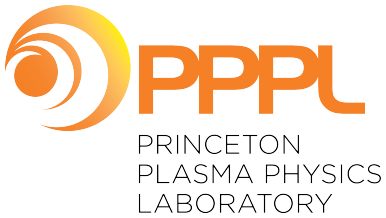
Python post-processing routines for M3D-C1

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



March 25, 2020

Contents

1 Introduction

1.1 General remarks

The routines are still in development and functionalities are added as needed.

This Python package is not supposed to replace the IDL routines. Instead it provides some functions that are also implemented in IDL, but also gives additional functionalities.

1.2 How to get help?

Apart from this document, an inline documentation is available in Python. To see informations about a certain function, type `m.function_name?`. This will display all arguments and their default values, as well as a description of this function.

For any further questions, comments and bug reports, please contact Andreas Kleiner (akleiner@pppl.gov).

2 Installation & Setup

2.1 Environment variables

The environment variable DIM_TXT should be set to $PYTHONPATH/m3dc1/M3D-C1_units.txt

2.2 Python

The Python routines are distributed as a Python package. This means it can be added to any Python environment by copying the directory 'm3dc1' found at 'unstructured/python/' in the M3DC1 repository to $PYTHONPATH. To use the routines within Python import the package using the command: `import m3dc1 as m`
Functions can then be called from the Python command line as `m.routine_name(args)`.

Note that the Python routines make use of Python packages that are not necessarily part of the standard distributions. In particular, the packages `numpy`, `scipy`, `matplotlib`, `pint` and `termcolor` are required. These can be installed either via `python3 -m pip install packagename` or `conda install packagename`.

As Python 2 is deprecated as of January 2020, and to make use of latest Python features it is recommended to run the routines in a Python 3 distribution. This can either be a normal Python installation or anaconda.

2.3 Fusion-IO

The output of M3DC1 simulations is read via the library Fusion-IO, which can be downloaded from `https://github.com/nferraro/fusion-io`. Please refer to the README file in this repository for compilation instructions. Furthermore, precompiled versions of Fusion-IO are available on portal, eddy and cori. Please contact akleiner@pppl.gov if you need more details.

3 Overview

3.1 Functions

3.1.1 Plotting of M3DC1 results:

```
plot_diagnostics()          plot_mesh()
plot_field()                plot_shape()
plot_field_basic()          plot_signal()
plot_field_mesh()           plot_time_trace()
plot_flux_average()         plot_time_trace_fast()
plot_line()                 plot_vector_field()

eigenfunction()
flux_average()
```

3.1.2 Input/output and post-processing calculations:

```
compensate_renorm()
eval_field()
flux_coordinates()
unit_conv()

readParameter()
readC1File()
readTimeFile()
```

In addition, the file `fpylib.py` contains more useful functions for processing M3DC1 results.

3.1.3 Manipulation of input data and job submission

```
extend_profile()
setup_equilibrium()
start_linear_runs()
```

3.2 Classes

```
sim_data()
Gamma_data()
Gamma_file()
```

4 Procedures

4.1 Reading and evaluating M3DC1 data

4.1.1 Using Fusion-IO (recommended)

This is the recommended and fully implemented method. 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.

Simulation objects can be created quite easily, after importing fpy:

```
sim = fpy.sim_data(time=time_slice, file_name='C1.h5')
```

This object gives access to data from the C1.h5 file and the fields at the specified time slice. Most of the plotting and other post-processing routines accept a simulation object to be passed as a argument. The recommended workflow is to first create a simulation object and then use the plotting routines with that object passed to them. For large meshes this saves time, because the M3DC1 mesh connection is calculated only once and for large meshes this can be time-consuming. However, if no simulation object is provided to any routine, it will read the simulation. This workflow is similiar to that in the IDL routines and for small meshes it is reasonably fast.

A field can be evaluated at any point within the vacuum boundary by using the function `eval_field()`.

```
eval_field(field_name, R, phi, Z, coord='scalar', sim=None,
           file_name='C1.h5', time=0)
```

Evaluates field 'field_name' at location R, ϕ, Z. If a simulation object is provided via the 'sim' argument, file_name and time are ignored and do not need to be specified. The field will then be avaluated for the simulation identified by the C1.h5 file and time slice specified when the simulation object was created. The argument 'coord' is used to specify the component of the field that shall be evaluated. For scalar field it should be 'salar', for vector fields it can be either 'R', 'phi', 'Z' or 'vector'. For 'vector' it returns all three components as a tupel. For tensor fields (used to evalute the derivatives of a vector field) 'coord' should be set to 'tensor'. It will return a tuple of length 9 containing the derivatives of all three vector field components with respect to the three variables (R, ϕ, Z).

The values that are returned by `eval_field()` are in SI units. Values can be converted using `unit_conv()`:

```
unit_conv(array, arr_dim='M3DC1', time=0, length=0, particles=0,
          magnetic_field=0, current=0, current_density=0, diffusion=0,
          energy=0, force=0, pressure=0, resistivity=0, temperature=0,
          velocity=0, voltage=0, viscosity=0, thermal_conductivity=0,
          electric_field=0)
```

Converts an array from M3DC1 units to mks or vice versa. **arr_dim denotes the units the array is in (so 'M3DC1', or 'mks')**. The routine will return a new array in the opposing units. The other parameters determine the dimensions of the array. If, for example, an array is given in units of energy**2/current_density, input energy=2 and current_density=-1.

4.1.2 Reading raw data from the HDF5 files

The file read_h5.py contains some functions to read data directly from the M3DC1 output files:

```
readParameter(pname,fname='C1.h5',listc=False)
```

Read and return a parameter 'pname' from any HDF5 file. If listc=True, the structure of the HDF file (names of all groups, datasets and attributes in root group) is printed to the screen.

```
readC1File(scalar=None,signal=None,fname='C1.h5',listc=False)
```

Read a scalar (also called time trace) or signal (diagnostic such as magnetic probes) from C1.h5 file. Returns two arrays containing the times and corresponding values respectively. If listc=True, the structure of the HDF file (names of all groups, datasets and attributes in root group) is printed to the screen.

If no scalar or signal is provided, but listc=True, this function simply shows the content of the HDF5 file.

```
readTimeFile(field=None,fname='time_000.h5',listc=False)
```

Read a field from a time slice 'time_xxx.h5' or 'equilibrium.h5'. If listc=True, the structure of the HDF file (names of all groups, datasets and attributes in root group) is printed to the screen.

If no scalar or signal is provided, but listc=True, this function simply shows the content of the HDF5 file.

Furthermore, since the evaluation of a scalar time series does not require the calculation of the mesh connectivity, routines have been implemented that read and process time traces after directly reading them from the HDF5 files. These routines are found in `time_trace_fast.py`.

```
get_timetrace(trace,file_name='C1.h5',units='m3dc1',growth=False,
              renorm=False,quiet=False)
```

Reads a time trace from the specified C1.h5 file. Returns two arrays containing the times and corresponding values respectively. If `growth=True`, the growth rate of 'trace' will be returned. If `renorm=True` spikes due to renormalization within M3DC1 will be removed. If `quiet=False`, no output will be written to the terminal.

4.2 Plot field

This section explains multiple functions to plot fields from M3DC1, either the fields itself or quantities like flux average. The most extensive routine to plot a field in a $R-Z$ plane is `plot_field()`.

```
plot_field(field, coord='scalar', row=1, sim=None, file_name='C1.h5', time=0, phi=0,
           linear=False, diff=False, tor_av=1, mesh=False, bound=False, lcfs=False,
           units='mks',res=250, cmap='viridis', cmap_midpt=None, save=False,
           savedir=None,pub=False)
```

Please type `m.plot_field?` for a full explanation of all arguments. This routine evaluates the mesh on a rectangular grid. The possibilities for `field` can be shown by `sim.available_fields`. The argument 'coord' is used to specify the component of the field that shall be evaluated. For scalar field it should be

'salar', for vector fields it can be either 'R', 'phi', 'Z' or 'vector'. For 'vector' it returns all three components as a tupel. For tensor fields (used to evalute the derivatives of a vector field) 'coord' should be set to 'tensor'. `row` is only relevant for tensor fields. `row=1` plots the first derivative of all vector field components (R, ϕ, Z) with respect to R. Similarly, `row=2` plots the vector field's derivative w.r.t. ϕ, and `row=3` the derivatives w.r.t. Z. `sim` can either be a sim_data object or a list/tupel of two such objects. `field_name` and `time` are only relevant when `sim=None`. ϕ is the toroidal angle where the field is plotted. Linear plots the difference of the field at `time=time` and the equilibrium.

```
plot_field_mesh(field, coord='scalar', file_name='C1.h5', time=0, phi=0, mesh=False,
                linear=False, diff=False, tor_av=1, bound=False, units='mks',res=0)
```

Creates a contour plot of a field similar to `plot_field()`, but evaluates the field on each mesh point of the triangular mesh (instead of a rectangular grid as in `plot_field()`). The options for plot formatting are limited as of now.

```
plot_field_basic(field, coord='scalar', file_name='C1.h5', time=0, phi=0, mesh=False,
                 linear=False, diff=False, tor_av=1, bound=False, units='mks',res=250)
```

Similar to `plot_field()`.

```
plot_flux_average(field, coord='scalar', sim=None, fcoords='', units='m3dc1',
                  file_name='C1.h5', time=0, phit=0, rms=False,pub=False)
```

Calculates and plots the flux average of a field.

```
plot_line(field, coord='scalar', angle=0, file_name='C1.h5', time=0, phi=0,
          linear=False, diff=False, tor_av=1, units='mks')
```

Plots the field along a line determined by `angle` lying inside the $R - Z$ plane.

```
plot_shape(sim=None, file_name='C1.h5', time=0, phi=0, Nlvl_in=10, Nlvl_out=1,
           mesh=False, bound=False, lcfs=False, ax=None, pub=False)
```

Plots the shape of the flux surfaces.

4.3 Linear eigenfunction and poloidal spectrum

In the linear limit the displacement $\xi \propto (p(t) - p(t = 0))$. This can be plotted via the command `plot_field('p',time='last',linear=True`. A more sophisticated analysis of the eigenfunction within the last closed flux surface is possible with the function

```
eigenfunction(sim=None,time=1,phit=0.0,file_name='C1.h5',fcoords=None,points=200,
              fourier=True,units='m3dc1',makeplot=True,nummodes=10,cmap='jet',pub=False)
```

This function calculates the eigenfunction $(\xi \propto (p(t) - p(t = 0)))$ on the magnetic surfaces in the flux coordinate system specified by `fcoords`. For tokamaks 'pest' is the natural straight field line coordinate system. But other flux coordinate systems can be used as well: Boozer coordinates, Hamda coordinates and the geometrical poloidal angle. The `eigenfunction()` returns either the eigenfunction on a flux-aligned grid or the poloidal Fourier spectrum, depending on the value of `fourier`.

`sim` should be a list of two sim_data objects. If none are specified, `time` should be set to the time slice where the eigenfunction is to be calculated, and `file_name` can be set to the path to the corresponding C1.h5 file. `phit` is the toroidal angle. For `fcoords` possibilities are: `'pest'`, `'boozer'`, `'hamada'`, for all other values the geometrical poloidal angle will be used. If `fourier=True`, the poloidal Fourier spectrum will be calculated and returned. If `makeplot=True`, the eigenfunction (and eventually the poloidal Fourier spectrum) will be plotted. `nummodes` specifies the number of dominant modes that will be labeled in the plot legend. `cmap` allows the specification of a different color map. If `pub=True`, the plots will be formatted for publication.

Note: When analysing ballooning or peeling-ballooning modes it is adviced to increase `points`. These modes are localized in the low field side, where the resolution of straight field line coordinates is low, thus requiring an increase of sampling points.

4.4 Plot mesh

The mesh is stored in object, which can be created by `elms = sim.get_mesh()`. After the creation of this object the mesh can be plotted via the function `plot_mesh()`.

```
plot_mesh(elms=None,time=0,file_name='C1.h5',boundary=False,ax=None,pub=False)
```

4.5 Plot scalar (time trace)

Unlike for fields and other type of quantities, the most efficient way to access and plot time traces (scalar quantities stored in C1.h5) is to read them directly from the HDF5 files. However, routines that are based on Fusion-IO exist and are described at the end of this section. First, routines that process and plot time traces without Fusion-IO are found in the file `time_trace_fast.py`. The most important functions are explained in the following.

```
plot_time_trace_fast(trace,units='mks',file_name='C1.h5',growth=False,renorm=False,
                     yscale='linear',rescale=False,save=False,savedir=None,pub=False)
```

Reads and plots time trace or growth ($\mathrm{d}/\mathrm{d}t \ln y$) of time trace.

```
double_plot_time_trace_fast(trace,file_name='C1.h5',renorm=False,rescale=False,
                            units='m3dc1',title=None,pub=False)
```

Plots a time trace y and its associated growth rate $\mathrm{d}/\mathrm{d}t \ln y$ in two subplots.

```
growth_rate(n=None,units='m3dc1',file_name='C1.h5',time_low_lim=500,slurm=True,
            plottrace=False)
```

This provides a better instrument to determine the linear growth rate than just plotting the growth rate with `plot_time_trace_fast`. After reading the time trace of the growth, the growth rate is averaged over a certain intervall. If the growth rate stays constant in that intervall, the averaged value is taken to be the growth rate. But should the growth rate vary considerably over time, several checks are performed to determine the quality of the result. If the algorithm is unsure about it, a prompt is opened and the user can decide how to determine the value of the growth rate. By default the average is taken over the second half of the simulation (ntimemax/2 until ntimemax). To exclude the noise in the beginning of the simulation, the first 500 Alfven times are ignored. However, for fast growing modes this value can be too

high, and it can be changed by setting `time_low_lim` to a different value. If `slurm=True`, the slurm logfile will be read and additional checks for convergence in the Grad-Shafranov solution and the final GS error are performed.

```
eval_growth_n(nmin=1,nmax=10,plotef=True,units='m3dc1')
```

Traverses subdirectories named 'nXX' where XX is the toroidal mode number, evaluated linear growth rates and writes the results into a text file among with other parameters related to the equilibrium.

```
plot_gamma_n(nmin=1,nmax=10,units='m3dc1',fignum=None,c=None,ls=None,lbl=None,
             slurm=True,plottrace=False)
```

Similar to `eval_growth_n`, but does not write the results to a file. If a file with results already exists, it can read this file, or evaluate the growth rates again.

4.6 Plot signals

Signals (from diagnostric probes such as magnetic probes and flux loops) are directly read from the HDF5 file.

```
plot_signal(signal='mag_probes', file_name='C1.h5', renorm=False, scale=False,
            deriv=False, pspec=False, units='mks', time_low_lim=500, pub=False)
```

4.7 Plot code diagnostics (timings)

```
plot_diagnostics(file_name='C1.h5',units='mks',fignum=None)
```

This function plots three diagnostics:

1. The time needed per timestep
2. The number of iterations needed per timestep
3. The temporal values of each time-slice

4.8 Calculate flux coordinates

```
flux_coordinates(sim=None, file_name='C1.h5', time=0, fcoords='', phit=0.0, points=200,
                 fbins=None, tbins=None, itor=None, r0=None, psin_range=None,
                 njac=False, makeplot=False,fignum=501)
```

4.9 Extend profiles

The p-file of kinetic EFITs contain the plasma profiles up to $\psi_N = 1$. After the profiles where extracted using `extract_profiles.sh`, the following function extends the profiles beyond the lastr closed flux surface.

With the default parameters the profile and its first derivative are continuous across the last closed flux surface and everywhere else.

```
extend_profile(file_name,psimax=1.05,fitrange=None,minval=None,match=True,smooth=0,
               weighted=True,suffix='.extended')
```

A tanh function will be fitted to the pedestal defined by `fitrange=[psistart,psimax]`. This resulting fitted tanh function is used to extend the profile beyond the last point given in the p-file up to the value given by `psimax`. With the default parameters (`match=True,smooth=0`), the height and width of the initially fitted tanh will be adjusted such that the profile and its first derivative are continuous across the point where it is extended. The argument `weighted` determines whether the data points that enter the fit are weighted by $1/y$, where $y(\psi_N)$ is the profile. This typically gives more accurate fit results and thus `weighted` should be set to `True`. If `match=False` and `smooth` has a value of larger than 0, the profile is extended in the same way as in the IDL routines. Note that the curve_fit function of scipy differs from the IDL curvefit function and thus the fit parameters differ slightly.

5 Interactively read and plot data with fpy.py

This section describes how to directly use the classes and methods of fpy.py to interactively read M3DC1 data via Fusio-IO and plot the results without using the function described in the sections above.

5.1 Fields

After creating a simulation object, the available fields can be displayed by:

```
sim.available_fields
```

This returns dictionary, which shows the strings that are used to label the different fields. As an example, we show how one would evaluate the current density field at time slice zero and location $(R, \phi, Z) = (4.0, 90.0, 0.0)$.

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

Fields are in SI units by default.

5.2 Diagnostics

The available diagnostics can be seen by:

```
sim.available_diagnostics
```

There are currently three diagnostics available. We will 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.

5.3 Time traces (Scalars)

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 by default.

5.4 Constants

The constants can be obtained by

```
constants = sim.get\_constants()
```

As an example we will 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.

5.5 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()
```