mathcal{L}_i(x)$, resp. $\mathcal{L}_j(y)$ are the Lagrange polynomials defined as the *tensorization of the one-dimensional Lagrange polynomials* in each direction of the space and localized at the *control* nodes.
- r_X and r_Y are the ranks of the interpolation for the target and source variables. If $r_{X,d}, r_{Y,d}$ are the rank of the interpolation in *the direction* d, then $r_X = \prod_{d=1}^3 r_{X,d}$ and $r_Y = \prod_{d=1}^3 r_{Y,d}$. For a prescribed accuracy ε on the interpolation, these ranks depend on the size of the interpolation domain (in other words: the size of the bounding boxes) and on the regularity of the kernel being interpolated (see for example Theorem 4.16 in [31] for one-dimensional polynomials). Since we build the multi-dimensional Lagrange polynomials as a tensorization of one-dimensional polynomials, the ranks r_X and r_Y are bounded above by the maximum rank encountered for the interpolation at the top of the tree and we have set

$$r_X \equiv r_Y \equiv r \equiv (r_1)^3 \tag{2.3}$$

in the FFM framework where r_1 is the maximum rank of the one-dimensional interpolations.

Remark 2.2. In the FFM, the one-dimensional Lagrange interpolators are computed by computing each of the Lagrange polynomials, but it may eventually be improved following the choice made in [23].

The interpolated matrix-vector product can be therefore recast as

$$\mathbf{v} \approx \mathbf{L}_X^T \cdot \big(\mathbf{T} \cdot (\mathbf{L}_Y \cdot \mathbf{u})\big), \tag{2.4}$$

where

- $\mathbf{u}$ is the source vector whose entries are localized at the nodes contained within the source box.
- $\mathbf{L}_Y : r \times n$, resp. $\mathbf{L}_X : r \times m$, is the interpolation matrix whose entries are the Lagrange polynomials localized at the source, resp. target, control nodes evaluated at the source, resp. target, nodes.
- $\mathbf{T} : r \times r$ is the *transfer*-matrix whose entries are the kernel evaluated for each possible couple of target and *source* control node.

However, r is not necessarily low and $\mathbf{T}$ can be further compressed using the Adaptive Cross Approximation, see [5], such that

$$\mathbf{T} \approx \mathbf{A} \cdot \mathbf{B}^T, \tag{2.5}$$

where $\mathbf{A} : r \times r_{\mathbf{T}}$ and $\mathbf{B} : r \times r_{\mathbf{T}}$ such that $r_{\mathbf{T}} \ll r$. Finally, the Lagrange interpolation consists in four successive matrix-vector products such that

$$\mathbf{v} \approx \mathbf{L}_X^T \cdot \left(\mathbf{A} \cdot \left(\mathbf{B}^T \cdot (\mathbf{L}_Y \cdot \mathbf{u})\right)\right). \tag{2.6}$$

As the rank r depends on the kernel, it may increase unacceptably when dealing with oscillatory kernels because the polynomial order must be high to fit the oscillations in the subdomains. In that case, it is beneficial to use kernel-specific compression method as illustrated in the next subsection.

2.2. The case of oscillatory kernels

In this section, we illustrate the change of compression method for the special case of the Helmholtz Green kernel

$$G(x,y) = \frac{1}{4\pi}\frac{e^{ik|x-y|}}{|x-y|}, \tag{2.7}$$

where $k \in [0,\infty[$ is the wavenumber. A test is performed on the value of $k \cdot d_l$ to determine whether the low-rank interaction is oscillating. For example, the evaluation of the Helmholtz kernel (2.7) for two nodes x_i and y_j sufficiently close can be detected as non-oscillating because the value of $k \cdot |x_i - y_j|$ is small. In this case, we use the Lagrange interpolation instead of a specific low-frequency FMM (see [11] or [12]). In the other case, we approximate the kernel using its Gegenbauer-series expansion like in the FMM.

Let x_0, resp. y_0, be the center of the target, resp. source, box, then

$$x - y = (x - x_0) + (x_0 - y_0) + (y_0 - y), \tag{2.8}$$

which can be reformulated like

$$\mathbf{r} = \mathbf{r}_0 + \mathbf{r}_{xy}, \tag{2.9}$$

where $\mathbf{r}_0 = x_0 - y_0$. Let also $\mathbb{S}^2$ be the unit sphere in $\mathbb{R}^3$, then following for example [17]

$$\frac{e^{ikr}}{r} = ik \lim_{L\to\infty} \int_{\mathbb{S}^2} e^{ik\hat{s}\cdot\mathbf{r}_{xy}} T_{L,\mathbf{r}_0}(\hat{s})\, d\hat{s}, \tag{2.10}$$

where $r = |x - y|$ and $T_{L,\mathbf{r}_0}(\hat{s})$ is the Gegenbauer series such that

$$T_{L,\mathbf{r}_0}(\hat{s}) = \sum_{p=1}^{L} \frac{(2p+1)i^p}{4\pi} h_p^{(1)}(k\,r_0) P_p(\hat{s}\cdot\hat{\mathbf{r}}_0), \quad \hat{\mathbf{r}}_0 = \mathbf{r}_0/|\mathbf{r}_0|, \quad r_0 = |\mathbf{r}_0|, \tag{2.11}$$

where $h_p^{(1)}$ is the spherical Hankel function of the first kind and of order p, P_p is the Legendre polynomial of order p. In practice, (2.11) is truncated at the rank

$$L = \lfloor k \cdot \sqrt{3} \cdot d_l - \log(\varepsilon) \rfloor, \tag{2.12}$$

where d_l is the size of the edge of the bounding box and ε is the prescribed accuracy on the matrix-vector product, see [18]. The integral (2.10) is computed using a spherical quadrature $\{\omega_q, \hat{s}_q\}_{q\in[\![1,n_Q]\!]}$ such that, for two nodes x_i and y_j,

$$\frac{e^{ik|x_i-y_j|}}{|x_i - y_j|} \approx ik \sum_{q=1}^{n_Q} e^{ikx_i\cdot\hat{s}_q} \left(\left(\omega_q T_{L,\mathbf{r}_0}(\hat{s}_q) e^{ik\mathbf{r}_0\cdot\hat{s}_q}\right) e^{-ik\hat{s}_q\cdot y_j} \right). \tag{2.13}$$

While it is worth noticing that any spherical quadrature is suitable in (2.13), we chose the following rule:

1. Compute the Gauss-Legendre quadrature $\{\omega_g, x_g\}$ on $[0, 1]$ of order L given by (2.12) where ω_g and x_g are the weights and nodes.
2. Compute the "elevation angle" nodes and weights using

$$\theta_g = \arccos(2x_g - 1) - \frac{\pi}{2}, \quad \omega_{\theta_g} = 2\omega_g \tag{2.14}$$

   in reversed order (in other words the first element θ_g corresponds to the last x_g). Compute the $n_\varphi = 2L + 1$ azimutal evenly spaced quadrature nodes such that the i-th quadrature node is

$$\varphi_{g,i} = (i - 1)\frac{2\pi}{n_\varphi}, \quad \omega_{\varphi_g,i} = \frac{2\pi}{n_\varphi}, \quad i \in [\![1, n_\varphi]\!]. \tag{2.15}$$

3. The spherical quadrature is obtained as a tensorization of the elevation and azimuthal quadratures computed at the steps 2 and 3.

The computation is therefore recast as three successive matrix vector products

$$\mathbf{v} \approx \mathbf{F}_{\hat{s}\to X} \cdot \left(\mathbf{T}_{L,\mathbf{r}_0} \cdot (\mathbf{F}_{Y\to\hat{s}} \cdot \mathbf{u})\right), \tag{2.16}$$

where

- $\mathbf{F}_{Y\to\hat{s}}$ is the *dense* matrix representing the discrete non-uniform forward Fourier transform from the *source* set to the Fourier domain such that

$$\tilde{u}_q = \sum_{j=1}^{n} e^{-ik\hat{s}_q\cdot y_j} u_j. \tag{2.17}$$

- $\mathbf{T}_{L,\mathbf{r}_0}$ is the *transfer diagonal* matrix whose entries contain the value of Gegenbauer series evaluated at $\hat{s}_q$ such that

$$\tilde{v}_q = ik\,\omega_q \left(T_{L,\mathbf{r}_0}(\hat{s}_q)e^{ik\mathbf{r}_0\cdot\hat{s}_q}\right)\tilde{u}_q. \tag{2.18}$$

- $\mathbf{F}_{\hat{s}\to X}$ is the *dense* matrix representing the discrete non-uniform backward Fourier transform from the Fourier domain to the *target* set such that

$$v_i = \sum_{q=1}^{n_Q} e^{ikx_i\cdot\hat{s}_q}\tilde{v}_q. \tag{2.19}$$

Of course, the Fourier-operators $\mathbf{F}_{Y\to\hat{s}}$ and $\mathbf{F}_{\hat{s}\to X}$ are never assembled and the Fourier transforms are computed using the corresponding Non-Uniform Fast Fourier Transform (NUFFT) introduced by A. Dutt and V. Rokhlin [8], later improved by Greengard [9]. We refer to these papers for a complete analysis of the algorithm.

Remark 2.3. In the FMM, the Fast Fourier Transforms are computed at the deepest level in the trees, forcing a back-propagation in the octree. In the FFM, they are performed on-the-fly enabling a downward-only algorithm in the sense that only a downward pass of the tree is performed.

Stopping criterion. The recursive partitioning is stopped whenever one of the following is verified:

- any compressible interaction has been computed,
- the average number of nodes in the boxes is below some value.

The remaining non-compressible interactions, if any, are computed as a dense matrix-vector product.

Comparison with a reference implementation of the Fast Multipole Method. We recall below the main steps of the FMM fmm3dlib available at [25]. According to the source code, these steps are

1. assign the source nodes to boxes and compute multipole expansions,
2. compute local expansions or evaluate directly,
3. merge multipole expansions (this is the end of the upward pass),
4. convert multipole expansions to local expansions,
5. split local expansions,

6. for the target node, evaluate multipole expansions or evaluate directly,
7. evaluate the local expansions,
8. evaluate the direct interactions.

Algorithm 2.1. The main FFM loop

```
Input: $X$, $Y$, $\mathbf{u}$ (input vector), $\varepsilon$ (rel. accuracy), $k$
Output: $\mathbf{v} = \mathbf{G} \cdot \mathbf{u}$ up to $\varepsilon$
$\mathbf{v} \leftarrow 0$
% initialize the root boxes. The octrees are not recursively implemented.
$[X_{\min},\ X_{\max},\ Y_{\min},\ Y_{\max}] \leftarrow [\min(X),\ \max(X),\ \min(Y),\ \max(Y)]$
$e_l \leftarrow 1.01 \cdot \max(X_{\max} - X_{\min},\ Y_{\max} - Y_{\min})$          ▷ Maximum edge length
$[X_{\text{ctr}},\ Y_{\text{ctr}}] \leftarrow [X_{\min} + e_l/2,\ Y_{\min} + e_l/2]$          ▷ Centers of the $X$ and $Y$ boxes
$[X_{\text{ind}},\ Y_{\text{ind}}] \leftarrow [1 : N_X,\ 1 : N_Y]$
$X_{\text{boxes}} \leftarrow \text{initBoxes}(X_{\text{ctr}},\ X_{\text{ind}},\ X)$          ▷ Init. the boxes with their data
$Y_{\text{boxes}} \leftarrow \text{initBoxes}(Y_{\text{ctr}},\ Y_{\text{ind}},\ Y)$
% initialize the list of interactions
$\mathcal{L}^l_{X\times Y} = [1,\ 1]$          ▷ only two boxes at step 0
% now the main loop
while $\text{length}(\mathcal{L}_{X\times Y}) > 0$ & $\text{average}(X_{\text{boxes}}) > 100$ & $\text{average}(Y_{\text{boxes}}) > 100$ do
    % subdivides the boxes
    $X_{\text{boxes}} \leftarrow \text{subdivide}(X_{\text{boxes}})$          ▷ subdivide target tree
    $Y_{\text{boxes}} \leftarrow \text{subdivide}(Y_{\text{boxes}})$          ▷ subdivide source tree
    $e_l \leftarrow e_l/2$
    % compute the list of compressible and non-compressible interactions
    $[\mathcal{L}^l_{X\times Y}, \mathcal{L}^l_{\text{far}}] \leftarrow \text{computeFarInteractions}(\mathcal{L}^l_{X\times Y})$
    % compute the compressible (far) interactions if any
    if $\text{length}(\mathcal{L}^l_{\text{far}}) > 0$ then
        if the interactions are computable using the Gegenbauer series
        if isHighFreqency($k$, $e_l$, $\varepsilon$) then
            % here, we use the Non Uniform Fast Fourier Transform
            $\mathbf{v} \leftarrow \mathbf{v} +$ computeHighFrequency($\mathbf{u}$, $k$, $\varepsilon$, $\mathcal{L}^l_{\text{far}}$, $X_{\text{boxes}}$, $Y_{\text{boxes}}$, $X$, $Y$)
        else
            % here, we use the Lagrange interpolation
            $\mathbf{v} \leftarrow \mathbf{v} +$ computeLowFrequency($\mathbf{u}$, $\varepsilon$, $\mathcal{L}^l_{\text{far}}$, $X_{\text{boxes}}$, $Y_{\text{boxes}}$, $X$, $Y$)
        end if
    end if
    now we can go to the next level
end while
% we deal with remaining non-compressible interactions if any
if $\text{length}(\mathcal{L}^l_{X\times Y}) > 0$ then
    $\mathbf{v} \leftarrow \mathbf{v} +$ computeFullProduct($\mathbf{u}$, $k$, $\varepsilon$, $\mathcal{L}^l_{X\times Y}$, $X_{\text{boxes}}$, $Y_{\text{boxes}}$, $X$, $Y$)
end if
% the end
return $\mathbf{v}$
```

The only common steps with the FMM are step 1. (computation of the octree) and step 8. (computation of the direct interactions). In the FFM, the computations of the octree and of the interactions are merged together at each level of subdivision of the source and target sets. The computation of the compressible interactions is performed in a much simpler manner thanks to the NUFFT algorithm or Lagrange interpolation.

3. Complexity Analysis

In this section, we prove the $\mathcal{O}(N)$ storage complexity and the $\mathcal{O}(N \cdot \log(N))$ computational complexity of the kernel-independent FFM, where $N = \max(N_X, N_Y)$. Then, we deal with the case of the Helmholtz Green kernel. To that purpose, we assume that each set X or Y consists in an uniform distribution of nodes in a cube. The case of surface node-distributions is eventually tackled as a particular case.

3.1. Complexity of an octree

We recall here some general well-known results on space partitioning trees. Assuming d_0 is the length of the edge of the root bounding box, the bounding boxes at level l have the edge length $d_l = d_0/2^l$. They contain (in average) $n = N/8^l$ nodes. Consequently, assuming uniformly distributed nodes, there are 8^l non-empty boxes and the depth of the octree is $l^{\max} = \lfloor \log_8(N) \rfloor$ in average. The construction of the tree itself requires $\mathcal{O}(N \cdot \log_8(N))$ operations. In general, the required storage is also $\mathcal{O}(N \cdot \log_8(N))$ but it is $\mathcal{O}(N)$ in the FFM framework because we store only the data needed at the current depth. For the sake of simplicity, we assume that the boxes contain exactly $N/8^l$ nodes as it does not modify the overall estimate.

The particular case of surface distributions. We emphasize the fact that using an octree to subdivide evenly distributed nodes on a surface amounts to consider a plane surface partitioned using a quadtree as illustrated on Fig. 3.1.

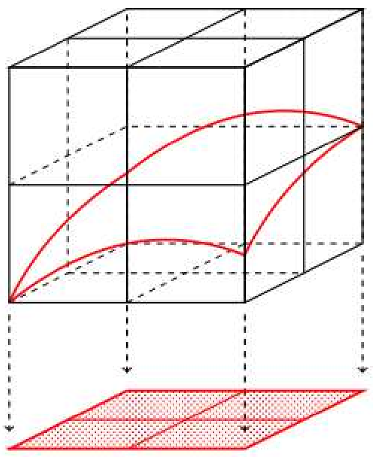
Fig. 3.1. Illustration of the subdivision of a surface using an octree.

Consequently, we replace the 8 by 4 in the aforementioned results: the depth of a quadtree for uniformly distributed nodes is $\lfloor \log_4(N) \rfloor$ in average, etc.

Remark 3.1. For the sake of simplicity, the subscript for the log indicating the type of the logarithm is omitted whenever it is not required for the comprehension.

3.2. Complexity estimates for the kernel-independent FFM

We prove here the storage and computational complexity for the kernel-independent version. We consider here *any* kernel of the form (1.2) as long as N does not depend on any kernel-related parameter like a wavenumber (we refer to the next Section 3.3 on the particular case of oscillating kernels). Since the FFM is a downward-only (downward tree traversal only) algorithm, we can discard data stored at the parent level in the tree. This minimal storage requirement leads to the following proposition.

Proposition 3.1. *The storage complexity of the FFM is $\mathcal{O}(N)$ and the computational complexity is $\mathcal{O}(N \cdot \log(N))$.*

Proof. Before we prove the estimates, we make some preliminary remarks. We first emphasize that the interpolation step for each of the source set, resp. target, is performed only once for all the corresponding subsets. Second, we drive the attention to the fact that, while there should be many interpolations to compute, most of the transfers are the same. This is a consequence of, first the translation invariance of the kernel, and second the previous assumption that the target and source root boxes have exactly the same size (see subsection 2.1 and eq. (2.1)). On Fig. 3.2, the two transfers (arrows) represented between the boxes from the source tree (full lines) and the target tree (dotted) have the same transfer matrix which is computed only once.

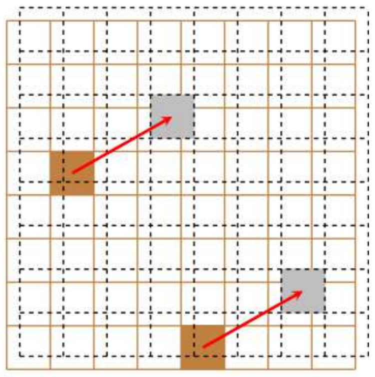

Fig. 3.2. Example of two equivalent transfer steps.

In fact, the amount of different translations is uniquely determined by the initial configuration of the root bounding boxes as illustrated in 2-dimension on Fig. 3.3. The left picture corresponds to two overlapping quadtrees. In this *very particular case*, we enumerate all the possible low-rank interactions for the filled box at the depth 2 and we find 27. All the other boxes are children of boxes involved in low-rank interactions at the previous level of refinement. On the right configuration, the trees are shifted and the amount of possible interactions is

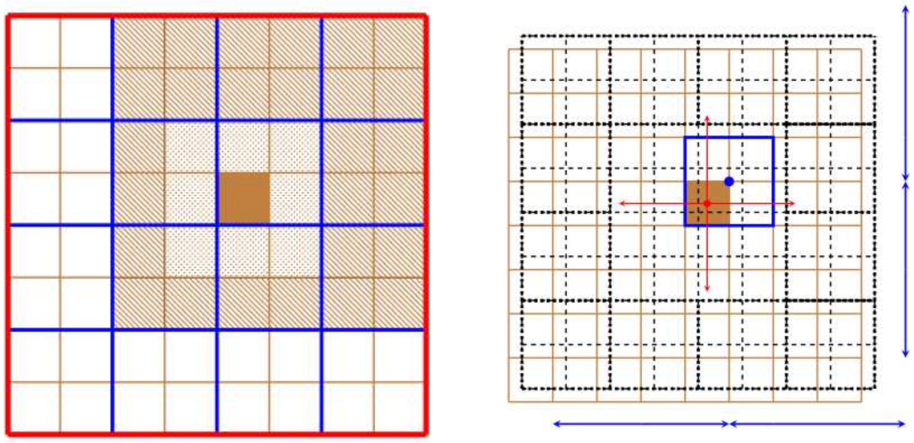

Fig. 3.3. Space configuration of two three-leveled quadtrees in 2-dimension. Left: the quadtrees overlap. Right: the quadtree are shifted. Very thick line: level-0, thick line: level-1, thin line: level-2. Fully colored cell: bounding box for which we want to determine the low-rank interactions, dotted cells: closest possible interactions with the cully colored cell, striped cells: possible low-rank interactions at the given level.

now 48. This number corresponds to the *total number of transfer matrices* which needs to be computed at the current depth l; it does *not* depend on l.

1. Let r_1 be the number of control nodes for the Lagrange interpolation in each direction of space, then $r = r_1^3$ is bounded above by a value independent on N (see 2.1). Let $n_Y = N_Y/8^l$ and $n_X = N_X/8^l$, then the storage requirement for the first interpolation is $\mathcal{O}(n_Y \cdot r) = \mathcal{O}(n_Y)$ per interpolation matrix. Since there are as many matrices as there are boxes in the tree at level l, the storage requirement is then $8^l \cdot \mathcal{O}(n_Y) = \mathcal{O}(N_Y)$. The interpolation consists in 8^l matrix-vector products, each of them of size $r \times n_Y$. The computational complexity of the complete Lagrange interpolation of the source set for any l is therefore $\mathcal{O}(N_Y)$.

2. Let $N_\mathbf{T}$ be the maximum number of unique transfers (see again Fig. 3.3), the storage and the computation of these matrices are each $\mathcal{O}(1)$ as the number and the sizes of the transfer matrices do not depend on N_Y nor N_X. Each interpolated source vector is then multiplied at most $N_\mathbf{T}$-times by a transfer matrix whose size is independent on N_X or N_Y. Consequently, this transfer step has $\mathcal{O}(N)$ complexity for the storage and the computational complexity where we recall that $N = \max(N_X, N_Y)$. The result of this transfer step is a set of "transferred vectors".

3. Finally, the last interpolation is performed in two steps:

   (a) The construction of the interpolation matrices for the target set has the same complexity as for the source set, i.e. it is $\mathcal{O}(N_X)$.

   (b) The computation of the *target* set of interpolations consist in assembling the "interpolation matrix"-"transferred vector" for each of the $N_\mathbf{T} \cdot 8^l$ "transferred vector". This step should be performed at most $N_\mathbf{T}$ times for each of the target i.e. $N_\mathbf{T} \cdot 8^l$-times.

Therefore, the second interpolation step has the same complexity as the step 1, i.e. it is $\mathcal{O}(N_X)$ in storage and computational complexity.

To these costs, we must add the worst case cost of the construction of the octrees which is $\mathcal{O}(N \cdot \log(N))$ and of the non-compressible interactions which is linear because the maximum number of close interactions for one node is bounded by the number of nodes per leaves multiplied by the number of neighbours, which is bounded.

We conclude that the storage requirement for the FFM is $\mathcal{O}(\max(N_X, N_Y))$. Regarding the computational requirement, the worst case consists in performing the interpolation $l^{\max}$-times. We conclude that the computational cost for the complete FFM product is $\mathcal{O}(N \cdot \log(N))$. $\square$

Remark 3.2. These complexity estimates do not depend on whether the nodes are evenly distributed in a volume (octree-based partitioning) or on a surface (equivalent quadtree-based partitioning). By evenly distributed, it is meant that the distance from one node to its closest neighbor is approximately the same for all nodes in the set.

3.3. Complexity estimates for the oscillating kernels

In the previous subsection 3.2, we proved very general estimates. When dealing with oscillating kernels, one must introduce the notion of wavelength λ and of "discretization" per wavelength. Let Δx be the average distance between two nodes,

$$k \cdot \Delta x = \mathcal{O}(1), \tag{3.1}$$

where we recall that $k = 2\pi/\lambda$ is the wavenumber. In the following, we assume that $k \cdot \Delta x = 1$, meaning that there are approximately six nodes per wavelength. This is fundamentally different from the generic case as the number of nodes in a given volume depends *explicitly* on k. We assume in the following that k is accordingly adjusted as a function of N.

In the FFM, we use a three-dimensional type-3 NUFFT[1)]. Following the companion paper [7] of one of the latest development in this field, the transform is performed in four steps:

0. The gridding step consists in creating the regular grid on which the input data will be spread on step 1.
1. The data is spread from the original nodes to the regular grid using a *spreading-kernel.*
2. A type-2 NUFFT[1)] is performed.
3. The output data is corrected in order to compensate for the spreading.

The number of nodes in the grid in each direction may be computed using the formula (3.23) given in [7]

$$n_i = \frac{2\sigma}{\pi} X_i S_i + \lceil |\log_{10}(\varepsilon)| \rceil + 1 \tag{3.2}$$

where σ is some constant upsampling factor, X_i is the largest extent in the direction i in the "physical space", and S_i is the largest extent in the Fourier space. In most applications, X_i does not depend on the number of nodes and it is assumed so in the following. By looking at

[1)] A *type-2* NUFFT is a *uniform to non-uniform* NUFFT. A *type-3* NUFFT is a *non-uniform to non-uniform* NUFFT. See [9]

(2.17), it appears that elements in the Fourier space are distributed over a sphere with radius k which depends on N implying that $S_i = k$. Since this dependence is not the same whether the nodes are distributed in a volume or over a surface, we treat both cases in two different subsections.

3.3.1. Nodes evenly distributed in a volume

Proposition 3.2. *The storage complexity for the oscillating FFM when the nodes are evenly distributed in a volume is $\mathcal{O}(N)$ and the computational complexity is $\mathcal{O}(N \cdot \log^2(N))$.*

Proof. The proof is very similar to the proof of Proposition 3.1. We first remark that the NUFFT is based on the Fast Fourier Transform which has a $\mathcal{O}(\max(n_Q, n_Y))$ storage requirement and a $O(\max(n_Q, n_Y)) \cdot \log(O(\max(n_Q, n_Y)))$ computational complexity where we recall that n_Q is the number of nodes in the spherical quadrature (see 2.2) and that $n_Y = N_Y/8^l$ (see proof of 3.1). The first step is to evaluate the value of n_i introduced in Eq. (3.2). Following the hypothesis $k \cdot \Delta x = 1$, we have that

$$N \sim \left(\frac{d_0}{\Delta x}\right)^3 \quad \Leftrightarrow \quad k \sim N^{1/3}, \tag{3.3}$$

where $\sim$ means "is proportional to". Consequently, n_i grows like $N^{1/3}$ and the total grid size is $n_i^3 = O(N)$. On the other side, we have

$$n_Q = 2 \cdot L \cdot (L+1), \tag{3.4}$$

where L is given by (2.12) and n_q is assumed small with respect to N. By substituting $d_l = d_0/2^l$ and expanding n_Q, we obtain

$$n_Q = A\left(\frac{k}{2^l}\right)^2 + B\left(\frac{k}{2^l}\right) + C, \tag{3.5}$$

where A, B, C are constants. Finally,

$$n_Q = A\frac{N^{2/3}}{4^l} + B\frac{N^{1/3}}{2^l} + C \tag{3.6}$$

meaning that $n_Q = \mathcal{O}(N^{2/3})$. We detail now the complexity of each step:

1. Following the previous remark on the gridding for the "volumic case", the NUFFT has the same complexity as the FFT. Therefore, the storage requirement for one NUFFT is $\mathcal{O}(n_y)$ meaning that the total storage requirement is $\mathcal{O}(N_Y)$. The total number of NUFFTs is 8^l and each has the computational complexity $\mathcal{O}(n_Y \cdot \log(n_Y))$. Since $n_Q < \max(n_Y, n_X)$, we conclude that the storage complexity of the first step is $\mathcal{O}(N)$ and that the computational complexity is $\mathcal{O}(N \cdot \log(N))$.

2. The second step is a simple multiplication in the Fourier domain which is performed $N_{\mathbf{T}} \cdot 8^l$-times on vectors of length n_Q, see the proof of Proposition 3.1. By remarking that $8^l \cdot n_Q = \mathcal{O}(N)$, we conclude that this step has a linear storage and computational complexity.

3. The last step is the backward step of the first one. Consequently, we obtain exactly the same complexities. □

3.3.2. Nodes evenly distributed on a surface

Proposition 3.3. *The storage complexity for the oscillating FFM when the nodes are evenly distributed on a surface is* $\mathcal{O}(N^{3/2})$ *and the computational complexity is* $\mathcal{O}(N^{3/2} \cdot \log^2(N))$.

Proof. In the particular case of surface distributions of nodes, we have $k \sim N^{1/2}$ and the polynomial (3.5) becomes

$$n_Q = A\frac{N}{4^l} + B\frac{N^{1/2}}{2^l} + C. \tag{3.7}$$

This means that for surface distributions of nodes, $n_Q = \mathcal{O}(N)$. Since the coefficients A, B, C are all positive, we conclude that we always have $n_Q \ll N$. The size of the grid in each direction is now $n_i = \mathcal{O}(N^{1/2})$ and the total number of nodes in the regular grid is $\mathcal{O}(N^{3/2})$. We conclude as in the proof of Proposition 3.3. $\square$

Remark 3.3. It has been shown previously that the real cost of the NUFFT is $\mathcal{O}(N^{3/2} \cdot \log^2(N))$. Following [7] for example, it appears that the costliest step is the evaluation of the spreading kernel which happens on the grid-points in the vicinity of the non-uniformly distributed points. This step is $\mathcal{O}(N)$ with eventually a large multiplicative constant depending on the space-dimension. For smaller problem sizes, it is expected that the NUFFT behaves nicely and that the complete algorithm behaves a little bit worse than $\mathcal{O}(N \cdot \log(N))$ as illustrated in the next Section 4. However, from the memory point of view, $\mathcal{O}(N^{3/2})$ nodes are allocated, mostly filled with 0.

4. Numerical Examples

In this section, we give examples of application of the FFM. Our implementation is written in the MATLAB language. First we illustrate the estimates proven in the previous Section 3. In a second time, we show how one can use the FFM to solve Boundary Integral Equations iteratively and we solve two publicly available benchmarks in acoustic and electromagnetic scattering. In the following and unless mentioned otherwise, the error is the relative error defined by

$$\varepsilon = \frac{|\mathbf{v}_{\text{ref}} - \mathbf{v}_{\text{FFM}}|}{|\mathbf{v}_{\text{FFM}}|}, \tag{4.1}$$

where $\mathbf{v}_{\text{ref}}$ is the "exact" matrix-vector product and $\mathbf{v}_{\text{FFM}}$ is the FFM product. When the computation of the reference is impossible because of its quadratic complexity, a subset of 100 entries of the target set is randomly chosen. The computation time is measured using the `tic` and `toc` functions of MATLAB.

4.1. Scalability of the FFM

We first illustrate the scalability of the FFM by computing the convolution product where the kernel is the Laplace Green kernel

$$G_0(x, y) = \frac{1}{4\pi}\frac{1}{|x - y|} \tag{4.2}$$

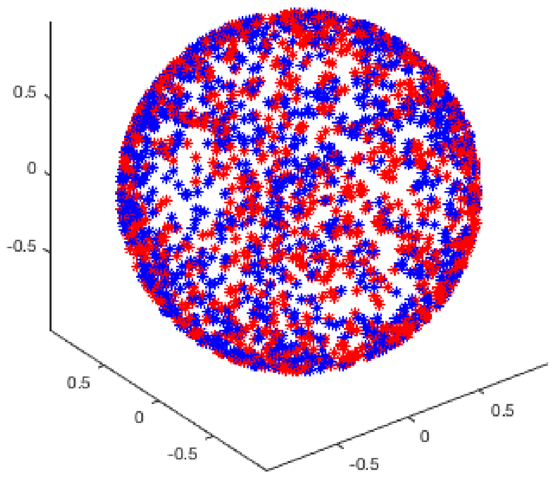

Fig. 4.1. Source (red) and target nodes (blue).

or the Helmholtz Green kernel

$$G_k(x,y) = \frac{1}{4\pi}\frac{e^{ik|x-y|}}{|x-y|}. \tag{4.3}$$

To that purpose, we pick randomly N *source* nodes and N *target* nodes on the unit sphere $\mathbb{S}^2$ centered at the origin as illustrated on Fig. 4.1.

Then, we simply compute the convolution product (1.4) with the FFM. For each of the computations, the number of nodes is chosen by multiplying the number of the previous computation by 2, starting from $N = 512$. For the Laplace kernel, the FFM is expected to perform quasi-linearly using the kernel-independent version. For the Helmholtz kernel, the wavenumber is adjusted to the number of nodes such that $k \cdot d \approx 1$ where $d \equiv \Delta x$ is the average distance between nodes and $\approx$ means "close to". This is the high-frequency regime where the FFM is expected to perform like $\mathcal{O}(N^{3/2} \cdot \log^2(N))$ using the oscillatory approach. For both cases, the scaling is compared to FMM library fmm3dlib-1.2 available on the personal web page of L. Greengard [25].

Remark 4.1 (Warning!). Please note that the sole purpose of this comparison is to compare the complexity. The CPU time and the RAM requirements are implementation-dependent and only given as "markers". We recall that the FMM is written in Fortran and the FFM in Matlab. Moreover, the adaptive FMM in fmm3dlib-1.2 is for educational or research purposes and is based, quoting the documentation, "on rotations and translation along the z-axis. For a fully optimized code, plane wave-based operators should be used."

We measure only the time taken by either the call to our ffmProduct() function or the call to the L. Greengard's code MATLAB interface. The computation[1)] was performed on a server on a single CPU core at 3.0 GHz, 512 GBytes of RAM and MATLAB R2019A using double

[1)] The convolution product between two sets of nodes may be computed using the `ffmProduct()` function, available within the open-source Gypsilab framework [21] in the `./openFfm` directory. An example is provided by running the `nrtFfmBuilder.m` script.

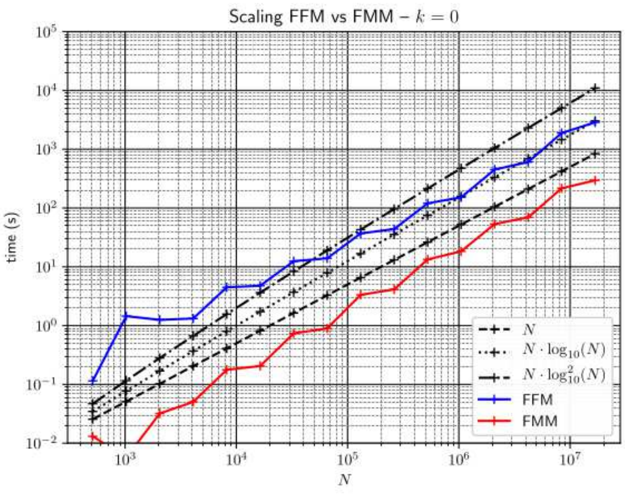

Fig. 4.2. Comparison of the complexity between the FFM and the FMM for fixed $k = 0$. We recall that the two implementations cannot be directly compared except for the computational complexity as the FMM is implemented in Fortran and the FFM is fully implemented in Matlab.

precision floating point numbers. The relative accuracy for the FFM is set to $\varepsilon = 10^{-6}$ which yields the same relative accuracy as the parameter `iprec=1` as defined in the documentation of `fmm3dlib-1.2` at [25]. The results for the case $k = 0$ are displayed on Fig. 4.2. The FFM switches automatically to the Lagrange interpolation and we can observe that both algorithms scale between $(O(N))$ and $\mathcal{O}(N \cdot \log(N))$.

The results for the computation with $k \cdot d \approx 1$ are given on Fig. 4.3. Moreover, by setting Ω the "longest dimension" of the geometry, which can also be seen as the maximum possible length between two nodes (here $\Omega = 2 \cdot R$ where $R = 1$ is the radius of the sphere), the product $k \cdot \Omega$ describes the number of wavelength on the geometry). If this product has a low value,

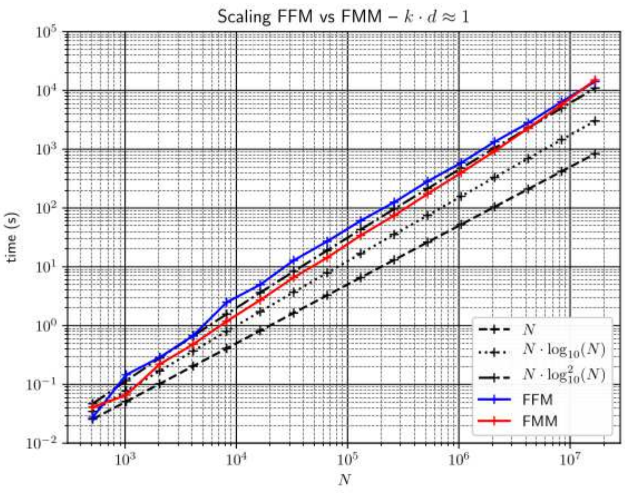

Fig. 4.3. Comparison of the complexity between the FFM and the FMM for $k \cdot d \approx 1$.

eventually much lower than 1, it corresponds to the *low-frequency* regime while a high value corresponds to the high-frequency regime. It also means that interactions at the lower-levels of the tree may be considered *low-frequency* while the interactions at levels closer to the root are *high-frequency*. In our case, it ranges from $k \cdot \Omega \approx 8$ to $k \cdot \Omega \approx 1500$. These results clearly show that the FFM is able to cover a large range of frequencies and that the scaling is comparable to the FMM and lies between $\mathcal{O}(N \cdot \log(N))$ and $\mathcal{O}(N \cdot \log^2(N))$, lower than the theoretical value.

The maximum memory requirement for both computations is given in Table 4.1. It is estimated using the `/usr/bin/time -v` command provided by the operating system Ubuntu Server. The memory requirement remains in an acceptable range, even at high frequencies. The RAM used by MATLAB is approximately 0.56 GBytes of the total amount.

Table 4.1: Maximum memory requirement for $N = 2^{24}(= 16777216)$ in **GBytes**.

	FFM	FMM
$k = 0$	15.25	15.25
$k \cdot d \approx 1$	15.27	22.3

Parallelization of the FFM. The FFM can be parallelized quite naturally. Assuming N_p the number of parallel processes, both source and target sets are split in as many subsets. It is equivalent to a block-representation of the original matrix with $N_p \times N_p$ sub-matrices. Finally, each process computes a single line-block. We illustrate the parallelization for the Laplace Green kernel

$$G(x,y) = \frac{1}{4\pi}\frac{1}{|x-y|} \tag{4.4}$$

and for the Helmholtz Green kernel. The computations are performed on a 12 cores processor at 2.9 GHz, 256 GBytes of RAM and MATLAB R2017A using single precision accuracy. The parallelization is performed using the spmd instruction (Single Process Multiple Dispatch) of the MATLAB Parallel Computing Toolbox. It is a very high-level overlay to MPI-like parallelization: entering the spmd environment, the data is sent to all the parallel workers meaning that there is no shared memory between them. Exiting the environment, the newly created data is sent back to the main worker in a so-called Composite object. We refer to the official documentation[1)] for more informations. The results for the Laplace kernel are gathered on Table 4.2 for a prescribed accuracy $\varepsilon = 10^{-3}$ on the matrix-vector product. We observe that the computational scalability is close to linear since the ratio of two successive computation times is close to 10 which is the multiplicative factor by which N is increased. More importantly we are able to achieve a matrix-vector product with one billion of nodes in each of the source and target set in less than four hours. In this last case, the memory peak is approximately 100 GBytes among which 40 GBytes are required for the storage of the coordinates of the nodes, the input vector and the output vector.

The same experiment is repeated for the Helmholtz Green kernel. The results are given in Table 4.3 with the corresponding maximum $k \cdot d$ value where d is the diameter of the sphere.

Remark 4.2. For smaller problem sizes, we notice that the parallel time is worse than the single-threaded time. We do not have a rigorous explanation. However, it is possible that

[1)] https://fr.mathworks.com/help/parallel-computing/spmd.html

Table 4.2: Summary of the computation time – Laplace kernel.

N	Time 1 core (s)	Time 12 cores (s)	Speed-up	Rel. error
10^4	2.04	9.08	0.22	$8.03 \cdot 10^{-5}$
10^5	9.30	17.1	1.84	$1.34 \cdot 10^{-4}$
10^6	87.8	33.4	2.63	$1.35 \cdot 10^{-4}$
10^7	1063	169	6.29	$1.98 \cdot 10^{-4}$
10^8	–	1499	–	$1.81 \cdot 10^{-4}$
10^9	–	11340	–	$3.11 \cdot 10^{-4}$

Table 4.3: Summary of the computation times – Helmholtz kernel.

N	f (Hz)	$k \cdot d$	Time 1 core (s)	Time 12 cores (s)	Speed-up	Rel. error
10^4	541	20	1.65	8.81	0.18	$2.98 \cdot 10^{-4}$
10^5	1893	70	16.2	16.3	0.99	$1.69 \cdot 10^{-4}$
10^6	5411	200	143	48.2	2.97	$2.77 \cdot 10^{-4}$
10^7	16234	600	1557	350	4.45	$2.91 \cdot 10^{-4}$
10^8	54113	2000	–	8246	–	$3.52 \cdot 10^{-4}$

it is a consequence of memory transfers since we do not have the possibility to tune directly the memory. It could also be dependant on the MATLAB version and the computer. Finally, the choice for the parallelization may not be optimal. An implementation using a compiled language like, `C`, `C++` or `Fortran` should give a better insight on the situation.

4.2. Boundary Integral Equations and the FFM

Boundary Integral Equations can be obtained from certain equations describing, for example, physical phenomena like the propagation of an acoustic or electromagnetic wave in a homogeneous medium. We refer to [2] starting p. 110, or [3] starting p. 66, for more details on how they are obtained. In order to explain how the FFM is used to solve such equations, we consider the scattering of an acoustic wave propagating in an infinite medium by a scatterer with boundary Γ on which we apply a Dirichlet boundary condition. This problem can be tackled by solving the following Boundary Integral Equation

$$\int_\Gamma G(x,y)\,\lambda(y)\,d\gamma_y = -u^{\mathrm{i}}(x), \quad x \in \Gamma, \tag{4.5}$$

where λ is some unknown, $G(x,y)$ is the Helmholtz Green kernel (see (2.7)) and u^{i} is the incident wave. This equation may be solved using different method. Here we present shortly the Boundary Element Method based on a Galerkin formulation. We introduce a test function $\lambda^\star$ to obtain the Galerkin formulation of Eq. (4.5)

$$\int_{\Gamma\times\Gamma} \lambda^\star(x)\,G(x,y)\,\lambda(y)\,d\gamma_y\,d\gamma_x = -\int_\Gamma u^{\mathrm{i}}(x)\,\lambda^\star(x)\,d\gamma_x \quad \text{for all } \lambda^\star. \tag{4.6}$$

We further introduce the discrete approximation spaces $(\lambda_j)_{j\in[\![1,N]\!]}$ and $(\lambda_i^\star)_{i\in[\![1,N]\!]}$ such that

$$\lambda(y) = \sum_{j=1}^{N} u_j \cdot \lambda_j(y), \tag{4.7}$$

$$\lambda^\star(x) = \sum_{i=1}^{N} v_i \cdot \lambda_i^\star(x). \tag{4.8}$$

There are multiple ways to deal with this singular integral. Here we integrate the double integral using a Gauss-Legendre quadrature. The resulting inaccurate integration of the singularity is tackled later. Let $\{\omega_{g,k}, x_{g,k}\}_{k\in[\![1,n_g]\!]}$ and $\{\omega_{g,k}, y_{g,k}\}_{k\in[\![1,n_g]\!]}$ be the weight and nodes of quadrature, the Eq. (4.6) now reads

$$\begin{aligned}&\int_{\Gamma\times\Gamma} \lambda^\star(x)\, G(x,y)\, \lambda(y)\, d\gamma_y\, d\gamma_x \\ &\approx \sum_{i=1}^{N} v_i \sum_{k=1}^{n_g} \lambda_i^\star(x_{g,k})\, \omega_{g,k} \sum_{l=1}^{n_g} G(x_{g,k}, y_{g,l}) \sum_{j=1}^{N} \omega_{g,l}\, \lambda_j(y_{g,l})\, u_j.\end{aligned} \tag{4.9}$$

Therefore, Eq. (4.5) is rewritten as linear system of equations,

$$\mathbf{S} \cdot \lambda = \mathbf{U}^{\mathrm{i}}. \tag{4.10}$$

The Galerkin matrix $\mathbf{S}$ can be recast as the product of three matrices such that

$$\mathbf{S} = (\Lambda^\star)^T \cdot (\mathbf{G} \cdot \Lambda), \tag{4.11}$$

where Λ is the matrix "transporting" the basis functions to the quadrature nodes, $(\Lambda^\star)^T$ is the matrix "quadrature-to-test-functions" and $\mathbf{G}$ is the matrix such that $\mathbf{G}_{ij} = G(x_{g,i}, y_{g,j})$. While the matrices Λ and $\Lambda^\star$ are sparse and can be stored with linear complexity, $\mathbf{G}$ is full. In the process of an iterative inversion algorithm such as GMRES, see [28], one or more matrix-vector products are required,

1. $\tilde{\mathbf{u}} = \Lambda \cdot \mathbf{u}$. This product has linear complexity.
2. $\tilde{\mathbf{v}} = \mathbf{G} \cdot \tilde{\mathbf{u}}$. This product is compressed using the FFM.
3. $\mathbf{v} = (\Lambda^\star)^T \cdot \tilde{\mathbf{v}}$. This product also has linear complexity.

In general, n_g is closely related to the number of elements in the discretization of Γ. Assuming for example that there are three quadrature nodes per element, then $n_g = 3\cdot$(number of elements) meaning that the size of $\mathbf{G}$ may be in fact much higher than the actual size of the linear system. The singular integral is computed independently using a semi-analytical method. It takes the form of an additional sparse matrix which "removes" the singularity integrated numerically in $\mathbf{G}$ and adds the "exact" integration of the kernel.

Our FFM library is interfaced with the GYPSILAB software. GYPSILAB is an open-source (GPL3.0) Finite Element framework entirely written in the MATLAB language aiming at assembling easily the matrices related to the variational formulations arising in the Finite Element Method or in the Boundary Element Method. Among other things, it features a complete $\mathcal{H}$-matrix algebra (sum, product, LU decomposition, ...) compatible with the native matrix types of MATLAB. For more details on the capabilities of GYPSILAB we refer to [20, 21]. In the context of this paper, we use it to manage the computation of the matrices Λ, $\Lambda^\star$ and the right-hand-side $\mathbf{U}^{\mathrm{i}}$ [1]) .

We present here two examples of application. The first one corresponds to the scattering of an underwater acoustic wave by a submarine and the second one is the scattering of an electromagnetic wave by a perfect electric conductor rocket launcher.

4.2.1. Acoustic scattering by a submarine

We solve the acoustic scattering by a submarine with Neumann Boundary condition. This example is based on the BeTSSi benchmark [26]. The mesh is provided by ESI Group and it is remeshed using the open-source Mmg Platform [27]. We could solve this problem using the following equation

$$\int_\Gamma \frac{\partial^2 G}{\partial n_x \partial n_y}(x,y)\,\mu(y)\,d\gamma_y = \frac{\partial u^{\mathrm{i}}}{\partial n_x}, \tag{4.12}$$

whose variational formulation is

$$\begin{aligned} k^2 &\int_{\Gamma\times\Gamma} \left(\mu^\star(x)\cdot G(x,y)\cdot \mu(y)\cdot (n_x\cdot n_y)\right)\,d\gamma_x\,d\gamma_y \\ &- \int_{\Gamma\times\Gamma} \left(\mathrm{rot}_\Gamma \mu^\star(x)\cdot G(x,y)\cdot \mathrm{rot}_\Gamma \mu(y)\right)\,d\gamma_x\,d\gamma_y = \int_\Gamma \mu^\star(x)\cdot \frac{\partial u^{\mathrm{i}}}{\partial n_x}(x)\,d\gamma_x, \end{aligned} \tag{4.13}$$

where μ is the unknown, $\mu^\star$ is the test function, k the wavenumber, n_x is the outbound normal vector at position x, and $\partial/\partial n_x$ is the normal derivative with respect to the variable x. Eq. (4.12) is very ill-conditioned and is also ill-posed for some frequencies. The integral operator is called the hypersingular boundary integral operator. The scattering problem could also be solved using another boundary integral equation

$$-\frac{\mu(x)}{2} + \int_\Gamma \frac{\partial G}{\partial n_x}(x,y)\,\mu(y)\,d\gamma_y = -u^{\mathrm{i}}(x), \tag{4.14}$$

which is better conditioned but which is also ill-posed for some frequencies and less accurate in practice. The boundary integral operator in Eq. (4.14) is the adjoint of the double layer operator. To circumvent these issues, we use a linear combination of the integral operators involved in Eqs. (4.12) and (4.14) called the Brakhage-Werner formulation, see [4]. This formulation is better conditioned and always well-posed. It is important to note that each iteration in the GMRES algorithm requires in fact $6 + 3 = 9$ FFM-products in our implementation: three for the part with the scalar product of the normal vectors, three for the part with the scalar product of the surface rotational, and three for the adjoint of the double layer operator.

The submarine is 60 m long and the frequency is set at 6.5 kHz. The source term is a *plane wave* propagating along the axis of the submarine. Assuming, the celerity of sound in water is 1500 m.s^{-1}, the wavelength is 0.231 m meaning that there are approximately 260 wavelengths along the submarine, or differently written we have $k\cdot r_{\max}\approx 1687$. The mesh features $12.8\cdot 10^6$ triangles. Since the numerical integration is performed using a quadrature rule with 3 nodes per triangle, the size of one FFM-product is $38.4\cdot 10^6 \times 38.4\cdot 10^6$. The problem is discretized with $\mathcal{P}^1$-elements implying that the total amount of (nodal) unknowns is

[1]) The high-level interface with the FFM is available as an other overloaded `integral()` function within Gypsilab. An example is provided by running the nrtFfmBuilderFem.m script.

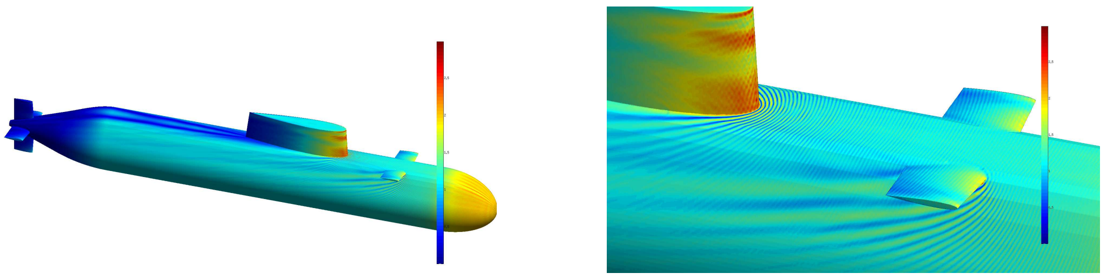
Fig. 4.4. Absolute value of μ on the surface of the submarine.

$6.4 \cdot 10^6$. It is solved using a preconditioned[1] GMRES algorithm without restart on a 32 cores server at 3.0 GHz with 512 GBytes RAM. The tolerance for both the GMRES and the FFM product is set to $\varepsilon = 10^{-3}$. Convergence is achieved in 12 iterations and 50000 seconds ($\approx$ 13 hours and 50 minutes) including the assembling of the preconditioner and the regularization matrix. Each iteration requires approximately 3570 seconds. The radiated field on the surface is represented on Fig. 4.4. This computation is also performed using the FFM. The memory peak is measured at approximately 200 GBytes when assembling the preconditioner and the regularization matrix[2] .

4.2.2. Perfect electric rocket launcher

This example is a slightly modified version of a test case extracted from the Workshop EM-ISAE 2018, see [24]. We study the scattering of an electromagnetic plane wave by a perfect electric launcher (the plane wave is propagating along the axis of the launcher). This problem is solved the Combined Field Integral Equation (CFIE) which is a linear combination of the Electric Field Integral Equation (EFIE) and the Magnetic Field Integral Equation (MFIE). For the construction of these equations, we refer once again to [2] starting p. 234, or [3] starting p. 108. The EFIE reads

$$\frac{1}{k^2}\nabla_\Gamma \int_\Gamma G(x,y)\,\nabla_\Gamma \cdot \mathbf{J}(y)\,d\gamma_y + \left(\int_\Gamma G(x,y)\mathbf{J}(y)\,d\gamma_y\right)_T = -\frac{(\mathbf{E}^{\mathrm{i}})_T}{ikZ}, \tag{4.15}$$

where $\mathbf{J}$ is the tangential trace of the magnetic field, $\mathbf{E}^{\mathrm{i}}$ is the incident electromagnetic wave, Z is an impedance, and $(.)_T$ is the *tangential trace* operator. The MFIE reads

$$\frac{(\mathbf{J}\times n_x)\,(x)}{2} + n_x \times \int_\Gamma \nabla_y G(x,y) \times \mathbf{J}(y)\,d\gamma_y = n_x \times \mathbf{H}^{\mathrm{i}}(x), \tag{4.16}$$

where $\mathbf{H}^{\mathrm{i}}$ is the incident magnetic field. The CFIE then reads

$$\mathrm{CFIE} = \alpha \cdot \mathrm{EFIE} + (1-\alpha)\cdot Z \cdot \mathrm{MFIE}, \qquad \alpha \in [0,1]. \tag{4.17}$$

[1] The system is preconditioned by performing an Incomplete LU decomposition of the (sparse) matrix of the singular or close-to-singular interactions.

[2] When assembling the BEM matrix, one issue which needs to be addressed is the computation of the singular integrals. One way to do that is to compute the matrix as if everything where regular using standard quadratures, then add a regularization matrix which contains the accurate computation of the singularity, for example using semi-analytical formulas.

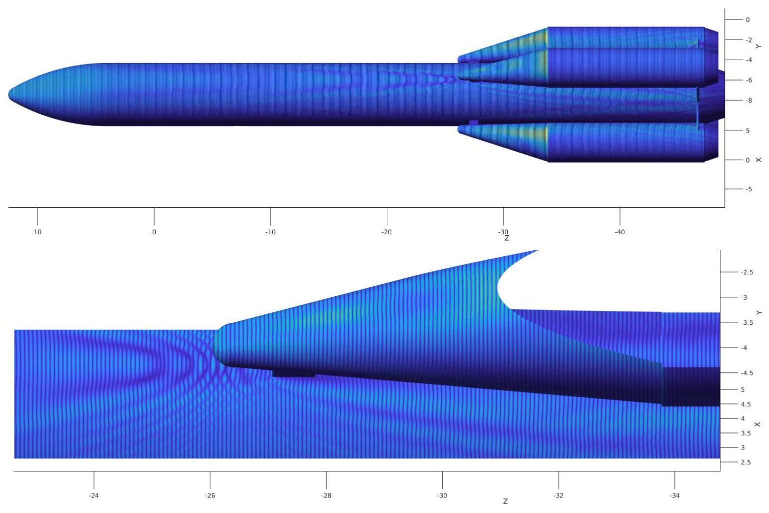

Fig. 4.5. Real part of **J** on the whole launcher (top) and detail (bottom).

We use the H_{div}-conforming Raviart-Thomas finite element space of order 0 (RT_0) and we choose $\alpha = 0.5$. The launcher is 60 m long and 12 m in diameter (including the boosters). The electromagnetic wave propagates at 2 GHz meaning that the wavelength is 0.15 m, assuming a celerity of light equals to $3 \cdot 10^8$ m.s^{-1}. Therefore, there are 400 wavelength along the launcher. The total number of unknowns is $60 \cdot 10^6$ for approximately $38 \cdot 10^6$ triangles. Consequently, the size of a FFM product is approximately $114 \cdot 10^6 \times 114 \cdot 10^6$. We use the same server as in subsubsection 4.2.1. One GMRES matrix-vector product involves now 10 (4 for the EFIE, 6 for the MFIE) FFM products and lasts approximately 4 hours for the prescribed accuracy for the FFM is $\varepsilon = 10^{-3}$. A GMRES convergence at 10^{-2} accuracy is achieved in approximately 6 days and 35 iterations. The real part of the solution is represented on Fig. 4.5.

5. Conclusions

We have proposed a powerful, scalable and easy-to-implement version of the well-known Fast Multipole Method. The simple inner structure makes possible the use of the NUFFT algorithm in the case of oscillating kernels. While probably not as fast as other implementations of the FMM available in the industry, the FFM is still capable of dealing with convolution products featuring dozens, or eventually hundreds, of millions of entries. We are even able to reach 10^9 nodes for the Laplace kernel on a modest server. We prove quasi-linear complexity estimates in the general case and a $\mathcal{O}(N^{3/2} \cdot \log^2(N))$ complexity for the oscillating kernel case. Regarding this last case, such scaling is not observed until the very high frequency regime (multiple dozens of millions of entries). Plugged into BEM methods, it is possible to solve problems featuring dozen of millions of nodes on a laboratory server at low- and high-frequencies. A powerful advantage of the FFM is that it can be implemented at a minimum cost using only standard algorithms provided in all modern programing languages, to the exception of the NUFFT. For example, our implementation requires as little as a few hundred Matlab lines. For the sake of reproducibility, this implementation is provided within the git repository of Gypsilab at [21] under the GPL 3.0 license.

Acknowledgments. We would like to thank Houssem Haddar for his help and Leslie Greengard for providing the NUFFT code used for the computation of the numerical results. This work was funded by the DGA-AID (Grant No. 2018600074).

References

[1] W. Hackbusch, *Hierarchical Matrices: Algorithms and Analysis*, Springer (2015).

[2] J.-C. Nédélec, *Acoustic and Electromagnetic Equations: Integral Representations for Harmonic Problems*, Springer, 2001.

[3] D.L. Colton and R. Kress, *Integral Equation Methods in Scattering Theory*, Malabar Fla. Krieger Pub. Co. (1992).

[4] R. Kress, Minimizing the condition number of boundary integral equations in acoustic and electromagnetic scattering, *Q. Jl Mech. appl. Math.*, **38** Pt. 2 (1985), 323–396.

[5] M. Bebendorf, Adaptive Cross Approximation of Multivariate Functions, *Constr. Approx.*, **34**:2 (2011), 149–179.

[6] S. Börm, L. Grasedyck and W. Hackbusch, *Hierarchical Matrices*, Max-Planck Gesellschaft, 2015.

[7] A.H. Barnett, J.F. Magland and L. af Klinterberg, A parallel non-uniform fast Fourier transform library based on an "exponential of semi-circle" kernel, *SIAM J. Sci. Comput.*, **41**:5 (2018), 479–504.

[8] A. Dutt and V. Rokhlin, Fast Fourier Transform for Nonequispaced data, *SIAM J. Sci. Comput.*, **14** (1993).

[9] L. Greengard and J.Y. Lee, Accelerating the Nonuniform Fast Fourier Transform, *SIAM Rev.*, **46**:3 (2004), 443–454.

[10] L. Greengard and J. Huang, A new version of the Fast Multipole Method for Screened Coulomb Interactions in Three Dimensions, *J. Comput. Phys.*, **180** (2002), 642–658.

[11] L. Greengard, J. Huang and V. Rokhlin, Accelerating fast multipole methods for the Helmholtz equation at low frequencies, *IEEE Comput. Sci. Eng.*, **5**:3 (1998), 32–38.

[12] E. Darve and P. Havé, Efficient fast multipole method for low-frequency scattering, *J. Comput. Phys.*, **197**:1 (2004), 341–363.

[13] L. Greengard and V. Rokhlin, A Fast Algorithm for Particle Simulations, *J. Computut. Phys.*, **73** (1987), 325–348.

[14] H. Cheng, L. Greengard and V. Rokhlin, A Fast Adaptive Multipole Algorithm in Three Dimensions, *J. Comput. Phys.*, **155** (1999), 468–498.

[15] F. Alouges and M. Aussal, The sparse cardinal sine decomposition and its application for fast numerical convolution, *Numer. Algorithms*, **70** (2015), 427–448.

[16] Q. Carayol and F. Collino, Error estimates in the Fast Multipole Method for scattering problems Part 2: Truncation of the Gegenbauer series, *ESAIM: M2AN*, **39**:1 (2005), 183–221.

[17] E. Darve, The Fast Multipole Method: Numerical Implementation, *J. Comput. Phys.*, **160** (2000), 195–240.

[18] E. Darve, The Fast Multipole Method I: Error Analysis and Asymptotic Complexity, *SIAM J. Numer. Anal.*, **38**:1 (2000), 98–128.

[19] L. Cambier and E. Darve, Fast Low-Rank Kernel Matrix Factorization through Skeletonized Interpolation, *SIAM J. Sci. Comput.*, **41**:3 (2017).

[20] F. Alouges and M. Aussal, FEM and BEM simulations with the Gypsilab framework, *The SMAI Journal of Computational Mechanics*, **4** (2018), 297–318.

[21] Gypsilab, https://github.com/matthieuaussal/gypsilab/ (on-coming native C++ version at https://github.com/marcbakry/ffm-cpp)

[22] G. Mastroianni and D. Occorsio, Optimal systems of nodes for Lagrange interpolation on bounded intervals. A survey, *J. Comput. Appl. Math.*, **134** (2001), 325–341.

[23] W. Fong and E. Darve, The black-box fast multipole method, *J. Comput. Phys.*, **228**:23 (2009), 8712–8725.

[24] Workshop EM-ISAE 2018, https://websites.isae-supaero.fr/workshop-em-isae-2018/

[25] L. Greengard, https://cims.nyu.edu/cmcl/fmm3dlib/fmm3dlib.html

[26] C.W. Nell and L.E. Gilroy, An improved BASIS model for the BeTSSi submarine, *DRDC Atlantic TR*, **199** (2003).

[27] C. Dapogny, C. Dobrzynski and P. Frey, Three-dimensional adaptive domain remeshing, implicit domain remeshing, and applications to free and moving boundary problems, *J. Comput. Phys.*, **262** (2014), 358–378.

[28] Y. Saad and M.H. Schultz, GMRES: A generalized minimal residual algorithm for solving non-symmetric linear systems, *SIAM J. Sci. Comput.*, **7**:3 (1986), 856–869.

[29] L. Ying, G. Biros and D. Zorin, A kernel-independent adaptive fast multipole algorithm in two and three dimensions, *J. Comput. Phys.*, **196** (2004), 591–626.

[30] K.L. Ho and L. Ying, Hierarchical interpolative factorization of Elliptic Operators: Integral Equations, *Commun. Pur. Appl. Math.*, **69**:7 (2016), 1314–1353.

[31] S. Börm, *Efficient Numerical Methods for Non-local Operators*, European Mathematical Society, 2010.