A practical introduction to the Particle in Cell simulation of plasma acceleration

Francesco Massimo

November 11, 2021

Contents

1 Introduction

In this practical you will familiarize with a Particle in Cell code (PIC) code, learn how to set up and run a PIC simulation of plasma acceleration, and how to analyze the results.

The PIC code you will use, Smilei, is not a reduced version, but a full version that you can use for your future studies. Although here you will be given all the instructions you need to use the code for the purposes of this practical, you can find more details and more instructions on the code use here `https://smileipic.github.io/Smilei/index.html`, or tutorials in `https://smileipic.github.io/tutorials/`.

At the end of the practical you will be able to run and analyse a full PIC simulation of a plasma acceleration set-up, where an intense laser pulse is injected in the plasma. The laser pulse excites a relativistic plasma wave that accelerates a properly injected electron beam behind the laser pulse. This plasma acceleration scheme is called laser wakefield acceleration (or LWFA). A common analogy is that of a surfer (the injected electrons) being accelerated by the waves in the wake of a boat (the laser pulse) in the sea (the plasma). In this particular LWFA set-up the a relativistic electron beam is externally injected in the plasma wave.

However, you won't simulate immediately this case study. You will arrive progressively to this simulation, familiarizing with the code and adding step by step all the necessary blocks to the input namelist file, called the **`InputNamelist.py`**. In this file, the lines starting with the symbol `#` are comments, i.e. they are ignored by the code. The comments in the input namelist of this practical will have one of these two purposes: 1) help understanding a certain section of the namelist or 2) deactivate a certain block or variable definitions in the namelist, e.g. a block introducing a laser in the simulation: to activate that block and introduce the laser, you will only need to remove the symbols `#` (also from the lines defining for example the laser length).

This namelist is written in Python, but you don't need a deep knowledge of this language to understand its content. Knowing how to define variables (and optionally how to define `numpy` arrays) should be largely sufficient. This practical assumes that you are able to navigate in a directory tree, create folders and copy files from the command line (to have a quick recap, see the Sections 4,5,7,8,9,13,14,17 of `https://www.computervillage.org/articles/CommandLine.pdf`).

You will start with a simple namelist which includes only a laser propagating in vacuum, then the plasma will be added to excite a plasma wave and finally the injected electron beam will be added. Each of these three elements is introduced into the simulation with a block of text that is read by the code. Since the very first exercises of this TP, you will have the full input namelist at your disposal. At first only few blocks will be functional, like the block to define a laser in the simulation, while the other blocks will be commented (i.e. having a symbol `#` at the beginning of the line).

To introduce a new physical element to the simulation, you just need to uncomment the corresponding block and the related variables (i.e. remove the symbols `#` before them) and run again your simulation. You will arrive to the end of the TP progressively uncommenting the blocks of the laser pulse, plasma, the electron relativistic beam.

In each of these three steps you will also analyse the results of the simulations, using the postprocessing library of Smilei called **`happi`**. Alternatively, you can use **`happi`** to export the data to arrays and text files that can be processed with other languages. A familiarity with the plotting of 1D and 2D data is recommended. If you want to work with the practical library **`happi`** (recommended choice), the exercices will show you the commands to copy, paste and adapt to create your plots. For more information on the postprocessing library, or more explanations on the plot commands, you will just need to read Appendix A.

A quick word on Smilei

As previously stated, the numerical tool you will use for this practical is the PIC code Smilei. A prior knowledge of Smilei is not mandatory for the purposes of the practical exercises. Yet, checking Smilei's website for information on how to write a namelist can be useful. Furthermore, the interested reader can find additional tutorials accessible from the code's website, and that focus on other physical processes and/or on how to use Smilei.

It is an open-source and collaborative code freely distributed under a CeCILL-B license (equivalent to the GPL license for free-softwares). The code, its documentation and post-processing tools are freely available on Smilei's website hosted on GitHub: `https://smileipic.github.io/Smilei/index.html`.

All simulations presented in this practical will be run in Cylindrical-3V geometry in order to run in a short time over a single CPU. Cylindrical-3V means that space is discretized on a

2D cylindrical grid, but particles move in a three-dimensional velocity space (that is the particle velocity is a three-dimensional vector). Note however that the version of Smilei you have is the full research code (not a downgraded version!), hence, it can address simpler problems in one or two dimensions with a cartesian geometry or more realistic problems in three dimensions.

1.1 Physical units normalization

The standard physical units of the scientific community are given by the SI, but simulation codes normally work with normalized units. All quantities given to the code in this practical, as well as all quantities provided by the code as outputs will be in normalized units.

You don't have to worry about conversions in the input namelist: using appropriate variable definitions the values in SI of physical quantities will be transparent reading the namelist. However, some exercises will ask you to perform the appropriate conversions from the normalized units in the output to SI units. It is possible to perform the conversions automatically through the postprocessing library **`happi`**, but this is not treated in this practical.

Units of time	ω_r^{-1}
Units of velocity	c
Units of charge	e
Units of mass	m_e
Units of momentum	$m_e\, c$
Units of energy, temperature	$m_e c^2$
Units of length	c/ω_r
Units of number density	$n_r = \epsilon_0\, m_e\, \omega_r^2/e^2$
Units of current density	$e\, c\, n_r$
Units of pressure	$m_e\, c^2\, n_r$
Units of electric field	$m_e\, c\, \omega_r/e$
Units of magnetic field	$m_e\, \omega_r/e$
Units of Poynting flux	$m_e\, c^3\, n_r/2$

Table 1: List of the most common normalizations used in Smilei.

As shown in Table 1, all charges and mass will be normalized to the elementary charge e and electron mass m_e, respectively. Furthermore, all velocities will be normalized to the speed of light in vacuum c that naturally appears from Maxwell's equations. In this set of units, the normalized speed of light in vacuum is 1. Now, the unit of time - here defined as ω_r^{-1}, with ω_r the reference angular frequency - is not defined a priori, and is chosen by the user. Once this unit of time (or equivalently a unit of length) is chosen, all other units are uniquely defined and follow as detailed in Table 1. Note however that number densities associated to the plasma species are not in units of $(\omega_r/c)^3$ but in units of $n_r = \epsilon_0\, m_e\, \omega_r^2/e^2$. In the following exercises where you will be asked to make conversions from code units, assume a laser wavelength $\lambda_0 = 0.8$ μm (that of a Ti:Sa laser system). This choice implies, following Table 1, a unit of length $c/\omega_r = \lambda_0/2\pi$ and a unit of time $\omega_r^{-1} = \lambda_0/2\pi c$).

Although it is possible to define your simulation in SI units, simply doing the conversion with

Python in the input namelist, the code (and many other codes in this world) will work with normalized units. Therefore it is important to learn how to work also with this kind of units, also called code units, for your future simulation activities.

- Exercise 1: Assuming $\lambda_0 = 0.8$ μm (a typical Ti:Sa laser system), what is the value of the critical density? What is the value of the normalizing electric field E_0? Note that we will use this choice for λ_0 in all the following exercises. *Hint:* Use table 1 for the calculations.

1.2 Prepare for the practical exercises

As in any experiment, you need to prepare your set-up. For a numerical experiment, your interface with the computing environment is the command line terminal. Follow the instructions in the file **`ClusterEnvironment.pdf`** to respectively: login to the machine for your simulations, compile the code, create your simulation folders, submit and postprocess your simulations.

It is recommendable to create a folder for each simulation you will perform in order to avoid losing data. In each of these directories, you'll need to copy the input namelist **`InputNamelist.py`** and the Smilei executable files (the ones called **`smilei`** and **`smilei_test`**). To check if your input file does not contain errors (e.g. when you will change some parameters), you can use the command **`./smilei_test name_inputfile`**. If everything goes well (the word **`END`** should appear), you can launch your simulation with the submission script. Normally 10 MPI processes and 2 threads OpenMP should be sufficient to run the longest of your simulations in this practical within 3 minutes (these parameters are already set in the submission script). Remember that you can submit multiple simulation jobs in different folders to let them run in parallel.

Always copy the file **`InputNamelist.py`** to the folder where you will run your simulation and uncomment the necessary lines and blocks as described in the next Sections. After these steps, everything you need for your simulations is ready. For the postprocessing, remember to compile the **`happi`** library as explained in Appendix A.

To resume, in each simulation folder you will need the files **`smilei`** and **`smilei_test`**, **`InputNamelist.py`** and the **`submission_script.sh`**.

1.3 Exploring the input namelist

The code needs an input namelist file that describes what you want to simulate. For this practical, this input file will be **`InputNamelist.py`**. Open this file with a text editor, e.g. **`nano`**, **`gedit`** or **`vim`** (if in **`vim`** the file syntax is not colored, use **`:syntax enable`**) or with another editor of your choice and give a look, for the moment ignoring the commented blocks and variables.

The file starts with the definition of the number of physical constants and units, mesh points, the integration timestep, etc. Some of these parameters are inserted in the first block of the simulation, called **`Main`**. Others may be useful for conversions between units or to define other variables in the file.

In the **Main** block you will find also the **geometry** of the simulation, which is **AMcylindrical**. The grid is defined on a cylindrical space and with cylindrical coordinates x, r but the particles will move in the 3D space x, y, z (See Fig. 1). Maxwell's Equations will be solved in cylindrical coordinates with cylindrical symmetry in the simulations of the practical. Since the set-up is cylindrically symmetric, this will allow to run quick simulations with a 3D accuracy.

- Exercise 2: What are the longitudinal size **Lx** and radial size **Lr** of the simulation window? How many mesh points are used? See Figures 1, 2 for reference, and find these lengths in the **InputNamelist.py**.

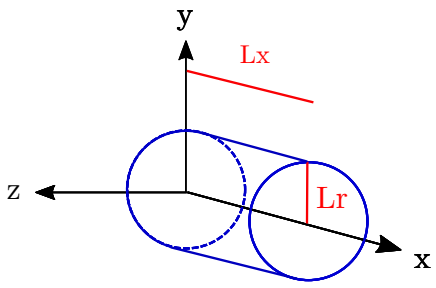

Figure 1: Reference axes of the simulation. The simulation window corresponds to the cylinder with radius **Lr** and length **Lx**.

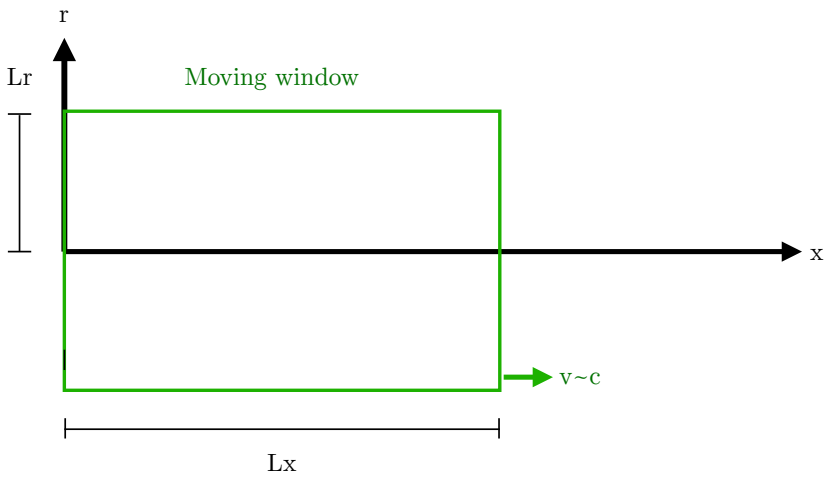

Figure 2: Simulation Setup in this Section (not in scale).

The MovingWindow block

Note a block called **MovingWindow** in the input namelist. In the considered physical case, a laser pulse with duration of the order of tens femtoseconds propagating for ≈ 400 μm along the positive x direction is simulated. We are interested only in phenomena near the laser pulse (within tens of microns) like plasma wave excitation, so it would be very inefficient to simulate all the physical space (and plasma particles when present) in a box with length of hundreds of microns. For this reason, the code has the option to use a *Moving Window*, i.e. like having a camera moving at the speed we want along the positive x direction. In this physical case, it is convenient to have a window/camera moving with the laser, so we will set the average speed of the moving window to c to follow the laser. For this reason, you will see the laser pulse almost immobile in your simulation window. The size of the our moving simulation window will be of tens of microns, so if it moves with the laser we will be sure to see all the relevant phenomena of plasma wave excitation near the pulse.

The Diag blocks

At the end of the namelist there are blocks starting with the word **Diag**. As you can imagine, they are the diagnostic of the code, our interface with the simulation results. The first **Diag** is a

`DiagProbe` defined on a line (so a 1D diagnostic), on the propagation axis of the laser (the x axis). This diagnostic will show in output the value of some physical fields along that axis. We call this probe `Probe0` (the 0 because it is the first `Probe` in the namelist). The second diagnostic block is a `DiagProbe` defined on the plane xy (so a 2D diagnostic). This is the second probe of the namelist, so it is called `Probe1`. Inside these `DiagProbe` blocks, the physical quantities in output we are interested in are specified. For this practical, you don't have to write new postprocessing scripts, since Smilei includes a postprocessing library, called **happi**, to analyze the code results. For the purposes of the practical, the **happi** commands explained in Appendix A and the postprocessing scripts you will be given are sufficient to start. However, if you prefer to work with **numpy** arrays in Python, at the end of Appendix A you will find also how to export the output to that format. If you prefer to work with other languages for postprocessing, you can export the data to a **numpy** array and then write them to a file readable with your language of choice. The last exercises of this practical will ask to write postprocessing scripts, using the commands you will find in the already provided postprocessing scripts.

2 Simulation Exercises

After the preliminary steps described in the Introduction (including the first two exercises), you should be ready to proceed with tha practical exercises. We will arrive to the simulations of plasma acceleration in three steps:

- simulation of the laser pulse propagating in vacuum;
- simulation of the plasma waves in the wake of the laser pulse;
- simulation of the acceleration of electrons in the laser wake wave

Initially the input namelist only has the essential blocks (e.g. the Main block), the other blocks are commented. The mentioned three steps will be completed progressively decommenting the laser pulse block, the plasma electron block and the blocks used for the electron bunch.

To complete the practical, read the instructions of the following sections, complete the related simulation exercises and ask the instructor in case of doubts.

Without further ado, let's get started!

2.1 Laser pulse in vacuum

Everything is ready to run your first simulation: just uncomment the lines with the laser pulse parameters and the `LaserEnvelopeGaussian` block. This block defines a Gaussian laser pulse in the simulation. The considered pulse will also have a Gaussian temporal envelope profile.

Normally you would simulate a laser pulse with all the high frequency oscillations at frequency $\omega_0 = 2\pi c/\lambda_0$. This is surely the most accurate approach, but it would need a very fine resolution and integration timestep and consequently a very long simulation time (see Fig. 3). Note that in the most simple laser wakefield acceleration set-ups the laser pulse temporal envelope (often modeled with a Gaussian profile) will contain many optical oscillations.

Thus, using a laser envelope model it will be possible to describe the laser and its interaction with the plasma only in terms of its envelope and have quick simulations (due to the coarser

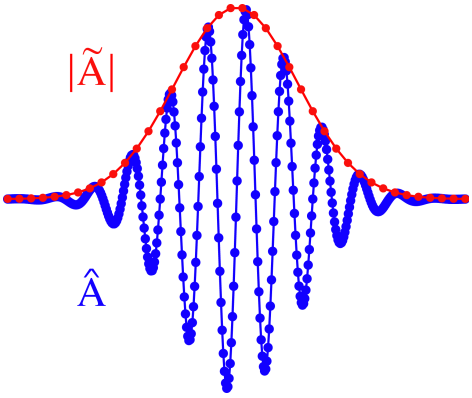

Figure 3: Blue line: vector potential $\hat{A}$ of a laser pulse with Gaussian envelope. Red line: its absolute value $|\tilde{A}|$. Both lines are sampled by a suitable number of points. Note that much less points are necessary to sample the envelope.

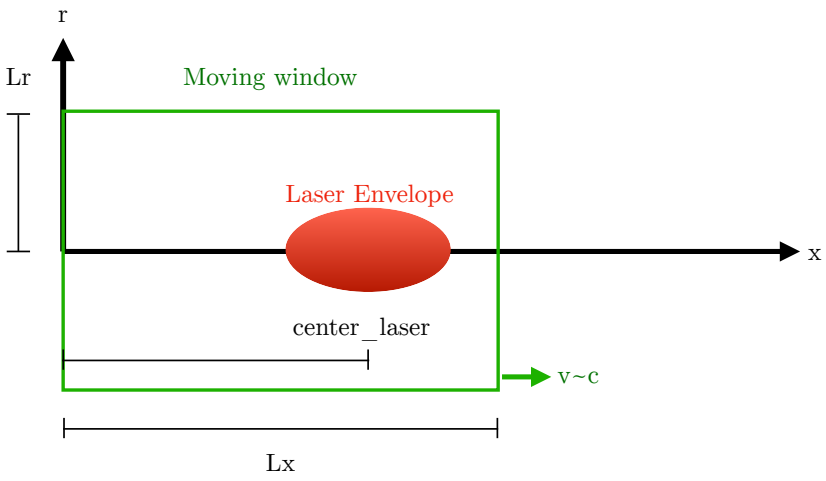

Figure 4: Simulation Setup in this Section (not in scale).

sampling). In this kind of model, the laser evolution is described by an envelope equation (like in a paraxial wave equation for example). The laser radiation pressure on the plasma particles is described by the ponderomotive force (which depends only on the laser pulse envelope). With this approach, the accuracy of the results will be enough to explore the basic physics of laser plasma acceleration and at the same time you will be able to perform quicker simulations.

Note: since an envelope model is used for the laser, the electromagnetic fields in output like **`Ex`**,**`Ey`** (and the densities like **`Rho`**) will only show the average of these quantities over the optical oscillations of the laser and not including the laser field. Thus, these quantities will not show the oscillations at ω_0 and $2\omega_0$ normally seen in a simulation of LWFA without an envelope model. The electromagnetic fields in output will be those generated in the plasma, while the absolute value of the envelope of the transverse electric field of the laser is contained in the quantity **`Env_E_abs`**.

The simulation now includes a moving window and a laser pulse, modeled with its envelope, as in Fig. 4.

- Exercise 3: What are the waist size, FWHM duration in field and intensity of the laser pulse specified in the namelist? Where is the laser pulse center placed in the simulation window at the start of the simulation? Where is the focal plane of the laser?

- Exercise 4: The normalized laser peak field is given by $a_0 = \frac{eE}{m_e \omega_0 c}$, where E is the peak laser electric field and $\omega_0 = 2\pi c/\lambda_0$ is the laser central frequency ($\lambda_0 = 0.8\ \mu m$.) What is the peak intensity $I = \frac{c\epsilon_0}{2}|E|^2$ of the laser pulse? Suggestion: the input namelist contains the physical quantities that you may need for the conversions.

After you have uncommented the necessary lines (try again `./smilei_test Inputnamelist.py`), launch the simulation. When the simulation is completed, open **IPython** with the command `ipython`. Then, you can check the initial position of the laser through the commands:

```
import happi; S = happi.Open("path/to/simulation")
S.Probe.Probe0("Env_E_abs",timesteps=0).plot( figure=1 )
S.Probe.Probe1("Env_E_abs",timesteps=0).plot( figure=2 )
```

`Probe0` is a 1D diagnostic defined on the laser propagation axis, while `Probe1` is a 2D diagnostic defined on the plane xy. Note that in the commands we have specified `timesteps=0` to see the laser pulse at the start of the simulation. Check that the initial laser position that you are seeing is the same specified in the input namelist. Remember that the laser pulse is modeled through its envelope, so you won't see its high frequency oscillations with wavelength λ_0.

Let's study the laser diffraction in vacuum. To see the evolution of the laser, use:

```
S.Probe.Probe1("Env_E_abs").animate( figure=3 )
```

Note that the use of the `Moving Window` will make the laser seem immobile in the simulation, but it is actually moving at speed c and the `Moving Window` is following it with the same speed.

If you don't specify a `vmax` value (the colorbar maximum) in the previous animation command, `happi` will change it at each iteration. To better see the laser diffraction, try to specify `vmax=0.05` in the previous command, or use `.slide()` instead of `.animate()`. For example:

```
S.Probe.Probe1("Env_E_abs").slide( figure=3,vmax=0.05 )
```

- Exercise 5: Let's check that the Gaussian laser pulse diffracts following the theory for a Gaussian beam: $w(x) = w_0\sqrt{1 + x^2/x_R^2}$, where w_0 is the initial laser waist size, $w(x)$ the laser waist size at propagation distance x, x_R is the Rayleigh length $x_R = \pi w_0^2/\lambda_0$.

What is the theoretical Rayleigh length x_R ?

- Exercise 6: Use the script `Laser_waist_theory_vs_Smilei.py` to compare the analytical diffraction law of the previous exercise and the code results. Copy the script in the simulation folder or call the script from that folder. The script loads the results, then loops over the iterations available in output and computes the laser pulse waist $w(x)$ as

$$w(x) = 2 \times \frac{\int\int |\tilde{A}|^2 (y - \bar{y})^2 dxdy}{\int\int |\tilde{A}|^2 dxdy}. \tag{1}$$

Since the laser field goes to zero at the borders of the simulation window, instead of integrating over the infinite extension of space we can limit our integrals to the simulation window space. After this computation, the simulated waist is compared to the corresponding analytical value. Run the script (from `IPython` use `%run Laser_waist_theory_vs_Smilei.py`) to plot the comparison.

2.2 Plasma waves

2.2.1 Linear plasma waves

In the previous section it was verified that the laser pulse behaves as expected in vacuum. Now, let's add a preionized hydrogen plasma to excite plasma waves. The laser will be intense enough to assume the gas was already ionized much before the arrival of the laser pulse peak (see the laser intensity computed in **Exercise 3**).

Uncomment the first **`Species`** block, the related variable definitions and take some time to read them carefully. This block defines a particle **`Species`** in the simulation, whose name is **`plasmaelectrons`**. Note the normalized mass and normalized charge of these particles defined in this block (`1.0`. and `-1.0`. respectively). Since the normalizing mass and charge are the electron mass and the unit charge (see Table 1) we know that these particles are electrons.

Since the simulation is in cylindrical coordinates, the plasma electron density profile in general will be defined at coordinates (x, r), where r is the distance from the laser propagation axis. As you can see, for each point in space (x, r) the plasma density profile is defined by the function **`plasma_density`**(x, r). After a short linear ramp, this plasma density profile is uniform in a region confined to a distance **`Radius_plasma`**=22 μ**`m`** from the laser propagation axis. This of course is not a realistic size for a real-world plasma for plasma acceleration. However we can simulate a small-scale plasma like this without significantly changing the results, since the phenomena of interest occur near the laser pulse. Including particles far from the laser propagation axis would increase the computation time without changing the results. You can check it at the end of the practical increasing the plasma transverse size, bear in mind that this will increase the simulation time.

The simulation now includes a moving window, a laser pulse (modeled with its envelope) and plasma electrons, as in Fig. 5.

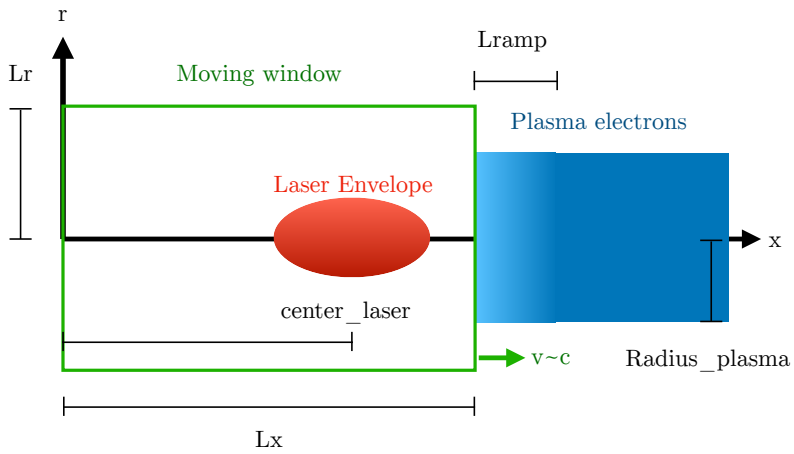

Figure 5: Simulation Setup in this Section (not in scale) at $t = 0$.

As we did with the simulation of the laser pulse diffraction in vacuum, the first step will be to verify that the plasma behaves as predicted by the analytical theory. If we reduce the laser pulse a_0 to 0.01, the laser pulse will satisfy the conditions for the applicability of the 1D linear

theory of plasma wave excitation. The analytical 1D linear theory predicts the formation of a sinusoidal wave at plasma frequency $\omega_p = \sqrt{e^2 n_0 / m_e \epsilon_0}$ behind the laser, where n_0 is the plasma density.

- Exercise 7: As you can see, the plasma density has a value $n_0 = 0.0008$ (in normalized units, i.e. normalized by the critical density in this case) in the uniform region. What is its value in physical units (electrons/cm^{-3})? *Hint:* see Table 1 in Section 1.1. If you wanted to specify n_0 from SI units to code units using the conversion with the variable **`ncrit`** in the namelist, what command should you use? You don't have to change the namelist.

- Exercise 8: Launch the simulation with $a_0 = 0.01$. Study the evolution of the electric field **`Ex`** with **`Probe0`** and **`Probe1`** (same plot commands of the previous section, but applied to **`Ex`**). What is the theoretical plasma wavelength $\lambda_p = 2\pi c/\omega_p$? What is the plasma wavelength inferred from the simulation results?

- Exercise 9: The longitudinal electric field on the axis of this linear plasma wave, according to the 1D linear theory, is given by (in physical units):

$$E_x(x) = \frac{m_e c^2}{e} \frac{k_p^2}{4} \int_x^{+\infty} |\tilde{A}|^2 \cos\left[\omega_p (\mathrm{x} - \mathrm{x}')\right] \mathrm{dx}'. \quad (2)$$

Use the script **`Ex_linear_theory_vs_Smilei.py`** to compare the analytical result given by Eq. 2 and the simulated results (**`%run Ex_linear_theory_vs_Smilei.py`** on **`IPython`**). Again, you will need to copy the script in the simulation folder or to call it from there. The script reads the absolute value of the envelope of the laser $|\tilde{A}|$ from the simulation at a given time and then implements Eq. 2 to compute the analytical result for E_x. Then it reads the longitudinal electric field **`Ex`** of the simulation to compare the theory with the simulation results. Does the simulation agree with theory?

2.2.2 Nonlinear plasma waves

With the laser and plasma parameters in the namelist, for $a_0 \ll 1$ the excited plasma waves will be in the linear regime. As we saw in the previous exercise, the plasma wave in the wake of the laser is sinusoidal in this regime. Increasing a_0, the laser will be more intense and when $a_0 \gtrsim 1$ the plasma electrons will start to be pushed to relativistic velocities. Their relativistically increased inertia will elongate the plasma period and plasma wavelength, and electron accumulation will occur at the end of each wave period. Increasing a_0, the longitudinal electric field waveform will change from a sinusoid to a sawtooth wave. In this regime of interest for plasma acceleration PIC simulations become necessary, because for this regime there are no general analytical solutions to the coupled Vlasov-Maxwell system of equations, and fluid theory cannot be applied.

Create three folders, **`sim1`**, **`sim2`**, **`sim3`**, where you will launch the simulation with $a_0 =$ 0.5, 1.4, 2.0 respectively. Take a look to the longitudinal electric field on axis (**`Probe0`**) and to the 2D plasma density (**`Probe1`**):

```
S.Probe.Probe0("Ex").animate( figure=1 )
S.Probe.Probe1("-Rho").animate( figure=2 )
```

In the linear case probably you will not be able to see easily the oscillations in the density, but they can be easily seen in the electric field E_x. In the nonlinear cases (higher a_0) you will need to lower the **vmax** in the plot/animate command to see the formation of the wake. This because at the end of the plasma wave period there is an accumulation of electrons, which hides the other values of the charge density if **vmax** is not properly chosen. As you can see, increasing the a_0, the nonlinearities of wake excitation will be progressively more evident. For each simulation, take a look to the excitation of the wave with the command **animate** of **happi**.

- Exercise 10: It is interesting to compare the longitudinal electric field of these three simulations to see how the wave changes increasing a_0. With **happi** (or Python if you prefer) you can easily do it. Here are the commands for **happi**:

```
import happi
S1 = happi.Open("path/to/sim1"); Ex1 = S1.Probe.Probe0("Ex",timesteps=1000,label="a0 =
0.5")
S2 = happi.Open("path/to/sim2"); Ex2 = S2.Probe.Probe0("Ex",timesteps=1000,label="a0 =
1.4")
S3 = happi.Open("path/to/sim3"); Ex3 = S3.Probe.Probe0("Ex",timesteps=1000,label="a0 =
2.0")
happi.multiPlot(Ex1,Ex2,Ex3,figure=3)
```

The last command, **multiPlot**, is used to superpose multiple lines in the same plot window. This command will be used also in some of the exercises of the following section.

Behind the curtain: why ions are not present? (Optional)

A plasma for laser wakefield acceleration is normally made of ions and electrons at least, so why ions are not present in this namelist? The answer can be found in the properties of Maxwell's Equations and implies some derivations. For the moment it is sufficient to say that, since we set to zero the plasma electromagnetic field at the beginning of these simulations, and that we solve carefully Maxwell's Equations and the particles equations of motion, defining the plasma electrons will make the code behave as if there was also a neutralizing layer of immobile ions. Since ions do not move in the timescales of interest for the phenomena we are simulating (their mass is at least 2000 times larger than the electron mass), this is a reasonable approximation that in addition removes the need to simulate also the ions, with a significant computational gain. The complete answer for the interested reader can be found in the dedicated section of this tutorial `https://smileipic.github.io/tutorials/advanced_wakefield_electron_beam.html`.

2.3 Laser plasma acceleration with external injection

Finally everything is ready to simulate a plasma accelerator. As a surfer rides the waves in the ocean, under some conditions a bunch of electrons can be accelerated by plasma waves. Remember that an immobile surfer will not be accelerated by a wave. To efficiently interact with the wave, the surfer must first pad to acquire some velocity. If the surfer moves near to the speed of the wave, he will see an accelerating phase of the wave for all the interaction. Following the same analogy, the electrons to be accelerated must be injected in the accelerated phase of the plasma wave, with a speed near the wave velocity ($\approx c$). Many clever injection schemes have been investigated since the years 2000s, where the electrons of the plasma itself are someway injected in the laser-driven wave. However, although it is more experimentally difficult to realize, in this practical external injection will be studied, i.e. the injection of a relativistic electron

beam from outside the plasma. This will allow to understand the basic concepts of electron injection in a plasma wave.

Set again the a_0 of the laser to the value 1.8 . Uncomment the second **Species** block, the related variable definitions and take some time to read them carefully. As you can see, this block defines a **Species** called **electronbunch**, which we will inject in the plasma wave for acceleration. As for the **Species** called **plasmaelectrons** of the previous Sections, these particles have normalized **charge** and **mass** equal to **-1.0** and **1.0** respectively, thus they are electrons. In this case, the number density will not be defined through a density profile function, but the charge, the coordinates and the momenta of each of the bunch's macro-particles will be given to the code through arrays. The variable **npart** defines the number of macro-particles (in this case 20000) of the bunch, and this will be also the length of the bunch arrays: to initialize **npart** particles one by one, you need an array of length **npart** for their x coordinates, an array of length **npart** for their y coordinates and for the z coordinates. These coordinates will be stored in the variable called **array_position**, which keeps also an array of length **npart** to store the charge of each macro-particle (you don't need to understand right now how it is defined). You will need also need to initialize similarly the momenta of the particle, which are stored similarly in the variable **array_momentum**. In this case, these coordinates and momenta are generated from a Gaussian pseudorandom distribution, multiplying the numbers by their standard deviations to have a bunch with a rms size, rms momentum spread and rms energy spread that we want. You will note that the initialized bunch is already relativistic and has a Gaussian distribution in all the coordinate-momenta subplanes in the phase space.

For your future simulation work, this initialization method can be used also to use a particle distribution obtained from another code (a magnetic transport code for conventional accelerators for example). Instead of generating randomly the particles coordinates and momenta, you only need to read them with Python. The arrays you will obtain can thus be directly used by Smilei. Note that you can do this also for a plasma density distribution or laser intensity profile that you measure in an experiment. Instead of defining them through a Python function, you can read the experimental data and feed them to Smilei to simulate your measured plasma and/or laser profile.

The simulation now includes a moving window, a laser pulse (modeled with its envelope), plasma electrons and an electron bunch, as in Fig. 6.

- Exercise 11: Reading the namelist, provide a description of the electron bunch at $t = 0$: total charge, energy, rms sizes along x, y, z, rms energy spread, emittance along the transverse planes. Where is the electron bunch placed in the window initially?

The DiagTrackParticles block

To have a close look to the electron bunch during its propagation, you will need to uncomment also the **DiagTrackParticles** block. As the name suggests, this diagnostic block allows to track particles, specified by their species name and some filter. Using a filter (e.g. selecting only the particles with energy higher than 50 MeV) is particularly useful when you have many particles in a **Species**, like in the plasma of the namelist In that case, not using a filter would make this diagnostic computationally heavy and would store the coordinates of too many particles. In the case of the bunch, there is no need to specify a filter, since the number of bunch macro-particles is sufficiently small to be manageable. As you can see from the namelist, in this diagnostic we will store the coordinates and momenta of their particles, as well

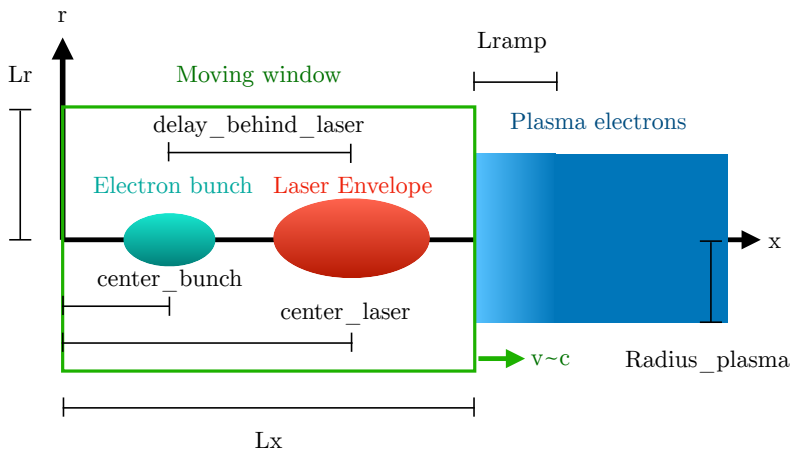

Figure 6: Simulation Setup in this Section (not in scale).

as their weight (from which their charge can be computed).

- Exercise 12: Launch the simulation, remembering to uncomment also the **`DiagTrackParticles`** block. This time the simulation will run a little longer, because the self-consistent electromagnetic field of the relativistic electron bunch must be computed first as initial condition and it requires some time. Plot the charge density at the end of the simulation and play with the parameter **`vmax`** to be able to see the electron bunch in the plasma wave, and include this image in your answers.

- Exercise 13: Use the command **`happi.multiPlot`** (see Appendix A) to plot in the same window the longitudinal electric field **`Ex`** and the charge density **`Rho`** from **`Probe0`** at the end of the simulation in the same window. You may need to rescale the quantities - see Appendix A. Playing with multiplying factors in the plot you should be able to clearly see where the electron bunch is placed in the plasma wave. Include this image in your answers.

- Exercise 14: Run the script **`Compute_bunch_parameters.py`** in the simulation folder to read the electron bunch parameters at the end of the simulation. For this purpose, from **`IPython`** you can use **`%run Compute_bunch_parameters.py timestep`**, where you specify the **`timestep`** you are interested in. What is the energy gain ΔE you measure from the start **`timestep = 0`** to the end of the simulation **`timestep = 5000`**? What is the simulated propagation distance L? From these data, compute the average accelerating gradient E_{acc}. What is the absolute and relative rms energy spread at the beginning and at the end of the simulation? Report all the bunch parameters at the start and at the end of the simulation.

- Exercise 15: Use the script **`Follow_electron_bunch_evolution.py`** to see how the beam has evolved during the simulation. The script reads the **`DiagTrackParticles`** output and then computes some bunch quantities (rms size, emittance, energy, energy spread) at each available output iteration. From the evolution of the bunch energy, can you estimate the average accelerating gradient? Compare this value to the one computed in the previous exercise

- Exercise 16: Create three folders, **`sim1`**, **`sim2`**, **`sim3`**, **`sim4`** where you will launch the simulation varying the bunch charge: 40, 60, 80, 100 pC. Adapt the commands you have used in **Exercise 9** to plot the longitudinal electric field **`Ex`** on axis of the three simulations with **`happi.multiPlot`** (see Appendix A)

and include this plot in the answers. What do you observe? Use the script `Compute_bunch_parameters.py` used for **Exercise 14** to find the energy of the electron bunch at timestep 5000. Can you explain the different final energies with the deformation of the `Ex` waveform? Include also a plot of the energy gain of the bunch obtained for charges 20, 40, 60, 80, 100 pC. In other words, include also a plot with the bunch charge on the horizontal axis and the energy gain on the vertical axis.

- Exercise 17: Create other four folders, `sim5`, `sim6`, `sim7`, `sim8`, where you will launch the simulation varying the bunch distance from the laser playing with the variable `delay_behind_laser` (Set again the charge to 20 pC). Try the values 17.8 μm, 18.1 μm, 18.4, μm, 18.7 μm. What is the final energy with these values? Using `happi.multiPlot` (see Appendix A) to plot the longitudinal electric field of the threes simulations in the same window (include this plot in the answers), can you explain why? Include also a plot of the energy gain of the bunch varying the `delay_behind_laser`. In other words, include also a plot with the `delay_behind_laser` on the horizontal axis and the energy gain on the vertical axis.

- Exercise 18: Using the same approach of the script `Compute_bunch_parameters.py` write a Python script to read the output of the `DiagTrackParticles` and draw the energy spectrum of the electron bunch, obtained using the original bunch parameters of the input namelist. Then, provide the script and the figure of the energy spectrum at timestep 5000. Hint: use the function `numpy.histogram` to compute a histogram of the particles energies and the bins/edges of the horizontal axis.

- Exercise 19: Using the `TrackParticles` diagnostic and `Probe` diagnostic, write a script that takes in input an iteration number, a variable called e.g. `timestep`. The script should plot in the same panel the longitudinal electric field `Ex` along the propagation axis x and a scatter plot of the bunch electrons' `x` and `px` to show the particles' positions in the accelerating phase of `Ex` in that iteration. For an example with normalized units, see Fig. 7. Include the script and a screenshot of this image for the iterations `3000` and `5000`. Differently from Fig. 7, use SI units in the plots, using the appropriate conversions.

Hint 1: Adapt the commands in the scripts you have already used to write this new script, in particular those ones reading the bunch electrons' coordinates and momenta and the `Ex` field on the propagation axis.

Hint 2: To export the `Ex` data from `happi`, use the function `getData()`, e.g.:

```
Ex = np.asarray(S.Probe.Probe0("Ex",timesteps= timestep).getData())[0,:]
```

.

Hint 3: The bunch electrons' `x` coordinates are absolute, while the Probe coordinates along the axis are relative to the moving window. To shift them including the movement of the `MovingWindow`, you can define an auxiliary variable to add to the moving window's relative coordinates through the `getXmoved()` command, e.g:

```
moving_window_x_shift = S.Probe.Probe0("Ex").getXmoved(timestep)
```

- Exercise 20: Write a new script adapting the one written for the previous exercise. As before, this new script takes in input an iteration number, a variable called e.g. `timestep`. The script should plot in the same panel the electric field `Ey` on the plane xy and a scatter plot of the bunch electrons' `x` and `y` to show the electrons' positions in the focusing phase of `Ey` in that iteration. For an example with normalized units, see Fig. 8. Include the script and a screenshot of this image for the iterations `1000` and `5000`. Differently from Fig. 8, use SI units in the plots, using the appropriate conversions.

Hint 1: You can obtain the `Ey` on the plane xy exporting `Probe1` to a numpy array, e.g:

```
Ey = np.asarray(S.Probe.Probe1("Ey",timesteps = timestep).getData())[0,:,:]
```

Hint 2: for a 2D plot, the Python function `imshow()` of `matplotlib.pyplot` is recommended, which takes in input the field to plot, the extent of the window coordinates and in case the minimum and maximum field number, e.g.:

```
imshow(Ey.T,extent = [Xmin,Xmax,Ymin,Ymax], vmin=..., vmax=...)
```

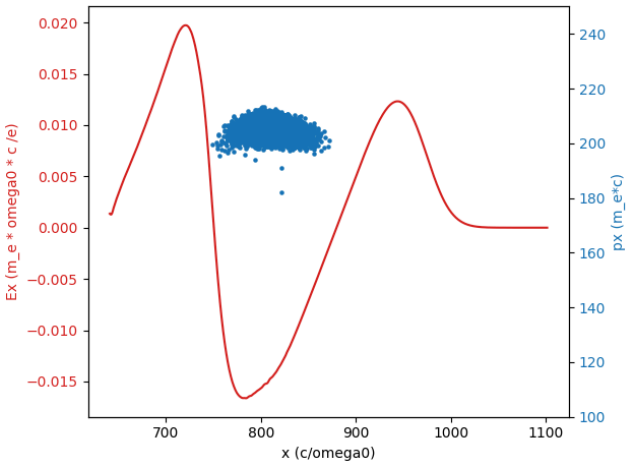

Figure 7: Longitudinal electric field **Ex** on axis and scatter plot of the bunch electrons' **x** and **px**, at iteration **1000**.

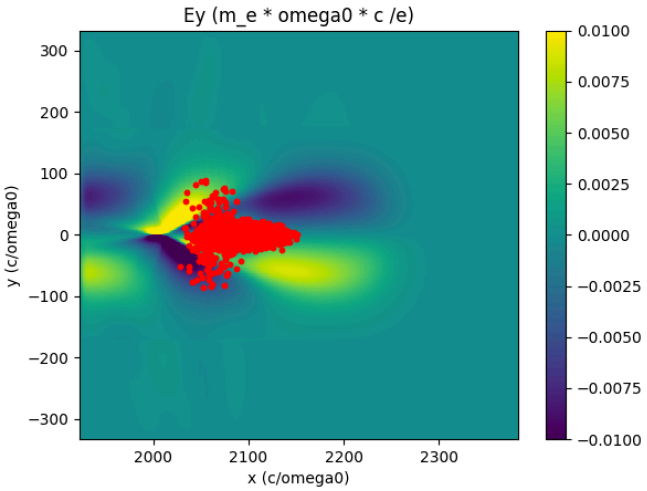

Figure 8: Electric field **Ey** on the plane xy and scatter plot of the bunch electrons' **x** and **y** coordinates, at iteration **3000**.

A Crash course on the happi postprocessing library

A fundamental part of working with simulation codes is of course the postprocessing of the results. Smilei includes an entire library for postprocessing based on Python. However, to plot your first results and make quantitative evaluations you don't need to be an expert of this language.

For your convenience and quick reference, here we include only the commands you will need for this practical. Do not hesitate to copy and paste the following commands in **IPython** and adapt them to the problem you are solving.

Remember that the results are in normalized units. The library **happi** also allows to convert to SI units, but this will not be taught in this practical (details in the documentation online `https://smileipic.github.io/Smilei/post-processing.html`).

Compilation of happi

It is sufficient to use the command **make happi** in the code folder (after you have loaded the Python modules, see the file **RucheEnvironment.pdf**). Then, to analyze the results of your simulation, open the **IPython** interface (just use the command **ipython** in the command line terminal).

Open a simulation

To import the library **happi** in **IPython** and open a simulation in the folder, called for example **"path/to/simulation"** use:

```
import happi; S = happi.Open("path/to/simulation")
```

This will create an object called `S`, our simulation, which contains all the necessary data, taken from the input namelist and from the output files. You can easily access parameters from the input namelist, for example:

```
S.namelist.dx
S.namelist.Main.geometry
```

In general, if you tap `S.` or adding the name of the blocks and then using the tab key, you will see the available blocks and variables.

Plot diagnostics

To open a specific diagnostic, like the `Probe1` defined in the namelist, and plot the longitudinal electric field `Ex` contained in that diagnostic, use:

```
S.Probe.Probe1("Ex").plot()
```

Other physical fields defined on the grid that you can plot are for example `Ey`, `Rho` (the charge density). Remember that you can also specify operations on the fields, like `2.*Ey-Ex`, when you declare your variable.

By default, the last command will only plot the requested field obtained in the last simulation output available for that diagnostic. You may instead be interested in a specific iteration of the simulation (in code units), like iteration 1200. To plot only that timestep, just specify it inside the diagnostic block:

```
S.Probe.Probe1("Ex", timesteps=1200).plot()
```

Remember that this timestep corresponds to physical time $1200 * \mathrm{dt}$, where dt is the simulation timestep, which can be found with `dt=S.namelist.Main.timestep`.

To know which iterations are available for your diagnostic, you can use:

```
S.Probe.Probe1("Ex").getAvailableTimesteps()
```

Visualize multiple timesteps

Normally you will have a sequence of outputs, so you may want to see an animation of the outputs or to be able to slide between the saved timesteps. It is possible to do it with these commands respectively:

```
S.Probe.Probe1("Ex").animate()
S.Probe.Probe1("Ex").slide()
```

In the last case, just slide with the horizontal bar to see the evolution of the plotted quantity at different iterations.

Modify elements to the plot

Like in Python, you may be interested into specifying the figure number, or change the colormap, or specifying a maximum or minimum value plotted. You can include the same corresponding keywords inside the plot/animate/slide command. As an example where all these elements are specified:

```
S.Probe.Probe1("Ex").plot( figure=2, vmin = -0.01, vmax = 0.01 , cmap = "seismic")
```

Plot multiple lines

You may be interested in visualizing multiple curves in the same plot window. The command **happi.multiPlot** is what you need.

For example, if you want to plot two quantities from the same simulation, scaling them through multiplying factors:

```
import happi; S = happi.Open("path/to/simulation")
E = S.Probe.Probe1("0.1*Ex", timesteps=1000, label = "E")
rho = S.Probe.Probe1("-10.*Rho", timesteps=1000, label="charge density")
happi.multiPlot(E, rho, figure = 1)
```

The previous example draws two curves, but you can use **multiPlot** to plot more curves.
Note that you can plot also different timesteps from the same simulation with the same procedure.
Similarly, you can plot two quantities from two or more simulations:

```
import happi
S1 = happi.Open("path/to/simulation1"); Ex1 = S1.Probe.Probe0("Ex",timesteps=1000)
S2 = happi.Open("path/to/simulation2"); Ex2 = S2.Probe.Probe0("Ex",timesteps=1000)
happi.multiPlot(Ex1,Ex2)
```

Export the data

Those shown above are all the **happi** commands you may need for this practical. If you prefer instead to analyze your results with **numpy** arrays in Python, you can easily export your diagnostic to a **numpy** array, for example:

```
import happi; import numpy as np
S = happi.Open("path/to/simulation")
myArrayVariable = S.Probe.Probe1("Ex").getData()
myArrayVariable = S.Probe.Probe1("Ex", timesteps=1200).getData()
myArrayVariable = np.asarray(myArrayVariable)
```

In case you want to export the data to a text file **.txt** and read it with another language, you can write this array on a text file using:

```
np.savetxt("file_name.txt", myArrayVariable)
```

B Why working with normalized units?

Interfacing with the real world is surely easier with SI units, nonetheless simulation codes work with normalized units. The normalization depends on the characteristic scales of the problem, so using normalized units has two main advantages:

- If you work with numbers near to 1, you are less prone to machine precision errors. For example, imagine to describe the spatial scales of molecules with chilometers, using 8 bits (1 byte). Your byte will not be able to keep the necessary precision. For this reason it is always recommended (and often necessary) to rescale your quantities with the scales of interest (e.g. in our case the velocities by the speed of light and so on);
- Once you perform a simulation with a certain number of normalized quantities, you can use your results for an infinite number of real-life experiments, you just need to rescale properly and coherently all your results. This provided that the physical equations involved do not change at the chosen scales.