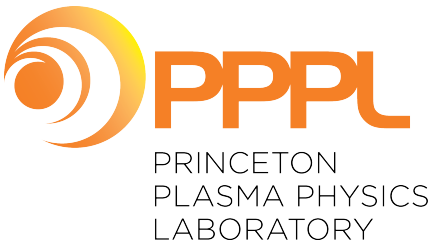
Pythonic plotting with M3D-C1

R.J.J. Mackenbach, C. Smiet, A. Kleiner



September 10, 2019

Overview

To make plotting as accessible and straightforward as possible, we have chosen an object-oriented approach to accessing data and plotting it. One can create a simulation object, which contains several subclasses. An illustrative diagram of this object can be seen in figure 1.

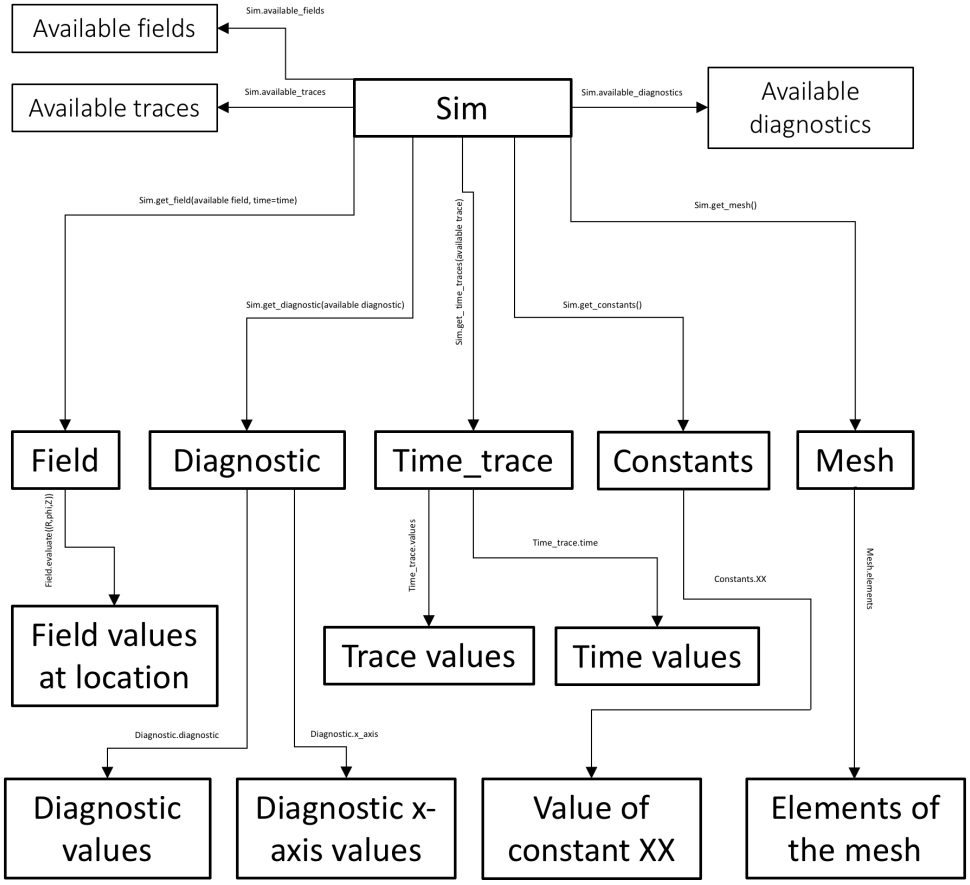


Figure 1: The main simulation object with all of its subclasses. The available fields, time traces, and diagnostics can all be accessed within the object. The constants can by found by checking all the attributes of "Constants".

Simulation objects can be created quite easily, after one imports fpy:

```
sim = fpy.sim_data('C1.h5')
```

Fields

After one has a simulation object, one can see what fields are available. This can be done by:

```
sim.available_fields
```

This returns a list of typedictionary, which shows what strings correspond to the different properties. As an example, we show how one could evaluate the current density field at timeslice zero and location $(R, \phi, Z) = (4.0, 90.0, 0.0)$.

```
import fpy
sim = fpy.sim_data('C1.h5')
j = sim.get_field('j', time=0)
values = j.evaluate((4.0, 90.0, 0.0))
```

Fields are in SI units.

Diagnostics

The available diagnostics can be seen by:

```
sim.available_diagnostics
```

There are currently three diagnostics available. We'll look at all of them. First off, `'slice times'` is the diagnostic telling what physical time each time slice corresponds to. So, for example, time_001 could correspond to $\tau_A = 10$. To plot the time slice label as a function of time, simply do

```
import matplotlib.pyplot as plt
import fpy
sim = fpy.sim_data('C1.h5')
slice_times = sim.get_diagnostic('slice times')
plt.scatter(slice_times.x_axis,slice_times.diagnostic)
plt.show()
```

Similarly, one can plot the (clock) time each time-step took as follows:

```
import matplotlib.pyplot as plt
import fpy
sim = fpy.sim_data('C1.h5')
timings = sim.get_diagnostic('timings')
for string in list(timings.diagnostic.keys()):
    plt.plot(timings.x_axis, timings.diagnostic[string])
plt.show()
```

Finally, one can plot the number of iterations per time-step as follows:

```
import matplotlib.pyplot as plt
import fpy
```

```
sim = fpy.sim_data('C1.h5')
iterations = sim.get_diagnostic('iterations')
plt.plot(iterations.x_axis, iterations.diagnostic[string])
plt.show()
```

Diagnostics are (if they have units) in M3DC1 units.

Time traces

After one has a simulation object, one can see what time traces are available. This can be done by:

```
sim.available_traces
```

Almost all of these are directly read from M3DC1, and can be plotted quite simply. For example, the toroidal plasma current can be plotted simply in the following fashion:

```
import matplotlib.pyplot as plt
import fpy
sim = fpy.sim_data('C1.h5')
Ip = sim.get_time_traces('toroidal_current_p')
plt.plot(Ip.time,Ip.values)
plt.show()
```

The exceptions here are the harmonics traces. They can be plotted similarly, but slightly different. For example:

```
import matplotlib.pyplot as plt
import fpy
sim = fpy.sim_data('C1.h5')
khmn = sim.get_time_traces('keharmonics')
plt.plot(khmn.time,khmn.values[:,0],label='n=0')
plt.plot(khmn.time,khmn.values[:,1],label='n=1')
plt.plot(khmn.time,khmn.values[:,2],label='n=2')
plt.plot(khmn.time,khmn.values[:,0],label='n=3')
plt.show()
```

Time traces are in M3DC1 units.

Constants

The contants can be obtained by:
`constants = sim.get_constants()`
As an example we'll list all possible constants, and save the major radius:

```
import matplotlib.pyplot as plt
import fpy
sim = fpy.sim_data('C1.h5')
```

```
const = sim.get_constants()
var(const)
R0 = const.R0
Constants are in SI units.
```

Mesh

This class represents the finite element mesh. The mesh can be read by `sim.get_mesh(time=time)` at a certain time slice. It returns an object with the most important attribute begin: 'elements': $n_{\text{elements}} \times 8$ list of elements in the mesh (each element is characterized by a, b, c, θ, R, Z and a number that identifies the element's relation to the mesh boundary and wall). To plot, for example, the nodes of the mesh at time zero, one could do the following:

```
import matplotlib.pyplot as plt
import fpy
sim = fpy.sim_data('C1.h5')
mesh = sim.get_mesh(time=0)
elms = mesh.elements
R = elms[:,4]
Z = elms[:,5]
plt.scatter(R,Z)
plt.show()
```

Additional routines

Some extra routines have been made available, to make plotting even easier.

unit_conv

The unit conversion makes use of a package called "pint", and a text file containing all the conversions. Pint can be found here: `https://github.com/hgrecco/pint`
We chose to make use of this module because one can write down all the correct conversion factors in some text file, which can easily be adjusted. To make use of `unit_conv` one must set the environment variable `DIM_TXT` equal to the path which points towards the text file containing the conversions.
The function is pretty straightforward to use. Simply input an array of some dimension, and state what units it is in (M3D-C1/mks). Furthermore state the dimensions of the object, (i.e. energy=2 says that the dimensions are in units of energy squared). The function will then return an array in the opposing units.

plotfpy

Plotfpy is a wrapper containing many plot functions. The following plotting options are available. Docstrings are added (and can be viewed by typing ? after the function). Nonetheless, here is a short summary:

plot_field

Plots a certain field. Similar to the plot_field IDL routine.

plot_line

Evaluates the field across a line. Similar to the plot_field, cutz=0 routine in IDL.

plot_diagnostics

Plots the diagnostics, which is the time needed per time-step, the number of iterations needed per time-step, and the time-slice label as a function of time.

plot_time_trace

Mirror IDL's plot_scalar routine. All the same strings can be called, with the addition of keharmonics and bharmonics.

Compiling and plotting

Eddy

To install and use the python plotting routines on eddy, make sure to have the module anaconda3/4.1.0 loaded before installing and compiling fio_py. After having followed the instructions fio_py, add the FIO_INSTALL_DIR/lib to your PYTHONPATH environment variable. Also add the python folder containing the plotting routines to your PYTHONPATH. Finally set the DIM_TXT environment variable equal to the path pointing towards the text file containing all unit conversion. Now we do something which one should not do normally - we will use pip to install additional packages on top of anaconda. This can create complex interdependencies, but we take this for granted over ease of use. To install the needed packages, simple do:

```
pip install pint
```

This should be sufficient for plotting. If an HDF5 error occurs, please set HDF5_DISABLE_VERSION_CHECK to one or higher. New plotting routines can simply be added to the python folder!