UNIVERSITY of York

BEAM

Broad Epoch Analysis Modules

Joel Adams, Peter Hill, Liam Pattison,
Shaun Doherty, Chris Ridgers

Modules Overview

- sdf-xarray: Processing .sdf files and converting them to xarray datasets
- epydeck: Parsing input (.deck) files and modifying parameters
- epyscan: Sample parameter space and create campaigns utilising epydeck
- epyrunner: Automate workflow and generate surrogate models

Introduction

- **SDF File Handling**: Loads and works with EPOCH-generated .sdf files as xarray datasets, enabling efficient data manipulation.
- **Xarray Ecosystem**: If the user is familiar with xarray then they will be able to interact with this easily
- **Lazy Loading & Performance**: Uses xarray's lazy loading to avoid excessive memory usage, with Dask integration for parallel and out-of-core computation.
- **Flexible Data Conversion**: Easily converts data to pandas or NumPy formats for further analysis.
- **Built-in Visualisation**: Supports quick and easy plotting with matplotlib, handling high-dimensional data effectively.
- **Documentation/Support**: https://sdf-xarray.readthedocs.io/en/latest/

Installation


```
pip install sdf-xarray
```

https://sdf-xarray.readthedocs.io/en/latest/key_functionality.html

Usage

Loading a single SDF file

```python
import xarray as xr
ds = xr.open_dataset("simluation/0000.sdf")
```

Loading multiple SDF files

```python
import xarray as xr
from sdf_xarray import SDFPreprocess
ds = xr.open_mfdataset("simulation/*.sdf", preprocess=SDFPreprocess())
```

▼ Coordinates:

X_Grid	(X_Grid)	float64	-1e-05 -9.98e-06 ... 2e-05
X_Grid_mid	(X_Grid_mid)	float64	-9.99e-06 -9.971e-06 ... 1.999e-05
long_name :	X		
units :	m		
point_data :	False		
full_name :	Grid/Grid_mid		
time	(time)	float64	2.606e-17 5.003e-15 ... 2e-13

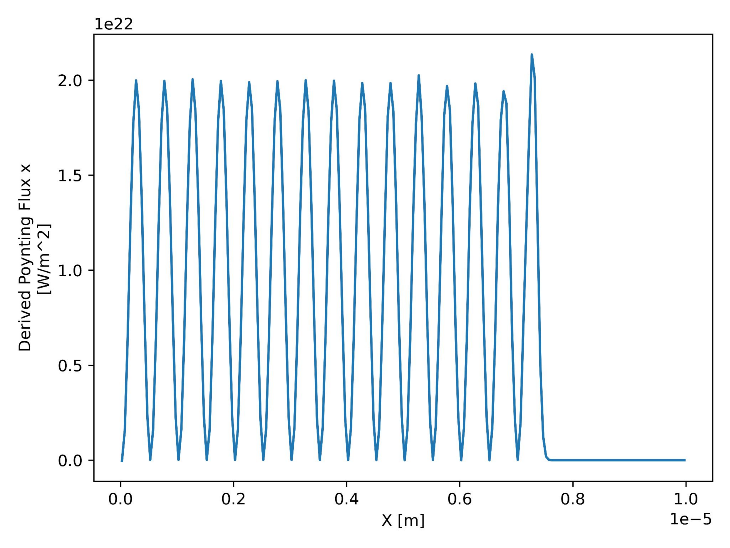
▼ Data variables:

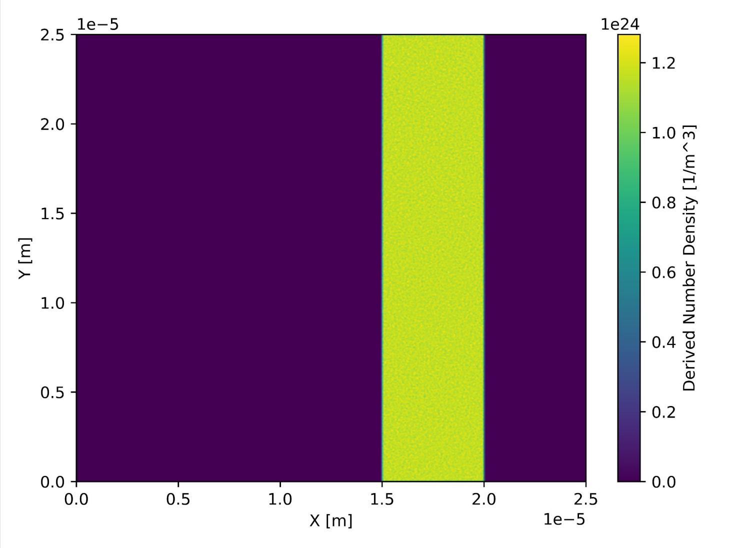
Wall_time	(time)	float64	0.4287 4.708 6.57 ... 316.7 317.6
Time_increment	(time)	float64	nan nan nan ... nan nan 5.212e-17
Plasma_frequency...	(time)	float64	nan nan nan ... nan nan 4.209e-16
Minimum_grid_po...	(time)	float64	nan nan nan ... nan nan -9.99e-06
laser_x_min_phase	(time, dim_laser_x_min_phase_0)	float64	dask.array<chunksize=(1, 1), meta=np.nd...
time_prev_normal	(time)	float64	nan nan nan nan ... nan nan 2e-13
walltime_prev_nor...	(time)	float64	nan nan nan nan ... nan nan nan 0.0
nstep_prev_normal	(time)	float64	nan nan nan nan ... nan nan nan 0.0
Particles_Particles...	(time)	float64	nan nan nan nan ... nan nan 64.0
Particles_Particles...	(time)	float64	nan nan nan nan ... nan nan 64.0
Particles_Particles...	(time)	float64	nan nan nan nan ... nan nan -1.0

Plotting

```
ds["Derived_Poynting_Flux_x"].plot()
```

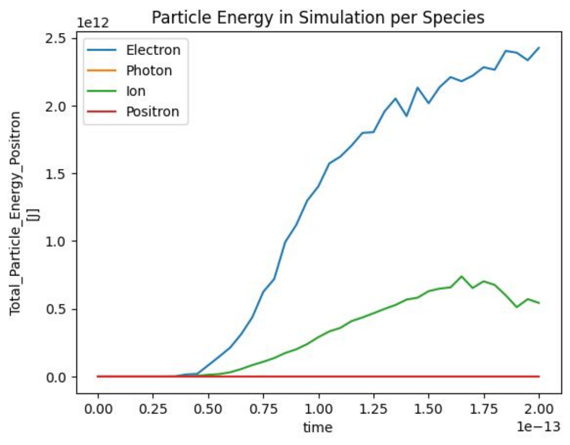
```
ds["Derived_Number_Density"].plot(x="X_Grid_mid", y="Y_Grid_mid")
```



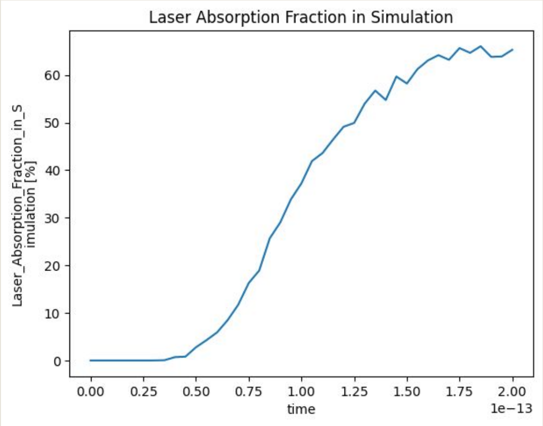


Advanced Plotting

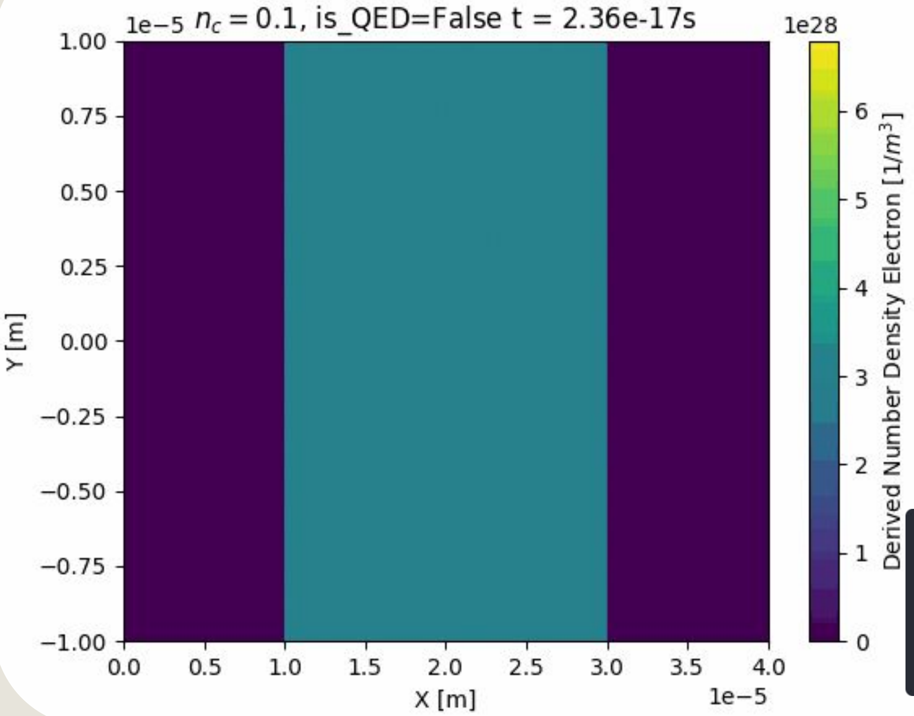
```python
ds["Total_Particle_Energy_Electron"].plot(label="Electron")
ds["Total_Particle_Energy_Photon"].plot(label="Photon")
ds["Total_Particle_Energy_Ion"].plot(label="Ion")
ds["Total_Particle_Energy_Positron"].plot(label="Positron")
plt.legend()
plt.title("Particle Energy in Simulation per Species")
plt.show()
```




```python
ds["Laser_Absorption_Fraction_in_Simulation"] = (
    ds["Total_Particle_Energy_in_Simulation"]
    /
    ds["Absorption_Total_Laser_Energy_Injected"]
) * 100
# We can also manipulate the units and other attributes
ds["Laser_Absorption_Fraction_in_Simulation"].attrs["units"] = "%"

ds["Laser_Absorption_Fraction_in_Simulation"].plot()
plt.title("Laser Absorption Fraction in Simulation")
plt.show()
```



Animations

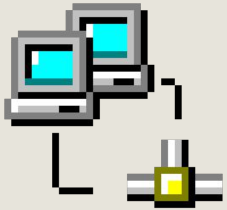



```python
ds["Derived_Number_Density"].epoch.animate()
```

Using EPOCH on Viking

Connecting to Viking

```
bmp535@login1:~
Last login: Tue Mar 25 10:09:59 on ttys025
➜  ~ ssh bmp535@viking.york.ac.uk
(bmp535@viking.york.ac.uk) Password:
Last login: Thu Mar 20 15:07:46 2025 from 10.241.38.45
  'o`
 'ooo`                                        Welcome to viking2
 `oooo`
  `oooo`          'o`                        Flight Direct r2024.1
    `ooooo`  `ooooo                        Based on Rocky Linux 8.9
       `oooo:oooo`
          `v  -[ alces flight ]-

TIPS:

'flight help'             - get help on available commands
'flight set'              - change login defaults (see 'flight info' for details)
'flight env'              - manage software package ecosystems
'flight desktop'          - manage interactive GUI desktop sessions

[bmp535@login1[viking2] ~]$
```



ssh username@viking.york.ac.uk

Epoch Viking Setup

See here for example jobscript:
https://github.com/PlasmaFAIR/SPLICE-docs/wiki/Running-EPOCH#on-viking

1. `cd scratch`
2. `git clone --recursive https://github.com/Warwick-Plasma/epoch.git`
3. `cd epoch/epoch2d`
4. `module load OpenMPI/4.1.6-GCC-13.2.0`
5. `make COMPILER=gfortran --debug -j4`

Viking do's and don'ts

How 2 run code on Viking

- ❌ Run code directly in the terminal
- ❌ Install packages from the internet using sudo apt install <package_name>
- ✔️ Use module load <package_name> to load packages
- ✔️ Utilise jobscripts to execute jobs
- ❌ Create jobscripts with a lot of resources (we only have a limited amount of compute power so be respectful of others)

Viking utilises something called jobscripts to run code via a process called Slurm

Viking Jobscripts

University of York

```bash
#!/usr/bin/env bash

#SBATCH --job-name=epoch_test ← Job name
#SBATCH --ntasks=96 ← The number of CPUS
#SBATCH --partition=teach ← For today use this partition but in the future use "nodes"
#SBATCH --time=02-00:00:00 # Time limit (DD-HH:MM:SS)
#SBATCH --account=pet-pic-2022 ← You'll probably all be using this one
#SBATCH --output=%x-%j.log
#SBATCH --mail-user=user.email@york.ac.uk ← Replace with your email address
#SBATCH --mail-type=ALL # Mail events (NONE, BEGIN, END, FAIL, ALL) ← These can be turned off and are not
                                                                      required

# Abort if any command fails
set -e

# Purge any previously loaded modules
module purge

# Load modules
module load OpenMPI/3.1.3-GCC-8.2.0-2.31.1 ← Loads the OpenMPI module required for EPOCH

# Prestart
cd epoch/epoch2d
echo Working directory: `pwd`
echo Running job on host:
echo -e '\t'`hostname` at `date`'\n'

# Script initialisation
mpiexec -n ${SLURM_NTASKS} ~/scratch/epoch/epoch2d/bin/epoch2d <<< ~/scratch/epoch/epoch2d/test1

# Job completed
echo '\n'Job completed at `date`
```

See here for example jobscript: https://github.com/PlasmaFAIR/SPLICE-docs/wiki/Running-EPOCH#on-viking

Has my job finished?

Viking has a few approaches to checking if jobs are done

- Get an email notification from the completed/failed job
- Check if the right number of sdf files have been created (bad idea as the last one can sometimes take a while)
- Poll Viking to check if the job has completed (see below)

```
[bmp535@login1[viking2] ~]$ squeue -u $USER
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
```

Try putting the word watch in front of this command, it should then run the squeue command every 2 seconds by default

Introduction: epydeck & epyscan

What are they?

- **epydeck**: A package that parses input.deck files allowing for both reading and writing
- **epyscan**: Parameter space (Linear and Log space support), sampling (gird or Latin Hypercube) and create campaign directory structure and then write input files using epydeck

Why use them?

- **Ease of use**: Simple interface with little to no Python experience required to use them
- **Automation**: deck parameters can be automated with Python instead of manual changing
- **epydeck Documentation/Support**: https://github.com/PlasmaFAIR/epydeck
- **epyscan Documentation/Support:** https://github.com/PlasmaFAIR/epyscan

Usage: epydeck

- Creates nested dicts if there are blocks with same name (e.g. species) and creates a nested dick using the name parameter
- Creates a list if multiple definitions of the same variable exist (e.g., number_density)
- Does not support mathematical or unit parsing
- Does not retain comments from template file

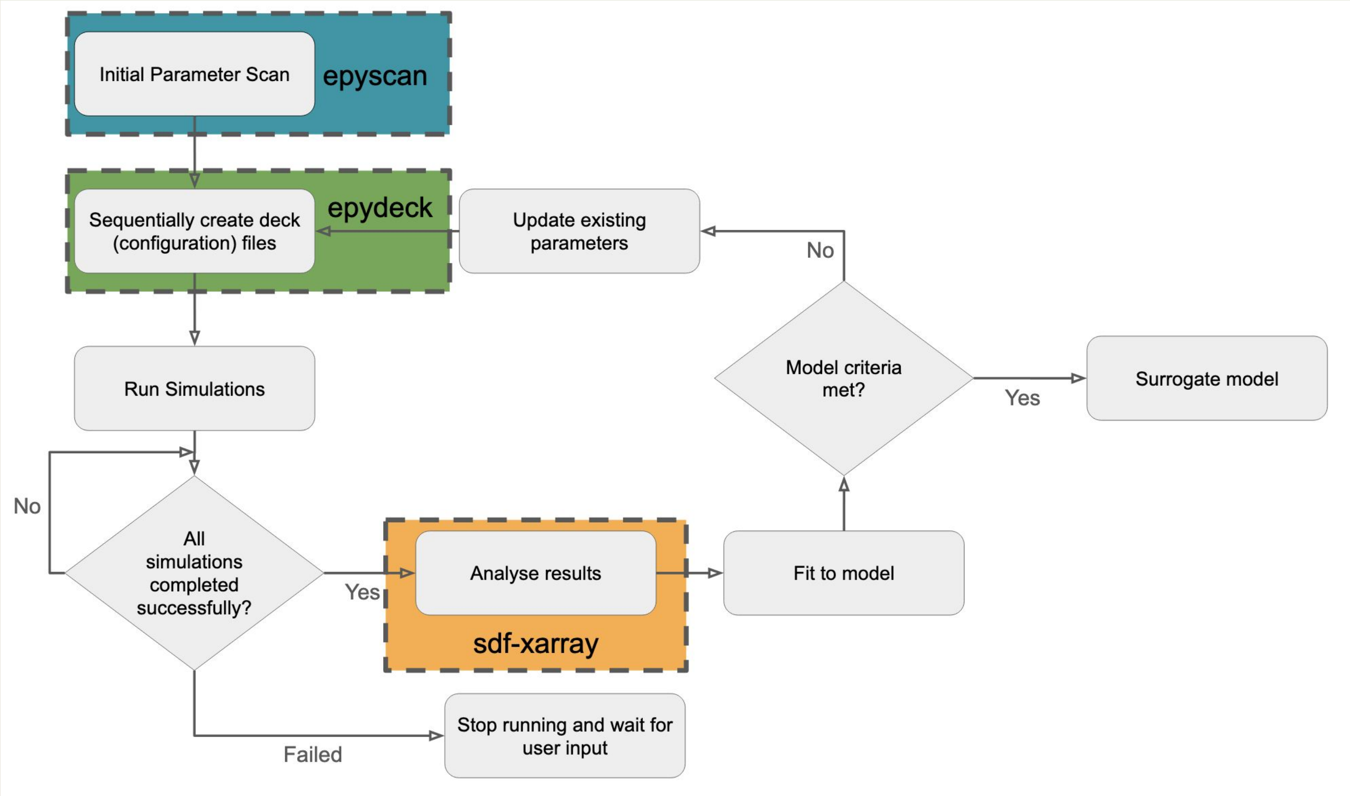
```python
import epydeck

# Read from a file with `epydeck.load`
with open(filename) as f:
    deck = epydeck.load(f)

print(deck.keys())
# dict_keys(['control', 'boundaries', 'constant', 'species', 'laser', 'output_global', 'output', 'dist_fn'])

# Modify the deck as a usual python dict:
deck["species"]["proton"]["charge"] = 2.0

# Write to file
with open(filename, "w") as f:
    epydeck.dump(deck, f)

print(epydeck.dumps(deck))
```

Usage: epyscan

- Works with either normal or log space
- campaign.setup_case() can be called later with new parameter values and will setup folders

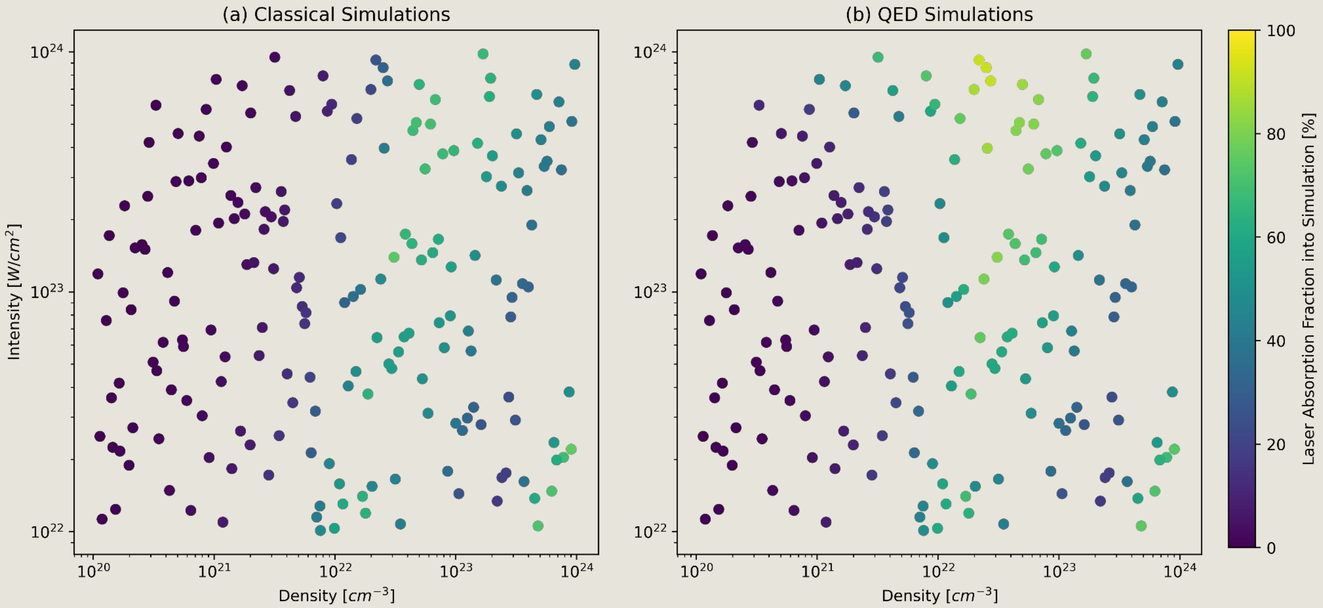
```python
import pathlib
import epydeck
import epyscan

with open("template.deck") as f:
    deck = epydeck.load(f)

parameters = {
    "constant:intens": {"min": 1.0e22, "max": 1.0e24, "log": True},
    "constant:nel": {"min": 1.0e20, "max": 1e24, "log": True},
}

# Sets up sampling of simulation and specifies number of times to run each simulation
hypercube_samples = epyscan.LatinHypercubeSampler(parameters).sample(30)
# OR
gridscan_samples = epyscan.GridScan(parameters).sample(30)

# Define the root directory where the simulation folders will be saved.
# This directory will be created if it doesn't exist
run_root = pathlib.Path("example_campaign")

# Takes in the folder and template and starts a counter so each new simulation gets saved to a new folder
campaign = epyscan.Campaign(deck, run_root)

# Randomly samples the parameter space and creates folders for each simulation
paths = [campaign.setup_case(sample) for sample in hypercube_samples]

# Save the paths to a file on separate lines
with open(simulation_dir_paths, "w") as f:
    [f.write(str(path) + "\n") for path in paths]
```

Surrogate Modelling - epyrunner



Epoch Simulations

- **Simulation**: 1D Epoch
- **Sampling:** Latin Hypercube sampling with 192 samples
- **Parameter Space:** Comparison of intensity and density
- **Analysis:** Similar results to graph



Surrogate Modelling

- **Training:** The models are trained on the all the data points from simulations, with separate models created for the classical and QED approaches
- **Kernel:** Implemented in scikit-learn using a kernel composed of a RBF with length_scale_bounds=(1e-1, 10) and WhiteNoise, applied to a GaussianProcessRegressor()
- **Fitting:** Models are fitted and scored on all data points. While not optimal, this approach provides a clear example