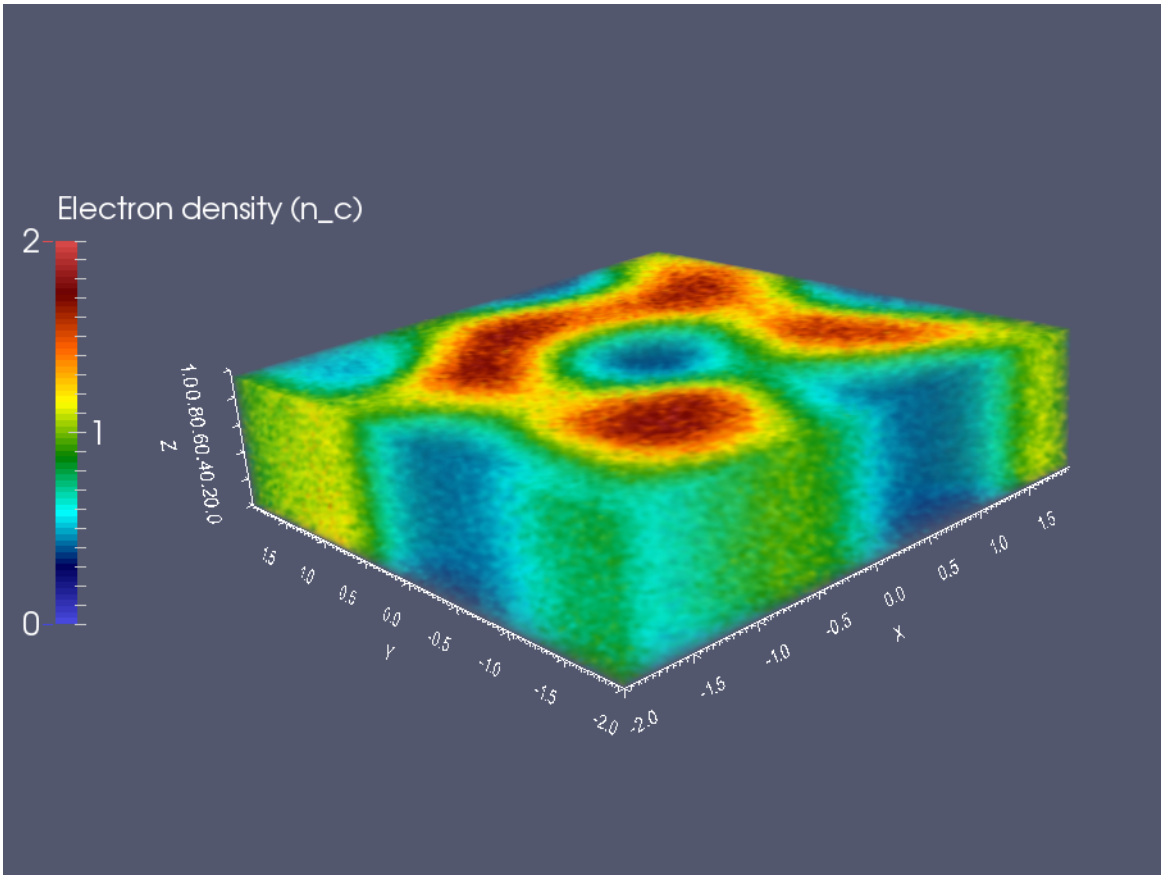
Chapter 1

Introduction

piccante, the code described in this document, is a massively parallel fully-relativistic 3D particle-in-cell code. The user is able to define an arbitrary number of particle species with arbitrary density functions and an arbitrary number of Gaussian laser pulses.
The code is proved to run on up to 2048 MPI processes.
In its present state, the code could be considered in "beta testing".
This document should provide all the information needed to compile the code, run simulations and analyse the results.



Chapter 2

How to get and compile the code

The following instructions describe the procedure to compile the code in a Linux system. However the code can be compiled also on a Windows system and should be compilable also on an Apple system.

2.1 How to get the code

2.1.1 Github

The code is distributed via an open-source repository hosted on `https://github.com/`.
Make sure that *git* (a distributed version control system) is installed on your machine (`http://git-scm.com/`). On a Debian-based system, just type:

```
$ sudo apt-get install git
```

Once *git* is installed on your system, it is possible to easily install and keep updated the code.
To install the code:

```
# git clone https://github.com/ALaDyn/piccante.git
```

A folder named `piccante` will be created.
To check for updates, `cd` into `piccante` folder and type:

```
# git pull
```

2.1.2 Ask a contributor

If you need further information, ask one of the contributors:

A. Sgattoni	andrea.sgattoni@gmail.com
L. Fedeli	l.fedeli@for.unipi.it
S. Sinigardi	stefano.sinigardi@gmail.com

2.2 Dependencies

This code depends on *GNU Scientific Library* and *Message Passing Interface*. You need these libraries to compile and run the code on your system.

2.2.1 GNU Scientific Library

In Debian-based Linux distributions, it should be possible to install the GNU Scientific Library typing:

```
$ sudo apt-get install libgsl0-dev
```

In any case, please refer to `http://www.gnu.org/software/gsl/`.

2.2.2 MPI

Our code has been tested using the *OpenMPI* implementation of the *Message Passing Interface*.
To install *MPI* on a Debian-based Linux distribution, just type:

```
$ sudo apt-get install openmpi-bin libopenmpi-dev
```

Plese refere to `http://www.open-mpi.org/` for further information.

2.3 Compilation of the code

The code is provided with a Makefile, so, in order to compile it, just type:

```
$ make
```

In this way, the code is compiled with `-O3` optimization level and no debug symbols.
If you want to run the code through a debugger, type (`make clean` erases the executable and the `.o` files):

```
$ make clean
$ make debug
```

Finally, if you want to use *Scalasca* profiler (`http://www.scalasca.org/`) type:

```
$ make clean
$ make scal
```

Chapter 3

How to prepare a simulation

In order to prepare a new simulation, among the source files only `main-1.cpp` needs to be modified.
In this chapter, the structure of this file is described in detail.
Please refer to `simple_1d_wakefield.cpp` in the `example` directory, which should be renamed `main-1.cpp` in the source directory, if you want to compile and launch this example.
Other example main files are described in the next chapter.

3.1 Headers and definitions

```
#define _USE_MATH_DEFINES

#include <mpi.h>
#include <cstdio>
#include <iostream>
#include <fstream>
#include <sstream>
#include <malloc.h>
#include <cmath>
#include <iomanip>
#include <cstring>
#include <ctime>
#if defined(_MSC_VER)
#include "gsl/gsl_rng.h"
#include "gsl/gsl_randist.h"
#else
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#endif
#include <cstdarg>

#include <vector>

using namespace std;
```

This section doesn't need to be modified. Standard libraries and GNU Scientific Library headers are included.

Set the dimensionality of the simulation

```
#define DIMENSIONALITY 1
```

The first parameter which the user needs to modify is the dimensionality of the code. It is very important to define `DIMENSIONALITY` before including `"access.h"`. Possible values are 1,2 and 3. An error at compilation time occurs if other values are selected or if `DIMENSIONALITY` is not defined.

```
#include "access.h"
#include "commons.h"
#include "grid.h"
#include "structures.h"
#include "current.h"
#include "em_field.h"
#include "particle_species.h"
#include "output_manager.h"
```

Here, the header files of the code are included. These lines should not be modified by the user.

Parallelization: set the number of MPI task

```
#define NPROC_ALONG_Y 1
#define NPROC_ALONG_Z 1
```

`NPROC_ALONG_Y` and `NPROC_ALONG_Z` set the number of MPI processes along y and along z. The total number of MPI processes when the application is launched (see chapter 4) should be a multiple of `NPROC_ALONG_Y` $\times$ `NPROC_ALONG_Z`. The number of processes along x is *automatically* selected so that [number of processes along x] $\times$ `NPROC_ALONG_Y` $\times$ `NPROC_ALONG_Z` is equal to the total number of MPI processes.

```
#define DIRECTORY_OUTPUT "TEST"
#define RANDOM_NUMBER_GENERATOR_SEED 5489
#define FREQUENCY_STDOUT_STATUS 5
```

These lines set, respectively, the output directory (*which should be created manually by the user*), the seed of the random number generator (more on that later) and the frequency of the visual status report when the code is in execution.

```
int main(int narg, char **args)
{
        GRID grid;
        EM_FIELD myfield;
        CURRENT current;
        std::vector<SPECIE*> species;
        vector<SPECIE*>::const_iterator spec_iterator;
        int istep;
        gsl_rng* rng = gsl_rng_alloc(gsl_rng_ranlxd1);
```

These lines should not be modified.
`grid` contains all the information related to the simulation grid (coordinates of grid points, number of processors ...). Moreover, it contains other important parameters of the simulation (simulation time, moving window properties ...).
`myfield` evolves the electromagnetic field.
`current` is used for current and density deposition.
`species` is a vector of `SPECIE` objects. The user is able to define an arbitrary number of species (see section 3.5).

3.2 Grid settings

```
        grid.setXrange(-50.0, 0.0);
        grid.setYrange(-1.0, 1.0);
        grid.setZrange(-1.0, +1.0);
```

```
grid.setNCells(2500, 0, 0);
grid.setNProcsAlongY(NPROC_ALONG_Y);
grid.setNProcsAlongZ(NPROC_ALONG_Z);
```

These code lines define the main properties of the simulation grid.
The first three lines define the ranges of the simulation box in X, Y and Z.
`grid.setNCells(1500, 0, 0)` defines 1500 grid points along x and 0 grid points along y and z. If `DIMENSIONALITY` is 1, the number of grid points along y and z is ignored.
Finally `grid.setNProcsAlongY(NPROC_ALONG_Y);` and `grid.setNProcsAlongZ(NPROC_ALONG_Z);` respectively set the number of processes along y according to `NPROC_ALONG_Y` and the number of processes along z according to `NPROC_ALONG_Z`.

```
//grid.enableStretchedGrid();
//grid.setXandNxLeftStretchedGrid(-15.0,1000);
//grid.setYandNyLeftStretchedGrid(-5.0, 70);
//grid.setXandNxRightStretchedGrid(15.0,1000);
//grid.setYandNyRightStretchedGrid(5.0, 70);
```

The stretched grid is commented out in this example (so in this example the simulation grid is not stretched). However, this feature is described in subsection 3.2.1.

```
grid.setBoundaries(xOpen | yOpen | zPBC);
```

`grid.setBoundaries(...)` sets the boundary conditions of the simulation.
The available options at the moment are: `xOpen xPBC yOpen yPBC zPBC` (so, open boundary conditions are not yet implemented along z).
Boundary conditions may not be compatible with other simulation parameters. For instance, `xPBC` is incompatible with the moving window. Moreover, while open boundaries are fine for EM fields, if a particle suddenly disappear from the simulation, disruptive numerical instabilities may occur (in this case, using the stretched grid feature is suggested). Of course particles can disappear safely from the left x boundary if there's a moving window with $\beta = 1.0$.

```
grid.mpi_grid_initialize(&narg, args);
```

This line initialises the grid (the user should not modify it).

```
grid.setCourantFactor(0.98);
```

The Courant Factor should be always less than 1.0 to insure numerical stability of the algorithm. The user is advised not to modify the default value of 0.98 without good reason.

```
grid.setSimulationTime(60.0);
```

Here the total simulation time in units of τ_p is set.

```
grid.with_particles = YES;
grid.with_current = YES;
```

The use of these two lines is suggested for debug purposes.
The first line enables (`YES`) or disables (`NO`) particle creation, while with the second line it is possible to disable current deposition. If `YES` for the first parameter and `NO` for the second are selected, particles are created and advanced, but current is ignored for EM fields evolution.

```
grid.setStartMovingWindow(0);
grid.setBetaMovingWindow(1.0);
grid.setFrequencyMovingWindow(FREQUENCY);
```

These lines set the properties of the moving window: the start time, the β parameter and the frequency of movement (every how many time steps the window moves).
If the first line is commented out, no moving window is set. Instead, the second and the third lines are optional. Commenting out these lines results in selecting the default values for β (1) and frequency (20).

```
grid.setMasterProc(0);
```

`grid.setMasterProc(0)` sets the master MPI task to zero (if needed the user can properly choose a different MPI ID).

```
srand(time(NULL));
grid.initRNG(rng, RANDOM_NUMBER_GENERATOR_SEED);
```

Here the seeds of the random number generators for each MPI task are set (each task ends up with a different seed). These lines should not be modified.

```
grid.finalize();
grid.visualDiag();
```

These lines (which should not be modified) conclude the initialization of the grid and display some information on the standard output.

3.2.1 Stretched grid

```
grid.enableStretchedGrid();
grid.setXandNxLeftStretchedGrid(-15.0,1000);
grid.setXandNxRightStretchedGrid(15.0,1000);

grid.setYandNyLeftStretchedGrid(-5.0, 70);
grid.setYandNyRightStretchedGrid(5.0, 70);

grid.setZandNZLeftStretchedGrid(-5.0, 70);
grid.setZandNZRightStretchedGrid(5.0, 70);
```

These lines set the properties of the stretched grid. If the first line is commented no stretched grid is set, no matter the presence of some of the other lines. `grid.setXandNxLeftStretchedGrid(X_START_LEFT, NP_LEFT)` sets respectively the coordinate of the left boundary of the uniform grid (i.e. the beginning of the stretching) and the number of grid points in the stretched region. The same idea is used for the right boundary and for the `Y` and `Z` directions.

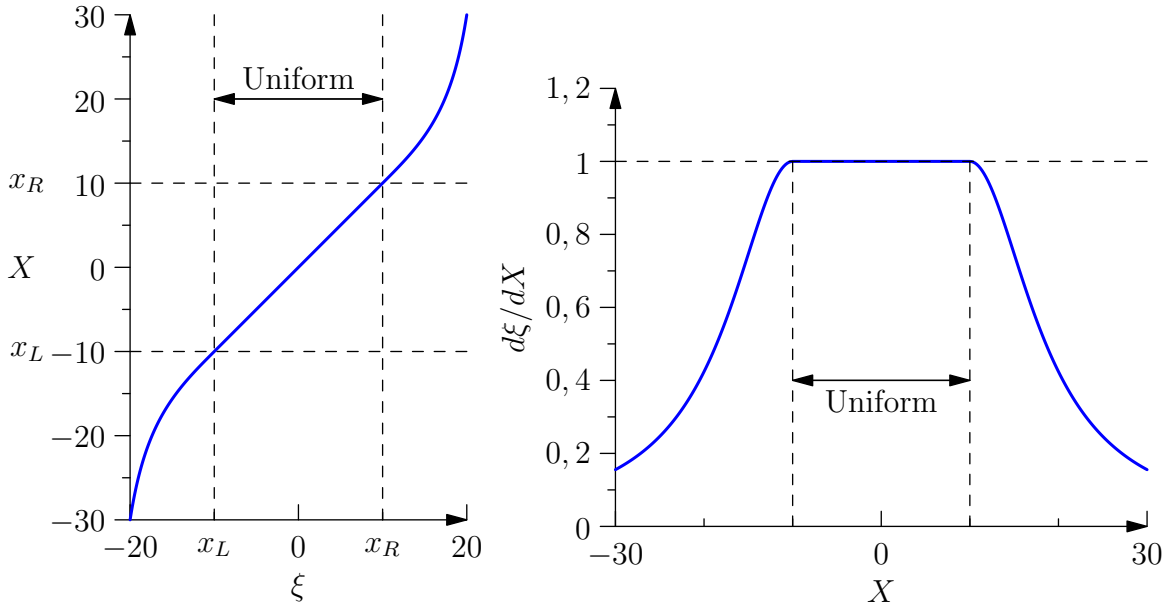


Figure 3.1: Stretching function (left) and inverse derivative (right)

The stretched grid is obtained considering a uniform grid in an auxiliary variable $\xi_i = \xi_{min} + \Delta\xi i$ and a piece-wise defined transfer function

$$x(\xi) = \begin{cases} x_{\text{right}} + \alpha_{\text{right}} \tan\left(\left|\xi - x_{\text{right}}\right| / \alpha_{\text{right}}\right), & \text{if } \xi > x_{\text{right}} \\ \xi, & \text{if } x_{\text{left}} < \xi < x_{\text{right}} \\ x_{\text{left}} - \alpha_{\text{left}} \tan\left(\left|\xi - x_{\text{left}}\right| / \alpha_{\text{left}}\right), & \text{if } \xi < x_{\text{left}} \end{cases}$$

See Fig. 3.1.

3.3 EM fields

```
myfield.allocate(&grid);
myfield.setAllValuesToZero();
```

Here the EM field is allocated and initialised to zero.

3.3.1 Laser pulse

```
laserPulse pulse1;
pulse1.type = COS2_PLANE_WAVE;
pulse1.polarization = P_POLARIZATION;
pulse1.t_FWHM = 5.0;
pulse1.laser_pulse_initial_position = -6.0;
pulse1.lambda0 = 1.0;
pulse1.normalized_amplitude = 1.0;

myfield.addPulse(&pulse1);
```

Here a 1D laser pulse is added to the EM field.
`type = COS2_PLANE_WAVE` sets the pulse type (use only `COS2_PLANE_WAVE` in 1D).
`polarization = P_POLARIZATION` sets the polarization of the laser pulse (options are: `P_POLARIZATION`, `S_POLARIZATION` and `CIRCULAR_POLARIZATION`).
`t_FWHM = 5.0` sets the FWHM of the pulse.
`laser_pulse_initial_position = -6.0` sets the initial position along x of the center of the pulse.
`lambda0 = 1.0` sets the wavelength of the pulse.
`normalized_amplitude = 1.0` sets the normalized amplitude of the pulse.

3.3.2 Gaussian laser pulse

```
laserPulse pulse1;
pulse1.type = GAUSSIAN;
pulse1.polarization = P_POLARIZATION;
pulse1.t_FWHM = 5.0;
pulse1.laser_pulse_initial_position = -6.0;
pulse1.lambda0 = 1.0;
pulse1.normalized_amplitude = 1.0;

pulse1.waist = 3.0;
pulse1.focus_position = 0.0;

pulse1.rotation = false;
pulse1.angle = 2.0*M_PI*(-90.0 / 360.0);
pulse1.rotation_center_along_x = 0.0;

myfield.addPulse(&pulse1);
```

This is an example of the definition of a Gaussian laser pulse.
In addition to the `COS2_PLANE_WAVE` pulse, in this case `waist` and `focus_position`.
In 2 or 3 dimensions, if an oblique incidence pulse is required, `rotation` needs to be set to `true`, `rotation_center_along_x` needs to be defined and an `angle` should be set (in radians).

3.3.3 EM field finalization

```
myfield.boundary_conditions();

current.allocate(&grid);
current.setAllValuesToZero();
```

These lines need to be called after pulse generation. They apply boundary conditions for the EM field and they allocate `current` (which is initialized to zero).

3.4 Plasma

```
PLASMA plasma1;
plasma1.density_function = left_soft_ramp;
plasma1.setXRangeBox(0.0,100.0);
plasma1.setYRangeBox(grid.rmin[1],grid.rmax[1]);
plasma1.setZRangeBox(grid.rmin[2],grid.rmax[2]);
plasma1.setDensityCoefficient(0.01);

plasma1.setRampLength(20.0);
plasma1.setRampMinDensity(0.0);
```

These lines define a plasma profile shaped as in figure 3.2.
The density function is `left_soft_ramp` ($\sin^2$ followed by constant density). `set*RangeBox` calls select the size of the box surrounding the plasma, while `setDensityCoefficient` sets the density (in units of n_c) of the constant density region of the plasma. The length of the ramp is set by `setRampLength` and `setRampMinDensity` sets the density before the ramp (0.0 in this example).

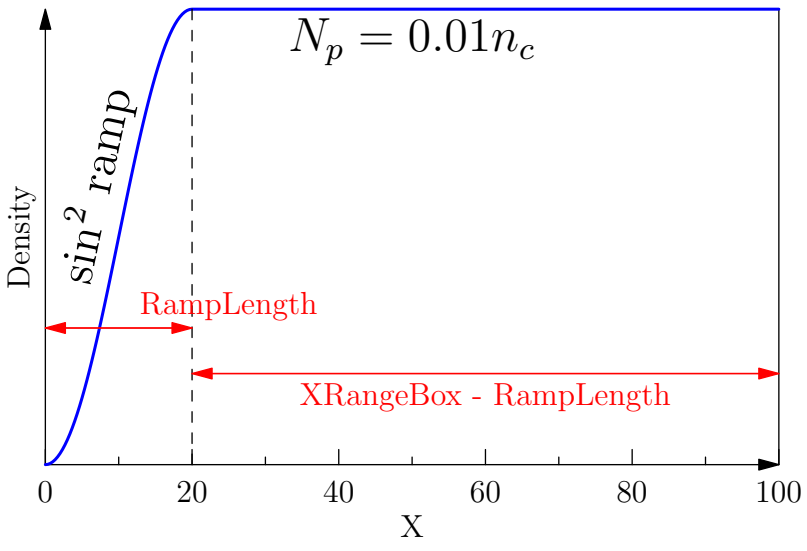


Figure 3.2: `left_soft_ramp` plasma profile

3.4.1 Other simple default plasma functions

`box`

This plasma shape is just a uniform box whose extremes are defined by the range parameters. Parameters related to the ramp are simply ignored.

`left_linear_ramp`

It's basically the same of the `left_soft_ramp` function. Simply the ramp is linear.

`left_grating`

This plasma function defines a grating target.
In addition to the settings for the soft ramp target, some additional parameters should be set as follows:

```
        PLASMA plasma1;
        plasma1.density_function = left_grating;
        plasma1.setXRangeBox(0.0,5.0);
        plasma1.setYRangeBox(grid.rmin[1],grid.rmax[1]);
        plasma1.setZRangeBox(grid.rmin[2],grid.rmax[2]);
        plasma1.setDensityCoefficient(100);

        plasma1.setRampLength(0.05);
        plasma1.setRampMinDensity(0.0);

        double grating_peak_to_valley_depth = 1.0;
        double grating_lambda = 0.5;
        double grating_phase = 0.0;

        double additionalParams[3];
        additionalParams[0] = grating_peak_to_valley_depth;
        additionalParams[1] = grating_lambda;
        additionalParams[2] = grating_phase ;

        plasma1.setAdditionalParams(additionalParams);
```

`left_grating` will be used as an example in subsection 3.4.3. Please refer to this subsection for further information.

3.4.2 Other functions

There are other plasma functions, such as `box_minus_box` and `rough_target`. Please refer to subsection 3.4.3 and to `structures.cpp` source file for further information.

3.4.3 How to implement a user-defined plasma function

If you want to implement a new plasma function, you have to define it in `main-1.cpp` file before use. In the following lines the definition of `left_grating` is reported as an example:

```
double left_grating(double x, double y, double z,
                    PLASMAparams plist, int Z, int A){
        double g_y0 = (plist.rmaxbox[1] - plist.rminbox[1])*0.5;
        double* paramlist = (double*)plist.additional_params;
        double g_depth = paramlist[0] * 0.5;
        double g_lambda = paramlist[1];
        double g_phase = paramlist[2];

        double phase = 2.0 * M_PI * ((y - g_y0) + g_phase) / g_lambda;
        double xminbound = plist.rminbox[0] +
                g_depth*(1.0 - cos(phase));

        if ((xminbound <= x) && (x <= plist.rmaxbox[0]) &&
```

```
        (plist.rminbox[1] <= y) && (y <= plist.rmaxbox[1]) &&
        (plist.rminbox[2] <= z) && (z <= plist.rmaxbox[2])){
                if ((x - xminbound) <= plist.ramp_length){
        return (plist.density_coefficient-plist.ramp_min_density)*
                        (x - xminbound) / plist.ramp_length +
                        plist.ramp_min_density;
                }
                else{
                        return plist.density_coefficient;
                }
        }
        else{
                return -1;
        }
}
```

It is worth to mention that, during the particle creation phase, the number of particles to be created is pre-calculated. This means that the density function is evaluated twice over each grid point. Consequently, care should be taken if something in the plasma function depends on randomly generated quantities (the best approach in this case is to pre-calculate all the random parameters in a separate function).

3.5 Species

```
        SPECIE  electrons1(&grid);
        electrons1.plasma = plasma1;
        electrons1.setParticlesPerCellXYZ(100, 1, 1);
        electrons1.setName("ELE1");
        electrons1.type = ELECTRON;
        electrons1.creation();
        species.push_back(&electrons1);
```

These lines create and initialize an electron species. A `PLASMA` object should be passed to define the plasma shape (in this case the plasma will be a `left_soft_ramp`).
`setParticlesPerCellXYZ` sets the number of particles per cell along x, y and z (100x1x1 in this case).
`setName` sets the species name. Each species should have a different name, or part of the simulation output may be overwritten.
`type` selects the species type. Options are: `ELECTRON`, `IONS` and `POSITRON`.
Finally, the lines

```
        electrons1.creation();
        species.push_back(&electrons1);
```

create the species `electrons1` and add it to the species vector.

```
        SPECIE ions1(&grid);
        ions1.plasma = plasma1;
        ions1.setParticlesPerCellXYZ(100, 1, 1);
        ions1.setName("ION1");
        ions1.type = ION;
        ions1.Z = 6.0;
        ions1.A = 12.0;
        ions1.creation();
        species.push_back(&ions1);
```

Here, carbon ions are created, with the same plasma profile of `electrons1`. In addition to the previous options, here `Z` and `A` should be set. Actually, only the ratio of `Z` over `A` is important.

```
        tempDistrib distribution;
        distribution.setMaxwell(1.0e-5);

        electrons1.add_momenta(rng, 0.0, 0.0, 0.0, distribution);
        ions1.add_momenta(rng,0.0, 0.0, 0.0, distribution);
```

These lines apply a Maxwell distribution to the species, with a temperature of 10^{-5} in code units (see appendixes).
It is also possible to add a momentum to each particle (after the application of the temperature distributions). For example, if a drift motion along z is required:

```
        electrons1.add_momenta(rng, 0.0, 0.0, 1.0, distribution);
```

The available distribution functions are listed hereunder:

`setWaterbag(p0)`	Waterbag distribution (momenta between $-p_0$ and p_0)
`setWaterbag3Temp(p0_x, p0_y,p0_z)`	the same, but with three different momenta on x,y,z
`setUnifSphere(p0)`	momentum distribution is a uniform sphere of radius p_0
`setSupergaussian(p0, alpha)`	spherical supergaussian distribution (radius p_0, exponent α)
`setMaxwell(temp)`	Maxwell distribution

WARNING: at the moment, if the moving window is activated, new particles entering in the simulation area are cold and with no drift. This will be fixed in future releases of the code.

3.6 Diagnostics

The following lines set up the diagnostics of the simulation.

```
        OUTPUT_MANAGER manager(&grid, &myfield, &current, species);
```

An `OUTPUT_MANAGER` object is deputed to manage all the diagnostics of the simulation. If some of the arguments are not valid, the output manager may not function properly.

```
        manager.addEMFieldBinaryFrom(0.0, 2.0);

        manager.addSpecDensityBinaryFrom(electrons1.name, 0.0, 2.0);
        manager.addSpecDensityBinaryFrom(ions1.name, 0.0, 2.0);

        manager.addCurrentBinaryFrom(0.0, 5.0);

        manager.addSpecPhaseSpaceBinaryFrom(electrons1.name, 0.0, 5.0);

        manager.addDiagFrom(0.0, 1.0);
```

These lines of code can be modified by the user according to its needs: they define exactly which diagnostics will be saved on disk and how frequently this will be done.

```
        manager.addEMFieldBinaryFrom(0.0, 2.0);
```

For example, the first line asks for the binary output of EM field every 2 τ_p, starting from time 0.0 .
It is also possible to ask for the output between two times with a given frequency or at a precise time. In these cases, the calls are as follows:

```
        manager.addEMFieldBinaryFromTo(START, FREQUENCY, END);
        manager.addEMFieldBinaryAt(TIME);
```

For the density output and the phase space output, it is necessary to specify also the name of the target species, as defined in section 3.5. Make sure that the name is written correctly, or the requested output will be ignored (a warning message is displayed in this case).
If two output requests overlap, this is automatically detected by the output manager and no duplicated

output is saved on disk.
A quick reference of the available diagnostics is given in the following table. A more detailed discussion about how to handle the output files can be found in chapter 5

EMFieldBinary	binary output of the EM field
SpecDensityBinary	binary output of the charge density of a given species
SpecPhaseSpaceBinary	binary output of the phase space of a given species
CurrentBinary	binary output of the total current density
Diag	lightweight text output (total energy, maximum momentum ...)

```
manager.initialize(DIRECTORY_OUTPUT);
```

The output manager should always be initialized passing `DIRECTORY_OUTPUT` as an argument.

3.7 Temporal cycle

The user should not modify the temporal cycle without a good reason. Anyway, the code is described here for reference purposes.

```
if (grid.myid == grid.master_proc){
        printf("-----␣START␣temporal␣cicle␣-----\n");
        fflush(stdout);
}

int Nstep = grid.getTotalNumberOfTimesteps();
for (istep = 0; istep <= Nstep; istep++)
{
        grid.istep = istep;

        grid.printTStepEvery(FREQUENCY_STDOUT_STATUS);


        manager.callDiags(istep);

        myfield.openBoundariesE_1();
        myfield.new_halfadvance_B();
        myfield.boundary_conditions();

        current.setAllValuesToZero();

        for (spec_iterator = species.begin();
                spec_iterator != species.end();
                spec_iterator++)
        {
(*spec_iterator)->current_deposition_standard(&current);
        }

        current.pbc();

        for (spec_iterator = species.begin();
                spec_iterator != species.end();
                spec_iterator++)
        {
                (*spec_iterator)->position_parallel_pbc();
        }
```

```
                myfield.openBoundariesB();
                myfield.new_advance_E(&current);

                myfield.boundary_conditions();

                myfield.openBoundariesE_2();
                myfield.new_halfadvance_B();

                myfield.boundary_conditions();

                for (spec_iterator = species.begin();
                         spec_iterator != species.end();
                         spec_iterator++)
                {
                        (*spec_iterator)->momenta_advance(&myfield);
                }

                grid.time += grid.dt;


                grid.move_window();

                myfield.move_window();
                for (spec_iterator = species.begin();
                         spec_iterator != species.end();
                         spec_iterator++){
                        (*spec_iterator)->move_window();
                }
        }
```

3.8 Finalization

```
        manager.close();
        MPI_Finalize();
        exit(1);

}
```

In these last lines of the code, `manager` executes its closing routine, `MPI_Finalize()` is called and the program exits with code 1. No modification by the user is required.

Chapter 4

How to run a simulation

First of all, make sure that the output directory has been created.

4.1 Run as a serial job

To run as a serial job, make sure that in `main-1.cpp` source file you have:

```
#define NPROC_ALONG_Y 1
#define NPROC_ALONG_Z 1
```

Then, `cd` to the directory containing the source file and type simply:

```
$make
$./piccante
```

You should get something like:

```
         d8b                                        888
         Y8P                                        888
                                                    888
88888b.  888  .d8888b .d8888b  8888b.  88888b.  888888 .d88b.
888 "88b 888 d88P"   d88P"        "88b 888 "88b 888   d8P  Y8b
888  888 888 888     888      .d888888 888  888 888   88888888
888 d88P 888 Y88b.   Y88b.    888  888 888  888 Y88b. Y8b.
88888P"  888  "Y8888P "Y8888P "Y888888 888  888  "Y888 "Y8888
888
888
888
================   version 1.0.0.0  ================
==========  GRID_INITIALIZATION ==========
master_proc=0:
Nproc   =     1 :     (    1,    1,    1)

single proc id and 3D coordinates:
    id = ( idx, idy, idz)
     0 = (   0,   0,   0)
==========          grid          ==========
  id:  Nloc = [   rmin :   rmax ]
X:  #proc=1 Nx=2501
               0:  2501 = [    -50 :      0 ]
Y:  #proc=1 Ny=1
               0:     1 = [     -1 :      1 ]
Z:  #proc=1 Nz=1
```

```
               0:     1 = [     -1 :      1 ]
==========                      ==========
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
dt=0.019600
dx=0.020000
dy=2.000000
dz=2.000000
Nstep=3061
Boundaries (X,Y,Z) : OPEN , PBC , PBC
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
----- START temporal cicle -----
     0/3061  0.000000   18:38:11  (29/01/2014)          0 sec.
     5/3061  0.098000   18:38:11  (29/01/2014)          0 sec.
    10/3061  0.196000   18:38:11  (29/01/2014)          0 sec.
    15/3061  0.294000   18:38:11  (29/01/2014)          0 sec.
    20/3061  0.392000   18:38:11  (29/01/2014)          0 sec.
```

These lines contains some information about the simulation. As far as the simulation time is of concern, please consider that the load balancing can vary during the simulation and thus the simulation speed may increase or decrease. It is worth to remember that writing output files on disk may be very time-consuming for large files.
You may want to redirect the standard output to a text file. If this is the case, launch the simulation as follows:

```
$./piccante > output.txt &
```

4.2 Run as a parallel job on a single multi-core machine

Make sure that `NPROC_ALONG_Y` times `NPROC_ALONG_Z` (in the `main-1.cpp`) is a multiple of the number of MPI tasks you want to launch.
Suppose, for example, that you want to launch a 2D simulation on a 8-core machine and that you have:

```
#define NPROC_ALONG_Y 4
#define NPROC_ALONG_Z 1
```

Launching the code with:

```
$mpirun -np 8 ./piccante
```

will result in having 4 processes along y and 2 along x.
You should get on the standard output a longer initial message, detailing the boundaries of each process.
If you want to redirect the standard output to a text file, type:

```
$mpirun -np 8 ./piccante > output.txt &
```

4.3 Run on a BG/Q supercomputer

Ask one of the contributors for the appropriate compile script.

Chapter 5

How to analyse the results

Most of the output files are in binary format. The code comes with a *reader* tool, which is able to covert the binary file into a text file. Then, the text file can be visualized using *Gnuplot*(`http://www.gnuplot.info/`) or *Paraview*(`http://www.paraview.org/`). *Paraview* is suggested for 3D plots.

5.1 Get the reader

reader is available as an open-source repository. To get the reader, simply type (see 2.1):

```
git clone https://github.com/ALaDyn/tools-pic.git
```

5.2 Compile the reader

To compile *reader*, it is sufficient to type:

```
$g++ field_reader.cpp -O3 -o reader
```

Then, you may want to launch *reader* from any location on your system. To make it possible, copy the executable in the `\usr\bin` folder.

```
$sudo cp reader /usr/bin/reader
```

5.3 Human readable outputs

5.3.1 diag.dat file

`diag.dat` is a text file created by the *master proc* at the beginning of the simulation. Each call to this output produces a new line in the file.
`diag.dat` contains information about the total energy of the simulation and how it is distributed among fields and particle species.
From left to right, the columns in the file are:

- simulation step
- simulation time
- total energy
- energy contribution of E_x EM field component
- energy contribution of E_y EM field component
- energy contribution of E_z EM field component

- energy contribution of B_x EM field component
- energy contribution of B_y EM field component
- energy contribution of B_z EM field component
- energy contribution of species 1
- energy contribution of species 2
- ...

For the visualisation in a terminal, due to the large number of columns, you may want to use the command:

```
$less -S diag.dat
```

5.3.2 EXTREMES_* files

If the *Diag* output is enabled, one *Extremes* file will be produced for the EM field (`EXTREMES_EMfield.dat`). In addition, for each species in the simulation, an *Extremes* file will be written on disk (for "ELE1" species, it will be `EXTREMES_ELE1.dat`).
The columns of the *Extremes* file for the EM field are:

- simulation step
- simulation time
- minimum value of E_x EM field component
- maximum value of E_x EM field component
- ... (the same for each field component)
- maximum value of $|\mathbf{E}|$
- maximum value of $|\mathbf{B}|$

For the *Extremes* file of a species the columns are:

- simulation step
- simulation time
- minimum value of the x coordinate of the particles
- maximum value of the x coordinate of the particles
- ... (the same for y, z, p_x, p_y, p_z)
- minimum value of $\gamma - 1$ factor
- maximum value of $\gamma - 1$ factor

If there are no particles in a given time-step (this may happen for example if you use the moving window), large numbers ($\pm 1.0e30$) are reported in the *Extremes* file. Also for these files, you may want to use the command `less -S` due to the large number of columns.

5.3.3 SPECTRUM_* files

This diagnostic is enabled with *Diag* output and it produces a separate file for each species and each output time. The syntax of these files is `SPECTRUM_NAME_TIME.dat`.
Each file contains an energy spectrum of a given species at a given time. You can visualize these spectra with *gnuplot*:

```
$plot "SPECTRUM_ELE1_020.000.dat" u ($1+$2):3 w l
```

5.3.4 *.map files

These files are basically for debug purpose only. They always come in pair with a binary file and they contain some of the information included in the binary header in a human readable format.

5.4 Binary outputs

Among the binary (`*.bin`) output files, all but the phase space files require *reader* to be analysed.

5.4.1 EM field files

These files (`EMfield_TIME.bin`) contains the EM field of the simulation at a given time-step.
First, launch *reader* passing the file name as an argument (this is usually extremely fast for 1D simulation, while it can take several minutes for 2D or 3D simulations):

```
$reader EMfield_050.000.bin
```

If you haven't or you couldn't copy *reader* in `\usr\bin`, copy the executable in the output folder and launch it as:

```
$./reader EMfield_050.000.bin
```

This command produces a file with a `.txt` extension (`EMfield_050.000.bin.txt`) in this case. The columns of this file are, respectively: x, y, z, E_x, E_y, E_z, B_x, B_y, B_z. So, if you want to plot the B_x field component of a 2D simulation with *gnuplot*, launch the application and type:

```
$plot "EMfield_050.000.bin.txt" u 1:2:7 w image
```

If you have used the stretched grid feature, plotting with image won't work correctly. In this case you have to type:

```
$set pm3d map
$splot "EMfield_050.000.bin.txt" u 1:2:7 w pm3d
```

You can avoid launching the reader for every single file implementing some simple *bash* scripts. The following script, for example, plots the E_x field component and saves it as a `.png` file for each *EMfield* file in the `TEST` directory:

```
#!/bin/bash
echo "set term pngcairo size 1000,1000">gnuplot.scr
echo "set size ratio -1">>gnuplot.scr
for f in TEST/EMfield_*.bin
        do ./reader $f
        echo "set output '${f}_Ex.png'">>gnuplot.scr
        echo "plot '${f}.txt' u 1:2:4 w image">>gnuplot.scr
done
gnuplot gnuplot.scr
```

For the visualization of 3D data it is strongly suggested to use *paraview*.

5.4.2 Density files

Most of what has been said for *EMfield* files is valid for *Density* files. Their name structure is `DENS_SPECNAME_TIME.bin` and, once converted with *reader*, they are 4 columns files (x, y, z, ρ). The charge density is always positive.
Suppose that you want the *total charge density* in a simulation where you have one electron species "ELE1" and one ion specie "ION1". With *gnuplot* you can accomplish this as follows:

```
plot "< paste DENS_ION1_024.000.bin.txt DENS_ELE1_024.000.bin.txt" u 1:($4-$8) w l
```

5.4.3 Current files

Most of what has been said for *EMfield* files is valid for *Density* files. Their name structure is `J_TIME.bin` and, once converted with *reader*, they are 6 columns files (x, y, z, J_x, J_y, J_z).

5.4.4 PhaseSpace files

The *PhaseSpace* files (`PHASESPACE_NAME_TIME.bin`) have a very simple structure: for each particle, 7 float numbers are saved. These numbers are, respectively: x, y, z, p_x, p_y, p_z and the weight of the particle. Due to their simplicity, these files can be plotted with *gnuplot*:

```
$plot "PHASESPACE_ELE1_055.000.bin" binary format="%f%f%f%f%f%f%f" u 1:4 w d
```

For example, this command plots the $x - p_x$ projection of the phase space.
Since these files are usually very large, you may want to plot just a fraction of the particles. Suppose that you want to plot one particle every 100:

```
$plot "PHASESPACE_ELE1_055.000.bin" binary format="%f%f%f%f%f%f%f" every 100 u 1:4 w d
```

Chapter 6

Main files collection

In this section, the example main files in the `Example` directory are briefly discussed. To launch these simulations, you have to copy the corresponding main file in the source directory, rename it `main-1.cpp` and compile the code.
The 2D and 3D simulations provided in the `Example` folder are designed to be launched on a supercomputer, due to the large computational requirements. However, it is possible to run "small" 2D simulations also on personal computers.

6.1 1D wakefield with moving window

main file name	`simple_1d_wakefield.cpp`
short description	simple simulation of a nonlinear wakefield with moving window
dimensionality	1D
expected execution time	around 1 minute on a modern machine (4 MPI processes)

In this simulation, a P polarized laser pulse with a normalized amplitude of 1.0 enters in an underdense plasma (density of 0.01, with a soft $\sin^2$ ramp). The moving window feature is used. It is worth to remind that, for the moment, new plasma is added with zero temperature.
Figure 6.1 reports the phase space of the electrons after 50.0 periods.

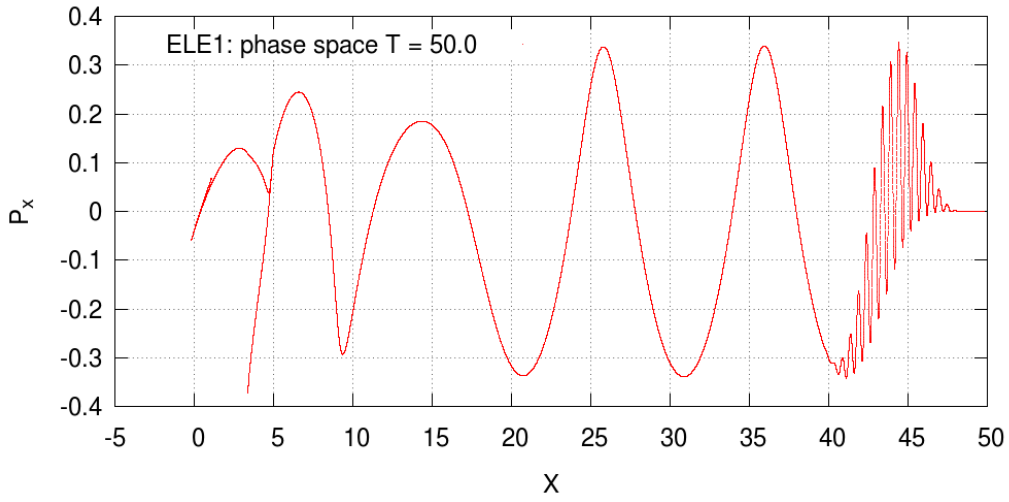


Figure 6.1: 1D wakefield: electrons phase space

6.2 1D simple TNSA simulation

main file name	`1d_TNSA.cpp`
short description	simple simulation of TNSA ion acceleration
dimensionality	1D
expected execution time	around 1 minute on a modern machine (4 MPI processes)

A P polarized laser pulse with a normalized amplitude of 8.0 interacts with a thin (1.5 λ) carbon target("ION1"). There's a very thin layer of hydrogen contaminants on the rear side of the target. Picture 6.2 shows the phase space of carbon ions.

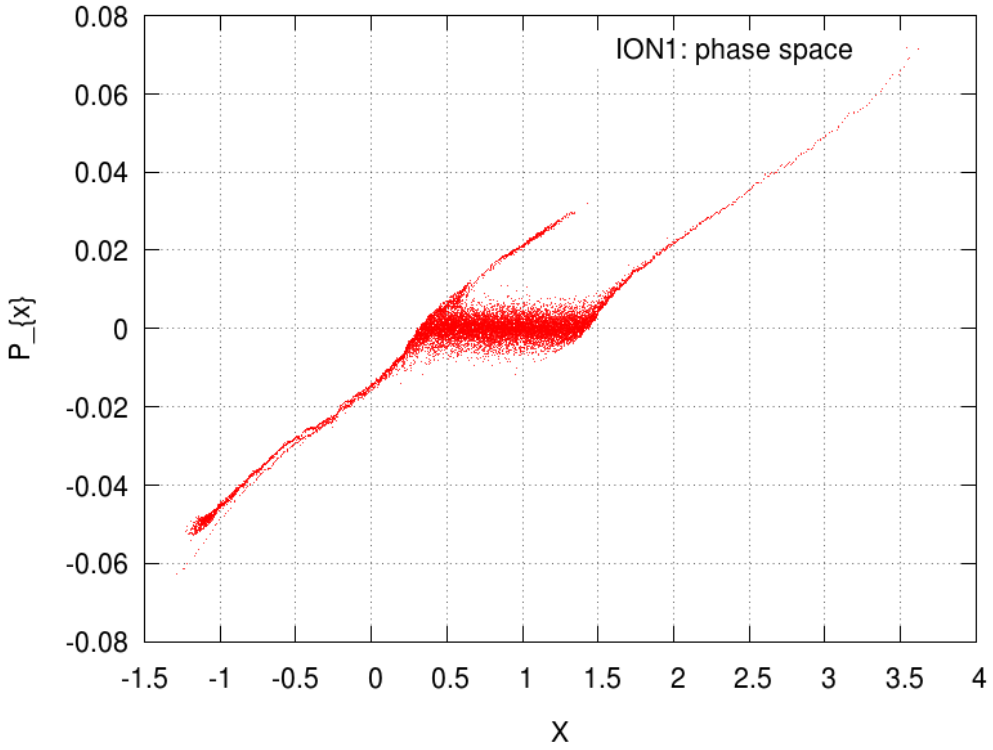


Figure 6.2: 1D TNSA: ion1 phase space

6.3 2D grating with stretched grid and 2 laser pulses

main file name	`2d_grating.cpp`
short description	two laser pulses a grating target
dimensionality	2D
expected execution time	around 1.5 hours on a supercomputer (1024 MPI processes on a BG/Q)

In this simulations, two inclined laser pulses interact with an overdense grating target.
Several features of emphpiccante code are used in this simulation, such as the stretched grid, complex target shapes and multiple laser pulses. Figure 6.3 shows the initial setup of this simulation.

6.4 3D two stream instability

main file name	`3d_twostream.cpp`
short description	two counter-streaming electron clouds
dimensionality	3D
expected execution time	several hours on a supercomputer (1024 MPI processes on a BG/Q)

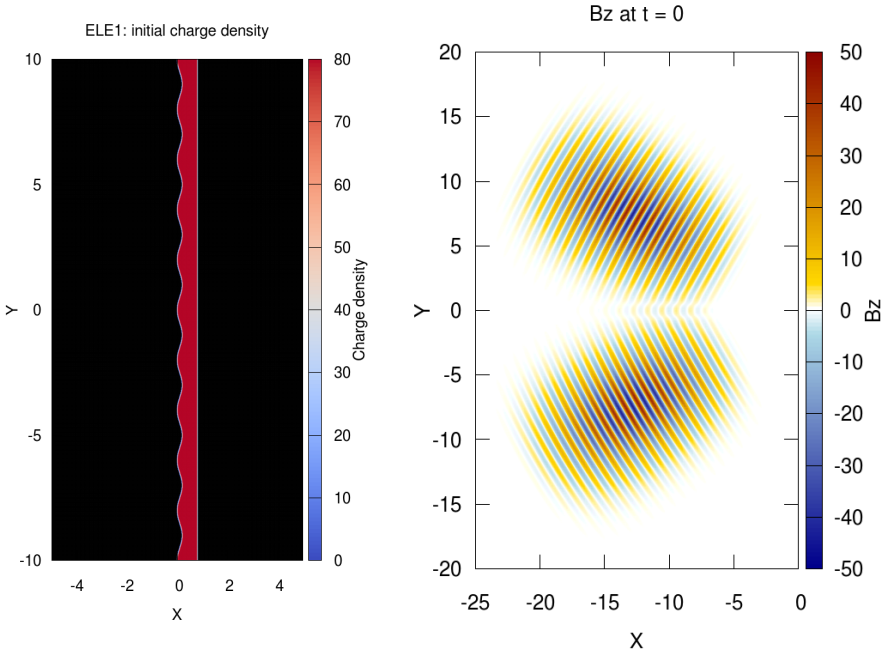


Figure 6.3: 2D grating: The initial setup (target and Bz field component)

In this simulation there are two counter-streaming electron clouds with normalized momentum along z equal to 1. There is a very small initial temperature.
An instability develops and a significant fraction of the initial kinetic energy is converted into EM field energy.
Figure 6.4 shows the charge density of one of the electron species after a few plasma periods.

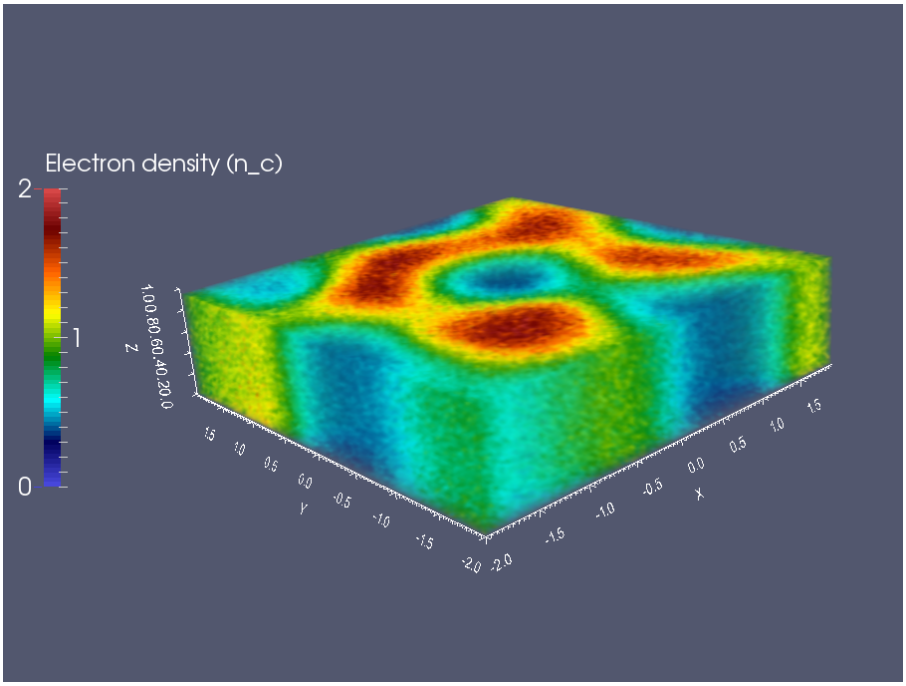


Figure 6.4: 3D twostream: development of the instability(ELE1 charge density)

Appendix A: code normalization

Everything in the code scales with ℓ_0 scale-length.
m_e and q_e are, respectively, the mass and the charge of an electron, while c is the speed of light.

Normalization of code quantities

$$
\begin{aligned}
q &= \widetilde{q}\, q_e \\
m &= \widetilde{m}\, m_e \\
\mathbf{E} &= \widetilde{\mathbf{E}}\, \frac{m_e c^2}{q_e \ell_0} \\
\mathbf{B} &= \widetilde{\mathbf{B}}\, \frac{m_e c^2}{q_e \ell_0} \\
t &= \widetilde{t}\, \frac{\ell_0}{c} \\
l &= \widetilde{l}\, \ell_0 \\
\mathbf{p} &= \widetilde{\mathbf{p}} \widetilde{m}\, m_e c \\
\mathbf{v} &= \widetilde{\mathbf{v}}\, c \\
\mathbf{J} &= \widetilde{\mathbf{J}}\, \frac{m_e c^3 \pi}{q_e \ell_0}
\end{aligned}
$$

Normalized Particle equations

$$
\begin{cases}
\partial_{\widetilde{t}} \widetilde{p} = & \frac{\widetilde{q}}{\widetilde{m}} \left(\widetilde{\mathbf{E}} + \widetilde{\mathbf{v}} \times \widetilde{\mathbf{E}} \right) \\
\widetilde{\mathbf{J}} = & \widetilde{q} \sum_n \frac{W_n}{\Delta \widetilde{V}} \widetilde{\mathbf{v}}_n
\end{cases}
$$

Normalized Maxwell equations

$$
\begin{cases}
\partial_{\widetilde{t}} \widetilde{\mathbf{B}} = & -\widetilde{\nabla} \times \widetilde{\mathbf{E}} \\
\partial_{\widetilde{t}} \widetilde{\mathbf{E}} = & \widetilde{\nabla} \times \widetilde{\mathbf{B}} - 4\pi^2 \widetilde{\mathbf{J}}
\end{cases}
$$