UCLA HEDP Experimental Plasma Physics Analysis Package Documentation

May 28, 2020

Contents

1 Overview

The UCLA HEDP package first loads

2 The Common Data Format (CDF)

2.1 Requirements

The UCLA HEDP package is designed around a standardized structured data format, hereafter referred to as common data format or CDF. A CDF object is an HDF file that conforms to a standard structure as defined in the function **tools.dataset.validDataset**. In particular, each CDF file must contain

- A CDF object is a group in an HDF file. It may be the root group, or multiple CDF objects may be housed within a HDF file as separate groups.

- A dataset called "data" with n dimensions. This array must have an attribute "dimensions" which holds a string array of list n which provides a name for each dimension in the dataset (by convention, all lowercase). A second attribute "unit" defines the units of the data in this array as a string that can by interpreted by the **astropy.units** string-to-unit parser.

  > In the future this could be generalized so that the CDF object could contain multiple data arrays with different names (but all with the same shape and the same axes). This would be conducive to datasets such as density and electron temperature planes from Langmuir measurements which share a set of axes and belong conceptually together. In this case, a new attribute array should be added to the parent group listing the names of all of the data arrays.

- For each entry in the "dimensions" attribute there must exist a dataset with the same name which contains the axis values for that dimension. This dataset must also include a mandatory "unit" attribute as defined above.

- Other attributes (commonly including metadata) are included as attributes of the root group in the HDF file. These entries should be tuples of the form (value, unit) where unit is a unit string as described in the previous points.

This format was designed to contain all of the information necessary to make plots of the dataset.

2.2 Conventions

- The convention [shots, time, channels] is commonly used for raw shot-ordered datasets.

- The convention [time, xaxis, yaxis, zaxis, repetition channel] is commonly used for processed volumetric datasets.

These conventions are chosen to allow easy plotting of subsets of an array. For example, if a dataset has dimensions B = [time, xaxis, yaxis, zaxis, repetition channel]. then it is easy to extract a time trace for a particular position, repetition, and channel B[t] = B[:, x, y, z, r, c] (where x,y,z,r,c represent fixed values for the other axes) or an XY plane B[x,y] = [t, :, :, z, r, c].

2.3 Considerations for Large Datasets

Large datasets are not stored on consecutive sections of computer memory but are rather broken into "chunks" that are stored separately. Accessing data spread across multiple chunks is much more computationally costly than accessing the same amount of data from a single chunk. The HDF format and the **h5py** package allow control over how a dataset is broken into chunks in order to maximize efficiency. Where possible, the UCLA HEDP package attempts to chunk datasets in a way that minimizes read times for common operations. For example, a dataset containing time arrays at a variety of spatial locations may chose to chunk the time axis together, anticipating that time traces may be accessed more often than large spatial volumes.

3 Structure of the Package

The UCLA HEDP package is separated into several sub-packages (folders within the package) that group similar routines. The logic of these groupings, and a brief explanation of the contents of each sub-package, is provided in the following sections.

3.1 **tools**

The **tools** package contains a variety of miscellaneous functions, many of which are used widely throughout other routines.

3.1.1 **csv.py**

The primary purpose of **csv.py** is to process a directory of comma-separated value (CSV) files in a file directory containing metadata about an experiment (see Sec. 4). A list of CSV files in a directory is created using **csv.getCSVList** and then those files are read into a python OrderedDictionary object by **csv.opencsv**. This package is commonly accessed from other routines through the function **csv.getAllAttrs**, which returns the metadata from across all files that pertains to a particular probe and run number. Functions **csv.getProbeList** and **csv.getRunList** provide lists of all valid probes and runs from an experiment, which are useful when creating loops to process data.

3.1.2 **dataset.py**

This file contains a function that tests whether an HDF group conforms with the CDF (as described in Sec. 2), which is **dataset.validDataset**. Another helper function **createDataset** can be used to create new datasets from a group of arrays.

    The remainder of this file contains functions that were intended to perform operations over large datasets in an efficient way. The function **dataset.chunked_ array_ op** was intended to provide an easy way to perform operations like thinning and averaging large datasets without forcing the user to manually chunk the data. However, this scheme turned out to be too time intensive because data operations are applied one-by-one. At this time, the most efficient way to process a large dataset is still to write a custom program that

performs all operations in a single pass through the file (a process which is difficult to generalize).

3.1.3 `hdf.py`

This file contains programs for interfacing with HDF files. In particular, **hdf.writeAttrs**, **hdf.readAttrs**, and **hdf.copyAttrs** are the preferred way to interact with attributes because they correctly translate strings to unicode and handle the (value,unit) format used in the CDF. This file also contains the **hdf.hdfPath** object which is provides a convenient way to pass both a filename and a HDF group path as one object. This was created with the idea of eventually supporting HDF files with multiple CDF objects within them (as separate groups).

3.1.4 `math.py`

This file contains mathematical functions used throughout the package. In practice most routines necessary are available in **numpy** or other packages, so the only function here is a discrete curl algorithm.

3.1.5 `pos.py`

This file contains routines for creating grids for volumetric measurements. The functions are generally called through the high-level function **pos.grid**, which takes a list of positions (along with some required attributes) and outputs a list of axes as and an array for placing shots into a grid. Two steps are necessary for this process: the creation of axes and the placement of points on those axes. Two methods exist for carrying out these operations, dubbed "strict" and "fuzzy" gridding. Which of these methods is applied is controlled by keywords.

Under strict axes, axes are created *a priori* based on metadata in the attributes dictionary, while under fuzzy axes axes are inferred from the provided position array. Similarly, under strict gridding shots are assigned to positions in the grid based on an *a priori* assumption about the movement of the probe (and metadata), whereas under fuzzy gridding positions are discritized (to within the grid precision) and placed at the grid cell that most closely matches the probes actual location. These methods can be mixed, eg. strict axes can be used with fuzzy gridding.

In general, fuzzy axes and gridding are more desirable. These ensure that the true location of the probes is used (a probe may not actually move to a location if it is physically unable to do so, which is not reflected in strict gridding). Fuzzy gridding also easily supports complicated probe paths and could easily be upgraded to support non-cartesian grids. Strict axes and gridding should be viewed as a backup for datasets on which fuzzy gridding fails for some reason.

> Modification of these routines to support grids in polar coordinates should be a relatively high priority in future in order to support "fan" grid patterns on radial probe drives like those on the LAPD.

3.1.6 `util.py`

This file contains several convenience functions, the most widely used of which is the **util.timeRemaining** object which is used to create progress timers for various operations throughout the package.

3.2 `load`

The **load** subpackage provides routines for transforming data from several sources into the standard CDF format. Once data is in this format, it can be processed by other routines in the package without regard to its origin.

Load routines also take a csv directory and merge the relevant attributes (based on the selected probe and run) with the dataset. This metadata is then attached to the HDF file, so changing the metadata necessitates reprocessing the data from the beginning starting with the load command. This choice was made intentionally to ensure that consistent metadata is used throughout the processing of a dataset.

3.2.1 `lapd.py`

These programs load data from HDF files output by the LAPD DAQ system using the **bapsflib** package. The routine can currently import position information from three probe drives: the XY drive, the XYZ drive, and the XY drive (6K Compumotor).

> Originally only the 6K Compumotor was supported by **bapsflib**, and so the other drives were handled manually. Support for other drives has now been added to **bapsflib**, so these routines could be updated to use that functionality.

3.2.2 `hrr.py`

These programs load data from HDF files output by the Phoenix Terminal's High Repetition Rate experiment software.

3.2.3 `imgdir.py`

This program loads a directory of images, commonly used to import frames from a camera. Images are loaded in an order determined by their filenames using a natural sorting algorithm.

3.2.4 `ascii.py`

This program loads data from an ASCII file, intended for creating CDF objects out of simulation output or output from instruments like oscilloscopes.

> This program may not be finished, but would be a useful tool. An additional desirable feature would be the ability to load whole directories of ASCII files and combine them intelligently into one CDF.

3.3 `process`

The **process** subpackage is the core of the UCLA HEDP package, containing routines that transform raw data into processed data ready for analysis. Each file described below processes data for a particular diagnostic or instrument. The example script **process.process.py** shows how all of these routines (along with those throughout the package) can be used to turn raw datasets and metadata files into processed output files.

3.3.1 `bdot.py`

The primary function in this file is **bdot.bdotRawToFull** which processes raw magnetic flux probe measurements into measurements of the vector magnetic field. Several of the keywords in this function merit some explanation

- **replace_ badshots** If this keyword is set, shots marked as bad by the tdiode routine will be copied over by the nearest good shot.

- **offset_ range** gives the range (in indices) over which the voltage offset of the signal will be calculated. This should done at a time when the real signal is nominally zero.

- **offset_ rel_ t0** allows the user to set values in the previous variable as being relative to to the laser firing. A common configuration is for the start of the offset range to be fixed at zero, but the end to be set to be just before the laser pulse. In this case the user may set **offset_ range** = (0, -50) while **offset_ rel_ t0** = (False, True).

This file also contains two functions **bdot.fullToCurrent** and **fbdot.ullToBmag** which take a fully processed bdot signal and calculate either the current (using the curl and neglecting the displacement current in Ampere's law) or the magnitude of B.

The file also contains the **bdot.calibrateProbe** function which performs fits to calibration data recorded in a CSV file by a network analyzer. This program assumes a standard Helmholtz coil configuration is used to calibrate the probe.

> The current **bdot.calibrateProbe** function can only handle CSV files, while the network analyzer used at UCLA for this purpose outputs **.dat** files. Reading these files directly should be possible, and adding that functionality to this function would be very helpful.

3.3.2 `tdiode.py`

This program takes in data from a timing diode (typically a photodiode positioned in the leak of a laser mirror that shows the laser pulse) and determines the index corresponding to the rise of the pulse (corresponding to t0). The program also identifies bad shots (no laser) by comparing the maximum of the pulse to the noise level.

3.3.3 `langmuir.py`

This file contains several programs for processing Langmuir probe measurements. `langmuir.isatRawToFull` processes ion saturation current measurements and is relatively straight forward. `langmuir.vsweepLangmuirRawToFull` processes swept Langmuir probe measurements and is considerably more complicated. The swept Langmuir probe routine makes use of several subroutines

- `langmuir.find_ sweeps` takes a time array and a voltage trace of the ramp voltage and identifies the time ranges corresponding to each ramp. By default the swept Langmuir analysis routine will attempt to fit ever ramp (although many at lower densities may fail).

- `langmuir.vsweep_ fit` takes a Langmuir IV curve (voltage and current traces) and fits the electron saturation and transition regions with a linear fit and an exponential fit respectively. The program attempts to intelligently determine the boundaries for these regions, but they may also be adjusted by the user. This function is designed to recognize failed fits, which are commonly caused when a probe moves into a region of lower density. These failed fits are intended to cause errors which results in the value `fail_ val` being output in the parameters so these datapoints can be easily ignored in analysis.

> Some rudimentary error analysis is in place for this function: in principle it should be possible to transform the covariences from the curve fits into error bars on the output quantities like density and electron temperature. In practice propagating the the errors from both fits through the formulas is non-trivial, but this would be a great project to finish in the future.

3.3.4 `interferometer.py`

This file contains a simple routine for computing the phase shift measured by an interferometer from a hetrodyne interferometer given the signal and the reference signal.

> The full calculation of the density from the phase shift has not yet been implemented.

3.3.5 `imgseq.py`

This file contains a function that transforms a CDF file created from an image directory corresponding to a sequence of images and creates appropriate physically meaningful axes for it. Attributes can be set in the metadata to change the scale of the pixels in physical units, set an origin, and create a time vector.

3.3.6 `scope.py`

This program is a dummy processing routine that does nothing to the data other than apply a timing diode correction. The intended use-case is for quick analysis during an experiment of instruments that are not otherwise handled by the UCLA HEDP package.

3.3.7 `process.py`

This script is an example/template script for running all of the analysis programs in the UCLA HEDP package. The program **process.process** takes runs the full sequence of analysis steps on a probe for a particular run. A raw file is created using the appropriate load function, then the appropriate final processing routine is applied.

process.processMany runs this program in a loop over a set of runs and probes provided by the user. Combinations of runs and probes that do not apply are ignored. This allows the user to set up a large list of runs and probes (potentially an entire experiment) to run unattended.

> This file should really live in the examples folder, since it's really just an example of a processing script.

3.4 `analyze`

The **analyze** subpackage contains useful routines for data analysis.

3.4.1 `filter.py`

This file contains several frequency filtering tools that can be used for either temporal or spatial frequency filtering. **`filter.fftFilter`** is a Fourier-filter than provides a bandpass (or band-block) filter on a 1D trace. **`filter.lowpassFilter2D`** is a lowpass filter intended for spatially filtering 2D planes prior to calculating the curl in order to calculate the current.

3.4.2 `polarization.py`

This file contains functions for analyzing the polarization of a signal based on a vector time trace either using a polarization decomposition algorithm or by creating a hodograph.

3.4.3 `spectral.spectrogram.py`

3.4.4 `thomson.thomson_scattering.py`

3.5 `user`

3.6 `dataview`

4 The CSV Metadata Format