The particle diagnostic in SMILEI

F. Pérez

March 19, 2015

The *particle diagnostic* collects 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.

Contents

1 How to add a particle diagnostic in the input file **2**

2 Examples **4**

3 How to view and post-process a particle diagnostic **6**

4 Bonus: view fields as well **10**

5 Tutorial **11**

1 How to add a particle diagnostic in the input file

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

```
diagnostic particles
        output = output
        every = every
        time_average = time_average
        species = species1 species2 ...
        axis = type min max nsteps [logscale] [edge_inclusive]
        axis = type min max nsteps [logscale] [edge_inclusive]
        ...
end
```

- *output* must be set to `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* must be a positive integer. It is the number of time-steps between each output.

- *time_average* must be a positive integer. It is the number of time-steps during which the data is averaged before output.

- *species1 species2 ...* must be one or several species. Species are recognized by their parameter `species_type` that is included in each group `species ... end`

- `axis` is an argument that describes one axis of the grid.
  Syntax: `axis = type min max nsteps [logscale] [edge_inclusive]`
  type can be `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 `axis` arguments as wanted in one `diagnostic particles` block.

There may be as many `diagnostic particles` 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 = density, charge_density, current_density_[xyz] or p[xyz]_density
#              => 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 one or several species whose data will be used
# axis   = _type_ _min_ _max_ _nsteps_ [logscale] [edge_inclusive]
#              => _type_ can be one of the following:
#                  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
#                  outside (_min_,_max_) to be counted in the extrema bins
#   example : axis = x 0 1 30
#   example : axis = px -1 1 100
# >>>> MANY AXES CAN BE ADDED IN A SINGLE DIAGNOSTIC <<<<
```

2 Examples

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

```
diagnostic particles
        output = density
        every = 5
        time_average = 1
        species = electron1
        axis = x    0    1    30
end
```

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

```
diagnostic particles
        output = density
        every = 5
        time_average = 1
        species = electron1
        axis = x    0    1    30
        axis = y    0    1    30
end
```

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

```
diagnostic particles
        output = density
        every = 5
        time_average = 1
        species = electron1
        axis = vx    -0.1    0.1    100
end
```

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

```
diagnostic particles
        output = density
        every = 5
        time_average = 1
        species = electron1
        axis = x    0    1    30
        axis = px    -1    1    100
end
```

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.

```
diagnostic particles
        output = density
        every = 5
        time_average = 1
        species = electron1
        axis = ekin    0.02    2   100  logscale
end
```

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 ∞.

```
diagnostic particles
        output = density
        every = 5
        time_average = 1
        species = electron1
        axis = x    0    1    30
        axis = y    0    1    30
        axis = ekin    0   6   3  edge_inclusive
end
```

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

```
diagnostic particles
        output = density
        every = 5
        time_average = 1
        species = electron1
        axis = charge    -0.5   10.5   11
end
```

3 How to view and post-process a particle diagnostic

Each diagnostic produces one file like *ParticleDiagnostic0.h5*, *ParticleDiagnostic1.h5*, etc. They are numbered the same way as the order of appearance in the input file.

A python script *Diagnostics.py* is provided to view or extract data from these files. 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")`

3.1 To prepare the data

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

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

- `diagNumber` = `_int_` or `_string_` (optional)
  If not given, then a list of available diagnostics is printed.
  If `_int_` : number of the diagnostic. The first diagnostic has number 0.
  If `_string_` : operation between several diagnostics. See section 3.7.

- `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.

- `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].

- `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 array.

3.2 To obtain the data as an array:

```
ParticleDiagnostic( ... ).getData()
```

This method returns only the data array.

```
ParticleDiagnostic( ... ).get()
```

This method returns the results as a python dictionary. The dictionary is of the form `{"data":data_array, axis1:axis1_array, axis2:axis2_array, ...}` where:

- *data_array* is the final array containing the data.
- *axis1* and *axis2* are the names of the axes (for example `"x"` or `"px"`).
- *axis1_array* and *axis2_array* are the locations of the axes bins.

Example:

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

This will take the 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"]`.

3.3 To plot the data

```
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.

Example:

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

This will take the diagnostic #1, and sum the grid over all the y axis. Then it will plot the resulting array in figure 1 from 0 to 3e14.

3.4 To simultaneously plot multiple diagnostics in the same figure

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

- `diag1` = diagnostic prepared by `ParticleDiagnostic(...)`
  `diag2` = diagnostic prepared by `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:

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

3.5 Advanced plotting options

In addition to `figure`, `data_min`, `data_max`, `xmin`, `xmax`, `ymin` and `ymax`, there are many more optional arguments for the function `ParticleDiagnostic(...)`. 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 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

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

```
>>> A = ParticleDiagnostic("path/to/my/results", diagNumber=0, figure=1, cmap="gray")
```

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

3.6 Updating the plotting options

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

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

3.7 Operations between several diagnostics

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.
Example:

```
>>> A = ParticleDiagnostic("path/to/my/results", diagNumber="(#1 + #2) / #0").plot()
```

4 Bonus: view fields as well

SMILEI provides maps of the fields $\overrightarrow{E}$, $\overrightarrow{B}$, ρ (etc.) controlled by the parameter `fieldDump_every` in the input file. They are written in the file `Fields.h5`. The script `Diagnostics.py` is capable of opening and plotting these fields together with the particle diagnostics. In fact the procedure is almost the same:

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

As you can see, `Field(...)` works almost the same way as `ParticleDiagnostic(...)`, with some exceptions:

- `field` must be one of `"Bx_m"`, `"By_m"`, `"Bz_m"`, `"Ex"`, `"Ey"`, `"Ez"`, `"Jx"`, `"Jy"`, `"Jz"`, `"Jx_`*species*`"`, `"Jy_`*species*`"`, `"Jz_`*species*`"`, `"Rho"` or `"Rho_`*species*`"` where *species* is the name of one of the existing species. If you omit the argument `field`, the list of available fields will be displayed.
- `slice` can only accept three axes: `"x"`, `"y"`, `"z"`. For instance, `slice={"x":"all"}`. Note that the slice does **not** calculate the sum of the axis, but the **average**.

Please refer to the previous section to understand the other arguments. In addition, all the plotting arguments discussed in the previous section are available for the fields as well.

All the functions `plot()`, `set()`, `get()` and `getData()` are also compatible with `Field(...)`.

Last but not least: `multiPlot()` can be used to combine field and particle diagnostics on the same figure.

Example:

```
>>> A = Field("path/to/my/results", field="Ex")
>>> B = ParticleDiagnostic("path/to/my/results", diagNumber=0)
>>> multiPlot( A, B )
```

5 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`

5.1 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 ..
```

5.2 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

```
>>> ParticleDiagnostic('tst1d_6_particle_diagnostic')
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.

5.3 Plot a diagnostic result at $t = 0$

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

```
>>> ParticleDiagnostic('tst1d_6_particle_diagnostic', 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

```
>>> ParticleDiagnostic('tst1d_6_particle_diagnostic', 1, timesteps=0 ).plot()
```

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

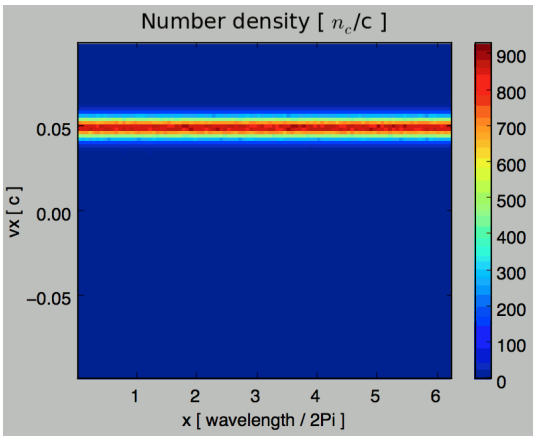

Figure 1: Phase-space of electrons at $t = 0$.

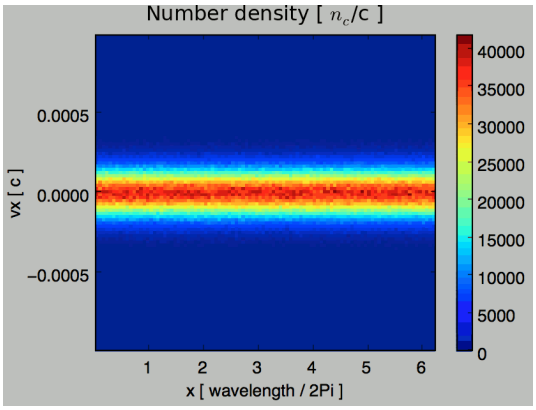

Figure 2: Phase-space of ions at $t = 0$.

5.4 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`:

```
>>> ParticleDiagnostic('tst1d_6_particle_diagnostic', 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.

```
>>> ParticleDiagnostic('tst1d_6_particle_diagnostic', 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.

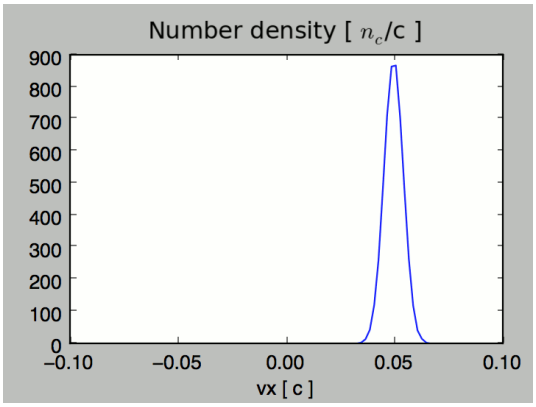

Figure 3: v_x distribution of electrons contained between $x = 3$ and 4, at $t = 0$.

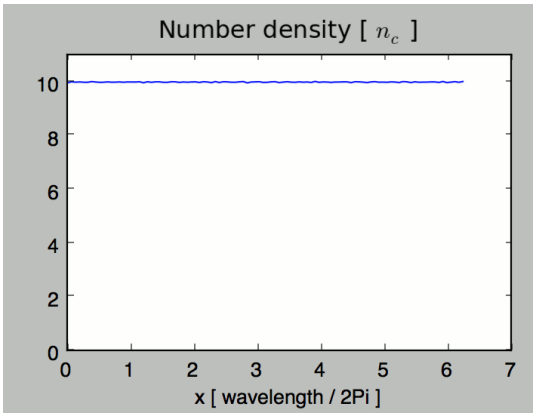

Figure 4: x distribution of electrons contained between $v_x = -0.1$ and 0.1, at $t = 0$.

5.5 Make animated plots

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

```
>>> ParticleDiagnostic('tst1d_6_particle_diagnostic', 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

```
>>> ParticleDiagnostic('tst1d_6_particle_diagnostic', 0, timesteps=[20,60] ).plot()
```

5.6 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 = ParticleDiagnostic('tst1d_6_particle_diagnostic', 0 )
>>> B = ParticleDiagnostic('tst1d_6_particle_diagnostic', 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 = ParticleDiagnostic('tst1d_6_particle_diagnostic',0,slice={"x":"all"})
>>> B = ParticleDiagnostic('tst1d_6_particle_diagnostic',1,slice={"x":"all"})
>>> multiPlot(A, B)
```

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

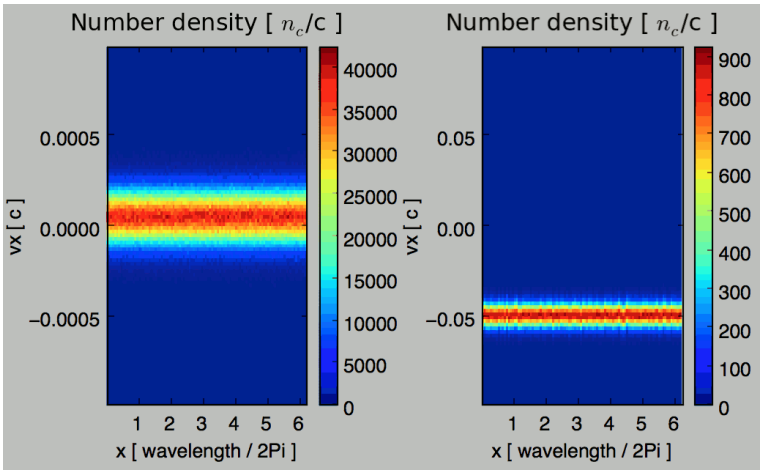

Figure 5: Two plots on the same figure.

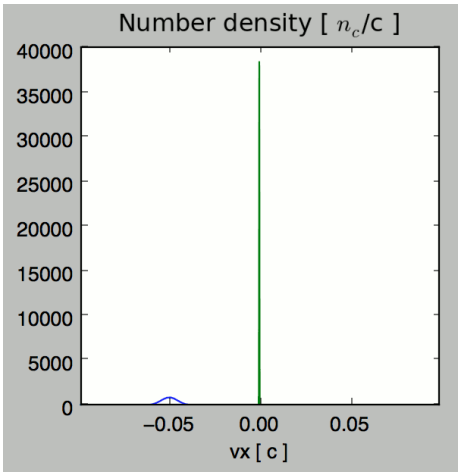

Figure 6: Two curves in the same axes.

5.7 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

```
>>> A=ParticleDiagnostic('tst1d_6_particle_diagnostic', 3, slice={"ekin":"all"})
>>> B=ParticleDiagnostic('tst1d_6_particle_diagnostic', 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.

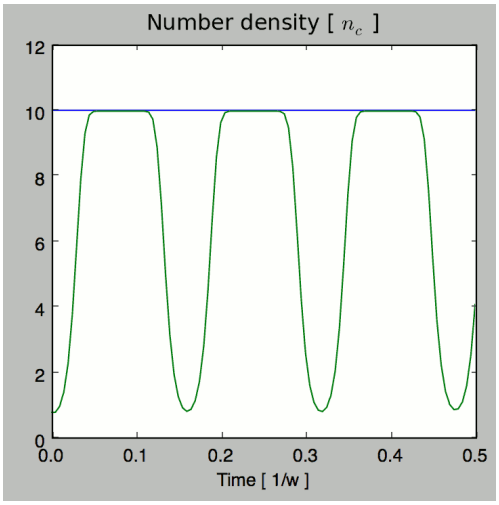

Figure 7: Blue: total density *vs* time. Green: density of slow electrons *vs* time.

5.8 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:

```
>>> ParticleDiagnostic('tst1d_6_particle_diagnostic', "#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"}`.

```
>>> ParticleDiagnostic('tst1d_6_particle_diagnostic', "#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.

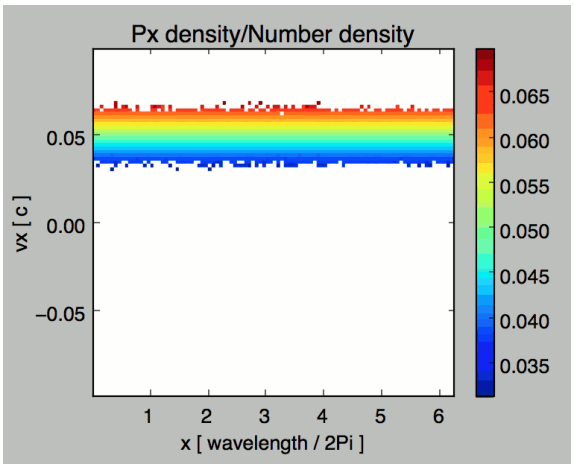

Figure 8: $\langle p_x \rangle$ as a function of x and v_x.

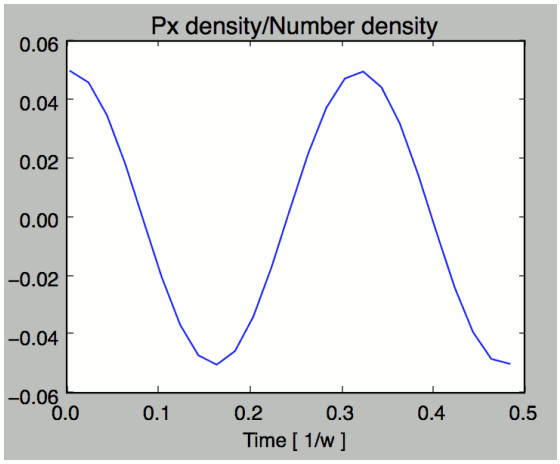

Figure 9: $\langle p_x \rangle$ as a function of time.