Diagnostics in SMILEI

F. Pérez

May 28, 2015

Contents

1 Scalars

SMILEI can collect various scalar data, such as total particle energy, total field energy, etc. This is done if the following block is included in the input file.

```
Scalar( every     = every,
        tmin = tmin,
        tmax = tmax,
        precision = precision )
```

- *every* is the number of timesteps between each output.
- *tmin* and *tmax* (optional) are the min and max times that will be used.
- *precision* (optional) is the number of digits of the outputs. Default = 10.

In the input file, you may add the following block as a reminder.

```
# DIAGNOSTICS ON SCALARS
# every = integer, number of time-steps between each output
# tmin and tmax = floats, min and max times that will be used
# precision = integer, number of digits of the outputs. Default = 10
```

The full list of scalars is:

Emw_fields	Field energy added through the moving window
Emw_lost_fields	Field energy lost through the moving window
Emw_part	Particle energy added through the moving window
Emw_lost	Particle energy lost through the moving window
Ebal_norm	Ebalance / Etot
Ebalance	Current energy − initial total energy
Etot	Current total energy
Elost	Lost particle energy during last timestep
Eparticles	Current energy in particles
EFields	Current energy in fields
Poynting	Accumulated Poyting flux through all boundaries
Z_abc	Average charge of species "abc"
E_abc	... their kinetic energy
N_abc	... and number of particles
Ex_U	$\int E_x^2 dV/2$
	... and similar for fields Ey, Ez, Bx_m, By_m and Bz_m
ExMin	Minimum of E_x
ExMinCell	... and its location (cell index)
ExMax	Maximum of E_x
ExMaxCell	... and its location (cell index)
	... and same things for fields Ey, Ez, Bx_m, By_m, Bz_m, Jx, Jy, Jz and Rho
PoyEast	Accumulated Poynting flux through eastern boundary
PoyEastInst	Current Poynting flux through eastern boundary
	... and same things for boundaries West, South, North, Bottom, Top

2 Fields

SMILEI can collect various field data (electric fields, magnetic fields, currents and density) taken at the location of the PIC grid, both as instantaneous values and averaged values. This is done if the following instructions are included in the input file.

```
fieldDump_every    = every
avgfieldDump_every = every_avg
ntime_step_avg     = t_avg
```

where

- *every* is the number of timesteps between each output of the instantaneous fields.
- *every_avg* is the number of timesteps between each output of the time-averaged fields.
- *t_avg* is the number of timesteps for time-averaging.

The full list of fields is:

Bx_m By_m Bz_m	Components of the magnetic field
Ex Ey Ez	Components of the electric field
Jx Jy Jz	Components of the total current
Jx_abc Jy_abc Jz_abc	Components of the current due to species "abc"
Rho	Total density
Rho_abc	Density of species "abc"

3 Probes

The fields from the previous section are taken at the PIC grid locations, but it is also possible to obtain the fields at arbitrary locations. These are called *probes*.
The probes interpolate the fields at either one point (0-D), several points arranged in a line (1-D) or several points arranged in a mesh (2-D).
To add one probe diagnostic in the input file, there are several cases:
For only one point (zero-dimensional probe)

```
DiagProbe(
    every      = every,
    pos        = [x0, y0, z0]
)
```

where

- `every` is the number of timesteps between each output.
- `x0`, `y0`, `z0` is the position of the point where to interpolate the fields.
  Note that `y0` (or `z0`) should only be used in the case of a 2-D (or 3-D) simulation.

For a series of points arranged in a line (one-dimensional probe)

```
DiagProbe(
    every      = every,
    pos        = [x0, y0, z0],
    pos_first  = [x1, y1, z1],
    number     = [n1]
)
```

where

- `x0`, `y0`, `z0` is the position of the starting point of the line.
- `x1`, `y1`, `z1` is the position of the ending point of the line.
- `n1` is the number of points along this line.

For a series of points arranged in a mesh (two-dimensional probe)

```
DiagProbe(
    every      = every,
    pos        = [x0, y0, z0],
    pos_first  = [x1, y1, z1],
    pos_second = [x2, y2, z2],
    number     = [n1, n2]
)
```

In this case, the three points define three vertices of a paralellogram.

Notes

- Probes only output the electromagnetic fields, the total current and the total density.
- The dimension of the probe is decided only by the instruction `number`: without it, the probe is 0-D, with `number = [n1]`, the probe is 1-D, and with `number = [n1, n2]`, the probe is 2-D.
- You can have several probes in the input file.

Examples of probe diagnostics

0-D probe in 1-D simulation

```
DiagProbe(
    every = 1,
    pos   = [1.2]
)
```

1-D probe in 1-D simulation

```
DiagProbe(
    every = 1,
    pos       = [1.2],
    pos_first = [5.6],
    number    = [100]
)
```

1-D probe in 2-D simulation

```
DiagProbe(
    every = 1,
    pos       = [1.2,  4.],
    pos_first = [5.6,  4.],
    number    = [100]
)
```

2-D probe in 2-D simulation

```
DiagProbe(
    every = 1,
    pos        = [0. ,   0.],
    pos_first  = [10. ,  0.],
    pos_second = [0.,    10.],
    number     = [100,   100]
)
```

4 Particle diagnostics

A *particle diagnostic* can collect data from the macro-particles and processes them during runtime. It does not provide information on individual particles: instead, it produces averaged quantities like the particle density, currents, etc.

The data may be collected from one or several particle species.

The data is discretized inside a "grid" chosen by the user. This grid may be of any dimension.
Examples:

- 1-dimensional grid along the position x (gives density variation along x)
- 2-dimensional grid along positions x and y (gives density map)
- 1-dimensional grid along the velocity v_x (gives the velocity distribution)
- 2-dimensional grid along position x and momentum p_x (gives the phase-space)
- 1-dimensional grid along the kinetic energy E_{kin} (gives the energy distribution)
- 3-dimensional grid along x, y and E_{kin} (gives the density map for several energies)
- 1-dimensional grid along the charge $Z^\star$ (gives the charge distribution)

Each dimension of the grid is called "axis".

The user may choose to average the data over several time-steps.

In the input file, you can add a particle diagnostic with the following block.

```
DiagParticles(
        output = output,
        every = every,
        time_average = time_average,
        species = [species1, species2, ...],
        axes = [
                [type, min, max, nsteps, "logscale", "edge_inclusive"],
                [type, min, max, nsteps, "logscale", "edge_inclusive"],
                ...
        ]
)
```

- *output* is a string: `"density"`, `"charge_density"`, `"current_density_x"`, `"current_density_y"`, `"current_density_z"`, `"p_density"`, `"px_density"`, `"py_density"` or `"pz_density"`.
  It determines the data that is summed in each cell of the grid.
  In the case of `"density"`, the *weights* are summed.
  In the case of `"charge_density"`, the *weights*×*charge* are summed.
  In the case of `"current_density_x"`, the *weights*×*charge*×v_x are summed (etc.).
  In the case of `"p_density"`, the *weights*×p are summed (etc.)
- *every* is a positive integer: the number of time-steps between each output.

- *time_average* is a positive integer: the number of time-steps during which the data is averaged before output.
- *species1, species2, ...* is a list of strings: the names of one or several species. Species are recognized by their parameter `species_type` that is included in each group `Species( ... )`
- `axes` is a list of "axes".
  Syntax of one axis: `[`*type, min, max, nsteps,* `"logscale", "edge_inclusive"]`
  type is one of `"x"`, `"y"`, `"z"`, `"px"`, `"py"`, `"pz"`, `"p"`, `"gamma"`, `"ekin"`, `"vx"`, `"vy"`, `"vz"`, `"v"` or `"charge"`.
  The axis is discretized for *type* from *min* to *max* in *nsteps* bins.
  The optional keyword `logscale` sets the axis scale to logarithmic instead of linear.
  The optional keyword `edge_inclusive` includes the particles outside the range [*min*, *max*] into the extrema bins.

There may be as many axes as wanted in one `DiagParticles( ... )` block.

There may be as many `DiagParticles( ... )` blocks as wanted in the input file.

In the input file, you may add the following block as a reminder.

```
# DIAGNOSTICS ON PARTICLES - project the particles on a N-D arbitrary grid
# ------------------------------------------------------------------------
# output       = string: "density", "charge_density" or "current_density_[xyz]"
#                parameter that describes what quantity is obtained
# every        = integer > 0: number of time-steps between each output
# time_average = integer > 0: number of time-steps to average
# species      = list of strings, one or several species whose data will be used
# axes         = list of axes
# Each axis is a list: [_type_,_min_,_max_,_nsteps_,"logscale","edge_inclusive"]
#   _type_ is a string, one of the following options:
#      x, y, z, px, py, pz, p, gamma, ekin, vx, vy, vz, v or charge
#   The data is discretized for _type_ between _min_ and _max_, in _nsteps_ bins
#   The optional "logscale" sets the scale to logarithmic
#   The optional "edge_inclusive" forces the particles that are outside (_min_,_max_)
#     to be counted in the extrema bins
#   Example : axes = [["x", 0., 1., 30]]
#   Example : axes =[["px", -1., 1., 100, "edge_inclusive"]]
```

Examples of particle diagnostics

Variation of the density of species `electron1` from $x = 0$ to 1, every 5 time-steps, without time-averaging.

```
DiagParticles(
        output = "density",
        every = 5,
        time_average = 1,
        species = ["electron1"],
        axes = [ ["x",    0.,    1.,    30] ]
)
```

Density map from $x = 0$ to 1, $y = 0$ to 1.

```
DiagParticles(
        output = "density",
        every = 5,
        time_average = 1,
        species = ["electron1"],
        axes = [ ["x",    0.,    1.,    30],
                 ["y",    0.,    1.,    30] ]
)
```

Velocity distribution from $v_x = -0.1$ to 0.1.

```
DiagParticles(
        output = "density",
        every = 5,
        time_average = 1,
        species = ["electron1"],
        axes = [ ["vx",   -0.1,    0.1,    100] ]
)
```

Phase space from $x = 0$ to 1 and from $px = -1$ to 1.

```
DiagParticles(
        output = "density",
        every = 5,
        time_average = 1,
        species = ["electron1"],
        axes = [ ["x",    0.,    1.,    30],
                 ["px",   -1.,   1.,    100] ]
)
```

Energy distribution from 0.01 to 1 MeV in logarithmic scale. Note that the input units are $m_e c^2 \sim 0.5$ MeV.

```
DiagParticles(
        output = "density",
        every = 5,
        time_average = 1,
        species = ["electron1"],
        axes = [ ["ekin",    0.02,    2.,   100, "logscale"] ]
)
```

x-y density maps for three bands of energy: $[0,1]$, $[1,2]$, $[2,\infty]$. Note the use of `edge_inclusive` to reach energies up to ∞.

```
DiagParticles(
        output = "density",
        every = 5,
        time_average = 1,
        species = ["electron1"],
        axes = [ ["x",    0.,    1.,    30],
                 ["y",    0.,    1.,    30],
                 ["ekin", 0.,    6.,    3,  "edge_inclusive"] ]
)
```

Charge distribution from $Z^\star = 0$ to 10.

```
DiagParticles(
        output = "density",
        every = 5,
        time_average = 1,
        species = ["electron1"],
        axes = [ ["charge",    -0.5,   10.5,   11] ]
)
```

5 How to view and post-process diagnostics

A python script *Diagnostics.py* is provided to view or extract data from all the diagnostics discussed above. To run this script, you will need *python2.7* with the following packages: numpy, matplotlib, pylab, h5py.

First, run python and include the script: `python -i scripts/Diagnostics.py`
Alternately, include this file into your own script: `execfile("scripts/Diagnostics.py")`

5.1 To select the simulation that you want to analyse

```
Smilei(results_path)
```

- `results_path` = `_string_`
  Path to the directory where the results of the simulation are stored.
  (Also, this has to contain one and only one input file *.in)

You can store this to a variable for later, for instance: `S = Smilei("path/to/my/results")`.

5.2 To prepare a scalar diagnostic

```
Smilei(...).Scalar(scalar=None, timesteps=None, units="code", data_log=False)
```

- `scalar` = `_string_` (optional)
  If not given, then a list of available scalars is printed.
  If `_string_`: name of the scalar (see available scalars in section 1).

- `timesteps` = `_int_` (optional)
  `timesteps` = `[_int_, _int_]` (optional)
  If omitted, all timesteps are used.
  If one number given, the nearest timestep available is used.
  If two numbers given, all the timesteps in between are used.

- `units` = `"nice"` (optional)
  If `"nice"` is chosen, then units are converted into usual units.
  Distances in microns, density in cm^{-3}, energy in MeV.

- `data_log` = `True` or (`False`) (optional)
  If True, then $\log_{10}$ is applied to the output.

5.3 To prepare a field diagnostic

```
Smilei(...).Field(field=None, timesteps=None, slice=None, units="code", data_log=False)
```

- `results_path`, `timesteps`, `units`, `data_log`: same as before.

- `field` = `_string_` (optional)
  If not given, then a list of available fields is printed.
  If `_string_`: name of the field (see available fields in section 2).
  The string can also be an operation between several fields, such as `"Jx+Jy"`.

- `slice` = { *axis* : `"all", ...` } (optional)
  `slice` = { *axis* : `_double_, ...` } (optional)
  `slice` = { *axis* : `[_double_, _double_], ...` } (optional)
  This parameter is used to reduce the number of dimensions of the array.
  axis must be `"x"`, `"y"` or `"z"`.
  The chosen axes will be removed:
  - If the value is `"all"`, then an average is performed over all the axis.
  - If the value is `_double_`, then only the bin closest to the value is kept.
  - If the value is `[_double_,_double_]`, then an average is performed between the two values.
  Example: `{"x":[4,5]}` will average all the data for x in the range [4,5].

5.4 To prepare a probe diagnostic

```
Smilei(...).Probe(probeNumber=None, field=None, timesteps=None, slice=None,
                  units="code", data_log=False)
```

- `results_path`, `timesteps`, `units`, `data_log`: same as before.

- `probeNumber` = `_int_` (optional)
  If not given, then a list of available probes is printed.
  If `_int_`: number of the probe. The first one has number 0.

- `field` = `_string_` (optional)
  If not given, then a list of available fields is printed.
  If `_string_`: field name, `"Bx"`, `"By"`, `"Bz"`, `"Ex"`, `"Ey"`, `"Ez"`, `"Jx"`, `"Jy"`, `"Jz"` or `"Rho"`.
  The string can also be an operation between several fields, such as `"Jx+Jy"`.

- `slice` is very similar to that of `Field()`, but it can only accept two axes: `"axis1"`, `"axis2"`. For instance, `slice={"axis1":"all"}`. Also, `"axis1"` and `"axis2"` are not necessarily x or y because the probe mesh may be rotated.

5.5 To prepare a particle diagnostic

```
Smilei(...).ParticleDiagnostic(diagNumber=None, timesteps=None, slice=None,
                               units="code", data_log=False)
```

- `results_path`, `timesteps`, `units`, `data_log`: same as before.

- `diagNumber` = `_int_` or `_string_` (optional)
  If not given, then a list of available particle diagnostics is printed.
  If `_int_`: number of the particle diagnostic. The first one has number 0.
  If `_string_`: operation between several particle diagnostics. For example, `"#0/#1"` computes the division by diagnostics 0 and 1.

- `slice` = { *axis* : `"all", ...` } (optional)
  `slice` = { *axis* : `_double_, ...` } (optional)
  `slice` = { *axis* : `[_double_, _double_], ...` } (optional)
  This parameter is used to reduce the number of dimensions of the array.
  axis must be `"x"`, `"y"`, `"z"`, `"px"`, `"py"`, `"pz"`, `"p"`, `"gamma"`, `"ekin"`, `"vx"`, `"vy"`, `"vz"`, `"v"` or `"charge"`.
  Any axis of the same name will be removed with the following technique:
  - If the value is `"all"`, then a sum is performed over all the axis.
  - If the value is `_double_`, then only the bin closest to the value is kept.
  - If the value is `[_double_,_double_]`, then a sum is performed between the two values.
  Example: `{"x":[4,5]}` will sum all the data for x in the range [4,5].

About Operations:
Sometimes, you have two (or more) diagnostics and you want to combine them with some operation. For example, imagine that you have one diagnostic with `output=px_density` and another one with `output=density`. You may want to divide the first by the second in order to obtain the average p_x.
This is directly possible here by using the argument `diagNumber`: you simply need to define this argument as a string containing an operation of your choice, and use a pound sign (#) to indicate each diagnostic. For example, the string `"#1 / #0"` will achieve the division between diagnostics #1 and #0. This feature is very useful as you can make any type of operation, as long as the two diagnostics have the same axes and the same timesteps.

5.6 To obtain the data as an array:

The method `getData` returns a list of the data arrays (for each timestep requested).

```
Smilei(...).Scalar(...).getData()
Smilei(...).Field(...).getData()
Smilei(...).Probe(...).getData()
Smilei(...).ParticleDiagnostic(...).getData()
```

The method `get` returns more things.

```
Smilei(...).Scalar(...).get()
Smilei(...).Field(...).get()
Smilei(...).Probe(...).get()
Smilei(...).ParticleDiagnostic(...).get()
```

This method returns the results as a python dictionary:

`{"data":`*data_array*`, "times":`*times_array*`, `*axis1*`:`*axis1_array*`, `*axis2*`:`*axis2_array*`, ...}`

where:

- *data_array* is the result of `getData()`.
- *times_array* is a list of the requested timesteps.
- *axis1*, *axis2* (...) are the names of the axes, if any (for example `"x"` or `"px"`).
- *axis1_array*, *axis2_array* (...) are the locations of the axes bins, if any.

Example:

```
>>> S = Smilei("path/to/my/results")
>>> Diag = S.ParticleDiagnostic(diagNumber=3, slice={"ekin":[1,10]}, timesteps=1000)
>>> result = Diag.get()
```

This will take the particle diagnostic #3 for the timestep nearest to 1000, and sum for all energies between 1 and 10. The results are stored in the variable `result`.
The data can be accessed with `result["data"]`.
If one of the axes is x, you can access the locations of the bins with `result["x"]`.

5.7 To plot the data

Use the method `plot` to display the data. New arguments are available to modify some plot options.

```
Smilei(...).Scalar            ( ... , figure=1, data_min=None, data_max=None,
                                   xmin=None, xmax=None, ymin=None, ymax=None ).plot()
Smilei(...).Field             ( ... , figure=1, data_min=None, data_max=None,
                                   xmin=None, xmax=None, ymin=None, ymax=None ).plot()
Smilei(...).Probe             ( ... , figure=1, data_min=None, data_max=None,
                                   xmin=None, xmax=None, ymin=None, ymax=None ).plot()
Smilei(...).ParticleDiagnostic( ... , figure=1, data_min=None, data_max=None,
                                   xmin=None, xmax=None, ymin=None, ymax=None ).plot()
```

- `figure` = `_int_` (optional)
  The figure number that is passed to matplotlib.
  If absent, figure 1 is used.

- `data_min` = `_double_` (optional)
  `data_max` = `_double_` (optional)
  If present, output is rescaled before plotting.

- `xmin` = `_double_` (optional)
  `xmax` = `_double_` (optional)
  `ymin` = `_double_` (optional)
  `ymax` = `_double_` (optional)
  If present, axes are rescaled before plotting.

If the data is **one**-dimensional, it is plotted as a **curve**, and is animated for all requested timesteps.
If the data is **two**-dimensional, it is plotted as a **map**, and is animated for all requested timesteps.
If the data is **zero**-dimensional, it is plotted as a **curve** as function of time.

Example:

```
>>> S = Smilei("path/to/my/results")
>>> S.ParticleDiagnostic(diagNumber=1, figure=1, data_min=0, data_max=1e14 ).plot()
```

This will take the particle diagnostic #1 and plot the resulting array in figure 1 from 0 to 3e14.

5.8 To simultaneously plot multiple diagnostics in the same figure

```
multiPlot(diag1, diag2, ... , figure=1, shape=None)
```

- `diag1` = diagnostic prepared by `Scalar()`, `Field()`, `Probe()` or `ParticleDiagnostic()`
  `diag2` = diagnostic prepared by `Scalar()`, `Field()`, `Probe()` or `ParticleDiagnostic()`
  ...
- `figure` = `_int_` (optional)
  The figure number that is passed to matplotlib.
  If absent, figure 1 is used.
- `shape` = `[_int_ , _int_]` (optional)
  The arrangement of plots inside the figure. For instance, `[2, 1]` makes two plots stacked vertically, and `[1, 2]` makes two plots stacked horizontally.
  If absent, stacks plots vertically.

Example:

```
>>> S = Smilei("path/to/my/results")
>>> A = S.Probe(probeNumber=0, field="Ex")
>>> B = S.ParticleDiagnostic(diagNumber=1)
>>> multiPlot( A, B, figure=1 )
```

This will plot the diagnostics #0 and #1 on the same figure, and make an animation for all available timesteps.

5.9 Advanced plotting options

In addition to `figure`, `data_min`, `data_max`, `xmin`, `xmax`, `ymin` and `ymax`, there are many more optional arguments. They are directly passed to the *matplotlib* package.

- Options for the figure: `figsize`, `dpi`, `facecolor`, `edgecolor`
  Please refer to http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.figure
- Options for the axes frame: `aspect`, `axis_bgcolor`, `frame_on`, `position`, `title`, `visible`, `xlabel`, `xscale`, `xticklabels`, `xticks`, `ylabel`, `yscale`, `yticklabels`, `yticks`, `zorder`
  Please refer to http://matplotlib.org/api/axes_api.html#matplotlib.axes.Axes.set
- Options for the lines: `color`, `dashes`, `drawstyle`, `fillstyle`, `label`, `linestyle`, `linewidth`, `marker`, `markeredgecolor`, `markeredgewidth`, `markerfacecolor`, `markerfacecoloralt`, `markersize`, `markevery`, `visible`, `zorder`
  Please refer to http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot
- Options for the image: `cmap`, `aspect`, `interpolation`
  Please refer to http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.imshow
- Options for the colorbar: `cbaspect`, `orientation`, `fraction`, `pad`, `shrink`, `anchor`, `panchor`, `extend`, `extendfrac`, `extendrect`, `spacing`, `ticks`, `format`, `drawedges`
  Please refer to http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.colorbar
- Options for the tick labels: `style_x`, `scilimits_x`, `useOffset_x`, `style_y`, `scilimits_y`, `useOffset_y`
  Please refer to http://matplotlib.org/api/axes_api.html#matplotlib.axes.Axes.ticklabel_format

Example:
To choose a gray colormap of the image, use `cmap="gray"`.

```
>>> S = Smilei("path/to/my/results")
>>> S.ParticleDiagnostic(0, figure=1, cmap="gray") .plot()
```

Many colormaps are available from the *matplotlib* package. With `cmap=""`, you will get a list of available colormaps.

5.10 Updating the plotting options

You can change the plotting parameters using the `A.plot()` or the `A.set()` functions.
Example:

```
>>> S = Smilei("path/to/my/results")
>>> A = ParticleDiagnostic(diagNumber=0, figure=1, data_max=1)
>>> A.plot( figure=2 )
>>> A.set( data_max=2 )
>>> A.plot()
```

5.11 Alternative syntax

Instead of `S = Smilei("path/to/my/results")`, you can directly choose which simulation you would like to use by specifying the `results_path` parameter as the first argument when calling a diagnostic.

For example, instead of

```
>>> S = Smilei("path/to/my/results")
>>> A = S.ParticleDiagnostic(0)
```

you may use

```
>>> A = ParticleDiagnostic("path/to/my/results", 0)
```

6 Tutorial

If you don't know how to run SMILEI, please refer to the appropriate documentation first. The commands can vary depending on your system and installation. A typical example of the command to run SMILEI is: `mpiexec -np 1 smilei mycase.in`

6.1 Running the test case

In the `benchmarks` directory, we provide a test case `tst1d_6_particle_diagnostic.in`. This case is very simple: it consists of a one-dimensional uniform neutral plasma composed by ions and electrons. The electrons all have a drift velocity of $0.05c$.

Run this case using SMILEI and collect the results in a directory of your choice. In this tutorial, we suppose that the results are in the directory `tst1d_6_particle_diagnostic`. Make sure this directory contains the input file `tst1d_6_particle_diagnostic.in` and the output files `ParticleDiagnostic0.h5`, `ParticleDiagnostic1.h5`, etc.

An example of the commands you may use from a UNIX *terminal* (or *console*) is

```
$ mkdir tst1d_6_particle_diagnostic
$ cp   benchmarks/tst1d_6_particle_diagnostic.in    tst1d_6_particle_diagnostic
$ cd tst1d_6_particle_diagnostic
$ mpiexec -np 1 smilei tst1d_6_particle_diagnostic.in
$ cd ..
```

6.2 Starting *python* and listing available diagnostics

From the same terminal, launch *python* using the command

```
$ python -i scripts/Diagnostics.py
```

Your are now in the *python* prompt.
Obtain a list of available particle diagnostics using

```
>>> S = Smilei('tst1d_6_particle_diagnostic')
>>> S.ParticleDiagnostic()
Printing available particle diagnostics:
----------------------------------------
Diag#0 - density of species # 1
    Every 4 timesteps, averaging over 2 timesteps
    x from 0.0 to 6.28319 in 100 steps
    vx from -0.1 to 0.1 in 100 steps
Diag#1 - density of species # 0
    Every 4 timesteps, no time-averaging
    x from 0.0 to 6.28319 in 100 steps
    vx from -0.001 to 0.001 in 100 steps
Diag#2 - density of species # 1
    Every 10 timesteps, averaging over 5 timesteps
    ekin from 0.0001 to 0.1 in 100 steps  [ LOG SCALE ]
```

Look at the diagnostic #0: it is the density of species #1 (here, electrons) with two axes: the position x and the velocity v_x. In other words, it is the phase-space of electrons.

6.3 Plot a diagnostic result at $t = 0$

To plot the phase-space in the initial conditions, use

```
>>> S.ParticleDiagnostic(0, timesteps=0 ).plot()
```

A window appears (see Figure 1). We can see that the electrons have indeed a drift velocity of $0.05c$.

To obtain the equivalent plot for the ions, use the diagnostic #1 with the command

```
>>> S.ParticleDiagnostic(1, timesteps=0 ).plot()
```

This results in the plot in Figure 2. The ions have a zero average velocity.

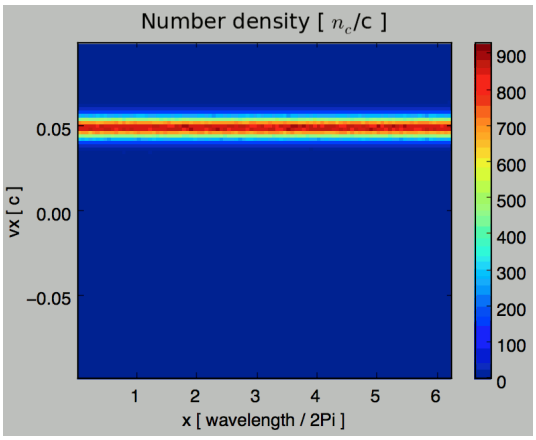

Figure 1: Phase-space of electrons at $t = 0$.

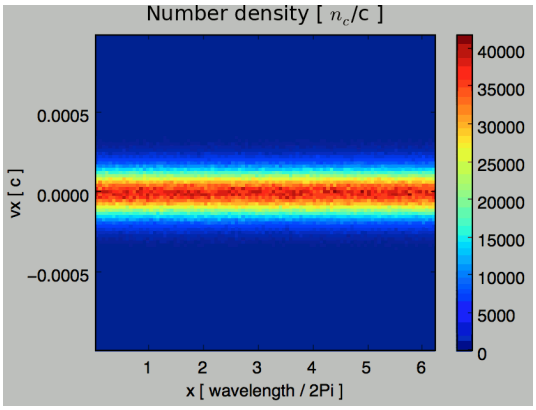

Figure 2: Phase-space of ions at $t = 0$.

6.4 Plot sections ("slices") of the array

The diagnostic #0 that we plotted in Figure 1 is the electron phase-space. Let us say we want to sum over the data that is contained between $x = 3$ and 4, and plot the result as a function of v_x. This is achieved by the argument `slice`:

```
>>> S.ParticleDiagnostic(0, timesteps=0, slice={"x":[3,4]} ).plot()
```

The result is shown in Figure 3. We can see that the peak is located at $v_x = 0.05c$, as we have already found before.

Now, let us do the slice on v_x instead of x.

```
>>> S.ParticleDiagnostic(0, timesteps=0, slice={"vx":"all"}
                         ).plot(data_min=0, data_max=11)
```

By choosing `"all"` in the argument `slice`, all the velocities v_x are sliced. In our case, as our diagnostic goes from $v_x = -0.1$ to 0.1, these limits are used.
Note that parameters `data_min` and `data_max` are used to have a nicer plot.
The result is shown in Figure 4. We obtain a constant density of $10n_c$, which is what was chosen in the input file.

6.5 Make animated plots

To have an animation of the electron phase-space with time, you have to remove the `timesteps` argument:

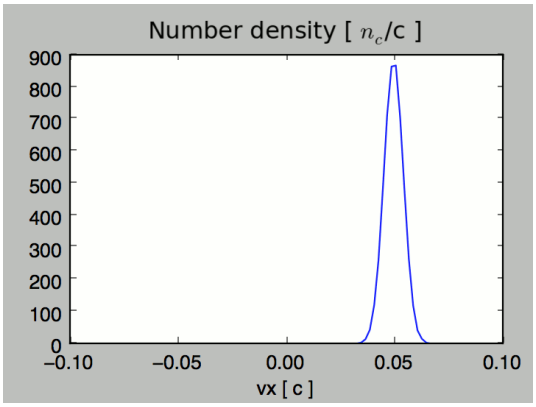

Figure 3: v_x distribution of electrons contained between $x = 3$ and 4, at $t = 0$.

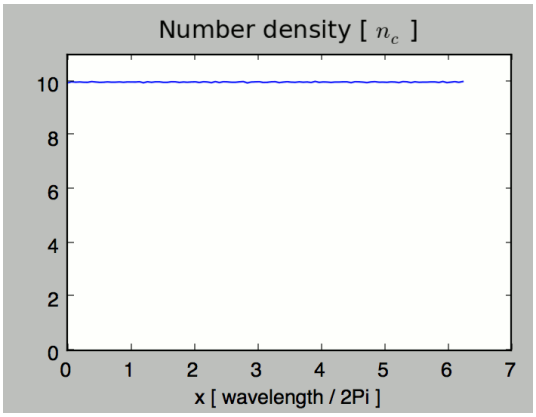

Figure 4: x distribution of electrons contained between $v_x = -0.1$ and 0.1, at $t = 0$.

```
>>> S.ParticleDiagnostic( 0 ).plot()
```

You will see the electron velocity oscillate from $0.05c$ to $-0.05c$. This is due to the fact that we are simulating a plasma wave with infinite wavelength.

Note that all the available timesteps are animated. If you want to only animate between timesteps 20 and 60, use

```
>>> S.ParticleDiagnostic( 0, timesteps=[20,60] ).plot()
```

6.6 Make multiple plots on the same figure

Use the following commands to have the animation with both electrons and ions on the same figure:

```
>>> A = S.ParticleDiagnostic( 0 )
>>> B = S.ParticleDiagnostic( 1 )
>>> multiPlot(A, B, shape=[1,2])
```

A snapshot of this double plot is given in Figure 5.

If the two plots are 1D, and are both of the same type, then they will automatically be plotted on the same axes.

```
>>> A = S.ParticleDiagnostic(0,slice={"x":"all"})
>>> B = S.ParticleDiagnostic(1,slice={"x":"all"})
>>> multiPlot(A, B)
```

This is shown in Figure 6 where you can see the two curves in blue and green.

6.7 Make a plot as a function of time

If you have sliced all the axes, then you obtain a 0-dimensional array (a scalar). In this case, the plots are automatically done as a function of time (they are not animated).
In our case, use

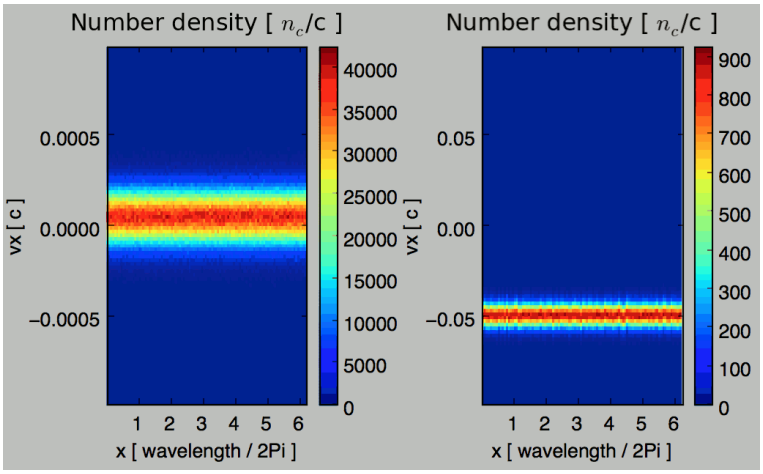

Figure 5: Two plots on the same figure.

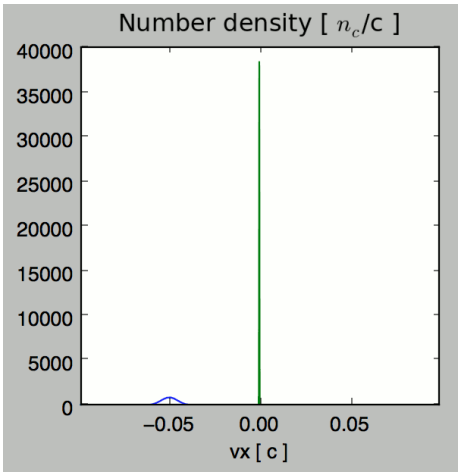

Figure 6: Two curves in the same axes.

```
>>> A=S.ParticleDiagnostic(3, slice={"ekin":"all"})
>>> B=S.ParticleDiagnostic(3, slice={"ekin":[0,0.001]})
>>> multiPlot(A,B)
```

The diagnostic that we employ here (#3) is the energy spectrum of electrons: the axis is along `ekin` which is the kinetic energy. In the first line of the code above, we are using a slice `"ekin":"all"`. Consequently, all the electrons, with all energies, will be summed, thus obtaining a scalar value equal to the total plasma density. In the second line of code, we are using `"ekin":[0,0.001]`, which means that only the electrons below 0.511 keV are considered.
Both these quantities `A` and `B` are scalars, not arrays: they will be plotted as a function of time. This is shown in Figure 7 where you can see `A` in blue and `B` in green. `A` represents all the electrons, and indeed, their density is constant. `B` represents only the slower electrons, and their number varies in time because, as we have seen before, all electrons oscillate and they do not have a constant energy. This appears on the green curve as an oscillating density.

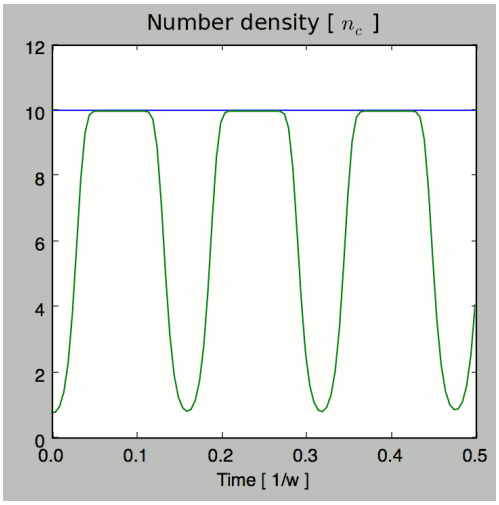

Figure 7: Blue: total density *vs* time. Green: density of slow electrons *vs* time.

6.8 Make an operation between diagnostics

Let us consider again the diagnostic #0, which is the density of electrons as a function of x an v_x. Diagnostic #2 is very similar to #0 as it has the same axes x and v_x, but it has `ouput=px_density` instead of `ouput=density`. Consequently, if we divide #2 by #0, we will obtain the average value of p_x as a function of x an v_x. To do this operation, we need to indicate `"#2/#0"` instead of the diagnostic number:

```
>>> S.ParticleDiagnostic("#2/#0").plot()
```

We obtain the plot of Figure 8, which is actually not very helpful because $\langle p_x \rangle$ varies with v_x. To have something nicer, let us slice all axes with `slice={"x":"all","vx":"all"}`.

```
>>> S.ParticleDiagnostic("#2/#0", slice={"x":"all","vx":"all"}).plot()
```

We obtain Figure 9 which nicely shows the average p_x as a function of time. This value oscillates, as we have seen previously.

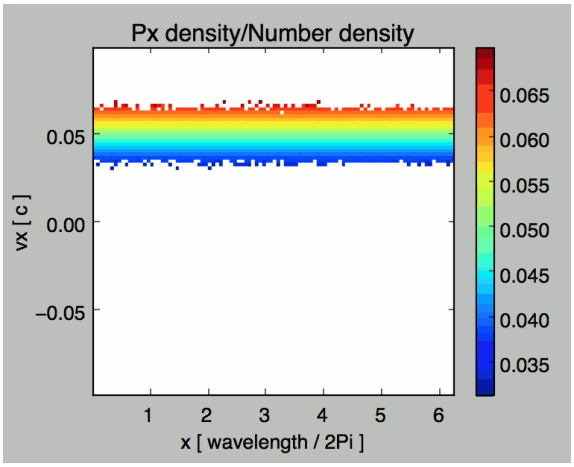

Figure 8: $\langle p_x \rangle$ as a function of x and v_x.

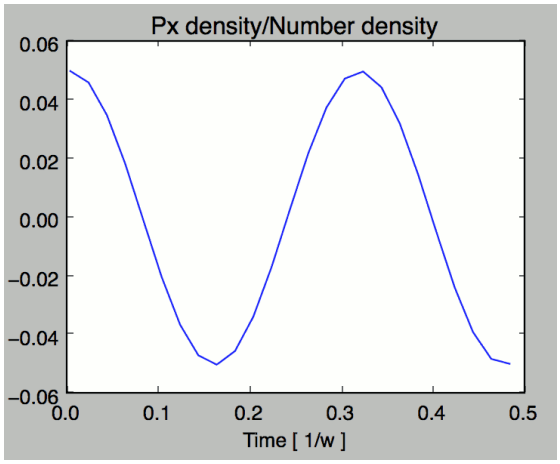

Figure 9: $\langle p_x \rangle$ as a function of time.