Direct Simulation Monte Carlo Methods Adapted for Coulomb Collisions

Semester Project

ETH Zurich

written by

Alexander Liemen | alexander.liemen@inf.ethz.ch

supervised by

Dr. Andreas Adelmann

scientific adviser

Dr. Sadr Mohsen

August 30, 2024

Abstract

This project focuses on simulating the relaxation behavior of a charged plasma, typically composed of electrons, using a particle-based approach. This study employs the Direct Simulation Monte Carlo (DSMC) approach to address the computational challenges inherent in solving kinetic equations for plasmas. The implementation leverages the IPPL (Independent Parallel Particle Layer) framework and evaluates two collision algorithms, proposed by TAKIZUKA and ABE (TA), and NANBU.
IPPL allows the algorithm to be fast and scalable. Both algorithms are benchmarked using three tests: the TRUBNIKOV test to evaluate the collisions, the relaxation of a delta-like initial distribution and the induced heating of a cold sphere.
Overall, the project concludes that NANBU's algorithm (1997) does not provide a significant advantage over TA (1977) in its proposed implementation due to its increased computation time caused by the need to solve an additional equation. However, further improvements are addressed at the end.

Contents

1 Introduction

This project aims to simulate the relaxing behavior of a charged plasma, typically consisting of electrons, although the methods are applicable to any charged particle cloud. Accurate plasma simulations have numerous applications in modern physics, ranging from advancing particle accelerator technology and understanding plasma-light interactions to even enhancing space propulsion systems [1].

There are several approaches to simulating a plasma. One method involves solving kinetic equations by discretizing them on a high dimensional grid – i.e. 3 in velocity, 3 in space and one in time – known as the discrete velocity approach. Another treats the plasma as a fluid, called momentum method. Additionally, various combinations of these methods exist. This project, however, employs a particle-based approach. Given the computational intensity of solving equations in this context, we utilize a MONTE-CARLO method. This involves an electric field solver for kinetic advancement and a statistical approach to handling collisions, which aims to significantly reduce computational demands.

The implementation of these methods is carried out using the IPPL framework [7]. A crucial aspect of this project is the MONTE-CARLO component for simulating collisions. Specifically, we compare two different algorithms – those proposed by TAKIZUKA and ABE [10], and NANBU [8] – in terms of their accuracy, efficiency, and scalability.

By evaluating these algorithms, this project aims to identify an effective method for scalable plasma simulations, contributing to the broader field of plasma physics and its many applications. All code referenced in the following can befound in [4].

2 Theory Background and Algorithms

As mentioned, the final simulation for non-equilibrium states consists of both a field solution to the long-range interaction (VLASOV term) and the statistical modeling of collisions. The field solution, which is already implemented by the IPPL (Integrated Plasma Physics Library), is a crucial part of the later investigation, but remains the same across both models. Therefore, the core of this discussion will be dedicated to outlining the most important equations and approximations used by the two collision algorithms.
The field solver and its implementation will be briefly explained in 2.5 to provide context.

2.1 Boltzmann and Fokker-Planck Equation

To derive the Direct Simulation MONTE-CARLO methods (in the following refered to as DSMC), the plasma is conveniently described as a particle distribution function $f_a(x^\mu, v^\mu)$, which described the number of particles of species[1] a per volume in phase space. x denotes position, v velocity.

[1]The test cases only deal with one species. Since everything also works with multiple species, the index a is used in the theory explanation.

The underlying equation to describe the behaviour of a distribution function f is the Boltzmann equation [9, p. 1]:

$$\frac{\partial f_a}{\partial t} + v^\mu \frac{\partial f_a}{\partial x^\mu} + \frac{F^\mu}{m}\frac{\partial f_a}{\partial v^\mu} = \left(\frac{\delta f_a}{\delta t}\right)_\mathrm{c}$$

with particle mass m and F includes external and self-consisting long-range force fields. Since we consider the electromagnetic field, we can insert the Lorentz force and rewrite it in terms of vectors:

$$\frac{\partial f_a}{\partial t} + \mathbf{v}\cdot\nabla_x f_a + \frac{e_a}{m_a}(\mathbf{E} + \mathbf{v}\times\mathbf{B})\cdot\nabla_v f_a = \left(\frac{\delta f_a}{\delta t}\right)_\mathrm{c}.$$

$\left(\frac{\delta f_a}{\delta t}\right)_\mathrm{c}$ describes the change of the distribution due to collisions, which can be calculated. Since inverse square laws have significant long-range interaction, we can assume that small angle collisions are the most probable, which leads to the following term in second order (component-wise notation) [9, p. 2]:

$$\left(\frac{\delta f_a}{\delta t}\right)_\mathrm{c} = -\frac{\partial}{\partial v^\mu}(f_a \langle \Delta v^\mu \rangle_a) + \frac{1}{2}\frac{\partial^2}{\partial v^\mu \partial v^\nu}(f_a \langle \Delta v^\mu \, \Delta v^\nu \rangle_a), \tag{1}$$

where $\langle \Delta v \rangle$ stands for the mean velocity increment per time.

Assuming that changes in velocity happen due to binary particle interactions, the expectation values can be calculated according to [9, p. 2] using the following differential cross section:

$$\sigma(\Omega) = \frac{e_a^2 e_b^2}{4 m_{ab}^2 u^4}\sin^{-4}\left(\frac{\Theta}{2}\right),$$

where a, b denote the different species included in the collision, e stands for the charge, $m_{ab} = \frac{m_a m_b}{m_a + m_b}$ is the reduced mass, Θ the scattering angle and u the absolute relative velocity between both particles. This gives the following collision operator and defines the Fokker-Planck equation together with (1) according to [11] in the Landau form:

$$\left(\frac{\delta f_a}{\delta t}\right)_\mathrm{c} = -\sum_b \frac{\partial}{\partial v_j}\frac{e_a^2 e_b^2 \ln\Lambda}{8\pi\epsilon_0^2 m_a}\int \mathrm{d}v' \left[\frac{\delta_{jk}}{u} - \frac{u_j u_k}{u^3}\right]\left[\frac{f_a}{m_b}\frac{\partial f_b(v')}{\partial v'_k} - \frac{f_b}{m_a}\frac{\partial f_a(v')}{\partial v_k}\right], \tag{2}$$

where $\ln\Lambda$ is the Coulomb logarithm[2].

Treating the distribution f as a sum of discrete particles, the integral in 2 can be solved. Then, after discretizing $\left(\frac{\delta f_a}{\delta t}\right)_\mathrm{c}$ and equation (1) in time, we can solve for $f^{t+\Delta t}$ and get the update equations outlined in 2.3 and 2.4.

[2]This parameter is calculated from the Debye length λ_D, which intuitively describes the cutoff around the electron until which collisions are significant. There is a formula that can be used to calculate the Coulomb logarithm. However, since this contains the temperature, the calculation might get computationally intense for very small gain. So, it is better to simply use $\ln\Lambda = 10$ from now on according to [3].

2.2 Algorithm Outline

The whole code as well as instructions to reproduce the plots can be found in A.1.
The general method is the same for both algorithms: First, the field solver is executed to determine the self consistent electric fields. Then, collision pairs are selected based on the specific algorithm being employed. Following this, binary collision updates are calculated for the selected pairs. Finally, a leap-frog integration step is utilized to advance the particles positions and velocities, taking into account both the calculated electromagnetic fields and the newly computed velocities from the collision updates. This basic outline of the simulation process is illustrated in algorithm 1.

Algorithm 1 General Simulation Step

```
 1: Select $\Delta t$

    /* First, the electric field is calculated on a grid.  Then the
    grid can be used to interpolate the field values onto the particle
    positions.  */
 2: $E \leftarrow$ runFieldSolver($f$)
 3: pairs $\leftarrow$ selectCollisionPairs($f$)
 4: for Every Particle Pair $(i, j)$ do
 5:     $\Delta v \leftarrow$ getCollisionUpdate($i$, $j$)

        /* Advance every particle by $\Delta t$, assume $m_a = m_b$.  */
 6:     $v_{i,j} \leftarrow v_{i,j} \pm \frac{\Delta v}{2}$
 7: end for

    /* Do a leap-frog step (every operation performed particle-wise).  */
 8: $v \leftarrow v + \frac{\Delta t}{2} \cdot \frac{e}{m} \cdot \frac{E}{\epsilon_0}$ // Kick 1
 9: $x \leftarrow x + v \cdot \Delta t$ // Drift
10: $v \leftarrow v + \frac{\Delta t}{2} \cdot \frac{e}{m} \cdot \frac{E}{\epsilon_0}$ // Kick 2

    /* Update how particles are distributed onto the computational grid in
    IPPL. */
11: particleNodeDistributionUpdate()
```

This demonstrates that the primary distinction between the algorithms lies in the selection of particle pairs for the collision step and the method used to calculate the post-collision velocity for each pair. These specifics are explained in the following two sections.

2.3 Takizuka and Abe's (1977) Collision Model

Selecting the particle pairs. This algorithm uses all available particle pairs. From the N particles, $\frac{N}{2}$ pairs are selected (assuming an even number of particles), where every particles collides exactly once.

Calculating the velocity update. Next, Δv mentioned in the algorithm 1 is calculated according to [11, p. 4310]:

- Assume two particles i and j with their respective masses, charges, number density n (the same for both particles) and relative velocity $\mathbf{u} = \mathbf{v}_i - \mathbf{v}_j$ and reduced mass m_{ij}. Also:

$$\mathbf{u} = \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix}, \qquad u_\perp = \sqrt{u_x^2 + u_y^2}$$

- Calculate the variance

$$\langle \delta^2 \rangle = \frac{e_i^2 e_j^2 n \ln \Lambda}{8\pi\epsilon_0^2 m_{ij}^2 u^3} \cdot \Delta t$$

  and sample the scattering angle Φ normally distributed around mean 0 and with variance $\langle \delta^2 \rangle$.

- Sample the azimuthal angle Θ uniformly distributed between 0 and 2π.
- Use the angles to calculate the cos and sin:

$$\sin\Theta = \frac{2\delta}{1+\delta^2}, \qquad \cos\Theta = 1 - \frac{2\delta^2}{1+\delta^2}.$$

- Finally use everything to calculate the velocity update:

$$\Delta\mathbf{v} = \begin{pmatrix} \frac{u_x u_z}{u_\perp} \sin\Theta\cos\Phi - \frac{u_y u}{u_\perp} \sin\Theta\sin\Phi \\ \frac{u_y u_z}{u_\perp} \sin\Theta\cos\Phi - \frac{u_x u}{u_\perp} \sin\Theta\sin\Phi \\ -u_\perp \sin\Theta\cos\Phi \end{pmatrix} - \mathbf{u}(1-\cos\Theta).$$

2.4 Nanbu's (1997) Collision Model

Nanbu's algorithm is similar to the previous one, but has one conceptual difference. It is built upon the concept of grouping many small-angle binary collisions together. This approach recognizes that particles in a plasma do not typically undergo isolated, significant collisions. Rather, they experience numerous minor deflections due to long-range Coulomb interactions. By modeling these small-angle collisions, the Nanbu algorithm captures the cumulative effects of multiple interactions over time, theoretically providing a better representation of the particle dynamics. This methodology should enhance the accuracy of the simulation, particularly in non-equilibrium conditions, where the frequent, subtle changes in particle trajectories are critical to understanding the overall behavior of the plasma.

However, as this algorithm has more complicated calculations and it remains to be seen whether it will actually produce a result with the same accuracy as Takizuka and Abe more quickly in the end.

Selecting the particle pairs. The $\frac{N}{2}$ pairs are selected exactly as in 2.3.

Calculate the velocity update. The difference to 2.3 is in the way the angles are sampled. Nanbu is supposed to be an improved version of Takizuka and Abe and uses a different probability density function to sample the collision angles Θ and Φ.

- First calculate a parameter $s \propto \frac{\Delta t}{t_{\text{relaxation}}}$ using the same variable names as before:

$$s = \frac{\ln \Lambda}{4\pi} \left(\frac{2e^2}{\epsilon_0 m} \right)^2 \frac{n \Delta t}{u^3}.$$

- Numerically solve[3] the following equation for A:

$$\coth A - A^{-1} = e^{-s}.$$

- Use A and two uniformly distributed random numbers $U_{1,2} \in [0,1]$ to calculate the angles:

$$\cos\chi = \frac{\ln\left(e^{-A} + 2U_1 \sinh A\right)}{A}, \quad \sin\chi = \sqrt{1 - \cos^2\chi}, \quad \varepsilon = 2\pi U_2.$$

- Calculate the velocity update (careful, here is a different convention in use: $u_\perp^2 = u_y^2 + u_z^2$):

$$\Delta \mathbf{v} = \mathbf{u}(1 - \cos\chi) - \frac{\sin\chi}{u_\perp^2} \begin{pmatrix} -u_\perp^2 \cos\varepsilon \\ u_y u_x \cos\varepsilon + u u_z \sin\varepsilon \\ u_z u_x \cos\varepsilon - u u_y \sin\varepsilon \end{pmatrix}.$$

  Also note that it is different by a minus sign compared to Takizuka and Abe.

This fully describes both algorithms as they were implemented and used for the whole project. How to find the probability distribution for the “grouped angles” is explained in detail in [8, p. 4644].

[3]Numerical root finding is quite computationally intense, which is why a lookup-table is used proved by [8, p. 4644] and the A value is interpolated.

2.5 FFT Field Solver and Implementation

As mentioned earlier, charged particles in the simulation induce a self-consistent electrical field, which is needed to during inhomogeneous initial distributions. To accurately model this phenomenon, we need to solve the Poisson equation, $\Delta\Phi_{\text{pot}} = -\rho$, where Φ_{pot} represents the electric potential and ρ is the (discrete) charge distribution.

IPPL has already implemented an optimized parallel FFT (Fast Fourier Transform) solver for this purpose. This solver is explained in detail in [5, p. 2] and used throughout this project.

The process of solving the Poisson equation using FFT can be summarized as follows:

1. Transform the charge density distribution into Fourier space.
2. The fundamental solution to the Poisson equation with open boundaries is the convolution of the density field with a $\frac{1}{r}$ potential. In Fourier space, this convolution simplifies to multiplication, transforming the Poisson equation into an algebraic equation.
3. Perform an inverse Fourier transform to obtain the potential in real space.

This method is particularly efficient for periodic boundary conditions[4] and can be parallelized effectively.

In practice, the implementation involves several steps. First, the charges are linearly interpolated onto a fixed grid. This step allows for the use of the FFT solver on the grid. Once the solution is obtained on the grid, the values are then interpolated back to the original charge positions. Finally, to obtain the electrical field, we use the calculated potential. This last step is already handled by IPPL.

3 Methodology and Testcases

This section aims to outline the exact methodology behind the different test cases used to evaluate performance and accuracy. The main goal for the first two tests is to see if the algorithms actually solve a given initial distribution correctly, given an already known theoretical result. The last two tests aim to study two interesting real-world examples – given that the first two results confirm that the algorithms work.

Since the algorithms and the sampling is of probabilistic nature, every test case is subject to statistical fluctuations. To ensure that the result is not dependent on those effects, every test case is calculated N times (in the following referred to as "N realizations") and the mean value and standard deviation of these realizations is presented in the corresponding plot.

[4] All implementations in this project use a fixed computational domain with periodic boundary conditions.

Finally, every test uses electron-electron collisions, such that charges and masses are all equal. However, the actual values might change in the code due to different unit systems.

Anisotropic Temperatures Relaxation Test Case.

This test aims to replicate the test described in [11, p. 4312]. Every equation in this test is taken from there. Since this part exists to verify the *collision* models, a spatially homogeneous particle distribution is used. Without self consistent field and without caring for position, the Landau-Fokker-Planck equation becomes

$$\frac{\partial f}{\partial t} = \left(\frac{\delta f}{\delta t}\right)_{\mathrm{c}}, \qquad f(\mathbf{v}, 0) = f_0(\mathbf{v}),$$

which for some f_0 can be solved analytically. That way, the electrical field is approximately homogeneous for enough particles and does not affect the relaxation process. Therefore, the field solver can be turned off to save computation time.

Now regarding how the initial distribution f_0 is sampled. We use the following distribution split in parallel and perpendicular temperature $T_{\parallel,\perp}$:

$$f_0(\mathbf{v}) = \left(\frac{m}{2\pi}\right)^{\frac{3}{2}} \frac{1}{\sqrt{T_\parallel} T_\perp} \exp\left(-\frac{m}{2}\left(\frac{v_\parallel^2}{T_\parallel^2} + \frac{v_\perp^2}{T_\perp^2}\right)\right), \qquad T = \frac{T_\parallel}{3} + \frac{2T_\perp}{3} \tag{3}$$

and $\frac{\mathrm{d}T}{\mathrm{d}t} = 0$, since the collision step conserves kinetic energy. Using $\Delta T = T_\perp - T_\parallel$, we get

$$\Delta T(t) = \Delta T \cdot e^{-\frac{t}{\tau}}, \qquad \tau = \frac{5}{8}\sqrt{2\pi}\,\tau_0, \qquad \tau_0 = \frac{\sqrt{m}}{\pi\sqrt{2}e^4} \frac{T^{\frac{3}{2}}}{n \ln \Lambda}. \tag{4}$$

This allows us to show that ΔT converges to 0 similar to the analytical solution and therefore show that both models work as intended. For that, $T_\perp = 0.01$ and $T_\parallel = 0.008$ are chosen ([11, pp. 4314, 4317]).

Further parameters that are not mentioned in this paper[5] are the initial number density, and how the Coulomb logarithm is calculated. I used the following values:

$$n_0 = 0.768\,\mathrm{eV}^3, \qquad \ln \Lambda = \sqrt{\frac{T}{4\pi n_0}},$$

where T is the total temperature as mentioned in (3) and the simulation domain is set to a value such that the density is given.

Finally, the computation is simplified by normalizing the time to $t := \nu_0 t$, where $\nu_0 = \tau_0^{-1}$. That way, the reference solution is not dependent on the density or $\ln \Lambda$. For a given time step size value $\nu_0 \Delta t$, the number of time steps becomes $\frac{\nu_0 t_{\text{final}}}{\nu_0 \Delta t}$.

[5]This is supposedly the Trubnikov test. However, it turns out that the original paper is almost impossible to get and every paper that conducts this test has incomplete coverage of their initial parameters. Therefore, I cannot directly compare the numerical results and only confirm the phenomenological differences!

These values are chosen, since n_0 in SI units is approximately $10^{18}\,\mathrm{m}^{-3}$, a common value for an electron plasma.

Error in Time Step Size and N Dependence.
This test case works exactly as the one before. The difference is that we look at the convergence error and convergence rate in Δt [11, pp. 4315–4318] as well as the statistical fluctuations due to the number of particles N [11, pp. 4319–4320].

Dirac Distribution Relaxation Problem.
In this test case, the implementation is straightforward: all particle positions and velocities are initialized to zero. However, this presents a challenge, as a field solver is required to convey any initial momentum to the particles. Without this initial momentum, the particles would remain stationary throughout the simulation.

A significant concern arises when all particles are located at a single point, which can lead to complications for the field solver. To mitigate this issue, the mesh sizes are adaptive[6] and a degree of randomness to the initial positions is introduced. For instance, the particle positions can be distributed uniformly within a small range, such as 1/200 of the domain size.

As the simulation progresses, it is anticipated that the self-consistent electric field will begin to influence the particles, leading to collisions that will contribute to their momentum. Notably, significant velocity updates are expected only when the change in velocity $\Delta\mathbf{v}$ falls within a specific range. That is because particle velocities change enough to make $s \propto u^{-3}$ significantly smaller or bigger over time, leading to insignificant (or no) collision updates.

Throughout the simulation, the relaxation processes will be observed, with particular focus on the temperature relaxation, both with and without collisions. The performance of the algorithms can also be compared. Given that many particles are initially clustered together, NANBU's algorithm may have an advantage in the early stages of the simulation, effectively handling the dynamics of the densely packed particles.

Induced Heating of a Cold Sphere.
The objective of this test is to replicate the findings presented in [6, p. 595], specifically regarding the heating of an initially cold sphere. The setup involves a particle sphere with a radius R, which is spatially homogeneous within that sphere. The initial momentum spread needs to be zero to make the sphere "cold". The mean velocity can be chosen freely and therefore set to zero. Consequently, all particle velocities can be initialized to zero.

As the simulation progresses, the self-consistent electric field will drive the particles away from each other. In cases where the sphere is confined, this self-consistent

[6]This means, that after some steps, the mesh for the field solver is changed such that it does not contain unnecessary free space around the particle cloud.

field will effectively “heat up” the sphere. To achieve the desired plasma oscillations presented in [6], collisions among the particles will be necessary as shown in section 4.3.

To confine the particles within the sphere, a force is applied that is equal and opposite to the radial component of the force induced by the self-consistent electric field. This approach ensures that the particles remain within the designated radius while allowing for the dynamics of the system to unfold.

Finally, we look at the exact initial parameters. The natural units of particle physics are used, where the constants $c = \hbar = k_\mathrm{B} = \epsilon_0 = 1$. In this units, the electron charge is $q_e \approx 0.30$, and the electron mass is $m_e \approx 511\,\mathrm{keV}$.

For the simulation setup, the initial radius is set to $R_0 = 17.96\,\mu\mathrm{m}$, with a domain size of $L = 100\,\mu\mathrm{m}$ and $N = 156055$ total particles. These parameters yield theoretical results according to [6, p. 595], including an oscillation period $\tau = 4.3 \cdot 10^{-11}\,\mathrm{s}$ and a final x-emittance value of $\varepsilon^{\mathrm{eq}}_{x,n} = 0.491\,\mathrm{nm}$. The x-emittance is the metric used for analyzing this test case and is calculated using the formula:

$$\varepsilon_{x,\mathrm{rms}} = \sqrt{\langle x^2 \rangle \langle v^2 \rangle - (xv)^2}.$$

To adequately resolve several oscillation periods, the simulation will use 1000 timesteps, with a time step size of $\Delta t = 2.16 \cdot 10^{-13}\,\mathrm{s}$.

Finally, the next section shows the results and the analysis of all four tests for both algorithms.

4 Results and Discussion

4.1 Anisotropic Temperatures Relaxation Test Case

To begin the analysis, we first need to verify that the sampling and relaxation processes are functioning as intended. The initial test will consist of a single iteration using $N = 20000$ particles. The final time is set to $\nu_0 t_{\mathrm{final}} = 5.0$ and the time step size to $\nu_0 \Delta t = 0.15$. The result for both methods is shown in figure 1.

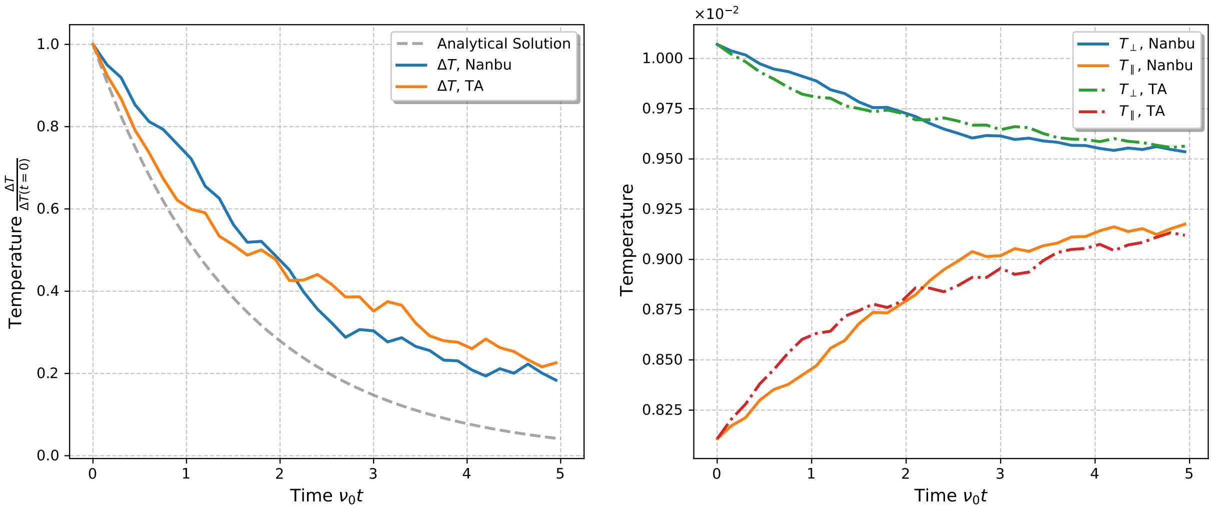

Figure 1: *Temperature evolution using two different algorithms. The left panel shows temperature change* $\Delta T = T_\perp - T_\parallel$ *over time. The dashed line represents the analytical solution* (4)*. The right panel displays the temperature components for each method. The simulation uses* 20000 *particles, a normalized final time of* $\nu_0 t = 5.0$*, and a normalized time step size of* $\nu_0 \Delta t = 0.15$*. Collision computation times were* 0.18 s *for Nanbu and* 0.11 s *for TA.*

As expected, we see that the initial temperature components are correct up to slight statistical fluctuations. More important, we see that both methods methods result in similar behaviour, were the temperatures seem to get closer to a common value.

However, it also becomes apparent that N is by far not big enough to mitigate fluctuations. Because of that, the following analysis will include many realizations of every simulation. That way, we can also look at the “uncertainty” of each plot.

Different time step sizes. The next part investigates the dependence of ΔT on $\nu_0 \Delta t$ for different values. Each simulation is done using 2000 realizations and $N = 3200$ particles. The result is shown in figure 2.

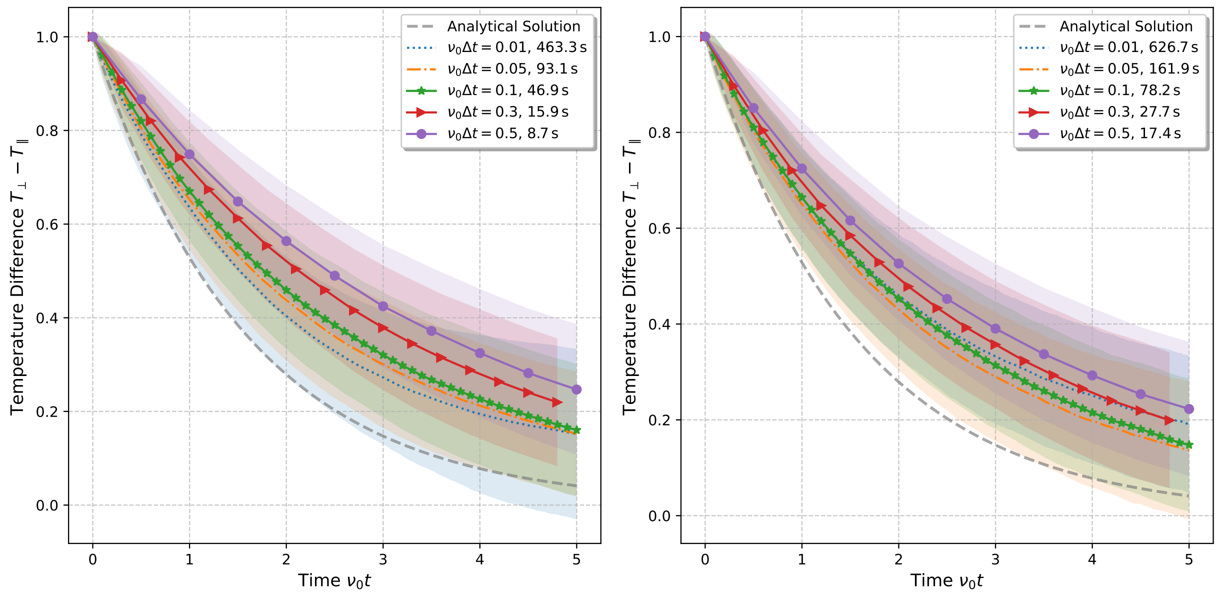

Figure 2: *Temperature evolution for Nanbu's (right) and TA's (left) method for different time step sizes. The simulation uses* 3200 *particles and 2000 different realizations. The shaded region is the standard deviation of the realizations, the graph itself represents the mean. The time in the label denotes the overall computation time for the respective collision steps.*

The result shows two thing. First that the graph seems to converge to some value that is well above the analytical solution and second that smaller step size leads to better accuracy. The first effect is also observable in [11, pp. 4313–4314]. Here, the analytical solution seems to be around ≈ 0.3 at $\nu_0 t = 5$, while all simulations end up well above 0.4. The paper does not mention it at all, but solves this discrepancy by comparing to a very fine calculated solution instead of the analytical graph. For this reason, the subsequent error analysis will be carried out in the same way by comparing to the $\nu_0\Delta t = 0.01$ solution.

Interestingly, the NANBU method seems to converge only up to a specific very small value of $\nu_0\Delta t$ (see blue dotted line). This effect is not representative for this method, but rather shows a computational limitation. According to [8, p. 4645], the A value used to sample the collision angle can be approximated by $A = s^{-1}$ for small s. If the time step size is too small, A becomes very large, which in turn can lead to an overflow in the calculation of the angle $\cos\chi$. As soon as this happens, the algorithm catches that and puts $\Delta\mathbf{v} = 0$ (since $\lim_{A\to\infty}\cos\chi = 1$). However, this might lead to some inaccuracy, since many collisions are missing, even though their effect might be small and the relaxation becomes noticably slower.

Different number of particles. The same experiment as before can be run using $\nu_0\Delta t = 0.2$, 2000 realizations and different number of particles. The result can be seen in figure 3.

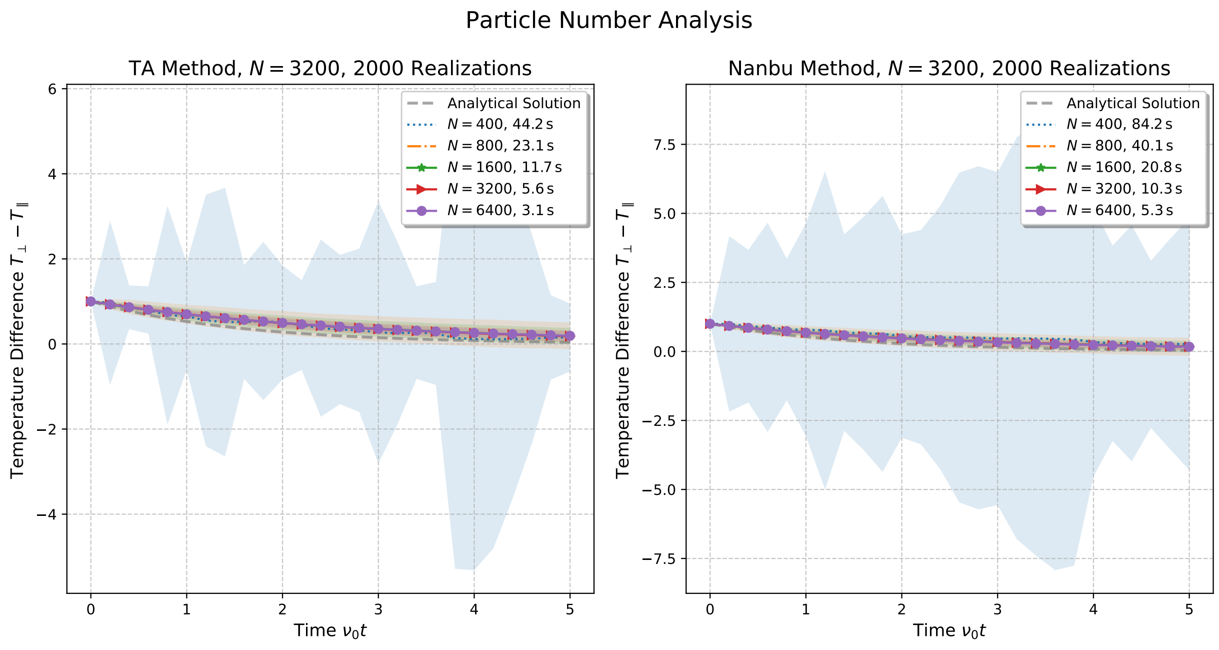

Figure 3: *Temperature evolution for Nanbu's (right) and TA's (left) method for different number of particles. The simulation uses* $\nu_0 \Delta t = 0.2$ *and* 2000 *realizations each. The shaded region is the standard deviation of the realizations, the graph itself represents the mean. The time in the label denotes the overall computation time for the respective collision steps.*

The most notable observation is that the shaded uncertainty is very large for a small number of particles. If we remove it, we get figure 4.

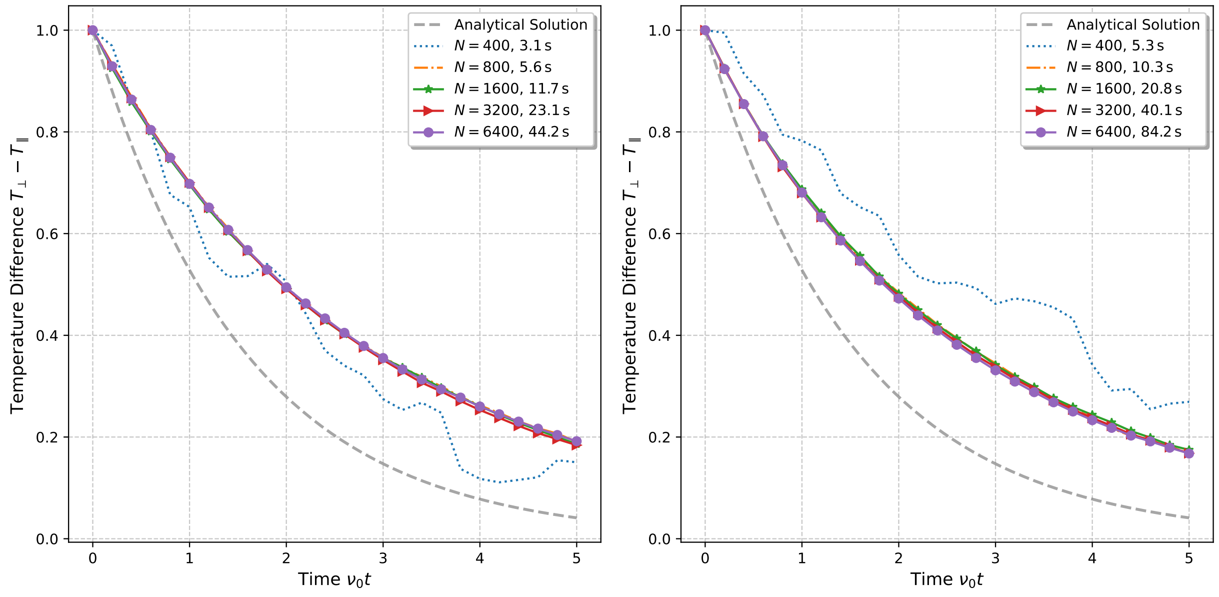

Figure 4: *Temperature evolution for Nanbu's (right) and TA's (left) method for different number of particles without uncertainty. The parameters are the same as in figure 3.*

We see that the number of particles not seem to influence the accuracy of the solution, only the precision. This directly contradicts the expectation set by [11, p. 4317]. The difference could be due to the fact that the density is set to be constant in figure 3. [11] does not mention anything about the density, so this is just a guess. However, it would make sense, since more particles mean a higher density and $\nu_0 \propto \sqrt{n}$. In that case, $\Delta t \propto \frac{\nu_0 \Delta t}{\sqrt{n}}$ and the time step size decreases when the density is not controlled (the simulation domain remains the same). This then boils down to the exact same effect already observed in figure 2.

Time step size error analysis. Next, we can look at the convergence of the relaxation rate of the mean graph in figure 2. An linear function is fitted to the logarithm of the dataset and the rate is extracted from its slope. This process is repeated for every graph and the difference from the finest solution is plotted against $\nu_0\Delta t$ in figure 5. Note that the blue dotted line in figure 2 of NANBU's method is not considered.

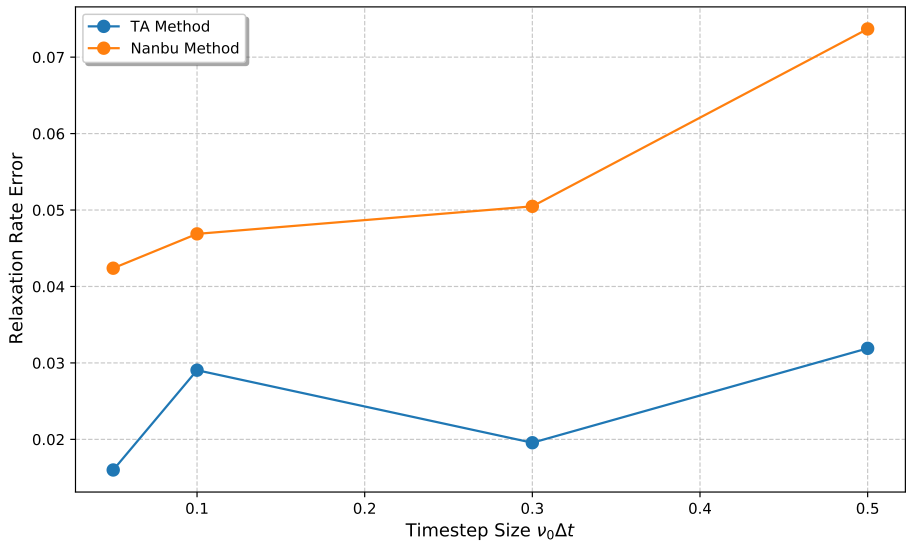

Figure 5: *Plot of the time step size $\nu_0\Delta t$ against the error of the relaxation rate of the mean graphs in figure 2.*

We see that the the error decreases for smaller time step sizes as expected. Fitting a linear function to these two graphs results in a slope of 0.18 for TAKIZUKA and ABE's method and 0.20 for NANBU. These results are not too far off from [11, p. 4316], however, in order to get a real result, one would have to perform many more simulations than just four. The next part will also make it clear that the uncertainty of the relaxation rate is quite high[7].

Uncertainty analysis due to the number of particles. Finally, the standard deviation at every point from the analysis in 2 and 3 can be plotted. The graph is shown in figure 6.

[7]A hint might already be that the shaded regions in 2 all contain all the other mean graphs as well.

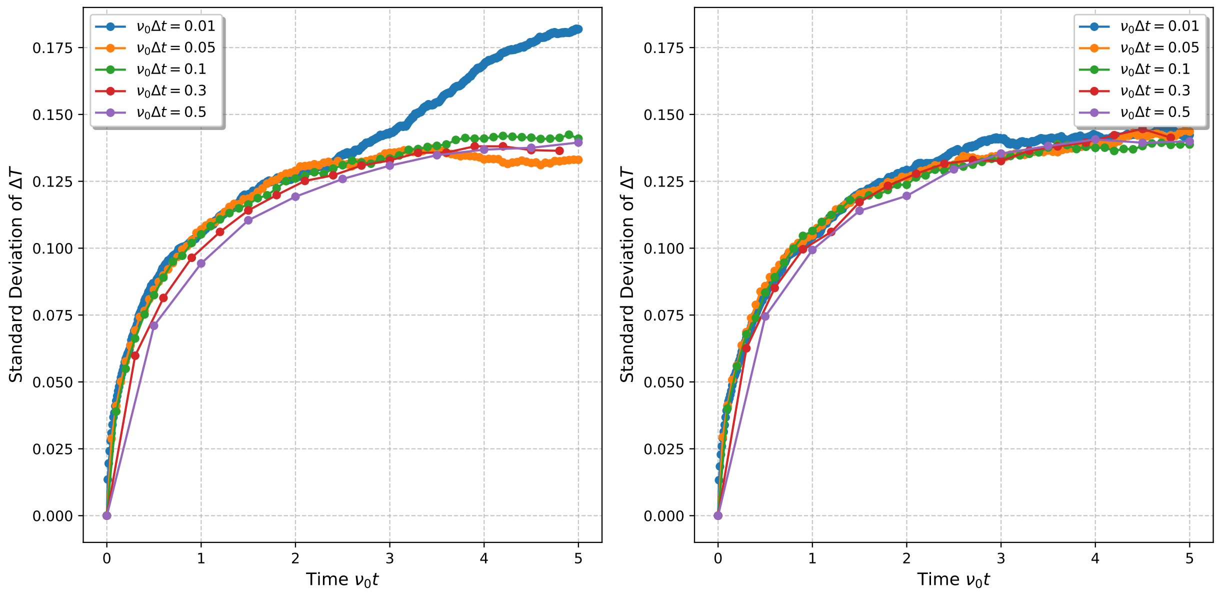

Figure 6: *Nanbu's (right) and TA's (left) plots of the standard deviation of different realizations in figure 2 for different time step sizes.*

You can see that the uncertainty for various values increases over time. The only outlier is $\nu_0\Delta t = 0.01$ in TA's method. This might be a similar effect for very small step sizes as the same step size in NANBU's method, which – as explained before – converges slower due to missing collision computations. However, since this seems to be an outlier, it is not further investigated here.

The same can now also be done for the different number of particles and can be seen in figure 7. Since the blue shaded area is much bigger and not smooth in figure 3, we can conclude that 2000 realizations are far too less to get a meaningful average. Because of that, we can ignore $N = 400$ in this plot.

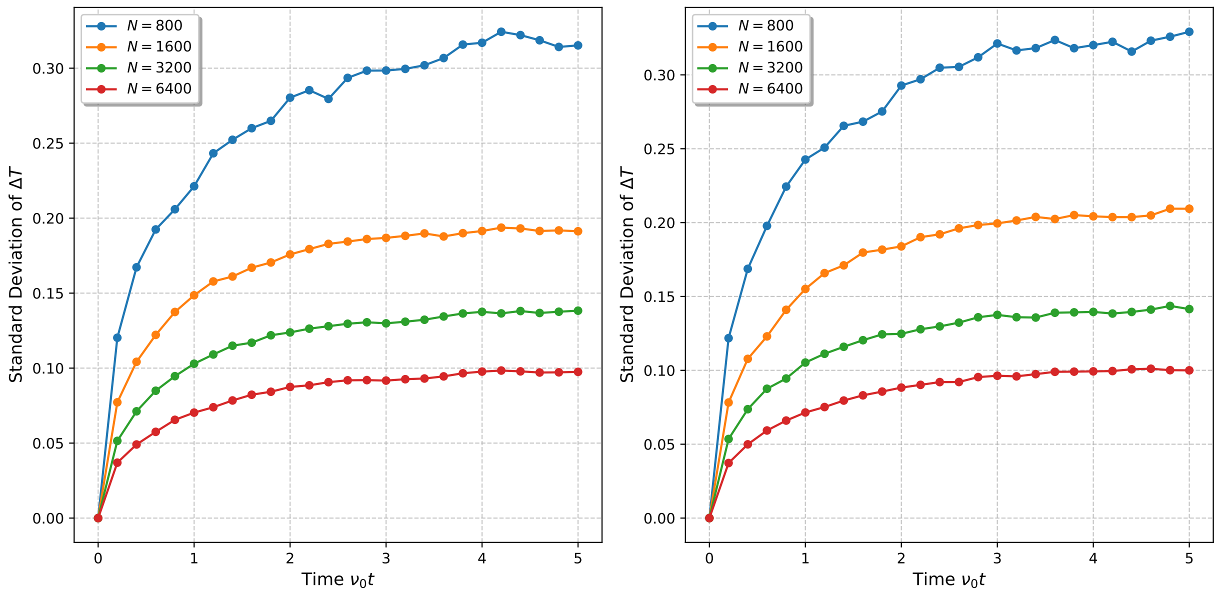

Figure 7: *Nanbu's (right) and TA's (left) plots of the standard deviation of different realizations in figure 3 for different number of particles.*

This graph also very much looks like the expected result in [11, p. 4320]. These two figure confirm that the uncertainty depends on the number of particles N, while the accuracy of the simulation mostly depends on the time step size $\nu_0 \Delta t$.

4.2 Dirac Distribution Relaxation Problem

Now that we established that both algorithms converge for small $\nu_0 \Delta t$, we can consider a more complicated test case and possibly compare the results for both algorithms. Given that the simulation domain is a cube with periodic boundary conditions and the initial velocities are zero, we have an initial temperature $T_0 = 0$ and can calculate the expected equilibrium temperature T_{eq}.

Equilibrium temperature. As mentioned in 3, the positions are sampled with slight deviation from the center of origin to not run into the problem of infinite potential energy. The total potential energy of N homogeneously distributed charged particles inside a cube of length d_x can be calculated. With particle charge q, the potential energy between two particles separated by r is given by $U = \frac{q^2}{4\pi r}$ (natural units). The average distance in a cube is given by $d_{\text{avg}} \approx 0.66170 d_x$ according to [12]. The number of unique possible pairs is $\binom{N}{2} = \frac{N(N-1)}{2}$. Therefore, we get

$$U_0 = \frac{N(N-1)q^2}{8\pi d_{\text{avg}}} \approx 6.0131 \cdot 10^{-2} \cdot \frac{N(N-1)}{d_x}$$

using $q = 1$. If L is the simulation domain length and we know that the system relaxes to a spatially homogeneous distribution, we can calculate the total potential energy that is converted to kinetic energy:

$$E_{\text{kin,eq}} = U_0 - U_{\text{eq}} = 6.0131 \cdot 10^{-2} \cdot N(N-1) \cdot \left(\frac{1}{d_x} - \frac{1}{L}\right).$$

Using $E_{\text{kin}} = \frac{m}{2} N \langle \Delta \mathbf{v}^2 \rangle$ and $m = 1$ and we get

$$T_{\text{eq}} = \frac{2}{3N} E_{\text{kin,eq}} \approx 1.3421 \cdot 10^{-2} \cdot (N-1) \cdot \left(\frac{1}{d_x} - \frac{1}{L}\right).$$

Adaptive mesh sizing. When the particles are homogeneously initialized inside a cube of size $d_x = \frac{L}{200}$ and a 64 mesh grid, then all particles are inside on single cell drastically reducing the accuracy of the self consistent field solver. A possible solution is to implement an adaptive mesh grid size. This is done by computing the smallest and the biggest coordinate value for every particle and using these as new outer domain boundaries[8]. This operation is performed using a custom `MinMaxReducer` struct that can be called inside a `Kokkos::parallel_reduce`. The outer boundaries are set before calling the field solver and reset after the gathering is done.

We can first test this algorithm with a rather large $d_x = 0.1$ and get figure 8.

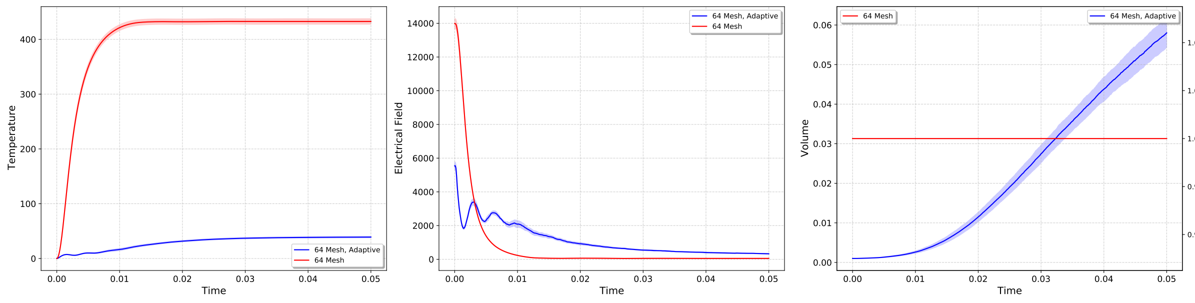

Figure 8: *Plot of the overall temperature (left), the mean field values of the self consistent field (middle) and the occupied domain volume against time (right). All simulations used a grid size of* 64*,* 1000 *particles,* $d_x = 0.1$ *and* 500 *timesteps over 10 realizations each. Both plots were obtained using* Takizuka *and* Abe*'s collision algorithm. The shaded area represents the standard deviation over the realizations.*

There are simply not enough particles per cell to get an usable charge density for the solver. In that case, we see that the mean electric field in the beginning massively overshoots the blue (adaptive) value, which leads to a far accelerated expansion. Interestingly, if we use

[8]Sidenote: There is a problem with this implementation. Since `AlpineManager.h` uses domain size variables to normalize the charge distribution, but the field solver objects uses the field layout as well as the mesh object. If the layout is changed without the mesh, we get badly scaled electric field values. However, adjusting the domain size variables also effects the application of the periodic boundary conditions. This results in a phenomenon where the particle domain becomes smaller and smaller, since the modified boundaries do not allow any expansion. The only way to fix this is to reset these boundary variables before calling the `.update()` routine. This fixes also potential charge conservation errors in multithreaded field solving.

a much smaller $d_x = 0.005$, where – without an adaptive grid – all particles are within one cell, we get figure 9.

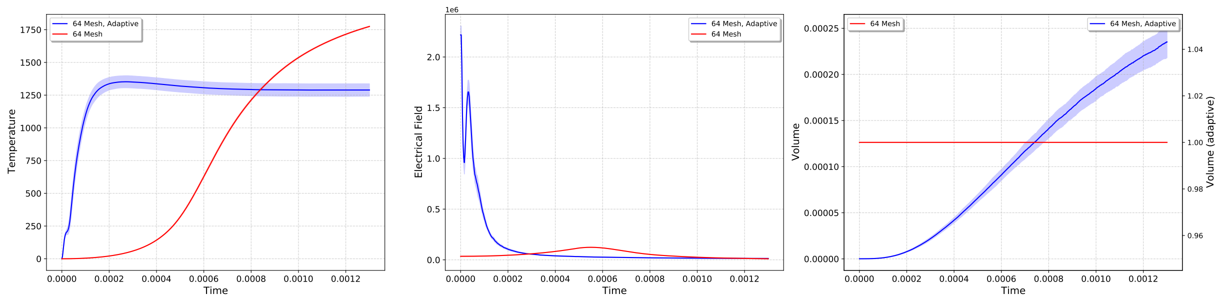

Figure 9: *Plot of the overall temperature (left), the mean field values of the self consistent field (middle) and the set domain volume against time (right), this time using* $d_x = 0.005$*. The rest matches figure 8.*

In this case, exactly the opposite happens and the solver massively underestimates the mean electric field. This is the more reasonable approach, since it actually uses a more "delta like" initial distribution. The reason for this behaviour is that underestimating the electric field leads to a smaller "velocity burst" that should rapidly expand the particle cloud in the beginning. Nevertheless, this E spike appears a bit later (at around 0.0006), which causes the temperature increase. After the cloud is big enough, where the occupied cloud volume is roughly 0.0002, which corresponds to $d_x = \sqrt[3]{0.0002} \approx 0.05$, we see the same behaviour as in figure 8. d_x then roughly matches the initial conditions of figure 8 and the temperature begins to overestimating the adaptive solution, leading to a bigger relaxation temperature. Therefore, it makes sense to only use the adaptive algorithm from now on.

Influence of Collisions. Next, we can plot a test case to compare the effect of collisions. A run with no collisions, with Nanbu and with TA are plotted in figure 10.

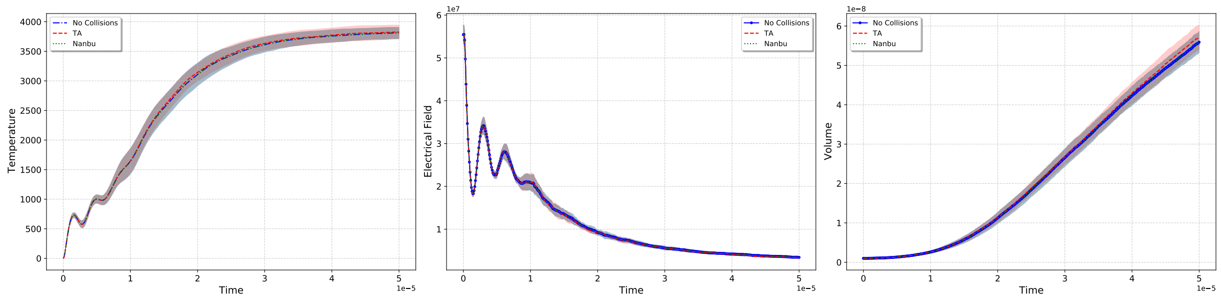

Figure 10: *Temperature (left), mean electric field (middle) and the set domain volume (right) against time. Grid size* 64*,* 500 *time steps,* 1000 *particles and* $d_x = 0.001$ *and 10 realizations each.*

At these conditions, there does not seem to be a big difference between the algorithms. The only notable difference is that TA is always slightly larger in all metrics than Nanbu and no collisions. The insignificance of the collisions comes from the fact that the in the beginning, the electric field is far too strong, which leads to high velocities and therefore suppresses the collision effect. The collisions can be observed only at either very large time steps, which in turn makes gives a large error due to velocities change from the electrical, or from low temperatures. To achieve that, we can either use less particles (reduces the electrical field) or use a rather large d_x. Doing both results in a slow relaxation, where the collisions become significant, see figure 11.

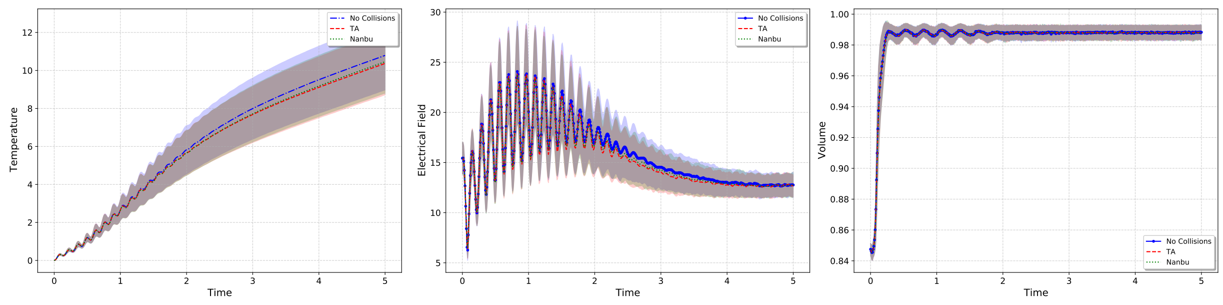

Figure 11: *Temperature (left), mean electric field (middle) and the set domain volume (right) against time. Grid size* 8*,* 500 *time steps,* 500 *particles and* $d_x = 0.95$ *and* 400 *realizations each.*

We see that the collisions significantly – even if the effect is small – lower the relaxation temperature. However, there is no visible effect on the rate of expansion, meaning the graphs of the mesh volume are identical. This makes sense, since this should only be dependent on an overall change in velocity magnitude, which is driven by the electrical field.

The analysis of accuracy and computation time for these collision algorithms appears to be of limited relevance when considering actual delta initial distribution cases, as demonstrated in figure 10. Instead, it might be more fruitful to explore other aspects of the simulation, such as grid size and step size, while disregarding collisions altogether.

Influence of Mesh Grid Size. To study the effect of grid size, we consider a simulation with $d_x = 0.001$ and focus on the initial rapid expansion. The result can be seen in figure 12.

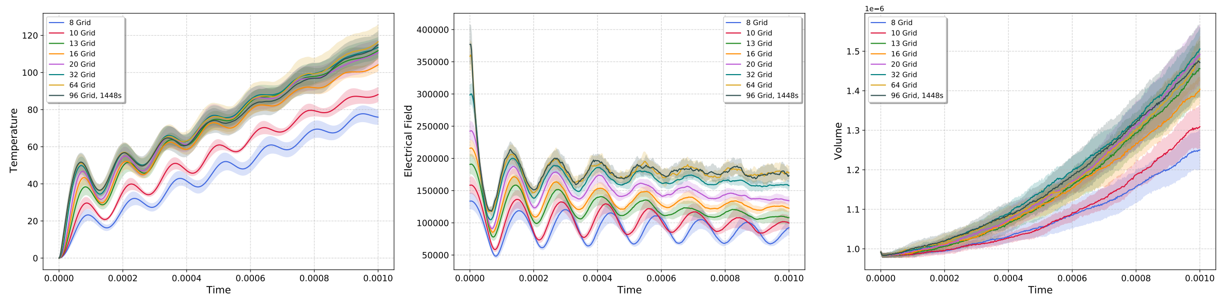

Figure 12: *Temperature (left), mean electric field (middle) and the set domain volume (right) against time for different grid sizes. Each simulation uses* $d_x = 0.01$*,* $N = 500$ *particles,* 400 *time steps and* 10 *realizations. The shaded region is the standard deviation over these realizations.*

Generally, one can observe that the mean electric field values increase with the grid size.

The reason for that is likely that there are not enough particles, which leads to possibly unoccupied cells. This also becomes apparent after some time for grids with size over 20, since the plot of the electrical field is not smooth anymore and the standard deviation increases.

As discussed before, the higher field values lead to higher velocities and therefore faster temperature increase and faster space expansion. Interestingly, there does not seem to be much of an accuracy increase after grid size 32 in terms of temperature. A possible explanation is the overall linear upwards trend of the temperature. The overall increase in kinetic energy (which comes from the potential energy of the self consistent electrical field) overshadows the small accuracy differences that are still visible at $t \approx 0.0001$.

Influence of Step Size. With the same simulation parameters as before, we get figure 13.

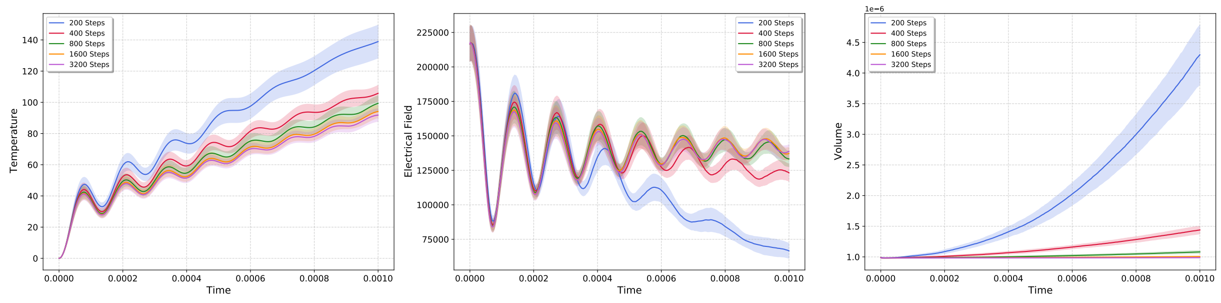

Figure 13: *Temperature (left), mean electric field (middle) and the set domain volume (right) against time for different numbers of time steps. Each simulation uses $d_x = 0.01$, $N = 500$ particles, a grid size of 16 and 100 realizations. The shaded region is the standard deviation over these realizations.*

The number of time steps (and therefore the time step size) directly influences the accuracy of the simulation. Even though, the temperature or electric field differences are not that far off for other time step sizes, the occupied grid volume seems to be affected a lot. This means that the particle cloud expands much faster for bigger step sizes, which might even lead to energy conservation issues when simulating for longer times.

Finally, we can compare the graphs against the "fine solution" with 3200 time steps and see how fast it converges (figure 14).

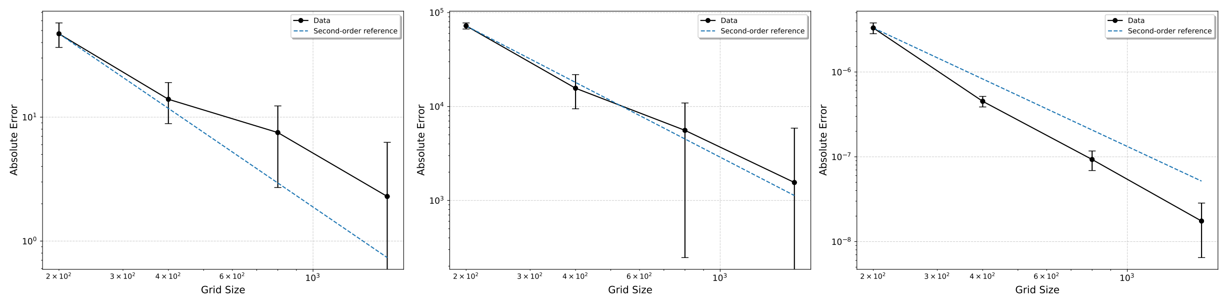

Figure 14: *Error plot against number of time steps for the graphs in figure 13.*

All of these metrics are quite close to a second order convergence. This is to be expected, since the algorithm uses a leapfrog step to advance particle velocity and position.

In conclusion, this test case was not ideal for comparing the TAKIZUKA & ABE and NANBU methods; a more appropriate scenario is still the TRUBNIKOV test, as discussed in section 4.1. Despite this limitation, some convergence behavior was observed, indicating that the results still hold value. Additionally, the findings confirmed the effectiveness of adaptive grid sizing, which proved to be particularly beneficial for this type of test.

4.3 Disorder Induced Heating of a Cold Sphere

Using the parameters outlined in 3, we want to replicate the results from [6, p. 595]. The paper uses a cutoff radius up to which "collisions" are relevant. This parameter does not exist for our algorithms. However, by choosing different grid sizes, we can effectively get a similar result, since only collisions in one mesh grid cell are considered. The result can be seen in figure 15.

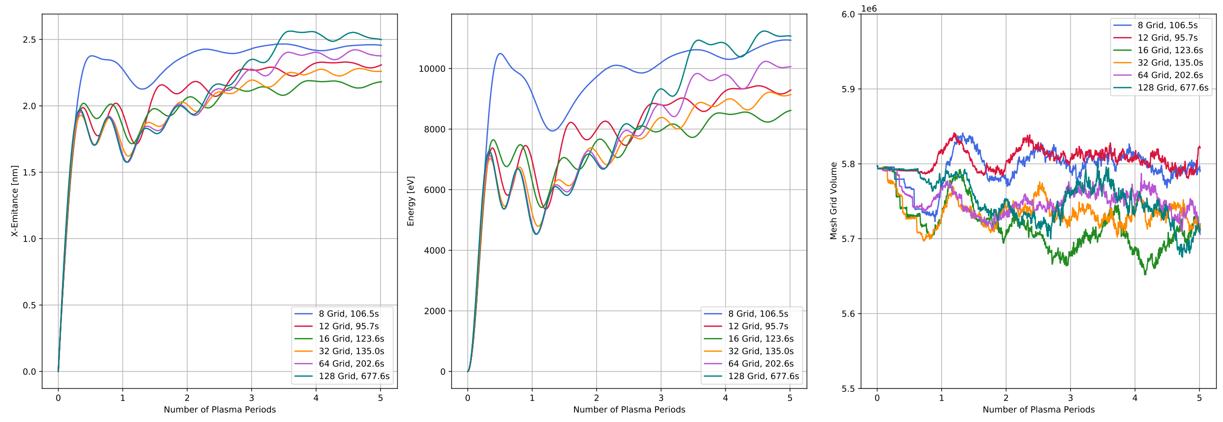

Figure 15: *x-emittance (left), total energy (middle) and mesh grid volume (right) against time for different mesh grid sizes. All simulations were done using only one realization with* TAKIZUKA *and* ABE*'s collision algorithm.*

As one can see, the confinement works well. However, it does not *only* consist of the linear focusing force, but also of reflecting boundary conditions in a sphere of radius R_0. This was necessary, since otherwise the bunch would always expand if the force is slightly to small and shrink if it is too big (which causes the energy to spike).

Looking at the oscillations themselves, we can see that they appear to be of the expected result. A fast FOURIER transform for the 64-grid reveals that the most represented wave period are

$$2.16 \cdot 10^{-11}\,\text{s}, \qquad 6.49 \cdot 10^{-11}\,\text{s}, \qquad 3.24 \cdot 10^{-11}\,\text{s}.$$

The average is roughly $4 \cdot 10^{-11}$ s, when the theoretical result is $\approx 4.31 \cdot 10^{-11}$ s. Even though the simulation does not look as clean as in [6, p. 595], it provides the correct result for the oscillation.

The simulation results appear less clean due to several factors related to adaptive mesh sizing. Notably, even a 32 mesh exhibits a density resolution comparable to that of a 128 mesh without an adaptive mesh, which is why only the 8-grid seems somewhat off in comparison. Discrepancies between each simulation primarily arise from collisions occurring at the beginning of the process. Initially, these collisions have a significant impact; however, as time progresses, the velocities increase, rendering collisions less relevant. Eventually, the field solution becomes the dominant factor after the initial burst, and it is evident that the solutions are very similar during the first oscillation period.

The lack of smoothness in the results can be attributed to scaling issues associated with the adaptive mesh size. Although the same parameters are used, the outcome does not appear as polished as previous results (refer to figure 18), where no adaptive grid was employed. In those earlier results, a constant grid size was maintained, which explains why the 16-grid already begins to “diverge”. Additionally, the previous simulation achieved the correct order of magnitude for the emittance (theory predicts an equilibrium at 0.49 nm), further confirming the scaling issue at play[9].

Of course, there is the other possibility that the electrical field gets underestimated by a too course grid leading to smaller values in the previous results. However, this would mean that the problem should vanish at very fine grids – which it does not seem to do.

Comparing both Collision Algorithms. Finally, we can look at the impact the collision algorithms have, see figure 16.

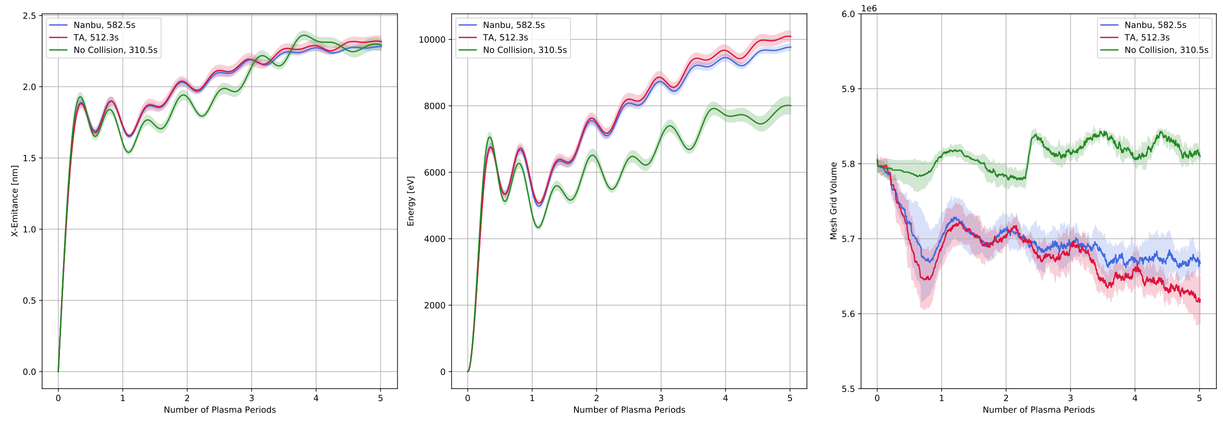

Figure 16: *x-emittance (left), total energy (middle) and mesh grid volume (right) against time for different collision algorithms. All simulations were done using* 5 *realization with* Takizuka *and* Abe*'s collision algorithm. The shaded region denotes the respective standard deviation for the realizations.*

The results align with the findings [6, p. 595], indicating that the absence of collisions

[9]I was not able to fully find the issue. It is likely because `AlpineManager.h` has a lot of different ways to “scale”. For example using the cell volume from the mesh, the field layout or even the variable `rmin_m` and `rmax_m`. I managed to catch a few of those and to change them back and forth everytime I run the solver. However, there still seems to be more of those, which is probably why the scaling is not perfect.

leads to overall lower emittance. However, the differences observed between the various collision algorithms are minimal. As previously mentioned, significant collisions only occur during a brief initial phase of the algorithm, resulting in only slight variations in outcomes. While it is encouraging to see that collisions do influence results compared to scenarios without them, the practical implications are limited. Specifically, the NANBU algorithm required 14% more computation time than the TAKIZUKA & ABE (TA) method, yet the uncertainties in the emittance measurements overlap, indicating no statistically significant improvement in accuracy. It is worth noting that the differences in performance might be more pronounced in scenarios where collisions are consistently significant, such as in the TRUBNIKOV test case discussed in 4.1.

In this implementation, NANBU was never faster than TA, as it involved more calculations without yielding a noticeable increase in accuracy. However, my algorithm implementation also incorporates variations such as "`Nanbu2`"—which performs fewer collisions than the standard NANBU without compromising accuracy (see [2, p. 7])—and "`Bird`'s" algorithm (see [2, p. 8]), which executes the minimum necessary number of collisions. For well-conditioned problems, these two alternatives may outperform the TA method significantly. Nonetheless, exploring these variations is beyond the scope of this project and remains a remnant from the CSP (FS24) lecture project.

4.4 Possible Algorithm Speedup

The computation times for the single threaded collision executions in figure 2 and 3 can be plotted against the time step size and the number of particles, respectively, in figure 17.

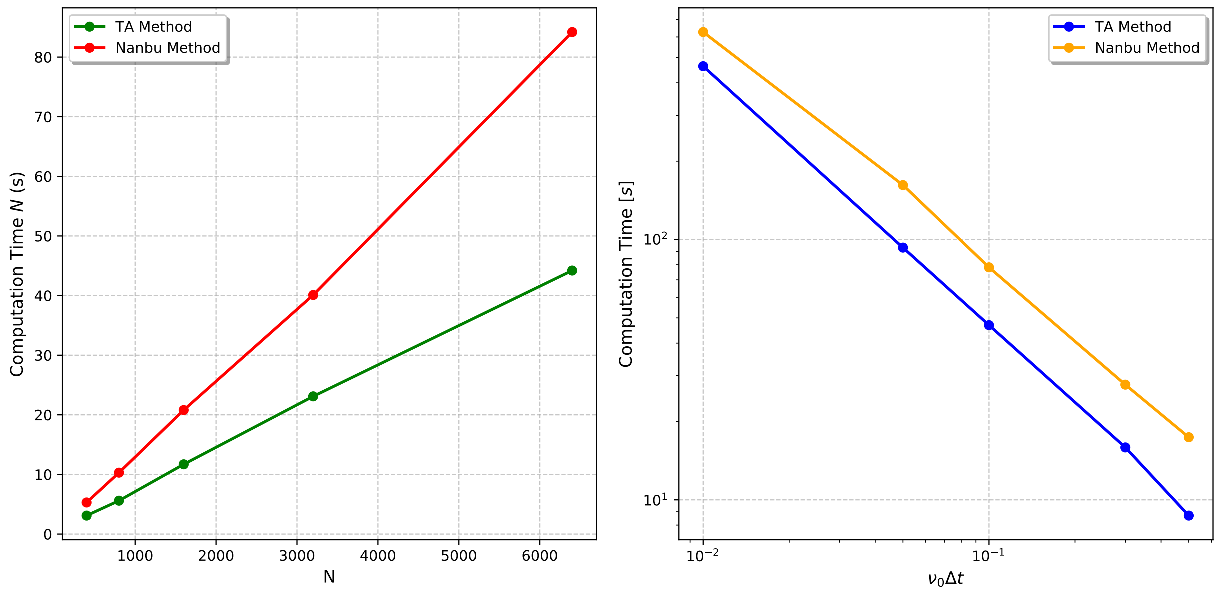

Figure 17: *Computation time plots for varying number of particles (left) and varying time step sizes (right). The right plot has logarithmic axis, the left one is linear.*

One can see that the collision computation time increases linear with the number of particles, which makes sense, since double the particles means double the number of collision function calls. For the logarithmic time step size plot, a linear fit to the values reveals a slope of almost exactly -1. This is also expected, since the number of time steps increases anti proportional to the time step size, while the computation time increases linearly with the number of time steps.

Multithreaded Computation. The whole program can successfully compile using OpenMP as well as CUDA. However, running the program is potentially limited:

OpenMP

It does compile and I ran it successfully on 4 and on 2 threads[10] and even saw some speedup. Since 4 cores utilizes my CPU 100%, it got slowed down at that point. But from 0 to 2 cores, I got an average of 29% speedup.
However, the multithreading currently does not work together with the adaptive mesh sizing. The problem seems to be what I mentioned before: since the program uses many different methods to "normalize using domain size", it is not easy to change every appearance of things like `.setGridSize()` or `rmax_m`. Currently, the program always crashes during `.accumulateHalo()` (it only ran once or twice by chance through that function – likely because there were no particles in the halo).

CUDA

I managed to compile the code and successfully run it for a few iterations. However, there is inevitably an error stating:

```
cudaDeviceSynchronize() error( cudaErrorIllegalAddress): an illegal
memory access was encountered /home/.../ippl\_build/kokkos/
kokkos-4.1.00/core/src/Cuda/Kokkos\_Cuda\_Instance.cpp:139
```

It seems to already happen during the initial `.pre_run()` before even calling the `initializeParticles()`. Sadly, I was not able to fully run the program yet.

5 Conclusion

The project successfully implements the DSMC method for simulating plasma relaxation, providing insights into the performance of different collision algorithms. The Takizuka & Abe method was found to be more computationally efficient than Nanbu's algorithm without significant loss of accuracy, even though Nanbu is the newer version. However, alternative variations like "Nanbu2" and "Bird" algorithms suggest potential improvements (those are not studied here, but can be used with the provided code [4]), which were beyond this project's scope. The scalability of the implementation is promising, although challenges remain in optimizing CUDA-based computations due to current limitations.

[10]This is due to the limitation of my laptop. Euler was at the time not available, since some required dependencies were removed after a operating system change.

To further the study, it would be beneficial to expanding the dataset for more comprehensive error analysis. Especially the TRUBNIKOV test would benefit from more data points. Another important improvement would be the revision of the domain customization to avoid inconsistencies and possibly take full advantage of multi threading.

Overall, this means that there is still room for improvement in the code. For the most part, however, the test cases were successfully executed and solved, which allowed the algorithms to be examined sufficiently.

References

[1] Wikipedia contributors. *Plasma propulsion engine.* Accessed: 2024-07-20. 2024. URL: https://en.wikipedia.org/wiki/Plasma_propulsion_engine.

[2] Giacomo Dimarco, Russell Caflisch, and Lorenzo Pareschi. *Direct simulation Monte Carlo schemes for Coulomb interactions in plasmas.* 2010. arXiv: 1010.0108. URL: https://arxiv.org/abs/1010.0108.

[3] Richard Fitzpatrick. *Coulomb Logarithm.* Accessed: 2024-07-20. 2024. URL: https://farside.ph.utexas.edu/teaching/plasma/Plasma/node39.html.

[4] Alexander Liemen. *FS24 Semester Project.* GitHub repository. The repository to my semester project in computational physics. 2024. URL: https://github.com/aliemen/FS24-Semester-Project.

[5] Sonali Mayani et al. *A Massively Parallel Performance Portable Free-space Spectral Poisson Solver.* 2024. arXiv: 2405.02603 [physics.comp-ph]. URL: https://arxiv.org/abs/2405.02603.

[6] C.E. Mitchell and J. Qiang. “A Parallel Particle-Particle, Particle-Mesh Solver for Studying Coulomb Collisions in the Code IMPACT-T”. In: *Proc. 6th International Particle Accelerator Conference (IPAC’15)* (Richmond, VA, USA). International Particle Accelerator Conference 6. Geneva, Switzerland: JACoW, June 2015, pp. 593–595. ISBN: 978-3-95450-168-7. DOI: https://doi.org/10.18429/JACoW-IPAC2015-MOPMA024. URL: http://jacow.org/ipac2015/papers/mopma024.pdf.

[7] Sriramkrishnan Muralikrishnan et al. “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).* SIAM. 2024, pp. 26–38. URL: https://github.com/IPPL-framework/ippl.

[8] K. Nanbu. “Theory of cumulative small-angle collisions in plasmas”. In: *Phys. Rev. E* 55 (4 Apr. 1997), pp. 4642–4652. DOI: 10.1103/PhysRevE.55.4642. URL: https://link.aps.org/doi/10.1103/PhysRevE.55.4642.

[9] Marshall N. Rosenbluth, William M. MacDonald, and David L. Judd. “Fokker-Planck Equation for an Inverse-Square Force”. In: *Phys. Rev.* 107 (1 July 1957), pp. 1–6. DOI: 10.1103/PhysRev.107.1. URL: https://link.aps.org/doi/10.1103/PhysRev.107.1.

[10] Tomonor Takizuka and Hirotada Abe. “A binary collision model for plasma simulation with a particle code”. In: *Journal of Computational Physics* 25.3 (1977), pp. 205–219. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(77)90099-7. URL: https://www.sciencedirect.com/science/article/pii/0021999177900997.

[11] Chiaming Wang et al. “Particle simulation of Coulomb collisions: Comparing the methods of Takizuka and Abe and Nanbu”. In: *J. Comp. Phys.* 227 (Apr. 2008), pp. 4308–. DOI: 10.1016/j.jcp.2007.12.027.

[12] Eric W. Weisstein. *Cube Line Picking.* Accessed: 2024-07-25. 2024. URL: https://mathworld.wolfram.com/CubeLinePicking.html.

A Appendix

A.1 Reproducibility and General Compile Instructions

The plots are generated using three Jupyter Notebooks, one for each test case. The can be found, together with source files for the code, the final project presentation, this report and some Jupyter Notebooks, in the public GitHub repository [4]. The Notebooks use the data files in the `/data` subdirectory. Command how these are generated, can be found in the repository, and the used parameters can be found in the respective section of this report.
More compile instructions for IPPL and all other source files are also listed in the repository.

I used the following software:

- IPPL version: `V3.0.1`.
- Modules: `gcc/11.4.0`, `cmake/3.26.3`, `cuda/12.2.1`, `openmpi/4.1.4`.
- Tested GPU Architecture: GPX1050, PASCAL61.
- Used CPU: Intel Core i7-7700HQ (4 cores, 8 threads).
- Operating system: Windows 10, IPPL was used in the Linux Subsystem for Windows.

I used the `999-build-everything` script provided by IPPL to compile everything. For example:

```
ippl-build-scripts/999-build-everything -t serial -k -f -i -u
ippl-build-scripts/999-build-everything -t openmp -k -f -i -u
ippl-build-scripts/999-build-everything -g PASCAL61 -t cuda -k -f -i -u
```

More instructions can be found in the GitHub repository with the code under [4].

A.2 Potential Problems and Bugs

A list of potential problems and bugs:

- There is a problem with applying the boundary conditions for very fast particles. When the boundaries are from 0 to L and the particle position is for example $> 2 \cdot L$, then the particle will still not land inside the domain. Instead of calculating the modulo, only "one domain size" is subtracted. I fixed it by implementing my own boundary check in `applyExtendedBC()`.
- The test cases not specifically designed for collisions, such as the delta-initial and cold sphere scenarios, often show minimal impact from collision implementations. This may be attributed to the primary purpose of collision terms: to alleviate the computational burden of fine grids in the field solver. In practice, a coarser grid combined with collisions should in theory yield comparable results. The adaptive

grid's effectiveness in resolving fields for in space uniformly distributed particles further diminishes the collision effects on relaxation, since the field values are very accurate already. Consequently, most results, barring the TRUBNIKOV test, demonstrate little dependence on collision implementation. However, it is worth noting that collisions become more significant in near-equilibrium simulations, where the equilibrium energy range can be tailored to the relevant spectrum, as illustrated in figure 11. "Relevant spectrum" means the point in which the relative velocity leads to a parameter $s \propto u^{-3}$ (for NANBU) that generates a significant collision update.

A.3 Old Cold Sphere Results

For the presentation in the CSP lecture (FS24), we presented figure 18.

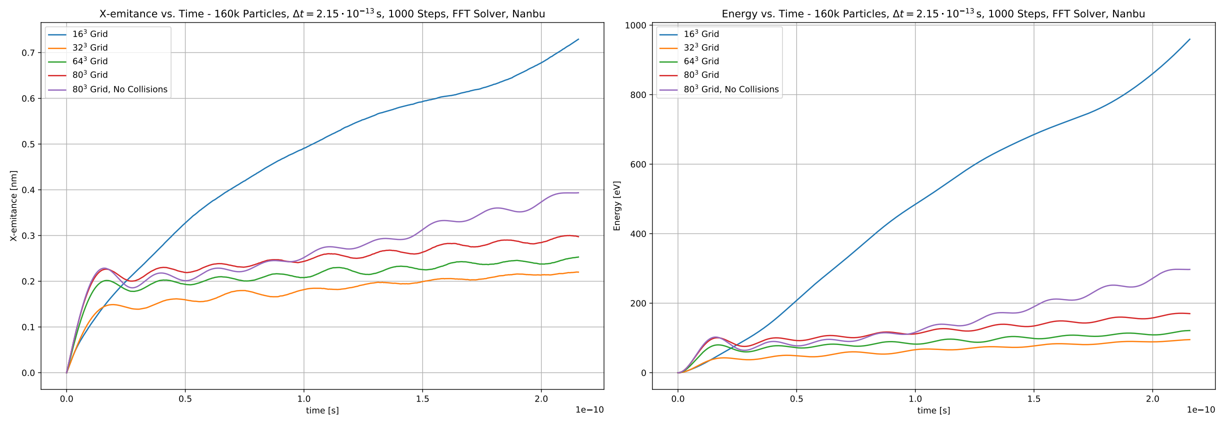

Figure 18: *x-emittance and total energy against time for different mesh grid sizes.*

The simulation was done using 156055 particles and NANBU's collision algorithm.

A.4 Scaling Issue in Adaptive Grid

The analysis mentions a few issues with the scaling of the electrical field when using the adaptive grid for the field solver.

Problem description. Consider the test case of a cold sphere. All particles are distributed in a 18 µm sphere, while the simulation domain has $L = 100$ µm. Therefore, we have a lot of free space around it. If we solve the field over the whole domain, we get a smaller field value than if we adapt the grid to exclude all the necessary free space. Even considering a similar "grid density" to match the number of particles per cell, with and without the adaptive grid, yields the same scaling issue. In this case, the field values lead to an x-emittance that is off by a factor of around 2.5.

I did not find a solution to this problem yet. However, I detailed the exact steps the code takes to adapt the grid every timestep. Inside `advance()`, calculate self-consistent field using:

```
if (this->computeSelfField_m) {
    if (this->adjust_field_dims) adjustFieldMeshDimensions();
    this->par2grid();
    this->fsolver_m->runSolver();
    this->grid2par();
    if (this->adjust_field_dims) resetBoundaries();
}
```

Then adjust grid dimensions, such that every particle is barely in it. After solving and interpolating onto the particles, the boundaries are resetted to the value before (which are hardcoded at the moment according to the corresponding test case).

Now we can look at the actual grid adjustment:

```
void adjustFieldMeshDimensions() {
    std::shared_ptr<ParticleContainer_t> pc = this->pcontainer_m;
    auto *mesh = &this->fcontainer_m->getMesh();
    auto *FL   = &this->fcontainer_m->getFL();

    // Calculate the maximum and minimum of all particle coordinates using Kokkos
    view_type* R                = &(pc->R.getView());
    MinMaxReducer<Dim> minMax;
    findMinMax(R, minMax);
    Vector_t<double, Dim> maxR = minMax.max_val, minR = minMax.min_val;

    // Now figure out componentwise global min/max values
    Vector_t<double, Dim> globalMaxR, globalMinR;
    for (size_t i = 0; i < Dim; i++) {
        ippl::Comm->reduce(&maxR[i], &globalMaxR[i], 1, std::greater<double>());
        ippl::Comm->reduce(&minR[i], &globalMinR[i], 1, std::less<double>());
    }

    // Calculate new mesh spacing
    Vector_t<double, Dim> hr = (globalMaxR-globalMinR) / mesh->getGridsize();

    // set the origin and mesh spacing of the mesh via
    mesh->setMeshSpacing(hr);
    mesh->setOrigin(globalMinR);

    this->rmin_m = globalMinR;
    this->origin_m = globalMinR;
    this->rmax_m = globalMaxR;
    this->hr_m = hr;

    extLayoutUpdate(FL, mesh);
    pc->update();
}
```

Herem we manually reset `rmin`, `rmax`, recalculate `hr` and update the mesh container.

The layout update is a bit more complicated and given in a new function:

```
void extLayoutUpdate(ippl::FieldLayout<Dim>* fl, ippl::UniformCartesian<T, Dim>* mesh) {
    std::shared_ptr<ParticleContainer_t> pc = this->pcontainer_m;
    std::shared_ptr<FieldContainer_t> fc    = this->fcontainer_m;

    Field_t<Dim>* rho_m   = &(fc->getRho());
    VField_t<T, Dim>* E_m = &(fc->getE());

    rho_m->updateLayout(*fl);
    E_m->updateLayout(*fl);
    pc->getLayout().updateLayout(*fl, *mesh);

    std::get<FFTSolver_t<T, Dim>>(this->fsolver_m->getSolver()).setRhs(*rho_m);
}
```

This might include a few (technically) redundant updates. Additional `updateLayout` calls did not make any differences.

Finally, after the field is solved and interpolated onto the particle positions, the boundaries are resetted:

```
void resetBoundaries() {
    this->origin_m = 0.0;
    this->rmin_m   = this->origin_m;
    if (this->initial_distr == "sphere") {
        this->rmax_m = 506.84;
    } else {
        this->rmax_m = 1.0;
    }
    this->hr_m     = this->rmax_m / this->nr_m;

    std::shared_ptr<ParticleContainer_t> pc = this->pcontainer_m;
    auto *FL = &this->fcontainer_m->getFL();
    auto *mesh = &this->fcontainer_m->getMesh();

    // set the origin and mesh spacing of the mesh via
    mesh->setMeshSpacing(this->hr_m);
    mesh->setOrigin(this->rmin_m);
}
```

This manually resets `rmin`, `rmax` and recalculates `hr`. Then, the mesh container is updated. We do not need to update `rho_m`, since its layout is updated in the next call of `adjustFieldMeshDimensions()`, which is before the next run of the solver.