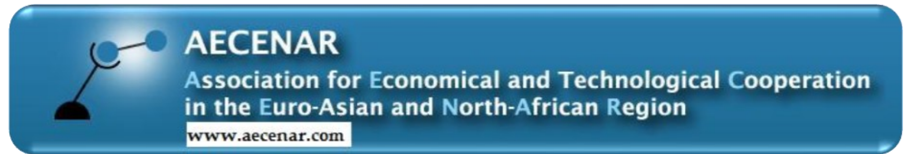
بسم الله الرحمن الرحيم



MASTER THESIS

In Order to Obtain the

RESEARCH MASTER

in

Physics of the Radiation-Matter Interaction

Presented and defended by:

Abdul Rahman Ibrahim

On Friday, October 30, 2020

<u>Title</u>

Laser Based Flue Gas Detection:

Integration and tuning of a laser system for flue gas detection and measurement

Supervisors

Dr. Hassan Amoud

Dr. Samir Mourad

Lebanese University-Faculty of Science

Contents

Acknowledgement 7

RESUME 8

Abstract 9

1 Introduction 10

1.1 Project Environment and Research Starting Point 11

1.2 Task of master thesis 11

2 Laser-plasma interactions 13

2.1 Introduction 13

2.2 kinetic-correlated plasmas 13

2.3 Plasma models for laser-plasma interactions including ultra-short laser pulses and high energy circumstances. 14

I. Static model 14

II. Fluid model 14

III. Kinetic model 15

2.4 Laser-plasma interaction 15

I. Single Electron Dynamics 15

II. Kinetic and Fluid Equations 15

2.5 Simulation of Interaction LASER-Plasma (numerical aspect) 16

I. Summary of numerical method to treat the kinetic model of plasma using Particle-in-Cell (PIC) method with Smilei code 16

II. The Maxwell-Vlasov model 16

III. Quasi-particles and the PIC method 17

IV. Time- and space-centered discretization 18

V. Initialization of the simulation and the PIC loop 19

3 Contribution: The Particle-In-Cell simulation of plasmas: 1D 21

3.1 Introduction 21

3.2 ESPIC 21

I. Core routines 21

II. Loop over time of simulation 23

III. Important parameters. 24

3.3 Results 24

I. Results without B 24

II. Results with B 28

3.4 Conclusion 30

4 Contribution: Laser-Matter Interaction in IAP-PSC code 31

4.1 Improvement in IAP-PSC code without laser-matter interaction 31

4.2 Modified code 32

4.3 Diagrams 33

4.4 Simulation in one dimension 34

4.5 Simulation in three dimensions 36

4.6 Conclusion 36

5 BIBLIOGRAPHY 37

6 LIST OF SYMBOLS 38

7 Annex 39

7.1 IAP_PSC C++ Code 39

7.2 Paraview Input files 66

I. Displaying data as points 69

II. Displaying data as structured grid 70

III. Saving Results 74

Acknowledgement

First, I want to thank Allah the Almighty for all.

Then I cannot express enough thanks to my committee for their continued support and encouragement: Dr. Adnan Naja; Dr. Samir Mourad; Dr. Hassan Amoud; I offer my sincere appreciation for the jury.

The concept for the measurement environment, which is the base of this work, was done by Siham Aisha and Mariam Abdelkarim. I would like to thank them very much for this.

My completion of this project could not have been accomplished without the support of my dad in the first place, my family, and my classmates. Thank you, Mum, you have been my primary supporter until I completed my higher education.

Finally, to my caring, loving, and supportive wife, Shaymaa: my deepest gratitude. Your encouragement when the times got rough are much appreciated and duly noted. My heartfelt thanks.

RESUME

L’objectif général de ce travail est de simuler numériquement le comportement de plasma dans le cadre d’un modèle statique, sans ou avec un champ électromagnétique externe. Le grand nombre des particules nous ramène à utiliser le modèle Particle in Celle (PIC). Il s’avère que même avec une approche PIC le calcul reste coûteux en termes de resources numériques en 3d. Cependant, le cas d’un gas de plasma unidimensionnel est traité numériquement avec succès et des résultats seront présentés ici où nous avons utilisé les deux langages Python et C++ afin d’effectuer la simulation.

Abstract

The general objective of this work is to numerically simulate the behavior of plasma as part of a static model, without or with an external electromagnetic field. The large number of particles brings us back to using the Particle in Celle (PIC) model. It turns out that even with a PIC approach the calculation is still expensive in terms of numerical resources in 3d. However, the case of a one-dimensional plasma gas is successfully processed numerically and results will be presented here where we used both Python and C ++ languages to perform the simulation.

1 Introduction

Initially, the project is to manufacture a device to generate energy by burning waste (figure 1.1).

The project consists of multiple sections, and one of these sections is a device for measuring the emissions of some gases by laser and its interaction with these gases (figure 1.2).

This step is divided into two parts: the practical side and the theoretical side. The theoretical side, is the study of simulation of the interaction of a laser with the gases emitted that we want to measure. This last side is what we will try to achieve in the Master.

Figure 1.1: Real picture of a waste incineration power generation device

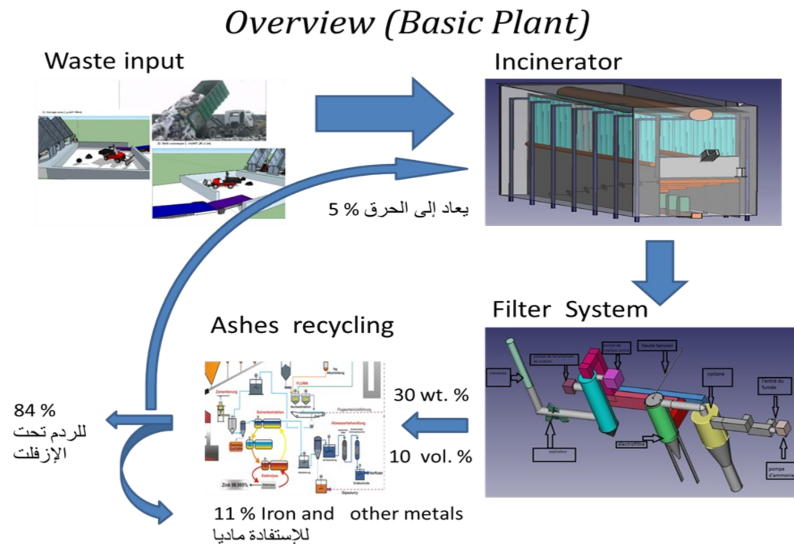


Figure 1.2: basic plant of a waste incineration power generation device.

1.1 Project Environment and Research Starting Point

AECENAR (Association for Economical and Technological Cooperation in the Euro-Asian and North-African Region) has built an incinerator and its filter system. As it is known, this device has many burning residues that will be treated in different ways according to the type of these remains. For the continuation of the waste purification process through filtration systems, it should be measured by the necessary devices.

The Gases are one of the incinerator emissions that must be treated, and to treat it we need to always measure it, so we need a device to measure these gases, to be proved that the concentration of the flue gas is under the norms, i.e. environmentally friendly.

The research committee of AECENAR was decided to use Laser based flue gas measurement technology. To further improve the flue gas measurement procedure, it shall be investigated through simulation, if it is sustainable to make the measurement through laser pulse rather than through continuous wave laser.

The objective of this work is the development of a suitable code of simulating laser-matter interaction, with short pulses, in order to quantify the amount of the flue gas.

1.2 Task of master thesis

First, we began by identifying the appropriate model to understand the interaction laser-plasma, and we confirm, that the Kinetic model of plasma is the most appropriate model for this idea.

After that, we used certain approaches which can simplify the model in order to facilitate the simulation process. The approach used in this project is Vlasov-Maxwell approach. After this step, we reached the stage where we need the numerical method that can achieve these simulations according to the previously selected model and approaches, which was (PIC). In conclusion, we have performed successfully a simulation for a 1d plasma gases by solving Maxwell's equations. In the 3d cases we have written a C++ code based on the Yee grid.

However, we can not compile this code due to a buffer overflow problem. A super computer will be necessary to overcome this problem.

In chapter 2, we briefly mentioned some theoretical information about the plasma, and then we identified the necessary model to achieve the simulations (Kinetic model) with the necessary approaches (the Maxwell-Vlasov) to the appropriate numerical method (PIC).

Chapter 3 describes the proposed numerical model for the interaction laser-plasma. This model is written in Python. In this chapter, we will display the simulation results and we discuss these results.

In chapter 4, we will present the problems of the code, used by the ACENAR for the simulation of the interaction plasma-LASER. And then, we will discuss the code realized in this project to simulate the solution of Maxwell's equations in the material (plasma). After that, we will present the results with a simplified explanation; using actual computer science methods as class diagrams (for static view of the code) and sequence diagrams (for dynamic view of the code).

2 Laser-plasma interactions

In this chapter, some theoretical background is provided to describe laser-material interaction. Beginning with presenting the existing models to explain the plasma, and choosing the model that achieves the simulations required in the thesis, then we will present the necessary approaches to simplify the laser-plasma interaction. At the end of the chapter, we will present the numerical method that we used to make the simulations.

2.1 Introduction

Plasma is defined as a group of charged particles, for which the electric aspect (macroscopically) is neutral; this means that the number of positive charge particles and negative charge particles are equal. Therefore, at equilibrium, the plasma contains n_e electrons, n_i ions and n_0 neutral per unit volume. All bodies are transformed into plasma when some conditions on temperature and/or density are respected.

About the interaction between the particles, the dominant force is the Coulomb interaction. As for the comparison between the plasma and gases, it should be noted that unlike a gas where the interactions are short range force[1], in the plasma the interactions are therefore long range force[2], this implies a collective behavior of the particles.

2.2 kinetic-correlated plasmas

The plasma is correlated at low temperature or high density, and is kinetic at high temperature or low density. The distinction between kinetic and correlated plasma is done by comparing the Coulomb interaction energy with the kinetic energy:

- Kinetic energy: $U_K=\frac{3}{2}k_bT$ (2.1)
- Coulomb interaction energy: $U_{int}=\frac{Z^2e^2}{4\pi\varepsilon_o d}$ (2.2)

  Where d is the distance between two particles, e elementary charge, ε_0 electric constant Z Charge number, k_b Boltzmann's constant and T temperature.

So when

[1] The collision forces in the case of ideal gases.

[2] The electromagnetic force in the plasma cases.

- $U_K >> U_{int}$ kinetic plasma, ideal gas behavior.
- $U_K << U_{int}$ correlated plasma, the electrostatic forces modify the behavior of charged particles.

Another distinction can be done using the length of Landau r_0 which is defined as follow:

Approach length of two particles of energy k_bT.

$$r_0 = \frac{z^2e^2}{4\pi\varepsilon_0 k_b T} \quad (2.3)$$

So when:

- $r_0 << d$: the plasma is in kinetic status.
- $r_0 >> d$: the plasma is in correlated status.

2.3 Plasma models for laser-plasma interactions including ultra-short laser pulses and high energy circumstances.

When we want to describe a phenomenon physically, we must use a model that takes into account the influencing factors, and must also respect the physical laws.

On the other hand, in choosing the model, we must keep in mind the objective of the study and the results we want to obtain. These same conditions must be applied when choosing a model to explain the interaction laser-plasma.

There are three models of plasma that can explain this interaction in the high-intensity (10^{18} W/cm^2) short pulse (1 *PS*):

I. Static model

This approach is not concerned with the dynamics of the plasma particles, so it is not a good option to explain the interaction laser-plasma. A static model can be used for describing the laser propagation.

II. Fluid model

This model is developed to describe specific situations when a particle-particle collision is the dominant factor.

III. Kinetic model

This model specifies the particle distributions self-consistently. It is commonly used in laser-plasma interaction simulations, and the mostly used numerical method for solving this model is PIC (Particle-in-cell). It follows the evolution of the laser pulse on the short timescale associated with the laser period and simulates motion of charged particles, or plasma accordingly [1].

2.4 Laser-plasma interaction

Exposure of the material to an electromagnetic (EM) field with a power of 10^{21} *W*, focused on a spot (within one micrometre for sub-picosecond systems), leads to intense ionization of the material, that is, its transformation into a plasma. The freed electrons oscillate with $m_e c$ energy (where m_e is the electron mass and *c* is the speed of light).

I. Single Electron Dynamics

We started by talking about the dynamics of one electron, meaning about a charged particle, to simplify the explanation of a system consisting of a large number of charged particles (plasma) for example.

The motion of an electron in a *given* EM field is described by the equations:

$$\frac{d\boldsymbol{P}}{dt} = -e\left(\boldsymbol{E} + \frac{\boldsymbol{v}}{c} \times \boldsymbol{B}\right); \qquad \frac{d\boldsymbol{r}}{dt} = \boldsymbol{v}; \qquad (2.4)$$

Where $\boldsymbol{P}=\boldsymbol{P}(t)$ is the momentum, $\boldsymbol{r}=\boldsymbol{r}(t)$ the position, $\boldsymbol{v}=\boldsymbol{v}(t)=\boldsymbol{P}/m_e\gamma$ the velocity, and the fields are evaluated at the electron position, i.e $\mathbf{E}=\mathbf{E}(r(t), t)$ and $\mathbf{B}=\mathbf{B}(r(t), t)$. By *given* fields we mean that we neglect their self-consistent modification by the motion of the electron.

II. Kinetic and Fluid Equations

After we have chosen the most appropriate model for understanding and simulating the laser-plasma interaction (kinetic model). We will begin by trying to understand and simplify this model, and then choose the best numeric model to convert it into a code that simulates this interaction.

To explain plasma dynamics more comprehensively, it is necessary to know the distribution function $f_a=f_a(r, p, t)$, which gives the density of particles in the phase space (r, p) for all species a (a = e, i) where i for a single ion distribution.

Note that we will not take into account the binary collisions, and for simplicity we neglect any process which may create or annihilation particles. This means that the number of particles of each

species (electrons and ions) is conserved, and the distribution function satisfies a continuity equation in the phase space (the Vlasov equation) [2]:

$$\frac{\partial f_a}{\partial t} + \frac{\partial}{\partial r}(\dot{r}_a f_a) + \frac{\partial}{\partial p}(\dot{p}_a f_a) = 0 \qquad (2.5)$$

Where $$\dot{r}_a = v = \frac{pc}{(p^2+m_a^2c^2)^{\frac{1}{2}}} \qquad \dot{p}_a = q_a\left(E + \frac{v}{c} \times B\right) \qquad (2.6)$$

And m_a is mass of species a (a = e, i), i for a single ion distribution, q_a charge of each species, v is the velocity, p momentum and $\dot{p}_a$ is the the Lorentz force.

The coupling with Maxwell equations for the EM fields **E** = **E**(**r**, t) and **B** = **B**(**r**, t)

occurs via the charge and current densities obtained from f_a:

$$\rho(r,t) = \sum_a q_a \int f_a\, d^3p\,, \qquad J(r,t) = \sum_a q_a \int v f_a\, d^3p \qquad (2.7)$$

Now that we have combined Vlasov's equation with Maxwell's equations (Vlasov-Maxwell), and then we have obtained the basic system on which the kinetic model of laser-plasma interaction is built. In most cases, the PIC method is used to perform this simulation [3].

2.5 Simulation of Interaction LASER-Plasma (numerical aspect)

I. Summary of numerical method to treat the kinetic model of plasma using Particle-in-Cell (PIC) method with Smilei code

PIC is a model developed for fluid dynamics studies, and its approach has many of advantages, like as conceptual simplicity and efficient implementation on massively parallel computers etc…

Smilei is a programme that is used in several range of physics studies, like a relativistic Laser-plasma interaction and astrophysical plasmas etc…

As for the source, Smilei is open-source, object-oriented (C++) particle-in-cell (PIC) code, and its core program is C++ object-oriented programming provides an efficient way of structuring the code.

II. The Maxwell-Vlasov model

After we talked about the best plasma model to explain the laser-plasma interaction in the paragraph 2.3.III. And, we referred to the approximations that we took into account as like as neglect any forces that lead to creat or distroy the particles, and we considered that the collision between particles is negligible. Then we can now apply the model of Vlasov-Maxwell. In talking about the latter, we note that the different species constituting the plasma are described by their respective distribution function f_s (t; **X**;**P**), where s denotes a given species consisting of particles

with charge q_s and mass m_s, and **X** and **P** denote the position and momentum of a phase-space element. The distribution f_s satisfies Vlasov's equation:

$$\left(\partial_t + \frac{P}{m_s\gamma} \cdot \nabla + \boldsymbol{F}_L \cdot \nabla_P\right) f_s = 0 \qquad (2.8)$$

Where $\gamma = \sqrt{1 + \frac{\boldsymbol{P}^2}{(m_s c)^2}}$ is the (relativistic) Lorentz factor, c is the speed of light in vacuum, and

$$\boldsymbol{F}_L = q_s(\boldsymbol{E} + \boldsymbol{v} \wedge \boldsymbol{B}) \qquad (2.9)$$

is the Lorentz force acting on a particle with velocity $\boldsymbol{v} = \boldsymbol{P} \,/\, (m_s\gamma)$.

This force follows from the existence, in the plasma, of collective electric [**E** (t; x)] and magnetic [**B** (t; x)] fields satisfying Maxwell's equations:

$$\nabla \cdot \boldsymbol{B} = 0 \qquad (2.10\text{ a})$$

$$\nabla \cdot \boldsymbol{E} = {}^{\rho}/_{\varepsilon_0} \qquad (2.10\text{ b})$$

$$\nabla \times \boldsymbol{B} = \mu_o \varepsilon_o \partial_t \boldsymbol{E} \qquad (2.10\text{ c})$$

$$\nabla \times \boldsymbol{E} = -\partial_t \boldsymbol{B} \qquad (2.10\text{ d})$$

Where ε_0 and μ_0 are the vacuum permittivity and permeability, respectively.

The Vlasov-Maxwell system of Eqs. 2.8 -2.10 describes the self-consistent dynamics of the plasma which constituents are subject to the Lorentz force, and in turn modify the collective electric and magnetic fields through their charge and current densities as follow:

$$\rho(r,t) = \sum_s q_s \int f_s \, d^3p(t, X, P)\,,$$

$$J(r,t) = \sum_s q_s \int v f_s \, d^3p(t, X, P).$$

Where ρ(r,t) is the charge and J(r,t) is the current densities.

III. Quasi-particles and the PIC method

The "Particle-In-Cell" method owes its name to the discretization of the distribution function f_s as a sum of N_s "quasi-particles" (also referred to as "super-particles" or "macro-particles"):

$$f_s(t, X, P) = \sum_{p=1}^{N_s} \omega_p S\left(X - X_p(t)\right) \delta\left(P - P_p(t)\right) \qquad (2.11)$$

Where ω_p is a quasi-particle "weight", x_p is its position, $\mathbf{P}_p$ is its momentum, δ is the Dirac distribution, and S(x) is the shape-function of all quasi-particles. The properties of the shape-function used in Smilei are given in reference [4].

In PIC codes, Vlasov's Eq. 2.8 is integrated along the continuous trajectories of these quasi-particles, while Maxwell's Eqs. 2.10 are solved on a discrete spatial grid. The spaces between consecutive grid points being referred to as "cells". Injecting the discrete distribution function of Eq. 2.11 in Vlasov's Eq. 2.8, multiplying the result by **p** and integrating over all **p** leads to:

$$\sum_{p=1}^{N_5} \omega_p \boldsymbol{P}_p \cdot \left[\partial_{xp} s\left(x - x_p\right) + \partial_x s\left(x - x_p\right)\right] \boldsymbol{v}_p + \sum_{p=1}^{N_5} \omega_p S\,(x - xp)\left[\partial_t \boldsymbol{P}_p - q_5\left(\boldsymbol{E} + \boldsymbol{v}_p \times \boldsymbol{B}\right)\right] = 0$$

(2.12)

Where we have introduced $\mathbf{v}_p$ = $\mathbf{P}_p/(m_s\gamma_p)$ = dx_p/dt the pth quasi-particle velocity, and γ_p =(1 + $P_P^2/(m^2{}_s))^{1/2}$ its Lorentz factor. Considering all p quasi-particles independently, and integrating over all (real) space x, the first term in Eq. 2.12 vanishes due to the properties of the shape-function, and one obtains that all quasi-particles satisfy the relativistic equations of motion:

$$\frac{d\boldsymbol{x}_p}{dt} = \frac{\boldsymbol{u}_p}{\gamma_p} \qquad (2.13)$$

$$\frac{d\boldsymbol{u}_p}{dt} = r_s\left(\boldsymbol{E}_P + \frac{\boldsymbol{u}_p}{\gamma_p} \times \boldsymbol{B}_p\right) \qquad (2.14)$$

Where $r_s = q_s/m_s$ the charge-over-mass ratio (for species s), $\mathbf{u}_p$ = $\mathbf{P}_p/m_s$ the quasi-particle reduced momentum, and the fields interpolated at the particle position:

$$\boldsymbol{E}_p = \int\, dx S\left(x - x_p\right)\boldsymbol{E}(x) \qquad (2.15)$$

$$\boldsymbol{B}_p = \int\, dx S\left(x - x_p\right)\boldsymbol{B}(x) \qquad (2.16)$$

Note that, because of the finite (non-zero) spatial extension of the quasi-particles, additional cells (called *ghost cells*) have to be added at the border of the simulation domain to ensure that the full quasi-particle charge and/or current densities are correctly projected onto the simulation grid.

IV. Time- and space-centered discretization

Maxwell's equations are solved here using the Finite Difference Time Domain (FDTD) approach [4] as well as refined methods based on this algorithm. In these methods, the electromagnetic fields

are discretized onto a staggered grid, the Yee-grid, which allows for spatial-centering of the discretized curl operators in Maxwell's Eqs. (2.10c) and (2.10d). Figure 2.2 summarizes at which points of the Yee-grid the electromagnetic fields, as well as charge and density currents, are defined. Similarly, the time-centering of the time-derivative in Maxwell's Eqs. (2.10c) and (2.10d) is ensured by considering the electric fields as defined at integer timesteps (n) and magnetic fields at half-integer time-steps (n + 1/2). Time-centering of the magnetic fields is however necessary for diagnostic purposes, and most importantly when computing the Lorentz force acting on the quasi-particles. It should also be noted, that a leap-frog scheme is used to advance the particles in time, so that their positions and velocities are defined at integer (n) and half-integer (n – 1/2) time-steps, respectively [4].

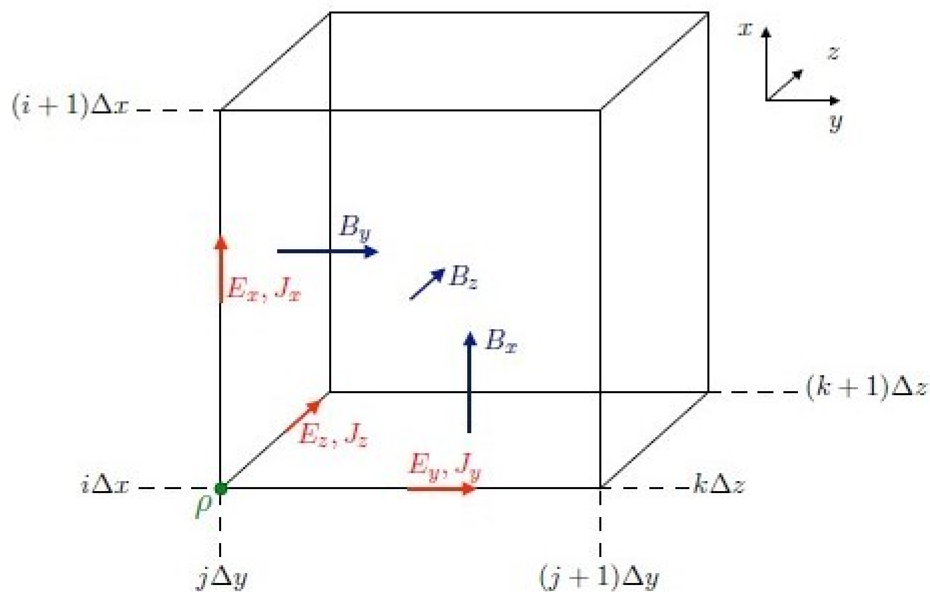


Figure 2.2: Representation of the staggered Yee-grid. The location of all fields and current densities follows from the (rather standard) convention to define charge densities at the cell nodes [4].

V. Initialization of the simulation and the PIC loop

The initialization of a PIC simulation is a three-step process consisting in:

(i) loading particles,

(ii) computing the initial total charge and current densities onto the grid,

(iii) computing the initial electric and magnetic field at the grid points.

At the end of the initialization stage [time-step (n = 0)], all quasi-particles in the simulation have been loaded and the electromagnetic fields have been computed over the whole simulation grid. The PIC loop is then started over N time-steps each consisting in:

(i) interpolating the electromagnetic fields at the particle positions,

(ii) computing the new particle velocities and positions,

(iii) projecting the new charge and current densities on the grid,

(iv) computing the new electromagnetic fields on the grid. (Tableau 2.1).

Table 2.1: summary of SMILEI's PIC algorithm [4].

Initialization	time step $n = 0$, time $t = 0$
Particle loading	$\forall p$, define $(\mathbf{x}_p)^{n=0}$, $(\mathbf{u}_p)^{n=-\frac{1}{2}}$
Charge projection on grid	$[\forall p, (\mathbf{x}_p)^{n=0}] \rightarrow \rho^{(n=0)}(\mathbf{x})$
Compute initial fields	- solve Poisson on grid: $\left[\rho^{(n=0)}(\mathbf{x})\right] \rightarrow \mathbf{E}_{\text{stat}}^{(n=0)}(\mathbf{x})$ - add external fields: $\mathbf{E}^{(n=0)}(\mathbf{x}) = \mathbf{E}_{\text{stat}}^{(n=0)}(\mathbf{x}) + \mathbf{E}_{\text{ext}}^{(n=0)}(\mathbf{x})$ $\mathbf{B}^{(n=\frac{1}{2})}(\mathbf{x}) = \mathbf{B}_{\text{ext}}^{(n=\frac{1}{2})}(\mathbf{x})$
PIC loop: from time step n to $n+1$, time $t = (n+1)\,\Delta t$	
Restart charge & current densities **Save magnetic fields value** (used to center magnetic fields)	
Interpolate fields at particle positions	$\forall p$, $[\mathbf{x}_p, \mathbf{E}^{(n)}(\mathbf{x}), \mathbf{B}^{(n)}(\mathbf{x})] \rightarrow \mathbf{E}_p^{(n)}, \mathbf{B}_p^{(n)}$
Push particles	- compute new velocity $\forall p$, $\mathbf{p}_p^{(n-\frac{1}{2})}$ $\left[\mathbf{E}_p^{(n)}, \mathbf{B}_p^{(n)}\right]$ $\mathbf{p}_p^{(n+\frac{1}{2})}$ - compute new position $\forall p$, $\mathbf{x}_p^{(n)}$ $\left[\mathbf{p}_p^{(n+\frac{1}{2})}\right]$ $\mathbf{x}_p^{(n+1)}$
Project current onto the grid using a charge-conserving scheme	$\left[\forall p\ \mathbf{x}_p^{(n)}, \mathbf{x}_p^{(n+1)}, \mathbf{p}_p^{(n+\frac{1}{2})}\right] \rightarrow \mathbf{J}^{(n+\frac{1}{2})}(\mathbf{x})$
Solve Maxwell's equations	- solve Maxwell-Faraday: $\mathbf{E}^{(n)}(\mathbf{x})$ $\left[\mathbf{J}^{(n+\frac{1}{2})(\mathbf{x})}\right]$ $\mathbf{E}^{(n+1)}(\mathbf{x})$ - solve Maxwell-Ampère: $\mathbf{B}^{(n+\frac{1}{2})}(\mathbf{x})$ $\left[\mathbf{E}^{(n+1)}(\mathbf{x})\right]$ $\mathbf{B}^{(n+\frac{3}{2})}(\mathbf{x})$ - center magnetic fields: $\mathbf{B}^{(n+1)}(\mathbf{x}) = \frac{1}{2}\left(\mathbf{B}^{(n+\frac{1}{2})}(\mathbf{x}) + \mathbf{B}^{(n+\frac{3}{2})}(\mathbf{x})\right)$

3 Contribution: The Particle-In-Cell simulation of plasmas: 1D

3.1 Introduction

In this chapter we will present the programs that we tried to use to study the simulation of a laser-plasma interaction. In the beginning, we preferred the C++ language. Unfortunately, the problem was that the programs operating In the C++ language need a very high-performance computer that was not available in our center in the previous chapter (section 2.5), we presented a PIC model implemented in a C++ program (Smilei). Unfortunately, I could not install and compile this numerical program on our computers. A special environment is required for this task. Therefore, we decided to move to other languages, and the option was the ESPIC code written in Python language. The choice for this program in this language was because it saves the time of calculation and does not consume computer memory. In the following paragraph, I will present the program that I chose to display the PIC results. To achieve this goal, I learned the Python language and downloaded the necessary programs to run the code.

3.2 ESPIC

In this section we present the Python code ESPIC that we use to simulate the behaviour of plasma in a finite one-dimensional grid. First, we present the general structure of this code as well as the important parameter that should be initialised by the user. Next, we present some results obtained by ESPIC with a brief discussion.

I. Core routines

ESPIC is a simple 1D1V electrostatic PIC code [5]. The code is written in the object-oriented language Python. The necessary libraries to run this code are:

— *NumPy [6]*: provides a basic tool to manipulate arrays.

— *PyLab [7]*: allows to plot results during the simulation.

Both, *NumPy* and *PyLab* are open-source Python libraries and can be installed easily on any operating system.

The main routines of ESPIC are:

- ***Loadx:*** This routine allows to initialize the positions of the particles onto a finite one-dimensional grid. In addition, *loadx* supports two type of boundaries: Reflective boundaries and periodic boundaries.

- ***Loadv:*** This routine is used to initialize the particle velocities v_i. By default, the plasma is considered as a cold plasma:

$$v_i=0. \qquad (3.1)$$

The user can also set up a thermal distribution. In this case, the velocities are chosen in term of a random variable θ_i such that:

$$v_i = v_0\sqrt{-2\log\left(\frac{i+0,5}{N_{part}}\right)}\sin(\theta_i) \quad \text{For I = 0, 1, ... , } N_{part}\text{ -1} \qquad (3.2)$$

Where N_{part} number of particles and v_0 is an input parameter.

- **Density:** the electron density is obtained by the positions of electrons on the grid. The ion density is considered to be fixed during the simulation. This approximation is justified by the big difference between the mass of ions and electrons. Compared to electrons, the ions can be considered at rest.
- **Field:** the electrostatic field is calculated by this "field" routine and the magnetic field is neglected in this code. In this case, the electric field is given by the integration over the well-known Gauss's law:

$$\nabla \cdot E = \frac{dE}{dx} = \frac{\rho}{\varepsilon_0} \qquad (3.3)$$

The integration is performed by using the trapezoidal approximation:

$$\int_a^b f(x)\,dx \approx \frac{(b-a)}{2}\big(f(b) - f(a)\big) \qquad (3.4)$$

Which provides a reasonable accuracy if $|b - a| << 1$.

- ***Push:*** Once the electric field is computed on the grid, the routine "push" interpolates this field from grid to particle, and then the new velocities of particles are obtained by solving the equation:

$$V_i = v_i + \frac{q}{m}E_{interp}\Delta t, \qquad (3.5)$$

With V_i the new velocity, v_i the precedent velocity, q and m are the charge and the mass of particles respectively, E_{interp} is the field after interpolation and Δt is the time step of the simulation. Next, the positions xi are updated by

$$X_i = x_i + \Delta t V_i. \qquad (3.6)$$

- **Particle_bc:** This routine is used after every iteration in order to verify that the boundaries conditions are not violated.
- **Diagnostics:** in this part, the electric field, the density, the distribution function (f (v)) and the phase space (v(x)) are plotted every time step. These plots are printed and saved (in *png* extension). Obviously, the number of plots can be chosen easily by the user as well as the time step and the duration of calculation.
- **Histories:** in the last routine, we check if the energy conservation is violated during the simulation. In fact, the kinetic energy

$$K = \frac{1}{2} m \sum_i v_i^2 \, , \qquad (3.7)$$

and the potential energy

$$U = \frac{1}{2} \Delta x \sum_j E_j^2 \, , \qquad (3.8)$$

as well as the total energy

$$E = K + U \, , \qquad (3.9)$$

are saved in the file energies.out and are plotted in energies.png.

II. Loop over time of simulation

The routines explained above are called in a closed loop over the time of simulation as follows:

- Call **loadx** to load particles onto grid (to initialize x).
- Call **loadv** to define velocity distribution (to initialize v).
- Centre positions for the first leap-frog step:

$$x \leftarrow x + \frac{1}{2} dtv. \qquad (3.10)$$

- Call **particle_bc** to apply the boundary conditions.
- Call **field** to compute the electric field.
- Call **diagnostics** to show the results after this initialization.
- Start the main iteration loop
- Call **push** to update x and v.
- Call **particle_bc** to check boundaries.
- Call **density** to compute the new density.
- Call **field** to find the new field.
- Call **diagnostics** to plot and save figures of electric field, density, the distribution function and *v(x).*
- At the end, call histories to check the total energy conservation during the simulation.

III. Important parameters.

Here, we present briefly the principal input (parameters) that should be provided by the user.

- The number of particles: *npart*.
- The number of grid points: *ngrid*.
- The number of steps (over time of simulation): *nsteps*.
- The length of the grid: *grid_length*.
- The time step: dt.
- The type of the field boundary conditions: bc_field = 1 for periodic boundaries and bc_ field = 2 for reflective boundaries.
- The type of the particle boundary conditions: bc_particle (similarly to bc_field).

3.3 Results

I. Results without B

In this section, we present and discuss some results obtained for a cold plasma without a magnetic field **B**, with the initial parameters as follows:

- *npart* = 10000.
- *ngrid* = 100.
- *nsteps* = 100.
- *grid_length* = 2p.
- *dt* = 0.05.
- *bc_ field* = *bc_particle* = 1.

The initialization is illustrated in the (figure 3.1). We show in this figure the density as a function of x. The positive maximum indicates the centre of distribution of ions, and the negative minimum shows the centre of distribution of electrons. As we can see, in the two cases (ions and electrons), the particles are distributed around the corresponding centre of concentration with a normal (Gaussian) distribution. The user can easily modify this choice of distribution by modifying the routine *loadx*. We can see also a small fluctuation which has been added also in *loadx*. We show also the electric field in function of x. As expected, when the density is positive, the field is decreasing and when it is negative, the field is increasing. This observation is predictable since we know that the derivation of E is proportional to $-\rho$. We see also that E (x = 0) = E (x = 2p), which corresponds to the periodic boundaries choice. Since the plasma was taken as cold plasma, the velocities are very small. Finally, we have also plotted the distribution function f (v).

In order to show the evolution of plasma simulation, we plot the results for t_{33} = 33dt, t_{66} = 66dt and t_{99} = 99dt in figure 3.2, figure 3.3 and figure 3.4, respectively. The quality of the curve of f (v) could be ameliorated by increasing the number of grid points.

In the figure 3.5, we show the evolution of potential, kinetic and total energies during the simulation. Of course, the maximum of kinetic energy corresponds to a minimum of potential energy and vice-versa. However, the total energy is not perfectly a horizontal straight line. An important reason for these fluctuations is the integration of Gauss theorem with finite development up to the first term. Nevertheless, the fluctuation of the total energy is relatively small and the global accuracy of the model is very reasonable, especially if we consider the low computational cost needed to run the code.

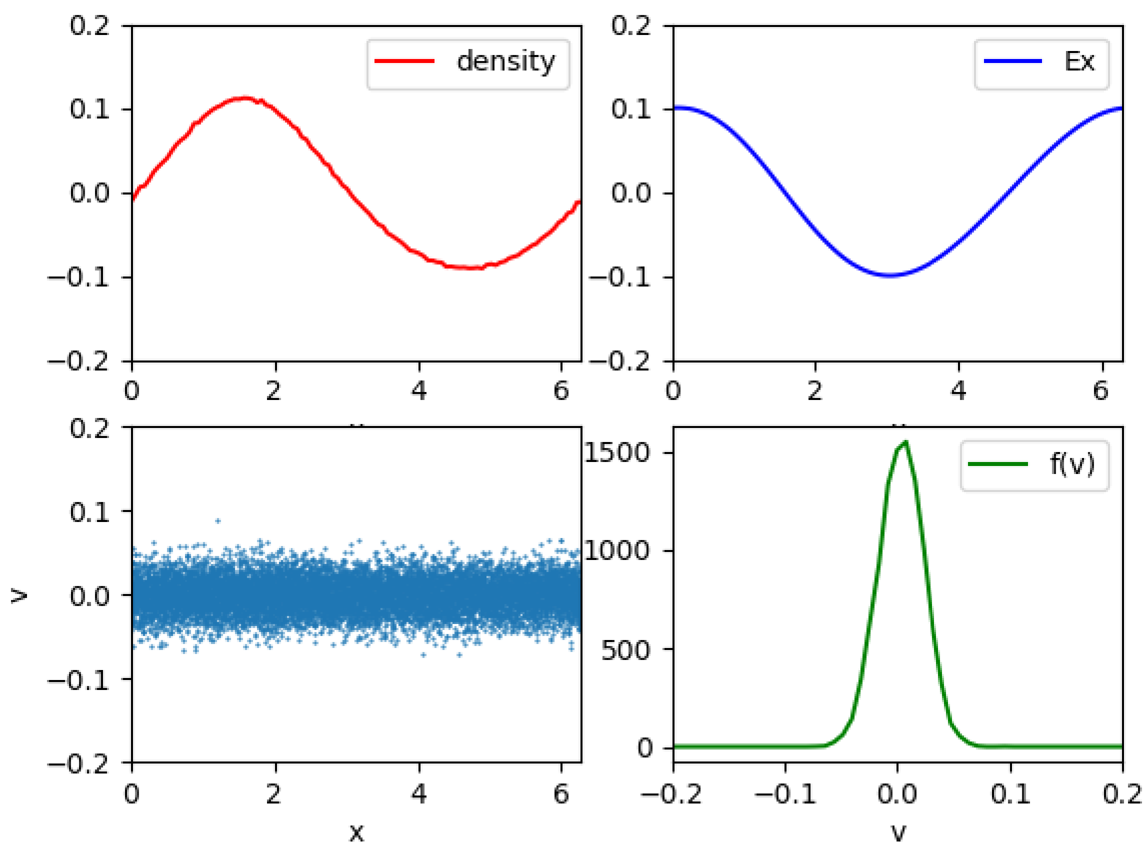


FIGURE 3.1: Results obtained at t_0. Top left: the density of charges as a function of x(m). Top right: the electric field E_x(v/m). Bottom left: the phase space v_x(m/s). Bottom right: the distribution function f (v).

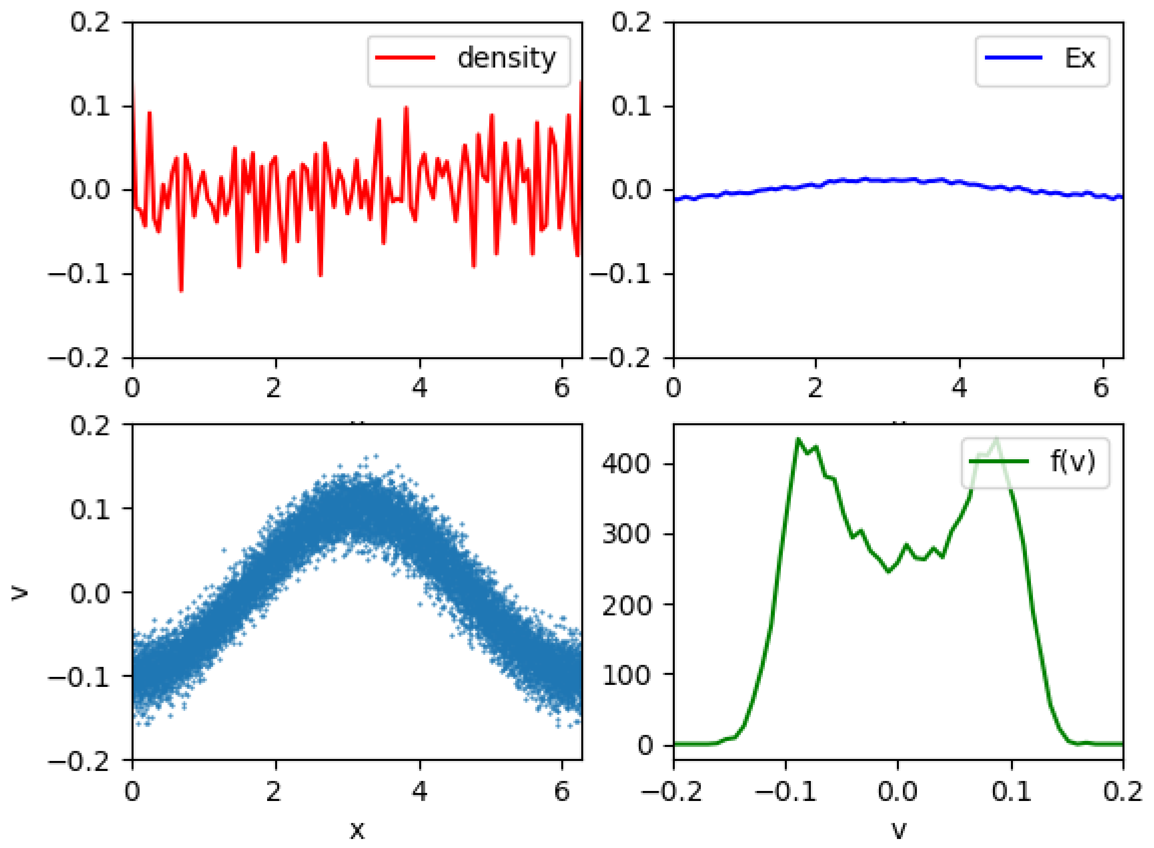


FIGURE 3.2: Similarly to figure 3.1 for t33 = 33dt.

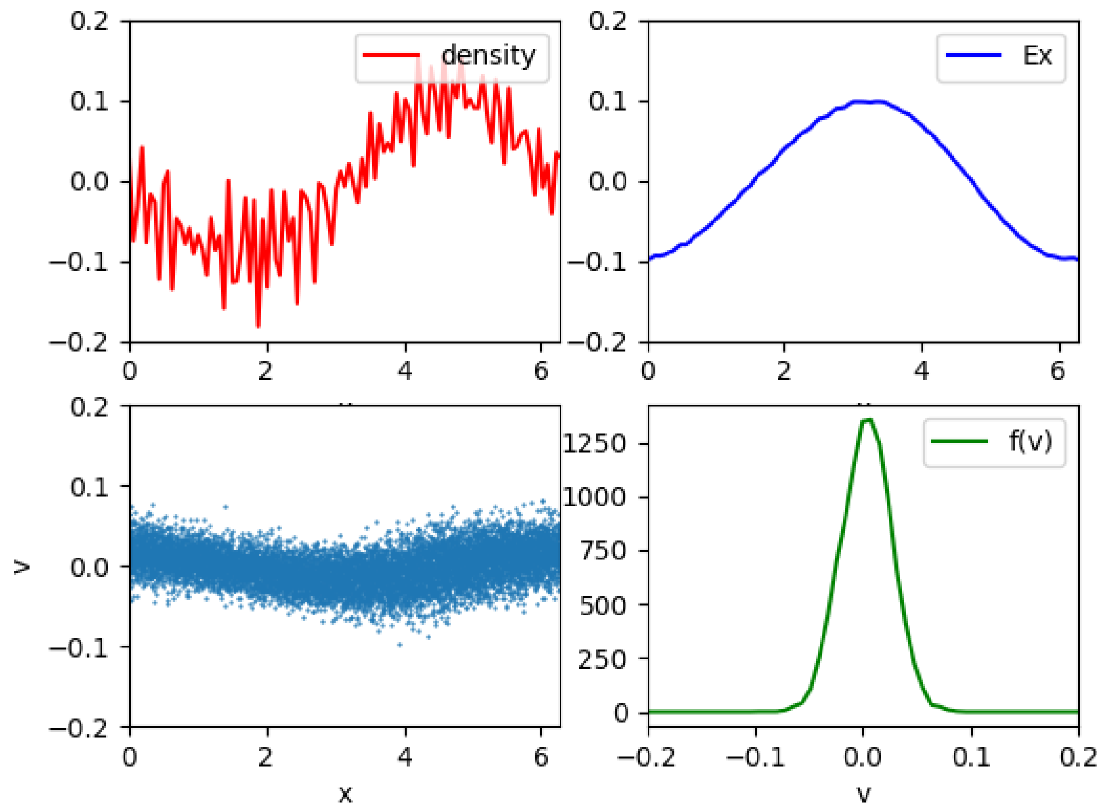


FIGURE 3.3: Similarly to figure 1 for t_{66} = 99dt.

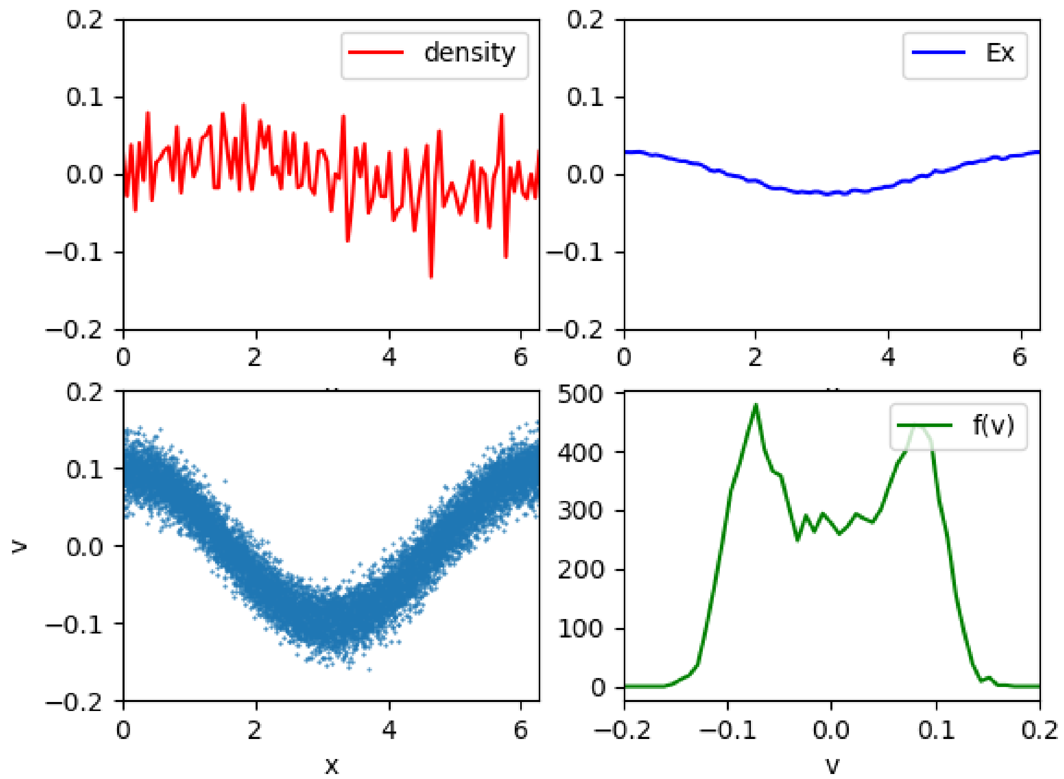


FIGURE 3.4: Similarly to figure 1 for t_{99} = 99dt (end of the simulation).

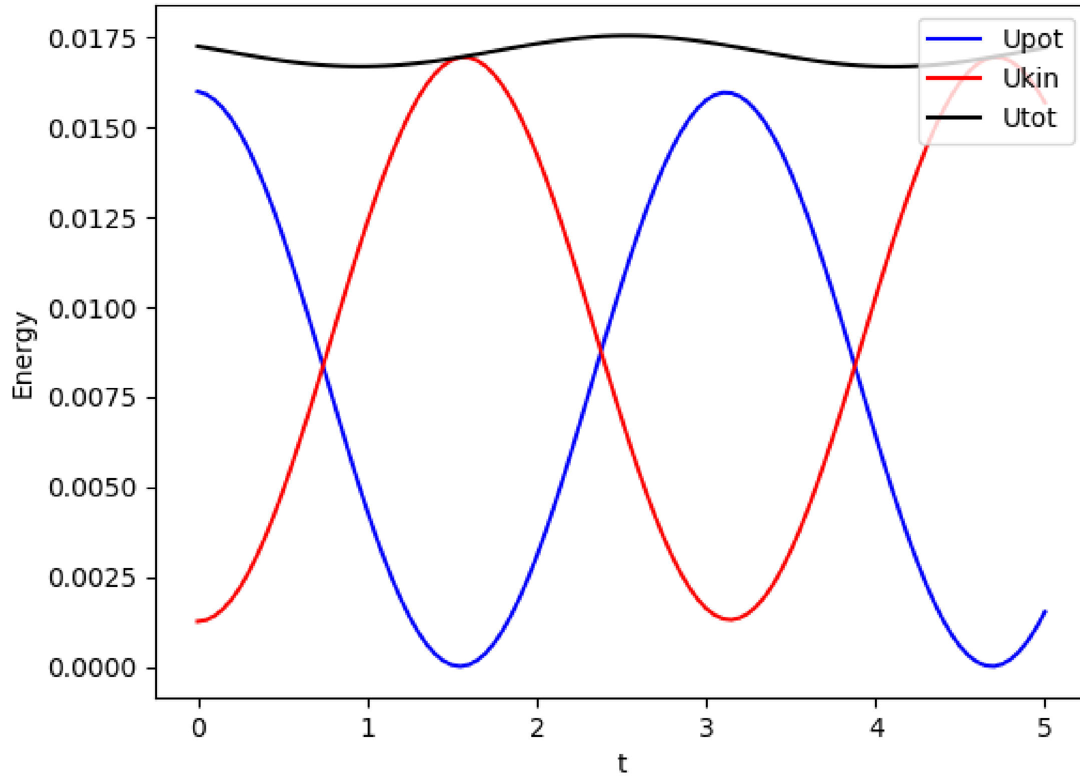


FIGURE 3.5: The evolution of potential (Upot (J)), kinetic (Ukin(J)) and total (Utot(J)) energies during the simulation.

II. Results with B

The existence of the magnetic field ***B*** will cause some changes in the results: for example, because of the particular shape of the magnetic field function, and the relation between ***E*** and ***B*** equ (2.10c), we expect that the modification in the electric field will be approximately a simple shift in the amplitude, and some perturbations will occur in the distribution of particles. On the other hand, the addition if an external source of energy will increase the total energy of the plasma gases.

We have made some modifications to the code and added an external magnetic field, and displayed the results as in the (figure 3.6) and (figure 3.7). To achieve this step, I assumed the existence of a magnetic field with the following function:

$$\boldsymbol{B} = B_0 \mu_0 \varepsilon_0 y \vec{u}_z \qquad (3.11)$$

Where ***B*** External magnetic field, and B_0=0.05 v/m.s.

The first thing that we can notice after adding the magnetic field is the big change in energy. The energy is no longer conserved as it appears in the picture (figure 3.5). The energy has become

increasing (figure 3.7). This is because the plasma is exposed to an external energy source, which is the magnetic field.

As for the electric field E_x, it appears that it was subjected to a shifting proportional to the value of B_0. (Figure 3.6).

With regard to the density and the distribution function, we see that it has changed somewhat due to the interaction of the magnetic field with the charged particles, and this result was expected in (figure 3.6).

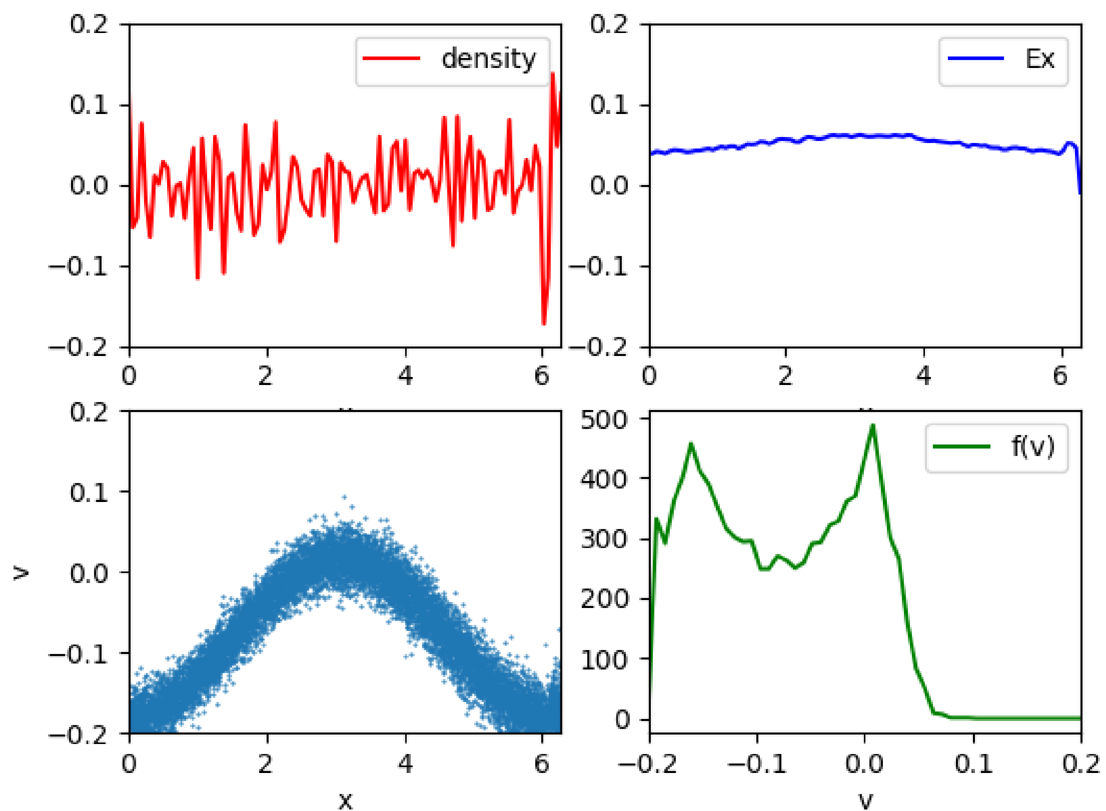


FIGURE 3.6: Similarly to figure 3.2 for t33 = 33dt, after the plasma is exposed to an external magnetic field.

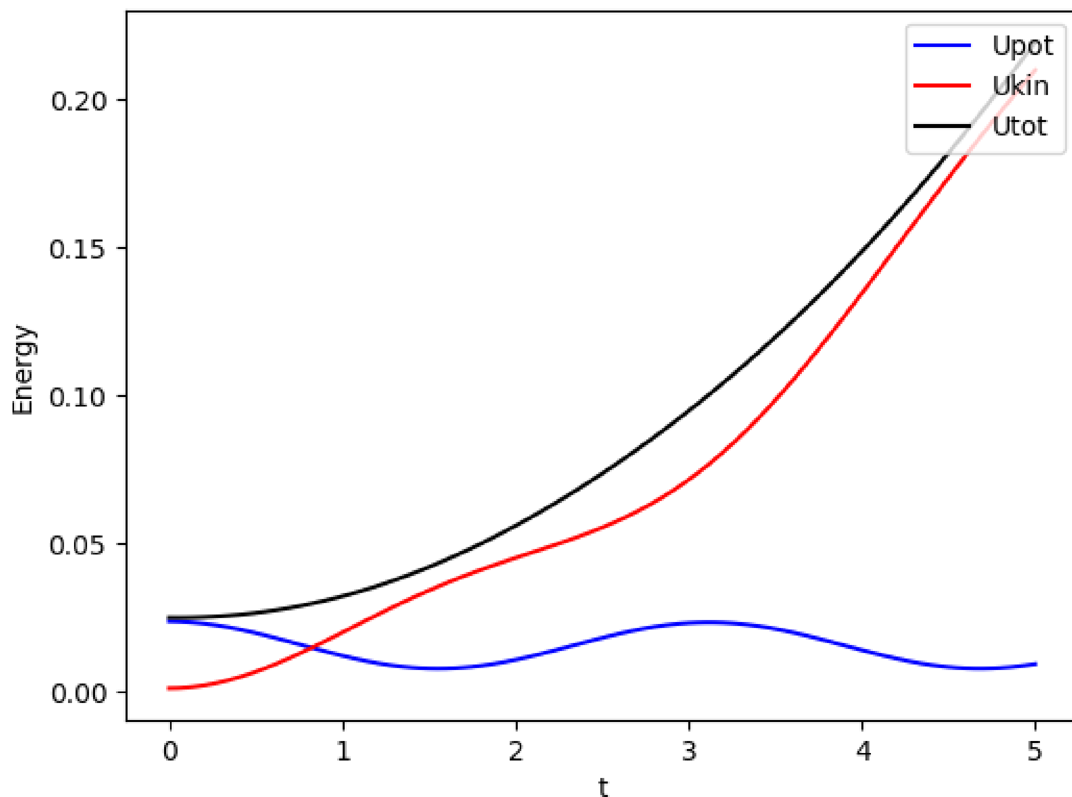


FIGURE 3.7: The evolution of potential (Upot (J)), kinetic (Ukin (J)) and total (Utot (J)) energies during the simulation t(s), after the plasma is exposed to an external magnetic field.

3.4 Conclusion

In this chapter, we present the structure of a Particle in Cell code in one dimension. We present few results obtained to illustrate the performance of this code. Also, we added a modification to the code to simulate the interaction of an external magnetic field with the plasma, the results were as we expected. Then we presented the results of this interaction and we analyzed and compared them with the results of simulating the plasma without an external magnetic field.

The advantage of this code is the possibility to simulate the behavior of a big number of particles with a very low computationally cost. And, on the other hand, in this code the magnetic field is neglected and the discretization of electric field equation is performed up to first order.

In the next chapter, we will present a more general and more realistic code that simulates a plasma in three dimensions and takes into account the magnetic field resulting from the movement of charged particles.

4 Contribution: Laser-Matter Interaction in IAP-PSC code

This chapter is a correction and development of the previous program created in AECENAR center, which is a simulation code for the interaction of plasma particles. As for correcting and improving the code, this development was based on the result of the study that we explained in the previous chapters of this thesis.

4.1 Improvement in IAP-PSC code without laser-matter interaction

This master thesis is a project development for IAP-PSC Plasma Simulation Code (Particle-in-Cell Code). The Plasma Simulation Code (PSC) is developed as an open source software. It is the intent of the project to provide a state-of-the-art simulation code for education and research in the field of intense laser-matter interaction [8].

In this project, write a code was supposed to accomplish tasks similar to that of the PSC code (simulations of the interaction of plasma's particles). However, the existing code had some errors that appeared in the results (fig 4.1). This matter is expected because the code should processes 6^{1000} equations (example used) and in addition, it contains many other errors.

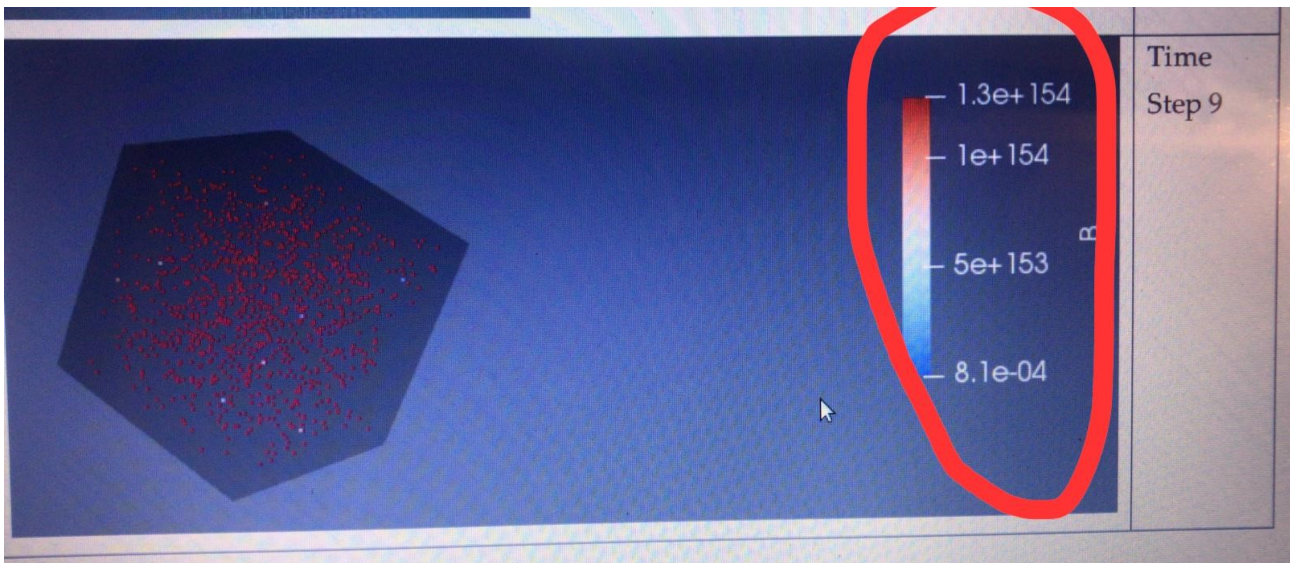


Figure 4.1: The magnetic field in order of 10^{154}! [10]

Before starting to study the interaction laser-plasma with relativistic approach, the program of IAP PSC code must be corrected, and this is what we will achieve in this chapter.

The improvement will be a series of consecutive steps.

Workshop:

1-Griding

2-write the equations (Maxwell-Vlasov) after that we will discrete the equations.

3- Solve the equations and we will develop at this point the initialization parameters and boundary conditions

4-Visualization.

Upon reaching the third stage we will have to get the correct results, then we move to the fourth stage.

4.2 Modified code

We have reformed nearly 95% of the code, and optimized it to produce acceptable results.

I will briefly mention these reforms in the form of points.

Noting that the class names remain the same, out of respect for those who started writing the code before me. However, the content of the new classes written in the code presented in this chapter is completely different, in addition, the results are different.

- New algorithm: initialization (X, Y, Z, V_X, V_Y, V_Z) of Elec and ions=>Densities ρ,
  $\rho = \rho_e + \rho_i$ => fields (E and B) => (new X, Y, Z, V_X, V_Y, V_Z).
- Corrected:
  - Initialization.
  - Grid (Yee grid).
  - Over floating.
  - Respect equation of continuity.
  - Respect boundary condition (periodic boundary).
- Results:
  - The disappearance of divergence.
  - Reasonable results (1D).

4.3 Diagrams

IAP-PSC is a C++ code of 4 classes, which solves the Maxwell equations in the presence of a gas of charged particles (plasma).

As shown in class diagram 4.2, these four classes are:

1. *parameter,*
2. *dens_currnt,*
3. *maxwell_equat,*
4. *pos_veloct.*

The class “parameter“ is used for the initialization of parameters. In the class “maxwell_equat“, we solve Maxwell's equations to get the magnetic and the electric field. We use the class “dens_currnt“ to obtain the density and the current. The class “pos_veloct“ allows to compute the position and the velocity of particles. An illustration of the code and the classes is given in the sequence diagram 4.3

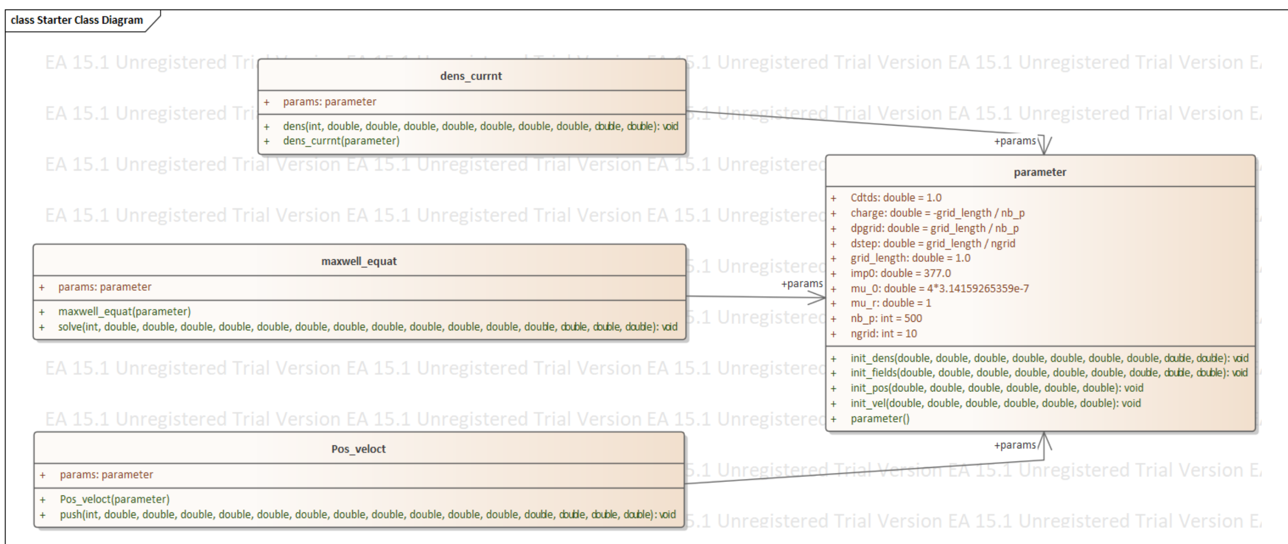


Diagram 4.2: Class diagram

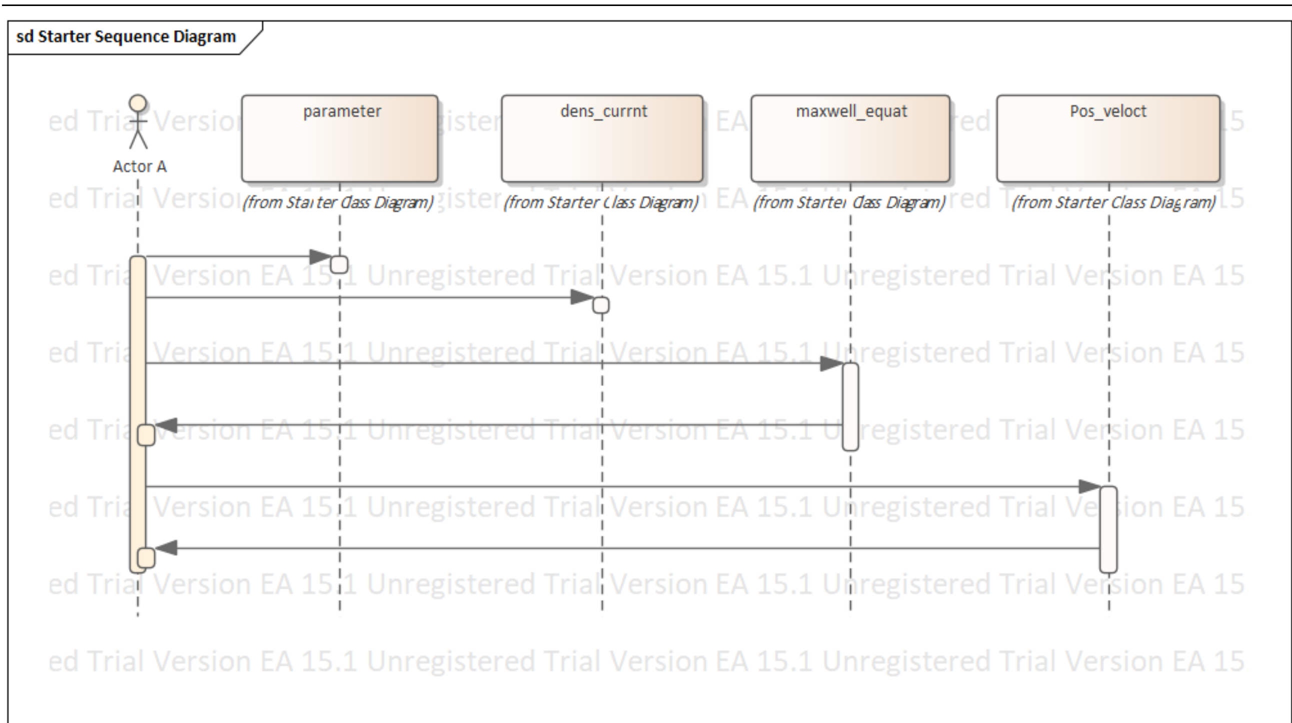


Diagram 4.3: Sequence diagram

4.4 Simulation in one dimension

We are not able to run our code in the case of 3d simulation because of the huge memory needed to save the fields E and B. However, In order to make a test for our code, we consider the 1d case and we performed a simulation similar to that discussed in the chapter 3 of Schneider [9].

The similation parameters are fixed as those chosen in the program 3.1 page 41 (Schneider...) [9] except the size of the distance box of simulation (100 spatial steps rather than 200).

We should emphasis again that the goal here is only to validate the code in the case of 1d, since we do not have the require cluster to compile the code in the 3d case.

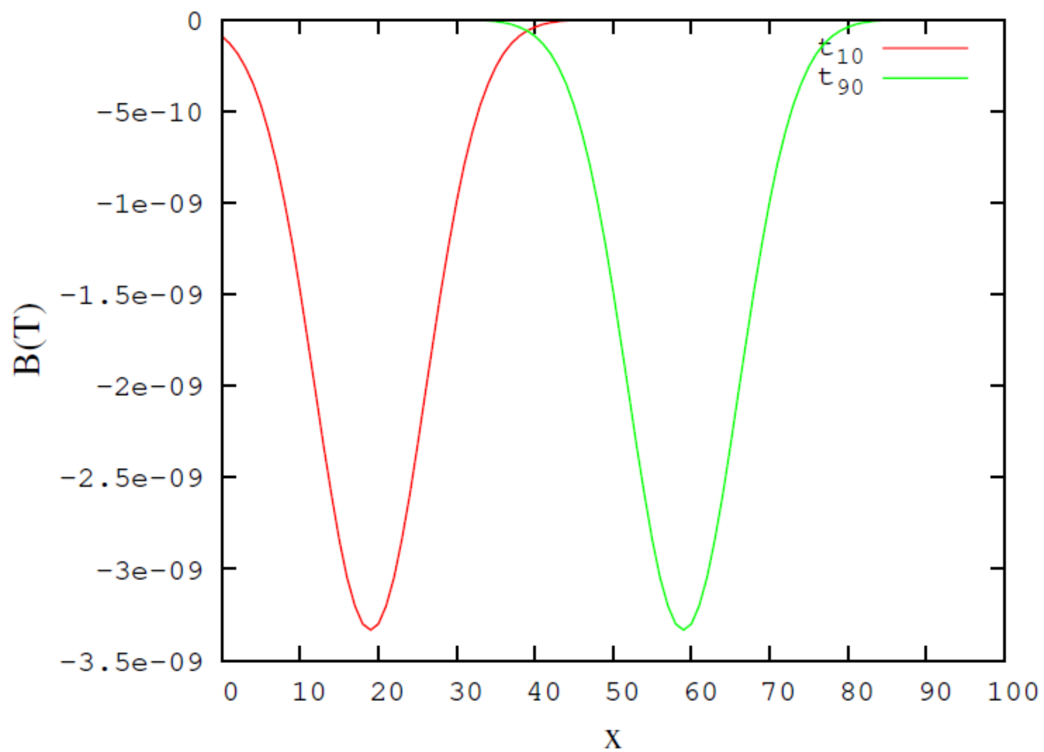


Figure 4.4: Results obtained at t10 and t90. Red line the magnetic field B(x) at t10. Green line the magnetic field B(x) at t90. It's generated by Bare-bones one-dimensional simulation with a hard source.

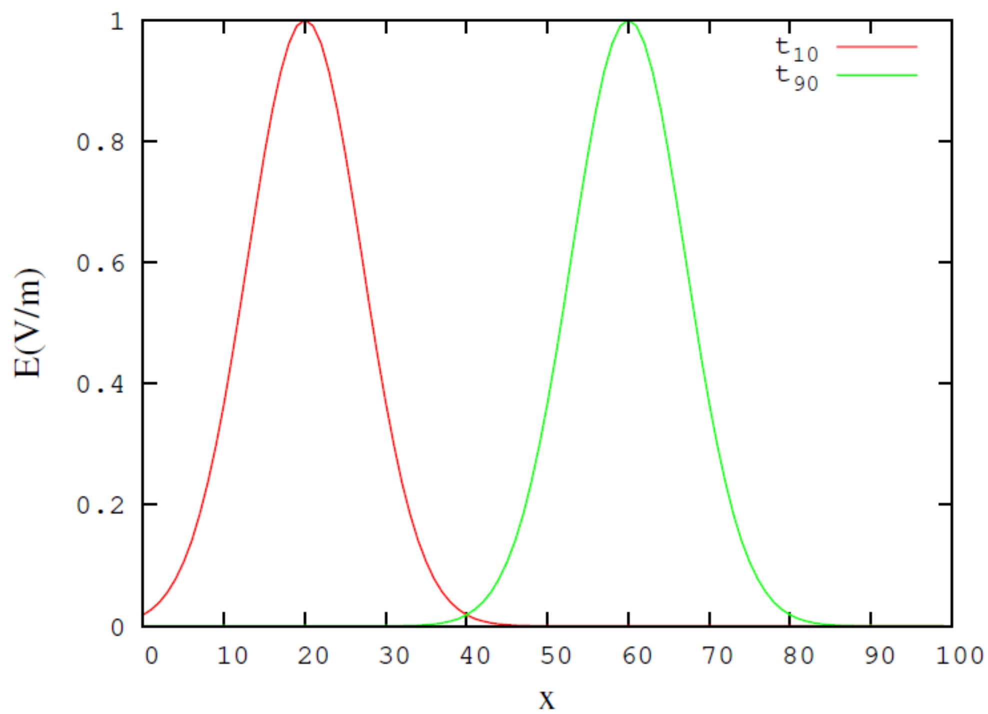


Figure 4.5: Results obtained at t10 and t90. Red line the electric field E(x) at t10. Green line the electric field E(x) at t90. It's generated by Bare-bones one-dimensional simulation with a hard source.

4.5 Simulation in three dimensions

In the case of the three dimensions, the matter was different, the memory is full due to the increased number of processed equations, which forced us to reduce the number of points in the descretisation. Hence the results were not as suspected. To solve this problem we need a supercomputer, which is not available. So we can not display the results in the three dimensions.

4.6 Conclusion

In this chapter we began by presenting the old code's problems, and its problems' results. Then we wrote a C++ code that can simulate the solution of Maxwell's equations in the material in an easy model. Then, we took into account the magnetic field inside the charged gas (plasma). And we explained in a simple way the equations used in the code. And we presented the results in the case of one dimension, then mentioned the problem in the case of the three dimensions.

5 BIBLIOGRAPHY

[1] Lee, Patrick. Modelling of a laser-plasma injector for multi-stage acceleration. Diss. 2017.

[2] Lontano, Maurizio, et al. "A kinetic model for the one-dimensional electromagnetic solitons in an isothermal plasma." Physics of Plasmas 9.6 (2002): 2562-2568.

[3] Gizzi, Leonida Antonio. "Laser-Driven Sources of High Energy Particles and Radiation." Laser-Driven Sources of High Energy Particles and Radiation. Springer, Cham, 2019. 1-24.

[4] Derouillat, Julien, et al. "Smilei: A collaborative, open-source, multi-purpose particle-in-cell code for plasma simulation." Computer Physics Communications 222 (2018): 351-373.

[5] P. Gibbon, KU-Leuven and FZ-Juelich, November 2013.

[6] McKinney, Wes. Python for data analysis: Data wrangling with Pandas, NumPy, and IPython. "O'Reilly Media, Inc.", 2012.

[7] Ranjani, J., A. Sheela, and K. Pandi Meena. "Combination of NumPy, SciPy and Matplotlib/Pylab-a good alternative methodology to MATLAB-A Comparative analysis." 2019 1st International Conference on Innovations in Information and Communication Technology (ICIICT). IEEE, 2019.

[8] Ruhl, Hartmut. "Classical Particle Simulations with the PSC code." Ruhr-Universität Bochum (2005).

[9] John B. Schneider, Understanding the Finite-Difference Time-Domain Method, sect 3, July 6, 2020.

[10] Mariam Abdel-Karim, Editor: Samir Mourad, IAP-PSC Plasma Simulation Code (Particle-in-Cell Code), 2019, http://aecenar.com/index.php/downloads/send/10-iap/496-iap-psc-plasma-simulation-code-pdf

6 IST OF SYMBOLS

Electron rest mass	m_e	9.109×10^{-31} kg
Electronic charge	e	1.6022×10^{-19} C
Speed of light in free space	c	2.9979×10^{8} m s^{-1}
Permeability of free space	μ_0	$4\pi \times 10^{-7}$ H m^{-1}
Permittivity of free space	ε_0	8.854×10^{-12} F m^{-1}
Boltzmann's constant	k_B	1.3807×10^{-23} J K^{-1}

7 Annex

7.1 IAP_PSC C++ Code

Annex A::

Main::

```
//
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
// useful libraries
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
//
#include <QCoreApplication>
#include <iostream>
#include <fstream>
#include <string>
// Headers
#include "parameter.h"
#include "pos_veloct.h"
#include "dens_currnt.h"
#include "maxwell_equat.h"
//
#include "math.h"
#include "algorithm"
#include "iostream"
#include "fstream"
#include "sstream"
#include "stdio.h"
#include <QtCore/QString>
#include <QtCore/QFile>
#include <QtCore/QDebug>
#include <QtCore/QTextStream>
#include <cstdlib>
using namespace std;
//
int main(int argc, char *argv[]){
    //
    QCoreApplication a(argc, argv);
    //
    int maxTime = 10 ;
    parameter param ;
    //
    double xe[500][10];
    double ye[500][10];
    double ze[500][10];
    double vex[500][10];
    double vey[500][10];
    double vez[500][10];
    //
    double xi[500][10];
    double yi[500][10];
    double zi[500][10];
    double vix[500][10];
    double viy[500][10];
    double viz[500][10];
    //
```

```
    double E[10][10][10];
    double B[10][10][10];
    //
    double rhox[10+1][10];
    double rhoy[10+1][10];
    double rhoz[10+1][10];
    //
    double gridx[10] ;
    double gridy[10] ;
    double gridz[10] ;
    //
    cout<<"The program has start\n";
    //
    // field's grid (uniform)
    //
        for(int mm = 0 ; mm < param.ngrid ; mm++){
        gridx[mm] = mm*param.dstep ;
        gridy[mm] = mm*param.dstep ;
        gridz[mm] = mm*param.dstep ;
    }
    //
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    // parameter initialization
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    //
    param.init_pos(xe, ye, ze, xi, yi, zi);
    param.init_vel(vex, vey, vez, vix, viy, viz);
    param.init_dens(rhox, rhoy, rhoz, xe, ye, ze, xi, yi, zi);
    double upd_Ex[10][10][10], upd_Ey[10][10][10], upd_Ez[10][10][10] ;
    double upd_Bx[10][10][10], upd_By[10][10][10], upd_Bz[10][10][10] ;
    double Ex_save[10][10][10], Ey_save[10][10][10], Ez_save[10][10][10] ;
    double Bx_save[10][10][10], By_save[10][10][10], Bz_save[10][10][10] ;
    param.init_fields(upd_Ex, upd_Ey, upd_Ez, upd_Bx, upd_By, upd_Bz, rhox,
rhoy, rhoz);
    //
    // save fields
    //
    for(int mm = 0 ; mm < param.ngrid ; mm++){
        for(int nn = 0 ; nn < param.ngrid ; nn++){
                for(int pp=0 ; pp < param.ngrid ; pp++){
                Ex_save[mm][nn][pp] = upd_Ex[mm][nn][pp] ;
                Ey_save[mm][nn][pp] = upd_Ex[mm][nn][pp] ;
                Ez_save[mm][nn][pp] = upd_Ex[mm][nn][pp] ;
                Bx_save[mm][nn][pp] = upd_Ex[mm][nn][pp] ;
                By_save[mm][nn][pp] = upd_Ex[mm][nn][pp] ;
                Bz_save[mm][nn][pp] = upd_Ex[mm][nn][pp] ;
            }
        }
    }
    //
    // total electric field
    //
    for(int mm = 0 ; mm < param.ngrid ; mm++){
        for(int nn = 0 ; nn < param.ngrid ; nn++){
                for(int pp=0 ; pp < param.ngrid ; pp++){
                E[mm][nn][pp] = upd_Ex[mm][nn][pp] + upd_Ey[mm][nn][pp] +
upd_Ez[mm][nn][pp] ;
            }
```

```
        }
    }
    //
    // total magnetix field
    //
        for(int mm = 0 ; mm < param.ngrid ; mm++){
                for(int nn = 0 ; nn < param.ngrid ; nn++){
                        for(int pp = 0 ; pp < param.ngrid ; pp++){
                B[mm][nn][pp] = upd_Bx[mm][nn][pp] + upd_By[mm][nn][pp] +
upd_Bz[mm][nn][pp] ;
            }
        }
    }
    //
    // output results
    //
    string filename = "PSC_E_0.csv" ;
    ofstream myfile ;
    // fields E and B on the grid
    myfile.open(filename.c_str());
    myfile<<"x,y,z,E,B\n";
        for(int mm = 0 ; mm < param.ngrid ; mm++){
                for(int nn = 0 ; nn < param.ngrid ; nn++){
                        for(int pp = 0 ; pp < param.ngrid ; pp++){
                myfile << gridx[mm] << "," << gridy[nn] << "," << gridz[pp] <<
","
                << E[mm][nn][pp] << "," << B[mm][nn][pp] << "\n" ;
            }
        }
    }
    myfile.close() ;
    // distributions
    double vv ;
    filename= "PSC_dist_0.csv" ;
    myfile.open(filename.c_str());
    myfile<<"x,y,z,v\n";
    for(int i = 0 ; i < param.nb_p ; i++){
        // electrons
        vv = sqrt( vex[i][0]*vex[i][0] + vey[i][0]*vey[i][0] +
vez[i][0]*vez[i][0] ) ;
        myfile << xe[i][0] << "," << ye[i][0] << "," << ze[i][0] << "," << vv <<
"\n";
        // ions
        vv = sqrt( vix[i][0]*vix[i][0] + viy[i][0]*viy[i][0] +
viz[i][0]*viz[i][0] ) ;
        myfile << xi[i][0] << "," << yi[i][0] << "," << zi[i][0] << "," << vv <<
"\n";
    }
    myfile.close() ;
    //
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    // Loop over time
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    //
    for(int t = 1 ; t < maxTime ; t++){
        //
        // Push position and velocity
        //
```

```
        Pos_veloct vel = Pos_veloct(param) ;
        vel.push(t, xe, ye, ze, vex, vey, vez, xi, yi, zi, vix, viy, viz,
upd_Ex, upd_Ey, upd_Ez) ;
        //
        // update density
        //
        dens_currnt densities = dens_currnt(param);
        densities.dens(t, xe, ye, ze, xi, yi, zi, rhox, rhoy, rhoz) ;
        //
        // solve maxwell equations
        //
        for(int mm = 0 ; mm < param.ngrid ; mm++){
                  for(int nn = 0 ; nn < param.ngrid ; nn++){
                      for(int pp=0 ; pp < param.ngrid ; pp++){
                           Ex_save[mm][nn][pp] = Ex_save[mm][nn][pp] + 1000;
                      }
                  }
              }
        maxwell_equat max = maxwell_equat(param);
        max.solve(t, upd_Ex, upd_Ey, upd_Ez, upd_Bx, upd_By, upd_Bz, rhox, rhoy,
rhoz, Ex_save,Ey_save,Ez_save,Bx_save,By_save,Bz_save) ;
        //
        // save fields
        //
                for(int mm = 0 ; mm < param.ngrid ; mm++){
                        for(int nn = 0 ; nn < param.ngrid ; nn++){
                                for(int pp=0 ; pp < param.ngrid ; pp++){
                            Ex_save[mm][nn][pp] = upd_Ex[mm][nn][pp] ;
                            Ey_save[mm][nn][pp] = upd_Ex[mm][nn][pp] ;
                            Ez_save[mm][nn][pp] = upd_Ex[mm][nn][pp] ;
                            Bx_save[mm][nn][pp] = upd_Ex[mm][nn][pp] ;
                            By_save[mm][nn][pp] = upd_Ex[mm][nn][pp] ;
                            Bz_save[mm][nn][pp] = upd_Ex[mm][nn][pp] ;
                        }
                    }
                }
        //
        // total electric field
        //
        for(int mm=0;mm<param.ngrid;mm++){
                for(int nn=0;nn<param.ngrid;nn++){
                        for(int pp=0;pp<param.ngrid;pp++){
                            E[mm][nn][pp] = upd_Ex[mm][nn][pp] + upd_Ey[mm][nn][pp] +
upd_Ez[mm][nn][pp] ;
                        }
                    }
                }
        //
        // total magnetix field
        //
        for(int mm=0;mm<param.ngrid;mm++){
                for(int nn=0;nn<param.ngrid;nn++){
                        for(int pp=0;pp<param.ngrid;pp++){
                            B[mm][nn][pp] = upd_Bx[mm][nn][pp] + upd_By[mm][nn][pp] +
upd_Bz[mm][nn][pp] ;
                        }
                    }
                }
        //
        // output results
        //
        string filename ;
```

```
        ofstream myfile ;
        stringstream d ;
        double vv ;
        d<<t;
        // fields E and B on the grid
        filename= "PSC_E_"+ d.str();
        filename+= ".csv";
        myfile.open(filename.c_str());
        myfile<<"x,y,z,E,B\n";
                for(int mm=0;mm<param.ngrid;mm++){
                        for(int nn=0;nn<param.ngrid;nn++){
                                for(int pp=0;pp<param.ngrid;pp++){
                    myfile << gridx[mm] << "," << gridy[nn] << "," << gridz[pp]
<< ","
                    << E[mm][nn][pp] << "," << B[mm][nn][pp] << "\n" ;
                }
            }
        }
        myfile.close();
        // distributions
        filename= "PSC_dist_"+ d.str();
        filename+= ".csv";
        myfile.open(filename.c_str());
        myfile<<"x,y,z,v\n";
        for(int i=0;i<param.nb_p;i++){
            // electrons
            vv = sqrt( vex[i][t]*vex[i][t] + vey[i][t]*vey[i][t] +
vez[i][t]*vez[i][t] ) ;
            myfile << xe[i][t] << "," << ye[i][t] << "," << ze[i][t] << "," <<
vv << "\n";
            // ions
            vv = sqrt( vix[i][t]*vix[i][t] + viy[i][t]*viy[i][t] +
viz[i][t]*viz[i][t] ) ;
            myfile << xi[i][t] << "," << yi[i][t] << "," << zi[i][t] << "," <<
vv << "\n";
        }
        myfile.close();
        //
        //system("pause");
    }
    //return a.exec();
    cout << "END \n" ;
    return 0 ;
// End
}
```

Parameter

Parameter.h::

```
#ifndef PARAMETER_H
#define PARAMETER_H

//int maxTime = 50 ;
//int nb_p = 500 ;
//int ngrid = 50 ;

class parameter{
  public:
  parameter(){}
  //
  void init_pos(double xe[500][10], double ye[500][10], double ze[500][10],
```

```
    double xi[500][10], double yi[500][10], double zi[500][10]) ;
  //
  void init_vel(double vex[500][10], double vey[500][10], double vez[500][10],
    double vix[500][10], double viy[500][10], double viz[500][10]) ;
  //
  void init_dens(double rhox[10+1][10], double rhoy[10+1][10],double
rhoz[10+1][10],
    double xe[500][10], double ye[500][10], double ze[500][10],
    double xi[500][10], double yi[500][10], double zi[500][10]) ;
  //
  void init_fields(double upd_Ex[10][10][10], double upd_Ey[10][10][10], double
upd_Ez[10][10][10],
    double upd_Bx[10][10][10], double upd_By[10][10][10], double
upd_Bz[10][10][10],
    double rhox[10+1][10], double rhoy[10+1][10],double rhoz[10+1][10]) ;

  int nb_p = 500 ;
  int ngrid = 10 ;
  double grid_length = 1.0 ;
  double dstep = grid_length / ngrid ;
  double dpgrid = grid_length / nb_p ; //particle spacing
  double charge = -grid_length / nb_p ; //pseudo-particle charge normalised to
give ncrit=1
  double imp0 = 377.0 ;
  double Cdtds = 1.0 ; // Courant number
  double mu_r=1 ;
  double mu_0 = 4*3.14159265359e-7 ;


};

#endif // PARAMETER_H
```

Parameter.cpp::

```
//
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
// useful libraries
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
//
#include "parameter.h"
#include <cstdlib>
#include <math.h>
//
#include <iostream>
using namespace std;
  //
  //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
  // load positions
  //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
  //
  void parameter::init_pos(double xe[500][10], double ye[500][10], double
ze[500][10],
```

```
    double xi[500][10] , double yi[500][10] , double zi[500][10]){
    //
//    int nb_p = 500 ;
//    int ngrid = 10 ;
//    double grid_length = 1.0 ;
//    double dstep = grid_length / ngrid ;
//    double dpgrid = grid_length / nb_p ; //particle spacing
//    double charge = -grid_length / nb_p ; //pseudo-particle charge normalised
to give ncrit=1
//    double imp0 = 377.0 ;
//    double Cdtds = 1.0 ; // Courant number
//    double mu_r=1 ;
//    double mu_0 = 4*3.14159265359e-7 ;
    //
    int b=0 ;
    //
    for(int l=0; l < nb_p; l++){
      //
      xe[l][0] = 0 + dpgrid * (l+0.5) ;
      xe[l][0] = xe[l][0] + 0.1 * cos(xe[l][0]) ;
      // periodic boundaries
      if( xe[l][0] < 0 ){
        xe[l][0] = xe[l][0] + grid_length ;
      } else if( xe[l][0] >= grid_length ){
        xe[l][0] = xe[l][0] - grid_length ;
      }
      //
      b=rand()%nb_p ;
      ye[l][0] = b * grid_length / nb_p ;
      // periodic boundaries
      if( ye[l][0] < 0 ){
        ye[l][0] = ye[l][0] + grid_length ;
      } else if( ye[l][0] >= grid_length ){
        ye[l][0] = ye[l][0] - grid_length ;
      }
      //
      b=rand()%nb_p ;
      ze[l][0] = b * grid_length / nb_p ;
      // periodic boundaries
      if( ze[l][0] < 0 ){
        ze[l][0] = ze[l][0] + grid_length ;
      } else if( ze[l][0] >= grid_length ){
        ze[l][0] = ze[l][0] - grid_length ;
      }
      //
      b=rand()%nb_p ;
      xi[l][0] = b * grid_length / nb_p ;
      // periodic boundaries
      if( xi[l][0] < 0 ){
        xi[l][0] = xi[l][0] + grid_length ;
      } else if( xi[l][0] >= grid_length ){
        xi[l][0] = xi[l][0] - grid_length ;
      }
      //
      b=rand()%nb_p ;
      yi[l][0] = b * grid_length / nb_p ;
      // periodic boundaries
      if( ye[l][0] < 0 ){
        yi[l][0] = yi[l][0] + grid_length ;
      } else if( yi[l][0] >= grid_length ){
        yi[l][0] = yi[l][0] - grid_length ;
      }
```

```
      //
      b=rand()%nb_p ;
      zi[l][0] = b * grid_length / nb_p ;
      // periodic boundaries
      if( ze[l][0] < 0 ){
        zi[l][0] = zi[l][0] + grid_length ;
      } else if( zi[l][0] >= grid_length ){
        zi[l][0] = zi[l][0] - grid_length ;
      }
    }
  // end void
  }
  //
  //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
  // load velocity (cold plasma as example but it can be modified)
  //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
  //
  void parameter::init_vel(double vex[500][10], double vey[500][10], double
vez[500][10],
    double vix[500][10], double viy[500][10], double viz[500][10]){
    //
//    int nb_p = 500 ;
//    int ngrid = 10 ;
//    double grid_length = 1.0 ;
//    double dstep = grid_length / ngrid ;
//    double dpgrid = grid_length / nb_p ; //particle spacing
//    double charge = -grid_length / nb_p ; //pseudo-particle charge normalised
to give ncrit=1
//    double imp0 = 377.0 ;
//    double Cdtds = 1.0 ; // Courant number
//    double mu_r=1 ;
//    double mu_0 = 4*3.14159265359e-7 ;
    //
    for(int l=0; l < nb_p; l++) {
      //
      vex[l][0] = 1e-5 ;
      vey[l][0] = 1e-5 ;
      vez[l][0] = 1e-5 ;
      //
      // ion is heavy
      vix[l][0] = 1e-10 ;
      viy[l][0] = 1e-10 ;
      viz[l][0] = 1e-10 ;
    }
  // end void
  }
  //
  //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
  // Compute densities
  //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
  //
  void parameter::init_dens(double rhox[10+1][10], double rhoy[10+1][10],double
rhoz[10+1][10],
    double xe[500][10], double ye[500][10], double ze[500][10],
```

```
    double xi[500][10], double yi[500][10], double zi[500][10]){
    //
//    int nb_p = 500 ;
//    int ngrid = 10 ;
//    double grid_length = 1.0 ;
//    double dstep = grid_length / ngrid ;
//    double dpgrid = grid_length / nb_p ; //particle spacing
//    double charge = -grid_length / nb_p ; //pseudo-particle charge normalised
to give ncrit=1
//    double imp0 = 377.0 ;
//    double Cdtds = 1.0 ; // Courant number
//    double mu_r=1 ;
//    double mu_0 = 4*3.14159265359e-7 ;
    //
    double re = charge / dstep ; //charge weighting factor
    double rhoe[10+1], rhoi[10+1] ;
    for(int l=0;l<ngrid+1;l++){
      rhoe[l] = 0.0 ; //background electron density
      rhoi[l] = 1.0 ; //background ion density
    }
    double xa, f1, f2 ;
    int j1, j2 ;
    // map charges onto grid
    for(int l=0; l < ngrid; l++){
      xa = xe[l][0] / dstep ;
      j1 = int(xa) ;
      j2 = j1 + 1 ;
      f2 = xa - j1 ;
      f1 = 1.0 - f2 ;
      rhoe[j1] = rhoe[j1] + re*f1 ;
      rhoe[j2] = rhoe[j2] + re*f2 ;
    }
    // periodic boundaries
    rhoe[0] = rhoe[0] + rhoe[ngrid] ;
    rhoe[ngrid] = rhoe[0] ;
    //
    for(int l=0; l < ngrid+1; l++){
      rhox[l][0] = rhoe[l] + rhoi[l] ;
      rhoy[l][0] = 0.0 ;
      rhoz[l][0] = 0.0 ;
    }
  // end void
  }
  //
  //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
  // solve maxwell equations (Ampere's and Faraday's laws)
  //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
  //
  void parameter::init_fields(double upd_Ex[10][10][10], double
upd_Ey[10][10][10], double upd_Ez[10][10][10],
    double upd_Bx[10][10][10], double upd_By[10][10][10], double
upd_Bz[10][10][10],
    double rhox[10+1][10], double rhoy[10+1][10],double rhoz[10+1][10]){
    //
//    int ngrid = 10 ;
//    double grid_length = 1.0 ;
//    double dstep = grid_length / ngrid ;
//    double imp0 = 377.0 ;
```

```
//    double Cdtds = 1.0 ; // Courant number
//    double mu_r=1 , mu_0 = 4*3.14159265359e-7 ;
    //
    double Ex_0[10][10][10], Ey_0[10][10][10], Ez_0[10][10][10] ;
    double Hx_0[10][10][10], Hy_0[10][10][10], Hz_0[10][10][10] ;
    //
    double s_ex=0, s_ey=0, s_ez=0 ; //need this for consistency with charge
conservation
    //
    for (int mm = 0; mm < ngrid ; mm++){
      for (int nn = 0; nn < ngrid; nn++){
        for (int pp = 0; pp < ngrid; pp++){
          //
          Ex_0[mm][nn][pp] = 0.0 ;
          Ey_0[mm][nn][pp] = 0.0 ;
          Ez_0[mm][nn][pp] = 0.0 ;
          //=
          Hx_0[mm][nn][pp] = 0.0 ;
          Hy_0[mm][nn][pp] = 0.0 ;
          Hz_0[mm][nn][pp] = 0.0 ;
          //
        }
      }
    }
    //
    // Electric field
    //
    for (int mm = 0; mm < ngrid - 1; mm++){
      for (int nn = 1; nn < ngrid - 1; nn++){
        for (int pp = 1; pp < ngrid - 1; pp++){
          Ex_0[mm][nn][pp] = 1.0 * Ex_0[mm][nn][pp] + (Cdtds / imp0) *
((Hz_0[mm][nn][pp] -
            Hz_0[mm][nn-1][pp]) - (Hy_0[mm][nn][pp] - Hy_0[mm][nn][pp-1])) ;
          Ex_0[mm][nn][pp] = Ex_0[mm][nn][pp] - 0.5*( rhox[mm][0] +
rhox[mm+1][0] )*dstep  ; //Additive Source
          s_ex = s_ex + Ex_0[mm][nn][pp] ; //need this for consistency with
charge conservation
        }
      }
    }
    //
    for (int mm = 1; mm < ngrid - 1; mm++){
      for (int nn = 0; nn < ngrid - 1; nn++){
        for (int pp = 1; pp < ngrid - 1; pp++){
          Ey_0[mm][nn][pp] = 1.0 * Ey_0[mm][nn][pp] + (Cdtds / imp0) *
((Hx_0[mm][nn][pp] -
            Hx_0[mm][nn][pp-1]) - (Hz_0[mm][nn][pp] - Hz_0[mm-1][nn][pp])) ;
          Ey_0[mm][nn][pp] = Ey_0[mm][nn][pp] - 0.5*( rhoy[mm][0] +
rhoy[mm+1][0] )*dstep  ; //Additive Source
          s_ey = s_ey + Ey_0[mm][nn][pp] ; //need this for consistency with
charge conservation
        }
      }
    }
    //
    for (int mm = 1; mm < ngrid - 1; mm++){
      for (int nn = 1; nn < ngrid - 1; nn++){
        for (int pp = 0; pp < ngrid - 1; pp++){
          Ez_0[mm][nn][pp] = 1.0 * Ez_0[mm][nn][pp] + (Cdtds / imp0) *
((Hy_0[mm][nn][pp] -
            Hy_0[mm-1][nn][pp]) - (Hx_0[mm][nn][pp] - Hx_0[mm][nn-1][pp])) ;
```

```
          Ez_0[mm][nn][pp] = Ez_0[mm][nn][pp] - 0.5*( rhoz[mm][0] +
rhoz[mm+1][0] )*dstep  ; //Additive Source
          s_ez = s_ez + Ez_0[mm][nn][pp] ; //need this for consistency with
charge conservation
        }
      }
    }
    //
    // Magnetic field
    //
    for (int mm = 0; mm < ngrid; mm++){
      for (int nn = 0; nn < ngrid - 1; nn++){
        for (int pp = 0; pp < ngrid - 1; pp++){
          Hx_0[mm][nn][pp] = (Cdtds / imp0) * Hx_0[mm][nn][pp] + 1.0 *
((Ey_0[mm][nn][pp+1] -
            Ey_0[mm][nn][pp]) - (Ez_0[mm][nn+1][pp] - Ez_0[mm][nn][pp])) ;
        }
      }
    }
    //
    for (int mm = 0; mm < ngrid - 1; mm++){
      for (int nn = 0; nn < ngrid; nn++){
        for (int pp = 0; pp < ngrid - 1; pp++){
          Hy_0[mm][nn][pp] = (Cdtds / imp0) * Hy_0[mm][nn][pp] + 1.0 *
((Ez_0[mm+1][nn][pp] -
            Ez_0[mm][nn][pp]) - (Ex_0[mm][nn][pp+1] - Ex_0[mm][nn][pp])) ;
        }
      }
    }
    //
    for (int mm = 0; mm < ngrid - 1; mm++){
      for (int nn = 0; nn < ngrid - 1; nn++){
        for (int pp = 0; pp < ngrid; pp++){
          Hz_0[mm][nn][pp] = (Cdtds / imp0) * Hz_0[mm][nn][pp] + 1.0 *
((Ex_0[mm][nn+1][pp] -
            Ex_0[mm][nn][pp]) - (Ey_0[mm+1][nn][pp] - Ey_0[mm][nn][pp])) ;
        }
      }
    }
    //
    // return Electric fields
    //
    for (int mm = 0; mm < ngrid ; mm++){
      for (int nn = 0; nn < ngrid ; nn++){
        for (int pp = 0; pp < ngrid ; pp++){
          upd_Ex[mm][nn][pp] = Ex_0[mm][nn][pp] - s_ex / ngrid  ;
        }
      }
    }
    upd_Ex[ngrid-1][ngrid-1][ngrid-1] = upd_Ex[0][ngrid-1][ngrid-1] ; //
periodic boundaries
    //
    for (int mm = 0; mm < ngrid ; mm++){
      for (int nn = 0; nn < ngrid ; nn++){
        for (int pp = 0; pp < ngrid ; pp++){
          upd_Ey[mm][nn][pp] = Ey_0[mm][nn][pp] - s_ey / ngrid  ;
        }
      }
    }
    upd_Ey[ngrid-1][ngrid-1][ngrid-1] = upd_Ey[ngrid-1][0][ngrid-1] ; //
periodic boundaries
    //
```

```
    for (int mm = 0; mm < ngrid ; mm++){
      for (int nn = 0; nn < ngrid ; nn++){
        for (int pp = 0; pp < ngrid ; pp++){
          upd_Ez[mm][nn][pp] = Ez_0[mm][nn][pp] - s_ez / ngrid ;
        }
      }
    }
    upd_Ez[ngrid-1][ngrid-1][ngrid-1] = upd_Ez[ngrid-1][ngrid-1][0] ; //
periodic boundaries
    //
    // return Magnetic field : B = mu_r * mu_0 * H
    //
    for (int mm = 0; mm < ngrid ; mm++){
      for (int nn = 0; nn < ngrid ; nn++){
        for (int pp = 0; pp < ngrid ; pp++){
          upd_Bx[mm][nn][pp] = mu_r * mu_0 * Hx_0[mm][nn][pp] ;
        }
      }
    }
    //
    for (int mm = 0; mm < ngrid ; mm++){
      for (int nn = 0; nn < ngrid ; nn++){
        for (int pp = 0; pp < ngrid ; pp++){
          upd_By[mm][nn][pp] = mu_r * mu_0 * Hy_0[mm][nn][pp] ;
        }
      }
    }
    //
    for (int mm = 0; mm < ngrid ; mm++){
      for (int nn = 0; nn < ngrid ; nn++){
        for (int pp = 0; pp < ngrid ; pp++){
          upd_Bz[mm][nn][pp] = mu_r * mu_0 * Hz_0[mm][nn][pp] ;
        }
      }
    }
  //end void
  }
```

Density

Dens_currnt.h::

```
#ifndef DENS_CURRNT_H
#define DENS_CURRNT_H
#include "parameter.h"
class dens_currnt
{
public :
    parameter params;
    dens_currnt(parameter p){
        this->params = p;
    }
    void dens(int t, double xe[500][10], double ye[500][10], double ze[500][10],
              double xi[500][10] , double yi[500][10] , double zi[500][10],
              double rhox[10+1][10], double rhoy[10+1][10],double
rhoz[10+1][10]) ;
};

#endif // DENS_CURRNT_H
```

Dens_currnt.cpp::

```
//
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
// useful libraries
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
#include "dens_currnt.h"
#include "parameter.h"
//
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
// Global parameters
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
//
//int maxTime = 50 ;
//int nb_p = 500 ;
//int ngrid = 50 ;
//
//double pi = 3.14159265359 ;
//
//double grid_length = 2*pi ;
//double dstep = grid_length / ngrid ;
//double dpgrid = grid_length / nb_p ; //particle spacing
//double charge = -grid_length / nb_p ; //pseudo-particle charge normalised to
give ncrit=1
//double mass = grid_length / nb_p ; //pseudo-particle mass
//double imp0 = 377.0 ;
//double mu_r=1, mu_0 = 4*3.14159265359e-7 ;
//double q_over_me=-1.0 ;
//double dt = 0.05 ; //normalised timestep
//
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
//
    parameter param1;
    void dens_currnt::dens(int t, double xe[500][10], double ye[500][10], double
ze[500][10],
                        double xi[500][10], double yi[500][10], double zi[500][10],
                        double rhox[10+1][10], double rhoy[10+1][10], double
rhoz[10+1][10]){
    //
//     int nb_p = 500 ;
//     int ngrid = 10 ;
//     //
//     double grid_length = 1.0 ;
//     double dstep = grid_length / ngrid ;
//     double charge = -grid_length / nb_p ;
    //
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    // Compute densities
```

```
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    //
    double re = param1.charge / param1.dstep ; //charge weighting factor
    double rhoe[10+1], rhoi[10+1] ;
    for(int l = 0 ; l < param1.ngrid + 1 ; l++){
        rhoe[l] = 0.0 ;
        rhoi[l] = 1.0 ; //background ion density
    }
    double xa, f1, f2 ;
    int j1, j2 ;
    //
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    // map charges onto grid
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    //
    for(int l=0;l<param1.ngrid;l++){
        xa = xe[l][t] / param1.dstep ;
        j1 = int(xa) ;
        j2 = j1 + 1 ;
        f2 = xa - j1 ;
        f1 = 1.0 - f2 ;
        rhoe[j1] = rhoe[j1] + re*f1 ;
        rhoe[j2] = rhoe[j2] + re*f2 ;
    }
    // periodic boundaries
    rhoe[0] = rhoe[0] + rhoe[param1.ngrid] ;
    rhoe[param1.ngrid] = rhoe[0] ;
    //
    for(int l=0;l<param1.ngrid+1;l++){
        rhox[l][t] = rhoe[l] + rhoi[l] ;
        rhoy[l][t] = 0.0 ;
        rhoz[l][t] = 0.0 ;
    }
// end void
}
```

Maxwell equation

Maxwell_equat.h::

```
#ifndef MAXWELL_EQUAT_H
#define MAXWELL_EQUAT_H

//int maxTime = 50 ;
//int nb_p = 500 ;
//int ngrid = 50 ;
#include "parameter.h"
class maxwell_equat
{
public:
    parameter params;
    maxwell_equat(parameter p){
        this->params = p;
    }
    //
```

```
    void solve(int t, double Ex[10][10][10], double Ey[10][10][10], double
Ez[10][10][10],
        double Bx[10][10][10], double By[10][10][10], double Bz[10][10][10],
        double rhox[10+1][10], double rhoy[10+1][10],double rhoz[10+1][10],
        double Ex_save[10][10][10], double Ey_save[10][10][10], double
Ez_save[10][10][10],
        double Bx_save[10][10][10], double By_save[10][10][10], double
Bz_save[10][10][10]);
};

#endif // MAXWELL_EQUAT_H
```

Maxwell_equat.cpp::

```
//
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
// useful libraries
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
//
#include "maxwell_equat.h"
#include "parameter.h"
#include "math.h"
//
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
// Global parameters
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
//
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
//
    parameter param;
    void maxwell_equat::solve(int t, double Ex[10][10][10], double
Ey[10][10][10], double Ez[10][10][10],
        double Bx[10][10][10], double By[10][10][10], double Bz[10][10][10],
        double rhox[10+1][10], double rhoy[10+1][10],double rhoz[10+1][10],
        double Ex_save[10][10][10], double Ey_save[10][10][10], double
Ez_save[10][10][10],
        double Bx_save[10][10][10], double By_save[10][10][10], double
Bz_save[10][10][10]){
    //
//  int ngrid = 10 ;
//  double grid_length = 1.0 ;
//  double dstep = grid_length / ngrid ;
//  double mu_0 = 4*3.14159265359e-7 ;
    //
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    // solve maxwell equations (Ampere's and Faraday's laws)
```

```
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    //
    double s_ex=0, s_ey=0, s_ez=0 ; //need this for consistency with charge
conservation
    //
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    // Magnetic field
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    //
        for (int mm = 0; mm < param.ngrid ; mm++){
                for (int nn = 0; nn < param.ngrid ; nn++){
            for (int pp = 0; pp < param.ngrid ; pp++){
                Bx[mm][nn][pp] = Bx_save[mm][nn][pp] / param.mu_0 ;
                By[mm][nn][pp] = By_save[mm][nn][pp] / param.mu_0 ;
                Bz[mm][nn][pp] = Bz_save[mm][nn][pp] / param.mu_0 ;
            }
        }
    }
    //
    for (int mm = 0; mm < param.ngrid; mm++){
        for (int nn = 0; nn < param.ngrid - 1; nn++){
            for (int pp = 0; pp < param.ngrid - 1; pp++){
                Bx[mm][nn][pp] = (1.0 / 377.0) * Bx_save[mm][nn][pp] + 1.0 *
(Ey_save[mm][nn][pp+1] -
                    Ey_save[mm][nn][pp]) - (Ez_save[mm][nn+1][pp] -
Ez_save[mm][nn][pp]) ;
            }
        }
    }
    //
    for (int mm = 0; mm < param.ngrid - 1; mm++){
        for (int nn = 0; nn < param.ngrid; nn++){
            for (int pp = 0; pp < param.ngrid - 1; pp++){
                By[mm][nn][pp] = (1.0 / 377.0) * By_save[mm][nn][pp] + 1.0 *
((Ez_save[mm+1][nn][pp] -
                    Ez_save[mm][nn][pp]) - (Ex_save[mm][nn][pp+1] -
Ex_save[mm][nn][pp])) ;
            }
        }
    }
    //
    for (int mm = 0; mm < param.ngrid - 1; mm++){
        for (int nn = 0; nn < param.ngrid - 1; nn++){
             for (int pp = 0; pp < param.ngrid; pp++){
                Bz[mm][nn][pp] = (1.0 / 377.0) * Bz_save[mm][nn][pp] + 1.0 *
((Ex_save[mm][nn+1][pp]
                    - Ex_save[mm][nn][pp]) - (Ey_save[mm+1][nn][pp] -
Ey_save[mm][nn][pp])) ;
            }
        }
    }
    //
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    // Electric field
```

```
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    //
    for (int mm = 0; mm < param.ngrid - 1; mm++){
        for (int nn = 1; nn < param.ngrid - 1; nn++){
            for (int pp = 1; pp < param.ngrid - 1; pp++){
                Ex[mm][nn][pp] = 1.0 * Ex_save[mm][nn][pp] + (1.0 / 377.0) *
((Bz_save[mm][nn][pp] -
                    Bz_save[mm][nn-1][pp]) - (By_save[mm][nn][pp] -
By_save[mm][nn][pp-1])) ;
                Ex[mm][nn][pp] = Ex[mm][nn][pp] - 0.5*( rhox[mm][t] +
rhox[mm+1][t] )*param.dstep  ; //Additive Source
                s_ex = s_ex + Ex[mm][nn][pp] ; //need this for consistency with
charge conservation
            }
        }
    }
    //
    for (int mm = 1; mm < param.ngrid - 1; mm++){
        for (int nn = 0; nn < param.ngrid - 1; nn++){
            for (int pp = 1; pp < param.ngrid - 1; pp++){
                Ey[mm][nn][pp] = 1.0 * Ey_save[mm][nn][pp] + (1.0 / 377.0) *
((Bx_save[mm][nn][pp] -
                    Bx_save[mm][nn][pp-1]) - (Bz_save[mm][nn][pp] - Bz_save[mm-
1][nn][pp])) ;
                Ey[mm][nn][pp] = Ey[mm][nn][pp] - 0.5*( rhoy[mm][t] +
rhoy[mm+1][t] )*param.dstep  ; //Additive Source
                s_ey = s_ey + Ey[mm][nn][pp] ; //need this for consistency with
charge conservation
            }
        }
    }
    //
    for (int mm = 1; mm < param.ngrid - 1; mm++){
        for (int nn = 1; nn < param.ngrid - 1; nn++){
            for (int pp = 0; pp < param.ngrid - 1; pp++){
                Ez[mm][nn][pp] = 1.0 * Ez_save[mm][nn][pp] + (1.0 / 377.0) *
((By_save[mm][nn][pp] -
                    By_save[mm-1][nn][pp]) - (Bx_save[mm][nn][pp] -
Bx_save[mm][nn-1][pp])) ;
                Ez[mm][nn][pp] = Ez[mm][nn][pp] - 0.5*( rhoz[mm][t] +
rhoz[mm+1][t] )*param.dstep  ; //Additive Source
                s_ez = s_ez + Ez[mm][nn][pp] ; //need this for consistency with
charge conservation
            }
        }
    }
    //
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    // return Electric fields
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    //
    for (int mm = 0; mm < param.ngrid ; mm++){
        for (int nn = 0; nn < param.ngrid ; nn++){
            for (int pp = 0; pp < param.ngrid ; pp++){
                Ex[mm][nn][pp] = Ex[mm][nn][pp] - s_ex / param.ngrid  ;
                Ey[mm][nn][pp] = Ey[mm][nn][pp] - s_ey / param.ngrid  ;
```

```
                Ez[mm][nn][pp] = Ez[mm][nn][pp] - s_ez / param.ngrid ;
            }
        }
    }
    Ex[param.ngrid-1][param.ngrid-1][param.ngrid-1] = Ex[0][param.ngrid-
1][param.ngrid-1] ; // periodic boundaries
    Ey[param.ngrid-1][param.ngrid-1][param.ngrid-1] = Ey[param.ngrid-
1][0][param.ngrid-1] ; // periodic boundaries
    Ez[param.ngrid-1][param.ngrid-1][param.ngrid-1] = Ez[param.ngrid-
1][param.ngrid-1][0] ; // periodic boundaries
    //
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    // return Magnetic field : B = mu_0 * H
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    //
    for (int mm = 0; mm < param.ngrid ; mm++){
        for (int nn = 0; nn < param.ngrid ; nn++){
            for (int pp = 0; pp < param.ngrid ; pp++){
                Bx[mm][nn][pp] = param.mu_0 * Bx[mm][nn][pp] ;
                By[mm][nn][pp] = param.mu_0 * By[mm][nn][pp] ;
                Bz[mm][nn][pp] = param.mu_0 * Bz[mm][nn][pp] ;
            }
        }
    }
//end void
}
```

Position and velocity

Pos_veloct.h

```
#ifndef POS_VELOCT_H
#define POS_VELOCT_H
#include "parameter.h"
class Pos_veloct
{
public:
    parameter params;
    Pos_veloct(parameter p){
        this->params = p;
    }
    void push(int t, double xe[500][10], double ye[500][10], double ze[500][10],
                double vex[500][10], double vey[500][10], double vez[500][10],
                double xi[500][10] , double yi[500][10] , double zi[500][10],
                double vix[500][10], double viy[500][10], double viz[500][10],
                double Ex[10][10][10], double Ey[10][10][10], double
Ez[10][10][10]);
};

#endif // POS_VELOCT_H
```

Pos_veloct.cpp::

```
//
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
// useful libraries
```

```
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
//
#include "pos_veloct.h"
#include "parameter.h"
//
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
// Global parameters
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
//
//int maxTime = 50 ;
//int nb_p = 500 ;
//int ngrid = 50 ;
//
//double pi = 3.14159265359 ;
//
//double grid_length = 2*pi ;
//double dstep = grid_length / ngrid ;
//double dpgrid = grid_length / nb_p ; //particle spacing
//double charge = -grid_length / nb_p ; //pseudo-particle charge normalised to
give ncrit=1
//double mass = grid_length / nb_p ; //pseudo-particle mass
//double imp0 = 377.0 ;
//double Cdtds = 1.0 ; // Courant number
//double mu_r=1, mu_0 = 4*3.14159265359e-7 ;
//double q_over_me=-1.0 ;
//double dt = 0.05 ; //normalised timestep
//
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
//
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
//
parameter param2;
void Pos_veloct::push(int t, double xe[500][10], double ye[500][10], double
ze[500][10],
                double vex[500][10], double vey[500][10], double vez[500][10],
                double xi[500][10] , double yi[500][10] , double zi[500][10],
                double vix[500][10], double viy[500][10], double viz[500][10],
                double Ex[10][10][10], double Ey[10][10][10], double
Ez[10][10][10]){
    //
//    int nb_p = 500 ;
//    int ngrid = 10 ;
//    double grid_length = 1.0 ;
//    double dstep = grid_length / ngrid ;
    double q_over_me=-1.0 ;
    double dt = 0.05 ; //normalised timestep
    //
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    // interpolate field E from grid to particle (B<<E)
```

```
    //
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!
    //
    double xa, b1, b2, exi ;
    int j1, j2 ;
    //
    for(int l = 0 ; l < param2.nb_p ; l++){
        //
        exi = 0 ;
        xa = xe[l][t-1] / param2.dstep ;
        j1 = int(xa) ;
        j2 = j1 + 1 ;
        b2 = xa - j1 ;
        b1 = 1.0 - b2 ;
        for(int nn=0;l<param2.ngrid;l++){
            for(int pp=0;l<param2.ngrid;l++){
                if( (j1 <= param2.ngrid-2)&&(j2 <= param2.ngrid-2) ){
                    exi = exi + b1*Ex[j1][nn][pp] + b2*Ex[j2][nn][pp] ;
                } else if( (j1 > param2.ngrid-2)&&(j2 <= param2.ngrid-2) ){
                    exi = exi + b1*Ex[j1-(param2.ngrid-2)][nn][pp] +
b2*Ex[j2][nn][pp] ;
                } else if( (j1 <= param2.ngrid-2)&&(j2 > param2.ngrid-2) ){
                    exi = exi + b1*Ex[j1][nn][pp] + b2*Ex[j2-(param2.ngrid-
2)][nn][pp] ;
                } else if( (j1 > param2.ngrid-2)&&(j2 > param2.ngrid-2) ){
                    exi = exi + b1*Ex[j1-(param2.ngrid-2)][nn][pp] + b2*Ex[j2-
(param2.ngrid-2)][nn][pp] ;
                }
            }
        }
        // update velocities
        vex[l][t] = vex[l][t-1] + q_over_me * dt * exi ;
        vey[l][t] = vey[l][t-1] ;
        vez[l][t] = vez[l][t-1] ;
        // update positions
        xe[l][t] = xe[l][t-1] + vex[l][t] * dt ;
        ye[l][t] = ye[l][t-1] + vey[l][t] * dt ;
        ze[l][t] = ze[l][t-1] + vez[l][t] * dt ;
        // freeze approximation for ions
        vix[l][t] = vix[l][t-1] ;
        viy[l][t] = viy[l][t-1] ;
        viz[l][t] = viz[l][t-1] ;
        xi[l][t] = xi[l][t-1] + vix[l][t] * dt ;
        yi[l][t] = yi[l][t-1] + viy[l][t] * dt ;
        zi[l][t] = zi[l][t-1] + viz[l][t] * dt ;
        //
        // periodic boundaries
        //
        if( xe[l][t] < 0 ){
          xe[l][t] = xe[l][t] + param2.grid_length ;
        } else if( xe[l][t] >= param2.grid_length ){
          xe[l][t] = xe[l][t] - param2.grid_length ;
        }
        //
        if( ye[l][t] < 0 ){
          ye[l][t] = ye[l][t] + param2.grid_length ;
        } else if( ye[l][t] >= param2.grid_length ){
          ye[l][t] = ye[l][t] - param2.grid_length ;
        }
        //
        if( ze[l][t] < 0 ){
```

```
          ze[l][t] = ze[l][t] + param2.grid_length ;
        } else if( ze[l][t] >= param2.grid_length ){
          ze[l][t] = ze[l][t] - param2.grid_length ;
        }
        //
        if( xi[l][t] < 0 ){
          xi[l][t] = xi[l][t] + param2.grid_length ;
        } else if( xi[l][t] >= param2.grid_length ){
          xi[l][t] = xi[l][t] - param2.grid_length ;
        }
        //
        if( yi[l][t] < 0 ){
          yi[l][t] = yi[l][t] + param2.grid_length ;
        } else if( yi[l][t] >= param2.grid_length ){
          yi[l][t] = yi[l][t] - param2.grid_length ;
        }
        //
        if( zi[l][t] < 0 ){
          zi[l][t] = zi[l][t] + param2.grid_length ;
        } else if( zi[l][t] >= param2.grid_length ){
          zi[l][t] = zi[l][t] - param2.grid_length ;
        }
    }
// end void
}
```

Annex B:: code python

```
# -*- coding: utf-8 -*-

""" ESPIC: a simple 1D1V electrostatic PIC code
    Based on 'espic.c' developed for 2000 Heraeus School, Jena

  (c) P. Gibbon, KU-Leuven & FZ-Juelich, November 2013

"""
import numpy as np  ## numeric routines; arrays
import pylab as plt  ## plotting


""" core routines
"""

#================================================================================
#  Particle loading - positions
#================================================================================

def loadx(bc_particle):
    global dx, grid_length, rho0, npart, q_over_me, a0
    global charge,mass,wall_left,wall_right
    print("Load particles")

# set up particle limits
    if (bc_particle >= 2):
        # reflective boundaries
        # place particle walls half a mesh spacing inside field boundaries
        wall_left = dx/2.
        wall_right = grid_length-3*dx/2.
        plasma_start = wall_left
        plasma_end = wall_right   # actually want min(end,wr) */
```

```
    else:
        # periodic boundaries
        plasma_start = 0.
        plasma_end = grid_length
        wall_left = 0.
        wall_right = grid_length

    xload = plasma_end - plasma_start  # length for particle loading */
    dpx = xload/npart                    #  particle spacing */
    charge = -rho0*dpx          #  pseudo-particle charge normalised to give
ncrit=1 (rhoc=-1)
    mass = charge/q_over_me  #  pseudo-particle mass (need for kinetic energy
diagnostic)

    for i in range(npart):
        x[i] = plasma_start + dpx*(i+0.5)  # Python ndarrays start at index 0
        x[i] += a0*np.cos(x[i]) # Include small perturbation

    return True



#===============================================================================
#  Particle loading - velocities
#===============================================================================

def loadv(idist,vte):
    global npart,v,grid_length,v0
    print("Set up velocity distribution")
    iseed = 76523  # random number seeds
    idum1 = 137

    if ( idist == 1 ):
       # >10 = set up thermal distribution
        for i in range(npart):
            vm = vte*np.sqrt( (-2.*np.log((i+0.5)/npart)) ) # inverted 2v-
distribution - amplitude */
            rs = np.random.random_sample() # random angle */
            theta = 2*np.pi*rs
            v[i] = vm*np.sin(theta) # x-component of v

      # scramble particle indicies to remove correlations between x and v
        np.random.shuffle(v)

    else:
        # Default is cold plasma */
         v[1:npart] = 0.

# add perturbation
    v += v0*np.sin(2*np.pi*x/grid_length)
    return True

#===============================================================================
#   Compute densities
#===============================================================================

def density(bc_field,qe):
   global x,rhoe,rhoi,dx,npart,ngrid,wall_left,wall_right
   j1=np.dtype(np.int32)
   j2=np.dtype(np.int32)

   re = qe/dx #  charge weighting factor
```

```
    rhoe=np.zeros(ngrid+1)          # electron density
     # map charges onto grid
    for i in range(npart):
       xa = x[i]/dx
       j1 = int(xa)
       j2 = j1 + 1
       f2 = xa - j1
       f1 = 1.0 - f2
       rhoe[j1] = rhoe[j1] + re*f1
       rhoe[j2] = rhoe[j2] + re*f2

    if (bc_field == 1):
       #  periodic boundaries */
       rhoe[0] += rhoe[ngrid]
       rhoe[ngrid] = rhoe[0]

    elif (bc_field == 2):
      #  reflective - 1st and last (ghost) cells folded back onto physical grid
*/
       iwl = wall_left/dx
       rhoe[iwl+1] += rhoe[iwl]
       rhoe[iwl] = 0.0
       iwr = wall_right/dx
       rhoe[iwr] += rhoe[iwr+1]
       rhoe[iwr+1] = rhoe[iwr]
    else:
       print("Invalid value for bc_field:", bc_field)

#  Add neutral ion density
   rhoi = rho0
#   print(rhoe[0:ngrid+1])
   return True

#===============================================================================
#  Compute electrostatic field
#===============================================================================

def field():
  global rhoe,rhoi,ex,dx,ngrid

  rhot=rhoe+rhoi  # net charge density on grid

  # integrate div.E=rho directly (trapezium approx)
  # end point - ex=0 mirror at right wall

  Ex[ngrid]=0.          # zero electric field
  edc = 0.0

  for j in range(ngrid-1,-1,-1):
      Ex[j] = Ex[j+1] - 0.5*( rhot[j] + rhot[j+1] )*dx
      edc = edc + Ex[j]

  if (bc_field == 1):
    # periodic fields:  subtract off DC component */
    #  -- need this for consistency with charge conservation */
      Ex[0:ngrid] -= edc/ngrid
      Ex[ngrid] = Ex[0]

  return True


#===============================================================================
```

```
#   Particle pusher
#==============================================================================

def push():
    global x,v,Ex,dt,dx,npart,q_over_me

    for i in range(npart):

      #  interpolate field Ex from grid to particle */
      xa = x[i]/dx
      j1 = int(xa)
      j2 = j1 + 1
      b2 = xa - j1
      b1 = 1.0 - b2
      exi = b1*Ex[j1] + b2*Ex[j2]
      v[i] = v[i] + q_over_me*dt*exi       #  update velocities */


    x += dt*v    #  update positions (2nd half of leap-frog)

    return True


#==============================================================================
#   check particle boundary conditions
#==============================================================================

def particle_bc(bc_particle,xl):
   global x
#  int iseed1 = 28631;        /* random number seed */
#  int iseed2 = 1631;         /* random number seed */


  #  loop over all particles to see if any have
  #     left simulation region: if so, we put them back again
  #     according to the switch 'bc_particle' **/

   if (bc_particle == 1):
     #  periodic
     for i in range(npart):
       if ( x[i] < 0.0 ):
         x[i] += xl
       elif ( x[i] >= xl ):
         x[i] -= xl

   elif (bc_particle == 2):
     # reflective
     print("Reflective boundaries not implemented yet.")

   elif (bc_particle == 3):
     # reflective
     print("Thermal boundaries not implemented yet.")

   else:
     print("Invalid value for bc_particle:", bc_particle)

   return True


#==============================================================================
#   Diagnostic outputs for fields and particles
#==============================================================================
```

```
def diagnostics():
    global rhoe,Ex,ngrid,itime,grid_length,rho0,a0
    global ukin, upot, utot, udrift, utherm, emax,fv,fm
    global iout,igraph,iphase,ivdist
    xgrid=dx*np.arange(ngrid+1)
    if (itime==0):
        plt.figure('fields')
        plt.clf()
    if (igraph > 0):
      if (np.fmod(itime,igraph)==0): # plots every igraph steps
    # Net density
        plt.subplot(2, 2, 1)
        if (itime >0 ): plt.cla()
        plt.plot(xgrid, -(rhoe+rho0), 'r', label='density')
        plt.xlabel('x')
        plt.xlim(0,grid_length)
        plt.ylim(-2*a0,2*a0)
        plt.legend(loc=1)
    # Electric field
        plt.subplot(2, 2, 2)
        if (itime >0 ): plt.cla()
        plt.plot(xgrid, Ex, 'b', label='Ex')
        plt.xlabel('x')
        plt.ylim(-2*a0,2*a0)
        plt.xlim(0,grid_length)

        plt.legend(loc=1)

        if (iphase > 0):
          if (np.fmod(itime,iphase)==0):
    # Phase space plots every iphase steps
            axScatter = plt.subplot(2, 2, 3)
            if (itime >0 ): plt.cla()
            axScatter.scatter(x,v,marker='.',s=1)
    #     axScatter.set_aspect(1.)
            axScatter.set_xlim(0,grid_length)
            axScatter.set_ylim(-vmax,vmax)
            axScatter.set_xlabel('x')
            axScatter.set_ylabel('v')

        if (ivdist > 0):
          if (np.fmod(itime,ivdist)==0):
    # Distribution function plots every ivdist steps
            fv=np.zeros(nvbin+1)          # zero distn fn
            dv = 2*vmax/nvbin    # bin separation */
            for i in range(npart):
                vax= ( v[i] + vmax )/dv    # norm. velocity */
                iv = int(vax)+1 # bin index */
                if (iv <= nvbin and iv > 0): fv[iv] +=1 #   /* increment dist. fn
if within range

            plt.subplot(2, 2, 4)
            if (itime >0 ): plt.cla()
            vgrid=dv*np.arange(nvbin+1)-vmax
            plt.plot(vgrid, fv, 'g', label='f(v)')
            plt.xlabel('v')
            plt.xlim(-vmax,vmax)
#         plt.ylim(-2*a0,2*a0)
            plt.legend(loc=1)
            fn_vdist = 'vdist_%0*d'%(5, itime)
```

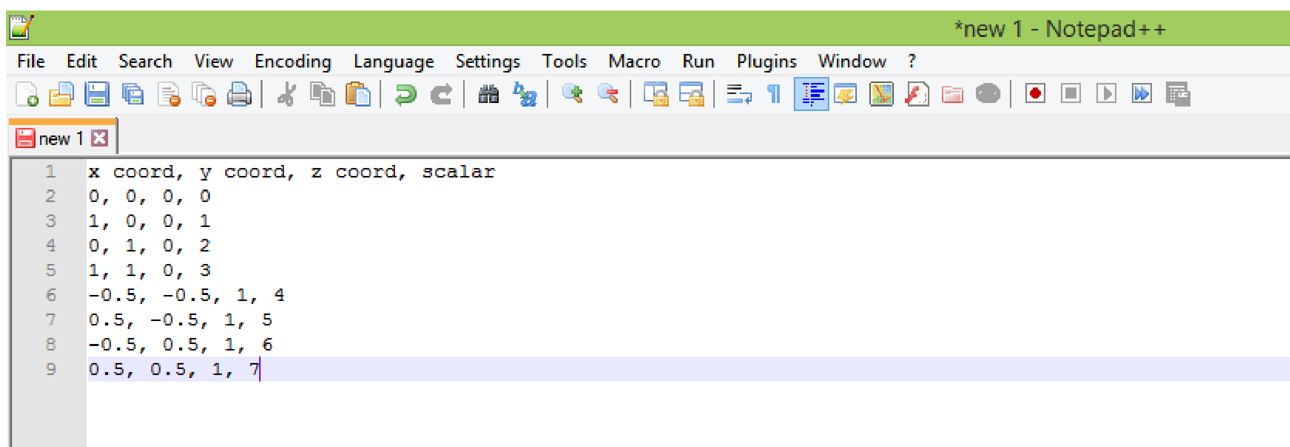
```
            np.savetxt(fn_vdist,
np.column_stack((vgrid,fv)),fmt=('%1.4e','%1.4e'))    # write to file


        plt.pause(0.0001)
        plt.draw()
        filename = 'fields_%0*d'%(5, itime)
        if (iout > 0):
          if (np.fmod(itime,iout)==0):  # printed plots every iout steps
            plt.savefig(filename+'.png')

#   total kinetic energy
    v2=v**2
    vdrift=sum(v)/npart
    ukin[itime] = 0.5*mass*sum(v2)
    udrift[itime] = 0.5*mass*vdrift*vdrift*npart
    utherm[itime] = ukin[itime] - udrift[itime]

#   potential energy
    e2=Ex**2
    upot[itime] = 0.5*dx*sum(e2)
    emax = max(Ex) # max field for instability analysis */

#  total energy
    utot[itime] = upot[itime] + ukin[itime]

    return True

#==============================================================================
#    Plot time-histories
#==============================================================================

def histories():

#FILE *history_file;     /* file for writing out time histories */

  global ukin, upot, utot, udrift, utherm
  xgrid=dt*np.arange(nsteps+1)
#  plt.clf()
  plt.figure('Energies')
#  plt.subplot(2, 2, 1)
  plt.plot(xgrid, upot, 'b', label='Upot')
  plt.plot(xgrid, ukin, 'r', label='Ukin')
  plt.plot(xgrid, utot, 'black', label='Utot')
#  plt.plot(xgrid, udrift, 'g', label='Udrift')
  plt.xlabel('t')
  plt.ylabel('Energy')

#  plt.xlim(0,grid_length)
#  plt.ylim(-2*a0,2*a0)
  plt.legend(loc=1)
  plt.savefig('energies.png')

#   write energies out to file */
  np.savetxt('energies.out',
np.column_stack((xgrid,upot,ukin,utot)),fmt=('%1.4e','%1.4e','%1.4e','%1.4e'))
# x,y,z equal sized 1D arrays
#  fohist = open("energies.data","w")
#  str.format("{0:<10.5f}", 3.14159265)
#   if (itime==1) {fprintf(history_file, " t, U_drift, U_therm, U_field,
U_total, Emax\n");}
```

```
#   fprintf( history_file, "%f  %e %e  %e  %e %e\n", itime*dt, udrift, utherm,
upot, utot, emax );


#==============================================================================
#  Main program
#==============================================================================


npart=10000           # particles
ngrid=100             # grid points
nsteps=100           # timesteps

# particle arrays
x = np.zeros(npart)  # positions
v = np.zeros(npart) # velocities#    particle_bc()

# grid arrays
rhoe=np.zeros(ngrid+1)        # electron density
rhoi=np.zeros(ngrid+1)        # ion density
Ex=np.zeros(ngrid+1)          # electric field
phi=np.zeros(ngrid+1)         # potential
# time histories
ukin=np.zeros(nsteps+1)
upot=np.zeros(nsteps+1)
utherm=np.zeros(nsteps+1)
udrift=np.zeros(nsteps+1)
utot=np.zeros(nsteps+1)

# Define main variables and defaults
# ----------------------------------

#ni = 0;          /* # ions (fixed) */
grid_length = 2.0*np.pi  # size of spatial grid
#grid_length = 16  # size of spatial grid
plasma_start = 0.          # LH plasma edge
plasma_end = grid_length    # RH plasma edge
dx = grid_length/ngrid
dt = 0.05               # normalised timestep
q_over_me=-1.0         # electron charge:mass ratio
rho0 = 1.0             # background ion density
vte = 0.02             # thermal velocity
nvbin=50              # bins for f(v) plot
a0 = 0.1               # perturbation amplitude
vmax = 0.2         # max velocity for f(v) plot
v0=0.0                # velocity perturbation
wall_left=0.
wall_right=1.
bc_field = 1         #  field boundary conditions:  1 = periodic
                      #                              2 = reflective
bc_particle = 1       # particle BCs:  1 = periodic
                      #                2 = reflective
                      #                3 = thermal
profile = 1        # density profile switch
distribution = 1  # velocity distribution switch: 0 = cold, 1=thermal
             #                         2 = 2-stream
ihist = 5          # frequency of time-history output
igraph = int(np.pi/dt/16)       # freq. of graphical snapshots
iphase = igraph
ivdist = -igraph
iout = igraph*1         # freq. of saved graphics files
```

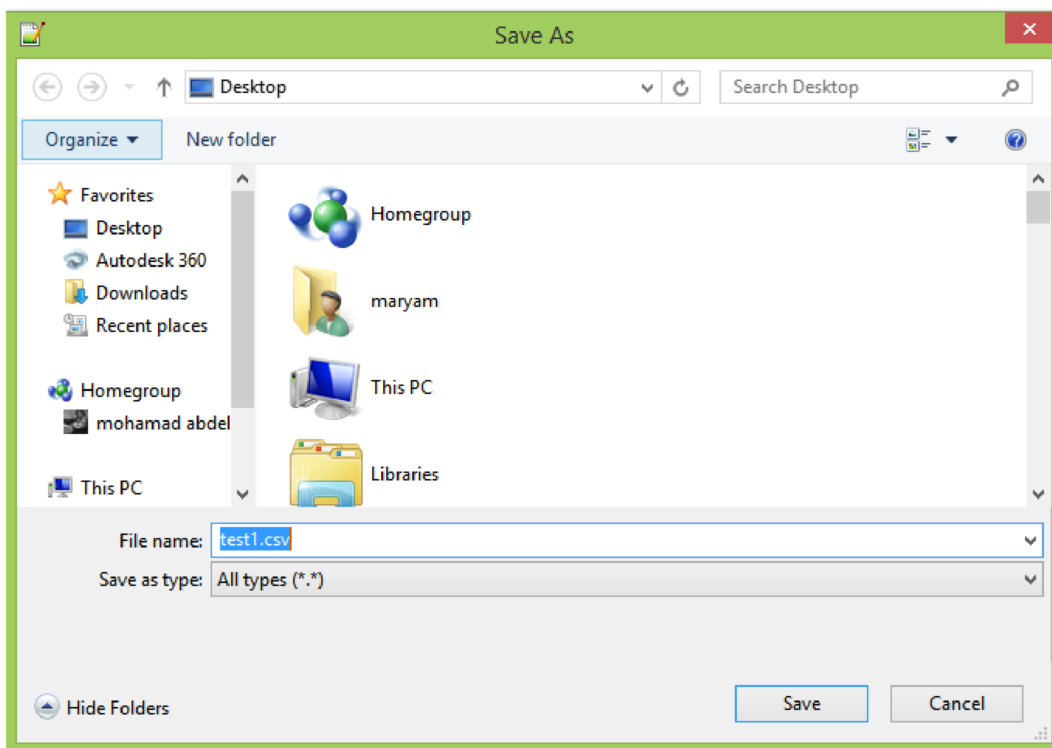
```
itime = 0          # initialise time counter


#  Setup initial particle distribution and fields
#  ----------------------------------------------

loadx(bc_particle)              # load particles onto grid
loadv(distribution,vte)         # define velocity distribution
x += 0.5*dt*v                   #  centre positions for 1st leap-frog step
particle_bc(bc_particle,grid_length)
density(bc_field,charge)          # compute initial density from particles
field()                       # compute initial electric field
diagnostics()                   # output initial conditions
print('resolution dx/\lambda_D=',dx/vte)

#  Main iteration loop
#  -------------------

for itime in range(1,nsteps+1):
    print('timestep ',itime)
    push()                   # Push particles
    particle_bc(bc_particle,grid_length) # enforce particle boundary conditions
    density(bc_field,charge)        # compute density
    field()                    # compute electric field (Poisson)
    diagnostics()    # output snapshots and time-histories

histories()    # Produce time-history plots

#raw_input("Press key to finish")
#plt.close()
print('done')
```
Annex C::

7.2 Paraview Input files

The type of files we are using to read our solution via Para view is the .csv files (comma separated variables). In this section we'll show a simple example (8 points). First start with defining the .csv file using notepad++ as shown in the figure bellows.

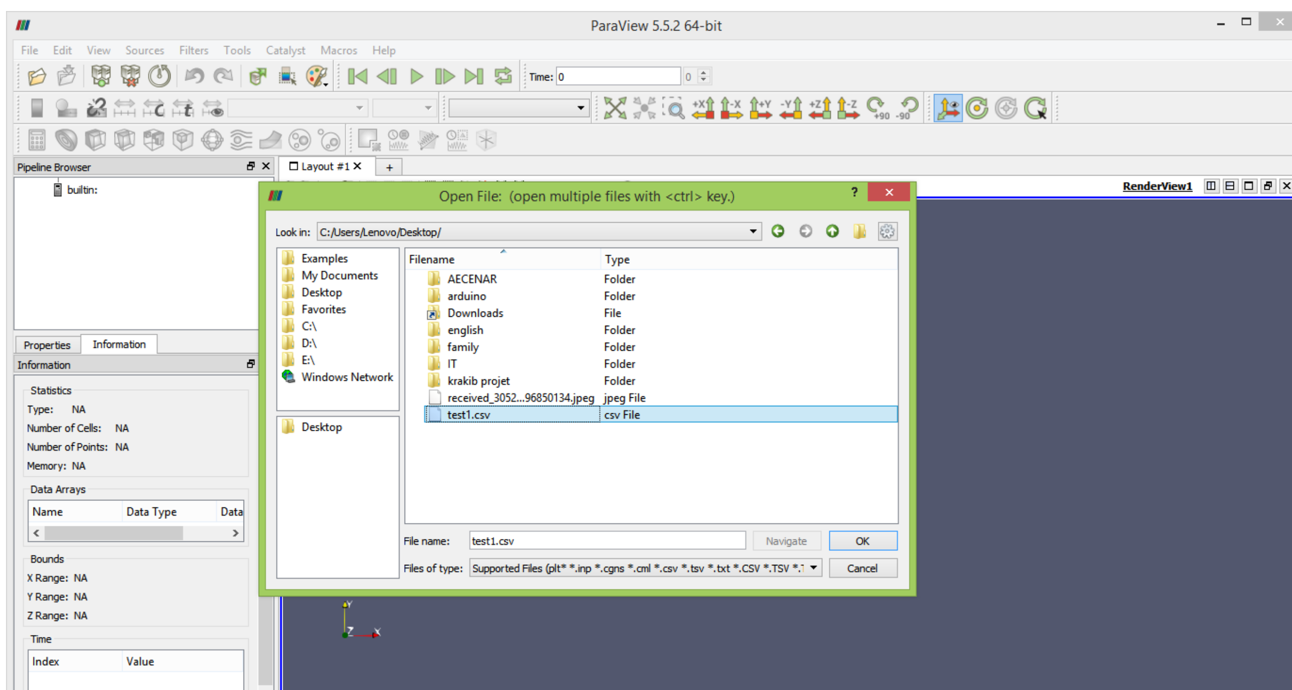


Then save your file as: All types (*. *) as shown in our example test1.csv.



Now it’s ready to be opened in Para view.

Select the open button and choose your file .csv.

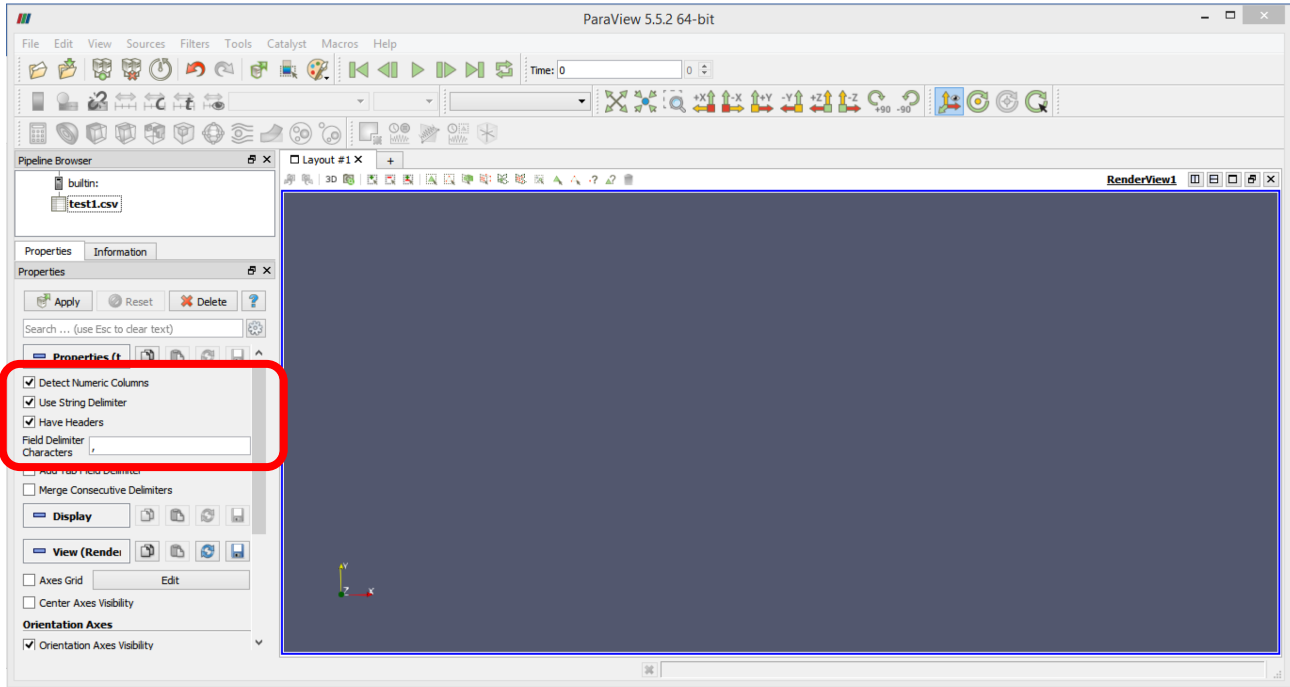


Start Para View, and read in this data. Note that the default settings should be used:

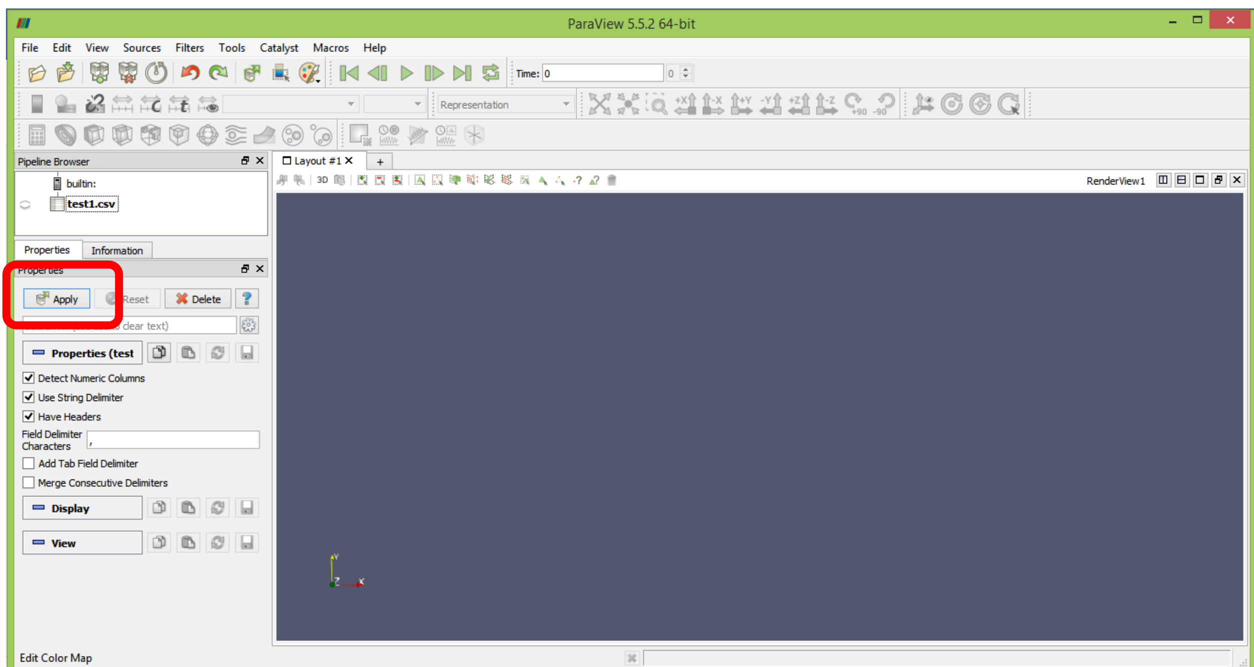
- Detect Numeric Columns ON
- Use String Delimiter ON

- Have Headers ON
- Field Delimiter Characters should be a comma - ','

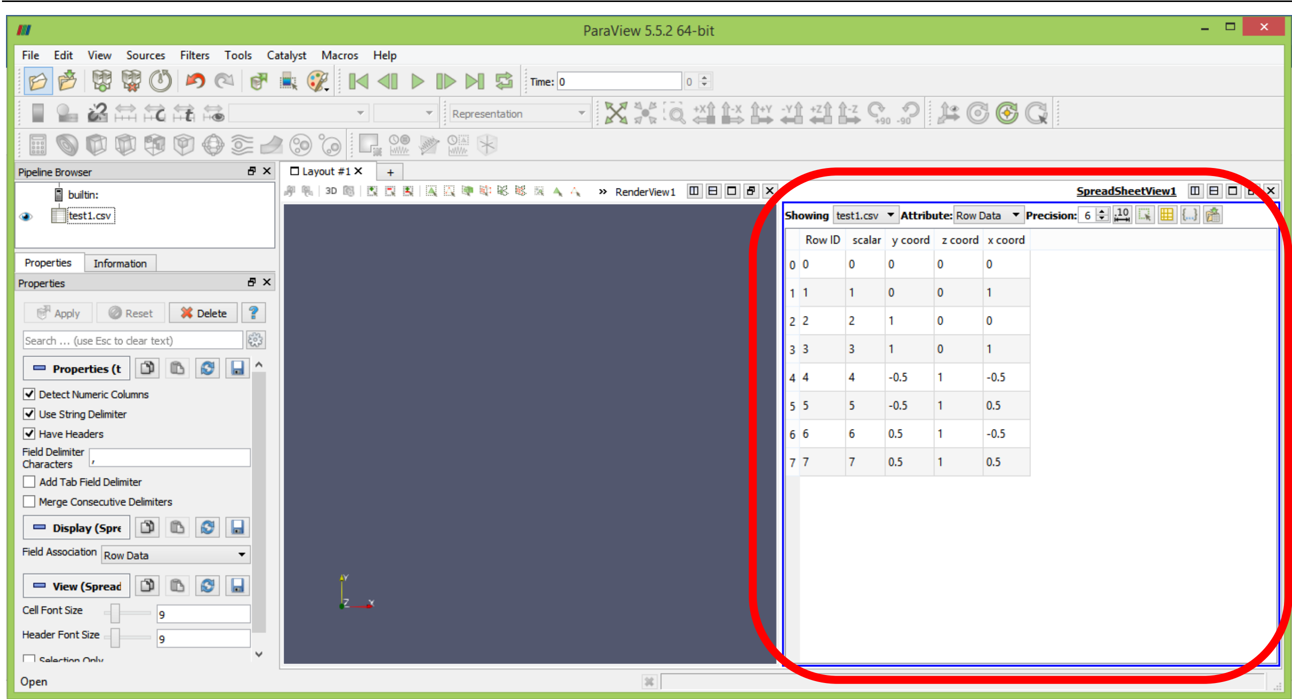
(See figure below)



Then press apply.

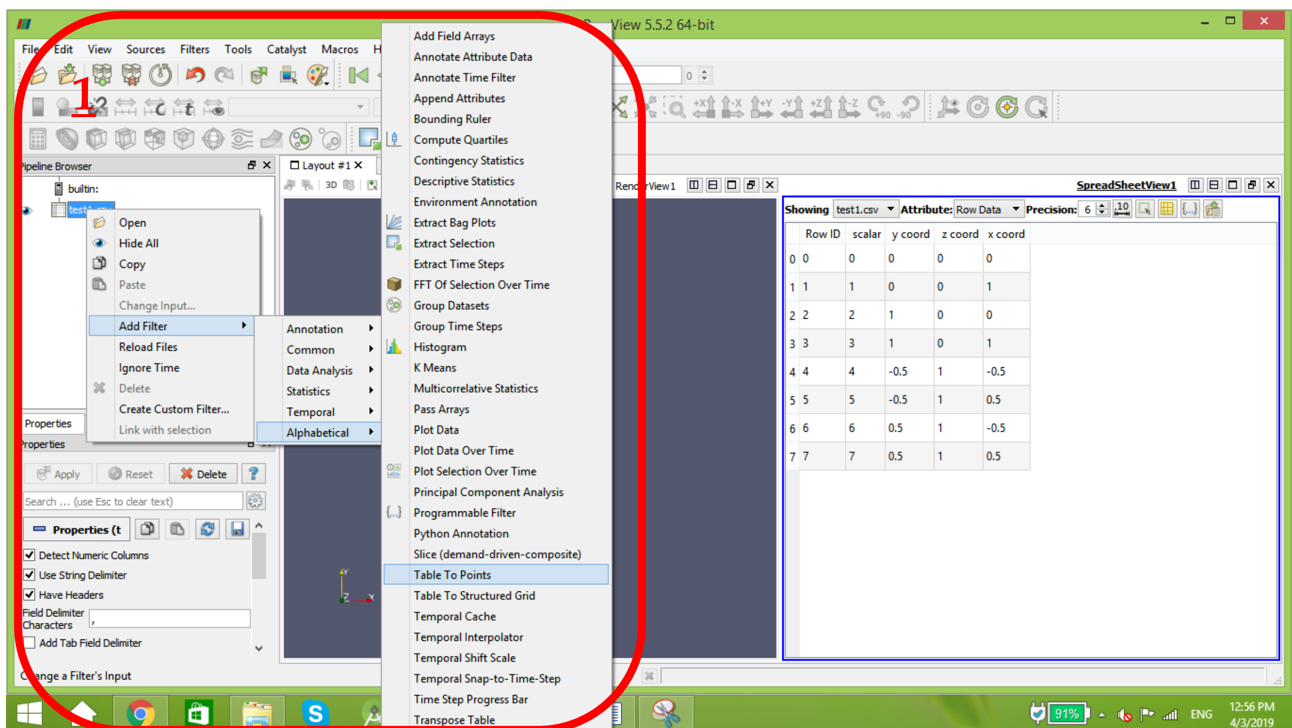


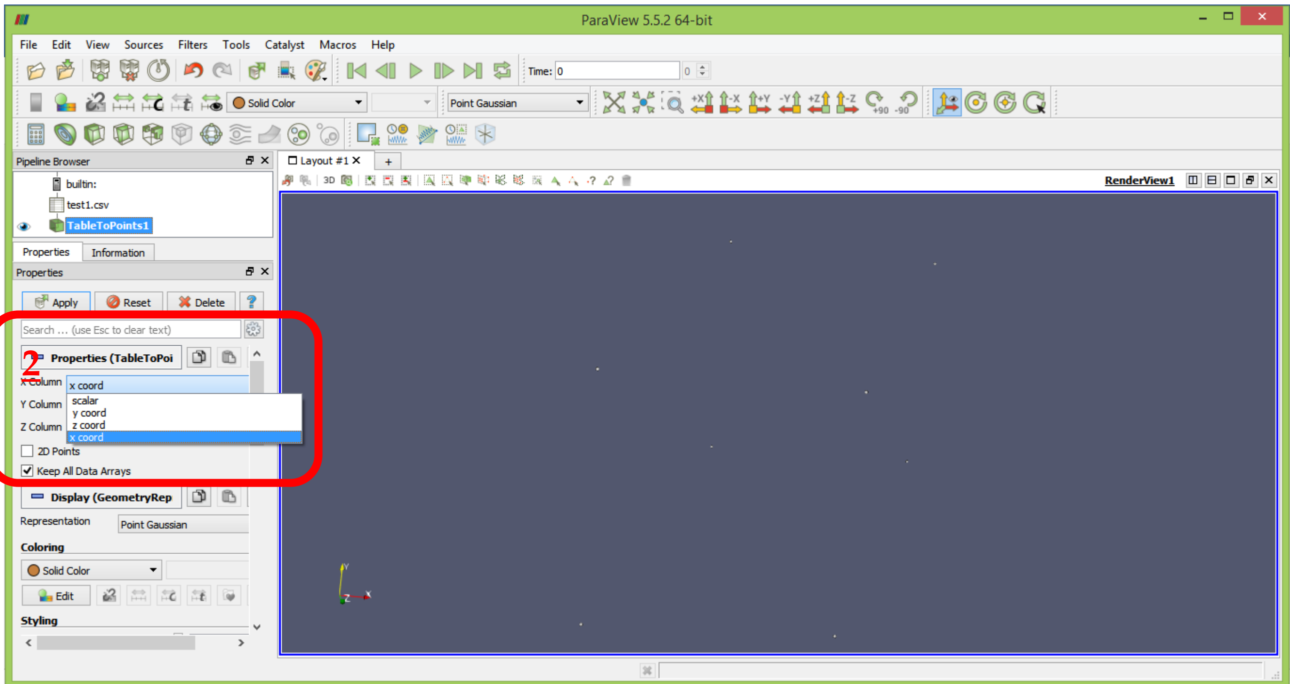
The data should show up as a table.



I. Displaying data as points

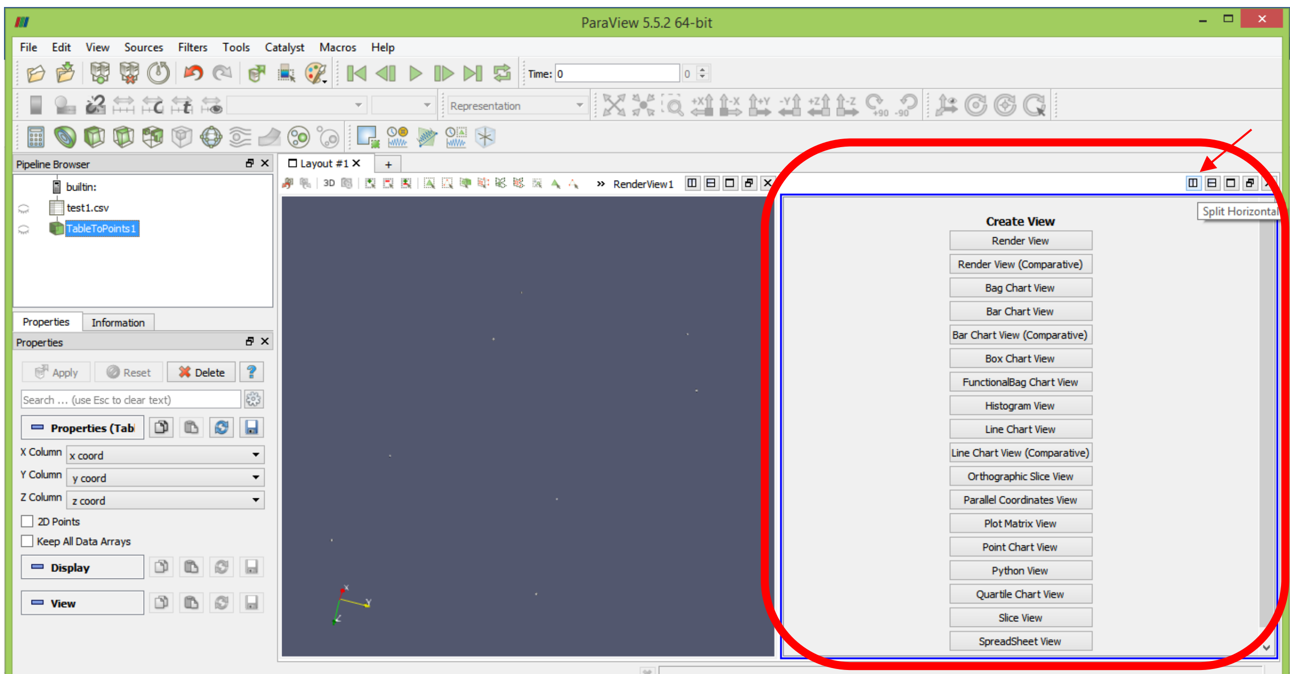
- Run the filter Filters/ Alphabetical/ Table to Points (right click on the table at the left as shown in the figure below).
- Tell Para View what columns are the X, Y and Z coordinate. Be sure to not skip this step. Apply.





Press apply and the points are visible now.

If your points didn't show up press on "split horizontal" button. And choose the desired view.



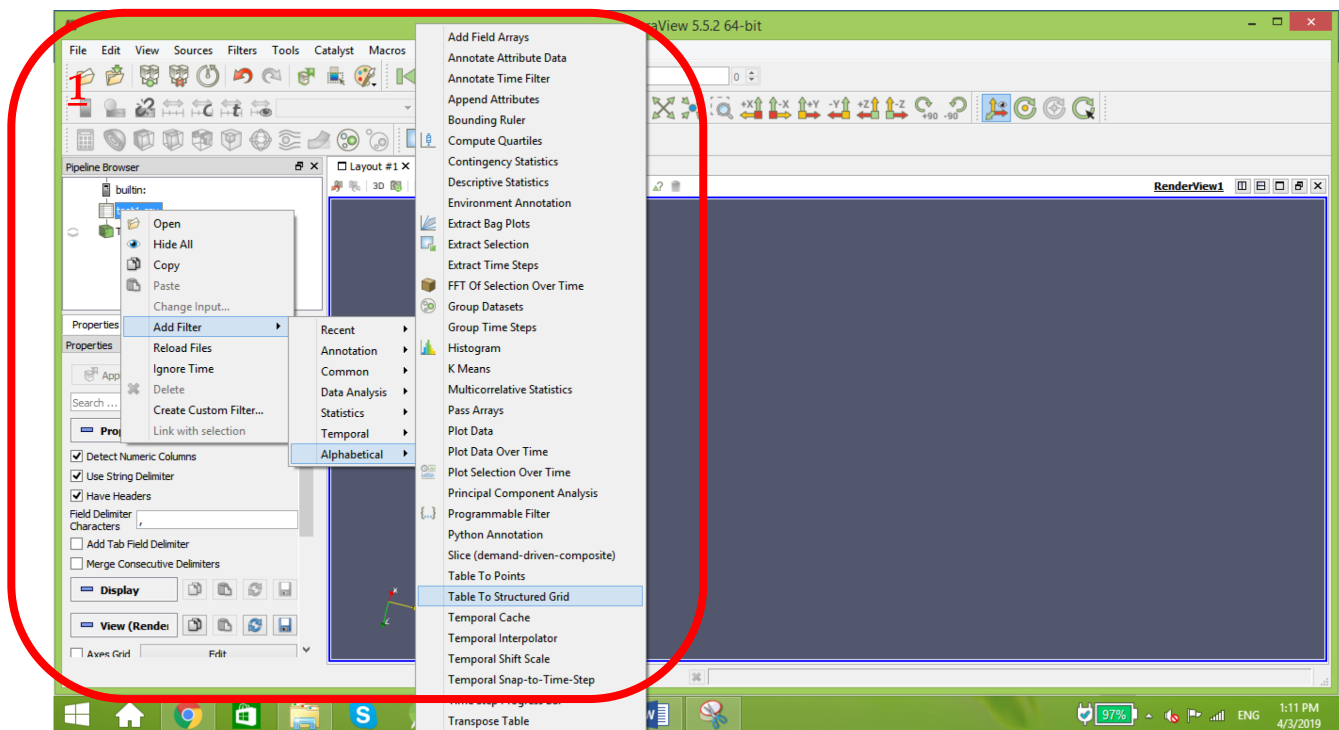
II. Displaying data as structured grid

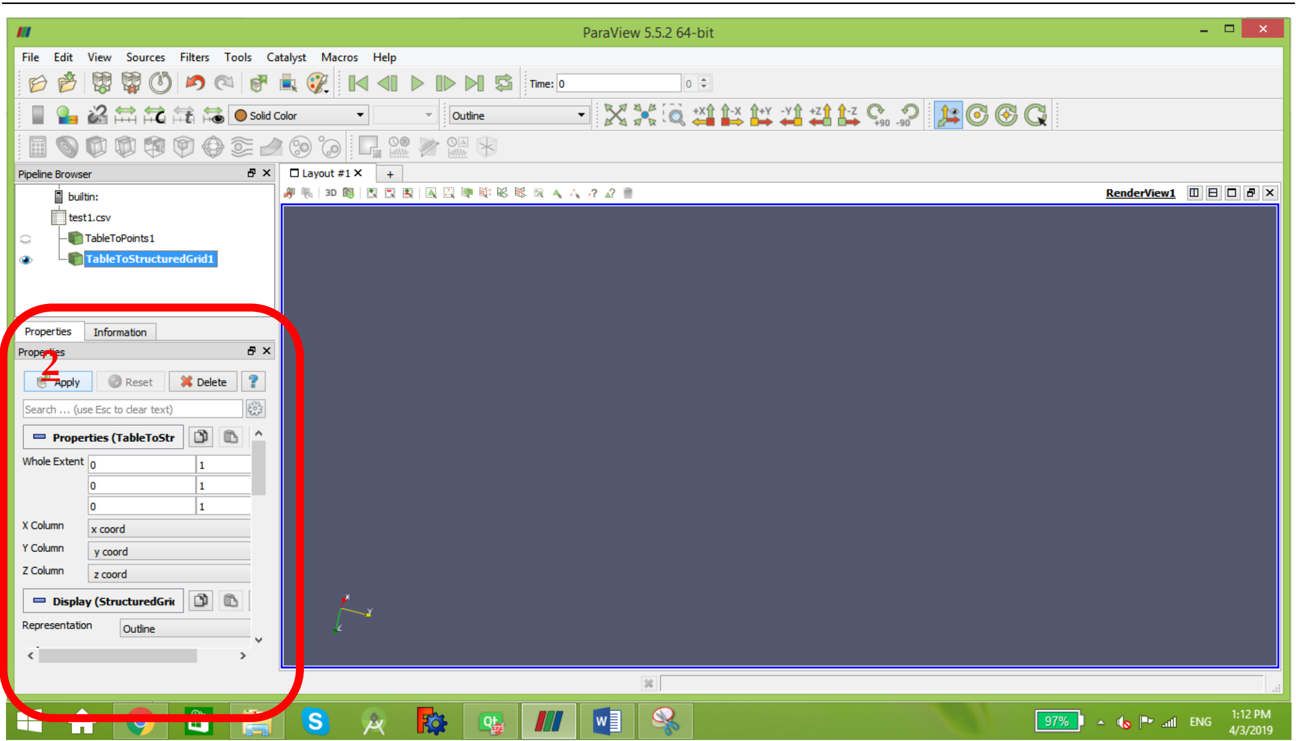
1. Run the filter Filters/ Alphabetical/ Table to Structured Grid.

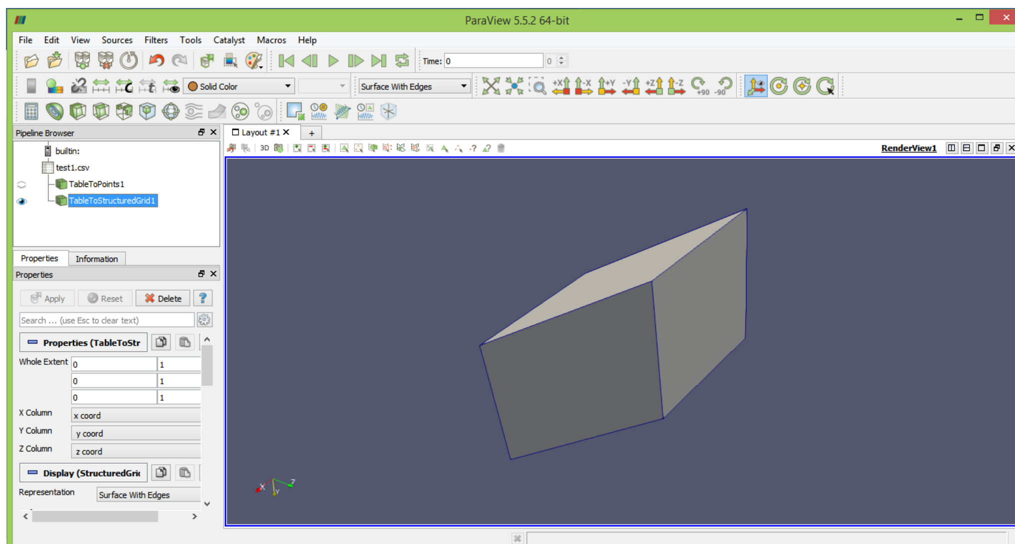
2. Tell Para View what extent, or array sizes, your data is in. For instance, the data above has 8 points, forming a leaning cube. Points arrays are in X == size 2, Y == size 2, and Z == size 2. In this example we will use C indexing for the arrays, thus they go from 0 to 1 (2 entries).

   - Whole extent is as follows:
   - 0 1
   - 0 1
   - 0 1

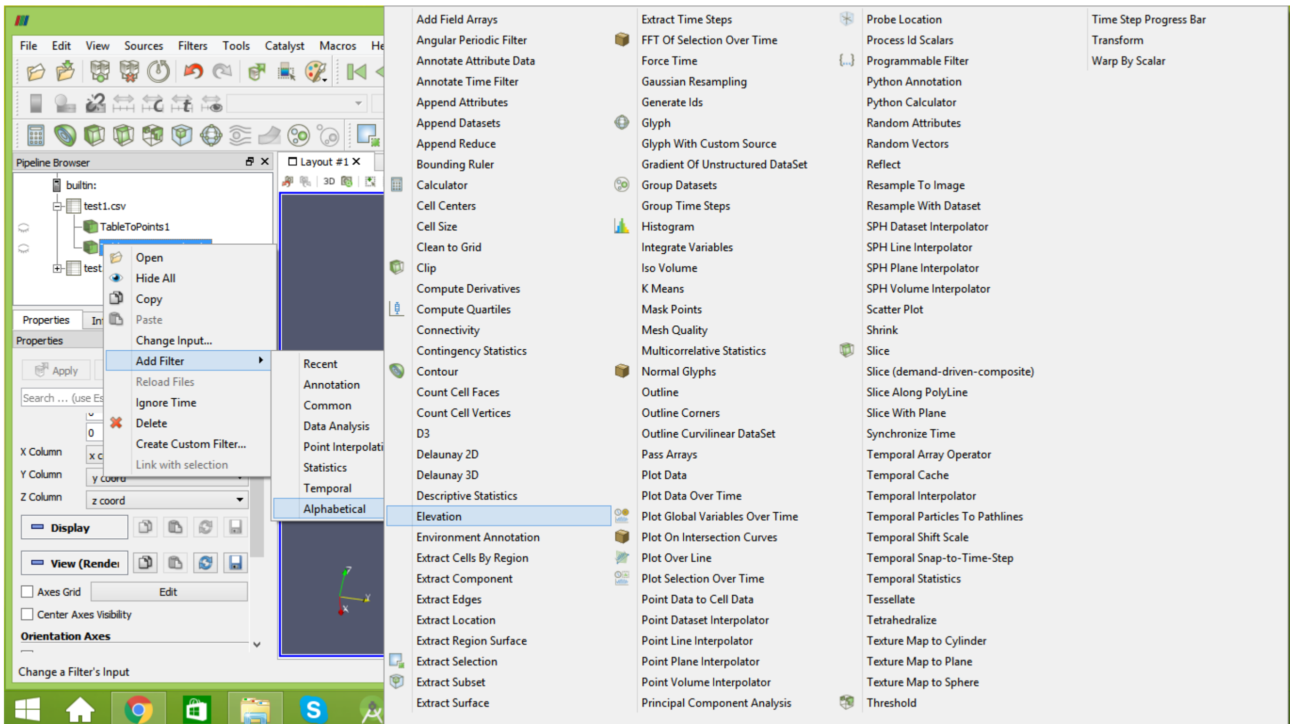
3. Tell Para View what columns are the X, Y and Z coordinate. Be sure to not skip this step. Apply.



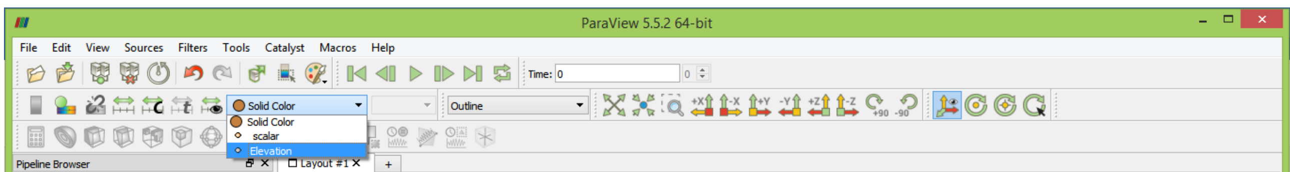

ParaView 5.5.2 64-bit
File Edit View Sources Filters Tools Catalyst Macros Help
Time: 0
Solid Color
Outline
Pipeline Browser
builtin:
test1.csv
TableToPoints1
TableToStructuredGrid1
Layout #1
RenderView1
Properties Information
2
Apply Reset Delete
Search ... (use Esc to clear text)
Properties (TableToStr
Whole Extent
X Column x coord
Y Column y coord
Z Column z coord
Display (StructuredGrid
Representation Outline
97%
ENG
1:12 PM
4/3/2019


ParaView 5.5.2 64-bit
File Edit View Sources Filters Tools Catalyst Macros Help
Solid Color
Surface With Edges
Pipeline Browser
builtin:
test1.csv
TableToPoints1
TableToStructuredGrid1
Layout #1
RenderView1
Properties Information
Apply Reset Delete
Properties (TableToStr
Whole Extent
X Column x coord
Y Column y coord
Z Column z coord
Display (StructuredGrid
Representation Surface With Edges

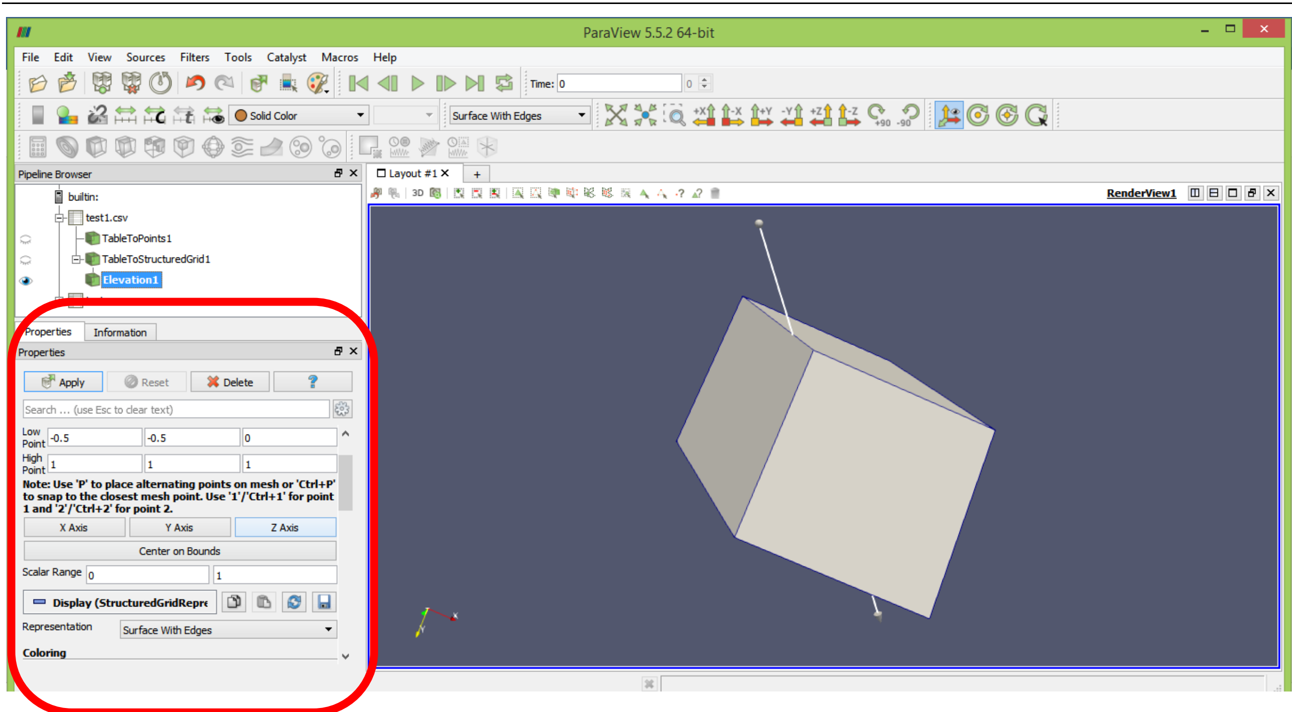
Now to represent yours results with colors, right click on “table to structure”→ add filter → Alphabetic→elevation.



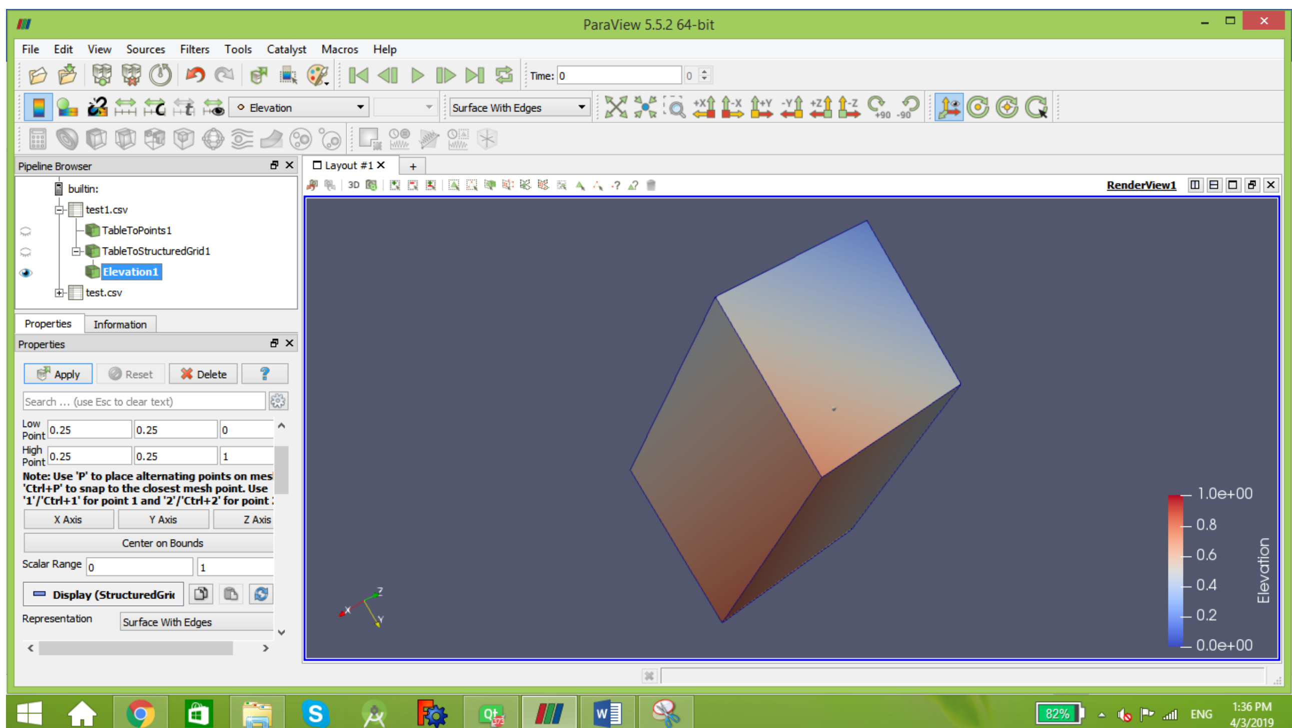
Make sure that you select the elevation button.



Finally choose the desired axis, then apply.



Now it’s ready.



III. Saving Results

A simple code has been added to the program to save the results automatically into files.csv.

```
#include "iostream"
#include "fstream"
#include "sstream"
```

```
string filename;

ofstream myfile;

stringstream b;
b<<i;//number of iteration
    filename= "SN_"+ b.str();
    filename+= ".csv";
    myfile.open(filename.c_str());

  for(int k=0;k<d;k++){
       rold[k]=r[k];//store the old radii
       accel[k]=dyn.acc(mass[k],r[k],rho[k],dp_over_dr[k],G);
       v[k]=v[k]*damping+accel[k] * dt;
       r[k]=r[k]+v[k] * dt+accel[k] * dt * dt;
       y=sqrt(r[k]*r[k]-(k+1)*(k+1));
       if(r[k]<0){
           r[k]=0;
       }

myfile<<0<<","<<r[k]<<",0,"<<r[k]<<","<<T[k]<<","<<dt<<","<<pressure[k]<<"\n";

    }
  myfile.close();
```

This code will create a file named SN000.csv for each iteration. They will be saved in the build folder.