# Standardised Code for EPOCH Python Analysis

Tobias Slade-Harajda

November 7, 2024

# Contents



# List of Code

# 1 Introduction

Welcome to the Standardised Code for EPOCH Python Analysis (SCEPA). Coincidentally, this shares an acroynm with the EU's: Scaling up the Energy Poverty Approach (SCEPA) - which I thought both linked and fitting. This was first compiled by previous Ph.D. students such as Omstavan Samant, James Cook, Ben Chapman-Oplopoiou (previously Ben Chapman), Leopoldo Carbajal, Bernard Reman and later adapted and collated by myself, Tobias Slade-Harajda.

The purpose of this collation is to provide a "one-stop-shop" for users and developers of simulations regarding the development of MCI, analysis of ICE and general plotting/correlation/coherence scripts for EPOCH 1D3V simulations.

The most important file one can use upon simulation first completion is the `batch_load` script, which takes in the key argument of simulation location, as a string, and nothing else. It's use, structure and output is discussed in Sec. 3, but examples laid out in Sec. 5 are more niché in nature, and define how one might use some of the other functions in `list_new`.

I began the creation of `list.py` and the final `list_new.py` that we see today in 2021. It has been expanded on many times, with the addition, removal and rearrangement of certain functions depending on whether they are more useful elsewhere. The re-modelling of the code's structure and commital to git began seriously in mid-2023, after my attendance to a data management and safety course in Manchester, which primarily taught me about using git, GitHub and version control. Readily equipped with this new knowledge, I created the simple modular structure of the code, and began improving its efficiency, readability and expanding on definitions of functions and processed by writing more comments (the latter of which is still somewhat lacking in places).

# 2 list_new.py

Named after the initial naïve naming convention employed by myself in 2021 at the start of my Ph.D., `list_new.py` is the updated version of the `list.py` file which was itself an expansion of the `functions` and `functions_basic` python files passed down to me through previous generations of Ph.D. students. Within these aptly named `.py` files we find a collection of basic functions one can use to study the outputs of EPOCH simulations. This included taking and plotting of the Fourier transforms (in 1&2d); plotting a quantity in 1d versus it's position; calculating the power spectra of the 2d FFT; and

plotting the energies through time for the user-defined majority and minority species.

Listed below are some of the most standard and useful functions found in `list_new`. The total number of lines in `list_new` as of writing exceeds 2700. Therefore, discovery of functions which are useful will naturally occur as one uses it. I understand that importing of this file with all of these functions is inefficient, and it would likely be better to further split this file into multiple sub-files each pertaining to a different aspect of analysis. However, this is not my main focus as of writing, so I have left it as an exercise to the reader. If you find that `list_new` is indeed shorter and more efficient, then ignore this last sentence as I have likely come around to it myself.

For now, however, the general functioning of the SCEPA code can be found below in these subsection summaries.

## 2.1 `getSimulation()`

A good starting function for any simulation, `getSimulation(DIR_PATH)` takes the positional string of the simulation directory and navigates to it using `os.chdir(DIR_PATH)`. It returns the `cwd` of the `DIR_PATH` once it has navigated to the simulation directory, therefore, it is commonplace in my code to name this as the variable

```
sim_loc = getSimulation(DIR_PATH)
```

This `sim_loc` is later used for functions to scan the directory, such as `list_sdf`, `energies` and `ciggies`.

## 2.2 `list_sdf()`

The total number of time-steps can be determined via the `list_sdf(SIM_LOC)` function, which takes in the string of the file location and returns an array of integers correlating to the total number of `.sdf` files within the directory, when and only when “TempOut” is not in the same string. This was specified, because I separately output input initialisation parameters from the EPOCH simulation within another type of `.sdf` file named “TempOut”.

The list of `sdf` files returned is typically expressed throughout the code as

```
ind_lst = list_sdf(SIM_LOC)
```

or some variance of this.

## 2.3 sdfread()

The reading of an sdf file is the primary goal of any Python analysis of EPOCH. The sdfutils and sdf packages must first be made within the downloaded epochNd folder, allow for the loading of the package via import sdf. SDF files are then read into Python as an object, with dictionary, Headers, and body, via sdf.read(n). This is done once the makefile has compiled, and is completed simply by typing

```
>make sdfutils
```

To load a .sdf file, we must pass a string of the file name, i.e. 00000.sdf. This makes the loading of .sdf files somewhat troublesome and inefficient, as the length of this string (also called n-zeros when ignoring ".sdf") varies depending on the length of number of time-steps EPOCH outputs. This means that each time you wish to read a file you have to pass the correctly formatted string. The function sdfread() takes any number, converts it into a n-zero length string and appends ".sdf" onto the end so that the sdf package can read the file correctly.

## 2.4 getKeys()

Determining the readable outputs from a given sdf file is vital for any analysis of an EPOCH simulation. One such method of doing this is to load the $n^{th}$ sdf file as an object, then from the dictionary one can list all the available outputs. This is summarised in the function getKeys(d) which takes the argument d, which is an sdfread(n) object.

To list all of the available keys for a given file, it is easy to write

```
for k in getKeys(sdfread(0)): print(k)
```

# 3 batch_load.py

Loading a simulation can require a large number of files to be open at once, i.e. greater than the computational limit set by the machine you're using (e.g. at CFSA the limit on those machines for files open is 1024). Therefore it is important that when loading in series, as opposed to in parallel, the maximum number of files loaded at any one time is less than this limit. This doesn't necessarily need to be a user-defined error check, as the machines should

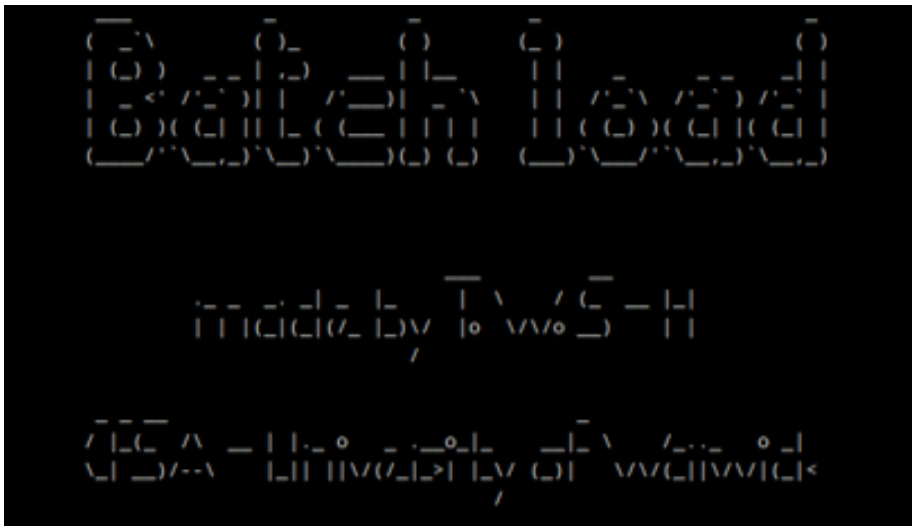

Figure 1: Batch load welcome message

flag when this has happened, but it is useful to have this in place before running to avoid memory overflow or crashes.

This is the principal behind batch saving and loading, as the arrays they represent are less than the memory of the machine (typically - this may become troublesome if you're running EPOCH locally, but given the options surrounding running EPOCH on HPCs, I'd be surprised if you encounter this [1]) but the number of files they span is larger than the machine's limit. It is imperative in the batch saving and loading process that the convention of file labelling and saving is consistent and concise.

The general process of the `batch_load` file, which takes in **only** the argument of simulation location as a string, is to load all of the fieldmatrix components ($B_z$, $E_y$, $E_x$ etc.) and energies of the particles through time and space. Once it has done this, it then moves onto processing said data via FFTs, cigarette plots, change in energy densities, growth rates, correlations, coherence and any other method the user specifies. `Batch_load` determines whether there are pre-made files to load before committing to their making.

Treating the total simulation as a class object helps to save on memory, as trying to conduct these operations all in one Python file without saving the Simulation class can lead to an out of memory error. This functionality is under-utilised in this project, as one could make smaller classes of many simulations to then compare in analysis. For this purpose however, it is sometimes easier to simply make a new .py file, loop through the simulations one wants to analyse and to then carry out the necessary analysis one wishes

[1] here

to perform. See for instance the examples in Sec. 5.

# 4 func_load.py

Loading all of the functions from `list_new` and packages relevant to further analysis, is handled in the `func_load` file, which is called at the start of every .py file before analysis takes place. This removes the need to import most packages, with exceptions for niché packages used for plotting or analysis.

It displays a message to the user that the general and supplementary/additional packages are loaded along with `list_new`.

Core packages required include:

- `numpy`
- `matplotlib.pyplot`
- `pickle`
- `sdf` (from EPOCH)

Additional packages include:

- `scipy`
- `multiprocessing` (for parallelisation)
- `matplotlib.path`

# 5 Examples

## 5.1 Field loading

Basic field loading and plotting as a field matrix in (T,X) space. In this example we loop through each simulation file in a given directory. The file structure for this will look something like

```
DIR_PATH
├── sims[0]
├── sims[1]
└── sims[2]
    ├── 0000.sdf
    └── 0001.sdf
```

```
├── 0002.sdf
├── ⋮
└── XXXX.sdf
```

We then sort the simulations (based off of your own naming convention), and get the extent of the image in terms of length (L) and end time (T). Notice in List. 1 that the loaded field-matrix is natively in the shape (T,X), but by using the **kwargs feature in `plotMatrix()` this sets the origin in `imshow()` to the lower left. We then change back to the parents directory (DIR_PATH in this example) and repeat the process for as many simulations as are found.

Example of how this data can be used and plotted is in Fig. 2, which uses the field matrix data in a three panelled plot.

```
# import all functions
from func_load import *
# change to directory where all sims are located
os.chdir(DIR_PATH)
# find sims in directory
sims = np.sort([i for i in os.listdir() if CRITERIA_STR in i])
# define the quantity you want to load as a T,X matrix
QUANTITY = 'Magnetic_Field_Bz'
# loop through simulations
for sim in sims:
    simloc = getSimulation(sim)
    field = load_batch_fieldmatrix([],QUANTITY)
    T = read_pkl('times')[-1]
    L = getGridlen(sdfread(0))
    plotMatrix(field,name=QUANTITY,extents=[0,L,0,T])
    # chdir back to parent dir
    os.chdir(DIR_PATH)
```

Listing 1: Field loading and plotting for multiple simulations

## 5.2 Change in energy densities

Here the energies of all of the particles within a given simulation are found, turned into particle densities, the difference is found (to show growth of energy) and then plotted through time.

The units of change in particle energy density ($\Delta u$) are $Jm^{-3}$, and as such the change in the field energy densities can also be plotted on the same axis.

For our first example, given in List. 2, we plot all of the energy densities of the fields **and** particles.

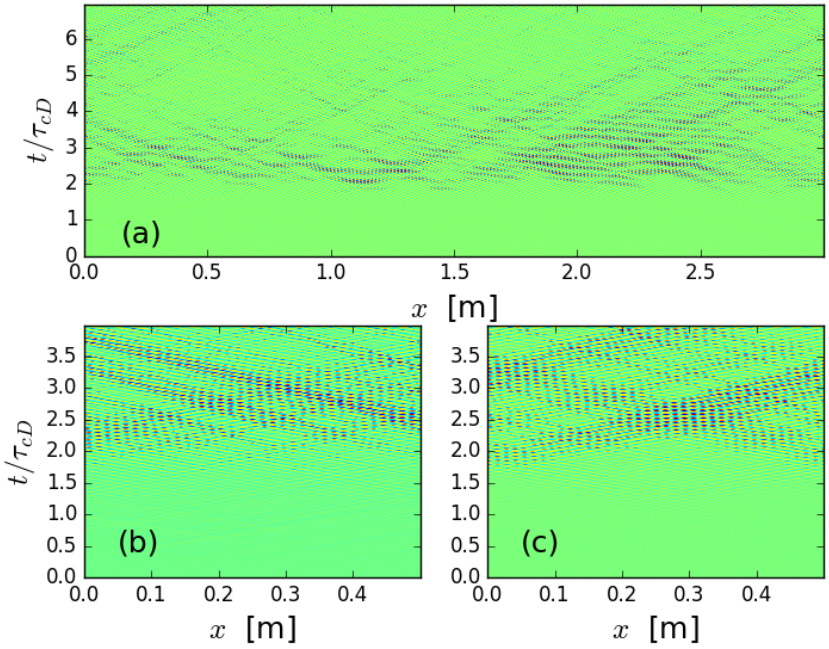


Figure 2: Example of the $B_z$ field component plotted through time and space.

```
# import all functions
from func_load import *
# chdir to sim
simloc = getSimulation(SIM_DIR)
d0 = sdfread(0)
# load energies of all species and fields
energy_integral = energies(simloc)
# returns the integrand of all energy traces (should equal or be close to 0)
```

Listing 2: Energy densities for particles and fields

For a more comprehensive delve into just particle energies, we can choose to load them ourselves per time-step and for only the given particle species, i.e. List. 3. And for an example of what the energies function outputs, see Fig. 3.

```
# import all functions
from func_load import *
# chdir to sim
simloc = getSimulation(SIM_DIR)
ind_lst = list_sdf(simloc)
d0 = sdfread(0)
# get particle species
species = getAllIons(d0)
# get just ions, no electrons
ionspecies = getIonSpecies(d0)
# fraction of files to use, 1 uses every file,
#   2 uses every other file etc.
frac = 1
n = len(ind_lst)//frac
fieldquant = species
energy_quant = [i+'_KEdens' for i in species]
# loop through each species
for i in range(len(species)):
        energy_data_mat, energy_data = getEnergies(energy_quant[i],\
                                        fieldquant[i],n,dump=True)
```

Listing 3: Energy densities for particles

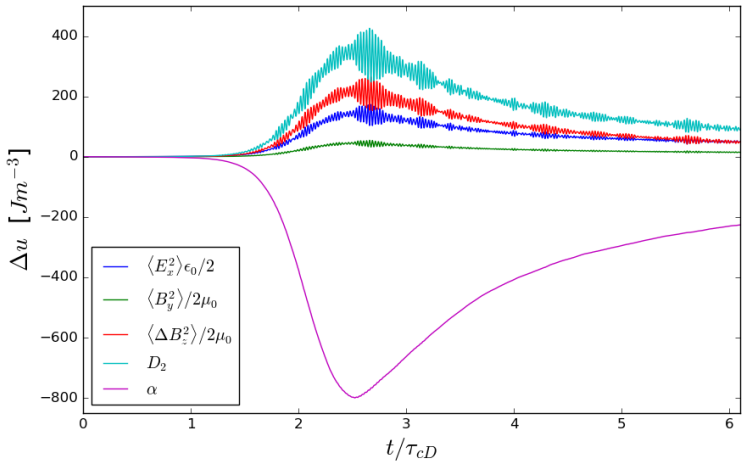


Figure 3: Energy densities for a 0% tritium JET26148-*like* simulation.

## 5.3 Correlation

List. 4 is the same example as shown on my github README. This finds the cross-correlation between all of the field components with one-another avoiding auto cross-correlation (with itself).

This method of spatial cross-correlation is summarised in L. Carbajal, 2014[2], specifically, Fig. 5.

```
# import all functions
from func_load import *
# import spatial correlation functions
import correlation.spatial_crosscor as sc
# instanstiate simulation location
sim = getSimulation(SIM_FILE_PATH)
# load array of times
times = read_pkl('times')
# list all available fields
fields = getFields()
# loop through all fields without repeating duplicates
for i in range(len(fields)):
    for j in range(i,len(fields)):
        F1 = fields[i]
        F2 = fields[j]
        _,_,norm1=Endict(F1)
        _,_,norm2=Endict(F2)
        dphase,Rtdx = sc.getSpatialCorrelation(d0=sdfread(0),F1=F1,F2=F2,\
                            norm=[norm1,norm2],plot=False)
        sc.plotSpatialCorrelation(F1,F2,Rtdx,dphase,times,MINSPECIES)
```

Listing 4: Spatial correlation between multiple fields for a given simulation

## 5.4 Cigarette plots

Cigarette plots (named after their appearance in results for $D-T-\alpha$ plasmas) were introduced by myself in 2023 as a method of observing the spreading of ion species energy distribution through time. Similar to Fig. (7) shown in L. Carbajal's, 2014 paper[3], these plot it on a log-scale with continuous colour-scale so as to reveal finer details of the energy transfer.

[2] https://pubs.aip.org/aip/pop/article-abstract/21/1/012106/107426/Linear-and-nonlinear-physics-of-the

[3] https://pubs.aip.org/aip/pop/article-abstract/21/1/012106/107426/Linear-and-nonlinear-physics-of-the

In the example shown in List. 5, the `vload` and `eload` flags determine whether we plot the velocity or energy distributions. By default this function dumps the matrix files (in T,X), plots and saves the cigarette plots for all of the species given.

Exchanging `vload` and `eload` gives the plot shown in Fig. 4.

```
# import all functions
from func_load import *
# load sim
sim_loc = getSimulation(SIM_DIR_PATH)
d0 = sdfread(0)
# get species
species = getIonSpecies(d0)
# get cigarette plot and save fig
ciggies(sim_loc, species, vload=False, eload=True)
```

Listing 5: Cigarette plots for multiple species of a simulation

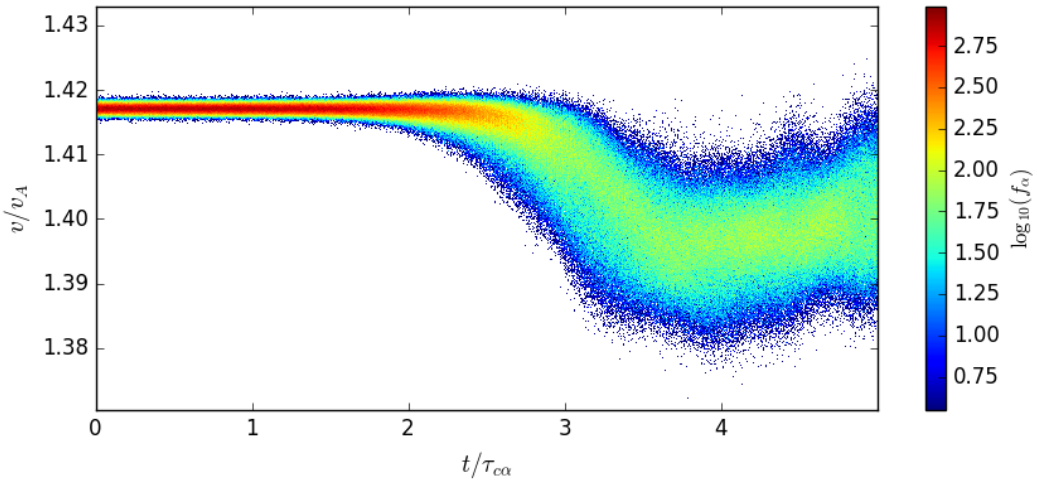


Figure 4: Cigarette plot of the 3.5MeV alpha-particles in a thermal DT(11%) plasma in velocity-time space. Ordinate and abscissa normalised to the Alfvén velocity and alpha cyclotron period respectively.