ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

PAUL SCHERRER INSTITUT
PSI

A PERFORMANCE PORTABLE CHARGE CONSERVING FDTD IMPLEMENTATION

Master Thesis

in High Energy Physics

Department of Physics

ETH Zurich

written by

Manuel Winkler

supervised by

Dr. A. Adelmann (ETH)

Scientific advisors

Sonali Mayani

May 17 2024

Abstract

We implement a Finite Difference Time Domain field solver combined with the Particle-in-Cell method to simulate the electromagnetic interaction processes between charged particles in strong and rapidly changing electromagnetic external fields. We aim to use nonstandard finite-difference schemes to update the electromagnetic fields and a Boris pusher to update the particles. The implementation is done on a performance portability layer, such that it can be used on large High Performance Computing clusters with heterogeneous hardware. Finally, we demonstrate its correctness by reproducing results of the simulation of an infrared Free Electron Laser (FEL) known from literature.

Contents

List of figures

Chapter 1

Introduction

Our electromagnetic simulations magnetic simulation consists of a solver for the general wave equation with a source term as described in Section 2.1. This solver is coupled with a particle-based solver for the source term. In order to denote and implement the equations in a physically correct way, a unit basis for their representation has to be selected. Several conventions exist; this introduction outlines the reasoning for our choice.

1.1 Dimension analysis

In order to simplify every equation in classical and quantum electrodynamics, we use aim to use a set of basis units such that the involved constants c, $\hbar$, ε_0 and μ_0 are all equal to one. All the code described and written in Section 3 and Section A is implemented using natural units, outlined in Table 1. Still, all formulae presented in this report are correct for any unit basis.

One problem that arises with the use of Planck units is that many real-world quantities do not fit into a C++ `float`, representing a 32-bit floating point number. For example, with the maximum value representable in a `float` being $\approx 3.4 \cdot 10^{38}$, it is not possible to represent even one second in Planck times, which would be larger than 10^{43}. To keep 32 bit floats an option we derive a set of scaled Planck units in Section 1.1.2.

1.1.1 Planck units

The way Planck units are usually defined [1, 2] is by setting the five fundamental constants

$$\hbar = 4\pi\varepsilon_0 = c = k_B = G = 1$$

where $\hbar$ denotes the Planck constant, ε_0 the vacuum permittivity, c the speed of light. The Boltzmann constant k_b and the gravitational constant G are irrelevant for our problem. One could also pick the vacuum permittivity ε_0 itself as 1, already leading to two ways to define the Planck units. Many more conventions exist [2].

$$\begin{cases} 4\pi\varepsilon_0 = 1 \Longleftrightarrow q_P = 11.7062371e & \text{(Gaussian normalization)} \\ \varepsilon_0 = 1 \Longleftrightarrow q_P = \frac{11.7062371}{\sqrt{4\pi}}e = 3.3022685e \end{cases}$$

The other three Planck base units are not affected by this choice. The Planck power P_P for example is unaffected since the Planck voltage and Planck current are scaled in opposite directions.

We will be using Planck units with $\varepsilon_0 = 1$.

1.1.2 Scaled Planck units

Since in our application, G is of no interest, scaling the Planck units by a factor of α, conserving

$$\hbar = \varepsilon_0 = c = 1$$

is a good way to keep the numbers appearing in the simulation closer to 1. We denote the *dimensionality* vector of a quantity S as follows:

$$\boldsymbol{D}_S = \begin{bmatrix} L_S \\ M_S \\ t_S \\ I_S \end{bmatrix}$$

where L, M, t, I denote length, mass, time and current. Such a dimensionality can also be assigned to each physical constant according to its units. A constant C with dimensionality D_C will transform from a unit system U to a scaled unit system U_α as follows:

$$C_{U_\alpha} = C_U \prod_i \alpha_i^{-D_{Ci}} \tag{1.1}$$

where α_i denotes the scale applied to the i-th dimension. As an example, consider the speed of light:

$$D_c = \begin{bmatrix} 1 \\ 0 \\ -1 \\ 0 \end{bmatrix}$$

If we stretch the length unit by a factor of 2 and the time by a factor of $\frac{1}{2}$, Equation 1.1 will evaluate to the following:

$$\begin{aligned} c_{U_\alpha} &= c_U \prod_{D_c} \alpha_i^{-D_{ci}} \quad \text{with } \alpha = \begin{bmatrix} 2 \\ 1 \\ \frac{1}{2} \\ 1 \end{bmatrix} \\ &= 2^{-1} \cdot \left(\frac{1}{2}\right)^1 = \frac{1}{4} \end{aligned}$$

Therefore, if we pick a unit system where the length unit is two Planck lengths and the time unit is half a Planck time, the speed of light represented in those units will be

$$c = \frac{1}{4} \frac{\text{double Planck lengths}}{\text{half Planck times}}$$

which is intuitively clear.

We can use Equation 1.1 to construct a scaling vector that keeps a set of m constants spanning n dimensions constant. The system of equations will be composed as follows:

$$\prod_i \alpha_i^{-D_{Ci}} = 1$$

$$\vdots \text{ for every constant } C$$

Let's consider the case

$$\alpha_i = \alpha^{x_i} \text{ where } x \in \mathbb{R}^{+^n}$$

This results in the following equation for a constant C

$$\prod_i \alpha^{-x_i D_{Ci}} = 1$$

Taking the logarithm with basis α on each side yields

$$\begin{aligned} &\sum_i -x_i D_{Ci} = 0 \\ \iff &\sum_i \quad x_i D_{Ci} = 0. \end{aligned} \tag{1.2}$$

This can be composed an $(m = 3, n = 4)$ linear system of equation in the case of fixating the three constants $\hbar$, ε_0 and c to 1:

$$\begin{bmatrix} 2 & 1 & -1 & 0 \\ -3 & -1 & 4 & 2 \\ 1 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \longrightarrow \qquad A \begin{bmatrix} -x \\ x \\ -x \\ x \end{bmatrix} = \mathbf{0}, x \in \mathbb{R}.$$

One useful solution of this system is the vector

$$e = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$$

which, given a scaling parameter α, leads to the scaling vector

$$\boldsymbol{\alpha} = \begin{bmatrix} \alpha \\ \frac{1}{\alpha} \\ \alpha \\ \frac{1}{\alpha} \end{bmatrix}$$

This vector is obtained by undoing the logarithm to obtain Equation 1.2.

Quantity	Value
Scaled Planck length	$\alpha \cdot l_P$
Scaled Planck mass	$\frac{m_P}{\alpha}$
Scaled Planck time	$\alpha \cdot t_P$
Scaled Planck force	$\frac{F_P}{\alpha^2} = \frac{1.210 \cdot 10^{44}}{\alpha^2}$
Scaled Planck current	$\frac{I_P}{\alpha} = \frac{9.813 \cdot 10^{24}}{\alpha}\mathrm{A}$
Scaled Planck charge	$q_P = 3.303e$
Scaled Planck velocity	$v_P = c = \frac{l_P}{t_P}$
Scaled Planck magnetic flux density	$\frac{\Phi_P}{\alpha^2} = \frac{7.630 \cdot 10^{53}}{\alpha^2}\mathrm{T}$
Scaled Planck electric strength	$\frac{E_P}{\alpha^2} = \frac{2.287 \cdot 10^{62}}{\alpha^2}\ \frac{\mathrm{V}}{\mathrm{m}}$
Scaled power	$\frac{P_P}{\alpha^2} = \frac{3.628 \cdot 10^{52}}{\alpha^2}\mathrm{W}$
Scaled power density	$\frac{P_P}{\alpha^2} = \frac{1.389 \cdot 10^{122}}{\alpha^4}\frac{\mathrm{W}}{\mathrm{m}^2}$

Table 1: Scaled Planck units

A good choice for the scaling factor α is 10^{30}. This yields reasonable values for elementary particle mass and simulation domain extents. A list of all those conversions in C++ can be found **here**.

Chapter 2

Governing equations

2.1 Wave equation

The general wave equation is as follows:

$$\frac{\partial^2 u}{\partial t^2} = \Delta u + f \tag{2.1}$$

where u denotes the target function and f the source term. Solutions of Equation 2.1 are of great interest, since all equations stated in Section 2.1.1 can be written in this form.

2.1.1 Maxwell's equations

The equations governing the electromagnetic fields are called **Maxwell's Equations**. We can write them an electromagnetic and a potential form. The electromagnetic form describes the evolution of the electric ane the magnetic field and is a system of first order partial differential equations and can be written as follows:

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\varepsilon_0} \tag{2.2}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{2.3}$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{2.4}$$

$$\nabla \times \mathbf{B} = \mu_0 \mathbf{J} + \frac{1}{c^2}\frac{\partial \mathbf{E}}{\partial t}. \tag{2.5}$$

Substituting Equation 2.5 with $\rho, \mathbf{J} = 0$ into Equation 2.4 and taking the curl on both sides yields

$$\nabla \times \nabla \times \mathbf{E} = -\mu_0 \varepsilon_0 \frac{\partial^2 \mathbf{E}}{\partial t^2} \tag{2.6}$$

$$\iff \Delta \mathbf{E} = -\mu_0 \varepsilon_0 \frac{\partial^2 \mathbf{E}}{\partial t^2} \tag{2.7}$$

where Equation 2.2 and Equation 2.3 were used to convert two curls into a laplacian to obtain a wave equation in the form of Equation 2.1.

An alternative equation can be written that does not depend on **E** and **B**, but on a scalar and vector potential. We define the scalar potential φ and vector potential A as follows:

$$\nabla \times \mathrm{A} = \mathbf{B}$$

$$-\nabla\varphi - \frac{\partial \mathrm{A}}{\partial t} = \mathbf{E}$$

We also define the four-source as follows:

$$\mathbf{J}^{\alpha} = \left[\frac{1}{\varepsilon_0}\rho \quad \mu_0 J_0 \quad \mu_0 J_1 \quad \mu_0 J_2\right]^T$$

and the four-potential

$$\mathbf{A}^{\alpha} = [\varphi \quad \mathrm{A}_0 \quad \mathrm{A}_1 \quad \mathrm{A}_2]^T$$

Maxwells equations can then be reduced to

$$\frac{1}{c^2}\frac{\partial^2 \mathbf{A}^{\alpha}}{\partial t^2} = \Delta \mathbf{A}^{\alpha} + \mathbf{J}^{\alpha} \tag{2.8}$$

For a full derivation we refer to [3]. In Equation 2.8 we now have one four-dimensional partial differential equation in the form of Equation 2.1.

2.1.2 Boundary Conditions

The evolution of values of $\mathbf{A}^{\alpha}$ on the domain boundary can be parametrized in several ways. The simplest equation to parametrize the boundary is to lock it to the constant value 0, known as **Dirichlet Boundary conditions**:

$$\mathbf{A}^{\alpha}(x) = 0 \ \forall x \in \partial\Omega \tag{2.9}$$

Alternatively, the outward gradient can be fixed to zero, leading to **Neumann boundary conditions:**

$$\frac{\partial \mathbf{A}^{\alpha}}{\partial \boldsymbol{n}}(x) = 0 \ \forall x \in \partial\Omega$$

For simulation taking place in open space, the boundary conditions usually of interest are absorbing boundary conditions. Absorbing boundary conditions are equivalent to an “open” domain boundary – where outgoing waves simply leave the domain. In what follows, we derive these absorbing boundary conditions.

To get an intuitive sense on absorbing boundary conditions, consider the one-dimensional wave equation

$$\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = 0 \tag{2.10}$$

defined on an interval $[0, L]$. This equation can be factorized into two factors:

$$\left(c\frac{\partial}{\partial x} + \frac{\partial}{\partial t}\right)\left(c\frac{\partial}{\partial x} - \frac{\partial}{\partial t}\right)u = 0 \tag{2.11}$$

Therefore, any u that satisfies either of the unidirectional advection equations

$$\left(c\frac{\partial}{\partial x} - \frac{\partial}{\partial t}\right)u^- = 0$$
$$\left(c\frac{\partial}{\partial x} + \frac{\partial}{\partial t}\right)u^+ = 0$$

is a solution for the wave equation, where u^- describes left-moving waves and u^+ describes right-moving waves. As long as we can represent u as a superposition

$$u = u^+ + u^-,$$

u satisfies Equation 2.10. As an example, consider the Equation 2.10 with initial conditions:

$$u(x, t = 0) = f(x)$$
$$\dot{u}(x, t = 0) = g(x)$$

By definition we have

$$\frac{\partial u^\pm}{\partial t} = \mp c \frac{\partial u^\pm}{\partial x},$$

leading to two conditions. Since

$$\frac{\partial}{\partial t}\left(u^+ - u^-\right)\Big|_{t=0} = g$$

$$\iff \begin{cases} c\dfrac{\partial}{\partial x}\left(u^+ - u^-\right)\Big|_{t=0} = g \\ \left(u^+ + u^-\right)\Big|_{t=0} = f \iff u^+ = f - u^- \end{cases}$$

Manipulating the terms and replacing u^+ we obtain the following for $t = 0$:

$$c\frac{\partial}{\partial x}(2u^- - f) = g \Longleftrightarrow c\frac{\partial}{\partial x}u^- = \frac{1}{2}\left(g + \frac{\partial}{\partial x}f\right)$$

$$\Longleftrightarrow u^-(t=0,x) = \frac{1}{2c}\left(\int_{L_{\text{lower}}}^{x} g(\chi)\mathrm{d}\chi + f(x)\right) \tag{2.12}$$

where the L_{lower} denotes the lower limit of our domain of definition. Equation 2.12 therefore describes a conversion from a state of the wave equation represented as u and its temporal derivative $\dot{u}$ to a representation as a left and a right moving part u^- and u^+. The three most important special cases are

$$\begin{bmatrix} u \\ \dot{u} \end{bmatrix}_{t=0} = \begin{bmatrix} f \\ 0 \end{bmatrix} \quad \Longrightarrow \quad \begin{bmatrix} u^+ \\ u^- \end{bmatrix} = \begin{bmatrix} \frac{f}{2} \\ \frac{f}{2} \end{bmatrix}$$

$$\begin{bmatrix} u \\ \dot{u} \end{bmatrix}_{t=0} = \begin{bmatrix} f \\ cf \end{bmatrix} \quad \Longrightarrow \quad \begin{bmatrix} u^+ \\ u^- \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} u \\ \dot{u} \end{bmatrix}_{t=0} = \begin{bmatrix} f \\ -cf \end{bmatrix} \quad \Longrightarrow \quad \begin{bmatrix} u^+ \\ u^- \end{bmatrix} = \begin{bmatrix} 0 \\ f \end{bmatrix}.$$

Complex representation: We can represent the state of the wave equation in the a complex form. We define the $\mathbf{U}$ and its temporal and spatial derivatives as follows:

$$\mathbf{U}(x,t) = \frac{\partial}{\partial x}u(x,t) + \boldsymbol{\imath}\frac{\partial}{\partial t}(x,t)$$

$$\mathbf{U}_{k,\omega} = e^{\boldsymbol{\imath}(kx-\omega t)}$$

$\frac{\partial}{\partial t}$	$\frac{\partial}{\partial x}$	$\frac{\partial^2}{\partial t^2}$	$\frac{\partial^2}{\partial x^2}$
$-\boldsymbol{\imath}\omega\mathbf{U}$	$\boldsymbol{\imath}k\mathbf{U}$	$-\omega^2\mathbf{U}$	$-k^2\mathbf{U}$

Notice that inserting the second derivatives into the wave equation (Equation 2.1) we obtain

$$\omega^2 = c^2k^2 \Longleftrightarrow \omega = \pm ck,$$

giving us a nice representation of right and left travelling waves, determined by whether ω and k have different signs. A Fourier transform of $\mathbf{U}$ yields the amplitude of every right and left moving mode.

Boundaries: Back to a purely real $u(x,t) \in \mathbb{R} \times \mathbb{R} \to \mathbb{R}$: If we guarantee

$$\left(\frac{\partial}{\partial x} - \frac{\partial}{\partial t}\right) u = 0 \Big|_{x=0}$$
$$\left(\frac{\partial}{\partial x} + \frac{\partial}{\partial t}\right) u = 0 \Big|_{x=L}$$

waves colliding with the boundary will be perfectly absorbed. For multidimensional absorbing boundary conditions, consider the derivation in [4]:

$$\left(\frac{\partial}{\partial x} - \sqrt{\frac{1}{c^2}\frac{\partial^2}{\partial t^2} - \frac{\partial^2}{\partial y^2} - \frac{\partial^2}{\partial z^2}}\right) = 0 \Bigg|_{x=0} \tag{2.13}$$

Equation 2.13 describes a perfectly absorbing boundary in $x = 0$, however evaluating the square root is not possible using local stencils, because it is a global operator. For a thorough explanation on global operators, the reader is referred to [4]. It is clear that Equation 2.13 is the three-dimensional equivalent of Equation 2.11, since clearly

$$\left(\frac{\partial}{\partial x} - \sqrt{\frac{1}{c^2}\frac{\partial^2}{\partial t^2} - \frac{\partial^2}{\partial y^2} - \frac{\partial^2}{\partial z^2}}\right)\left(\frac{\partial}{\partial x} + \sqrt{\frac{1}{c^2}\frac{\partial^2}{\partial t^2} - \frac{\partial^2}{\partial y^2} - \frac{\partial^2}{\partial z^2}}\right) = \Delta + \frac{\partial^2}{\partial t^2} = \Box \tag{2.14}$$

The Taylor expansions of Equation 2.13 to first and second order are called Mur's First and Second order Absorbing Boundary Conditions [5]. The expansions are as follows:

$$\begin{aligned} \pm\frac{\partial^2 u}{\partial x \partial t} - \frac{1}{c}\frac{\partial^2 u}{\partial t^2} &= 0 \quad \text{(first order)}, \\ \pm\frac{\partial^2 u}{\partial x \partial t} - \frac{1}{c}\frac{\partial^2 u}{\partial t^2} - \frac{c}{2}\frac{\partial^2 u}{\partial y^2} - \frac{c}{2}\frac{\partial^2 u}{\partial z^2} &= 0 \quad \text{(second order)} \end{aligned} \tag{2.15}$$

These are two expressions that only depend on derivatives that are approximatable by stencils. It should be noted that the second order variant from Equation 2.15 is equivalent to

$$\pm\frac{\partial u}{\partial x} - \frac{1}{c}\frac{\partial u}{\partial t} = 0$$

which is a particularly easy expression to discretize, see Equation 3.15.

2.1.3 The Dirac equation

As of Equation 2.14, we can construct a real factorization for the one-dimensional wave equation. Constructing a real (or even complex) factorization of the three-dimensional wave equation

$$\left(\frac{1}{c^2}\frac{\partial^2}{\partial t^2} - \frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial y^2} - \frac{\partial^2}{\partial z^2}\right) u = 0 = \Box u$$

is not possible. We can however simply write down the equation and then use the desired result to define our coefficients γ_i:

$$\left(\gamma_0\frac{1}{c}\frac{\partial}{\partial t}+\gamma_1\frac{\partial}{\partial x}+\gamma_2\frac{\partial}{\partial y}+\gamma_3\frac{\partial}{\partial z}\right)\left(\gamma_0\frac{1}{c}\frac{\partial}{\partial t}+\gamma_1\frac{\partial}{\partial x}+\gamma_2\frac{\partial}{\partial y}+\gamma_3\frac{\partial}{\partial z}\right)\stackrel{!}{=}\Box$$

Expanding the terms, we obtain a condition for γ_i

$$\gamma_i\gamma_j+\gamma_j\gamma_i=\begin{cases}2 & \text{if } i=j\\ 0 & \text{otherwise}\end{cases}$$

These conditions are fulfilled by the Dirac-matrices $\gamma_{1..4}$ [6, 7].

2.1.4 Energy conservation

The source-free wave equation conserves the energy contained in $\mathbf{A}^\alpha$. The energy density of $\mathbf{A}^\alpha$ is obtained through the electromagnetic energy-stress tensor derived in [8]:

$$T_{\mu\nu}=T^{\mu\nu}=F^{\mu\alpha}F^{\nu}_{\alpha}-\frac{1}{4}\eta^{\mu\nu}F_{\alpha\beta}F^{\alpha\beta} \tag{2.16}$$

where $F_{\mu\nu}$ is the Faraday Tensor and $\eta^{\mu\nu}$ is the Minkowski metric:

$$F^{\mu\nu}=\partial^\mu\mathbf{A}^\nu-\partial^\mu\mathbf{A}^\nu. \tag{2.17}$$

and

$$\eta^{\mu\nu}=\begin{bmatrix}-1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} \tag{2.18}$$

Energy Density:
The energy density per unit volume is given by

$$T_{00}=\frac{1}{2}\left(\varepsilon_0\mathbf{E}^2+\frac{1}{\mu_0}\mathbf{B}^2\right).$$

Evaluated in SI units this yields an energy density of units $\frac{\mathrm{J}}{\mathrm{m}^3}=\frac{\mathrm{N}}{\mathrm{m}^2}=\mathrm{Pa}$.
Energy Flux:
The first row (or column, since $T^{\mu\nu}=T^{\nu\mu}$) of $T^{\mu\nu}$ reads

$$\begin{bmatrix}T_{00}&\frac{1}{c}S_x&\frac{1}{c}S_y&\frac{1}{c}S_z\end{bmatrix}$$

where

$$S = \frac{1}{\mu_0}(\mathbf{E} \times \mathbf{B}) \tag{2.19}$$

denotes the **Poynting Vector**. The unit of S is $\frac{\mathrm{W}}{\mathrm{m}^2}$, therefore the unit of $\frac{1}{c}S$ is $\frac{\mathrm{J}}{\mathrm{m}^3}$.

We can write the following conservation law:

$$\boldsymbol{\nabla} \cdot S = -\frac{\mathrm{d}\mathcal{U}}{\mathrm{d}t} \quad \text{where } \mathcal{U} = T_{00} \tag{2.20}$$

As always, we can rewrite Equation 2.20 in integral form for any compact $\Omega \subset \mathbb{R}^3$:

$$\frac{\mathrm{d}\int_\Omega \mathcal{U}}{\mathrm{d}t} = \int_{\partial\Omega} S \cdot \mathrm{d}\boldsymbol{n} \tag{2.21}$$

Verifying if Equation 2.21 is fulfilled at all times is numerically trivial and a good sanity check on a FDTD implementation.

2.1.5 Transforming the fields

The electric and magnetic field are transformed from laboratory to bunch coordinates as seen in [9]:

$$\mathbf{E}_{\text{bunch}} = \gamma\Big(\mathbf{E}_{\text{lab}} + \boldsymbol{v} \times \mathbf{B}_{\text{lab}}\Big) - (\gamma - 1)(\mathbf{E}_{\text{lab}} \cdot \hat{\boldsymbol{v}})\hat{\boldsymbol{v}} \tag{2.22}$$

$$\mathbf{B}_{\text{bunch}} = \gamma\left(\mathbf{B}_{\text{lab}} - \frac{\boldsymbol{v} \times \mathbf{E}_{\text{lab}}}{c^2}\right) - (\gamma - 1)(\mathbf{B}_{\text{lab}} \cdot \hat{\boldsymbol{v}})\hat{\boldsymbol{v}} \tag{2.23}$$

where

$\boldsymbol{v}$ denotes the frame's relative velocity

$\hat{\boldsymbol{v}}$ denotes the normalized velocity $\dfrac{\boldsymbol{v}}{\|\boldsymbol{v}\|}$

and γ denotes the corresponding Lorentz factor $\dfrac{1}{\sqrt{1 - \frac{\|v\|_2^2}{c^2}}}$

Furthermore, for the special case that $\boldsymbol{v} = \begin{bmatrix} 0 \\ 0 \\ v_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ c\beta_z \end{bmatrix}$ the relativistic Doppler effect is described by the formula

$$\begin{aligned}\lambda_{\text{lab}} &= \lambda_{\text{bunch}}\sqrt{\frac{1-\beta_z}{1+\beta_z}} \\ &= \frac{1}{\gamma(1-\beta)}\lambda_{\text{bunch}}\end{aligned} \tag{2.24}$$

Where λ denotes the wavelength of an electromagnetic wave. In contrast, the Poynting vector transforms as follows

$$\mathbf{E}_{\text{lab}} \times \mathbf{B}_{\text{lab}} = \frac{1+\beta_z}{1-\beta_z}\left(\mathbf{E}_{\text{bunch}} \times \mathbf{B}_{\text{bunch}}\right) \tag{2.25}$$

Equation 2.24 and Equation 2.25 are different in that the scaling factor of Equation 2.24 is the root of the scaling factor of Equation 2.25. This is due to the fact that Equation 2.25 doesn't take into account that a stationary detector in one frame is not stationary in another. For a thorough derivation on the radiation passing through a screen measured in different inertial frames (or the radiation reflected off it), the reader is referred to [10, Section 10.9.2].

2.2 Initial conditions

2.2.1 Electrostatic initial conditions

If an initial charge distribution is known and the initial current distribution is zero, simply solving a Poisson equation

$$-\Delta\varphi = \frac{\rho}{\varepsilon_0} \tag{2.26}$$

will yield a suitable

$$\mathbf{A}^{\alpha} = \begin{bmatrix}\varphi \\ \mathrm{A}_0 \\ \mathrm{A}_1 \\ \mathrm{A}_2\end{bmatrix}, \mathrm{A} = \mathbf{0}$$

Several approaches on fast Poisson solvers exist [11; 13].

2.2.2 Boosted electrostatic initial conditions

If all particles are moving with the exact same speed, one can compute an initial condition in an exactly co-moving frame according to Equation 2.26, and then use Equation 2.22 to transform them back to the original frame. Analogous to the explanation in paragraph Correcting the time, we end up with grid values captured at different points in time. This artifact is usually ignored [14].

2.2.3 Lienard-Wiechert potentials

A generalized and classically perfect condition for the electromagnetic fields imposed by a moving charge is given by the Liénard-Wiechert Potentials [15]: For a particle with a trajectory $\boldsymbol{R}(t)$ and velocity

$$\boldsymbol{\beta}(t) = \dot{\boldsymbol{R}}(t),$$

$$\Phi(\boldsymbol{r}, t) = \frac{q}{4\pi\varepsilon_0(1 - \boldsymbol{\beta}(t_{\text{ret}}) \cdot \boldsymbol{n})|r - \boldsymbol{R}(t_{\text{ret}})|} \tag{2.27}$$

and also the vector potential

$$A(\boldsymbol{r}, t) = \frac{\boldsymbol{\beta}(t_{\text{ret}})}{c}\varphi(\boldsymbol{r}, t), \tag{2.28}$$

where

$$t_{\text{ret}}(t, \boldsymbol{r}) = t - \frac{1}{c}|\boldsymbol{r} - \boldsymbol{R}(t_{\text{ret}})| \tag{2.29}$$

and

$$\boldsymbol{n} = \frac{\boldsymbol{r} - \boldsymbol{R}}{|\boldsymbol{r} - \boldsymbol{R}|} \tag{2.30}$$

For a proof as to why Equation 2.29 always has a unique solution, the reader is referred to [16].
For the special case of a straight line

$$\boldsymbol{R}(t) = \begin{bmatrix} 0 \\ 0 \\ c\beta t \end{bmatrix},$$

we can solve the value of $t_{\text{ret}}(\boldsymbol{r})$ analytically.

$$t_{\text{ret}}(t, \boldsymbol{r}) = \frac{ct - \beta r_2 - \sqrt{\beta^2 c^2 t^2 - \beta^2 r_0^2 - \beta^2 r_1^2 - 2\beta c r_2 t + r_0^2 + r_1^2 + r_2^2}}{c(1 - \beta^2)} \tag{2.31}$$

We can insert the analytical solution Equation 2.31 into Equation 2.27 and Equation 2.28 to obtain a generally correct expression for $\mathbf{A}^\alpha$ at any location. However, this can become extremely expensive to do. For a grid with resolution N^3 and N_p particles, we have to evaluate Equation 2.31 $N^3 N_p$ times. For the special case that all particles share a common $\boldsymbol{\beta}$, we can implement an FFT-based convolution solver similar analogous to the Vico-Greengard method presented in [12], reducing the cost to $\approx (2N)^3 \log N + N_p$. For the general case, the evaluation of Equation 2.31, let alone a solution for more complex trajectories, do not reduce to a convolution and are not optimizable any further.

2.3 Lorentz Force

2.3.1 Relativistic limit

An analysis of a charged particle moving in an electromagnetic field is best done by representing it in a covariant four-momentum:

$$p^\alpha = \left\{ \frac{\mathcal{E}}{c}, \gamma \boldsymbol{p} \right\} = \gamma \{ mc, \boldsymbol{p} \} = \frac{\mathrm{d}x^\alpha}{\mathrm{d}\tau} = mu^\alpha \tag{2.32}$$

where $\mathcal{E}$ denotes the total energy (symbol $\mathbf{E}$ is reserved for the electric field) τ denotes *proper* time and therefore

$$\gamma \, \mathrm{d}\tau = \mathrm{d}t \tag{2.33}$$

and

$$u^\alpha = \frac{\mathrm{d}x^\alpha}{\mathrm{d}\tau},$$
$$u_\alpha = \frac{\mathrm{d}x_\alpha}{\mathrm{d}\tau}.$$

A fully relativity-compliant rate of change for p^α can be written with the Faraday tensor $\mathbf{F}$ as follows

$$\frac{\mathrm{d}p^\alpha}{\mathrm{d}t} = q\mathrm{F}^{\alpha\beta} u_\beta \tag{2.34}$$

Inspection of the first component of Equation 2.34 yields [17]:

$$\frac{\mathrm{d}\mathcal{E}}{\mathrm{d}t} = q\mathbf{E} \cdot \boldsymbol{u}. \tag{2.35}$$

As a consequence, a change of energy is only possible by moving non-perpendicularly to a nonzero electric field.

Inspection of the second component of Equation 2.34 yields: [17]

$$\frac{\mathrm{d}p}{\mathrm{d}\tau} = q\gamma [\mathbf{E} + u \times \mathbf{B}]_x$$

which, due to Equation 2.33 is consistent with the classical Lorentz force

$$\mathbf{F} = q\mathbf{E} + q\mathbf{v} \times \mathbf{B}.$$

We can characterize a particle's motion in a constant external magnetic field of strength B_0 for two different regimens, the nonrelativistic and the relativistic one. For the nonrelativistic case we have:

$$\frac{m_e v^2}{r} = qvB_0$$
$$\iff r = \frac{m_e v}{qB_0},$$

which is known as the synchrotron radius. whereas for the relativistic case we have [17]

$$r = \frac{m_e \gamma v}{qB_0}.$$

This expression for the radius can be used to verify that the Boris Pusher is implemented in a relativistically correct way.

Chapter 3

Discretization

In order to solve Equation 2.2 or Equation 2.8, several approaches exist [18, 19, 10, 5]. We will pursue variation of [19] in potential form, called *Finite Difference Time Domain*, usually abbreviated with FDTD.

3.1 Finite Difference Time Domain

In order to solve Equation 2.8 numerically, we discretize the field $\mathbf{A}^\alpha$ onto a grid with resolution n_x, n_y, n_z and spacing $\Delta_x, \Delta_y, \Delta_z$ in each direction. Spatial and temporal derivatives are computed with local convolutions called stencils. A finite-difference stencil is a formula used to approximate derivatives at a given position in a grid using function values sampled at finite intervals around the point of interest. The use of the word "local" is warranted since most of the time, only the 2-6 nearest samples are taken into account to approximate operators like $\frac{\partial}{\partial x}$ or Δ. We use the following notation to describe a sample of u in space and time:

$$u^n_{i,j,k}$$

which denotes the value of u at the point

$$\begin{bmatrix} i\Delta x \\ i\Delta y \\ i\Delta z \end{bmatrix} \text{ at time } n\Delta t$$

Because both superscript and subscript indices are required, from here on the four-potential is denoted simply as $\mathbf{A}$. For a technical definition and introduction to stencils, the reader is referred to [20] and Section 3.1.2.

3.1.1 FDTD Discretization

Equation 2.8 is discretized as follows [21]:

$$\begin{aligned}
\frac{\partial^2 \mathbf{A}^n_{i,j,k}}{\partial x^2} &= \frac{\mathbf{A}^n_{i+1,j,k} - 2\mathbf{A}^n_{i,j,k} + \mathbf{A}^n_{i-1,j,k}}{\Delta x^2} \\
\frac{\partial^2 \mathbf{A}^n_{i,j,k}}{\partial y^2} &= \frac{\mathbf{A}^n_{i,j+1,k} - 2\mathbf{A}^n_{i,j,k} + \mathbf{A}^n_{i,j-1,k}}{\Delta y^2} \\
\frac{\partial^2 \mathbf{A}^n_{i,j,k}}{\partial z^2} &= \frac{\mathbf{A}^n_{i,j,k+1} - 2\mathbf{A}^n_{i,j,k} + \mathbf{A}^n_{i,j,k-1}}{\Delta z^2} \\
\frac{\partial^2 \mathbf{A}^n_{i,j,k}}{\partial t^2} &= \frac{\mathbf{A}^{n+1}_{i,j,k} - 2\mathbf{A}^n_{i,j,k} + \mathbf{A}^{n-1}_{i,j,k}}{\Delta t^2}
\end{aligned} \tag{3.1}$$

We solve Equation 2.8 with the Equation 3.1 for $\mathbf{A}^{n+1}_{i,j,k}$ and obtain:

$$\begin{aligned}
\mathbf{A}^{n+1}_{i,j,k} = &-\mathbf{A}^{n-1}_{i,j,k} + \alpha_1 \mathbf{A}^n_{i,j,k} + \alpha_2 \mathbf{A}^n_{i+1,j,k} + \alpha_2 \mathbf{A}^n_{i-1,j,k} \\
&+\alpha_4 \mathbf{A}^n_{i,j+,k} + \alpha_4 \mathbf{A}^n_{i,j-1,k} + \alpha_6 \mathbf{A}^n_{i,j,k+1} + \alpha_6 \mathbf{A}^n_{i,j,k-1} \\
&+\alpha_8 \mathbf{J}^n_{i,j,k}
\end{aligned} \tag{3.2}$$

where

$$\begin{aligned}
\alpha_1 &= 2\left(1 - \left(\frac{c\Delta t}{\Delta x}\right)^2 - \left(\frac{c\Delta t}{\Delta y}\right)^2 - \left(\frac{c\Delta t}{\Delta z}\right)^2\right) \\
\alpha_2 &= \left(\frac{c\Delta t}{\Delta x}\right)^2 \\
\alpha_4 &= \left(\frac{c\Delta t}{\Delta y}\right)^2 \\
\alpha_6 &= \left(\frac{c\Delta t}{\Delta z}\right)^2 \\
\alpha_8 &= (c\Delta t)^2
\end{aligned} \tag{3.3}$$

The term $\mathbf{J}^n_{i,j,k}$ represents the source term at timestep $n\Delta t$:

$$\mathbf{J}^n_{i,j,k} = \begin{bmatrix} \frac{\rho^n_{i,j,k}}{\varepsilon_0} \\ \mu_0 J^n_{x,i,j,k} \\ \mu_0 J_{y,i,j,k} \\ \mu_0 J_{z,i,j,k} \end{bmatrix} \tag{3.4}$$

How the the values for ρ and J are obtained for Equation 3.4 is described in the section about Quantity interpolation.

3.1.2 Accuracy Analysis

In order to analyze how closely a stencil approximates a given derivative, consider the Taylor Expansion of an arbitrary analytical function f at 0:

$$f(x) = \sum_{k=0}^{\infty} \frac{\mathrm{d}^k f(0)}{\mathrm{d}x^k} \frac{x^k}{k!}$$

The evaluation of the k-th term on a gridpoint $i\Delta x, i \in \mathbb{Z}$ is

$$F_i^k = \frac{\mathrm{d}^k f(0)}{\mathrm{d}x^k} \frac{(i\Delta x)^k}{k!}$$

We define the p-th derivative-basis vector b_p^n as follows

$$b_p^n = \begin{bmatrix} \begin{cases} \dfrac{\mathrm{d}^n f(0)}{\mathrm{d}x^n} & \text{if } p = n \\ 0 & \text{otherwise} \end{cases} \\ \vdots \end{bmatrix}$$

Therefore, F_i^k expressed in basis b_p^n is

$$\mathbb{F}_i^k = \frac{(i\Delta x)^k}{k!}$$

To obtain the weight vector w for the n-th derivative the following linear system of equations has to be solved [22]:

$$\begin{aligned} \mathbb{F}_i^k w^i &= e_n \\ e_n^i &= \begin{bmatrix} \begin{cases} 1 & \text{if } i{=}n \\ 0 & \text{otherwise} \end{cases} \\ \vdots \end{bmatrix} \end{aligned} \tag{3.5}$$

Solving Equation 3.5 amounts to solving a linear system of equations. As an example, let's derive a first-order accurate stencil for the second derivative using three points $\{-\Delta x, 0, \Delta x\}$. The linear system of equations that we obtain is

$$\begin{bmatrix} 1 & 1 & 1 \\ -\Delta x & 0 & \Delta x \\ \frac{\Delta x^2}{2} & 0 & \frac{\Delta x^2}{2} \\ -\frac{\Delta x^3}{6} & 0 & \frac{\Delta x^3}{6} \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \end{bmatrix}$$

As indicated by the red line, we can only guarantee a solution for the top three entries. The solution obtained is

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \frac{1}{\Delta x^2} \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}$$

However, due to a fortunate coincidence, our solution also happens to satisfy the fourth equation. We therefore obtain second order accuracy "for free", since the linear combination of expansion is as follows:

$$\Delta x^2 \left(0 + 0 + \frac{\mathrm{d}^2 f}{\mathrm{d}x} + 0 + \sum_{i=4}^{\infty} \Delta x^{i-2} \frac{\mathrm{d}^i f}{\mathrm{d}x^i} \right)$$

The 0 coefficient for the cubic term causes the coincidental increase in accuracy from order 1 to order 2.

3.1.3 Numerical Dispersion

One effect of the inaccuracy derived in Section 3.1.2 is a spurious slowing effect on high-frequency waves. The analysis of a plane wave parameterized by

$$u(x,t) = e^{i(\boldsymbol{k}\cdot\boldsymbol{x}-\omega t)}$$

travelling through a discretized domain gives insight into why this slowdown happens and eventually yield a relation between the temporal frequency ω and the spatial wavenumber k

Numerical Dispersion: Consider the source-free 1-Dimensional FDTD update:

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} = \frac{1}{c^2} \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} \tag{3.6}$$

As described in [10], we replace all occurrences of u with a plane-wave with frequency ω:

$$u_i^n = e^{\boldsymbol{i}(k(i\Delta x) - \omega(n\Delta t))}$$

where $\boldsymbol{i}$ denotes the imaginary unit.
After some manipulation and the usage of the identity

$$e^{\boldsymbol{i}\varphi} + e^{-\boldsymbol{i}\varphi} = 2\cos(\varphi)$$

Equation 3.6 becomes

$$\begin{aligned} 2\frac{\cos(\Delta t\omega)}{c^2\Delta t^2} - 2\frac{\cos(\Delta x k)}{\Delta x^2} + \frac{2}{\Delta x^2} - \frac{2}{c^2\Delta t^2} &= 0 \\ \iff \cos(\Delta x k)\frac{c^2\Delta t^2}{\Delta x^2} - \frac{c^2\Delta t^2}{\Delta x^2} + 1 &= \cos(\Delta t\omega) \end{aligned}$$

We solve for k following the simplifications outlined in [10, Section 2.6.3] and obtain:

$$k = \frac{1}{\Delta x} \cos^{-1}\left(1 + \left(\frac{\Delta x}{c\Delta t}\right)^2 (\cos(\omega \Delta t) - 1)\right) \tag{3.7}$$

Equation 3.7 is the dispersion relation for the 1D FDTD scheme: it relates the dependence of k on c and ω with respect to the domain discretization parameters Δx and Δt. Comparing it to the dispersion relation of an electromagnetic wave travelling in vacuum:

$$k = \frac{\omega}{c}$$

we notice that this linear dependence is only fulfilled for $\Delta x = c\Delta t$, since in that case Equation 3.7 simplifies to

$$k = \frac{1}{\Delta x} \cos^{-1}\left(\cos\left(\frac{\omega}{c}\Delta x\right)\right) = \frac{\omega}{c}$$

Unfortunately, setting $\Delta x = c\Delta t$ is only possible for one dimension, due to the CFL condition which is derived abundantly in literature [23, 21, 10]:

$$\Delta t \leq \frac{1}{c\sqrt{\sum_{i=0}^{N} \left(\frac{1}{\Delta x_i}\right)^2}} \tag{3.8}$$

where N is the number of dimensions of the simulation. Evidently, $\Delta x = c\Delta t$ is only satisfiable for $N = 1$. For $N = 3$, we have

$$\Delta t \leq \frac{1}{\sqrt{3}} \Delta x \approx 0.57 \Delta x$$

However, dispersion can be nullified for one direction. Following some steps out of [21], we obtain a modified version of the FDTD update equation Equation 3.2:

$$\begin{aligned}
\mathbf{A}^{n+1}_{i,j,k} = &-\mathbf{A}^{n-1}_{i,j,k} + \alpha'_1 \mathbf{A}^{n}_{i,j,k} \\
&+\alpha'_2\left(\mathcal{A}\mathbf{A}^{n}_{i+1,j,k-1} + (1-2\mathcal{A})\mathbf{A}^{n}_{i+1,j,k} + \mathcal{A}\mathbf{A}^{n}_{i+1,j,k+1}\right) \\
&+\alpha'_3\left(\mathcal{A}\mathbf{A}^{n}_{i-1,j,k-1} + (1-2\mathcal{A})\mathbf{A}^{n}_{i-1,j,k} + \mathcal{A}\mathbf{A}^{n}_{i-1,j,k+1}\right) \\
&+\alpha'_4\left(\mathcal{A}\mathbf{A}^{n}_{i,j+1,k-1} + (1-2\mathcal{A})\mathbf{A}^{n}_{i,j+1,k} + \mathcal{A}\mathbf{A}^{n}_{i,j+1,k+1}\right) \\
&+\alpha'_5\left(\mathcal{A}\mathbf{A}^{n}_{i,j-1,k-1} + (1-2\mathcal{A})\mathbf{A}^{n}_{i,j-1,k} + \mathcal{A}\mathbf{A}^{n}_{i,j-1,k+1}\right) \\
&+\alpha'_6 \mathbf{A}^{n}_{i,j,k+1} + \alpha'_7 \mathbf{A}^{n}_{i,j,k-1} + \alpha'_8 \mathbf{J}^{n}_{i,j,k}
\end{aligned} \tag{3.9}$$

Where α'_i are obtained as follows:

$$
\begin{aligned}
\alpha_1' &= 2\left(1-(1-2\mathcal{A})\left(\frac{c\Delta t}{\Delta x}\right)^2-(1-2\mathcal{A})\left(\frac{c\Delta t}{\Delta y}\right)^2-\left(\frac{c\Delta t}{\Delta z}\right)^2\right)\\
\alpha_2' &= \alpha_3' = \left(\frac{c\Delta t}{\Delta x}\right)^2\\
\alpha_4' &= \alpha_5' = \left(\frac{c\Delta t}{\Delta y}\right)^2\\
\alpha_6' &= \alpha_7' = \left(\frac{c\Delta t}{\Delta z}\right)^2-2\mathcal{A}\left(\frac{c\Delta t}{\Delta x}\right)^2-2\mathcal{A}\left(\frac{c\Delta t}{\Delta y}\right)^2\\
\alpha_8' &= (c\Delta t)^2
\end{aligned}
\tag{3.10}
$$

The following conditions have to be fulfilled for discretization parameters:

1.

$$
\mathcal{A} > \frac{1}{4}\text{ , in our case: }\frac{1}{4}\left[1+\frac{0.02}{\left(\frac{\Delta z}{\Delta x}\right)^2+\left(\frac{\Delta z}{\Delta y}\right)^2}\right]
$$

2.

$$
\Delta t = \frac{\Delta z}{c}
$$

3.

$$
\left(\frac{\Delta z}{\Delta x}\right)^2+\left(\frac{\Delta z}{\Delta y}\right)^2 < 1
$$

Equation 3.9 and Equation 3.10 are also found in [21], with a typo corrected in the formula for α_6' and α_7'

3.1.4 Boundary condition

Reflecting boundary conditions: If we simply lock the values on the boundary to zero as follows:

$$
\begin{aligned}
\mathbf{A} &= \mathbf{0} \quad \text{on } \partial\Omega\\
\mathbf{A}_{i,j,k}^{n+1} &= \mathbf{0} \quad \text{for } \{i,j,k\} \quad \text{on the boundary}
\end{aligned}
$$

we obtain a boundary condition that reflects incoming waves perfectly. These boundary conditions are called Dirichlet Boundary Conditions. For simulation domains enclosed by perfect conductors this is desirable, however we also want to perform simulations in open space.

Absorbing boundary conditions: As outlined in Section 2.1.2, we want waves in our domain to be able to freely leave through the boundary.

First Order Mur Absorbing Boundary Conditions:

$$\frac{\partial \mathbf{A}}{\partial \mathbf{n}} - \frac{1}{c}\frac{\partial \mathbf{A}}{\partial t} = 0 \tag{3.11}$$

Let's take a look at Equation 3.11 on the boundary at the $x = 0$ boundary:

$$x = 0, \mathbf{n} = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$$

$$\frac{\partial \mathbf{A}}{\partial x} + \frac{1}{c}\frac{\partial \mathbf{A}}{\partial t} = 0 \tag{3.12}$$

In order discretize Equation 3.12 with second order accuracy, we have to perform the discretization half a spatial step away from the boundary and half a timestep into the future. The rationale behind this is outlined in Section 3.1.2, the accuracy of a two-point stencil is second order in the arithmetic mean of the two points. We have

$$\begin{aligned} \left.\frac{\partial \mathbf{A}}{\partial x}\right|_{x=\frac{\Delta x}{2}, t=\frac{\Delta t}{2}} &\approx \frac{\mathbf{A}_1^n - \mathbf{A}_0^n + \mathbf{A}_1^{n+1} - \mathbf{A}_0^{n+1}}{2\Delta x} \\ \left.\frac{\partial \mathbf{A}}{\partial t}\right|_{x=\frac{\Delta x}{2}, t=\frac{\Delta t}{2}} &\approx \frac{\mathbf{A}_0^{n+1} - \mathbf{A}_0^n + \mathbf{A}_1^{n+1} - \mathbf{A}_1^n}{2\Delta t}. \end{aligned}$$

Solving Equation 3.12 with these discretizations applied for $\mathbf{A}_0^{n+1}$ yields [4, 5]:

$$\mathbf{A}_0^{n+1} = \frac{-\mathbf{A}_0^n c\Delta t + \mathbf{A}_0^n \Delta x + \mathbf{A}_1^n c\Delta t + \mathbf{A}_1^n \Delta x + \mathbf{A}_1^{n+1} c\Delta t - \mathbf{A}_1^{n+1}\Delta x}{c\Delta t + \Delta x}$$

$$\iff \mathbf{A}_0^{n+1} = \mathbf{A}_1^n + \frac{c\Delta t - \Delta x}{c\Delta t + \Delta x}(\mathbf{A}_1^{n+1} - \mathbf{A}_0^n) \tag{3.13}$$

Where $\mathbf{A}_i$ represents $\mathbf{A}_{i,j,k}$, the transversal indices j, k are dropped for brevity. It should be noted that Equation 3.13 only depends on two temporal instances of $\mathbf{A}$. The discretization found in [21] is done differently. instead of Equation 3.12, the following expression is used on the $x = 0$ boundary:

$$\frac{\partial^2 \mathbf{A}}{\partial x \partial t} + \frac{1}{c}\frac{\partial^2 \mathbf{A}}{\partial t^2} = 0 \tag{3.14}$$

Following [21], Equation 3.14 is then discretized as follows:

$$\begin{aligned}\left.\frac{\partial^2 \mathbf{A}}{\partial x \partial t}\right|_{x=\frac{\Delta x}{2},t=0} &\approx \frac{1}{2\Delta t}\left(\frac{\mathbf{A}_1^{n+1}-\mathbf{A}_0^{n+1}}{\Delta x}-\frac{\mathbf{A}_1^{n-1}-\mathbf{A}_0^{n-1}}{\Delta x}\right)\\ \left.\frac{1}{c}\frac{\partial^2 \mathbf{A}}{\partial t^2}\right|_{t=0} &\approx \frac{1}{2c}\left(\frac{\mathbf{A}_1^{n+1}-2\mathbf{A}_1^{n}+\mathbf{A}_1^{n-1}}{\Delta t^2}+\frac{\mathbf{A}_0^{n+1}-2\mathbf{A}_0^{n}+\mathbf{A}_0^{n-1}}{\Delta t^2}\right)\end{aligned} \tag{3.15}$$

Solving for $\mathbf{A}_0^{n+1}$, we obtain

$$\mathbf{A}_0^{n+1} = \beta_2 \mathbf{A}_1^{n-1} + \beta_0\Big(\mathbf{A}_0^{n-1}+\mathbf{A}_1^{n+1}\Big) + \beta_1\Big(\mathbf{A}_0^{n}+\mathbf{A}_1^{n}\Big) \tag{3.16}$$

where

$$\begin{aligned}\beta_0 &= \frac{c\Delta t-\Delta x}{c\Delta t+\Delta x}\\ \beta_1 &= \frac{2\Delta x}{c\Delta t+\Delta x}\\ \beta_2 &= -1\end{aligned} \tag{3.17}$$

There is a clear drawback in terms of memory footprint: the values of $\mathbf{A}$ are required at three different timesteps. For this reason, we chose to use the discretization as of Equation 3.13. Second Order Mur's ABC: Expanding Equation 2.13 to the second order, we obtain the equation implying second order absorbing boundary conditions at the $x = 0$ boundary.

$$\frac{1}{c}\frac{\partial^2 u}{\partial \boldsymbol{x}\partial t}-\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2}+\frac{1}{2}\left(\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}\right)=0. \tag{3.18}$$

Equation 3.18 can be discretized as follows:

$$\text{Equal to first order}\begin{cases}\dfrac{\partial^2 \mathbf{A}}{\partial x\partial t} \approx \dfrac{1}{2\Delta t}\left(\dfrac{\mathbf{A}_{1,j,k}^{n+1}-\mathbf{A}_{1,j,k}^{n+1}}{\Delta x}-\dfrac{\mathbf{A}_{1,j,k}^{n-1}-\mathbf{A}_{0,j,k}^{n-1}}{\Delta x}\right)\\ \dfrac{1}{c}\dfrac{\partial^2 \mathbf{A}}{\partial t^2} \approx \dfrac{1}{2c}\left(\dfrac{\mathbf{A}_{1,j,k}^{n+1}-2\mathbf{A}_{1,j,k}^{n}+\mathbf{A}_{1,j,k}^{n-1}}{\Delta t^2}+\dfrac{\mathbf{A}_{0,j,k}^{n+1}-2\mathbf{A}_{0,j,k}^{n}+\mathbf{A}_{0,j,k}^{n-1}}{\Delta t^2}\right)\end{cases}$$

$$\frac{1}{c}\frac{\partial^2 \mathbf{A}}{\partial y^2} \approx \frac{1}{2c}\left(\frac{\mathbf{A}_{1,j+1,k}^{n}-2\mathbf{A}_{1,j,k}^{n}+\mathbf{A}_{1,j-1,k}^{n}}{\Delta y^2}+\frac{\mathbf{A}_{0,j+1,k}^{n}-2\mathbf{A}_{0,j,k}^{n}+\mathbf{A}_{0,j-1,k}^{n}}{\Delta y^2}\right)$$

$$\frac{1}{c}\frac{\partial^2 \mathbf{A}}{\partial z^2} \approx \frac{1}{2c}\left(\frac{\mathbf{A}_{1,j,k+1}^{n}-2\mathbf{A}_{1,j,k}^{n}+\mathbf{A}_{1,j,k-1}^{n}}{\Delta y^2}+\frac{\mathbf{A}_{0,j,k+1}^{n}-2\mathbf{A}_{0,j,k}^{n}+\mathbf{A}_{0,j,k-1}^{n}}{\Delta z^2}\right)$$

We can solve these expressions for $\mathbf{A}_{0,j,k}^{n+1}$ and obtain:

$$\begin{aligned}\mathbf{A}_{0,j,k}^{n+1} = \gamma_0 \mathbf{A}_{0,j,k}^{n-1} &+ \gamma_1 \mathbf{A}_{0,j,k}^{n} &+ \gamma_2 \mathbf{A}_{1,j,k}^{n-1} + \gamma_3 \mathbf{A}_{1,j,k}^{n} + \gamma_4 \mathbf{A}_{1,j,k}^{n+1} + \\ \gamma_5 \mathbf{A}_{i+1,j,1}^{n} + \gamma_6 \mathbf{A}_{i-1,j,1}^{n} &+ \gamma_7 \mathbf{A}_{i,j+1,1}^{n} + \gamma_8 \mathbf{A}_{i,j-1,1}^{n} + \\ \gamma_9 \mathbf{A}_{i+1,j,0}^{n} + \gamma_{10} \mathbf{A}_{i-1,j,0}^{n} &+ \gamma_{11} \mathbf{A}_{i,j+1,0}^{n} + \gamma_{12} \mathbf{A}_{i,j-1,0}^{n}\end{aligned} \tag{3.19}$$

where

$$\begin{aligned}\gamma_0 = \gamma_4 &= \frac{c\Delta t - \Delta z}{c\Delta t + \Delta z} \\ \gamma_1 = \gamma_3 &= \frac{\Delta z\Big(2 - \big(c(\Delta t)(\Delta y)\big)^2 - \big(c(\Delta t)(\Delta x)\big)^2\Big)}{c\Delta t + \Delta z} \\ \gamma_2 &= -1 \\ \gamma_5 = \gamma_6 = \gamma_9 = \gamma_{10} &= \frac{\left(\frac{c\Delta t}{\Delta x}\right)^2 \Delta z}{2(c\Delta t + \Delta z)} \\ \gamma_7 = \gamma_8 = \gamma_{11} = \gamma_{12} &= \frac{\left(\frac{c\Delta t}{\Delta y}\right)^2 \Delta z}{2(c\Delta t + \Delta z)}\end{aligned} \tag{3.20}$$

Particular attention should be taken when implementating the second order absorbing boundary condition in cells which lie on edges and corners, as Equation 3.19 will contain values outside of the boundary in those cases. Equation 3.19 is therefore only applicable on faces. We consult the implementation of the second order absorbing boundary conditions derived in [24] for further details. Let us define the edge-weights γ^E as follows:

$$d = \frac{\frac{1}{\Delta x_1} + \frac{1}{\Delta x_2}}{4\Delta t} + \frac{3}{8c(\Delta t^2)}$$

$$\begin{aligned}\gamma_0^E &= \frac{-\frac{\frac{1}{\Delta x_2} - \frac{1}{\Delta x_1}}{4\Delta t} - \frac{3}{8c\Delta t\Delta t}}{d} \\ \gamma_1^E &= \frac{\frac{\frac{1}{\Delta x_2} - \frac{1}{\Delta x_1}}{4\Delta t} - \frac{3}{8c\Delta t\Delta t}}{d} \\ \gamma_2^E &= \frac{\frac{\frac{1}{\Delta x_2} + \frac{1}{\Delta x_1}}{4\Delta t} - \frac{3}{8c\Delta t\Delta t}}{d} \\ \gamma_3^E &= \frac{\frac{3}{4c\Delta t^2} - \frac{c}{4(\Delta x_e)^2}}{d} \\ \gamma_4^E &= \frac{\frac{c}{8(\Delta x_e)^2}}{d}\end{aligned} \tag{3.21}$$

In Equation 3.21, x_e signifies the direction which the edge is parallel to. The two directions $\{x_1, x_2\}$ signify the two directions perpendicular to the edge, ordered as follows:

$$\begin{aligned} x_e = x &\longrightarrow \{y, z\} \\ x_e = y &\longrightarrow \{z, x\} \\ x_e = z &\longrightarrow \{x, y\} \end{aligned} \tag{3.22}$$

The update equation for a value lying on the edge is given by

$$\begin{aligned} A_{i,j,k}^{n+1} = &\gamma_0^E(v_3 + v_8) + \\ &\gamma_1^E(v_5 + v_6) + \\ &\gamma_2^E(v_2 + v_9) + \\ &\gamma_3^E(v_1 + v_4 + v_7 + v_{10}) + \\ &\gamma_4^E(v_{12} + v_{13} + v_{14} + v_{15} + v_{16} + v_{17} + v_{18} + v_{19}) - v_{11} \end{aligned} \tag{3.23}$$

Where the vector v_n is given by:

$$\begin{aligned} &v_1 = A_{i,j,k}^n, \quad v_2 = A_{i,j,k}^{n-1} \\ &v_3 = A_{a_1}^{n+1}, \quad v_4 = A_{a_1}^n, \quad v_5 = A_{a_1}^{n-1} \\ &v_6 = A_{a_2}^{n+1}, \quad v_7 = A_{a_2}^n, \quad v_8 = A_{a_2}^{n-1} \\ &v_9 = A_{a_3}^{n+1}, \quad v_{10} = A_{a_3}^n, \quad v_{11} = A_{a_3}^{n-1} \\ &v_{12} = A_{a_0-a_h}^n, \quad v_{13} = A_{a_1-a_h}^n, \quad v_{14} = A_{a_2-a_h}^n, \quad v_{15} = A_{a_3-a_h}^n \text{ and} \\ &v_{16} = A_{a_0+a_h}^n, \quad v_{17} = A_{a_1+a_h}^n, \quad v_{18} = A_{a_2+a_h}^n, \quad v_{19} = A_{a_3+a_h}^n \end{aligned} \tag{3.24}$$

In the above notation, $a_0, a_1, a_2, a_3, a_h \in \mathbb{Z}^3$ are index triplets, defined as follows:

$$\begin{aligned} a_0 &= (i, j, k) \\ a_1 &= a_0 + \text{ the first normal vector basis vector defined by Equation 3.22} \\ a_2 &= a_0 + \text{ the second normal vector basis vector defined by Equation 3.22} \\ a_3 &= \text{ both 1st and 2nd normal vector basis vector defined by Equation 3.22} \\ a_h &= (i, j, k) + \text{ unit vector in edge direction} \end{aligned} \tag{3.25}$$

Due to this formula being so complicated, it is advisable to take a look at the code in the Section A.1.
The update formula for corners is can be described analogously. We define the corner-weights γ^C:

$$
\begin{aligned}
\gamma_0^C &= \frac{-\frac{1}{\Delta x}-\frac{1}{\Delta y}-\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2}, \quad \gamma_1^C = \frac{\frac{1}{\Delta x}-\frac{1}{\Delta y}-\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2} \\
\gamma_2^C &= \frac{-\frac{1}{\Delta x}+\frac{1}{\Delta y}-\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2}, \quad \gamma_3^C = \frac{-\frac{1}{\Delta x}-\frac{1}{\Delta y}+\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2} \\
\gamma_4^C &= \frac{\frac{1}{\Delta x}+\frac{1}{\Delta y}-\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2}, \quad \gamma_5^C = \frac{\frac{1}{\Delta x}-\frac{1}{\Delta y}+\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2} \\
\gamma_6^C &= \frac{-\frac{1}{\Delta x}+\frac{1}{\Delta y}+\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2}, \quad \gamma_7^C = \frac{\frac{1}{\Delta x}+\frac{1}{\Delta y}+\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2} \\
\gamma_8^C &= -\frac{-\frac{1}{\Delta x}-\frac{1}{\Delta y}-\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2}, \gamma_9^C = -\frac{\frac{1}{\Delta x}-\frac{1}{\Delta y}-\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2} \\
\gamma_{10}^C &= -\frac{-\frac{1}{\Delta x}+\frac{1}{\Delta y}-\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2}, \gamma_{11}^C = -\frac{-\frac{1}{\Delta x}-\frac{1}{\Delta y}+\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2} \\
\gamma_{12}^C &= -\frac{\frac{1}{\Delta x}+\frac{1}{\Delta y}-\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2}, \quad \gamma_{13}^C = -\frac{\frac{1}{\Delta x}-\frac{1}{\Delta y}+\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2} \\
\gamma_{14}^C &= -\frac{-\frac{1}{\Delta x}+\frac{1}{\Delta y}+\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2}, \gamma_{15}^C = -\frac{\frac{1}{\Delta x}+\frac{1}{\Delta y}+\frac{1}{\Delta z}}{8\Delta t}-\frac{1}{4c\Delta t^2} \\
\gamma_{16}^C &= \frac{1}{2c\Delta t^2}
\end{aligned}
\tag{3.26}
$$

An update formula for all eight corners can then be written as follows:

$$
\begin{aligned}
A_{i,j,k}^{n+1} = -\frac{1}{\gamma_0^C}\big(& v_1\gamma_{16}^C + v_2\gamma_8^C + \\
& v_3\gamma_1^C + v_4\gamma_{16}^C + v_5\gamma_9^C + \\
& v_6\gamma_2^C + v_7\gamma_{16}^C + v_8\gamma_{10}^C + \\
& v_9\gamma_3^C + v_{10}\gamma_{16}^C + v_{11}\gamma_{11}^C + \\
& v_{12}\gamma_4^C + v_{13}\gamma_{16}^C + v_{14}\gamma_{12}^C + \\
& v_{15}\gamma_5^C + v_{16}\gamma_{16}^C + v_{17}\gamma_{13}^C + \\
& v_{18}\gamma_6^C + v_{19}\gamma_{16}^C + v_{20}\gamma_{14}^C + \\
& v_{21}\gamma_7^C + v_{22}\gamma_{16}^C + v_{23}\gamma_{15}^C\big)
\end{aligned}
$$

where v_1 are defined by

$$
\begin{aligned}
&v_1 = A^n_{i,j,k}, && v_1 = A^{n-1}_{i,j,k} && \\
&v_3 = A^{n+1}_{i+1,j,k}, && v_4 = A^n_{i+1,j,k}, && v_5 = A^{n-1}_{i+1,j,k}, \\
&v_6 = A^{n+1}_{i,j+1,k}, && v_7 = A^n_{i,j+1,k}, && v_8 = A^{n-1}_{i,j+1,k}, \\
&v_9 = A^{n+1}_{i,j,k+1}, && v_{10} = A^n_{i,j,k+1}, && v_{11} = A^{n-1}_{i,j,k+1}, \\
&v_{12} = A^{n+1}_{i+1,j+1,k}, && v_{13} = A^n_{i+1,j+1,k}, && v_{14} = A^{n-1}_{i+1,j+1,k}, \\
&v_{15} = A^{n+1}_{i+1,j,k+1}, && v_{16} = A^n_{i+1,j,k+1}, && v_{19} = A^{n-1}_{i+1,j,k+1}, \\
&v_{18} = A^{n+1}_{i,j+1,k+1}, && v_{19} = A^n_{i,j+1,k+1}, && v_{20} = A^{n-1}_{i,j+1,k+1}, \\
&v_{21} = A^{n+1}_{i+1,j+1,k+1}, && v_{22} = A^n_{i+1,j+1,k+1}, && v_{23} = A^{n-1}_{i+1,j+1,k+1},
\end{aligned}
$$

3.1.5 High frequency problems

It should be noted that for the one-dimensional case there exists no evolution like Equation 2.15 (second order), since no transverse directions exist. They are therefore equivalent. The same argument can be made for axis-aligned plane waves, for which transverse derivatives also completely vanish. To analyze the absorption behaviour for this ideal case we can build an evolution matrix $\mathbb{U}$ for the one-dimensional wave equation with Mur's absorbing boundary conditions, using the discretization from Equation 3.14, as found in [21, 24]. This evolution operator acts on a wave equation state in the form

$$
S^n_W = \begin{bmatrix} A^n_0 \\ \vdots \\ A^n_{n_x-1} \\ A^{n-1}_0 \\ \vdots \\ A^{n-1}_{n_x-1} \end{bmatrix},
$$

and is defined in a way such that

$$
S^{n+1}_W = \mathbb{U} S^n_W.
$$

Note that therefore $\mathbb{U} \in \mathbb{R}^{2n \times 2n}$, and can be built as follows

$$
\mathbb{U}_{ij} = \left[\begin{array}{cc} \multicolumn{2}{c}{\mathrm{U}} \\ \hline \mathbb{I}_n & 0 \end{array}\right].
$$

The lower part just shifts the A^n values to A^{n-1}, and $\mathrm{U} \in \mathbb{R}^{n \times 2n}$ is is most easily described as

$$\forall(i \geq 1 \text{ and } i < n-1) \quad \begin{cases} \mathrm{U}_{i,i+n} = -1 \\ \mathrm{U}_{i,i} \quad = \alpha_1 \\ \mathrm{U}_{i,i+1} = \alpha_2 \\ \mathrm{U}_{i,i-1} = \alpha_2 \end{cases},$$

$$\begin{aligned} \mathrm{U}_{0,n} &= \beta_0, \\ \mathrm{U}_{0,0} &= \beta_1, \\ \mathrm{U}_{0,1+n} &= \beta_2, \\ \mathrm{U}_{0,1} &= \beta_1, \\ \mathrm{U}_{0,:} &+= \mathrm{U}_{1,:} \cdot \beta_0, \\ \mathrm{U}_{n,0} &= 1, \\ \mathrm{U}_{n-1,2n-1} &= \beta_0, \\ \mathrm{U}_{n-1,n-1} &= \beta_1, \\ \mathrm{U}_{n-1,2n-2} &= \beta_2, \\ \mathrm{U}_{n-1,n-2} &= \beta_1, \\ \mathrm{U}_{n-1,:} &+= \mathrm{U}_{n-2,:} \cdot \beta_0, \\ \mathrm{U}_{2n-1,n-1} &= 1. \end{aligned}$$

$$\text{otherwise, } \mathrm{U}_{ij} = 0 \text{ if none of the conditions above apply}$$

The weights α_i and β_i are obtained from Equation 3.3 and Equation 3.17.

We observe

- $\mathbb{U}$ does not have any real eigenvalues nor eigenvectors
- For all eigenvalues λ of $\mathbb{U}$, $|\lambda| < 1$ holds true. This is what we would expect; all initial conditions eventually converge towards zero.
- With increasing n, some eigenvalues rapidly converge towards one. This corresponds to a very low rate of absorption.

We define the "half-life" $\tilde{t}^n_{\frac{1}{2}}$ of such an update matrix as follows:

$$\tilde{t}^n_{\frac{1}{2}} = \frac{\log \frac{1}{2}}{\log|\lambda_{\max}|}, \tag{3.27}$$

$$t^n_{\frac{1}{2}} = \min_k \left(\|\mathbb{U}^k v\|_2 < \frac{\|v\|_2}{2} \right) \tag{3.28}$$

where $\lambda_{\max}$ is the biggest eigenvalue of $\mathbb{U}$ in absolute value. $t^n_{\frac{1}{2}}$ corresponds to the amount of timesteps required to halve every occurring mode in a grid with n points in absolute value. Much to our consternation and dismay, we notice that for increasing n, we get rapidly increasing values for $t^n_{\frac{1}{2}}$.

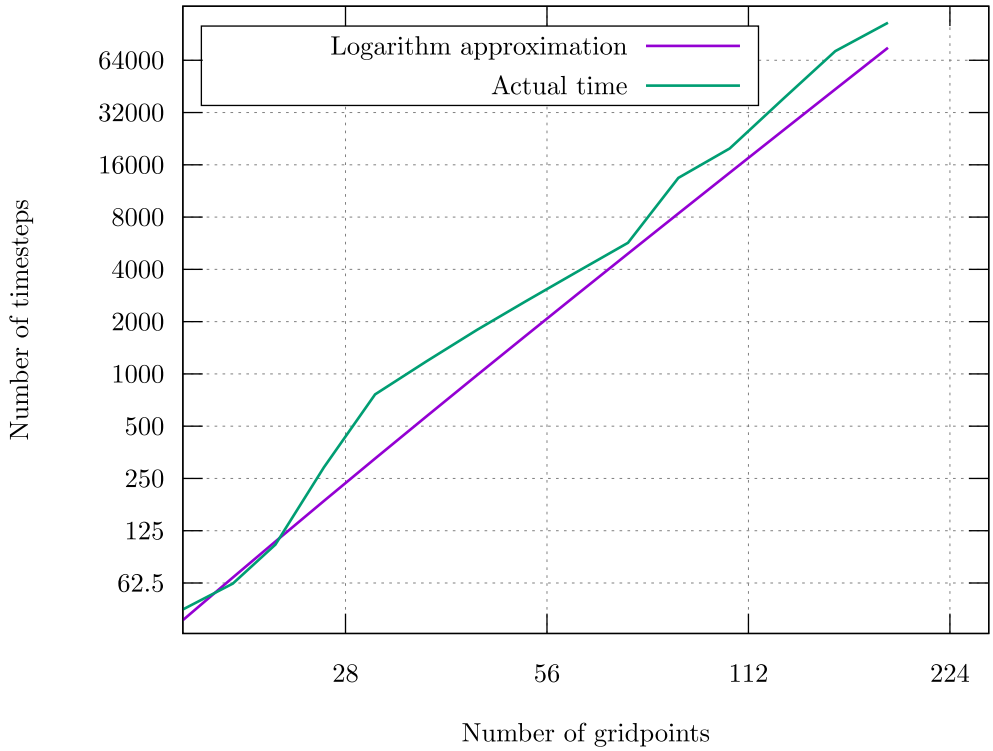


Figure 1: Relation of gridpoint count to half life of high modes

We see that the values of $\tilde{t}^n_{\frac{1}{2}}$, labelled **Logarithm approximation** and $t^n_{\frac{1}{2}}$, labelled **Actual time**, diverge cubically with respect to the gridpoints. This means that the finer our mesh is, the worse the extinction of high order modes actually gets. For this reason, it may be advisable to introduce a numerical diffusion term to get rid of these high-frequent modes.

3.2 Particle In Cell

The particle-in-cell (PIC) method is used to solve a specific set of partial differential equations by tracing individual particles (or fluid elements) within a Lagrangian framework across continuous phase space for parts that obey a transport equation. At the same time, distribution moments like densities and currents are stored at fixed Eulerian mesh points.

3.2.1 Boris Pusher

In order to discretize the particle motion described in Section 2.3.1, it is best to track both the particle position $\boldsymbol{r}$, and it's **normalized momentum** $\gamma\boldsymbol{\beta}$. The equations of motion for a particle with charge q are

$$\frac{\mathrm{d}\gamma\boldsymbol{\beta}}{\mathrm{d}t} = q(\mathbf{E} + \boldsymbol{v} \times \mathbf{B})$$
$$\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}t} = \frac{\gamma\boldsymbol{\beta}}{\sqrt{1 + \|\gamma\beta\|^2}}$$

Note that the normalized momentum is the ratio between relativistic momentum $\boldsymbol{p}$ and cm_0:

$$\boldsymbol{p} = m_0\gamma\boldsymbol{\beta}c = m_0\gamma\boldsymbol{v}$$

We use the method presented in [25] called Boris Pusher to discretize time:

$$\begin{aligned} t_1 &= \gamma\beta^{m-\frac{1}{2}} + \frac{q\Delta t\mathbf{E}_t^m}{2mc} \\ \alpha &= \frac{q\Delta t}{2m\sqrt{1 + \|t_1\|^2}} \\ t_2 &= t_1 + \alpha t_1 \times \mathbf{B}_t^m \\ t_3 &= t_1 + t_2 \times \frac{2\alpha\mathbf{B}_t^m}{1 + \alpha^2\ \|\mathbf{B}_t^m\|^2} \\ \gamma\beta^{m+\frac{1}{2}} &= t_3 + \frac{q\Delta t\mathbf{E}_t^m}{2mc} \\ r^{m+1} &= r^m + c\Delta t\frac{\gamma\beta^{m+\frac{1}{2}}}{\sqrt{1 + \|\gamma\beta^{m+\frac{1}{2}}\|^2}} \end{aligned} \tag{3.29}$$

Equation 3.29 assumes that electrons carry negative charge. We refer to [26] for a thorough explanation of why this scheme performs so well.

3.2.2 Quantity Interpolation

In order for the charge-carrying particles and the electromagnetic field to interact, the quantities carried by particles must be interpolated or "deposited" to the grid in order for the grid update functions to obtain the correct source terms. In our case this is the particle charge q and the current length $q\boldsymbol{v}$.

Charge deposition:

The charge is interpolated to the grid using the Cloud-in-Cell interpolation [27]. For gridpoints positioned at

$$\mathbf{A}_{i,j,k} \quad \text{at} \quad \begin{bmatrix} i\Delta x \\ j\Delta y \\ k\Delta z \end{bmatrix}$$

the charge q of a particle located at

$$\boldsymbol{r} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix}$$

is interpolated onto the grid as follows:

$$\begin{aligned}\begin{bmatrix} i_{\text{int}} \\ j_{\text{int}} \\ k_{\text{int}} \end{bmatrix} &= \begin{bmatrix} \left\lfloor \frac{r_x}{\Delta x} \right\rfloor \\ \left\lfloor \frac{r_y}{\Delta y} \right\rfloor \\ \left\lfloor \frac{r_z}{\Delta z} \right\rfloor \end{bmatrix} \\ \begin{bmatrix} \delta x \\ \delta y \\ \delta z \end{bmatrix} &= \begin{bmatrix} \frac{r_x}{\Delta x} - i_{\text{int}} \\ \frac{r_y}{\Delta y} - j_{\text{int}} \\ \frac{r_z}{\Delta z} - k_{\text{int}} \end{bmatrix}\end{aligned} \tag{3.30}$$

The contributions to the eight closest gridpoints are given by

$$\begin{aligned}\rho_{i+I,j+J,k+K} &= \frac{1}{\Delta x \Delta y \Delta z} q W_I^x W_J^y W_K^z \\ \text{where } W_L^a &= \begin{cases} 1 - \frac{\delta a}{\Delta a} & \text{if } L = 0 \\ \frac{\delta a}{\Delta a} & \text{if } L = 1 \end{cases}\end{aligned} \tag{3.31}$$

$$\text{with } (I, J, K) \in (0, 1)^3$$

A branchless formulation can be found in [21]:

$$\rho_{i+I,j+J,k+K} = \frac{1}{\Delta x \Delta y \Delta z} \left(\frac{1}{2} + (-1)^I \left| \frac{1}{2} - \frac{\delta x}{\Delta x} \right| \right) \left(\frac{1}{2} + (-1)^J \left| \frac{1}{2} - \frac{\delta y}{\Delta y} \right| \right) \left(\frac{1}{2} + (-1)^K \left| \frac{1}{2} - \frac{\delta z}{\Delta z} \right| \right) \tag{3.32}$$

Branchless implementations of these formulae are especially of concern in the realm of GPU computing, outlined in Section 5.4.

3.2.3 Current deposition

At first, current deposition seems like particle-to-grid interpolation as any other: Every particle carries a velocity and a charge that must be interpolated to the eight nearest values of $\mathbf{J}^\alpha$. We can define the current-length of a particle

$$\boldsymbol{\mathcal{J}}_p = q_p \boldsymbol{v}_p$$

which would have SI-Units $\frac{\text{C}}{\text{s}} \cdot \text{m} = \text{A} \cdot \text{m}$. We interpolate current lengths onto the grid according to Equation 3.31, multiplying it by a factor of $\frac{1}{\Delta x \Delta y \Delta z}$. This will yield the correctly scaled desired contributions to $\mathbf{J}$ with units $\frac{\text{A}}{\text{m}^2}$.

3.2.4 Charge-conserving current deposition schemes

In order not to violate the charge conservation law

$$\frac{\partial \mathbf{J}^{\alpha}}{\partial x^{\alpha}} = 0,$$

[28] came up with a specific scheme to interpolate currents to split the trajectory a particle travels in one timestep into multiple parts. We refer to the trajectory $\{\boldsymbol{r}_{n-1}, \boldsymbol{r}_n\}$ as "line". The reason this is necessary in the first place is because a trajectory can span multiple cells, therefore affecting more than eight grid points. However, splitting the line into parts that only span one cell in the manner that [28] causes a performance hit. The main reason named for its cost are the many unpredictable branches. One approach to improve the computational performance of this line-splitting approach is presented in [29, 30, 21], called "ZigZag". The method can be detailed as follows:

1. Given a line $\{\boldsymbol{r}_{n-1}, \boldsymbol{r}_n\}$, compute its "Relay point" as follows:

$$\boldsymbol{r}_{\text{relay}} = \begin{bmatrix} x_{\text{relay}} \\ y_{\text{relay}} \\ z_{\text{relay}} \end{bmatrix} = \begin{bmatrix} \min\Big(\min(i_{n-1}\Delta x, i_n \Delta x) + \Delta x, \max\Big(\max(i_{n-1}\Delta x, i_n \Delta x), \frac{r_{n-1,x}+r_{n,x}}{2}\Big)\Big) \\ \min\Big(\min(j_{n-1}\Delta y, j_n \Delta y) + \Delta y, \max\Big(\max(j_{n-1}\Delta y, j_n \Delta y), \frac{r_{n-1,y}+r_{n,y}}{2}\Big)\Big) \\ \min\Big(\min(k_{n-1}\Delta z, k_n \Delta z) + \Delta z, \max\Big(\max(k_{n-1}\Delta z, k_n \Delta z), \frac{r_{n-1,z}+r_{n,z}}{2}\Big)\Big) \end{bmatrix}$$

2. Interpolate the two lines $\{\boldsymbol{r}_{n-1}, \boldsymbol{r}_{\text{relay}}\}$ and $\{\boldsymbol{r}_{\text{relay}}, \boldsymbol{r}_n\}$ naively by using weights obtained with Equation 3.31 and plugging in the midpoints $\frac{\boldsymbol{r}_{n-1}+\boldsymbol{r}_{\text{relay}}}{2}$ and $\frac{\boldsymbol{r}_n+\boldsymbol{r}_{\text{relay}}}{2}$ respectively.

Grid to Particle:

Grid to particle interpolation can be done analogously. We also use Equation 3.30 to obtain grid position and obtain subsequent interpolation weights with Equation 3.31. We can write down a summation formula:

$$\psi_p = \sum_{I,J,K \in (0,1)^3} \psi_{i+I,j+J,k+K} \left(\frac{1}{2} + (-1)^I \left|\frac{1}{2} - \frac{\delta x}{\Delta x}\right|\right) \left(\frac{1}{2} + (-1)^J \left|\frac{1}{2} - \frac{\delta y}{\Delta y}\right|\right) \left(\frac{1}{2} + (-1)^K \left|\frac{1}{2} - \frac{\delta z}{\Delta z}\right|\right) \quad (3.33)$$

where ψ_r denotes the interpolated value of a quantity ψ at point $\boldsymbol{r}$, and $\psi_{i,j,k}$ denotes the value of quantity ψ on the respective gridpoint.

Chapter 4

Free Electron Laser

4.1 Input Parameters

All input parameters are in laboratory coordinates. The particles are initialized in laboratory coordinated and then transformed into the co-moving **bunch coordinates** that move with a velocity of β_0, where $\beta_0 = \frac{\sqrt{\gamma_0^2 - 1}}{\gamma_0}$, is introduced determined with the Undulator Parameter).

4.1.1 Undulator

An undulator is an apparatus consisting of $2N$ consecutive magnets arranged in opposing direction ($2N$ because one period requires **two** opposing magnets), each magnet block. The electromagnetic field inside an undulator can be describe with two parameters: The Undulator period denoted with λ_u and the Undulator parameter, denoted K. The magnetic field strength B_0 results from the definition of K, the Undulator Parameter. m_e and e denote electron mass and charge, respectively.

$$B_0 = \mathrm{K}\ \frac{2\pi m_e c}{e\lambda_u}$$

Undulator Period: The undulator is defined by twice the length of opposing magnets, or simply the wavelength of the magnetic field inside the undulator, seen in Equation 4.3.
Radiation Period: The radiation period of an undulator-bunch setup is defined as

$$\lambda_r = \frac{\lambda_u}{2\gamma_0^2}.$$

Undulator Parameter: A dimensionless parameter describing how much an undulator deflect a particle from a straight trajectory can be defined as

$$\mathrm{K} = \frac{eB_{\mathrm{mag}}\lambda_u}{2\pi m_e c}. \tag{4.1}$$

When a bunch with initial mean Lorentz factor γ enters an undulator with undulator parameter K, its velocity along the the longitudinal component is reduced. A widely-known formula to approximate this slowdown is derived in [21, 31, 32]:

$$\gamma_0 \approx \frac{\gamma}{\sqrt{1 + \frac{\mathrm{K}^2}{2}}} \tag{4.2}$$

$$\beta_0 = \frac{\sqrt{\gamma_0^2 - 1}}{\gamma_0}$$

with γ being the initial Lorentz factor of the bunch, K the undulator parameter and γ_0 the slowed Lorentz factor. γ_0 will be the Lorentz factor of our co-moving frame. Note that the Equation 4.2 only is an approximation. While it is cumbersome to derive an exact formula, we can simulate a reference particle trajectory purely in the laboratory frame and simply measure the average velocity.

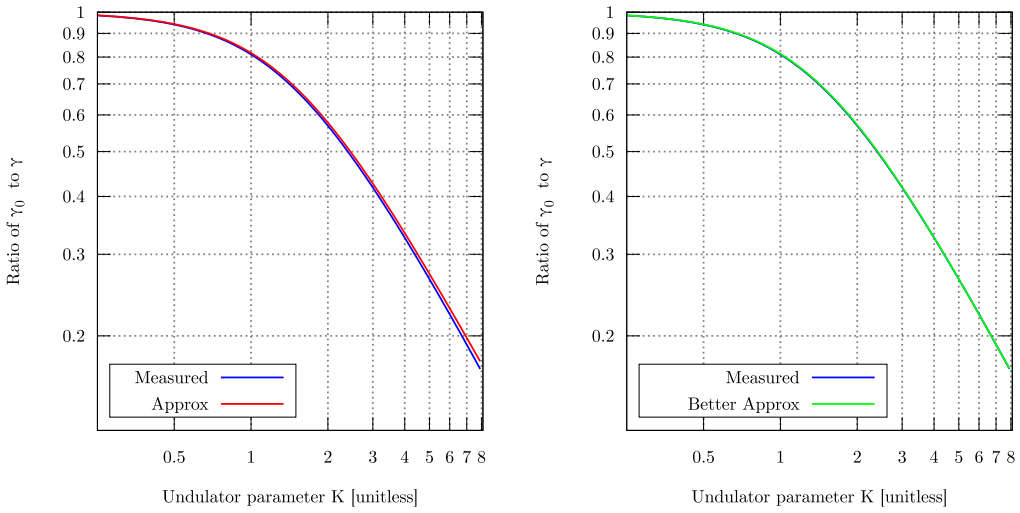


Figure 2: Exact and approximate relations of $\frac{\gamma_0}{\gamma}$

Figure 2 shows the exact relation between γ_0 and γ. For large values of K, the discrepancy becomes bigger. The right part of Figure 2 shows a comparison of the exact $\frac{\gamma_0}{\gamma}$ with a better approximation:

$$\text{Better approx:} \qquad \gamma_0 \approx \frac{\gamma}{\sqrt{1 + \alpha_1 \mathrm{K}^2 + \alpha_2 \mathrm{K}^3}}$$

$$\alpha_1 = 0.51$$
$$\alpha_2 = 0.00375$$

The coefficients α_1 and α_2 were found through numerical bisection.
Undulator field: Let

$$\boldsymbol{r}_{\text{lab}} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

denote a position in the laboratory frame. The magnetic field of a static undulator is defined as

$$\mathbf{B}_{\text{u,lab}}(\boldsymbol{r}) = \begin{bmatrix} 0 \\ B_0 \cosh\big(k_u r_y\big) \sin(k_u r_z) \\ B_0 \sinh\big(k_u r_y\big) \cos(k_u r_z) \end{bmatrix} \tag{4.3}$$

Where

$$k_u = \frac{2\pi}{\lambda_u}$$

4.1.2 Bunch

We call set of particles, each consisting of charge, mass, position and normalized momentum a bunch.

Phase space: To represent the full state of a charged particle, [33] introduces the notion of *phase pace*, defined as the concatenation of the position and the normalized momentum vector:

$$\boldsymbol{\rho} = \begin{bmatrix} \boldsymbol{r} \\ \gamma\boldsymbol{\beta} \end{bmatrix} = \begin{bmatrix} r_x \\ r_y \\ r_z \\ \gamma\beta_x \\ \gamma\beta_y \\ \gamma\beta_z \end{bmatrix} \in \mathbb{R}^6$$

Charge Distribution: The macrostate of a bunch can be represented as a Gaussian distribution with average position-momentum vector $\overline{\boldsymbol{\rho}} = \mathbf{E}(\boldsymbol{\rho})$ and its covariance matrix $\boldsymbol{\Sigma}(\boldsymbol{\rho})$, where

$$\begin{aligned} \boldsymbol{\Sigma} &\in \mathbb{R}^{6\times 6} \\ \boldsymbol{\Sigma}_{ij} &= \mathbf{E}\Big[\big(\boldsymbol{\rho}_i - \overline{\boldsymbol{\rho}}_i\big)\big(\boldsymbol{\rho}_j - \overline{\boldsymbol{\rho}}_j\big)\Big] \end{aligned}$$

Two frequently occurring setups of the bunch are:

- Initializing all six components $\boldsymbol{\rho}$ with a Gaussian distribution, fully characterizable by $\overline{\boldsymbol{\rho}}$ and $\boldsymbol{\Sigma}$
- Initializing all components but the longitudinal position r_z of $\boldsymbol{\rho}$ with a Gaussian distribution, fully characterizable by $\overline{\boldsymbol{\rho}}$ and $\boldsymbol{\Sigma}$. Then interpret $\sqrt{\boldsymbol{\Sigma}_{33}}$, which would be

the standard deviation of r_z, as the interval width of a *uniform* distribution, and therefore sample r_z in $\left[\overline{\boldsymbol{r}}_2 - \frac{1}{2}\sqrt{\boldsymbol{\Sigma}_{33}}, \quad \overline{\boldsymbol{r}}_2 + \frac{1}{2}\sqrt{\boldsymbol{\Sigma}_{33}}\right]$

Bunching Factor: The bunching factor is defined as the average phase of the electrons with respect to the period of the undulator field from Equation 4.3:

$$\alpha_b = \frac{\sum_{\text{all particles}} \left[\exp\left(\frac{2\pi \boldsymbol{i} r_{z,\text{bunch}}}{\lambda_{u,\text{bunch}}} \right) \right]}{N_{\text{particles}}}, \tag{4.4}$$

where the bunch subscripts imply the evaluation of those quantities in bunch space:

$$\lambda_{u,\text{bunch}} = \frac{\lambda_u}{\gamma_0}.$$

The transformation of particle coordinates is outlined in Section 4.2.1. If we simply initialize the particles with random positions, the bunching factor tends to zero only for an infinite particle count. For any finite number of particles, the expectation value of α_b is nonzero. Since the bunching factor is an important trait of a bunch, it is of great interest to lock α_b to a given value at the start of our simulation. We achieve this by first initializing the particles such that $\alpha_b = 0$. This is achieved by quadrupling every particle, inserting four particles at the phase angles $\left[0, \frac{\pi}{2}, \pi, \frac{3}{2}\pi\right]$ with respect to the undulator period $\lambda_{u,\text{bunch}}$. However, since we are initializing particles in the laboratory frame, the modification formula for $r_{z,\text{lab}}$ looks different:

$$\begin{aligned} r_{z,i} &:= r_z - \lambda_r \frac{i}{4} \quad \text{for i} \in [0,3] \\ \text{where } \lambda_r &= \frac{\lambda_u}{2\gamma_0^2} \end{aligned} \tag{4.5}$$

Inserting four different particles with longitudinal position $r_{z,i}$ as described in Equation 4.5 will cause α_b to be zero. However, we usually want to set it to a small initial value, for example to 0.01 as later outlined in Table 4. We perform another modification to r_z:

$$r_z := r_z - \left(\frac{\lambda_r}{\pi} \alpha_{b,\text{init}} \right) \sin\left(\frac{2\pi r_z}{\lambda_r} \right) \quad \text{for i} \in [0,3] \tag{4.6}$$

Now with Equation 4.6, the bunching factor α_b will be exactly equal to the the desired value $\alpha_{b,\text{init}}$.

4.1.3 Mesh

The mesh simply is scaled by the inverse of a length contraction, having each of its spacings multiplied:

$$\begin{bmatrix}\Delta x\\ \Delta y\\ \Delta z\end{bmatrix}_{\text{bunch}} = \mathbb{L}_3 \begin{bmatrix}\Delta x\\ \Delta y\\ \Delta z\end{bmatrix}_{\text{lab}}$$

where $\mathbb{L}_3$ denotes the lower right 3×3 block of the Lorentz transform matrix L, described in Section 4.2.

4.2 Lorentz Transform

We perform the simulation in a boosted frame, moving with relative velocity β_0 with respect to the (rest) laboratory frame. As seen in Undulator Parameter, β_0 is defined as follows

$\boldsymbol{r}_{\text{lab}}$ Moving with $\beta = \mathbf{0}$	$\mathbb{L} \cdot \boldsymbol{r}_{\text{lab}} \longrightarrow$ $\longleftarrow \mathbb{L}^{-1} \cdot \boldsymbol{r}_{\text{bunch}}$	$\boldsymbol{r}_{\text{bunch}}$ Moving with $\beta = \beta_0$

4.2.1 Transforming the bunch

A particle with laboratory frame position

$$\boldsymbol{r}_{\text{lab}} = \begin{bmatrix}t\\ x\\ y\\ t\end{bmatrix}$$

is transformed into

$$\begin{bmatrix}t\\ x\\ y\\ z\end{bmatrix}_{\text{bunch}} = \mathbb{L} \begin{bmatrix}t\\ x\\ y\\ z\end{bmatrix}_{\text{lab}} \tag{4.7}$$

$$\text{where } \mathbb{L} = \begin{bmatrix} \gamma & -\gamma\beta_x & -\gamma\beta_y & -\gamma\beta_x \\ -\gamma\beta_x & 1+\frac{(\gamma-1)\beta_x^2}{\|\beta\|^2} & \frac{(\gamma-1)\beta_x\beta_y}{\|\beta\|^2} & \frac{(\gamma-1)\beta_x\beta_z}{\|\beta\|^2} \\ -\gamma\beta_y & \frac{(\gamma-1)\beta_x\beta_y}{\|\beta\|^2} & 1+\frac{(\gamma-1)\beta_y^2}{\|\beta\|^2} & \frac{(\gamma-1)\beta_y\beta_z}{\|\beta\|^2} \\ -\gamma\beta_z & \frac{(\gamma-1)\beta_x\beta_z}{\|\beta\|^2} & \frac{(\gamma-1)\beta_y\beta_z}{\|\beta\|^2} & 1+\frac{(\gamma-1)\beta_z^2}{\|\beta\|^2} \end{bmatrix} \tag{4.8}$$

with $\beta = [\beta_x \ \ \beta_y \ \ \beta_z]^T$ denoting the relative velocity of the bunch frame with respect to the Laboratory frame. In our case, only $\gamma\beta_z$ is nonzero. The velocity is transformed as follows [34]:

$$\mathbf{v}_{\text{bunch}} = \frac{1}{1 - \dfrac{\boldsymbol{\beta} \cdot \mathbf{v}_{\text{lab}}}{c}} \left[\frac{\mathbf{v}_{\text{lab}}}{\gamma} - c\boldsymbol{\beta} + \frac{\gamma}{1+\gamma} (\mathbf{v}_{\text{lab}} \cdot \boldsymbol{\beta}) \boldsymbol{\beta} \right] \tag{4.9}$$

Usually the transformation of the particle β is more useful:

$$\boldsymbol{\beta}_{\text{bunch}} = \frac{1}{1 - (\boldsymbol{\beta} \cdot \boldsymbol{\beta}_{\text{lab}})} \left[\frac{\boldsymbol{\beta}_{\text{lab}}}{\gamma} - \boldsymbol{\beta} + \frac{\gamma}{1+\gamma} (\boldsymbol{\beta}_{\text{lab}} \cdot \boldsymbol{\beta}) \boldsymbol{\beta} \right] \tag{4.10}$$

where $\boldsymbol{\beta}$ denotes the velocity of the moving frame, $\boldsymbol{\beta}_{\text{lab}}$ denotes the velocity of an object in Laboratory coordinates.

Correcting the time: When transforming the particles from laboratory to bunch coordinates, particles not located on the $z = 0$ plane will end up with a nonzero time coordinate. This must be corrected to capture all particles at the same time. This cannot be done analytically and exactly, so we assume the particles travel only in straight lines. Therefore we simply perform these operations on the particles in bunch coordinates and reassign the spacetime coordinates as follows:

$$\begin{aligned} \boldsymbol{r}_{xyz} &:= \boldsymbol{r}_{xyz} - \boldsymbol{\beta} r_t \\ r_t &:= 0 \end{aligned}$$

4.2.2 Laboratory-Bunch interactions

In order to compute effect of the external undulator field on a particle in bunch coordinates, one needs to transform the particle to laboratory coordinates, evaluate the undulator field at that point in spacetime and transform the electromagnetic field back into bunch coordinates.

$$\begin{aligned} \boldsymbol{r}_{\text{lab}} &= \mathbb{L}^{-1} \cdot \boldsymbol{r}_{\text{bunch}} \\ \mathbf{E}_{\text{lab}} &= 0, \qquad \mathbf{B}_{\text{lab}} = \mathbf{B}_u(\boldsymbol{r}_{\text{lab}}) \end{aligned}$$

and then transform $\mathbf{E}_{\text{lab}}$ and $\mathbf{B}_{\text{lab}}$ to bunch coordinates using Equation 2.22.

Chapter 5

Implementation

5.1 Programming model

The implementation of our solver is written in pure C++. There exist several tools [35; 37] to compile C++ to different hardware backends. At the core of our framework are functions that take our program state and an index as an argument and perform an update. If a set of function executions is completely independent of each other, it can be executed with arbitrary parallelism. We want to at very least guarantee that invocations with different indices are independent.

Such functions are easy to model with Lambda Expressions [38].

Naive implementation	`std::vector<T> a, b;`
	`for(int i = 1;i < n - 1;i++){`
	`  b[i] = a[i - 1] + a[i + 1];`
	`}`

OpenMP	`std::vector<T> a, b;`
	
	`#pragma omp parallel for`
	`for(int i = 1;i < n - 1;i++){`
	`  b[i] = a[i - 1] + a[i + 1];`
	`}`

Thrust	`thrust::device_vector<T> a, b;`
	
	`thrust::for_each(`
	`  thrust::make_counting_iterator(  1),`
	`  thrust::make_counting_iterator(n - 1),`
	`  [a, b]__host__ __device__(int i){`
	`  b[i] = a[i - 1] + a[i + 1];`
	`  }`
	`)`

Kokkos	`Kokkos::View<T*> a, b;`
	
	`Kokkos::parallel_for(`
	`  Kokkos::RangePolicy<Kokkos::DefaultExecutionSpace>(1, n - 1),`
	`  [a, b]KOKKOS_LAMBDA(int i){`
	`  b[i] = a[i - 1] + a[i + 1];`
	`  }`
	`)`

Table 2: Loop abstraction synopsis

5.2 Threads

One way to run function invocations in parallel is to split the workload into N parts and launch a thread to handle every part. The key parts of threads are:

- Threads spawned by the same process can all access the same memory through the same virtual addresses.
- The ratio of active threads to logical CPU cores is at most 1. The only way for a core to run more than one thread is to rapidly switch back and forth. See Section 5.4 for a juxtaposition to Cuda.
- Every thread is it's own execution context. The issued instructions are completely individual to every thread. Synchronization between threads must be done explicitly.

Additionally, threads are extremely expensive to launch. Starting a thread for less than a couple of thousands of computations will lead to a decrease in performance. One possibility to outsorce the challenge of optimal thread spawning and scheduling is OpenMP [39], that has the capability to efficiently divide the work and schedule normal C/C++ for-loops.

5.3 Ranks

The feasibility of utilizing threads hinges on the underlying hardware's capability to offer homogeneous memory. Homogeneous memory is already challenging on task on single processor machines, as a lot of care has to be taken to guarantee cache coherency [40, 41]. As soon as tasks begin to span several separate processor and memory units, uniform memory access becomes infeasible. Apart from a few kernel-specific features such as Linux's `mmap` [42], the address spaces of different processes are completely separated, implying that all communication between processes has to be done explicitly. A standardized way of launching a group of processes is MPI [43]. A group of N processes of the same program is called a "World", whereas the i-th launched process is called the i-th "Rank".

The consequence of separate memory spaces is the need to subdivide the simulation domain into N disjoint subdomains that cover the entire simulation domain. Cells lying on the boundary between subdomains or particles travelling across need to be detected and sent to adjacent ranks explicitly. For a detailed explanation of cell and particle communication as well as their implementation, the reader is referred to [13]. We use the implementation presented in [13] for our multi-rank implementation.

5.4 Cuda

Cuda [35, 44, 45] and its AMD equivalent Rocm [37] are C++-based programming toolkits that allow GPU programming in a manner very similar to threads. However threads launched by the Cuda runtime can call functions compiled and assembled for the GPU. We refer to code executed by Cuda threads as **Device code**, and to code executed by regular threads as **Host code**. Furthermore, the memory spaces are also split between host and device; Pointers in general (with the exception of *Unified Memory* [46]) are only accessible from either host or device code, also implying the need for explicit communication.

5.4.1 Instructional Throughput

Unlike CPU threads, Cuda threads are not able to execute independently from each other. Cuda threads run in groups of 32 (see [45]) called "Warps", and every Warp belongs to a "Block", consisting of 1-32 warps. A thread block runs on one gpu processing core, called "Streaming Multiprocessors" and frequently abbreviated SM.

- The ratio of threads to SM can range up to 2048 depending on hardware (see the device attribute cudaDevAttrMaxThreadsPerMultiProcessor).
- All threads that are part of one warp execute in lockstep; only one instruction can be executed on all warps. The runtime of a warp is determined by the sum of all of the different branches taken. inside that warp.
- A block, consisting of one or more warps, can run on only one SM. One SM however can run up to 16 blocks simultaneously (see the device attribute cudaDevAttrMaxBlocksPerMultiprocessor). It is therefore advisable that the ratio $\frac{\text{ThreadsPerMultiProcessor}}{\text{ThreadsPerBlock}}$ evaluates to an integer, otherwise SMs cannot be used to their full extents.
- The builtin function `__syncthreads()` causes all warps within one block to wait for eachother. Inside every warp, all threads are completely synchronized all the time. Furthermore, all threads in a threadblock have access to a common `__shared__` memory block that has about cudaDevAttrMaxSharedPerMultiProcessor available for each SM.

5.4.2 Memory Types

There are four different types of memory available to code running on Cuda devices.

Constant	Global	Shared	Register
Fast Read-only $\approx$ 64 KiB	Slow & High Latency Read-Write Several GiB	Extremely fast Read-Write $\approx \frac{32\ \text{KiB}}{\text{Block}}$	Instantaneous Read-Write Size limited

Table 3: Cuda memory types synopsis

5.5 Performance

There exists an abundance of literature on speeding up finite-differences schemes on cuda [47; 51]. However, while [51] is concerned with 5-point stencils, [49] and [48] deal with 8-point stencils. This increases the reuse count of loaded values. Employing similar optimization techniques, such as explicitly prefetching the stencil support over a thread block into shared memory, we were unable to achieve a speedup over a naive stencil evaluation.

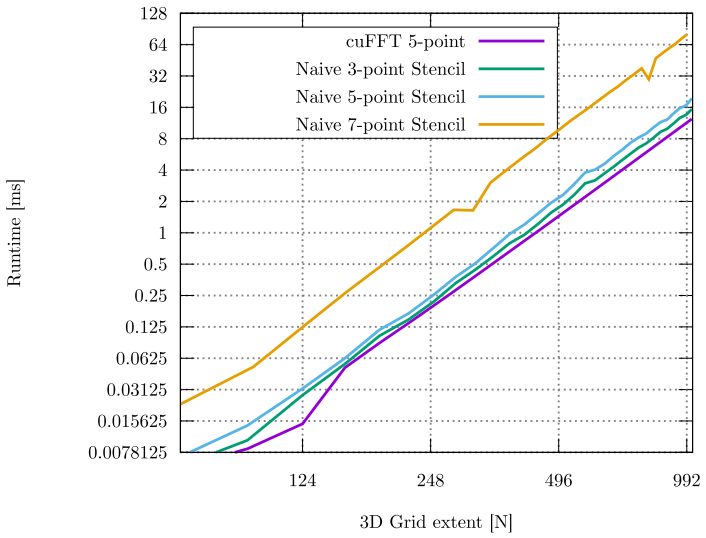


Figure 3: Stencil performance on a NVidia A100 card

It should be noted that evaluating stencils (which is in essence a convolution) in a naive might not be the most performant way on GPUs. Figure 3 shows a performance benefit to computing two FFTs with $(N+4)^3$ cells, allowing a convolution with a 5^3 stencil, over naively evaluating the simplest $[\ \ 1, -2, \ \ 1]$ stencil in each direction at every point.

5.5.1 Memory Locality

The cloud-in-cell interpolation schemes described by the Equation 3.31 and Equation 3.33 are highly nonlocal since the particles can be arbitrarily scattered across the computational

domain. A good way to increase memory locality is to sort the particle with respect cell ordering they are in. The index mapping function of a cartesian mesh

$$\mathrm{I}(i, j, k) : \mathbb{Z}^3 \to \mathbb{Z}^+$$

is evaluated for the smallest scatter index for a given particle, according to which the particles are then sorted. If the particles move in a highly correlated manner, this sorting only has to be done every N timesteps, where N can be a very large number, since particles that are close with respect to grid index tend stay close.

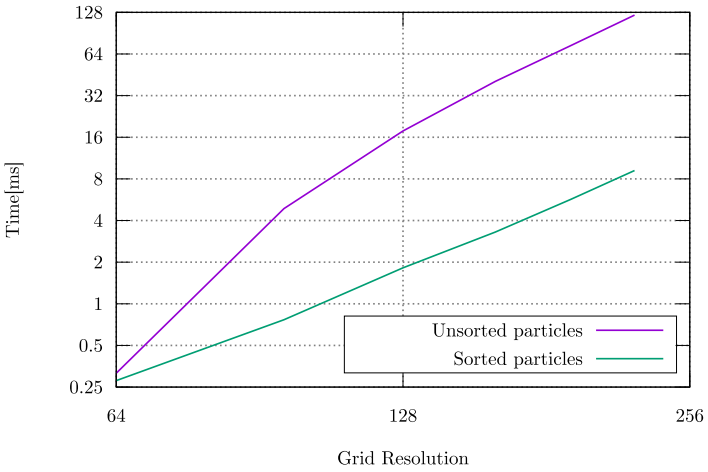


Figure 4: Time required for a scatter operation

Figure 4 shows the quantitative speedup gained by sorting the particles with respect to index before interpolating them according to Equation 3.31. While for small grid sizes, cache effects are not dominating the runtime, we obtain a speedup of roughly factor 15 for grids of 256^3 cells. The amount of particles was picked to be $4N^3$, where N denotes the grid resolution. The benchmark was performed on an NVIDIA RTX 3070 using 32-bit floats.

5.5.2 Memory footprint

The numerical update equation Equation 3.2 and Equation 3.9 involve the state of $\mathbf{A}$ at three different timesteps: $n-1, n$ and $n+1$. However, it is clear that there is only one dependency on the state at timestep $n-1$, which is at position (i, j, k), so every thread is allowed to overwrite that value once the value for step $n+1$ has been computed. For the interior field update, it is therefore enough to allocate two arrays for $\mathbf{A}$. To perform a field update, we simply write the values for $\mathbf{A}^{n+1}$ over the values of $\mathbf{A}^{n-1}$ subsequently perform a swap of $\mathbf{A}^{n-1}$ and $\mathbf{A}^n$. However, as mentioned in Section 3.1.4, some discretizations of absorbing boundary conditions contain inputs of $\mathbf{A}^{n-1}$, $\mathbf{A}^n$ and $\mathbf{A}^{n+1}$.

5.6 Numerically Stable Reductions

Computing statistical quantities, such as mean position, momentum and their covariance matrix is a ubiquitous task for particle simulations, both involving an averaged sum over all particles. However, performing large sums using floating-point numbers carries some well-known problems. If the value $\alpha + \beta$ with $\alpha > \beta$ is computed using 32-bit floats, the lowest k

bits of β are lost where $l = \log_2 \alpha - \log_2 \beta$. In the extreme case where $\frac{\alpha}{\beta} > 2^{24}$, this results in the identity $\alpha + \beta = \alpha$, since all 23 bits of a 32-bit floating-point mantissa will be lost. This is known as roundoff error. To illustrate further, consider the averaging sum

$$\frac{1}{N}\sum_{i}^{N} \alpha_i, \alpha_i \in \text{ FP32}$$

We can implement this in a straightforward way:

```
1 float accum = 0;
2 for(auto it = alpha.begin(); it != alpha.end();it++){
3   accum += alpha[i];
4 }
5 accum /= alpha.size();
```

Even for a not particularly problematic case where all α_i are roughly equal in magnitude α_0, the value of `accum` increases to the same order of magnitude as $i\alpha_0$ while we keep adding values in the order of magnitude α_0, so every time a truncation happens. From a value for i of $\approx 2^{24}$ and onward, the addition does not have any effect anymore. In order to improve this we use an idea presented in [52]:

$$\overline{\alpha}_n = \overline{\alpha}_{n-1} + \frac{\alpha_n - \overline{\alpha}_{n-1}}{n} \tag{5.1}$$

The variance can be computed analogously:

$$\begin{aligned} M_n &= M_{n-1} + (\alpha_n - \overline{\alpha}_{n-1})(\alpha_n - \overline{\alpha}_n) \\ \sigma_n &= \frac{M_n}{n} \end{aligned} \tag{5.2}$$

However, Equation 5.1 and Equation 5.2 require to be evaluated sequentially, which highly undesirable. We can however generalize it to a *transform-reduce*, which is supported by all four surveyed librares from Table 2. We define a "Sample" to be a tuple:

$$\begin{bmatrix} \overline{x} \\ \sigma^2 \\ w \end{bmatrix}$$

where $w \in \mathbb{Z}$ describes the weight of that sample. We can define a combination operator $+$ for samples:

$$\begin{bmatrix} \overline{x}_1 \\ \sigma_1^2 \\ w_1 \end{bmatrix} + \begin{bmatrix} \overline{x}_2 \\ \sigma_2^2 \\ w_2 \end{bmatrix} \rightarrow \begin{bmatrix} \frac{w_1\overline{x}_1 + w_2\overline{x}_2}{w_1+w_2} \\ \left(\frac{\sigma_1^2 w_1 + \sigma_2^2 w_2}{w_1+w_2}\right) + \\ \left(\frac{(\overline{x}_1 - \overline{x}_{\text{combined}})(\overline{x}_1 - \overline{x}_{\text{combined}})^T w_1 + (\overline{x}_2 - \overline{x}_{\text{combined}})(\overline{x}_2 - \overline{x}_{\text{combined}})^T}{w_1+w_2}\right) \\ w_1 + w_2 \end{bmatrix} \tag{5.3}$$

where $\overline{x}_{\text{combined}} = \frac{w_1\overline{x}_1 + w_2\overline{x}_2}{w_1+w_2}$, which is also the mean element of the combined tuple. In C++, we can define a `Dataset` type as follows:

```cpp
template<typename T, unsigned N>
struct Dataset {
    Vector<T, N> mean;
    Matrix<T, N> covariance;
    uint64_t weight;

    //Constructor for one sample
    Dataset(Vector<T> value) : mean(value), covariance(0), weight(1){}

    //Perform weight-corrected combination of two sets
    Dataset operator+(const Dataset& other) const {
        size_t total_weight = weight + other.weight;
        Vector<T, N> combined_mean = (mean * weight + other.mean * other.weight) / total_weight;

        // Calculate mean deviations
        Vector<T, N> mean_dev1 = mean - combined_mean;
        Vector<T, N> mean_dev2 = other.mean - combined_mean;

        // Calculate outer products of mean deviations
        Matrix<T, N> outer_prod_mean_dev1 = outer_product(mean_dev1, mean_dev1);
        Matrix<T, N> outer_prod_mean_dev2 = outer_product(mean_dev2, mean_dev2);

        Matrix<T, N> combined_covariance =
        ((covariance * weight + other.covariance * other.weight) / total_weight) +
        ((outer_prod_mean_dev1 * weight + outer_prod_mean_dev2 * other.weight) / total_weight);

        return Dataset(combined_mean, combined_covariance, total_weight);
    }

};
```

Using the above `Dataset` type, computing average and covariance can be done by simply converting every `Vector` to `Dataset` and subsequently perform a reduction using the `+` operator. The function `outer_product` denotes the outer product of two column-vectors:

$$\text{outer\_product}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}\boldsymbol{y}^T$$

Chapter 6

Results

6.1 Source-free Plane Wave

A simple verification test we can perform is to solve a source free wave. After a certain amount of time we can check how much the predicted solution deviates from the analytical one. In order to separate the field solver from errors generated by boundary conditions, we use the simple periodic boundary conditions. As an initial condition we use a Gaussian pulse:

$$\mathbf{A}_0(\boldsymbol{r}) = \exp\left[-(\boldsymbol{r}\cdot\boldsymbol{d})^2\right],$$
$$\text{where } \boldsymbol{r} \in \left[-\frac{1}{2}, \frac{1}{2}\right]^3$$

and d is a normalized, axis-aligned direction vector. Since in Equation 3.9, we only correct for dispersion in the Z-direction, we expect our results to be different based on $\boldsymbol{d}$. Since the extents of the domain are exactly one and the boundary conditions are periodic, we expect the wave to return to its initial state after a duration of 1, irrespective of the units picked.

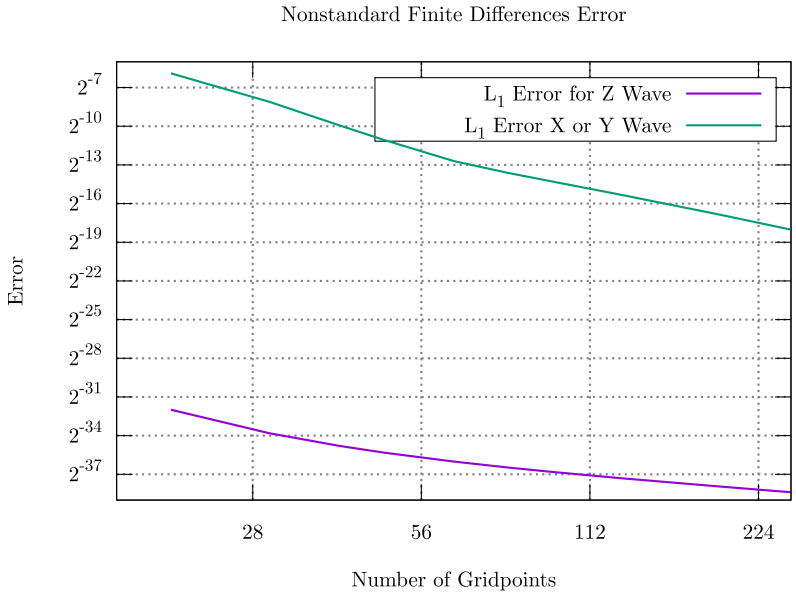


Figure 5: Convergence plot for Nonstandard Finite-Differences

Figure 5 shows more than sufficient convergence for $\boldsymbol{d}$ aligned with X- and Y-axes, and only roundoff error for $\boldsymbol{d}$ aligned with the Z-axis. Waves travelling purely in Z-direction are therefore evolved exactly up to rounding errors.

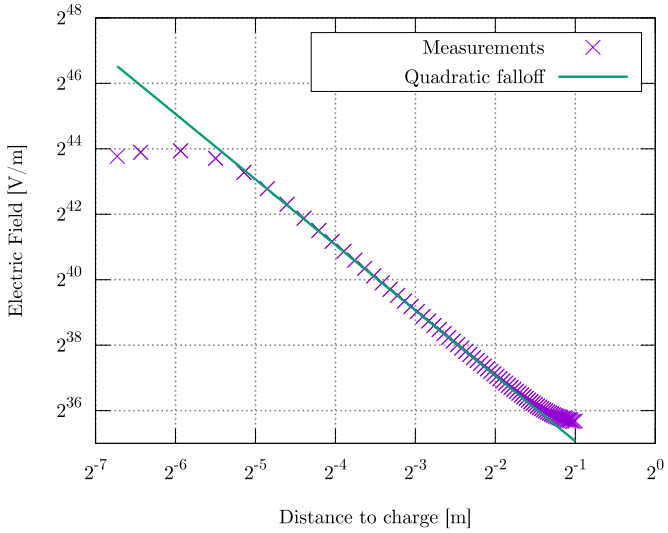
6.2 Source terms

6.2.1 Gauss's Law

To verify the satisfaction of Gauss's divergence law

$$\boldsymbol{\nabla} \cdot \mathbf{E} = \frac{\rho}{\varepsilon_0}$$

we place N particles of $1C$ in the center of the domain of $1\mathrm{m}^3$, distributed according to a normal distribution with $\sigma = 1\mathrm{cm}$ and wait for a duration of $\frac{8\mathrm{m}}{c}$. This a reasonable time to wait until an equilibrium has settled.



For Figure 6, the electric field was sampled at several points. Because the simulation uses scaled planck units it needed to be converted back to SI-Units. This proves quantitative correctness for our unit conversions and charge deposition.

Figure 6: Result of the Gauss Test Case

Figure 6 shows the expected quadratic falloff according to

$$\mathbf{E}_{r(r)} = \frac{1}{4\pi\varepsilon_0 r^2}$$

where $\mathbf{E}_r$ denotes the radial component of the electric field. However, for small r, the sampled $\mathbf{E}_r$ is inside the charge bunch, leading to a linear dependence on r. Also, when coming close to to the boundary at ± 0.5m the compute field is too big, which is due to the absorbing boundary conditions used. Equation 3.11 permits a stationary solution that is equivalent to a Neumann Boundary condition:

$$\frac{\partial \mathbf{A}^{\alpha}}{\partial \boldsymbol{n}} = 0$$

This does not correspond to an open boundary for the electromagnetic fields $\mathbf{E}$ and $\mathbf{B}$. For electrostatics we have $\mathbf{E} = \nabla \mathbf{A}^0$, however $\mathbf{E}$ should not be zero on the boundary.

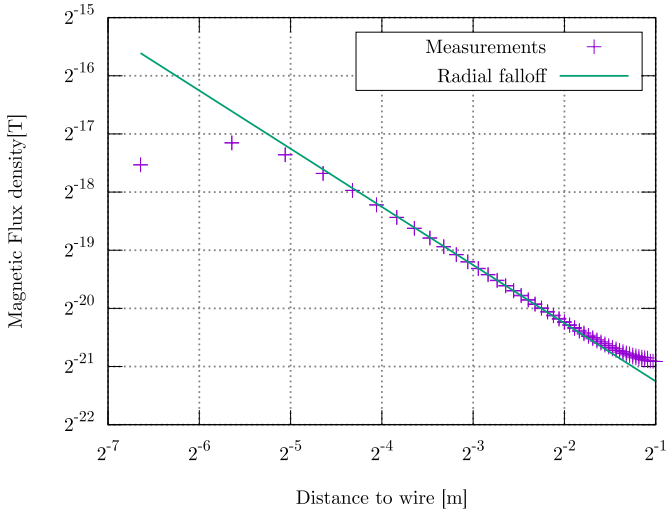
6.2.2 Ampere's Law

We can perform an analogous testcase for the Ampere's Law:

$$\oint_{\partial S} \mathbf{B} \cdot \mathrm{d}\boldsymbol{l} = \oiint_{S} \mu_0 \mathbf{J} \tag{6.1}$$

A constant current I flowing through a cable therefore creates a field strength

$$B(r) = \frac{\mu_0}{2\pi r} \tag{6.2}$$



For this test, particles were arranged in a line through the center of a 1m^3 domain, with a total charge of 1C moving with $1\frac{\text{m}}{\text{s}}$, which amounts to a current of 1A. Analogous to Figure 6, the magnetic flux density was sampled at different distances to that wire, converted to Tesla and compared against the analytical Equation 6.2.

Figure 7: Result of the Ampere Test Case

6.3 Synchrotron radiation test

To test the correctness of the current deposition scheme, we lock a bunch of particles with total charge q onto a fixed circular trajectory with radius r and velocity $\boldsymbol{v} = \boldsymbol{\beta} c$. The desired total radiation can be computed with the Larmor formula [53]:

$$P = \frac{q^2}{6\pi\varepsilon_0 c}\gamma^6\left(\|\dot{\boldsymbol{\beta}}\|^2 - \|\boldsymbol{\beta}\times\dot{\boldsymbol{\beta}}\|^2\right) \tag{6.3}$$

In case $\boldsymbol{\beta}$ and $\dot{\boldsymbol{\beta}}$ are perpendicular, Equation 6.3 can be simplified to

$$P = \frac{q^2 c}{6\pi\varepsilon_0}\frac{\gamma^4\beta^4}{r^2} \tag{6.4}$$

6.3.1 Radiation sampling

A computation that arises frequently is the numerical evaluation of a surface integral over a certain quantity. This could for example be the radiation that is emitted by a particle bunch. Specifically, we want to measure the poynting vector integrated over a sphere with radius such that the particles never go outside it:

$$\text{Radiation power} = \int_{\partial S} (\mathbf{E}\times\mathbf{B})\cdot \mathrm{d}\boldsymbol{n}$$

where ∂S denotes the surface of the sphere with surface normal $\boldsymbol{n}$.
Tracer particle approach: In order to evaluate such an integral over a sphere with radius

r, we uniformly place N tracer particles on the spheres surface. To generate a random distribution on a sphere, we use the following transformation:

$$\begin{aligned}\varphi &= 2\pi\nu_1 \\ \theta &= \operatorname{acos}(2\nu_2 - 1)\end{aligned}$$

where $\nu_{1,2}$ are random variables in $[0, 1]$. We transform the angles φ and θ to cartesian coordinates according to the following convention:

$$\mathbf{r} = r \begin{bmatrix} \cos(\varphi)\sin(\theta) \\ \sin(\varphi)\sin(\theta) \\ \cos(\theta) \end{bmatrix}$$

Our integral is then evaluated with a sum

$$\frac{4\pi r^2}{N} \sum_i^N \mathbf{E}(\mathbf{r}_i) \times \mathbf{B}(\mathbf{r}_i) \cdot \boldsymbol{n}_i$$

The evaluation of $\mathbf{E}$ and $\mathbf{B}$ is done using the linear interpolation scheme described in Section 3.2.3.

6.3.2 Radiation measurements

The results obtained from this test have been very inaccurate, for a lot of parameters not matching in order of magnitude. This might be explained by the extemely small wavelength of emitted synchrotron radiation, which leads to larger dispersion error as described in Equation 3.7.

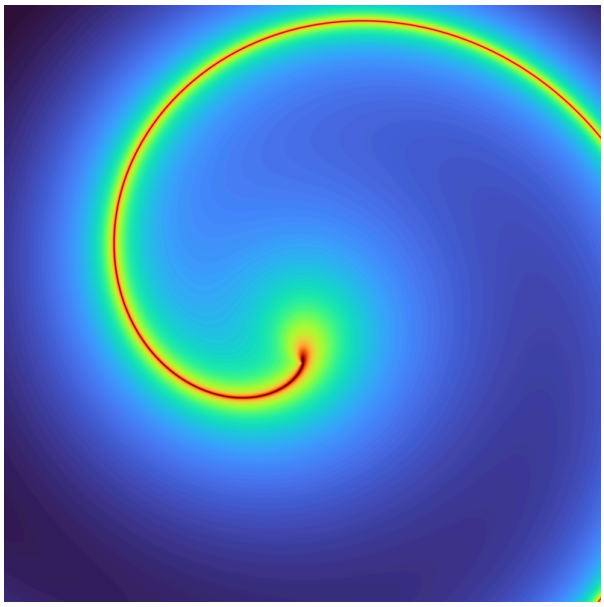


Figure 8: Exact evaluation of $\log_{10} \mathbf{S}$

Figure 8 shows a heatmap of the exact values for the logarithmic magnitude of the Poynting vector

$$\log_{10} \|\mathbf{E} \times \mathbf{B}\| = \log_{10} \mathbf{S}$$

for a particle spinning with $\gamma = 10$ and an orbit radius of $\frac{1}{10}$, where the width and height of the image are 1. The original version of this image was generated with a resolution of 5000×5000, and the region around the wave peak was still not fully captured, which implies a very small wavelength. Figure 8 was generated using a bisection-based Liénard-Wiechert potential (Equation 2.27, Equation 2.28) solver available on [54].

6.4 Free Electron Laser (Infrared)

A complete test we can run to test whether all parts of the simulation are implemented correctly is the FEL-IR-a test from the MITHRA [24]. In order to reproduce this test case, we use a JSON parser [55] to parse the relevant parameters:

Parameter name	Value
length-scale	micrometers
time-scale	picoseconds
extents	4200.0, 4200.0, 280.0
resolution	96, 96, 3000
mesh-center	0.0, 0.0, 0.0
total-time	30000.0
bunch-time-step	1.6
mesh-truncation-order	2
space-charge	false
solver	NSFD
charge	1.846e8
mass	1.846e8
number-of-particles	131072 = 2^{17}
gamma	100.41
sigma-position	260.0, 260.0, 50.25
sigma-momentum	10e-8, 1.0e-8, 100.41e-4
distribution-truncations	1040.0, 1040.0, 90.0
bunching-factor	0.01

Table 4: Input parameters for the FEL-IR test case

As stated in Section 1, the simulation runs completely in scaled Planck units. The parameters `length-scale` and `time-scale` simply make the configuration parser interpret the `30000.0` as picoseconds and the `extents` as micrometers.

6.4.1 Forward radiation

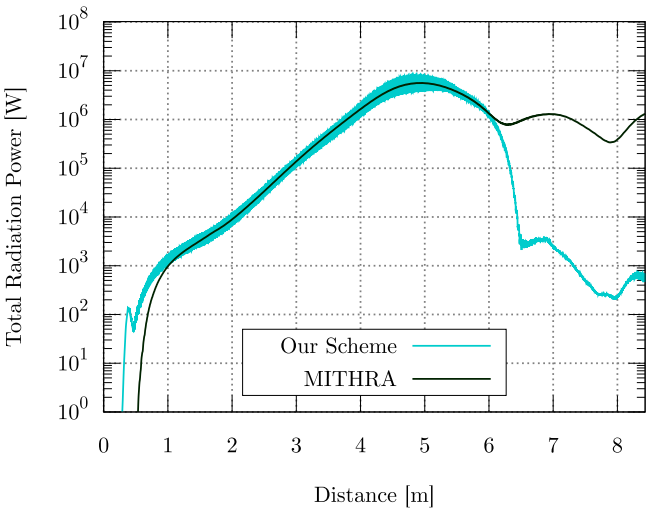
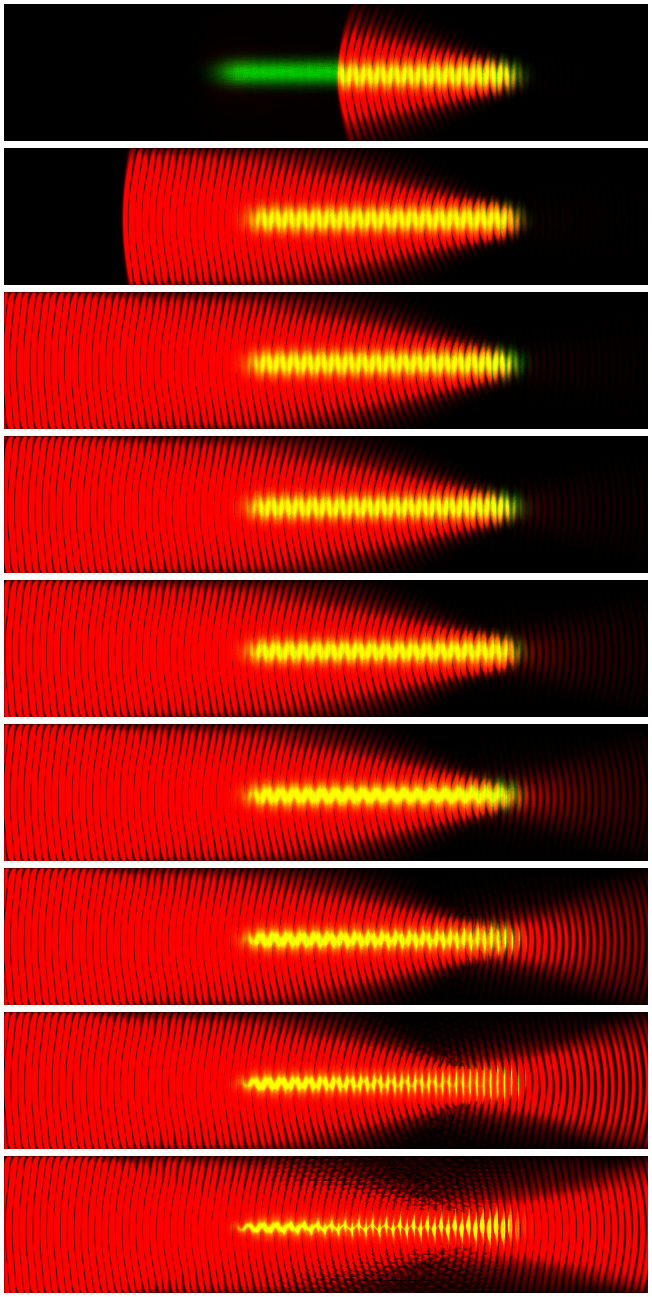


Figure 9: Comparison of measured forward radiations

Our simulation shows a satisfactory agreement with the reference Solver MITHRA [24]. The reason for the noise in our radiation curve is the absence of a fourier transform to only measure the radiation of a certain wavelength. Furthermore, after the bunch exits the undulator, the radiation measured by our scheme plummets; all that remains are spurious reflections of boundaries. It is unclear to us why MITHRA's predicted radiation remains so high. The undulator ends at 5m (from Table 4)



The following pictures show the evolution of the

- **Electron distribution**
- **Radiation power density**

evaluated purely in the bunch frame as the bunch progresses through the undulator. Only the z-component of the Poynting vector is used for coloring. In the beginning, there is almost no forward radiation and a lot of backward radiation. This does not mean the results are wrong, but the forward radiation's Poynting vector will have its z-component multiplied by a factor of

$$\frac{1+\beta}{1-\beta},$$

while for the backward radiation it's

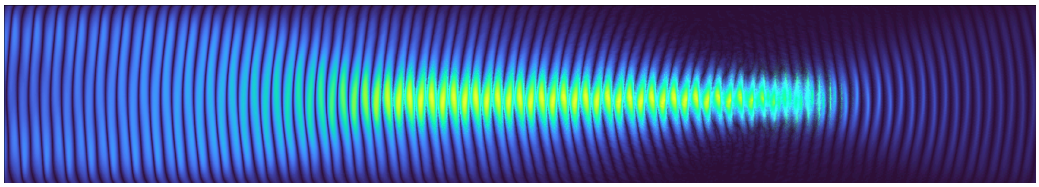
$$\frac{1-\beta}{1+\beta}.$$

As the bunch progresses, the Bunching Factor increases steadily, causing the forward radiation to become constructive. The bunch starts to separate into smaller microbunches; regions of very high particle density, spaced out with gaps of length

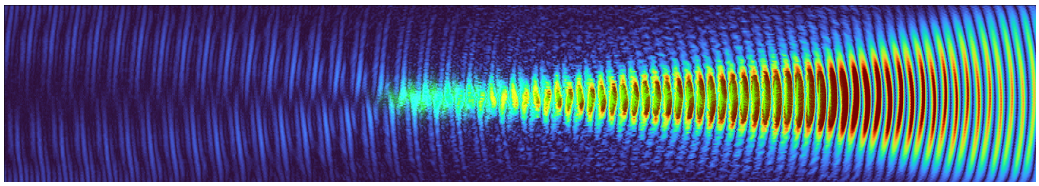
$$\lambda_{u,\text{bunch}} = \frac{\lambda_u}{\gamma_0}$$

inbetween.

The following two pictures show a juxtaposition of the radiation measured in the bunch frame, and measured in the lab frame. If we apply the Lorentz transform from Equation 2.22, forward radiation is amplified while backward radiation is diminished.



$$\text{Color} = \mathbf{E}_{\text{bunch}} \times \mathbf{B}_{\text{bunch}}$$



$$\text{Color} = \mathbf{E}_{\text{lab}} \times \mathbf{B}_{\text{lab}}$$

6.5 Scaling

In an idealized setting, dividing the work to N different processors divides the time required to perform a certain task by N:

$$t_{\mathrm{N,ideal}} = \frac{t_1}{N} \tag{6.5}$$

In some cases, even a better ratio can be achieved [56], mainly due to cache effects. However, many real-world applications consists of a purely sequential task, coupled with a parallelizable task. The well-known Amdahl's Law [57] states the maximum speedup depending under the assumption that a certain fraction of the program can only be run sequentially:

$$t_N = (1-p)t_1 + \frac{pt_1}{N} \tag{6.6}$$

where p denotes the parallelizable fraction of the program. Note that Equation 6.6 reduces to Equation 6.5 for $p = 1$.

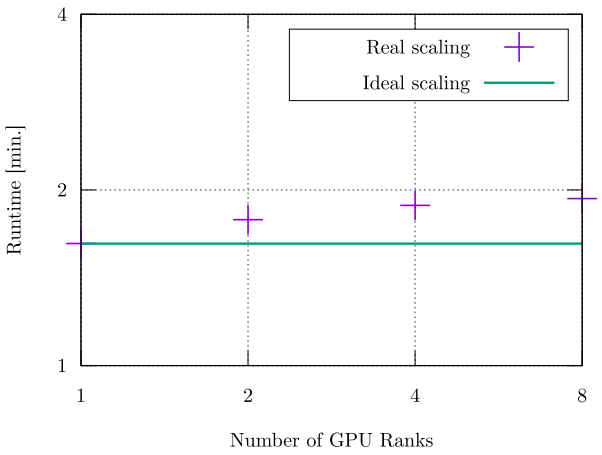
Another well-known relation is Gustafson's Law [58]:

$$(1-p) + pN \tag{6.7}$$

which states the theoretically achievable speedup under the assumption that the parallelizable part of the program also grows linearly with N.

The behavior of the program with a constant problem size with respect to the processor count N is called **strong scaling**(Section 6.5.2). The behavior of the program with respect to the processor count N and a workload linearly proportional to N is called **weak scaling**(Section 6.5.1).

6.5.1 Weak scaling



For the multi-GPU benchmark, the parameters were modified:

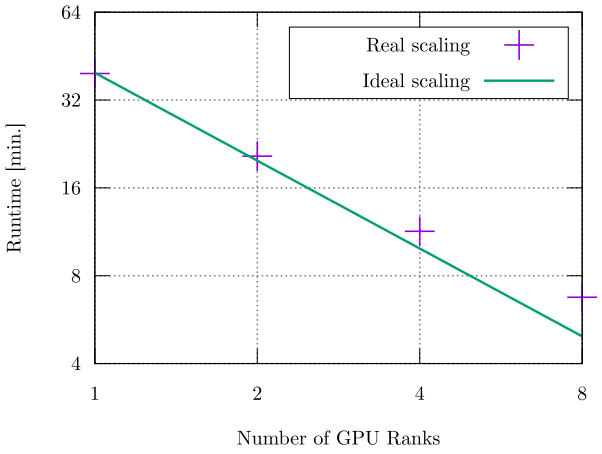
- The resolution was increased to $128 \times 128 \times 2000N$.
- The particle count was increased to 10^6.

Further improvements could be made, especially by employing load-balancing. NVIDIA A100 GPUs were used for the benchmark shown in Figure 10.

Figure 10: Weak scaling for the FEL-IR Test case

The work factor was introduced to the weak scaling benchmark by multiplying the resolution in z direction with N, while keeping the total amount of timesteps constant. Since the CFL condition for nonstandard finite differences dictates $\Delta t = \Delta z$, this also leads to a linear decline in total simulation time.

6.5.2 Strong scaling



For the Multi-GPU strong-scaling benchmark, the parameters were modified in the same way:

- The resolution was increased to $128 \times 128 \times 10000$, leading to an ≈ 12 increase in grid points compared to Table 4.
- The particle count was increased to 10^6.

Figure 11: Strong scaling for the FEL-IR Test case

Even without multiple ranks, Figure 11 shows good performance: With the same parameters, [24] compiled for and run on a 12-Core AMD Ryzen 3900X prints an ETA of 36 hours and 17 minutes, compared to our runtime of 39 minutes on one rank. This corresponds to a speedup of ≈ 56 for going from 12 Cores and 24 Threads to one GPU. Figure 12 shows furthermore promising conservation of efficiency

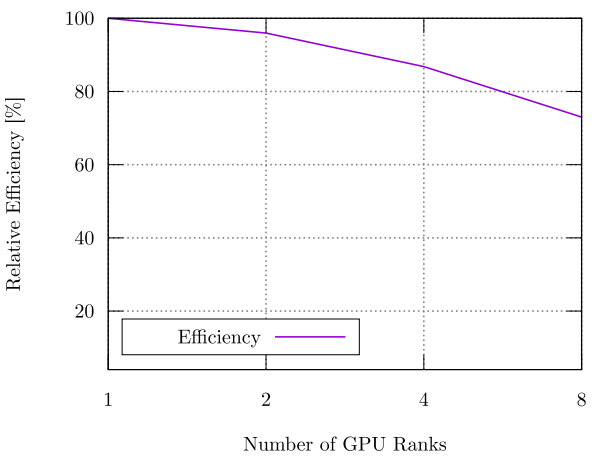


Figure 12: Relative efficiency in strong scaling analysis

Chapter 7

Conclusion

7.1 Achieved Goals

The goal was to implement a performance portable FDTD solver coupled with a charge conserving particle interpolation scheme. Literature describing the numerical procedures for dispersion correction [21] and charge conservation [29, 30] were consulted and successfully implemented. In order to generalize our implementation to 32- and 64-bit floats, a scaled set of natural units was derived. Several test cases demonstrate the correctness of the implemented algorithms as well as their quantitative correctness in SI-Units, see Figure 5 as well as Figure 6 Figure 7.

We were able to reproduce a widely known FEL testcase outlined in Table 4 and shown in Figure 9. Furthermore, excellent scalability for extremely high-throughput GPUs was maintained as shown by the scaling analysis results Figure 11, Figure 10 and Figure 12.

The synchrotron of Section 6.3 has turned out to be an obstacle rather than a useful test case. While the Larmor formula serves its purpose to verify a Liénard-Wiechert Solver(Section 2.2.3), measuring the radiation of single particles leads to high dispersion errors.

The excellent scaling results of Figure 10 are due to the good separability of the chosen domain: The resolution along the z-axis is huge in comparison to x and y.

The implementation is fully compatible with C++20 and implemented according to Kokkos [59] and IPPL [13] paradigms. The source code and steps to reproduce are openly available at [54, 13].

7.2 Further Research

Several improvements are still in relatively close reach. As described in Section 2.2.3, a better algorithm to compute initial conditions should be implemented. Furthermore, the results of Figure 3 should prompt further investigation into spectral solvers and their uses to implement perfectly absorbing boundary conditions.

Another use of the Liénard-Wiecher potentials Equation 2.28 is the exact evaluation of forward radiation given the trajectory of a particle. Knowing that the particles are updated using the Boris scheme of Equation 3.29, we can reconstruct all the relevant information of a particle's trajectory only given a list of time-position pairs. This is feasible even for high particle and timestep counts.

Appendix

A.1 Second order ABC edge implementation

```
/*
 * Scalar is simply the number type.
 * edge_axis is the index of the edge axis:
 * 0 if we're on an x-aligned axis,
 * 1 for an y-aligned axis and
 * 2 for a z-aligned axis
 * The normal_axis1 and normal_axis2 are the axis indices for
 * the axes perpendicular to the edge
 * na1_zero is whether along the normal_axis1 we're at 0
 * na2_zero is whether along the normal_axis2 we're at 0
 */
template<typename _scalar, unsigned edge_axis, unsigned normal_axis1, unsigned normal_axis2, bool na1_zero, bool na2_zero>
struct second_order_abc_edge{
    using scalar = _scalar;
    scalar Eweights[5];

    KOKKOS_FUNCTION second_order_abc_edge(ippl::Vector<scalar, 3> hr, scalar dt){
        static_assert(normal_axis1 != normal_axis2);
        static_assert(edge_axis != normal_axis2);
        static_assert(edge_axis != normal_axis1);
           static_assert((edge_axis == 2 && normal_axis1 == 0 && normal_axis2 == 1) || (edge_axis == 0 && normal_axis1 == 1 && normal_axis2 == 2) || (edge_axis == 1 && normal_axis1 == 2 && normal_axis2 == 0));
        constexpr scalar c0_ = scalar(1);
        scalar d    =    ( 1.0 / hr[normal_axis1] + 1.0 / hr[normal_axis2] ) / ( 4.0 * dt ) + 3.0 / ( 8.0 * c0_ * dt * dt );
        if constexpr(normal_axis1 == 0 && normal_axis2 == 1){ // xy edge (along z)
           Eweights[0] = ( - ( 1.0 / hr[normal_axis2] - 1.0 / hr[normal_axis1] ) / ( 4.0 * dt ) - 3.0 / ( 8.0 * c0_ * dt * dt )) / d;
           Eweights[1] = (   ( 1.0 / hr[normal_axis2] - 1.0 / hr[normal_axis1] ) / ( 4.0 * dt ) - 3.0 / ( 8.0 * c0_ * dt * dt )) / d;
           Eweights[2] = (   ( 1.0 / hr[normal_axis2] + 1.0 / hr[normal_axis1] ) / ( 4.0 * dt ) - 3.0 / ( 8.0 * c0_ * dt * dt )) / d;
            Eweights[3] = ( 3.0 / ( 4.0 * c0_ * dt * dt ) - c0_ / (4.0 * hr[edge_axis] * hr[edge_axis])) / d;
            Eweights[4] = c0_ / ( 8.0 * hr[edge_axis] * hr[edge_axis] ) / d;
    }
        template<typename view_type, typename Coords>
      KOKKOS_INLINE_FUNCTION auto operator()(const view_type& A_n, const view_type& A_nm1,const view_type& A_np1, const Coords& c)const -> typename view_type::value_type{
```

```
33          uint32_t i = c[0];
34          uint32_t j = c[1];
35          uint32_t k = c[2];
                  ippl::Vector<int32_t, 3> normal_axis1_onehot = ippl::Vector<int32_t,
36  3>{normal_axis1 == 0, normal_axis1 == 1, normal_axis1 == 2} * int32_t(na1_zero ? 1 : -1);
                  ippl::Vector<int32_t, 3> normal_axis2_onehot = ippl::Vector<int32_t,
37  3>{normal_axis2 == 0, normal_axis2 == 2, normal_axis2 == 2} * int32_t(na2_zero ? 1 : -1);
38          ippl::Vector<uint32_t, 3> acc0 = {i, j, k};
39          ippl::Vector<uint32_t, 3> acc1 = acc0 + normal_axis1_onehot.cast<uint32_t>();
40          ippl::Vector<uint32_t, 3> acc2 = acc0 + normal_axis2_onehot.cast<uint32_t>();
             ippl::Vector<uint32_t, 3> acc3 = acc0 + normal_axis1_onehot.cast<uint32_t>()
41  + normal_axis2_onehot.cast<uint32_t>();
42          2}).cast<uint32_t>();
43          ippl::Vector<uint32_t, 3> axisp{edge_axis == 0, edge_axis == 1, edge_axis == 2};
44          return advanceEdgeS(
45                                              A_n(i, j, k),      A_nm1(i, j, k),
46              apply(A_np1, acc1),   apply(A_n, acc1   ), apply(A_nm1, acc1),
47              apply(A_np1, acc2),   apply(A_n, acc2   ), apply(A_nm1, acc2),
48              apply(A_np1, acc3),   apply(A_n, acc3   ), apply(A_nm1, acc3),
               apply(A_n, acc0 - axisp), apply(A_n, acc1 - axisp), apply(A_n, acc2 - axisp),
49  apply(A_n, acc3 - axisp),
               apply(A_n, acc0 + axisp), apply(A_n, acc1 + axisp), apply(A_n, acc2 + axisp),
50  apply(A_n, acc3 + axisp)
51          );
52      }
53      template <typename value_type>
54      KOKKOS_INLINE_FUNCTION value_type advanceEdgeS
55              (               value_type v1 , value_type v2 ,
56                              value_type v3 , value_type v4 , value_type v5 ,
57                              value_type v6 , value_type v7 , value_type v8 ,
58                              value_type v9 , value_type v10, value_type v11,
59                              value_type v12, value_type v13, value_type v14,
60                              value_type v15, value_type v16, value_type v17,
61                              value_type v18, value_type v19)const noexcept{
62      value_type v0 =
63      Eweights[0] * (v3 + v8) +
64      Eweights[1] * (v5 + v6) +
65      Eweights[2] * (v2 + v9) +
66      Eweights[3] * (v1 + v4 + v7 + v10) +
67      Eweights[4] * (v12 + v13 + v14 + v15 + v16 + v17 + v18 + v19) - v11;
68      return v0;
69    }
```

A.2 Second order boundary condition corners

```
template<typename _scalar, bool x0, bool y0, bool z0>
struct second_order_abc_corner{
    using scalar = _scalar;
    scalar Cweights[17];
     KOKKOS_FUNCTION second_order_abc_corner(ippl::Vector<scalar, 3> hr, scalar
dt){
         constexpr scalar c0_ = scalar(1); //Since the entire simulation is in
(scaled) natural units, we can assume c = 1
        Cweights[0]  =   ( - 1.0 / hr[0] - 1.0 / hr[1] - 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[1]  =   (   1.0 / hr[0] - 1.0 / hr[1] - 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[2]  =   ( - 1.0 / hr[0] + 1.0 / hr[1] - 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[3]  =   ( - 1.0 / hr[0] - 1.0 / hr[1] + 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[4]  =   (   1.0 / hr[0] + 1.0 / hr[1] - 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[5]  =   (   1.0 / hr[0] - 1.0 / hr[1] + 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[6]  =   ( - 1.0 / hr[0] + 1.0 / hr[1] + 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[7]  =   (   1.0 / hr[0] + 1.0 / hr[1] + 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[8]  = - ( - 1.0 / hr[0] - 1.0 / hr[1] - 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[9]  = - (   1.0 / hr[0] - 1.0 / hr[1] - 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[10] = - ( - 1.0 / hr[0] + 1.0 / hr[1] - 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[11] = - ( - 1.0 / hr[0] - 1.0 / hr[1] + 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[12] = - (   1.0 / hr[0] + 1.0 / hr[1] - 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[13] = - (   1.0 / hr[0] - 1.0 / hr[1] + 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[14] = - ( - 1.0 / hr[0] + 1.0 / hr[1] + 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[15] = - (   1.0 / hr[0] + 1.0 / hr[1] + 1.0 / hr[2] ) / (8.0 *
dt) - 1.0 / ( 4.0 * c0_ * dt * dt);
        Cweights[16] = 1.0 / (2.0 * c0_ * dt * dt);
    }
    template<typename view_type, typename Coords>
        KOKKOS_INLINE_FUNCTION  auto  operator()(const  view_type&  A_n,  const
view_type&  A_nm1,const  view_type&  A_np1,  const  Coords&  c)const  ->  typename
view_type::value_type{
        //First implementation: 0,0,0 corner
```

```
28          constexpr uint32_t xoff = (x0) ? 1 : uint32_t(-1);
29          constexpr uint32_t yoff = (y0) ? 1 : uint32_t(-1);
30          constexpr uint32_t zoff = (z0) ? 1 : uint32_t(-1);
31          constexpr ippl::Vector<uint32_t, 3> offsets[8] = {
32              ippl::Vector<uint32_t, 3>{0,0,0},
33              ippl::Vector<uint32_t, 3>{xoff,0,0},
34              ippl::Vector<uint32_t, 3>{0,yoff,0},
35              ippl::Vector<uint32_t, 3>{0,0,zoff},
36              ippl::Vector<uint32_t, 3>{xoff,yoff,0},
37              ippl::Vector<uint32_t, 3>{xoff,0,zoff},
38              ippl::Vector<uint32_t, 3>{0,yoff,zoff},
39              ippl::Vector<uint32_t, 3>{xoff,yoff,zoff},
40          };
41          return advanceCornerS(
42                                                apply(A_n, c), apply(A_nm1, c),
              apply(A_np1, c + offsets[1]), apply(A_n, c + offsets[1]), apply(A_nm1,
43  c + offsets[1]),
              apply(A_np1, c + offsets[2]), apply(A_n, c + offsets[2]), apply(A_nm1,
44  c + offsets[2]),
              apply(A_np1, c + offsets[3]), apply(A_n, c + offsets[3]), apply(A_nm1,
45  c + offsets[3]),
              apply(A_np1, c + offsets[4]), apply(A_n, c + offsets[4]), apply(A_nm1,
46  c + offsets[4]),
              apply(A_np1, c + offsets[5]), apply(A_n, c + offsets[5]), apply(A_nm1,
47  c + offsets[5]),
              apply(A_np1, c + offsets[6]), apply(A_n, c + offsets[6]), apply(A_nm1,
48  c + offsets[6]),
              apply(A_np1, c + offsets[7]), apply(A_n, c + offsets[7]), apply(A_nm1,
49  c + offsets[7])
          );
50      }
51      template<typename value_type>
52      KOKKOS_INLINE_FUNCTION value_type advanceCornerS
53                               (value_type v1 , value_type v2 ,
54                                value_type v3 , value_type v4 , value_type v5 ,
55                                value_type v6 , value_type v7 , value_type v8 ,
56                                value_type v9 , value_type v10, value_type v11,
57                                value_type v12, value_type v13, value_type v14,
58                                value_type v15, value_type v16, value_type v17,
59                                value_type v18, value_type v19, value_type v20,
60                                       value_type v21, value_type v22, value_type
  v23)const noexcept{
61      return      - ( v1  * (Cweights[16]) + v2  * (Cweights[8]) +
62      v3  * Cweights[1] + v4  * Cweights[16] + v5  * Cweights[9] +
63      v6  * Cweights[2] + v7  * Cweights[16] + v8  * Cweights[10] +
64      v9  * Cweights[3] + v10 * Cweights[16] + v11 * Cweights[11] +
65      v12 * Cweights[4] + v13 * Cweights[16] + v14 * Cweights[12] +
66      v15 * Cweights[5] + v16 * Cweights[16] + v17 * Cweights[13] +
67      v18 * Cweights[6] + v19 * Cweights[16] + v20 * Cweights[14] +
68      v21 * Cweights[7] + v22 * Cweights[16] + v23 * Cweights[15]) / Cweights[0];
69
```

```
70     }
71  };
```

A.3 Rendering

OpenGL requires all rendering to be done in the unit cube

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \in [-1, 1]^3$$

which are also known as **normalized device coordinates.**
The default viewport will map x, y from the interval $[-1, 1]$ to $[0, w]$ and $[0, h]$, where w and h denote the (pixel) resolution of the rendering surface. The coordinate z is only used for depth-testing. It is therefore required that we map 3D objects into normalized coordinates based on a camera position and look direction.

1.3.1 Projective Geometry

Given a camera position p and a look direction l, a "view matrix" can be computed [60]. Given a field of view angle, near and far clipping planes a perspective matrix can be computed as described in [61]. This allows a transformation of almost arbitrary geometric objects into normalized device coordinates.

The OpenGL context is obtained through the EGL library. This allows the program to render pixels on an offscreen server without opening a window.

1.3.2 Framebuffer Reduction

In a split-memory environment such as MPI, every process has to obtain its own OpenGL context. To render everything to a single surface, every process has to draw its local elements to a local surface, followed by a reduction over all local surfaces. The reduction operation used is called depth blend:

$$o_{i,j} = \begin{cases} a_{i,j} & \text{if } d_{a_{i,j}} < d_{b_{i,j}} \\ b_{i,j} & \text{otherwise} \end{cases}$$

where $o_{i,j}$ denotes an output pixel, $a_{i,j}, b_{i,j}$ denotes input pixels and d denotes the corresponding depth buffer. This means for whichever depth buffer contains the "closer" value, the corresponding color value will be picked.

Acknowledgements

I extend my heartfelt gratitude to Sonali and Andreas for their invaluable assistance and unwavering support throughout the duration of this project. Their dedication and expertise were instrumental in navigating the challenges we encountered along the way, ensuring the project's success. I am particularly indebted to Dr. Arya Fallahi for his generous commitment to guiding me through the intricacies of electromagnetic theory. His willingness to patiently address a multitude of questions greatly enriched my understanding and significantly contributed to the project's depth and quality. Furthermore, I am immensely thankful to Arnau Albà for his exceptional dedication and tireless efforts in overcoming obstacles to bring our final experiment to fruition. Arnau's perseverance and ingenuity were indispensable in resolving complex technical issues, ultimately leading to the successful completion of our objectives. Their collective contributions have greatly enriched this project. I am truly fortunate to have had such remarkable individuals by my side, and I extend my deepest appreciation to each of them for their invaluable support and guidance.

Bibliography

[1] V. Pankovic, Dynamical determination of the basic Planck units, (2021)

[2] Wassermaus et al, (2024). https://de.wikipedia.org/wiki/Planck-Einheiten

[3] (n.d.). https://phys.libretexts.org/Bookshelves/Electricity_and_Magnetism/Electricity_and_Magnetism_(Tatum)/15%3A_Maxwell%27s_Equations/15.11%3A_Maxwell%E2%80%99s_Equations_in_Potential_Form

[4] V. Betz, R. Mittra, Absorbing boundary conditions for the finite-difference time-domain analysis of guided-wave structures, (1993)

[5] G. Mur, Absorbing Boundary Conditions for the Finite-Difference Approximation of the Time-Domain Electromagnetic-Field Equations, Transactions on Electromagnetic Compatibility 23 (1981) 1–6

[6] A. J. Macfarlane, Dirac Matrices and the Dirac Matrix Description of Lorentz Transformations, (1965). https://doi.org/https://doi.org/10.1007/BF01773348

[7] P. Collas, D. Klein, The Dirac Equation in Curved Spacetime: A Guide for Calculations, Springer International Publishing, 2019. https://books.google.ch/books?id=YymODwAAQBAJ

[8] C. W. Misner, K. S. Thorne, J. A. Wheeler, Gravitation, 1973

[9] H. Daniel, Elektrodynamik - Relativistische Physik, De Gruyter, 1997. https://books.google.ch/books?id=8vAC8YG41goC

[10] A. Taflove, Computational Electrodynamics: The Finite-difference Time-domain Method, Artech House, 1995. https://books.google.ch/books?id=viVRAAAAMAAJ

[11] S. Mayani, V. Montanaro, A. Cerfon, M. Frey, S. Muralikrishnan, A. Adelmann, A Massively Parallel Performance Portable Free-space Spectral Poisson Solver, (2024)

[12] V. Montanaro, (2023). https://amas.web.psi.ch/people/aadelmann/ETH-Accel-Lecture-1/projectscompleted/cse/Montanaro_report_final.pdf

[13] S. Muralikrishnan, M. Frey, A. Vinciguerra, M. Ligotino, A. J. Cerfon, M. Stoyanov, R. Gayatri, A. Adelmann, Scaling and performance portability of the particle-in-cell scheme for plasma physics applications through mini-apps targeting exascale architectures, in: Proceedings of the 2024 SIAM Conference on Parallel Processing for Scientific Computing (PP), 2024: pp. 26–38

[14] A. Alba, Private conversation with A. Albà, (2024). https://www.psi.ch/de/lsm/people/arnau-alba-jacas

[15] D. ALBA, L. LUSANNA, THE LIENARD–WIECHERT POTENTIAL OF CHARGED SCALAR PARTICLES AND THEIR RELATION TO SCALAR ELECTRODYNAMICS IN THE REST-FRAME INSTANT FORM, International Journal of Modern Physics a 13 (1998) 2791–2831. https://doi.org/10.1142/s0217751x98001426

[16] H. van Hees, Die Liénard-Wiechert-Potentiale, (2018). https://itp.uni-frankfurt.de/~hees/publ/lw-potentiale.pdf

[17] ESSENTIAL GRADUATE PHYSICS - CLASSICAL ELECTRODYNAMICS, 2024. https://phys.libretexts.org/Bookshelves/Electricity_and_Magnetism/Essential_Graduate_Physics_-_Classical_Electrodynamics_(Likharev)/09%3A_Special_Relativity/9.06%3A_Relativistic_Particles_in_Electric_and_Magnetic_Fields

[18] F. Zhen, Z. Chen, J. Zhang, Toward the development of a three-dimensional unconditionally stable finite-difference time-domain method, IEEE Transactions on Microwave Theory and Techniques 48 (2000) 1550–1558. https://doi.org/10.1109/22.869007

[19] K. Yee, Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media, IEEE Transactions on Antennas and Propagation 14 (1966) 302–307. https://doi.org/10.1109/TAP.1966.1138693

[20] M. L. Winkler, High-order well-balanced finite difference schemes based on equilibrium flux reconstruction, (2019). https://gitlab.psi.ch/AMAS-students/winkler_msc/-/raw/main/report/report_2019.pdf?ref_type=heads&inline=false

[21] A. Fallahi, MITHRA 2.0: A Full-Wave Simulation Tool for Free Electron Lasers, (2020)

[22] M. Sadr, Private conversation with Dr. Mohsen Sadr, (2023)

[23] C. A. d. Moura, C. S. Kubrusly, The Courant-Friedrichs-Lewy (CFL) Condition: 80 Years After Its Discovery, Birkhäuser Basel, 2012

[24] A. Fallahi, MITHRA 2.0: A Full-Wave Simulation Tool for Free Electron Lasers, (2020). https://github.com/aryafallahi/mithra

[25] J. P. Boris, ACCELERATION CALCULATION FROM A SCALAR POTENTIAL., 1970

[26] H. Qin, S. Zhang, J. Xiao, J. Liu, Y. Sun, W. M. Tang, Why is Boris algorithm so good?, Physics of Plasmas 20 (2013) 84503–84504. https://doi.org/10.1063/1.4818428

[27] S. Markidis, G. Lapenta, The energy conserving particle-in-cell method, Journal of Computational Physics 230 (2011) 7037–7052. https://doi.org/https://doi.org/10.1016/j.jcp.2011.05.033

[28] J. Villasenor, O. Buneman, Rigorous charge conservation for local electromagnetic field solvers, Computer Physics Communications 69 (1992) 306–316. https://doi.org/10.1016/0010-4655(92)90169-Y

[29] T. Esirkepov, Exact charge conservation scheme for Particle-in-Cell simulation with an arbitrary form-factor, Computer Physics Communications 135 (2001) 144–153. https://doi.org/https://doi.org/10.1016/S0010-4655(00)00228-9

[30] T. Umeda, Y. Omura, T. Tominaga, H. Matsumoto, A new charge conservation method in electromagnetic particle-in-cell simulations, Computer Physics Communications 156 (2003) 73–85. https://doi.org/https://doi.org/10.1016/S0010-4655(03)00437-5

[31] Y. Ding, S. Huang, J. Zhuang, Y. Wang, K. Zhao, J. Chen, Design and optimization of IR SASE FEL at Peking University**Work supported by Chinese department of Science and Technology under the National Basic Research Projects No. 2002CB713600., Free Electron Lasers 2003 (2004) 416–420. https://doi.org/https://doi.org/10.1016/B978-0-444-51727-2.50093-6

[32] M. R. Howells, B. M. Kincaid, The Properties of Undulator Radiation, in: A. S. Schlachter, F. J. Wuilleumier (Eds.), New Directions in Research with

Third-Generation Soft X-Ray Synchrotron Radiation Sources, Springer Netherlands, Dordrecht, 1994: pp. 315–358. https://doi.org/10.1007/978-94-011-0868-3_13

[33] D. D. Nolte, The tangled tale of phase space, Physics Today 63 (2010) 33–38. https://doi.org/10.1063/1.3397041

[34] (n.d.). https://phys.libretexts.org/Bookshelves/University_Physics/University_Physics_(OpenStax)/University_Physics_III_-_Optics_and_Modern_Physics_(OpenStax)/05%3A___Relativity/5.07%3A_Relativistic_Velocity_Transformation

[35] (2020). https://developer.nvidia.com/cuda-toolkit

[36] (n.d.). https://github.com/llvm/llvm-project

[37] (n.d.). https://rocm.docs.amd.com/

[38] cppreference.com, (n.d.). https://en.cppreference.com/w/cpp

[39] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, J. McDonald, Parallel programming in OpenMP, Morgan kaufmann, 2001

[40] C. V. Ravishankar, J. R. Goodman, Cache implementation for multiple microprocessors, (1983)

[41] (n.d.). https://www.cs.cmu.edu/afs/cs/academic/class/15418-s19/www/lectures/13_directory.pdf

[42] B. Gallmeister, M. Kerrisk, (n.d.). https://man7.org/linux/man-pages/man2/mmap.2.html

[43] M. P. Forum, MPI: A Message-Passing Interface Standard, University of Tennessee, USA, 1994

[44] (n.d.). https://docs.nvidia.com/cuda/

[45] N. Corporation, (2020). https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[46] T. Allen, R. Ge, In-depth analyses of unified virtual memory system for GPU accelerated computing, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Association for Computing Machinery„ St. Louis, Missouri, 2021. https://doi.org/10.1145/3458817.3480855

[47] M. Harris, (n.d.). https://developer.nvidia.com/blog/finite-difference-methods-cuda-cc-part-1

[48] M. Harris, (n.d.). https://developer.nvidia.com/blog/finite-difference-methods-cuda-c-part-2

[49] P. Micikevicius, 3D finite difference computation on GPUs using CUDA, in: 2009: pp. 79–84. https://doi.org/10.1145/1513895.1513905

[50] A. Gloster, L. Ó. Náraigh, cuSten — CUDA finite difference and stencil library, Softwarex 10 (2019) 100337–100338. https://doi.org/https://doi.org/10.1016/j.softx.2019.100337

[51] D. Michéa, D. Komatitsch, Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards, Geophysical Journal International 182 (2010) 389–402. https://doi.org/10.1111/j.1365-246X.2010.04616.x

[52] G. H. G. Tony F. Chan, R. J. Leveque, Algorithms for Computing the Sample Variance: Analysis and Recommendations, The American Statistician 37 (1983) 242–247. https://doi.org/10.1080/00031305.1983.10483115

[53] J. Larmor, LXIII. On the theory of the magnetic influence on spectra; and on the radiation from moving ions, The London, Edinburgh, And Dublin Philosophical Magazine and Journal of Science 44 (1897) 503–512. https://doi.org/10.1080/14786449708621095

[54] M. L. Winkler, AMAS Gitlab Repository, (2024). https://gitlab.psi.ch/AMAS-students/winkler_msc/

[55] N. Lohmann, JSON for Modern C++, (2023). https://github.com/nlohmann

[56] S. Ristov, R. Prodan, M. Gusev, K. Skala, Superlinear speedup in HPC systems: Why and when?, in: 2016 Federated Conference on Computer Science and Information Systems (Fedcsis), 2016: pp. 889–898

[57] R. Bryant, D. O'Hallaron, Computer Systems: A Programmer's Perspective, Global Edition, Pearson Education, 2019. https://books.google.ch/books?id=578oEAAAQBAJ

[58] J. L. Gustafson, Gustafson's Law, Springer US, Boston, MA, 2011. https://doi.org/10.1007/978-0-387-09766-4_78

[59] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, J. Wilke, Kokkos 3: Programming Model Extensions for the Exascale Era, IEEE Transactions on Parallel and Distributed Systems 33 (2022) 805–817. https://doi.org/10.1109/TPDS.2021.3097283

[60] (n.d.). https://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

[61] (n.d.). https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/opengl-perspective-projection-matrix.html