Delft University of Technology
Faculty Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

GPU Acceleration Of The PWTD Algorithm For Application In High-Frequency Communication And Fotonics

Literature Report for the
Delft Institute of Applied Mathematics
as part of

the degree of

MASTER OF SCIENCE
in
APPLIED MATHEMATICS

by

Rory Gravendeel

Delft, the Netherlands
April 2019

Copyright © by Rory Gravendeel. All rights reserved.

TU Delft

MSc Literature report APPLIED MATHEMATICS

"GPU Acceleration Of The PWTD Algorithm For Application In High-Frequency Communication And Fotonics"

RORY GRAVENDEEL

Delft University of Technology

Thesis advisor

Dr. K. Cools

Members of the graduation committee

Prof.dr.ir. C. Vuik

April 2019 Delft

Contents

Introduction

This report contains the Literature Studies for the Master Thesis, for the master Applied Mathematics at Delft University of Technology, written by Rory Gravendeel and supervised by Kristof Cools (Delft University of Technology) and Rens van Driel (Capgemini). Rory is working on his thesis at the Numerical Analysis department of Applied Mathematics, whilst also working as an intern at Capgemini.

The title of the thesis is 'GPU acceleration of the PWTD algorithm for application in high-frequency communication and fotonics'. Scatter-like physical phenomena, like the high-frequency communication and fotonics, are quite difficult to model, due to the high computational cost that for instance a Marching-On-in-Time algorithm would require. To reduce the computational cost and computational time, the thesis will explore applying the Plane-Wave Time-Domain algorithm to MOT and run it on GPUs. PWTD will be applied in both a two-level and a multilevel algorithm. Further optimization will also be done to the algorithm via FFT optimization and parallel programming.

The report is structured as follows. In chapter 1 the main problem of the thesis will be sketched. The second chapter will look at some of the mathematical theory necessary to develop the Plane-Wave Time-Domain algorithm. This includes the Marching-On-In-Time method, the Fast Multipole Method and some additional theory. The third chapter will look at the Plane-Wave Time-Domain algorithm itself. It will look at the algorithm and some implementation issues, before applying it to the Marching-On-In-Time method in both a two-level and multilevel algorithm. Chapter 4 concerns itself with Fourier transforms. PWTD contains many convolutions, for which we will use the Fast Fourier Transform to solve them efficiently. The chapter will discuss the basics of Fourier transforms, FFT algorithms and the Fast Fourier Transform of the West. It will also discuss convolution and how we can optimize FFTW. In chapter 5 we will look at programming. This will include a look at Julia, the programming language the PWTD algorithm will be developed in. It will also look at GPU programming since that will be a core part of optimizing the algorithm. Chapter 6 will discuss the research direction of the thesis, including research questions and further areas that can be developed during the thesis. It will also give a short introduction into Capgemini and the hardware that will be used to develop the algorithm on.

Chapter 1

Problem Sketch

In this chapter, we will give a brief description of the problem situation for this thesis. Many devices, such as antenna's and aeroplanes, experience electromagnetic fields whilst they are operative. It is imperative for developers of such devices to know what happens when these fields interact with their devices and how the field might scatter after collision. We will look at a mathematical approach to this and see how we can solve this numerically with the lowest possible computational costs and computational time.

Assume we have a scatterer bounded by a surface S as below (the device). We then consider an incident field $u^i(\mathbf{r}, t)$ that is fired upon the scatterer, where we assume that u^i is temporally bandlimited by $\omega_{\max}$ (i.e. it is bounded in the time-domain). Figure 1.1 below sketches the situation nicely.

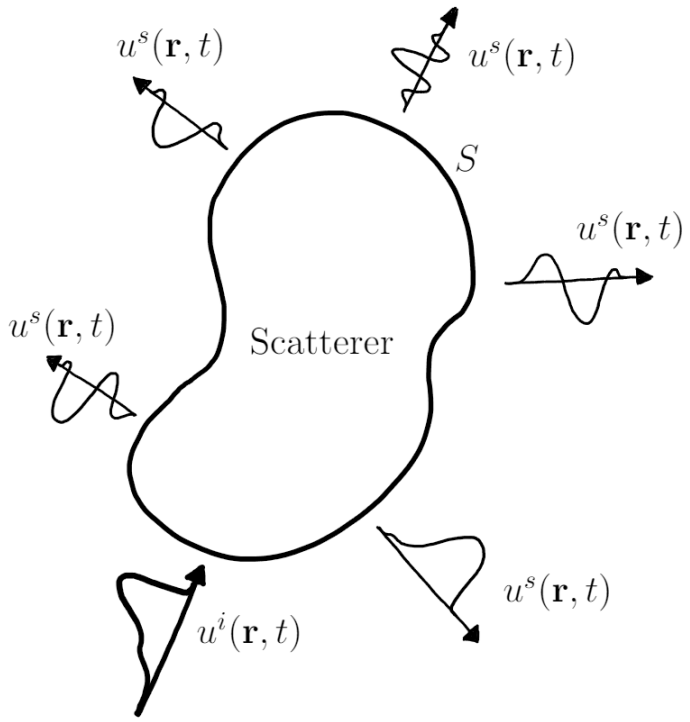

Figure 1.1: Surface scattering problem.

Here $\mathbf{r}$ is the directional vertor in the $x-y-z$-domain and t the time. When u^i interacts

with S it creates a scattered field denoted by $u^s(\mathbf{r}, t)$. The total field is then $u(\mathbf{r}, t) = u^i(\mathbf{r}, t) + u^s(\mathbf{r}, t)$.

We wish to determine the total field for all t. Note that the total field satisfies the wave equation

$$\nabla^2 u(\mathbf{r}, t) - \frac{\partial^2}{\partial t^2} \frac{1}{c^2} u(\mathbf{r}, t) = 0 \tag{1.1}$$

Using the boundary condition, which is assumed to be a Dirichlet boundary condition on S, we find that u^s can be represented in terms of surface sources $q(\mathbf{r}, t)$ on S such that

$$u^s(\mathbf{r}, t) = \int_S d\mathbf{r}' \frac{\delta(t - R/c)}{4\pi R} * q(\mathbf{r}', t) \tag{1.2}$$

We can solve this numerically with a Marching-On-In-Time algorithm, which we can then speed up by applying the Plane-Wave Time-Domain algorithm.

As we will see in chapter 3 and can already note in 1.2, there will be quite a lot of convolutions involved in the PWTD algorithm. Convolutions can be quite cumbersome to compute numerically, therefore we will use Fast Fourier Transforms to compute these more efficiently. A GPU will be able to do this in parallel, hence the interest in implementing the Plane-Wave Time Domain algorithm on a GPU, as this can reduce the time the algorithm takes. These parts will be worked out in the following chapters.

Chapter 2

Mathematical Theory

2.1 Introduction

In this chapter we will look at mathematical theorems and algorithms that do not merit their own separate chapter, but are necessary for the development and evaluation of the Plane-Wave Time-Domain algorithm. We start with developing the Marching-On-In-Time algorithm, which we do by looking at the Finite Element Method as a starting point and developing the MOT algorithm from there. Next, we look at the Fast Multipole Method. Lastly, we will discuss some additional theory like Bessel functions and Quadrature rules.

2.2 Marching-On-In-Time Algorithm

For this thesis, we will apply the Plane-Wave Time-Domain to the Marching-On-In-Time algorithm. The MOT-algorithm, as it is commonly abbreviated to, works in the same way as the Finite Element Method, only then with added temporal basis functions to accommodate the time-variable in equations. Before discussing the MOT algorithm in depth, we will look at FEM first to discuss its main building blocks, which will then be expanded upon for the MOT algorithm. In both cases an example or two will be used to illustrate the process of each algorithm.

2.2.1 Finite Element Method

The Finite Element Method is a widely-used algorithm that can be applied in a wide range of problems, especially in for instance problems with complex geometries. The main situation of a FEM problem can be sketched as follows:

Assume we have a surface with a boundary over which we wish to determine the solution or evaluate a system of partial differential equations with a boundary value problem. The surface is denoted by Ω, its boundary by $\partial\Omega$. A vector $\mathbf{n}$ is the normal vector perpendicular to the boundary, with $||\mathbf{n}|| = 1$.

To work through the algorithm, we introduce a Boundary Value Problem (BVP):

$$\begin{cases} -\Delta u + u = f(x,y) & \text{in } \Omega \\ \frac{\partial u}{\partial n} = g(x,y) & \text{on } \partial\Omega \end{cases}$$

First we multiply the system by a test function φ, where φ is chosen as a continuous function. We then obtain

$$(-\Delta u + u)\varphi = \varphi f$$

Next, we integrate over Ω to obtain

$$\int_\Omega \varphi(-\Delta u + u)d\Omega = \int_\Omega \varphi f d\Omega$$

By applying Integration by Parts, the Divergence Theorem of Gauss and the natural boundary condition, we obtain the weak formulation

$$\int_\Omega \nabla u \nabla \varphi + u\varphi d\Omega = \int_\Omega f\varphi d\Omega + \int_{\partial\Omega} g\varphi d\Gamma$$

We then apply Galerkin's Method. Here we set

$$u(x,y) = \sum_{j=1}^{\infty} c_j\varphi_j(x,y) \simeq \sum_{j=1}^{n} c_j\varphi_j(x,y) = u^n(x,y)$$

We assume $\{\varphi_j(x,y)\}_{j=1}^n$ is a basis, so the φ_j are linearly independent.

Next, we divide Ω and $\partial\Omega$ into meshpoints, as figure 2.1 below illustrates. The meshpoints are numbered.

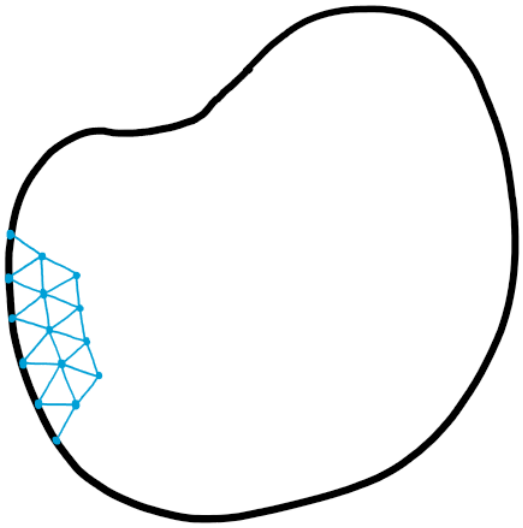

Figure 2.1: Surface divided into meshpoints

For each gridnode, we assume that φ_i belongs to node i and φ_i is piecewise polynomial. Furthermore, we have

$$\varphi_i(x_j, y_j) = \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

We then add the u determined above for the Galerkin method into the weak formulation, where we then find

$$\sum_{j=1}^{n} c_j \left[\int_\Omega \nabla\varphi_i \nabla\varphi_j + \varphi_i\varphi_j d\Omega \right] = \int_\Omega \varphi_i f d\Omega + \int_\Omega \varphi_i g d\Gamma \qquad \forall i \in \{1, \ldots, n\}$$

We can set the right-hand side as b_i and the part between the []-brackets as S_{ij} and rewrite the system as

$$\sum_{j=1}^{n} S_{ij} c_j = b_i$$

Depending on the chosen elements, S_{ij} is then evaluated over each element before the final system is determined.

There are various methods that can help in the last step, such as the Holand and Bell Theorem over triangular elements, combined with Newton-Côtes. The type of element can also be varied, one could opt for extra nodes per triangle or for quadratic elements such as Taylor-Hood elements. The choice of basis functions also plays an important role. Note that this walkthrough of the Finite Element Method is based upon notes for the course WI4450 Special Topics given bij Fred Vermolen in 2018.

2.2.2 Marching On

The Marching-On-In-Time algorithm follows the same pattern, where we add extra time functions to the Galerkin part of FEM. We will change the notation so it is more aligned to the notation of the original problem. See also section 18.2 of [1].

Let $q(\mathbf{r}, t)$ be an unknown source density. We have $u^i(\mathbf{r}, t)$ as the field incident on the scatterer, which is bounded by a surface S as described in the problem sketch section. A scattered field is generated, denoted $u^s(\mathbf{r}, t)$. The total field is then $u(\mathbf{r}, t) = u^i(\mathbf{r}, t) + u^s(\mathbf{r}, t)$, which satisfies the wave equation:

$$\nabla^2 u(\mathbf{r}, t) - \frac{\partial^2}{\partial t^2} \frac{1}{c^2} u(\mathbf{r}, t) = 0$$

We can represent the incident and scattered fields in terms of equivalent surface sources $q(\mathbf{r}, t)$ that reside on S, such that we have

$$\begin{aligned} u^s(\mathbf{r}, t) &= \int_S d\mathbf{r}' \frac{\delta(t - R/c)}{4\pi R} * q(\mathbf{r}', t) \\ -u^i(\mathbf{r}, t) &= \int_S d\mathbf{r}' \frac{\delta(t - R/c)}{4\pi R} * q(\mathbf{r}', t) \qquad \forall \mathbf{r} \in S \end{aligned}$$

To solve this numerically, we represent $q(\mathbf{r}, t)$ in terms of spatial and temporal basis functions. These are $f_n(\mathbf{r}), n = 1, \ldots, N_s$ and $T_i(t), i = 0, \ldots, N_t$ respectively. We then rewrite q as

$$q(\mathbf{r}, t) = \sum_{n=1}^{N_s} \sum_{i=0}^{N_t} q_{n,i} f_n(\mathbf{r}) T_i(t)$$

Here $q_{n,i}$ represent unknown expansion coefficients, like the c_j in FEM.

Next we substitute $q(\mathbf{r}, t)$ in its basis functions form into the equation for $u^i(\mathbf{r}, t)$. We then test the resulting equation at time $t = t_j = j\Delta t$ with test function $\tilde{f}_m(\mathbf{r})$ for $m = 1, \ldots, N_s$. We obtain a matrix equation for the system:

$$\bar{Z}Q_j = U_j^i - \sum_{k=1}^{j-1} \bar{Z}_k Q_{j-k} \tag{2.1}$$

where

- The m-th element of vector Q_j is given by $q_{m,j}$
- The m-th element of vector U_j^i is given by $-\int_S d\mathbf{r} \tilde{f}_m(\mathbf{r}) u^i(\mathbf{r}, t_j)$
- $$\bar{Z}_{k,mn} = \int_S d\mathbf{r} \tilde{f}_m(\mathbf{r}) \int_S d\mathbf{r}' f_n(\mathbf{r}') \left[\frac{\delta(t - R/c)}{4\pi R} * T_{j-k}(t) \right] |_{t=t_j} \tag{2.2}$$

With this equation we can determine the expansion coefficients $q_{n,j}$ by starting at the first time step $j = 0$, from which we can determine the next value of the above equation per time step.

We see that the setup is similar to that of the Finite Element Method, but with an added temporal basis function. The evaluation process also differs and can be computationally expensive. To reduce the computational costs we apply the Plane-Wave Time-Domain algorithm. Do note that, unlike with FEM, only the boundary is turned into a mesh since we are dealing with a Boundary Element problem, where only the boundary of the scatterer affects the system.

2.2.3 Rao-Wilton-Glisson Basis Functions

For Computational Electromagnetics, the Rao-Wilton-Glisson functions are often used when applying FEM or MOT. We will discuss them here shortly. Assume the surface Ω is discretized into triangles, as in figure 2.2. We follow the definition of the RWS basis functions from [2].

Assume T_n^+ and T_n^- are two triangles of the discretization, with edge n in common. See also figure 2.2. We then define two mutually orthogonal basis functions associated with this n-th edge:

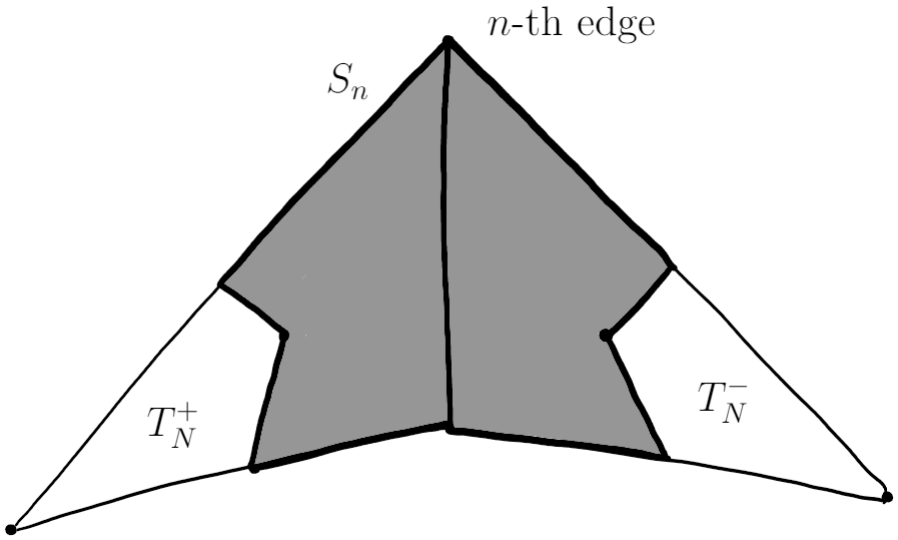

Figure 2.2: Rao-Wilton-Glisson Basis functions

$$\boldsymbol{f}_n(\mathbf{r}) = \begin{cases} \boldsymbol{a}_n^{\pm} \times \hat{\boldsymbol{\ell}}, & \mathbf{r} \in S_n \\ 0, & \text{otherwise} \end{cases} \tag{2.3}$$

$$\boldsymbol{g}_n(\mathbf{r}) = \begin{cases} \hat{\boldsymbol{\ell}}, & \mathbf{r} \in S_n \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

Here S_n is created by connecting the midpoints of the edges of each triangle that is not edge n to the centroid of the triangle, hereby representing an area with eight edges. This can also be seen in figure as the shaded black area. $\hat{\boldsymbol{\ell}}$ represents the unit vector along the n-th edge; $\boldsymbol{a_n^{\pm}}$ represents the unit normal vector to the plane of the triangle $T_n^{\pm}$. Most of the time the functions of 2.4 won't be used, as they represent pulse functions.

2.3 Fast Multipole Method

In this section we will look at the Fast Multipole Method, following [3]. We work from the matrix Z of 2.1. For scattering problems, the matrix elements can be written as

$$Z_{nn'} = -i \int_S d^2\mathbf{x} \int_S \mathbf{x}' f_n(\mathbf{x}) \frac{e^{ik|\mathbf{x}-\mathbf{x}'|}}{4\pi|\mathbf{x}-\mathbf{x}'|} f_n(\mathbf{x}') \tag{2.5}$$

Here f_n are again the basis functions. We see that this is comparable to 2.2. We will use the following elementary identities to rewrite 2.5:

$$\frac{e^{ik|\mathbf{X}+\mathbf{d}|}}{|\mathbf{X}+\mathbf{d}|} = ik \sum_{l=0}^{\infty} (-1)^l (2l+1) j_l(kd) h_l^{(1)}(kX) P_l(\hat{d} \cdot \hat{X}) \tag{2.6}$$

$$\int d^2\hat{k} e^{i\mathbf{k}\cdot\mathbf{d}} P_l(\hat{k} \cdot \hat{X}) = 4\pi i^l j_l(kd) P_l(\hat{d} \cdot \hat{X}) \tag{2.7}$$

We then find

$$Z_{nn'} \approx \frac{k}{(4\pi)^2} \int_S d^2\mathbf{x} f_n(\mathbf{x}) \int_S d^2\mathbf{x}' f_{n'}(\mathbf{x}') \int d^2\hat{k} e^{i\mathbf{k}\cdot(\mathbf{x}-\mathbf{x}'-\mathbf{X})} \mathcal{T}_L(kX, \hat{k}\cdot\hat{X}) \tag{2.8}$$

where

$$\mathcal{T}_L(\kappa, \cos\theta) \equiv \sum_{l=0}^{L} t^l (2l+1) h_l^{(1)}(\kappa) P_l(\cos\theta) \tag{2.9}$$

The Legendre polynomials and Bessel functions found in 2.8 and 2.9 can be found in the next section.

2.4 Additional Theory

Before the Plane-Wave Time-Domain algorithm can be implemented, we must look at some additional mathematical theory. These theorems and functions will be used in the next chapter for the implementation of PWTD.

2.4.1 Bessel Functions

The Bessel functions [5] are the solutions of Bessel's differential equation:

$$z^2 \frac{d^2w}{dz^2} + z\frac{dw}{dz} + (z^2 - m^2)w = 0 \tag{2.10}$$

Here m is an arbitrary complex number, which is also known as the order of the Bessel function. Of special interest are the cases that m is an integer or a half-integer, since these functions appear in the solutions of Laplace's equation in cylindrical coordinates respectively the solutions of the Helmholtz equation in spherical coordinates.

There are two kinds of Bessel functions, of the first kind and of the second kind. These are, in the case of cylindrical coordinates,

$$J_m(z) = \left(\frac{1}{2}z\right)^m \sum_{k=0}^{\infty} (-1)^k \frac{\left(\frac{1}{4}z^2\right)^k}{k!\,\Gamma(m+k+1)} \tag{2.11}$$

$$Y_m(z) = \frac{J_m(z)\cos(m\pi) - J_{-m}(z)}{\sin(m\pi)} \tag{2.12}$$

Here Γ is the gamma-function, which is equal to

$$\Gamma(n) = (n-1)!$$

In the case of spherical coordinates and assuming that m is a positive integer, the Bessel functions become

$$\mathbf{j}_m(z) = \sqrt{\frac{1}{2}\pi/z} J_{m+\frac{1}{2}}(z) \tag{2.13}$$

$$\mathbf{y}_m(z) = \sqrt{\frac{1}{2}\pi/z} Y_{n+\frac{1}{2}}(z) \tag{2.14}$$

2.4.2 Legendre Polynomials

For a variable x, we can write the Legendre polynomial [4] as

$$P_n(x) = \frac{1}{2^n} \sum_{k=0}^{n} \binom{n}{k}^2 (x-1)^{n-k}(x+1)^k \tag{2.15}$$

Legendre polynomials can be useful for expanding $1/r$-potentials and for multipole expansions.

2.4.3 L^2 inproduct

Assume that f and g are two real functions on a measure space X with respect to a measure μ. Then the L^2-inner product [6] is given by

$$\langle f, g \rangle = \int_X fg \; d\mu \tag{2.16}$$

2.4.4 Parseval's Theorem and Identity

Marc-Antoine Parseval was a mathematician who developed various theorems in mathematical analysis. Two of these will be discussed below that require Fourier series or transforms. For Fourier transforms, see chapter 5.

For Parseval's Theorem [9], assume that $A(x)$ and $B(x)$ are two square integrable (w.r.t. Lebesque measure), complex values functions on $\mathbb{R}$ of period 2π with a Fourier series, denoted by

$$A(x) = \sum_{n=-\infty}^{\infty} a_n e^{inx}$$
$$B(x) = \sum_{n=-\infty}^{\infty} b_n e^{inx}$$

$$\sum_{n=-\infty}^{\infty} a_n \overline{b_n} = \frac{1}{2\pi} \int_{-\pi}^{\pi} A(x)\overline{B(x)} dx \tag{2.17}$$

Here $\overline{x}$ denotes the complex conjugate of x.

For Parseval's Identity [10], assume that function f is square integrable. We then have

$$\|f\|^2 = \int_{-\pi}^{\pi} |f(x)|^2 \, dx = 2\pi \sum_{n=-\infty}^{\infty} |c_n|^2 \tag{2.18}$$

$$c_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) e^{inx} dx \tag{2.19}$$

2.4.5 Quadrature rules

In numerical analysis and, in particular, numerical integration, we wish to compute an approximate solution to a definite integral $\int_a^b f(x)dx$. To approximate this, we use so-called quadrature rules; the term is derived from the historical mathematical term to calculate area.

We can define a quadrature rule Q as follows [7]:

$$\int_a^b f(x)dx = \sum_{j=1}^{N} w_j f(x_j) + E(f) \tag{2.20}$$

Here w_j are the weights of the rule, s_j are the quadrature points (also known as nodes) and $E(f)$ is the error term.

An example of this is the trapezoidal rule [8], which can be expressed as follows:

$$\int_a^b f(x)dx \approx \sum_{j=1}^{N} \frac{f(x_{j-1}) + f(x_j)}{2} \Delta x_j \tag{2.21}$$

Another quadrature rule that is often used with the Finite Element Method is the Newton-Côtes rule, which is denoted as

$$\int_a^b f(x)dx = (b-a) \sum_{j=1}^{N} w_j f(x_j) \qquad \text{where } \sum_{j=1}^{N} w_j = 1 \tag{2.22}$$

Chapter 3

Plane-Wave Time-Domain Algorithm

3.1 Introduction

In this chapter, we will look at the Plane-Wave Time Domain algorithm, that will be at the heart of this thesis. We will start by looking at the PWTD algorithm itself. After that, we will apply it to the Marching-On-In-Time algorithm from chapter 2 and present a two-level and multilevel implementation of this PWTD-enhanced MOT scheme. At the end of the chapter some notes are included that link to other parts of the report.

3.2 The Algorithm

In this section we will introduce the Plane-Wave Time-Domain algorithm, which will express transient fields in terms of plane waves. We follow the steps in [1] since it is the main source for this thesis.

The algorithm requires all source signals to be of a limited duration, which works since the source signal $q(\mathbf{r}, t)$ can always be broken up into subsignals

$$q(\mathbf{r}, t) = \sum_{l=1}^{L} q_l(\mathbf{r}, t). \tag{3.1}$$

Each subsignal q_l is identically zero outside of an interval $(l-1)T_s \leq t \leq lT_s$, where T_s is the duration of the subsignals. The field radiated by source q_l can then be denoted by $u_l(\mathbf{r}, t)$; we can express the total field by

$$u_l(\mathbf{r}, t) = \sum_{l=1}^{L} u_l(\mathbf{r}, t) \tag{3.2}$$

The relation between the subsignal and subfield is then

$$u_l(\mathbf{r}, t) = \int_S d\mathbf{r}' \frac{\delta(t - R/c)}{4\pi R} * q_l(\mathbf{r}, t) \tag{3.3}$$

3.2.1 Plane Wave Decomposition

We first wish to determine a suitable plane wave representation for our transient wave fields. This step is motivated by frequency domain fast multipole schemes. We consider the expression

$$\tilde{u}_l(\mathbf{r},t) = -\frac{\partial_t}{8\pi^2 c}\int_0^{\theta_{\text{int}}} d\theta \sin\theta \int_0^{2\pi} d\phi \int_S d\mathbf{r}' \delta\left[t - \hat{\mathbf{k}}\cdot(\mathbf{r}-\mathbf{r}')\Big/c\right] * q_l(\mathbf{r}',t) \tag{3.4}$$

Here $\hat{\mathbf{k}} = \hat{\mathbf{x}}\sin\theta\cos\phi + \hat{\mathbf{y}}\sin\theta\sin\phi + \hat{\mathbf{z}}\cos\theta$ is a unit direction vector. The integral over S is a projection of the source distribution q_l onto a plane wave travelling in the $\hat{\mathbf{k}}$ direction. Furthermore, if we set the upper limit of θ, θ_{int}, equal to π, then $\tilde{u}_l$ is expressed as a superposition of plane waves that travel in all directions. We can then find a necessary relation between u_l and $\tilde{u}_l$ by integrating over θ and ϕ in equation 3.4. To do this efficiently, we cast 3.4 into a new coordinate system $(x',y',z') \equiv (\rho',\theta',\phi')$ where we align the z'-axis with $\mathbf{R} = \mathbf{r} - \mathbf{r}'$. Figure 18.3 of [1] shows how the translation to the new coordinate system works in a 2D system. In this new coordinate system the upper limit on θ', θ'_{int}, is a function of $\phi', \mathbf{r}$ and $\mathbf{r}'$, so $\theta'_{\text{int}}(\phi',\mathbf{r},\mathbf{r}')$. Then

$$\tilde{u}_l(\mathbf{r},t) = -\frac{\partial_t}{8\pi^2 c}\int_S d\mathbf{r}' \int_0^{2\pi} d\phi' \int_0^{\theta'_{\text{int}}} d\theta' \sin\theta' \delta\left(t - \hat{\mathbf{k}}'\cdot\mathbf{R}'\Big/c\right) * q_l(\mathbf{r}',t) \tag{3.5}$$

Here $\hat{\mathbf{k}}' = \hat{\mathbf{x}}'\sin\theta'\cos\phi' + \hat{\mathbf{y}}'\sin\theta'\sin\phi' + \hat{\mathbf{z}}'\cos\theta'$ and $\mathbf{R}' = \tilde{\mathbf{z}}'|\mathbf{R}|$. We also assume $\theta_{\text{int}} > \cos^{-1}(\hat{\mathbf{z}}\cdot\mathbf{R}/R)$.
Next, we define $R = |\mathbf{R}|$, use $\hat{\mathbf{k}}'\cdot\mathbf{R}' = R\cos\theta'$ and set $\tau = R\cos\theta'/c$ in 3.5, from which we obtain

$$\begin{aligned}
\tilde{u}_l(\mathbf{r},t) &= -\int_S d\mathbf{r}' \int_0^{2\pi} d\phi' \int_{-(R/c)\cos\theta'_{\text{int}}}^{R/c} d\tau \frac{\partial_t \delta(t-\tau)}{8\pi^2 R} * q_l(\mathbf{r}',t) \\
&= \int_S d\mathbf{r}' \frac{\delta(t-R/c)}{4\pi R} * q_l(r r',t) - \int_S d\mathbf{r}' \frac{\delta(t+R\cos\theta'_{\text{int}}/c)}{4\pi R} * q_l(\mathbf{r}',t) \\
&= u_l(\mathbf{r},t) - \int_S d\mathbf{r}' \frac{\delta(t+R\cos\theta'_{\text{int}}/c)}{4\pi R} * q_l(\mathbf{r}',t)
\end{aligned} \tag{3.6}$$

In the special case that $\theta_{\text{int}} = \pi$, 3.6 reduces to

$$\tilde{u}_l(\mathbf{r},t) = u_l(\mathbf{r},t) - \int_S d\mathbf{r}' \frac{\delta(t+R/c)}{4\pi R} * q_l(\mathbf{r}',t) \tag{3.7}$$

The second term in equations 3.6 and 3.7 is called the ghost term, which is anticausal. To time-gate out the ghost signal, we set $T_s < R/c$.

Next, we shall look at why the derivation of equation 3.4 helps us in the development of a fast algorithm for evaluating transient fields. Consider a source distribution confined to a sphere of radius R_s. Furthermore, consider a set of observers that are also located in a (different) sphere of the same radius. The centers of the source- and observer-sphere are denoted by $\mathbf{r}_s$ and $\mathbf{r}_o$, respectively. The vector connecting the two centers

is denoted by $\mathbf{R}_c = \mathbf{r}_o - \mathbf{r}_s$. We assume $R_c = |\mathbf{R}_c| > 2R_s$. Also note that the vector $\mathbf{r} - \mathbf{r}' = (\mathbf{r} - \mathbf{r}_o) - \mathbf{R}_c - (\mathbf{r}' - \mathbf{r}_s)$. We can then rewrite 3.4 into

$$\begin{aligned}\tilde{u}_l(\mathbf{r}, t) = &\int d^2\hat{\mathbf{k}}\delta\left[t - \hat{\mathbf{k}}\cdot(\mathbf{r} - \mathbf{r}_o)\Big/c\right] * \mathcal{T}(\hat{\mathbf{k}}, \mathbf{R}_c, t) \\ &* \int_S d\mathbf{r}'\delta\left[t + \hat{\mathbf{k}}\cdot(\mathbf{r}' - \mathbf{r}_s)\Big/c\right] * q_l(\mathbf{r}', t)\end{aligned} \tag{3.8}$$

Here we used that

$$\int d^2\hat{\mathbf{k}} = \int_0^{\pi} d\theta \sin\theta \int_0^{2\pi} d\phi \qquad \text{integration over the unit sphere}$$

$$\mathcal{T}(\hat{\mathbf{k}}, \mathbf{R}_c, t) = -\frac{\partial_t}{8\pi^2 c}\delta\left(t - \hat{\mathbf{k}}\cdot\mathbf{R}_C\Big/c\right) \qquad \text{translation function}$$

Now we shall look at a three-stage implementation of 3.8 to evaluate $\tilde{u}_l$.

1. We first perform the rightmost integration and convolution of 3.8, which is equal to

$$q_l^{\text{out}}(\hat{\mathbf{k}}, t) = \int_S d\mathbf{r}'\delta\left[t + \hat{\mathbf{k}}\cdot(\mathbf{r}' - \mathbf{r}_s)\Big/c\right] * q(\mathbf{r}, t) \tag{3.9}$$

   This operation is known as the slant stack transform (SST) of q_l. The quantities q_l^{out} can be interpreted as rays leaving the source sphere in direction $\hat{\mathbf{k}}$.

2. Next, we perform the second integration and convolution, so we evaluate

$$q_l^{\text{in}}(\hat{\mathbf{k}}, t) = \mathcal{T}(\hat{\mathbf{k}}, \mathbf{R}_c, t) * q_l^{\text{out}}(\hat{\mathbf{k}}, t) \tag{3.10}$$

   The operator $\mathcal{T}$ translates each outgoing ray from the source to the observer sphere. The q_l^{in} can be seen as rays entering the observer sphere.

3. Lastly, we perform the leftmost integration and convolution:

$$\tilde{u}_l(\mathbf{r}, t) = \int d^2\hat{\mathbf{k}}\delta\left[t - \hat{\mathbf{k}}\cdot(\mathbf{r} - \mathbf{r}_o)\Big/c\right] * q_l^{\text{in}}(\hat{\mathbf{k}}, t) \tag{3.11}$$

   This operation shifts all the projections of the incoming rays correctly to the observer.

Setting $T_s < (R_c - 2R_s)/c$ ensures that

$$u_l(\mathbf{r}, t) = \begin{cases} 0 & \text{for} \quad t < lT_s \\ \tilde{u}_l(\mathbf{r}, t) & \text{for} \quad t \geq lT_s \end{cases}$$

3.2.2 Implementation Issues

Reference [1] also highlights some implementation issues that need to be addressed before the algorithm can be implemented correctly.

Spatial integration

Whilst performing the integration of 3.9 and 2.2, make sure the correct quadrature rules are used; this depends entirely on the discretization function used.

Spectral integration

To numerically evaluate the fields using 3.8, we have to use the correct quadrature rules to perform the integration over the unit sphere. Three basic observations help whilst evaluating 3.8 numerically.

1. The integral in 3.8 without the translation function can be seen as the time-dependent radiation pattern of a source distribution enclosed in a sphere of radius $2R_s$. If we assume $q_l(\mathbf{r}, t)$ is temporarily bandlimited to a $\omega_s > \omega_{\max}$, we can represent this part of the integral in terms of spherical harmonics $Y_{km}(\theta, \phi)$ as

$$\begin{aligned} g(\hat{\mathbf{k}}, \mathbf{r}, t) &= \delta\left[t - \hat{\mathbf{k}} \cdot (\mathbf{r} - \mathbf{r}_o)\Big/c\right] * \int_S d\mathbf{r}' \delta\left[t + \hat{\mathbf{k}} \cdot (\mathbf{r}' - \mathbf{r}_s\Big/c\right] * q_l(\mathbf{r}', t) \\ &= \int_S d\mathbf{r}' \delta\left[t + \hat{\mathbf{k}} \cdot [(\mathbf{r}' - \mathbf{r}_s) - (\mathbf{r} - \mathbf{r}_o)]\Big/c\right] * q_l(\mathbf{r}', t) \\ &= \sum_{k=0}^{K} \sum_{m=-k}^{k} g_{km}(\mathbf{r}, t) Y_{km}(\theta, \phi) \end{aligned} \tag{3.12}$$

   We set $K = \lceil \chi_1 2R_s\omega_s/c \rceil$ where $\chi_1 > 1$ is an excess bandwidth factor. This bandwidth ensures rapid convergence for the series in 3.12.

2. The translation function in 3.8 is only a function of the angle θ' between $\hat{\mathbf{k}}$ and $\mathbf{R}_c$. We can express it either in terms associated Legendre polynomials in θ' or in spherical harmonics in (θ, ϕ) as

$$\begin{aligned} &\mathcal{T}(\hat{\mathbf{k}}, \mathbf{R}_c, t) = \\ &= \begin{cases} -\frac{\partial_t}{16\pi^2 R_c} \sum_{k=0}^{\infty} (2k+1) P_k(ct/R_c) P_k(\cos\theta') & |t| \le R_c/c \\ 0 & \text{elsewhere} \end{cases} \end{aligned} \tag{3.13}$$

$$= \begin{cases} \sum_{k=0}^{\infty} \sum_{m=-k}^{k} \mathcal{T}_{km}(\mathbf{R}_c, t) Y_{km}(\theta, \phi) & |t| \le RC/c \\ 0 & \text{elsewhere} \end{cases} \tag{3.14}$$

3. Due to orthogonality of the spherical harmonics, the terms in 3.14 for which $k > K$ do not contribute to the result whilst integrating $g * \mathcal{T}$ over the unit sphere. We can therefore truncate the summation in 3.14 at $k = K$ and replace $\mathcal{T}$ by $\bar{\mathcal{T}}(\hat{\mathbf{k}}, \mathbf{R}_c, t)$ in all previous expressions. We denote $\bar{\mathcal{T}}$ by

$$\bar{\mathcal{T}}(\hat{\mathbf{k}}, \mathbf{R}_c, t) = \begin{cases} -\frac{\partial_t}{16\pi^2 R_c} \sum_{k=0}^{K} (2k+1) P_k(ct/R_c) P_k(\cos\theta') & |t| \le R_c/c \\ 0 & \text{elsewhere} \end{cases} \tag{3.15}$$

From these three observations, it is now clear that the integral in 3.8 can be written as the product of two distinct functions, both which can be expressed in terms of spherical

harmonics $Y_{km}, k = 0, \ldots, K; m = -k, \ldots, k$. The integral can then be evaluated exactly by using a $(2K+1)$-point trapezoidal rule in the ϕ-direction and a $(K+1)$-point Gauss-Legendre quadrature in the θ-direction. This yields the following expression for $\tilde{u}_l$:

$$\begin{aligned}\tilde{u}_l(\mathbf{r},t) = &\sum_{p=0}^{K}\sum_{q=-K}^{K} w_{pq}\delta\left[t-\hat{\mathbf{k}}_{pq}\cdot(\mathbf{r}-\mathbf{r}_o)\Big/c\right] * \bar{\mathcal{T}}(\hat{\mathbf{k}}_{pq},\mathbf{R}_c,t)\\ &* \int_S d\mathbf{r}'\delta\left[t+\hat{\mathbf{k}}_{pq}\cdot(\mathbf{r}'-\mathbf{r}_s)\Big/c\right] * q_l(\mathbf{r}',t)\end{aligned} \tag{3.16}$$

where

$$\begin{aligned}&w_{pq} = \frac{4\pi(1-\cos^2\theta_p)}{(2K+1)[(K+1)P_K(\cos(\theta_p)]^2}\\ &\hat{\mathbf{k}}_{pq} = \hat{\mathbf{x}}\sin\theta_p\cos\phi_q + \hat{\mathbf{y}}\sin\theta_p\sin\phi_q + \hat{\mathbf{z}}\cos\theta_p\\ &\phi_q = 2\pi q/(2K+1)\\ &\theta_p \text{ is the } (p+1)^{\text{th}} \text{ zero of } P_{K+1}(\cos\theta)\end{aligned} \tag{3.17}$$

Subsignal temporal representation

In part 1 of Spectral integration, we assumed that the subsignals were temporally bandlimited. This contradicts the requirement that each subsignal must be time limited, though this is easily fixed. The entire signal is bandlimited to $\omega_{\max}$ and can be divided into subsignals that are both bandlimited to $\omega_s = \chi_0\omega_{\max}$ where $\chi_0 > 1$ and approximately time limited. This can be done by using proper local interpolation functions.

We know $q(\mathbf{r},t)$ is temporally bandlimited. We can then locally interpolate it using temporally bandlimited and approximately time limited functions as

$$q(\mathbf{r},t) \cong \sum_{k=1}^{N_t} q(\mathbf{r},k\Delta_t)\psi_k(t). \tag{3.18}$$

Here Δ_t is the time step and $\psi_k(t)$ is a bandlimited interpolant. A near optimal ψ_k is given by

$$\psi_k(t) = \frac{\omega_+}{\omega_s}\frac{\sin(\omega_+(t-k\Delta_t))}{\omega_+(t-k\Delta_t)}\frac{\sinh\left(\omega_- p_t\Delta_t\sqrt{1-[(t-k\Delta_t)/p_t\Delta_t]^2}\right)}{\sinh(\omega_- p_t\Delta_t)\sqrt{1-[(t-k\Delta_t)/p_t\Delta_t]^2}} \tag{3.19}$$

where

$\omega_s = \pi/\Delta_t = \chi_0\omega_{\max}$
$\chi_0 > 1$ is the oversampling ratio
$\omega_\pm = (\omega_s \pm \omega_{\max})/2$
p_t integer that defines the approximate duration of the interpolation function.

If we plot these for $\Delta_t = 0.01, \chi_0 = 3, p_t = 2$ we get the following two figures

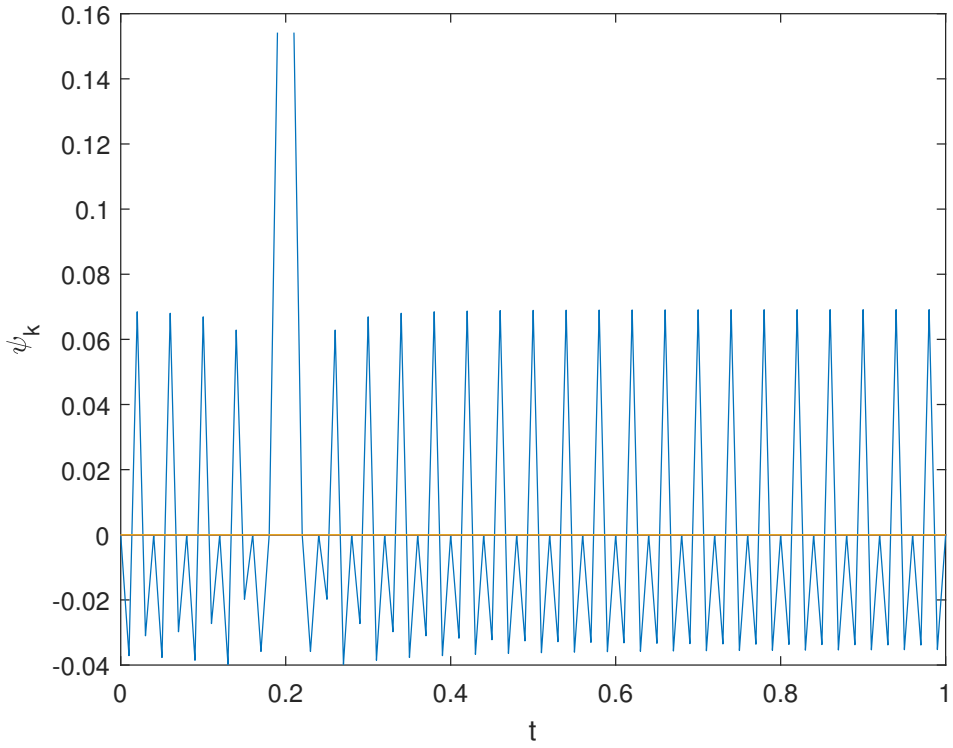

Figure 3.1: Bandlimited interpolant for $k = 20$

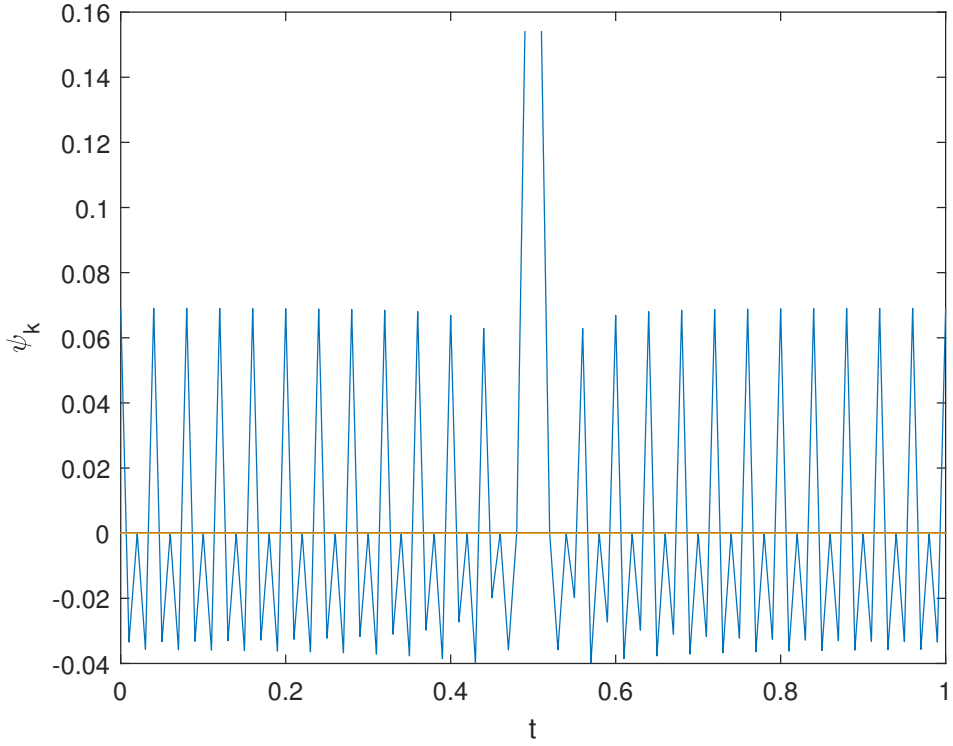

Figure 3.2: Bandlimited interpolant for $k = 50$

We see that the interpolant oscillates around the t-axis, though it creates a massive spike at k; in figure 3.2, for instance, we have 100 datapoints and $k = 50$, so naturally its spike is halfway along the axis.

From 3.18 it follows that we can divide the signal into subsignals q_l given by

$$q_l(\mathbf{r}, t) = \sum_{k=l\ M_t}^{(l+1)M_t-1} q(\mathbf{r}, k\Delta_t)\psi_k(t) \tag{3.20}$$

Here each subsignal q_l is defined in terms of M_t smaples of the signal q but spans $M_t' = M_t + 2p_t$ time steps. This results in the required time limitation.

We can easily see that the accuracy of this three-step PWTD algorithm is determined by the choices of χ_0 and χ_1. [1] has run various tests to verify this, the results of these tests can be found at the end of section 18.3.

3.3 Implementation

In this section we will combine the Plane-Wave Time-Domain algorithm with the Marching-On-In-Time method to make it more cost efficient. [1] calls it a PWTD-enhanced MOT scheme. There are two different algorithms presented, a two-level algorithm and a multi-level algorithm. Both will be discussed in this section.

3.3.1 Two-Level Algorithm

The most computationally expensive part of the MOT scheme is the computation of 2.1, especially the sum of the right-hand side. The idea behind the two-level PWTD-enhanced MOT algorithm is quite simple: we first divide the scatterer into subscatterers. We then determine the contributions of all the subscatterers that are nearby directly, the other contributions are determined using the PWTD scheme.

To formalize this, consider a cubical volume around the scatterer S. We divide this cube into smaller cubes that are equally sized and each fit into a sphere of radius R_s. Set N_g as the total number of nonempty boxes and N_s as the number of spatial basis functions for the MOT scheme. We call the collection of spatial basis functions in a nonempty box a group. The average number of spatial basis functions per group is denoted by $M_s = N_s/N_g$, where $M_s \propto (R_s\omega_{\max}/c)^2$.

Let $\gamma, \gamma' = 1, \ldots, N_g$ be groups, then each group pair (γ, γ') is either near field or far field. This depends on the distance between the group centers. If this distance is less than a preset distance $R_{c,\min}$ the pair is near field, if it is larger than $R_{c,\min}$ it is far field. We assume that $R_{c,\min} = \xi R_s$ with $3 \leq \xi \leq 6$ chosen beforehand.

To avoid ghost signals in the PWTD, we must define a correct signal duration. To that end, we define the fundamental subsignal duration T_s, and consequently we define $T_{s,\gamma\gamma'}$ as the duration of a subsignal evaluated of fields related to other far field pairs (γ, γ'). Then

$$T_s = M_t \Delta_t \tag{3.21}$$
$$M_t = \min_{\gamma,\gamma'} \{\lfloor (R_{c,\gamma\gamma'} - 2R_s)/(c\Delta_t) \rfloor\}$$
$R_{c,\gamma\gamma'}$ the distance between the centers of group-pair (γ, γ')
$$T_{s,\gamma\gamma'} = M_{t,\gamma\gamma'} \Delta_t \tag{3.22}$$
$$M_{t,\gamma\gamma'} = M_t \lfloor (R_{c,\gamma\gamma'} - 2R_s)/(cT_s) \rfloor$$

To evaluate the convolution for 3.10, we define the Fourier Transform of the translation function:

$$\begin{aligned} \mathcal{F}\left\{\bar{\mathcal{T}}(\hat{\mathbf{k}}, \mathbf{R}_{c,\gamma\gamma'}, t)\right\} &= \int_{-\infty}^{\infty} dt \bar{\mathcal{T}}(\hat{\mathbf{k}}, \mathbf{R}_{c,\gamma\gamma'}, t) e^{-i\omega t} \\ &= -\frac{i\omega}{8\pi^2 c} \sum_{k=0}^{K} (2k+1)(-i)^k \mathbf{j}_k(\omega R_{c,\gamma\gamma'}/c) P_k(\cos\theta') \end{aligned} \tag{3.23}$$

Here $\mathbf{j}_k$ is the spherical Bessel function, see also 2.13. P_k is a Legendre polynomial as in 2.15. For any sphere pair (γ, γ') we can construct these functions easily by using the normalized translation function

$$\tilde{\mathcal{T}}(\theta', \Omega) = R_{c,\gamma\gamma'} \left\{\bar{\mathcal{T}}(\hat{\mathbf{k}}, \mathbf{R}_{c,\gamma\gamma'}, t)\right\} \tag{3.24}$$
$$= -\frac{i\Omega}{8\pi^2} \sum_{k=0}^{K} (2k+1)(-i)^k \mathbf{j}_k(\Omega) P_k(\cos\theta') \tag{3.25}$$

Here θ' is the ray angle and $\Omega = \omega R_{c,\gamma\gamma'}/c$ the normalized frequency. These normalized translation functions are then saved in a table for later use.

Using everything defined before, we can now finally define a two-level algorithm for the evaluation of the sum on the right-hand side of equation 2.1. The contributions to the sum of the near and far field pairs are evaluated separately.

1. *Evaluation of the near field contributions*
   We evaluate the following sum during every time step:
   $$\sum_{k=1}^{j-1} \bar{\mathbf{Z}}_k^{\gamma\gamma'} \mathbf{Q}_{j-k}^{\gamma'} \tag{3.26}$$
   Here $\bar{\mathbf{Z}}_k^{\gamma\gamma'}$ denotes a submatrix of $\bar{\mathbf{Z}}_k$ from 2.1 that relates fields over group γ to sources in group γ'. $\mathbf{Q}_{j-k}^{\gamma'}$ is a vector with values defined by the $q_{n,j-k}$ for all sources n in group γ'.

2. *Evaluation of the far field contributions*
   We follow the three steps of the Plane-Wave Time-Domain algorithm.

(a) *Construction of outgoing rays*
A set of outgoing rays is constructed, generated by subsignals of duration T_s , constructed every M_t time steps. We determine the contribution from the spatial basis function $f_n(\mathbf{r})$ to the outgoing ray in the $\hat{\mathbf{k}}_{pq}$ direction by convolving the subsignal associated with f_n with

$$V_n^+(\hat{\mathbf{k}}_{pq}, t) = \int_S d\mathbf{r}' \delta \left[t + \hat{\mathbf{k}}_{pq} \cdot (\mathbf{r}' - \mathbf{r}_c) \Big/ c \right] f_n(\mathbf{r}') \tag{3.27}$$

Here $\mathbf{r}_c$ denotes the center of the group to which f_n belongs.

(b) *Translation*
The outgoing rays are translated to incoming rays from group γ to group γ'. The translation is described in the following steps:

i. An outgoing ray described by $\mathcal{O}(M_{t,\gamma\gamma'})$ time samples is constructed by concatenating the rays stored in 2a. The FFT of the translation function is also computed, in anticipation of the expected convolution.

ii. The spectrum of the applicable function is determined from the $\tilde{\mathcal{T}}$ table through local interpolation.

iii. The outgoing ray spectrum is multiplied with the translation function spectrum.

iv. We apply the inverse Fourier transform to transform the result back into the time domain. The obtained rays are superimposed onto incoming rays that impact upon the receiver group from all the other source groups.

(c) *Projection of incoming rays onto observers*
The field at the nth observer is evaluated by convolving the incoming rays with

$$\tilde{V}_n^-(\hat{\mathbf{k}}_{pq}, t) \int_S d\mathbf{r}' \delta \left[t - \hat{\mathbf{k}}_{pq} \cdot (\mathbf{r}' - \mathbf{r}_c) \Big/ c \right] \tilde{f}_n(\mathbf{r}') \tag{3.28}$$

The step is completed by performing the spherical integration.

3.3.2 Multilevel Algorithm

The multilevel algorithm is built upon the foundations of the two-level algorithm. In principle, we aggregate small subscatterers into larger entities before the translation, on multiple levels. To do so, we will introduce the following in this section:

- Systematic scheme for a multilevel partitioning
- Transformation operations
- Description of the algorithm

Multilevel partitioning

We will describe a hierarchical subdivision of the scatterer. This is achieved by recursively subdividing a cubical box that encloses the scatterer. Assume there is a box that encapsulates the scatterer. Divide this box into eight smaller boxes. A box that is divided into smaller boxes is called the parent, making each smaller box its child. This is done

recursively. The smallest/finest box is designated at level 1; this goes up all the way to level N_l. For levels $i = 1, \ldots, N_l$ we then define the following:

$N_g(i)$ the number of nonempty boxes
$M_s(i)$ the average number of sources in each group
$R_s(i)$ the radius of the sphere that encloses a level i box
$K(i)$ the number of spherical harmonics for the translation functions

As in the two-level algorithm, we construct a set of near and far field group pairs. Again, every source/observer combination belongs to only one group pair. However, not all far field pairs have to reside on the same level. The greater the distance between source and observer, the higher the level they are in. Next, define $R_{c,\min}(i) = \xi R_s(i)$ as the cutoff separation for each level. To construct the near and far field pairs, we first classify group pairs at the highest level whose centers are separated by more than $R_{c,\min}(i)(N_l)$ as a 'level N_l far field pair'. Next, all the level $N_l - 1$ pairs, with centers separated by more than $R_{c,\min}(i)(N_l - 1)$ that describe interactions not accounted for by any of the N_l-level far field pairs, are called 'level $N_l - 1$ far field pairs'. We continue this process recursively. At level 1 the group centers that are separated by less than $R_{c,\min}(i)(1)$ are classified as near field pairs.

As before, the fundamental subsignal duration for level i can be set as

$$
\begin{aligned}
&T_s(i) = M_t(i)\Delta_t \\
&M_t(i) = \min_{\gamma,\gamma'} \{\lfloor (\mathbf{R}_{v,\gamma\gamma'} - 2R_s(i))/(c\Delta_t) \rfloor\} \\
&T_{s,\gamma\gamma'}(i) = M_{t,\gamma\gamma'}\Delta_t \\
&M_{t,\gamma\gamma'}(i) = M_t(i)\lfloor (\mathbf{R}_{v,\gamma\gamma'} - 2R_s(i))/(cT_s(i)) \rfloor
\end{aligned}
$$

Transformation operations

To make this algorithm more efficient, for a level i we can reuse information stored in level $i - 1$ rays. This is done by implementing two operations, interpolation and splicing. Its two complementary operations at the observer side are resection and anterpolation.

Interpolation and anterpolation manipulate the outgoing respectively incoming rays. Interpolation expresses the rays in terms of spherical harmonics, increasing the sampling rate and zero-padding the excess spherical spectrum. Anterpolation truncates the spherical harmonics and lowers the sampling rate over the sphere.

An outgoing ray can be spliced into two rays and interpolated; resection is its complementary operation. Figure 18.11 of [1] illustrates this operation well.

The algorithm

We define the multilevel evaluation of 2.1 as follows:

1. *Evaluation of the near field contributions.*
   As in the two-level scheme, all the near-field interactions are dealt with classically. They all reside on the finest level.

2. *Evaluation of the far field contributions*
   These contributions will be evaluated in a three-stage process, which is comparable to the two-level scheme. Since independent generation of the outgoing and incoming rays is too costly, we shall apply interpolation, splicing, recentioning and anterpolation for a more efficient construction.

   (a) *Construction of outgoing rays*
   At level 1, we convolve the source signatures of $V_n^+(\hat{\mathbf{k}}_{pq}, t)$ from 3.27. Higher level rays are constructed from previous levels through interpolation and splicing. Levels must be traversed starting from level 1.

   (b) *Translation*
   As in the two-level scheme, the rays are translated between the far field pairs (γ, γ'). Note that a different $\tilde{TT}(\theta', \Omega)$ table has to be constructed for each level since the number of harmonics $K(i)$ is also level dependent.

   (c) *Projection of incoming rays onto observers.*
   Starting at level $N_l - 1$, the incoming rays are resected and interpolated at the higher level. The fields at the observer are constructed by convolving incoming rays at level 1 via $\tilde{V}_n^-(\hat{\mathbf{k}}_{pq}, t)$ from 3.28.

3.4 Some notes

In 3.25 we recognize the Fast Multipole Method here from 2.9. It is easy to see why: FMM and the Fast Fourier Transform (see also chapter 5) are related per [3]. Both permit a sparse matrix for the matrix Z of 2.1. Since by 3.23 we apply a Fourier Transform to find the translation function 3.25, the relation with FMM is understandable. This also peaks our interest in the Fourier Transform and speeding it up: a large part of the algorithm is applying Fourier transforms to translation functions. Optimizing that part of the algorithm accelerates the algorithm itself considerably. The same can be said for the multilevel algorithm.

Chapter 4

Fourier Transforms

4.1 Introduction

This chapter focuses on the Fourier transform, especially on the Discrete Fourier Transform and the Fast Fourier Transform, the last of which significantly improves the speed of Fourier transforms. The reason we are interested in this is because we can use Fourier transforms to determine convolutions, which the Plane-Wave Time-Domain algorithm has plenty of.

We will first look at the Fourier transform itself and the discrete form. After that, we will have a look at various algorithms that implement a fast Fourier transform. The FFTW, or Fast Fourier Transform of the West, incorporates most of these algorithms. We will look at FFTW as it provides the necessary toolset to speed up our DFTs, such as parallel computation and memory management. Finally, we will also discuss how to use FFT to evaluate a convolution.

4.2 Basic Principles

4.2.1 Fourier Transform

For this thesis, the discrete Fourier transform is of the most interest. As a reminder, we add the analytic Fourier transform and its inverse:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} dx \tag{4.1}$$

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\xi) e^{2\pi i x \xi} d\xi \tag{4.2}$$

4.2.2 Discrete Fourier Transform

For the Discrete Fourier Transform (DFT), assume that we have a finite sequence of samples of a function. The DFT then converts this sequence into a sequence of same length discrete-time Fourier Transforms. If $\{\mathbf{x_n}\} := x_0, x_1, \ldots, x_{N-1}$ is the sequence we wish to convert and $\{\mathbf{X_k}\} := X_0, X_1, \ldots, X_{N-1}$ the sequence we convert to, the DFT can be denoted as

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \tag{4.3}$$

$$= \sum_{n=0}^{N-1} x_n \left[\cos(2\pi kn/N) - i\sin(2\pi kn/N)\right] \tag{4.4}$$

This transformation is denoted by the symbol $\mathcal{F}$ such that $\mathbf{X} = \mathcal{F}(\mathbf{x})$. Furthermore, we can set $w = e^{2\pi i/N}$, which corresponds to the first complex N-th root of 1. Evaluating the DFT directly requires $\mathcal{O}(n^2)$ operations.

Furthermore, we can use the DFT to approximate the Fourier transform of functions on an interval. Assume we have a function $f(t)$ and we wish to estimate the Fourier transform over the interval $[t_0, t_1]$. We split the interval into N grids, where we then denote Δt as the distance between two gridpoints and T the total distance of the interval. We can then write $t_1 = t_0 + (N-1)\Delta t$. We can then approximate the integral with a sum. This is denoted as

$$\int_{t_0}^{t_1} f(t)e^{-it\omega}dt = \Delta t \sum_{j=0}^{N-1} f(t_j)e^{-it_j\omega}$$

$$= \Delta t\ e^{-it_0\omega} \sum_{j=0}^{N-1} f_j e^{-ij\Delta t\omega} = (*)$$

If we then set $\omega_k = k\frac{2\pi}{N\Delta t} = k\frac{2\pi}{T}$, then

$$(*) = \Delta t\ e^{-it_0\omega_k} \sum_{j=0}^{N-1} f_j e^{-ijk\frac{2\pi}{N}}$$

Here we see that

$$\sum_{j=0}^{N-1} f_j e^{-ijk\frac{2\pi}{N}} = \text{DFT}\left[f_j|_{i=0}^{N-1}\right]_k$$

4.2.3 Fast Fourier Transform

To compute the DFT, a fast Fourier transform (FFT) is almost always used. FFTs can rapidly compute DFTs by factorizing the DFT into a product of sparse matrices or factors. This can reduce the computational cost form $\mathcal{O}(n^2)$ to $\mathcal{O}(n\log n)$. There are various FFT algorithms, some of which will be discussed in the next section.

4.3 FFT Algorithms

In this section we will look at four different FFT algorithms. These four are the ones incorporated by FFTW, which is why we will be discussing them. They also show some insight in how implementing these algorithms speeds up the DFT.

4.3.1 Cooley-Tukey Algorithm

The Cooley-Tukey algorithm [11] is one of the most well-known and widely-used FFT algorithms. Simply said, it rewrites the DFT into terms of smaller DFTs recursively, and solves these. This reduces the computation time to $\mathcal{O}(N \log N)$. Furthermore, these smaller DFTs can be combined with other algorithms to solve them.

In general, Cooley-Tukey algorithms follow the same re-expression of the DFT, which has a composite size $N = N_1 N_2$:

1. Perform N_1 DFTs of size N_2
2. Multiply by complex roots of unity
3. Perform N_2 DFTs of size N_1

In part 2, these complex roots are also called twiddle factors. Normally, either N_1 or N_2 is a small factor called the radix. If N_1 is the radix, the algorithm is called a 'decimation in time' (DIT) algorithm; if N_2 is the radix it is called a 'decimation in frequency' (DIF) algorithm. As an example, we will look at the radix-2 DIT algorithm.

We define the DFT as in 4.3. The radix-2 DIT first computes the even-indexed DFTs ($x_{2m} = x_0, x_2, \ldots, x_{N-2}$) and of the odd-indexed DFTs ($x_{2m+1} = x_1, x_3, \ldots, x_{N-1}$). These two results are combined to determine the DFT of the entire sequence. This can then be performed recursively. This is the same as writing

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k} \tag{4.5}$$

We can then factor out a common multiplier $e^{-\frac{2\pi i}{N}k}$ out of the odd-sum. The two sums are the DFT of the even-indexed part x_{2m} (which we will denote E_k) and the DFT of the odd-indexed part x_{2m+1} (which we will denote O_k). We then obtain

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2}mk} + e^{-\frac{2\pi i}{N}k} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2}mk} \tag{4.6}$$

$$= E_k + e^{-\frac{2\pi i}{N}k} O_k \tag{4.7}$$

Due to the periodicity of the complex exponential, we also obtain $X_{k+\frac{N}{2}}$; its derivation can be found in [11]

$$X_{k+\frac{N}{2}} = E_k - e^{-\frac{2\pi i}{N}k} O_k \tag{4.8}$$

The algorithm gains its speed by re-using the results of these intermediate computations.

4.3.2 Prime-Factor Algorithm

The prime-factor algorithm [12] re-expresses the DFT of size $N = N_1 N_2$ as a two-dimensional $N_1 \times N_2$ DFT, **if** N_1 and N_2 are relatively prime (e.g. the only positive integer that divides both numbers is 1). The smaller DFTs of size N_1 and N_2 are evaluated either recursively via the prime-factor algorithm or through another algorithm.

We will now look at the algorithm. Recall the definition of DFT as in 4.3. First the input and output arrays are re-indexed and then substituted into the DFT formula, resulting in a two-dimensional DFT. We then define the following re-indexing of the input n and output k into n_1, n_2, k_1 and k_2, all four running from $0, \ldots, N-1$:

$$\begin{aligned} n &= n_1 N_2 + n_2 N_1 \mod N \\ k &= k_1 N_2^{-1} N_2 + k_2 N_1^{-1} N_1 \mod N \end{aligned}$$

Here N_1^{-1} indicates the modular multiplicative inverse of $N_1 \mod N_2$ (such that $N_1 N_1^{-1} \equiv 1 \mod N_2$), vice versa for N_2^{-1}.

The re-indexing is then substituted into 4.3. Any $N_1 N_2 = N$ cross terms in the nk product can be set to zero; in the same way X_k and x_n are implicitly periodic in N, so their subscripts are evaluated $\mod N$. We can then define the remaining terms as:

$$X_{k_1 N_2^{-1} N_2 + k_2 N_1^{-1} N_1} = \sum_{n_1=0}^{N_1-1} \left(\sum_{n_2=0}^{N_2-1} x_{n_1 N_2 + n_2 N_1} e^{-\frac{2\pi i}{N_2} n_2 k_2} \right) e^{-\frac{2\pi i}{N_1} n_1 k_1} \tag{4.9}$$

Both the inner and outer sum are DFTs of size N_2 and N_1, respectively, which can then be evaluated separately.

4.3.3 Rader's Algorithm

Rader's algorithm [13] works by re-expressing the DFT as a cyclic convolution, assuming the size of the DFT is a prime number. We again start with the definition of a DFT by 4.3. Assuming N is a prime number, the set of indices $n = 1, \ldots, N-1$ forms a group under multiplication modulo N. From number theory, there exists an integer g, called the generator, such that $n = g^q \mod N$ for all n and a unique q in $0, \ldots, N-2$. Similarly, $k = g^{-p}$ for index k and unique p in $0, \ldots, N-2$. Here g^{-p} is the modular multiplicative inverse as before. We can then rewrite the DFT sing the new indexes:

$$\begin{aligned} X_0 &= \sum_{n=0}^{N-1} x_n \\ X_{g^{-p}} &= x_0 + \sum_{q=0}^{N-2} x_{g^q} e^{-\frac{2\pi i}{N} g^{-(p-q)}} \qquad p = 0, \ldots, N-2 \end{aligned}$$

The summation is exactly a cyclic convolution of the following two sequences of length $N-1$, $q = 0, \ldots, N-2$:

$$\begin{aligned} a_q &= x_{g^q} \\ b_q &= e^{-\frac{2\pi i}{N} g^{-q}} \end{aligned}$$

This convolution can then be directly performed via the convolution theorem or via FFT algorithms, which will be discussed in a later section.

4.3.4 Split-Radix Algorithm

The split-radix algorithm [14] is a variant of the Cooley-Tukey algorithm that uses a combination of the radix-2 and radix-4 algorithms. It recursively re-expresses a DFT of length N into three DFTs, one of length $N/2$ and two of length $N/4$. The algorithm therefore only works if N is a multiple of 4. Since it breaks the DFT into smaller DFTs, it can be combined with other algorithms.

We again start with the DFT definition 4.3. Then we first sum over the even indices x_{2n_2}, then we sum over the odd indices via two pieces: x_{4n_4+1} and x_{4n_4+3}, depending on the index being either 1 mod 4 or 3 mod 4; n_m denotes an index that runs over $0, \ldots, N/m - 1$. Using $\omega_N = \exp(-2\pi i/N)$, the summation then looks like

$$X_k = \sum_{n_2=0}^{N/2-1} x_{2n_2}\omega_{N/2}^{n_2 k} + \omega_N^k \sum_{n_4=0}^{N/4-1} x_{4n_4+1}\omega_{N/4}^{n_4 k} + \omega_N^{3k} \sum_{n_4=0}^{N/4-1} x_{4n_4+3}\omega_{N/4}^{n_4 k} \tag{4.10}$$

The smaller DFTs can then be performed recursively and combined. Using the twiddle factors and denoting U_k as the sum of size $N/2$ and Z_k and Z'_k as the first and second sums of size $N/4$, respectively, we can write

$$\begin{aligned}
X_k &= U_k + \left(\omega_N^k Z_k + \omega_N^{3k} Z'_k\right) \\
X_{k+N/2} &= U_k - \left(\omega_N^k Z_k + \omega_N^{3k} Z'_k\right) \\
X_{k+N/4} &= U_{k+N/4} - i\left(\omega_N^k Z_k - \omega_N^{3k} Z'_k\right) \\
X_{U_k+3N/4} &= U_{k+N/4} + i\left(\omega_N^k Z_k - \omega_N^{3k} Z'_k\right)
\end{aligned}$$

This gives us all the outputs for X_k by letting k run over $0, \ldots, N/4 - 1$, thus requiring less computations.

4.4 Fast Fourier Transform Of The West

The Fast Fourier Transform Of The West, or FFTW, is a software library that can be used to compute a DFT efficiently. It was developed by Matteo Frigo and Steven G. Johnson at MIT [15]. It is a free library and is known to be the fastest implementation in existence.

FFTW's strength is that it accepts any input, independent of length, rank or it being real or imaginary. FFTW incorporates a planner that chooses the best algorithm performance-wise. The most-used algorithms are the Cooley-Tukey algorithm, the prime-factor algorithm, Rader's algorithm and the split-radix algorithm, which were discussed in the previous section. FFTW can also produce code for arbitrary array sizes, though the code generator can produce algorithms that the developers do not completely understand (see also the introduction of [16]).

Since it is free and reliably fast, many programming languages support the library. The programming language used for this thesis, Julia, also supports it. FFTW can be added by installing the packages `AbstractFFTs` and `FFTW`. The first contains the supporting libraries for FFTW, the second contains all the operations. (Julia used to have the libraries for FFT built in, but it was decided to separate these into a distinct package.) More information on the two packages can be found here [17] and here [18].

4.5 Convolution

We can use FFT to compute convolution efficiently. To do so, we need the Convolution theorem, which we will state below.

Theorem 1 (Convolution Theorem)**.** *Let f and g be two functions with convolution $f * g$. Let $\mathcal{F}$ denote the Fourier transform, then $\mathcal{F}(f)$ and $\mathcal{F}(g)$ are the Fourier transforms of f respectively g. Denoting $\cdot$ as point-wise multiplication, we then have*

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g) \tag{4.11}$$
$$\mathcal{F}(f \cdot g) = \mathcal{F}(f) * \mathcal{F}(g) \tag{4.12}$$

This also works for the Fourier inverse:

$$f * g = \mathcal{F}^{-1}\left(\mathcal{F}(f) \cdot \mathcal{F}(g)\right) \tag{4.13}$$
$$f \cdot g = \mathcal{F}^{-1}\left(\mathcal{F}(f) * \mathcal{F}(g)\right) \tag{4.14}$$

The proof can be found here [19]. A standard convolution algorithm has a complexity of $\mathcal{O}(n^2)$; the fast Fourier transform reduces this to $\mathcal{O}(n \log n)$. The equations above are also applicable for the DFT. We see that using FFT to evaluate the convolutions in the PWTD can reduce the costs considerably.

4.6 Optimization

Besides being able to compute a DFT efficiently, the FFTW library also lets users optimize a lot of the processes to fit the user's needs. Since we will be using a GPU to compute the most computationally-heavy parts of the Plane-Wave Time-Domain algorithm, namely the discrete Fourier transforms, it is essential that we optimize the entire process to minimize the costs. In this section we will look at various ways that we can do this; implementation of these optimizations will follow in the thesis itself.

4.6.1 Planner

As mentioned before, FFTW uses a planner to determine which algorithm is best suited for the problem at hand. This can be very helpful when a user is not sure which algorithm to use or when speed is not of the utmost importance, but it does take some computational cost to use this planner. Since many of the convolutions, and their corresponding Fourier transforms, are of the same size and nature, it can be beneficial to re-appropriate the same planner for all the convolutions. The planner itself can also be optimized, by using beforehand knowledge of the input arrays or by benchmarking the system with various plans. More information on plans can be found in section 4.2 of [16].

4.6.2 Memory Allocation

Another operation that comes at a great computational cost is the transferal of data from the CPU to the GPU, and back. There are two ways that we can reduce these costs, namely memory allocation and parallel computation, which will be discussed in the next section.

Memory allocation is the process where a program or routine is assigned a set of physical or virtual memory. Programs can even go a step further: memory is divided into blocks, which can individually be assigned to certain routines inside the program. Allocating memory in this way can speed up processes significantly, especially since Julia can be quite finicky when it comes to memory usage.

4.6.3 Parallel Computation

As mentioned before, another way to speed up the PWTD is to use parallel computations. The great thing about GPUs is that they have many smaller processors that can run in parallel. Many of the convolutions present in the PWTD can run independently and therefore parallel. It is therefore essential that we run the convolutions in batches on the GPU, by transferring the convolutions from the CPU to the GPU, computing them in parallel as much as possible, and then transferring the evaluations back to the CPU. Further optimization in this area includes benchmarking the size of batches and determining which processes can also be run on the GPU to lower total transferal costs.

Note that many CPUs nowadays have multiple cores and can run multiple threads at the same time. This is call multithreaded computation. Part of the research of this thesis will also focus on incorporating multithreading into the PWTD algorithm.

Chapter 5

Programming

To analyze the problem sketched in Chapter 1, we need to discretize the system before we can apply the Plane-Wave Time-Domain algorithm to it. We will program the entire system in Julia, a relatively new programming language that has some elements of C, MatLab and Python. The great thing about Julia is that we can offload part or all of the code to the GPU, which can then do most of the heavy lifting thus reducing the computational cost and computation time. GPU programming can be done via CUDA, which only works on Nvidia GPU's, or OpenCL, which works on nearly all graphics processors.

In this chapter, we will have a look at Julia, including some basic information, what sets it apart and what makes it so powerful. Furthermore, we will look at GPU programming in general as well as at CUDA and OpenCL programming.

5.1 Julia

5.1.1 Introduction

Julia was developed by Jeff Bezanson, Stefan Karpinski, Viral B. Shah and Alan Edelman [20]. They wanted to build a programming language that combined all the best parts of MatLab, Python, Ruby, Perl, R and C, whilst also being extremely fast. It was to be a programming language that was easy to use, but could still be extremely powerful when needed and could excel at everything that was thrown at it.

They have more or less succeeded. Though still seen as up and coming, Julia is picking up more and more users, especially those performing numerical mathematical computations. It has currently been downloaded over 4 million times.

What makes Julia so powerful are some unique features that make it stand out from other programming languages. Some of these will be discussed below, including its unique syntax, macros, types and packages; other features are the ability to call Python and C functions and built-in parallel and distributed computing.

5.1.2 History And Versions

Julia's first version was released on 14 February 2012, dubbed version 0.0. Exactly a year later version 0.1 was released to the general public. A new version was released every year, with 0.2 and 0.3 in 2014, 0.4 in 2015, 0.5 in 2016 and 0.6 in 2017. The first huge

upgrade was released in August 2018 together with a version 0.7. This latter version was released to help test packages for the big 1.0 release and to guide users into any syntax and underlying code changes to the language.

Version 1.0 has been designated a long-term release, with support for at least a year. Version 1.1 was released in January 2019, with a release of version 1.2 imminent. New updates for 1.0 are released monthly to iron out any bugs that may have popped up.

5.1.3 Syntax

Julia has its own unique syntax, which has elements of MatLab and Python in it but also its own unique approach to mathematical programming. It follows Python in that it uses tabbed-indents for for-loops and such instead of curly brackets that open and close an operation. On the other hand, when using an index it will start at 1 like MatLab, instead of 0 such as Python does. Many functions like `zeroes` that create an array of matrix filled with zeroes are equivalent to those found in MatLab.

It does add its own unique twist by adding options to include symbols such as $\in, \subset$ and Greek letters; these are defined in the same way as in LaTeX. These work in various ways:

- As Booleans; for instance, running `A = [1,2,3,4,5], 2∈A` will return `True`.
- As an index; we can run through a for-loop by using the set A above and setting `for i∈A, println(i), end` which will return 1, 2, 3, 4, 5.
- Setoperations; setting `A=Set([1,2,3])` and `B=Set([2,4,6])`, the operation `intersect(A,B)` returns `Set([2])`.

Macros

Macros are a set of code that can be generated before the general code is run. In this way an expression is compiled from the macro directly instead of requiring a runtime call. As an example, we shall give a Hello World-macro.

```
macro sayhello()
    return :( println("Hello World"))
end
```

The Julia REPL will then return `@sayhello (macro with 1 method)`. The compiler will then replace any instance of `@sayhello` by the code above. This can be made as complicated as the user wishes by for instance including variables.

Types

One of the hallmarks of Julia is its approach to types. Though we will not discuss them at length in this report, know that they are highly dynamic. This ranges from values being of any type when typesetting is omitted, to setting detailed types and type unions to values. Types can also be parametrized by other types. An extensive list of all the type-operations available can be found in the corresponding Julia documentation [21].

5.1.4 Packages

Julia has a built-in package manager that manages installed packages in its own unique way. It is designed around environments, which are sets of packages that can be used by an individual project or can be used globally on a system. Environments can be updated independently from each other, which can prevent updates from breaking other packages.

Denoted by `Pkg` and called by pressing the]-key in the Julia REPL, it can install packages from the general Julia repository or from specific Github repositories by using the `add 'Package'`-command. It can also handle updates of packages for the user; the command `status` returns a list of all installed packages and their versions, the command `update` updates the packages.

There are a few ways to create a package.

- Create a GitHub (or GitLab) repository and install it as a package. Adding files can then be done via Git.
- Use the package PkgTemplates [22] to create a package.
- Use the command `pkg> generate` to create a new package.

For more information on packages, see the Julia documentation [23].

5.2 GPU Programming

5.2.1 Introduction

The GPU, or Graphics Processing Unit, is an integral part of any personal computer, be it a laptop, phone or gameconsole. At its core it is a specialized unit for creating and storing images and outputting them to a display. The GPU relies on parallel programming, which makes GPUs excellent candidates for algorithms that require parallel processing of large amounts of data.

Though traditionally used for video games, in recent years GPUs have been used for various other processes. These include video decoding and deep learning, where especially the latter has gained popularity. AI development has thrived due to the general availability and affordability of powerful graphics cards.

Scientific Computing has also gained much from the implementation of graphics cards. The power of GPUs alone can speed up large-scale problems; parallel programming makes this even more powerful. In this section, we will look at GPU programming by first taking a look at how GPUs work and how they differ from CPUs. After that, we will look at GPU programming languages, including the most popular languages CUDA and OpenCL.

5.2.2 The Workings Of A GPU

A GPU differs from a CPU in various different ways. The most obvious difference is in the amount of cores each processing unit has. A CPU mostly has two to four, even up to 24 cores. A GPU, on the other hand, can have thousands. The downside of the GPU is that, compared to a CPU, it can perform a fraction of the operations a CPU can perform. However, it can do them very, very fast by utilizing all those cores in parallel. Also of

some importance is that CPUs run at higher clock speeds and are able to manage input and output of the system, something which the GPU cannot.

Most operating systems are also dependent on CPUs to run and GPU usage is only effective for larger-scale problems. Therefore most programs and algorithms start with CPU operations; parts of the code are then programmed directly onto the GPU before the results are sent back to the program. A systematic approach can be found in figure 5.1. We see that via the IO data is sent to the CPU, from where data is sent to the GPU, where we apply parallel programming. The data is then sent back to the central computing system, which can in turn also send data between the two computing units. Finally, data is sent back to the IO before the program is stopped.

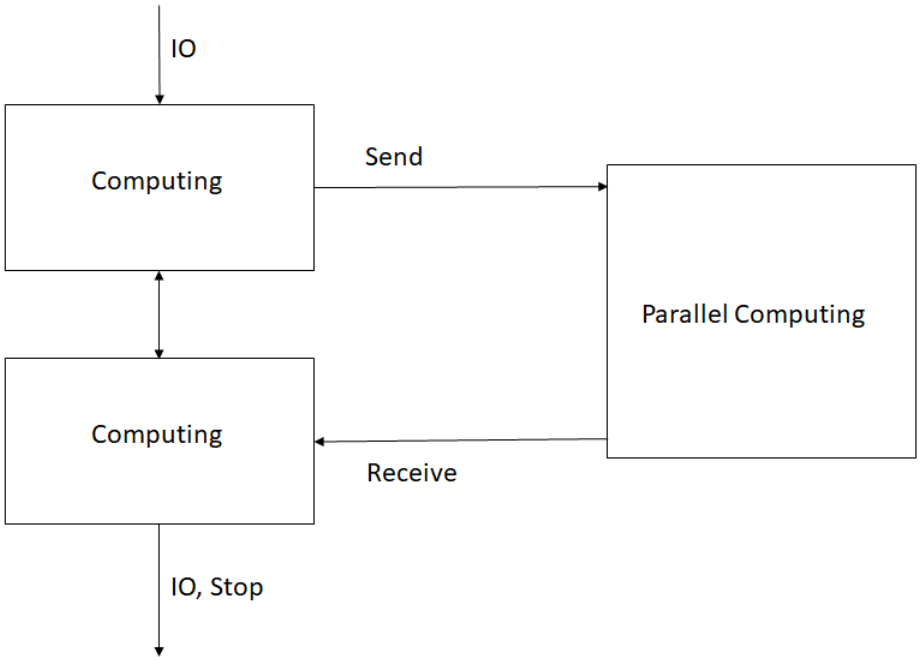

Figure 5.1: Operations in a regular CPU-GPU system

This data transfer between the CPU and GPU is the entire crux of GPU programming: though programming simple operations in batches on a GPU might reduce computational costs significantly, data transfer from the CPU to GPU and back is extremely slow. All the data has to be copied from the CPU RAM to the GPU VRAM before the operations can start, then afterwards the new data has to be copied back before it can be used by the CPU.

5.2.3 GPU Architecture

In this section we will go into somewhat more detail of the architecture of a GPU. As discussed before, a GPU has a multitude of processors that all execute the same set of instructions in parallel, without any dependence. The GPU has a fixed number of multiprocessors, each which contains a further eight scalar processors. The following image from [27] shows how this works in practice.

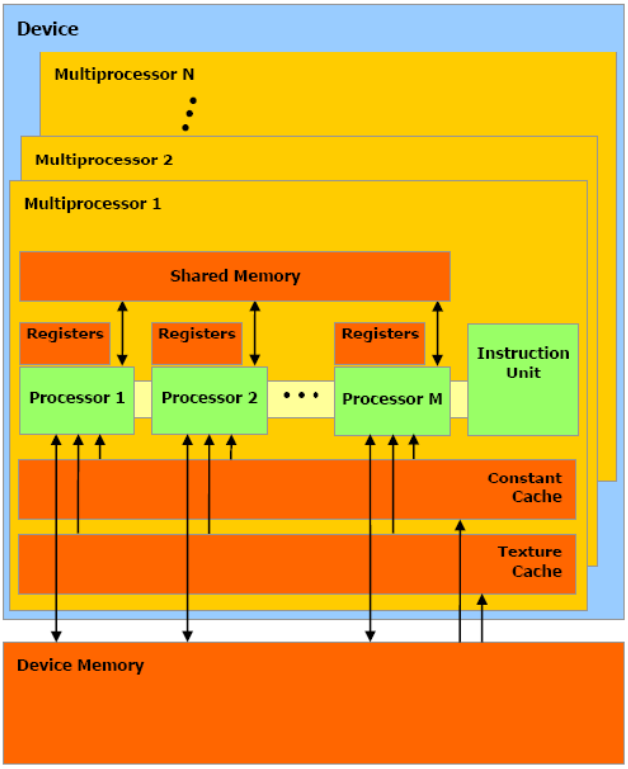

Figure 5.2: Architecture of a GPU, per [27]

As a sidenote, we will look at the different types of memory in the device:

- **Global Memory** Memory of the device itself. Largest in size of all four memories, though also has the largest access time, about 200 times larger.
- **Register Memory** Each processing unit has its own register memory. Each thread that is run can only use one register.
- **Shared Memory** Memory that can be accessed by all threads in a multiprocessor block.
- **Texture Memory** Read-only memory on a multiprocessor.

Efficient utilization of the register and shared memory should cut down on computational and latency costs. Exploration in this area together with FFTW should speed up the PWTD algorithm.

[27] chapter 3 goes into greater detail in how to use the architecture of the processors and memory effectively. It includes various methods and techniques that have been optimized for GPU computations. The rest of the reader is also quite helpful. Chapter 2 describes various ways to use parallel programming with certain methods such as matrix-vector products and LU decomposition.

5.2.4 CUDA

CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform that is developed by Nvidia for computing on GPUs. If developers have a GPU

with CUDA cores, the CUDA platform acts as a software layer that gives direct access to the GPU's instruction set and parallel processing cores. It is designed to work with C, C++ and Fortran, eliminating the need for extensive GPU programming skills. Julia uses a third-party wrapper for CUDA access.

CUDA has a few advantages over regular GPU programming:

- Unified and shared memory; this encapsulates a fast shared memory region that can be shared among threads.
- Faster downloads and readings to and from a GPU
- Robust documentation
- Regular updates

The last two are especially important. Nvidia regularly updates CUDA with new features, the current version 10.1.105 is from 27 February 2019. Because of these regular updates and support from Nvidia, developers are more likely to support and use CUDA. The platform has become synonymous with development in machine learning and AI development because of this. The fact that even Nvidia's mainstream graphics cards [24] support and use CUDA cores means that the bar is low for developers to start using CUDA in their programs.

Since Julia is a high-level scripting language, it allows us to write both the GPU kernel (program that runs on the GPU) and the code surrounding it in Julia, whilst it runs on the GPU. This can be done by implementing the correct packages, including CUDAnative.jl, CuArrays.jl and GPUArrays.jl.

5.2.5 OpenCL

OpenCL stands for Open Computing Language, and it is a framework for writing programs that can run on any platform consisting of CPUs, GPUs and other types of processers and hardware accelerators, as OpenCL views the system as a set of computing devices. It is an open standard that is maintained by the non-profit Khronos Group. [26]

OpenCL has its own C-like programming language called OpenCL C, which can make it cumbersome to progam in. Third-party APIs do exist for most programming languages. OpenCL C includes ways to implement parallelism with vectors, operations and synchronization.

One of the advantages of OpenCL is that it is vendor-independent, meaning that it runs on almost every GPU, including for instance built-in Intel GPUs. This counteracts vendor lock-in. Furthermore, since it is maintained by a non-profit organization, the premise of the platform is to empower developers [25] instead of selling as much hardware as possible. The software is royalty-free and an open standard, meaning that anyone can use the platform for free. This does mean that there is less documentation available in comparison to CUDA, and due to the difficulty of implementing OpenCL due to its programming language, it is a lot less popular than CUDA.

5.2.6 Current State Of Affairs

Development of a programming language, such as Julia, is an ongoing affair, which means that it is continually upgraded to newer versions. As mentioned befor, as of August 2018, Julia has reached version 1.0 and it has reached version 1.1 in January of this year. Because of this continuing process, packages must also be updated to work with newer versions, since dependencies or syntax may change during upgrades.

Since CUDA development is a lot more popular than OpenCL, work on OpenCL for Julia is a lot less fast-paced than that of CUDA, especially for a programming language that hasn't reached the deployment level of Python or JavaScript. As of writing, Julia version $0.4.x$ is supported in the release version of OpenCL.jl, with support going up to officially version $0.6.x$ in the master branch, with version 0.7 and therefore version 1.0 working experimentally. Some testcode has been found to work with the master branch of OpenCL.jl, though this does not mean that other packages depending on OpenCL.jl, such as CLArrays.jl or CLFFT.jl, will work flawlessly with newer versions of Julia. It is therefore advisable to either wait for updated versions of these packages, especially OpenCL.jl, or to use an older version of Julia until updates have gone through. As an example, CLFFT.jl does not install on systems with versions higher than Julia 0.5.2.

CUDA development is a whole other story. The most widely-used package, CUDAnative.jl, currently supports version 1.0 and is even a requirement to use it. Support for drivers and API's is also up to date, and packages such as CuArrays.jl also require Julia 1.0. We can clearly see that CUDA development is a lot more thriving in Julia and more beneficial for development, as future releases of Julia have a lower chance of unsupported CUDA packages.

For the interested reader, there is also an entire page on GitHub [28] dedicated to GPU programming in Julia, with links to various packages for both CUDA and OpenCL.

Chapter 6

Research

The previous chapters have discussed the building blocks for this thesis, now we need to put them to good use. We shall formulate the main research question and sub-questions for the thesis in the first section. In the second section, we shall explore how to answer these questions and which procedures we will follow. In the last section we will have a quick look at Capgemini, where the author is working on this thesis.

As a reminder, a short version of the thesis. Scatter-like physical phenomena, like the high-frequency communication and fotonics, are quite difficult to model, due to the high computational cost that for instance a Marching-On-in-Time algorithm would require. To reduce the computational cost and computational time, the thesis will explore applying the Plane-Wave Time-Domain algorithm to MOT and run it on GPUs; the PWTD can achieve a near-linear complexity. PWTD will be applied in both a two-level and a multilevel algorithm. Further optimization will also be done to the algorithm via FFT optimization and parallel GPU programming.

6.1 Research Questions

The main question for this master thesis is as follows:

> *By how much can we reduce the computational time of the Plane-Wave Time-Domain algorithm by using GPUs?*

The sub-questions for this question are:

- *How can we best optimize the performance of the FFT on GPUs?*
- *How can we best minimize the number of (and concurrently the computational costs and time from) transferals of data between CPU and GPU?*
- *By what other means can we optimize the PWTD algorithm?*

6.2 Exploring Answers

As discussed in the FFT chapter, Fourier transforms will play an important part in the acceleration of the PWTD algorithm. Optimizing those will be one of the main parts to answer the main thesis question. To do so, we will focus on several parts of the fast Fourier transform, some of which have been discusses before:

- **Planner optimization.** The planner determines which algorithm FFTW will use to compute the DFT. Reusing this will almost surely reduce computational costs. We will look by how much costs can be reduced and if it is beneficial to reuse the planner.
- **GPU acceleration.** The main exploration of this thesis. We will test how much we can accelerate FFT by using GPUs, and explore which factors contribute to an optimal algorithm. This includes variables such as batch sizes.
- **Memory management.** We wish to minimize computational costs by exploring memory management and memory allocation to see what kind of effects these can have on the computational complexity.

Transfer of data between the CPU and GPU will also be an important issue to explore and minimize. This transfer can be slow since data has to be transferred from the RAM to the VRAM before it can be processed by the GPU. It is often seen as a bottleneck for GPU programming. We wish to minimize these computational costs by determining how much data size and batch size can affect the costs. Furthermore, we will look into how we can compute as many operations as possible, in the most effective way, before we have to transfer data back.

Lastly, we will look at other, maybe smaller, ways that we can optimize the Plane-Wave Time-Domain algorithm. These at least include the following list, though that may expand during further research for the thesis.

- **Mathematical optimization.** The choice of basis functions can matter for the Marching-On-In-Time method. It might therefore be beneficial to see which mathematical choises can impact the overal computational costs of the algorithm.
- **More GPU operations.** Explore which other operations can be transferred to the GPU to minimize transfer costs and maximize GPU usage.

6.3 Capgemini

Work on this thesis will be done at Capgemini [29]. Capgemini is a large multinational country that originated in France in 1967. It provides professional services and consultancy to its clients, which include large entities such as the NS and the Ministry of Justice and Security. It has over $200,000$ employees worldwide in over 40 countries.

In the past, Capgemini has also dabbled in IT development and research, which they have expanded in recent years. For instance, Capgemini Leidsche Rijn (the Dutch headquarters) have opened a research floor known as the CoZone where research is done into leading technologies such as blockchain, machine learning, VR and development of own technologies.

They have also started recruiting more interns to help with these projects as well as to facilitate internships into new projects and developing technologies. This helps Capgemini to stay on top in research and development as well as creating extra recruiting options.

For their research projects, Capgemini has aqcuired a HP server with the following specifications:

- Two Intel Xeon Gold 12-core processors
- 192 GB RAM
- 4TB SSD
- Two Nvidia Tesla V100 GPUs

The two GPUs are especially of interest for this thesis. They are quite new and very powerful, and it will be interesting to see how acceleration scales over two connected GPUs.

Bibliography

[1] A. Arif Ergin, Balasubramaniam Shanker and Eric Michielssen, *The plane-wave time-domain algorithm for the fast analysis of transient wave phenomena*,
IEEE Antennas and Propagation Magazine Volume 41, Issue 4, September 1999

[2] Anne Mackenzie, Michael Baginski and Sadasiva Rao, New Basis Functions for the Electromagnetic Solution of Arbitrarily-shaped, Three Dimensional Conducting Bodies Using Method of Moments
Microwave and Optical Technology Letters 50(4): pages 1121-1124, April 2008

[3] Ronald Coifman, Vladimir Rokhlin and Stephen Wandzura, *The Fast Multipole Method For Electromagnetic Scattering Calculations*,
Proceedings of IEEE ANtennas and Propagation Society International Symposium, July 1993

[4] *Legendre Polynomials*,
`https://en.wikipedia.org/wiki/Legendre_polynomials`

[5] *Digital Library of Mathematical Functions: Bessel- and Spherical Bessel Functions*,
`https://dlmf.nist.gov/10.47`

[6] *L^2-inner product*,
`http://mathworld.wolfram.com/L2-InnerProduct.html`

[7] *Quadrature Rules*,
`https://link.springer.com/content/pdf/bbm%3A978-1-4612-0101-4%2F1.pdf`

[8] *Trapezoidal Rule*,
`https://en.wikipedia.org/wiki/Trapezoidal_rule`

[9] *Parseval's Theorem*,
`https://en.wikipedia.org/wiki/Parseval%27s_theorem`

[10] *Parseval's Identity*,
`https://en.wikipedia.org/wiki/Parseval%27s_identity`

[11] *Cooley-Tukey FFT algorithm*,
`https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm`

[12] *Prime-factor algorithm*,
`https://en.wikipedia.org/wiki/Prime-factor_FFT_algorithm`

[13] *Rader's FFT algorithm*,
`https://en.wikipedia.org/wiki/Rader%27s_FFT_algorithm`

[14] *Split-radix algorithm*,
https://en.wikipedia.org/wiki/Split-radix_FFT_algorithm

[15] Matteo Frigo and Steven G. Johnson, *The Design and Implementation of FFTW3*, Proceedings of the IEEE 93(2), pages 216-231, February 2005

[16] Matteo Frigo and Steven G. Johnson, *FFTW Documentation*
fftw.org/fftw3.pdf, version 3.3.8, 24 May 2018

[17] *AbstractFFTs package*,
https://github.com/JuliaMath/AbstractFFTs.jl

[18] *FFTW package*,
https://github.com/JuliaMath/FFTW.jl

[19] *Convolution Theorem*,
https://en.wikipedia.org/wiki/Convolution_theorem

[20] Jeff Bezanson, Stefan Karpinski, Viral B. Shah and Alan Edelman, *Why We Created Julia*
https://julialang.org/blog/2012/02/why-we-created-julia, February 2012

[21] *Julia Documentation On Types*,
https://docs.julialang.org/en/v1/manual/types/index.html

[22] *Julia PkgTemplate Repository*,
https://github.com/invenia/PkgTemplates.jl

[23] *Julia Documentation On Packages*,
https://docs.julialang.org/en/v1.0/stdlib/Pkg/

[24] *CUDA GPUs*,
https://developer.nvidia.com/cuda-gpus

[25] *Khronos Group About page*,
https://www.khronos.org/about/

[26] *OpenCL Overview*,
https://www.khronos.org/opencl/

[27] C. Vuik and C.W.J. Lemmens, *Programming on the GPU with CUDA*,
Reader for a course on CUDA programming on the GPU, January 2019

[28] Jake Bolewski, Krys Kamieniecki, Tim Besard, Hendrik Ranocha, Simon and Valentin Churavy, *JuliaGPU*
https://github.com/JuliaGPU

[29] *Capgemini Website*,
https://www.capgemini.com/