FLAP User's Guide

S. Zoletnik, M. Vécsei, Wigner RCP

zoletnik.sandor@wigner.mta.hu

Document version 1.4, 19 May, 2019

Contents

Introduction

The Fusion Library of Analysis Programs (FLAP) is a Python framework to work with large multi-dimensional data sets especially for turbulence data evaluation in fusion experiments. Data are stored in data objects together with coordinates, data names and coordinates, thus the built-in plotting functions create figures with correct axes. The data set can be sliced to reduce dimensions and thus enable visualization of more than 2D data sets. FLAP is a modular package: data are read through modules which register themselves to the FLAP framework. This way data are read through a uniform interface by defining data source, experiment ID and data name. Also coordinate conversion is done by functions in the data source modules.

Configuration

FLAP can be adapted to the local environment using a configuration file. When flap is imported the default configuration file "flap_defaults.cfg" is loaded from the working directory. If the file is not found a warning is printed. A configuration file can also be read explicitly using the flap.config.read() function.

The configuration file is a windows-style configuration file consisting of sections and elements. Sections start with their name in []. The elements follow on individual lines. The name of the element and the value is separated by =. Space can be used in both the section names and element names. Lowercase and uppercase characters are different. Usually names start with upper case but this is not a requirement. An example:

```
[PS]
 Resolution = 1e3
 Range = [1e3, 1e6]
[Module TESTDATA]
 PS/Resolution = 100
 Name = 'This is a string'
```

In the above example section "PS" contains two elements. Section "Module TESTDATA" refers to the TESTDATA data source module. The element PS/Resolution refers to the resolution element in the PS section and enables overriding section settings with module specific values.

All elements in a section can be read with the flap.config.get_all_section() function. This returns a dictionary with keys referring to element names. The values are converted using the following rules:

- True and Yes is converted to boolean True.
- False and No is converted to boolean False.
- An element enclosed in single or double quotes is handled as string (without the quotes).
- Elements which can be converted to int, float or complex are interpreted accordingly.
- An element enclosed in square brackets ([]) is interpreted as a list. List elements should be separated by commas. Each list element is interpreted using the same rules as one element of the section.
- If all the above interpretation attempt fails the element is handled as string.

Options and defaults

Standard flap functions (like flap.get_data(), flap.apsd, ...) take an "options" keyword argument. A default options list is defined inside the routine, ehich contains all possible option keys understood by the function and default values for them. If no options are passed to the function these will take effect. The function might also be linked to a section in the configuration file. Options read from this section override the default options. The data object processed by the function might contain a data_source variable. If that is present and not None configuration file elements in the section of that module are read. (That is in the above example for TESTDATA data_source Module TESTDATA section is read.) elements starting with the section name of the function followed by / override settings in the section. Finally values in the function input options dictionary override all the above. As all possible options of the function are known from the default options dictionary it is allowed to abbreviate the option names in the function input options list up to the point where it matches only one key. (In the configuration file full option keys should be used.) This also means that any key in the default key list cannot be an abbreviation of another one. (E.g. 'A' and 'A1' are not allowed.)

This procedure is handled by the flap.config.merge_options() function.

Data objects

Data objects are flap.DataObject class variables. They contain a multi-dimensional data array, an optional error array (symmetric or asymmetric), data name and unit and various information elements (info, history) which at present are not fully developed. A data object can optionally contain an exp_id variable describing the experiment ID from which the data originates from and the data source module name. An arbitrary number of corodinates are contained in the data object. This enables automatic plotting with proper axes and various calibrations.

Data sources

FLAP can make use of various data read modules which can be dynamically added to the package. Each package registers its data read and optional coordinate addition function in FLAP so as data are read using a single FLAP function called get_data. The parameters are the data source name, data name (interpreted by the data source module), experiment ID and additional corodinate information so as data can be limited to certain ranges or resampled in channels, time, etc. A single get_data call can read any number of measurement channels, there is an extended wildcard interpretation method which enables e.g. using Signal[2-28] to read signals from 2 to 28 into one data array. The module data read function can add as many coordinates to the data array as it desires to be useful. (See information on FLAP corodinates below.) Standard coordinates are Time, Signal name, Channel, etc. For information on writing a FLAP data source please see the appropriate section below.

Coordinates in FLAP

In the FLAP program package coordinates are stored with the data. This document describes the implementation of this feature.

Data storage and coordinates

Data are stored in an n-dimensional numpy array in the FLAP.DataObject class variable. This n-dimensional space we call *data sample space*. Different dimensions of the array are

associated with primary coordinates, like sample number, channel number, or e.g. for simulated data x, y. However, these primary coordinates are often not useful and we need to make plots along physical coordinates. This can be handled by adding other coordinates to the DataObject. Also during processing some coordinates might be turned to others. An example is calculating power spectra. From a 2D measurement data with channel, time coordinates spectrum calculation creates another 2D array with channel, frequency as coordinates.

In order to be more general by coordinate we will consider all information related to the data, like measurement times, spatial locations, frequency, etc, especially what is variable for the data array elements. However, this is not necessarily the case, a single scalar coordinate value can be assigned to all elements as well. Coordinate information is not necessarily of numeric type, e.g. channel name can also be considered as coordinate information. On the other hand, other information (e.g. date of the measurement, measurement device configuration information) are not considered as coordinate but stored in the info dictionary of DataObject.

Multiple coordinate information may be present in the DataObject but all of them assign a value to all the array data elements. Storage of the coordinate information is designed by considering that a coordinate mostly changes along one or a few dimensions of the data, in a lot of cases coordinate values are equidistant but in special cases a coordinate value might change along all dimensions of the data array. This way the simple cases are described with minimal amount of data, while enabling even the most complicated case when data is practicably doubled by adding a randomly varying coordinate. Data processing, plotting is optimal if a coordinate changes along one dimension only.

Representation of coordinates

Coordinates have a name and unit, both described by a string. Standard names are Channel name, Channel number, Signal name, Time, Sample, Device x, Device y, Device z, Device R, Device Z, Device phi, Flux r, Flux Theta, Flux phi, Image x, Image y, Frequency, Time lag. Any other names and units can be used, but it is preferred to use the above where possible. The names are case sensitive as usual in Python. The type of the coordinate values is dependent on the coordinate type. E.g. Sample, is integer, Time is either float or Decimal, Signal name is string.

The following variables are defined in the Coordinate class, but not all of them are used in all definitions: name, unit, mode, shape, step, start, values, value_index, value_ranges, dimension_list. Unused variables are set to None.

Coordinates are not stored in the data matrix but each coordinate description is contained in a FLAP.Coordinate class object. Such an object describes the coordinate values in a d-dimensional rectangular *coordinate sample space* described by the shape variable what is a tuple of sample numbers (s_1,s_2,...s_d) in each dimension, similarly to shape in numpy arrays. If shape has 0 elements it means that the coordinate value is constant and described by the 'values' and 'value_ranges' variables. The coordinate sample space is a subarray of the data sample space. As an example consider measurements on a 2D spatial mesh. At each measurement point a time signal is collected, thus the data sample space is 3D. If the 2D mesh is rotated relative to physicsal x,y coordinates then these physical coordinates will change on the 2D mesh. This way the coordinate sample space of x and y will be 2D, while the coordinate samples space for the time coordinate will be 1D. The link between the coordinate sample space and the data sample space is established by the dimension_list element of FLAP.Coordinate. This has

number of elements equal to the dimension of the coordinate sample space and each element contains the index of the related data sample space.

The coordinate values are described in the coordinate sample space [0...s_1-1, 0...s_2-1, ..., 0...s_d-1] in one of two ways.

- If FLAP.Coordinate.mode.equidistant is False samples of the coordinate value are given on a regular or irregular grid in the coordinate sample space. The following cases are considered:
  - If value_index is None and the shape of the coordinate sample space is identical to the corresponding subspace of the data sample space, then there is a one-to-one correspondence between data samples and coordinate samples. The coordinate values do not change along dimensions which are not in FLAP.Coordinate.dimension_list.
  - If the two above shapes are different but value_index is None interpolation is done in the directions with different number of elements assuming that first and last samples match.
  - If value_index is not None than coordinate samples are on an irregular grid. The coordinate sample locations are given in the 'value_index' (d by N_{samp}) array where N_{samp} is he number of coordinate samples. The coordinates in the sample space are between 0 and s_i in the i-th dimension. The coordinate values are given in the 1D 'values' array which has N_{samp} elements. To calculate the coordinate value for the data array points a (multi-dimensional) interpolation is done between the sample coordinate system and the data sample coordinates.
- If FLAP.Coordinate.mode.equidistant is True then the coordinate sample space is assumed to be identical to the subspace of the data sample space selected by 'dimension_list'. (FLAP.Coordinate.shape is not used.) The coordinates change linearly in each dimension: $c=b+s_1x_1+...s_dx_d$, where b is the 'start' element of FLAP.Coordinate and s_i is the step size in dimension i of the data sample space. The si values are stored in the 'step' element which is a d long 1D array.

The coordinate values may have a range which is either symmetric or asymmetric around the values. This can be considered either as an error of the coordinate or measurement range, and it is described by a value_ranges variable. If FLAP.Coordinate.mode.range_symmetric is True the range is symmetric around the coordinate values, otherwise there is a low and high range. For the equidistant coordinate description 'value_ranges' is either a scalar or 2-element array depending whether the range is symmetric or asymmetric. For the non-equidistant coordinate description in the symmetric case 'value_ranges' has the same shape as 'values', for the asymmteric case it is a dictionary with 'low' and 'high' keys. Each dictionary element has the same shape as values.

The 'coordinates' variable of the FLAP.DataObject is a list of FLAP.Coordinate class objects.

Converting coordinates

Each data source may name a function in the registration process in the add_coord_func keyword variable. The add_coordinate() method of FLAP.DataObject gets coordinate name(s) (string, or string list) and options dictionary. It calls the function registered for the given data source with the data object, the new coordinate name(s) and options arguments. The function

should add the named coordinate(s) to the data object or raise a ValueError. The function knows the experiment ID and other information about the data, therefore it should be possible to calculate the new coordinate.

Explanations and examples

The above definition is complex but it has a reason. It contains all possibilities from the most simple to the most complex. The coordinate descriptions are usually prepared in the data read module and the coordinate values accessed by the data() method of the Coordinate class, therefore the user should not take care of details of the coordinate description. Additionally, the most often encountered cases are very simple, difficulty arises only e.g. when random points are measured in time dependent flux coordinates at random time samples.

In the examples below we do not indicate the coordinate ranges, it can be simply added as described above.

Some typical situations:

- **Constant coordinate.** This is useful where e.g. a measurement is done with all measuring points in the Device z=const. coordinate. This constant can be entered in the DataObject description to be used later when e.g. mapping is done from device to flux coordinates.
  shape = []
  values = <z>
- **Independent equally spaced coordinates along each dimension of the data array.** In this case a coordinate is defined for each dimension of the data array. The definition of each coordinate contains a scalar start and a step value. The shape variable is one number, only the number of elements of shape is used showing that the coordinate description is 1D.
  shape = 1
  mode.equidistant = True
  start = <start>
  step = <step>
  dimension_list = [0]

  In the above example the coordinate changes along the first dimension of the data array.
- **Array of N temporal signals measured at N different points in the device coordinate space.** The data is stored in a 2D array, one dimension (0) is time, the other is channel. In this case a 'Time' coordinate is described with equidistant spacing as shown in the previous example. To describe the measurement spatial coordinates additionally to 'Time' 3 coordinates are entered in the coordinates list of the DataObject: 'Device x', 'Device y' and 'Device z'. The description for the x coordinate is:
  shape = N
  mode.equidistant = False
  values = *<array with N elements of coordinate values>*

  The other two coordinates are entered similarly. The time vector and x,y,z coordinates of measuring channel i can be obtained from the d DataObject as:

  time = d.coordinate('Time',(...,i))

x = d.coordinate('Device x',(i,0))
y = d.coordinate('Device y',(i,0))
z = d.coordinate('Device z',(i,0))

In this example it is also useful to additionally define a 'Signal name' and maybe a 'Channel' coordinate. Signal has normally string values (that is non-equidistant array, values is a list of strings).

- **Fast measurement signals at an array of spatial points mapped to a temporally slowly variable flux coordinate system.**
  The data are stored in a 2D array, 1-st dimension is channel, second is time. The data read routine enters the device coordinates into the DataObject. From this the flux coordinate calculation method generates the flux coordinates of the measurement points at a few time points (N_t) during the measurement time. 3 coordinates are added to DataObject, the three flux coordinates. For each coordinate the calculated values are put into a 1D array. The value_index will be a $2xN_t$ array, at each time point the channel number and the flux coordinate calculation time will be entered. The time is normalized to $(t-t_{start})/(t_{end}-t_{start})*(N_t-1)$. The shape variable is (N_{ch}, N_t), where Nch is the number of channels, mode is set to 0 and dimension_list to [0,1]. As in the channel direction the mapping is 1:1 from the coordinate sample coordinate and the data matrix coordinate no interpolation will occur. In the time direction interpolation will be done and the flux coordinates of each measurement channel will be interpolated values between the sparsely known flux coordinates. The Time coordinate is entered as an equidistant coordinate description.

Data storage

Data object variables can be passed between functions in a program as any other variable. However, additionally to this FLAP contains a memory storage facility where data objects can be stored under a name and experiment ID. This enables loading and processing various data without the need of passing around a large number of variables. Data can be entered into the storage by the flap.add_data_object function and retrieved by flap.get_data_object. It is also possible to directly enter a data object from the flap.get_data function or all of the data processing functions.

Listing the content of FLAP storage

The flap.list_data_objet can be used to list properties of the data objects in FLAP storage. Data can be selected by name and exp_id (wildcards can be used). The data shape, properties and properties of coordinates are listed.

Save/Load

Data objects can be saved to a file either from the FLAP data storage or from variables using the flap.save function. It can take a list of data objects or other variables or a list of strings and experiment IDs. In the latter case the data objects named by the strings and experiment IDs are loaded from FLAP storage before saving them. The save routine uses the pickle Python module to encode data. The file contains information whether data originates from FLAP storage or from variables. When data are loaded using the flap.load function they are returned as list of variables. If the data were saved from FLAP storage it can be entered there with the same naes as well.

Data processing

The data processing routines are always available in two versions:

- A method of the flap.DataObject class. The method does not change the original data object, rather returns the processed object.
- A flap.<xxx> function, where <xxx> is the same name as the respective method in the flap.DataObject class. These functions read a data object from FLAP storage, call the method on them and store the result either the same or new name.

The function always have the same arguments as the method plus a few additional ones:

- The first positional argument is the object name.
- An exp_id keyword argument sets the exp_id of the data object. Default is '*', therefore exp_id need not be set unless there are data objects with the same nae and different exp_id in the storage.
- an output_name keyword sets the name of the resulting data object. If it is not set the result will be stored under the same name as the input.

Each processing function/method returns the resulting data object, therefore operations can be chained:

d.filter_data().apsd().plot()

Setting defaults for the processing method (see section "Options and Defaults") the exact parameters of the processing need not be written out in the most often used cases.

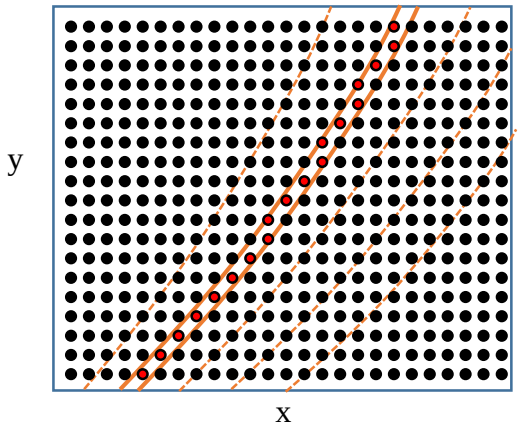
Slicing

Slicing means selecting certain elements in the data matrix and optionally taking their sum, minimum, maximum, or doing some other operation on them. Description of the silicing operation is based on coordinates. (Although originally it was foreseen to do slicing along data dimensions, this is not considered useful now.)

Slicing is performed with the slice_data method. In the slicing argument it takes a dictionary with keys referring to coordinates. The values describe how slicing is done. If the slicing dictionary has multiple keys the slicing operations are done sequentially, except a special case, see below. Summing is done after slicing. (If the slicing argument is omitted only summing is done.) Summing again defined by a dictionary where keys refer to coordinate names.

If the slicing coordinate changes only along one dimension of the data array slicing is done on the data along the associated dimension (see dimension_list). Other coordinates changing along this dimension are adjusted. It has to be noted that coordinate changes might result in changing from equidistant to non-equidistant type, which can cause more data in the data object. If only one data remains in the sliced dimension that dimension is dropped from the data and also coordinate dimension lists are adjusted correspondingly.

If the slicing coordinate changes along multiple dimensions the situation is more complex as shown in the 2D example below. Here x,y are the original coordinates in a 2D array and R is some coordinate derived from them. The points are arranged in an x-y coordinate system. The orange lines indicate constant R contours. The two solid lines indicate slicing in the R coordinate, the red filled dots are the selected points. Selecting elements in the data in the range of a coordinate which changes in multiple dimensions means that the selected sub-array becomes non-rectangular. In this case the data along these dimensions will be flattened to 1D before slicing and the slicing operation will be done on the flattened dimensions.



To illustrate this further let us consider a 2D image. A polar coordinate system with origin in the center of the image is introduced and slicing is done in the radial coordinate. Selecting one radial area results in a 1D array. The angle coordinate will change non-monotonically on this. However, coordinates are corrected accordingly and it is still possible to plot as a function e.g. of polar angle.

Two basic slice types are distinguished:

- Simple slice is an operation when a single interval or individual elements are selected along a coordinate. This is described above.
- Multi-slice is an operation when multiple intervals are selected from the data. In this case the dimensions along which the slicing coordinate changes are flattened as described at simple slice and the intervals are selected. Two new dimensions are added to the data matrix. Along one the interval number, along the other the data index inside the intervals change. The interval data are distributed into these new dimensions and the original flattened dimension is removed. With this procedure it becomes possible to plot/sum data in individual intervals or across intervals.

Multi-slice is a complicated procedure and the above described scheme breaks down when multi-slice is intended on two coordinates which change on (partly) common dimensions. E.g. in the above described case of multi-slicing the data on an x-y grid to a r-phi grid poses problems. After multi-slicing with r one gets the two dimensions along and accross the intervals. However, the multi-slicing along phi would flatten these into one dimension and create new intervals in phi. To avoid this multi-slicing along multiple coordinates with common dimensions is done in one step and data are distributed into boxes arranged along each dimension. In case of n such slicing operations n+1 dimensions are added with one dimension

where the interval number changes along each coordinate and a single dimension where the sample index in one interval box changes. This case is not implemented yet.

After a multi-slice operation coordinates changing along the flattened dimensions are split into two coordinates: “Rel. <coord> in int(<sl_coord>” and “Start <coord> in int(<sl_coord>”. (Except for string coordinates where this is not possible and the original coordinate shape will be changed.) Here <coord> is the name of the coordinate and <sl_coord> is the name off the slicing coordinate. Also coordinates with names “Interval(<sl_coord>)” and “Interval(<sl_coord> index)” are added storing the interval number (along one coordinate) and the sample index in one interval.

If multi-slice operation results in different interval length, the dimension along the samples in the intervals will be set to the longest. Where data is shorter in one interval np.NaN values will be filled in case of float data and 0 for int. (There is no integer Not-a-number value in Python.) In the coordinate matrix missing elements will be filled similarly to data.)

Slicing can be described with the following objects:

1. For simple slice:
   a. A Python slice or range object, to select a sequence of regularly spaced elements.
   b. A scalar value or a list of scalars to select random elements.
   c. A numpy array to select random elements.
   d. flap.DataObject without error and with data unit.name equal to the slicing coordinate name.
   e. flap.DataObject with the data unit.name not equal to the slicing coordinate, but one of the coordinate names equal to the slicing coordinate and the coordinate has no value_ranges.
   f. flap.Intervals object with one interval.

In cases 1a-e the data elements with closest coordinates will be selected, while in case 1f all elements in the interval will be selected. In case a string type slicing coordinate matching between the slicing and the coordinate value is required instead of close match. (There is no sense in close match for strings.) However, extended wildcards can also be used, e.g. slicing=’{Signal name’:’TEST-*-3’} is a valid slicing expression.

2. For multi-slice:
   a. flap.Intervals object with more than one interval.
   b. flap.DataObject with data unit.name equal to the slicing coordinate. The error values give the intervals.
   c. flap.DataObject with the data unit.name not equal to the slicing coordinate name but name of one coordinates equal to the slicing coordinate. The value_ranges select the intervals.

The summing input argument to the slice_data method can be used for processing the sliced data. This is also a dictionary with coordinate names as keys. Before processing the dimensions where the summing coordinate changes will be flattened. The values of the dictionary can be the following:

- ‘Sum’: Add all elements.
- ‘Mean’: Take the mean of all elements
- ‘Min’: Take the minimum of all elements

- ‘Max’: Take the maximum of all elements

As the result of the processing is a single value along the summing coordinate, this dimension will be removed from the data. After processing the data the coordinate changes will be done. In the case of ‘Sum’ and ‘Mean’ the mean of the coordinates of the summed data will be taken, while in the case of ‘Min’ and ‘Max’ the coordinate of the minimum or maximum value will be selected.

Examples

As an example we read all signals from the TESTDATA module for a 1 ms piece and store under name TESTDATA in flap storage:

```
d=flap.get_data('TESTDATA',name='*', options={'Scaling':'Volt'},
                object_name='TESTDATA', coordinates={'Time':[0,0.001]})
```

This results in a 3D data object where signals are arranged in row and column and the third dimension is time. ‘Time’, ‘Sample’, ‘Row’, ‘Column’ and ‘Signal name’ coordinates are supplied by the data read routine. Then we add spatial coordinates:

```
flap.add_coordinate('TESTDATA',
                    coordinates=['Device x','Device z','Device y'])
```

We can list the content of the data object using the flap.list_data_objects() call:

```
TESTDATA(exp_id:None) "Test data" shape:[15,10,1001]
  Coords:
    'Sample'[n.a.](Dims:2) [<Equ.><R. symm.>] Start:  0.000E+00, Steps:  1.000E+00
    'Time'[Second](Dims:2) [<Equ.><R. symm.>] Start:  0.000E+00, Steps:  1.000E-06
    'Signal name'[n.a.](Dims:0,1, Shape:15,10) [<R. symm.>] Val:TEST-1-1, TEST-1-2, TEST-1-3,
TEST-1-4, TEST-1-5, TEST-1-6, TEST-1-7, TEST-1-8, TEST-1-9, TEST-1-10, ...
    'Column'[n.a.](Dims:0, Shape:15) [<R. symm.>] Val. range:  1.000E+00 -  1.500E+01
    'Row'[n.a.](Dims:1, Shape:10) [<R. symm.>] Val:1, 2, 3, 4, 5, 6, 7, 8, 9, 10
    'Device x'[cm](Dims:0,1, Shape:15,10) [<R. symm.>] Val. range: -1.112E+00 -  6.657E+00
    'Device z'[cm](Dims:0,1, Shape:15,10) [<R. symm.>] Val. range:  0.000E+00 -  5.587E+00
    'Device y'[cm](Dims:, Shape:) [<R. symm.>] Val: 0.000E+00
```

The above list shows that Sample and Time changes along dimension 2 (3-rd dimension), Signal names change on the first tow dimensions, Column and Row changes on dimension 0 and 1, respectively. Device y does not change at all (emply dimension list) as the measurement channels are in the y=0 plane. Device x and y both change on dimensions 0,1 as the measurement matrix is inclined in the x, y plane.

A simple slice to select one signal looks like:

```
flap.slice_data('TESTDATA', slicing={'Signal name': 'TEST-1-3'},
                output_name='TESTDATA_slice')
```

The result is a 1D array:

```
TESTDATA_slice(exp_id:None) "Test data" shape:[1001]
  Coords:
    'Sample'[n.a.](Dims:0) [<Equ.><R. symm.>] Start:  0.000E+00, Steps:  1.000E+00
    'Time'[Second](Dims:0) [<Equ.><R. symm.>] Start:  0.000E+00, Steps:  1.000E-06
    'Signal name'[n.a.](Dims:, Shape:1) [<R. symm.>] Val:TEST-1-3
    'Column'[n.a.](Dims:, Shape:1) [<R. symm.>] Val:1
    'Row'[n.a.](Dims:, Shape:1) [<R. symm.>] Val:3
    'Device x'[cm](Dims:, Shape:1) [<R. symm.>] Val:-2.472E-01
    'Device z'[cm](Dims:, Shape:1) [<R. symm.>] Val: 7.608E-01
    'Device y'[cm](Dims:, Shape:) [<R. symm.>] Val: 0.000E+00
```

Extended regular expressions can also be used. In the following expression 3 signals are selected, resulting in a 3x1001 data array:

```
TESTDATA_slice(exp_id:None) "Test data" shape:[3,1001]
```

```
Coords:
  'Sample'[n.a.](Dims:1) [<Equ.><R. symm.>] Start:  0.000E+00, Steps:  1.000E+00
  'Time'[Second](Dims:1) [<Equ.><R. symm.>] Start:  0.000E+00, Steps:  1.000E-06
  'Signal name'[n.a.](Dims:0, Shape:3) [<R. symm.>] Val:TEST-1-8, TEST-1-9, TEST-1-10
  'Column'[n.a.](Dims:0, Shape:3) [<R. symm.>] Val:1, 1, 1
  'Row'[n.a.](Dims:0, Shape:3) [<R. symm.>] Val:8, 9, 10
  'Device x'[cm](Dims:0, Shape:3) [<R. symm.>] Val:-8.652E-01, -9.889E-01, -1.112E+00
  'Device z'[cm](Dims:0, Shape:3) [<R. symm.>] Val: 2.663E+00,  3.043E+00,  3.424E+00
  'Device y'[cm](Dims:, Shape:) [<R. symm.>] Val: 0.000E+00
```

Data object conversions

The following methods of data object result in a modified object.

real

Returns a new data object with the data the real part of the original data. Coordinates and other properties are not modified.

imag

Returns a new data object with the data the imaginary part of the original data. Coordinates and other properties are not modified.

amp

Returns a new data object with the amplitude of the (complex) data.

error_value

Returns a new data object where the data is a copy of the error of the original data object and error is None.

Signal processing methods

Most signal processing methods/functions have a similar logic and interface. They operate on one or more “signals” which change along one coordinate, e.g. time. However, they can also be used to perform processing along any other coordinate, e.g. some spatial coordinate. The processing of these methods can be limited to a set of intervals in a coordinate. Two coordinates are distinguished. The processing coordinate is the one along which the processing is done, the selection coordinate is in which the intervals are defined. The common input arguments are the following:

- coordinate: name of a coordinate along which the processing is done.
- intervals: description of the intervals to which the operation is restricted. There are multiple possibilities:
    - Intervals is None. In this case no intervals are used, all data are processed.
    - Intervals is a dictionary. In this case only one key is expected to be present and the key is a coordinate name. The intervals are taken in this coordinate, which is not necessarily the same as the coordinate argument. The value of the dictionary is the interval description (see below).
    - Intervals is neither None, nor dictionary. In this case the intervals are selected along the processing coordinate and the intervals argument is the interval description.

  The interval description can be any of the following:

    - List of two numbers: describes a single interval between these tow coordinate values.

- flap.Intervals object. Can describe regular or irregular intervals.
- flap.DataObject. There are two cases similarly to slice_data. If the data name equals the selection coordinate then the data and errors are used as intervals. If the data name is different than the selection coordinate but one of the coordinates in the data object is identical to the selection coordinate then the coordinate values an value ranges are used as intervals.

- options: any other parameter for processing.

Signal processing methods are the following:

detrend

This method subtracts a trend from the data along a given coordinate. If the coordinate is not set 'Time' is used.

select

Selects intervals from a signal either manually with the mouse or automatically based on some events. The selection can be limited to a set of intervals as in all signal processing methods. The intervals are saved in a data object. This can be used in other functions which use intervals, e.g. most of the data evaluation methods and slice_data. With slice data conditional averaging can be implemented

apsd

APSD stands for Auto Power Spectral Density. This method converts the processing coordinate (default is Time) to "Frequency" or "Wavenumber". The intervals argument is the same as in the case of other signal processing methods, but it is used differently. Identical length intervals with a given minimum number are placed into the processing intervals. APDS is calculated in all intervals and the mean and statistical error is determined from them at each frequency/wavenumber point.

cpsd

This method calculates complex cross-spectra or coherency (and crossphase) between all signals in two data objects. It does not calculate cross-spectra in one object, to do this the second data object can be omitted. It is similar to apsd as it converts a processing coordinate to Frequency/Wavenumber. However, the number of dimensions of the resulting data object is the sum of the dimension number of the two data objects minus 1.

filter_data

Filters the signals along a certain axis with one of Numpy's filters. For defining the filter it has a simplified interface capturing the important aspects.

Writing a data source module

A data source module differs from a normal Python module in the in that it can provide a few functions for the FLAP framework. These are the following:

In order to implement data read into FLAP the module should provide a function with the following input arguments:

```
def get_data(exp_id=None,
             data_name=None,
             no_data=False,
             options=None,
             coordinates=None)
```

The name of the function is irrelavant. The input arguments are the following:

exp_id:

The experiment ID that is shot number or any other numeric or string describing the particular experiment from which the data is requested.

data_name:

Name of the data requested. This is a string or list of strings. The strings may contain Unix-style wildcards like CH[1-8] but also an extended wildcards can be used like CH[1-23]. This allows reading an arbitrary number and configuration of measurement channels.

no_data:

If False then no data should be returned only the coordinates filled.

options:

A dictionary of options. No make use of options in the configuration file and abbreviation of options follow the description below.

coordinates:

This list may contain flap.Coordinate objects which can precisely define the coordinate ranges from which the data should be read through the "values" and "value_ranges" in the flap.Coordinate object. As a simplification the "c_range" element of the flap.Coordinate object gives a simple range.

The **return value** should be a flap.DataObject. The arguments are described in the data_object.py function of the flap package. Any number of coordinates can be added to the flap.DataObject.coordinates list.

To make use of options in the configuration file and option abbreviation the following should be provided in the data read function:

- A dictionary should be created with all possible options and their default values.
- The flap.config.merge_options() function should be called the following way:

```
_options  = flap.config.merge_options(default_options,
                                      options,
                                      data_source='DATASOURCE')
```

In the above call DATASOURCE should be repalced with the data source name used in registering the data read function (see below). default_options is the default options dictionary and options is the options dictionary received by the get data function. In the above example the merge_options function reads entries from the configuration file section 'Module

DATASOURCE' and all the entries which has no {xxx} at their beginning are added (or replaced) to the options list received in the default options argument. This means function defaults are replaced by configuration defaults. Finally the dictionary keys in the options arguments are checked whether they can be interpreted as abbreviation of only one option and if is fulfilled their value is entered into that options. This overwrites both function and configuration file defaults with input options. The resulting options dictionary is returned by the function. It should be named differently then the options input to the data read function to avoid overwriting the input argument. The _options dictionary should be used in the data read function after merging all options. This procedure is not compulsory but it should be done if configuration file options and option abbreviations are to be used.

An optional function can also be provided by the module if it is capable of calculating additional coordinates to the ones returned in the data read function. For e.g. some derived spatial coordinates or flux coordinates can be added to an already existing data object with the following function:

```
def add_coordinate(data_object,
                   coordinates=None,
                   exp_id=None,
                   options=None)
```

This function takes a flap.DataObject and adds new coordinates named in the string list "coordinates". The arguments:

data_object:

The input data object.

coordinates:

A list of strings describing the requested new coordinates. In response the function should add these coordinates to the data object or raise an error.

exp_id:

If for the coordinate calculation another experiment ID should be used this argument is set and used instead of the exp_id in the data obejct. This can be the situation e.g. when no calibration data is available for the original experiment.

options:

A dictionary of options.

To make use of these tow functions in the flap environment the module should register them. The module should define a function named "register" the following way:

```
def register():
  flap.register_data_source('DATA_SOURCE_NAME',
                            get_data_func=get_data,
                            add_coord_func=add_coordinate)
```

The "DATA_SOURCE_NAME" string should be replaced by the desired data source name in FLAP. The two functions in the get_data_func and add_coord_func are the two functions described above. This register function will be called when the module is used. (See in the next section.)

Using data source modules

To make use of a data source module it should be first imported in the user program as any other module. The data read and coordinate add functions are integrated into FLAP by calling the register function of the. In the example below we will use module name testdata and data source name TESTDATA.

```
import testdata
testdata.register()

print(flap.list_data_sources())

d=flap.get_data('TESTDATA',
                exp_id=12345,
                name=['Signal-1','Signal-[20-23]'],
                options={'Scaling':'Volt'},\
                coordinates={'Time': [1,3]},
                object_name='TESTDATA_SIGNALS')
```

The above example code imports the testdata module and registers it in FLAP. the flap.list_data_sources function lists all actually registered data sources. The flap.get_data function is the general data read interface in FLAP. In the example it reads from the TESTDATA data source the data named Signal-1, Signal-20, ... Signal-23 in the tie interval 1-3s. The resulting data object will be stored in FLAP storage under name "TESTDATA_SIGNALS" ald also returned by the function. In the options it is requested to scale the data to volts.

When the flap.get_data function is called it first determines whether the requested data source is registered. If it is then the data read function of the module is called with the exp_id, name, options, coordinates arguments. If the coordinates argument is a dictionary as in the above example then it is converted to a flap.Coordinate object and the range ([1,3] in the example) is entered in the c_range field of it. This coordinate is passed to the module data rad function. If in the coordinates arguments already flap.Coordinate objects are listed than they are passed without modification. When the module data rad function returns the data object it is stored in FLAP storage if an object name is named. The shape, contents, coordinates of the data object is fully determined by the module data read function.

In the simplest form new coordinates can be requested for data object d the following way:

```
d.add_coordinate(coordinates=['Device x','Device y'])
```

This call requests the module which created the d data object to add new coordinates named Device x and Device y. The add_coordinate method also has a data_source, exp_id argument so as different data source or experiment can be used for coordinate calculation than the ones used for reading the data. This is useful e.g. when a separate flux coordinate mapping module is defined. It is also possible to add coordinates directly to a data object in FLAP storage:

```
flap.add_coordinate('TESTDATA_SIGNALS',
                    coordinates=['Device x','Device y'])
```

Plotting

FLAP can generate various plots from data objects in a simple way using the plot method of flap.DataObject or the flap.plot function. Graphics are created using the matplotlib package but a FLAP plot may consist of multiple matplotlib plots. Each FLAP plot is identified by a flap.PlotID class variable returned by the plot function/method. It is possible to set the actual plot ID to a previously created one and thus add more data to an already existing plot.

The hierarchy of graphics is the following:

- Matplotlib may use multiple "figures", each figure is a separate window in the computer.
- A figure may contain multiple subplots created by the Matplotlib subplot, GridSpec and similar functions.
- When the FLAP plot function/method is called it uses the current subplot and places its (possibly multiple) plots into the subplot area. This is done by creating sub-sub plots and storing their ID and other plot related data in flap.PlotID class variable.

FLAP uses the coordinate information in the data objects to create plots. The axes can be specified. Any coordinate can be selected for the axes and additionally data and constant values can also be assigned to various axes.

The FLAP plot function/method has default plot types and plot axes for various types of data objects, therefore in many cases it is enough to use d.plot() to prepare a reasonable plot of the data object d. A slicing and summing keyword argument allows to call the slice_data function from within the plot call so as a plot can be made of a slice of the data without the need to store the slice.