FLAP

S. Zoletnik et al.

Sep 29, 2020

CONTENTS:

Welcome to the FLAP documentation. This main page points to the start of a great journey in exploring the Fusion Library of Analysis Programs Python package.

CHAPTER

ONE

FLAP STYLEGUIDE

written by Gabor Csehlast modified: 1st December 2019

The purpose of this document is to give an overview of the coding style requirements for developing the main components of the Fusion Library of Analysis Programs (FLAP) package. (It is not a requirement for developing your own user programs, however, it is still suggested to use this coding style, even when just using FLAP. Or always.).

The coding style in FLAP package follows the PEP 8 style guide (see examples later), so if you are already know it's rules, you are good to go. Otherwise here are some useful tips about installing and using a style- and code-checking extension (flake8) in Visual Studio Code and Spyder.

1.1 Extensions/packages for style checking

Before getting to the actual code style guide, here is the list of packages/extensions, one can use to automatically check the style of one's coding.

1.1.1 Python extension for Visual Studio Code

Visual Studio Code is an open source, cross-platform code editor, which is widely used in many environments, e.g. it is available from the Anaconda Python distribution.

To install flake8 code analyzer, you have to install first the Python extension for VSCode:

1. Go to the Extension sidebar (or press Ctrl+Shift+X).
2. Type Python into the "Search Extension in Marketplace" search field.
3. Click on the green Install button.
4. That's it!

After installing the Python extension, a text will appear on the bottom sidebar of the VSCode window, which says Python a version number and 32/64 bit in the form of "Python 3.7.2 64-bit". By clicking on this text, you can select the Python version, you want to use for code analysis, and running your Python programs. This list contains (in principle) all the available Python installations on your machine, not just the ones, which are in the PATH variable.

To have your code checked, you need to install a linting module to your Python installation. The recommended module for this is flake8 (as mentioned before), which not only checks your code, but also analyze your programs and notices you about uninitialized/unused variables etc.

To install flake8, you just have to set VSCode settings accordingly. This means, that go to File –> Preferences –> Settings (Ctrl+,). Then type "python." in the search bar, and click on the "edit in settings.json" option (it appears multiple times, it does not matter, which one you click on). Then you have to insert the following lines or - if they are present - check, that the settings are the same as below.

```
{
    "python.linting.pylintEnabled": false,
    "python.linting.flake8Enabled": true,
    "python.linting.enabled": true,
    "python.linting.flake8Args": [
        "--max-line-length=120"
    ],
}
```

If you have no flake8 installed with your Python distribution, an automatic message will come up, offering the possibility to install this module automatically. You can choose either this to install flake8 (in this case, an embedded command line will come up inside the VSCode window and install the extension), or you can install it manually by using the

```
> python -m pip install flake8
```

In this manual case, take care that you use the python version you mean, and not another installation.

After installing flake8, the style and code analysis messages will appear in a dedicated block, if you click the error/warning sign icons in the navigation bar at the bottom of the screen.

1.1.2 Python with Spyder

Since Spyder is optimized solely around the Python language, Spyder is coming pylint preinstalled, so to have a good linter, you just have to switch it on. You have to check on the:

```
Tools --> Preferences --> Editor --> Code Instrospection/Analysis -->
Real-time code style analysis
```

checkbox, and you are good to go. After checking this checkmark and applying the settings, the warning/error messages will appear next to the line numbering with a yellow warning sign. Hovering the mouse over these warning signs will give you the exact message.

1.2 Style guide - a short summary

Since we follow the guidelines articulated in Python Enchancment Proposal 8 (PEP 8), if something is not clear and/or not written in this document, this is a good web page to start. Otherwise I try to give a short, but comprehensive summary about the ideas described there.

1. Use 4 spaces for indentation - no tabs!! (It can be easily set in modern code editors, even VI(m) has this possibility. It is called either “indent using spaces” or “soft tab”.)
2. No trailing spaces at the end of the line.
3. Line continuation:

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Add 4 spaces (an extra level of indentation) to distinguish
# arguments from the rest.
def long_function_name(
        var_one, var_two, var_three,
```

(continues on next page)

(continued from previous page)

```
        var_four):
    print(var_one)

# The closing brace/bracket/parenthesis on multiline constructs
# line up under the first non-whitespace character of the last
# line of list (or the first character of the line)
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]

result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

4. Line length: **79 characters** for code, **72 characters** for docstrings/comments. The original argument for this decision is:

   > Limiting the required editor window width makes it possible to have several files open side-by-side, and works well when using code review tools that present the two versions in adjacent columns.

   However, if this is not the case at your coding style (means you use one), it is allowed (as a local rule) to use **120 characters** for code and/or docstrings/comments.

5. Wrapping long lines:

   > The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation. Backslashes may still be appropriate at times. For example, long, multiple with-statements cannot use implicit continuation, so backslashes are acceptable:

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
     open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

6. Line breaks with binary operators (short: operator should be *before* the operand):

```
# Yes: easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

7. Blank lines.

   - Surround ***top-level functions*** and ***class definitions*** with ***two*** blank lines.
   - ***Method definitions*** inside a class are surrounded by a ***single*** blank line.
   - Use blank lines in functions, sparingly, to indicate logical sections.

8. Source file encoding: UTF-8. Set this every time. However, you should use English words and ASCII *characters* in those files. Except explicit test cases for non-ASCII characters and author names.

9. Naming conventions

- Names to avoid: 'l' (lowercase letter L), 'o' (lowercase letter 'oh'), 'I' (uppercase letter 'eye') as single-letter variable names. Reason: confusing.
- All the names should be ASCII compatibility (however, the character-encoding is UTF-8).
- ***Module*** names should be ***short, lowercase*** letters (like flap).
- ***Class names: CamelCase.***
- ***Function names: lowercase***, words separated by ***underscores.***
- Always use self for the first argument to instance methods.
- Always use cls for the first argument to class methods.
- At name collision (e.g. with a reserved keyword) use a trailing underscore. E.g. class_.

1.3 Logging

Instead of using print messages and verbose keywords etc. throughout the whole code, it it strongly advised to use Python's built-in logging system. It is capable of save the log messages in a stream, on the console or in a file (actually, the latter two are also kinds of streams) based on predefined criterions, e.g. severity. The logging system is easy-to-use and easy-to-config. Some examples are below.

- A logging.conf file for the logger setup.

```
[loggers]
keys=root,flapLogger

[handlers]
keys=consoleHandler,fileHandler

[formatters]
keys=fileFormatter,consoleFormatter

[logger_root]
level=DEBUG
handlers=consoleHandler

[logger_edvisLogger]
level=DEBUG
handlers=consoleHandler,fileHandler
qualname=edvisLogger
propagate=0

[handler_fileHandler]
class=logging.handlers.RotatingFileHandler
level=DEBUG
formatter=fileFormatter
args=('./flap.log', 'a', 5*1024*1024, 1)

[handler_consoleHandler]
class=StreamHandler
level=INFO
formatter=consoleFormatter
args=(sys.stdout,)

[formatter_fileFormatter]
```

(continues on next page)

(continued from previous page)

```
format=%(asctime)s - %(name)s - %(filename)s - line: %(lineno)s -
%(levelname)s - %(message)s
datefmt=

[formatter_consoleFormatter]
format=%(asctime)s - %(levelname)s - %(message)s
datefmt=
```

- Usage of the logger after setting up a logging.conf file.

```
import logging
import logging.config

# Loading the logger config file
logging.config.fileConfig('logging.conf')

# Create logger
logger = logging.getLogger('flapLogger')

logger.info("The FLAP logger facility has been initialized.")

# Doing some everyday work (only need info about that if we are in verbose mode)
print("Pam pam param...")
logger.debug("Here is some verbose, debug level logging message
              about 'Pam pam param...'")
```

1.4 Documentation strings

The Sphinx Python Documentation Generation system uses reStructuredText (RST) (for quick editing purposes, use this cheat sheet) to generate documentation for various formats include HTML, PDF, DOCX and various ebook formats.

It is capable of creating a whole documentation with separate documents (e.g. tutorials, detailed explanations, intentions etc.), but from the user point of view, the most important parts are the so-called docstrings. These are comment-like sections in the source code, where the user can on-the-fly describe the purpose and usage of the given part of the code. There are a few examples below.

A full code example (grabbed from the following link) is below:

```
"""
.. module:: useful_1
   :platform: Unix, Windows
   :synopsis: A useful module indeed.

.. moduleauthor:: Albert Example <albert@invalid.com>

"""


def public_fn_with_sphinxy_docstring(name, state=None):
    """This function does something.

    :param name: The name to use.
    :type name: str.
```

(continues on next page)

(continued from previous page)

```
    :param state: Current state to be in.
    :type state: bool.
    :returns:  int -- the return code.
    :raises: AttributeError, KeyError

    """
    return 0


class MyPublicClass(object):
    """We use this as a public class example class.

    You never call this class before calling :func:`public_fn_with_sphinxy_docstring`.

    .. note::

       An example of intersphinx is this: you **cannot** use :mod:`pickle` on this␣
↪class.

    """

    def __init__(self, foo, bar='baz'):
        """A really simple class.

        :param foo: We all know what foo does.
        :type foo: str.
        :param bar: Really, same as foo.
        :type bar: str.

        """
        self._foo = foo
        self._bar = bar
```

1.5 How to create the documentation

After properly written the documentation strings in the code, one can generate the full documentation of the project. To do this, you need to follow the steps below.

1. Install Sphinx:

   pip install -U Sphinx

1. Navigate into your ***documentation*** (not necessarily the same as the code) directory and type:

   sphinx-quickstart

1. You'll need to answer a few questions (see this link).
2. You got a basic documentation!

Of course, the documentation can be tweaked later by hand and manual pages can also be added later. PDFs and various ebook file types also can be generated, but the default is a HTML structure, where you can search and navigate.

If you want to update the existing (html) documentation, you should write (on Windows) the following command in the documentation source directory.

    make.bat html

Of course, if you need any other format, you can use that as well. For example

    make.bat latex

will generate LaTeX files for you, from which you can generate e.g. a pdf document.

    make.bat epub

will generate an ebook format of your documentation.

The documentation by default is written in ReStructured Text format (.rst file extension), but there is a possibility to extend Sphinx's capabilities to recognize MarkDown format (.md). For this you have to install the recommonmark module for your Python distribution with the following command.

    pip install recommonmark

CHAPTER

TWO

FLAP SOURCE CODE DOCUMENTATION

Contents

2.1 Config

Created on Wed Jan 23 21:45:43 2019

@author: Zoletnik

flap.config.**interpret_config_value**(*value_str*)

Determine the data type from the input string and convert. Conversions: 'True', 'Yes' –> True 'False', 'No' –> False Starting and ending with ' or " –> string If can be converted to int, float or complex –> converted numeric value Starting and ending with [] –> list If all the above fails keep as string

flap.config.**merge_options**(*default_options*, *input_options*, *data_source=None*, *section=None*)

Merges options dictionaries. Uses default options of function, input options of function and options read from config file from <section> section. If exp_id is set will also look for options in section Module exp_id for options starting with {section}. The precedence of options is:

default_options < section options < module options < input_options

INPUT: default_options: Default options in a function. This should contain all the possible options. input_options: Contents of options argument of function. Option keys can also be abbreviated. data_source: The data source of the measurement. (May be None) section: Name of the section in the config file related to the fnction. (May be None.)

Return value: The merged options dictionary. Abbreviated keys are expanded to full name.

2.2 Coordinate

Created on Wed Jan 23 09:44:50 2019

@author: Zoletnik

This is the coordinate description for FLAP

class flap.coordinate.**Coordinate**(*name=None*, *unit=None*, *mode=<flap.coordinate.CoordinateMode object>*, *shape=[]*, *start=None*, *step=None*, *c_range=None*, *values=None*, *value_index=None*, *value_ranges=None*, *dimension_list=[]*)

Class for the description of a mapping of coordinate values from an n-dimensional coordinate sample space to coordinates of an m-dimensional data matrix. Coordinate sample space is a rectangular equidistant point matrix, with equal steps in each dimension. For dimension i sample index is from 0...n-1 if shape[i] == n The sample

space in this coordinate description does not necessarily match the shape of any sub-matrix of the data object. If the shape is different then interpolation is done assuming the coordinate of the first(last) point of the coordinate matrix is the coordinate of the first(last) data point.

Coordinate can be anything described by name and unit. Standard coordinates: Time, Channel, Channel number, Device_x, Device_y, Device_z, Device_R, Device_Z, Device_phi,

Flux_r, Flux_theta, Flux_phi, Frequency, Time lag

See the description of variables in the __init__ function. The ranges and start-step pair of inputs are alternatives.

change_dimensions()
Return the list of dimensions of the data array along which this coordinate changes

data(*data_shape=None*, *index=None*, *options=None*)
Returns the coordinates, low and high limits for a sub-array of the data array in a DataObject.

index: list, tuple describing the elements in DataObject.data for which the coordinates are required. The length of the array should be identical to the number of dimensions of the data array. Elements can be a mixture of numbers, slice objects, lists, numpy arrays, integer iterators and ellipses. Examples for 3D array:

(. . . ,0,0) coordinates of the elements in the first row of the data array (slice(2,5),2,. . .)

data_shape: The shape of the data array (without slicing) for which coordinates are requested.

options: Dictionary with options for processing:

'Interpolation': 'Linear' (default, for non-equidistant axis when values shape is different from data shape)

'Change only': Return only the data for those dimensions where this coordinate changes. E.g. if it changes only along one dimension the output array will have 1 element in all other dimensions.

Return value: values, value_range_low, value range_high The low and high values are the absolute values not the difference from values

data_range(*data_shape=None*)
Returns the data range and the data range with errors for the coordinate. Both are lists.

dtype()
Return the data type of the coordinate. Returns standard Python types: str, int, float, complex, boolean, object

nochange_dimensions(*data_shape*)
Return the list of dimensions of the data array along which this coordinate changes

non_interpol(*data_shape*)
Return True if the shape of the coordinate description is the same as the sub-data-array for which it applies and self.value_index is None. In this case there is no need for interpolation, just copy coordinate values

Applicable only for non-equidistant case.

class flap.coordinate.**CoordinateMode**(*equidistant=True*, *range_symmetric=True*)
Class for storing mode flags of the coordinate description

class flap.coordinate.**Intervals**(*start*, *stop*, *step=None*, *number=None*)
A class to describe a series of intervals. Regular intevals are identical length ones repeating with a fixed step. Irregulars are just a list of start-stop values. For integer type values both the start and stop value is included. The optional number gives the number of ranges for the regular intervals.

interval_limits(*limits=None*, *partial_intervals=False*)
Return the range lower and upper limits as two numpy arrays. Limit the ranges within argument limits. (limits[0] < limits[1]) limits: 2 elements list with lower and upper limit

partial_intervals: Return also partial ranges which extend over limits. Their size will be truncated.

Raises a ValueError if there are no intervals within limit or other problem.

interval_number(*limits=None*, *partial_intervals=False*)
Return the number of intervals and the index of the start interval. Limit the ranges within argument limits. limits: 2 elements list with lower and upper limit

partial_intervals: Take into account also partial ranges which extend over limits.

Raises a ValueError in case of problem.

class flap.coordinate.**Unit**(*name=''*, *unit=''*)
Class for the unit of the data

title(*language='EN'*, *complex_txt=None*, *new_unit=None*)
Returns a title string which will be used in plotting. complex_txt: List of 2 numbers:

[0,0]: Amplitude [0,1]: Phase [1,0]: Real [1,1]: Imaginary

2.3 Data object

Created on Tue Jan 22 17:37:32 2019

@author: Zoletnik

class flap.data_object.**DataObject**(*data_array=None*, *error=None*, *data_unit=None*, *coordinates=None*, *exp_id=None*, *data_title=None*, *info=None*, *data_shape=None*, *data_source=None*)
This is the data object

add_coordinate(*coordinates=None*, *data_source=None*, *exp_id=None*, *options=None*)

This is a general coordinate conversion interface. Adds the requested coordinate(s) to the data_object.

INPUT: coordinates: List of coordinates to add. (string array) data_source: Optional data_source. Use this not the one in data_object exp_id: Optional exp_id. Use this not the one in data_object options: Dictionary of options.

Returns the modified data object. Note that the input data object remanins the same.

add_coordinate_object(*coordinate*, *index=None*)
Adds a flap.Coordinate instance to the list of coordinates If index is not set adds to the end of the list. Otherwise adds to the position shown by index. (0 is beginning of list.)

apsd(*coordinate=None*, *intervals=None*, *options=None*)
Auto power Spectral Density caclculation for the data object d. Returns a data object with the coordinate replaced by frequency or wavenumber. The power spectrum is calculated in multiple intervals (described by slicing) and the mean and variance will be returned.

INPUT: d: A flap.DataObject. coordinate: The name of the coordinate (string) along which to calculate APSD.

This coordinate should change only along one data dimension and should be equidistant. This and all other cordinates changing along the data dimension of this coordinate will be removed. A new coordinate with name Frequency/Wavenumber will be added. The unit will be derived from the unit of the coordinate (e.g., Hz cm-1, m-1)

intervals: Information of processing intervals.

If dictionary with a single key: {selection coordinate: description}) Key is a coordinate name which can be different from the calculation coordinate. Description can be flap.Intervals, flap.DataObject or a list of two numbers. If it is a data object with data name identical to the coordinate the error ranges of the data object will be used for interval. If the data name is not the same as coordinate a coordinate with the same name will be searched for in the data object and the value_ranges will be used fromm it to set the intervals.

If not a dictionary and not None is is interpreted as the interval description, the selection coordinate is taken the same as coordinate.

If None, the whole data interval will be used as a single interval.

options: Dictionary. (Keys can be abbreviated)

'Wavenumber' [True/False. Will use 2*Pi*f for the output coordinate scale, this is useful for] wavenumber calculation.

'Resolution': Output resolution in the unit of the output coordinate. 'Range': Output range in the unit of the output coordinate. 'Logarithmic': True/False. If True will create logarithmic frequency binning. 'Interval_n': Minimum number of intervals to use for the processing. These are identical

length intervals inserted into the input interval list. Default is 8.

'Error calculation' [True/False. Calculate or not error. Omitting error calculation] increases speed. If Interval_n is 1 no error calculation is done.

'Trend removal': Trend removal description (see also _trend_removal()). A list, string or None.

None: Don't remove trend. Strings:

'mean': subtract mean

Lists: ['poly', n]: Fit an n order polynomial to the data and subtract.

Trend removal will be applied to each interval separately.

'Hanning': True/False Use a Hanning window.

Return value: The new data object

This method is implemented by the _apsd function in spectcral_analysis.py

ccf(*ref=None*, *coordinate=None*, *intervals=None*, *options=None*)

N dimensional Cross Correlation Function or covariance calculation for the data object self taking d_ref as reference. If ref is not set self is used as reference, that is all CCFs are calculated within self. Calculates all CCF between all signals in ref and sel, but not inside self and ref. Correlation is calculated along the coordinate(s) listed in coordinate which should be identical for the to input data objects. Returns a data object with dimension number self.dim+ref.dim-len(coordinate). The coordinates are replaced by coordinate+' lag'. The CCF is calculated in multiple intervals (described by intervals) and the mean and variance will be returned.

INPUT: self: A flap.DataObject. ref: Another flap.DataObject coordinate: The name of the coordinate (string) along which to calculate CCF or a list of names.

Each coordinate should change only along one data dimension and should be equidistant. This and all other cordinates changing along the data dimension of these coordinates will be removed. New coordinates with name+' lag' will be added.

intervals: Information of processing intervals.

If dictionary with a single key: {selection coordinate: description}) Key is a coordinate name which can be different from the calculation coordinate. Description can be flap.Intervals, flap.DataObject or a list of two numbers. If it is a data object with data name identical to the coordinate the error ranges of the data object will be used for interval. If the data name is not the same as coordinate a coordinate with the same name will be searched for in the data object and the value_ranges will be used fromm it to set the intervals.

If not a dictionary and not None is is interpreted as the interval description, the selection coordinate is taken the same as coordinate.

If None, the whole data interval will be used as a single interval.

options: Dictionary. (Keys can be abbreviated) 'Resolution': Output resolution for each coordinate. (list of values or single value) 'Range': Output ranges for each coordinate. (List or list of lists) 'Interval_n': Minimum number of intervals to use for the processing. These are identical

length intervals inserted into the input interval list. Default is 8.

'Error calculation' [True/False. Calculate or not error. Omitting error calculation] increases speed. If Interval_n is 1 no error calculation is done.

'Trend removal': Trend removal description (see also _trend_removal()). A list, string or None.

None: Don't remove trend. Strings:

'mean': subtract mean

Lists: ['poly', n]: Fit an n order polynomial to the data and subtract.

Trend removal will be applied to each interval separately. At present trend removal can be applied to 1D CCF only.

'Hanning': True/False Use a Hanning window. 'Normalize': Normalize with autocorrelations, that is calculate correlation instead of covariance.

check()
Does a consistency check for the data object and raises errors if problems found.

coordinate(*name*, *index=None*, *options=None*)
Returns the coordinates of a subarray of the data array. name: Coordinate name (string) index: The indexes into the data array (tuple with various elements, see Coordinate.data()) options: The same options as for Coordinate.data()

returns 3 np.arrays:

data: the coordinates. The number of dimension is the same as the dimension of the data array, but the number of elements are taken from index.

data_low: low ranges, same shape as data. None if no range data is present data_high: high ranges, same shape as data. None if no range data is present

coordinate_change_dimensions(*name*)
Return the list of dimensions of the data array along which the named coordinate changes

coordinate_change_indices(*name*)

Returns the indices to the data array for which the coordinate changes. The returned value is a tuple of indices, the number of elements equals the dimension of the data. This can be directly used to get the coordinate values using coordinate()

name: Coordinate name (string)

coordinate_names()
Returns a list with the coordinate names.

coordinate_nochange_dimensions(*name*)
Return the list of dimensions of the data array along which the named coordinate changes

coordinate_range(*name*, *index=Ellipsis*)
Returns the ranges of a coordinate. name: Coordinate name (string)

Returns the data range and the data range with errors for the coordinate. Both are lists.

cpsd(*ref=None*, *coordinate=None*, *intervals=None*, *options=None*)
Complex Cross Power Spectrum calculation for the data object self taking d_ref as reference. If self is not set d is used as reference, that is all spectra are calculated within self. Calculates all spectra between all signals in ref and self, but not inside self and ref. self and ref both should have the same equidistant coordinate with equal sampling points. Returns a data object with dimension number self.dim+ref.dim-1. The coordinate is replaced by frequency or wavenumber. The spectrum is calculated in multiple intervals (described by intervals) and the mean and variance will be returned.

INPUT: self: A flap.DataObject. ref: Another flap.DataObject coordinate: The name of the coordinate (string) along which to calculate CPSD.

This coordinate should change only along one data dimension and should be equidistant. This and all other cordinates changing along the data dimension of this coordinate will be removed. A new coordinate with name Frequency/Wavenumber will be added. The unit will be derived from the unit of the coordinate (e.g., Hz cm-1, m-1)

intervals: Information of processing intervals.

If dictionary with a single key: {selection coordinate: description}) Key is a coordinate name which can be different from the calculation coordinate. Description can be flap.Intervals, flap.DataObject or a list of two numbers. If it is a data object with data name identical to the coordinate the error ranges of the data object will be used for interval. If the data name is not the same as coordinate a coordinate with the same name will be searched for in the data object and the value_ranges will be used fromm it to set the intervals.

If not a dictionary and not None is is interpreted as the interval description, the selection coordinate is taken the same as coordinate.

If None, the whole data interval will be used as a single interval.

options: Dictionary. (Keys can be abbreviated)

'Wavenumber' [True/False. Will use 2*Pi*f for the output coordinate scale, this is useful for] wavenumber calculation.

'Resolution': Output resolution in the unit of the output coordinate. 'Range': Output range in the unit of the output coordinate. 'Logarithmic': True/False. If True will create logarithmic frequency binning. 'Interval_n': Minimum number of intervals to use for the processing. These are identical

length intervals inserted into the input interval list. Default is 8.

'Error calculation' [True/False. Calculate or not error. Omitting error calculation] increases speed. If Interval_n is 1 no error calculation is done.

'Trend removal': Trend removal description (see also _trend_removal()). A list, string or None.

None: Don't remove trend. Strings:

'mean': subtract mean

Lists: ['poly', n]: Fit an n order polynomial to the data and subtract.

Trend removal will be applied to each interval separately.

'Hanning': True/False Use a Hanning window. 'Normalize': Calculate coherency instead of crosspower

detrend(*coordinate=None*, *intervals=None*, *options=None*)

Trend removal. INPUT:

coordinate: The x coordinate for the trend removal. intervals: Information of processing intervals.

If dictionary with a single key: {selection coordinate: description}) Key is a coordinate name which can be different from the calculation coordinate. Description can be flap.Intervals, flap.DataObject or a list of two numbers. If it is a data object with data name identical to the coordinate the error ranges of the data object will be used for interval. If the data name is not the same as coordinate a coordinate with the same name will be searched for in the data object and the value_ranges will be used fromm it to set the intervals.

If not a dictionary and not None is is interpreted as the interval description, the selection coordinate is taken the same as coordinate.

If None, the whole data interval will be used as a single interval.

options:

'Trend removal': Trend removal description (see also _trend_removal()). A list, string or None. None: Don't remove trend. Strings:

'Mean': subtract mean

Lists:

['Poly', n]: Fit an n order polynomial to the data and subtract. Trend removal will be applied to each interval defined by slicing separately.

Return value: The data object with the trend removed data.

error_value(*options=None*)

Returns a data object with the error of self in it. options: 'High': Use high error if error is asymmetric

'Low': Use low error is error is asymmetric

filter_data(*coordinate=None*, *intervals=None*, *options=None*)

1D Data filter. INPUT:

coordinate: The x coordinate for the trend removal. intervals: Information of processing intervals.

If dictionary with a single key: {selection coordinate: description}) Key is a coordinate name which can be different from the calculation coordinate. Description can be flap.Intervals, flap.DataObject or a list of two numbers. If it is a data object with data name identical to the coordinate the error ranges of the data object will be used for interval. If the data name is not the same as coordinate a coordinate with the same name will be searched for in the data object and the value_ranges will be used fromm it to set the intervals.

If not a dictionary and not None is is interpreted as the interval description, the selection coordinate is taken the same as coordinate.

If None, the whole data interval will be used as a single interval.

options:

'Type' : None: Do nothing. 'Int': Single term IIF filter, like RC integrator. 'Diff': Single term IIF filter, like RC differentiator. 'Bandpass', 'Lowpass', 'Highpass': Filters designed by scipy.signal.iirdesign.

The filter type is in 'Design'

Bandpass: f_low - f_high Lowpass: - f_high Highpass: - f_low

'Design': The design type of the bandpass, lowpass or highpass filter. ('Elliptic', 'Butterworth, 'Chebyshev I', 'Chebyshev II', 'Bessel') The numpy.iirdesign function is used for generating the filter. Setting inconsistent parameters can cause strange results. E.g. too high attenuation at too low frequency relative to the smapling frequency can be a problem.

'f_low', 'f_high': Cut on/off frequencies. (Middle between passband and stopband edge.) 'Steepness': Difference between passband and stopband edge frequencies as a fraction

of the middle frequency.

'Loss': The maximum loss in the passband in dB 'Attenuation': The minimum attenuation in the stopband dB

'Tau': time constant for integrator/differentiator (in units of the coordinate) 'Power': Calculate square of the signal after filtering. (boolean) 'Inttime': Integration time after power calculation. (in units of oordinate)

Return value: The data object with the filtered data.

get_coordinate_object(*name*)

Returns the Coordinate class having the given name. The returned object is a link not a copy.

index_from_coordinate(*name*, *coord_values*)

Returns the closest data indices of the coordinate values. Coordinates should change only along one dimension. It is assumed that coordinates change monotonically.

INPUT: name: Coordinate name (string) coord_values: The coordinate values to convert to indices.

(numpy array or list is scalar)

Return value: Returns the indices in the same format as the input coordinates.

plot(*axes=None*, *slicing=None*, *slicing_options=None*, *summing=None*, *options=None*, *plot_type=None*, *plot_options=None*, *plot_id=None*)

Plot a data object. axes: A list of coordinate names (strings). They should be one of the coordinate

names in the data object or 'Data' They describes the axes of the plot. If the number of axes is less than the number required for the plot, 'Data' will be added. If no axes are given default will be used depending on the plot type. E.g. for x-y plot the default first axis is the firs coordinate, the second axis is 'Data'

plot_type: The plot type (string) 'xy': Simple 1D plot. Default axes are: first coordinate, Data 'multi xy': In case of 2D data plots 1D curves with vertical shift

Default x axis is first coordinate, y axis is Data The signals are named in the label with the 'Signal name' coordinate or the one named in options['Signal name']

plot_options: Dictionary. Will be passed over to the plot call. slicing, summing: arguments for slice_data. Slicing will be applied before plotting. options:

Matplotlib options like xtitle, xlimit, etc Plot options:

'All points' True or False default is False. If True will plot all points otherwise will use the sample_to_plot function

'Error' True: Plot all error bars (default: True) False: Do not plot errors number > 0: Plot this many error bars in plot

'Y separation' Vertical separation of curves in multi xy plot. For linear scale this will be added to consecutive cureves. For Log scale consecutive curves will be multiplied by this.

'Log x' : Logscale X axis 'Log y' : Logscale Y axis

proc_interval_limits(*coordinate*, *intervals=None*)

Determine processing interval limits, both in coordinates and data indices. This is a helper routine for all functions which do some calculation as a function of one coordinate and allow processing only a set of intervals instead of the whole dataset. INPUT:

coordinate: Name of the coordinate along which calculation will be done. This must change only along a single data dimension.

intervals: Information of processing intervals.

If dictionary with a single key: {selection coordinate: description}) Key is a coordinate name which can be different from the calculation coordinate. Description can be flap.Intervals, flap.DataObject or a list of two numbers. If it is a data object with data name identical to the coordinate the error ranges of the data object will be used for interval. If the data name is not the same as coordinate a coordinate with the same name will be searched for in the data object and the value_ranges will be used fromm it to set the intervals.

If not a dictionary and not None is is interpreted as the interval description, the selection coordinate is taken the same as coordinate.

If None, the whole data interval will be used as a single interval.

Return value: calc_int, calc_int_ind, sel_int, sel_int_ind Each return value is a list of numpy arrays: [start, end] The calc_xxx values are for the calculation coordinate, the sel_xxx are for the selection coordinate. xxx_int is in coordinate values, xxx_int_ind is in data index values which follows the Python convention, end index is not included. The index start indices will be always smaller than the end indices.

save(*filename*, *protocol=3*)

Save the data object to a binary file using pickle. Use load to read the object. Filename: The name of the output file protocol: The pickle protocol to use

slice_data(*slicing=None*, *summing=None*, *options=None*)

Slice (select areas) from the data object along one or more coordinates. Return the sliced object.

slicing:

Dictionary with keys referring to coordinates in the data object.

Values can be:

a:SIMPLE SLICE: cases when closest value or interpolated value is selected. a1 slice objects, range objects, scalars, lists, numpy array. a2 flap.DataObjects without error and with data unit.name equal to

the coordinate

a3 flap.DataObject with the name of one coordinate equal to the dictionary key without having value_ranges values.

a4 flap.Intervals objects with one interval

b: MULTI SLICE: Various range selection objects. In this case ranges are selected and a new dimension is added to the data array (only of more than 1 interval is selected) going through the intervals. If intervals are of different length the longest will be used and missing elements filled with float('nan'). Two new coordinates are added: "<coordinate> in interval", "<coordinate> interval" b1 flap.Intervals objects with more than one interval b2 flap.DataObjects with data unit.name equal to the slicing coordinate. The error

values give the intervals.

b3 flap.DataObject with the name of one coordinate equal to the slicing coordinate. The value_ranges select the intervals.

If range slicing is done with multiple coordinates which have common element in the dimension list they will be done in one step. Otherwise the slicing is done sequentially.

summing: Summing is applied to the sliced data. It processes data along one coordinate and the result is a scalar. This way summing reduces the number of dimensions. Dictionary with keys referring to coordinates and values as processing strings. If the processed coordinate changes along multiple dimensions those dimensions will be flattened.

For mean and avereage data errors are calculated as error of independent variables, that is taking the square root of the squared sum of errors. For coordinates the mean of the ranges is taken.

Processing strings are the following:

None: Nothing to be done in this dimension

'Mean' : take mean of values in selection/coordinate 'Sum' : take sum of values in selection/coordinate 'Min' : take the minimum of the values/coordinate 'Max' : take the maximum of the values/coordinate

options: 'Partial intervals' (bool). If true processes intervals which extend over the coordinate limits.

If false only full intervals are processed.

'Slice type': 'Simple': Case a above: closest or interpolated values are selected, dimensions

are reduced or unchanged.

'Multi': Case b above: multiple intervals are selected and their data is placed into new dimension.

None: Automatically select. For slicing data in case b multi slice, otherwise simple

'Interpolation: 'Closest value' 'Linear'

'Regenerate coordinates': True/False (defaultL True) If True and summing is done then looks for pairs of coordinates 'Rel. <coord> in int(<coord1>)' 'Start <coord> in int(<coord1>)'. If such pairs are found and they change on the same dimension or one of them is constant then coordinate <coord> is regenerated and these are removed.

slicing_to_intervals(*slicing*)
Convert a multi-slicing description to an Intervals object. For possibilities see DataObject.slice_data().

to_intervals(*coordinate*)
Create an Intervals class object from either the data error ranges or the coordinate value ranges.

class flap.data_object.**FlapStorage**
This class is for data and data source information storage

find_data_objects(*name*, *exp_id='*'*)
" Find data objects in flap storage.

Returns name list, exp_ID list

flap.data_object.**abs_value**(*object_name*, *exp_id='*'*, *output_name=None*)
Absolute value

flap.data_object.**add_coordinate**(*object_name*, *coordinates=None*, *exp_id='*'*, *options=None*, *output_name=None*)
This is the function to add coordinates to a data object in flap storage. This function is an interface to the add_coordinate method of flap.DataObject
INPUT: object_name, exp_ID: These identify the data object in the storage coordinates: List of new coordinates (string list) output_name: The name of the new data object in the storage. If None

the input will be overwritten

options: Dictionary

'exp_ID': Use this exp_id for calculating coordinates instead of the one in the data object

'data_source' [Use this data source instead of the one in the] data object.

Other elements of options are passed over to flap.add_coordinate()

flap.data_object.**add_data_object**(*d*, *object_name*)
Add a data object to the flap storage

flap.data_object.**apsd**(*object_name*, *exp_id='*'*, *output_name=None*, *coordinate=None*, *intervals=None*, *options=None*)
Auto Power Spetctrum for an object in flap storage. This is a wrapper for DataObject.apsd() If output name is set the APSD object will be written back to the flap storage under this name. If not set the APSD object will be written back with its original name.

flap.data_object.**ccf**(*object_name*, *ref=None*, *exp_id='*'*, *ref_exp_id='*'*, *output_name=None*, *coordinate=None*, *intervals=None*, *options=None*)

Cross Corrrelation Function or covariance calculation between two objects in flap storage. This is a wrapper for DataObject.ccf() If output name is set the CPSD object will be written back to the flap storage under this name.

flap.data_object.**cpsd**(*object_name*, *ref=None*, *exp_id='*'*, *output_name=None*, *coordinate=None*, *intervals=None*, *options=None*)

Cross Power Spectrum between two objects in flap storage. (ref can also be a data object.) This is a wrapper for DataObject.cpsd() If output name is set the CPSD object will be written back to the flap storage under this name.

flap.data_object.**delete_data_object**(*object_name*, *exp_id='*'*)

Delete data object(s) from the flap storage

flap.data_object.**detrend**(*object_name*, *exp_id='*'*, *output_name=None*, *coordinate=None*, *intervals=None*, *options=None*)

DETREND signal(s) INPUT:

object_name: Name of the object in flap storage (string) exp_id: Experiment ID output_name: Name of the output object. If set result will be stored under this name. coordinate: The coordinate for the detrend. If necessary data will be fitted with this coordinate

as x value.

intervals: Information of processing intervals.

If dictionary with a single key: {selection coordinate: description}) Key is a coordinate name which can be different from the calculation coordinate. Description can be flap.Intervals, flap.DataObject or a list of two numbers. If it is a data object with data name identical to the coordinate the error ranges of the data object will be used for interval. If the data name is not the same as coordinate a coordinate with the same name will be searched for in the data object and the value_ranges will be used fromm it to set the intervals.

If not a dictionary and not None is is interpreted as the interval description, the selection coordinate is taken the same as coordinate.

If None, the whole data interval will be used as a single interval.

options:

'Trend removal': Trend removal description (see also _trend_removal()). A list, string or None.
None: Don't remove trend. Strings:

'mean': subtract mean

Lists:

['poly', n]: Fit an n order polynomial to the data and subtract. Trend removal will be applied to each interval defined by slicing separately.

Return value: The resulting data object.

flap.data_object.**error_value**(*object_name*, *exp_id='*'*, *output_name=None*, *options=None*)

Returns a data object with the error of self in it. options: 'High': Use high error if error is asymmetric
'Low': Use low error is error is asymmetric

flap.data_object.**filter_data**(*object_name*, *exp_id='*'*, *output_name=None*, *coordinate=None*, *intervals=None*, *options=None*)

1D Data filter. INPUT:

object_name: Name of the object in flap storage (string) exp_id: Experiment ID output_name: Name of the output object. If not set object_name will be used. coordinate: The x coordinate for the trend removal. intervals: Information of processing intervals.

If dictionary with a single key: {selection coordinate: description})
Key is a coordinate name which can be different from the calculation coordinate. Description can be flap.Intervals, flap.DataObject or a list of two numbers. If it is a data object with data name identical to the coordinate the error ranges of the data object will be used for interval. If the data name is not the same as coordinate a coordinate with the same name will be searched for in the data object and the value_ranges will be used fromm it to set the intervals.

If not a dictionary and not None is is interpreted as the interval
description, the selection coordinate is taken the same as coordinate.

If None, the whole data interval will be used as a single interval.

options:

'Type' : None: Do nothing. 'Int': Single term IIF filter, like RC integrator. 'Diff': Single term IIF filter, like RC differentiator. 'Bandpass', 'Lowpass', 'Highpass': Filters designed by scipy.signal.iirdesign.

The filter type is in 'Design'

Bandpass: f_low - f_high Lowpass: - f_high Highpass: - f_low

'Design': The design type of the bandpass, lowpass or highpass filter.
('Elliptic', 'Butterworth, 'Chebyshev I', 'Chebyshev II', 'Bessel') The numpy.iirdesign function is used for generating the filter. Setting inconsistent parameters can cause strange results. E.g. too high attenuation at too low frequency relative to the smapling frequency can be a problem.

'f_low', 'f_high': Cut on/off frequencies. (Middle between passband and stopband edge.) 'Steepness': Difference between passband and stopband edge frequencies as a fraction

of the middle frequency.

'Loss': The maximum loss in the passband in dB 'Attenuation': The minimum attenuation in the stopband dB

'Tau': time constant for integrator/differentiator (in units of the coordinate) 'Power': Calculate square of the signal after filtering. (boolean) 'Inttime': Integration time after power calculation. (in units of oordinate) Return value: The data object with the filtered data.

`flap.data_object.`**`find_data_objects`**(*name*, *exp_id='*'*)
Find data objects in flap storage.

Returns list of names, list of expID.s

`flap.data_object.`**`get_addcoord_function`**(*data_source*)
Return the add_coord function object for a given data source

`flap.data_object.`**`get_data`**(*data_source*, *exp_id=None*, *name=None*, *no_data=False*, *options=None*, *coordinates=None*, *object_name=None*)
This is a general data read interface. It will call the specific data read interface for the registered data sources. data_source: The name of the data source (string) exp_id: Experiment ID coordinates: Two options:

1. List of flap.Coordinate objects. Thes can precisely describe which part of the data to read
2. **Dictionary. Each key is a coordinate name, the values can be**

   - A list of two elements (describes a range in the coordinate).
   - A single element. Will be converted into a list with two identical elements

The dictionary will be converted to a list of flap.Coordinate objects and the data source modulee will get that.

name: The name of the data to get no_data: Set to True to check data but do not read options: Module specific options object_name: the name of the data object in flap storage where data will be placed.

Return value is the data object.

flap.data_object.**get_data_function**(*data_source*)
Return the get data function object for a given data source

flap.data_object.**get_data_object**(*object_name*, *exp_id='*'*)
Return a data object from the flap storage A coppy is returned not the object in the storage.

flap.data_object.**get_data_object_ref**(*object_name*, *exp_id='*'*)
Return a data object reference from the flap storage.

flap.data_object.**imag**(*object_name*, *exp_id='*'*, *output_name=None*)
Real value. (Does nothing for real data)

flap.data_object.**list_data_objects**(*name='*'*, *exp_id='*'*, *screen=True*)
Prepare a printout of data objects is flap storage or the listed data objects. name: name (with wildcards) or list of data objects exp_id: exp id for name screen: (bool) If True print to screen

Return value: The text

flap.data_object.**list_data_sources**()
Return a list of registered data source names as a list.

flap.data_object.**load**(*filename*, *options=None*)
Loads data saved with flap.save() If the data in the file were written from the flap storage it will also be loaded there, unless no_storage is set to True.

INPUT: filename: Name of the file to read. options: "No storage": (bool) If True don't store data in flap storage just return a list of them.

Return value: If data was not written from flap storage the original object will be returned. If it was written out from flap storage a list of the read objects will be returned..

flap.data_object.**phase**(*object_name*, *exp_id='*'*, *output_name=None*)
Phase

flap.data_object.**plot**(*object_name*, *exp_id='*'*, *axes=None*, *slicing=None*, *summing=None*, *options=None*, *plot_type=None*, *plot_options=None*, *plot_id=None*, *slicing_options=None*)
plot function for an object in flap storage. This is a wrapper for DataObject.plot()

flap.data_object.**real**(*object_name*, *exp_id='*'*, *output_name=None*)
Real value. (Does nothing for real data)

flap.data_object.**register_data_source**(*name*, *get_data_func=None*, *add_coord_func=None*)
Register a new data source name and the associated functions.

flap.data_object.**save**(*data*, *filename*, *exp_id='*'*, *options=None*, *protocol=3*)
Save one or more flap.DataObject-s using pickle.

INPUT:

data: If flap.DataObject than save this. If string or string list then find these data objects in flap storage and save them. Will also use exp_id to select data objects. These data objects can be restored into flap storage using load. If any other object save it.

exp_id: Experiment ID to use in conjuction with data of it is a string. options: None at present protocol: The protocol to use filename: Name of the file to save to.

flap.data_object.**slice_data**(*object_name*, *exp_id='*'*, *output_name=None*, *slicing=None*, *summing=None*, *options=None*)
slice function for an object in flap storage. This is a wrapper for DataObject.slicec_data() If output name is set the sliced object will be written back to the flap storage under this name.

2.4 Plot

Created on Sat May 18 18:37:06 2019

@author: Zoletnik @coauthor: Lampert

class flap.plot.**PddType**(*value*)
An enumeration.

class flap.plot.**PlotDataDescription**(*data_type=None*, *data_object=None*, *value=None*)
Plot axis description for use in PlotID() and plot(). data_object: The data object from which the data for this coordinate originates. This may
be None if the data is constant.

data_type: PddType

PddType.Coordinate: A coordinate in data_object. self.value is a flap.Coordinate object.

PddType.Constant: A float constant, stored in value. PddType.Data: The data in self.data_object.

value: Value, see above

flap.plot.**axes_to_pdd_list**(*d*, *axes*)
Convert a plot() axes parameter to a list of PlotAxisDescription and axes list for PlotID d: data object axes: axes parameter of plot()

return pdd_list, ax_list

flap.plot.**get_plot_id**()
Return the current PlotID or None if no act plot.

flap.plot.**sample_for_plot**(*x*, *y*, *x_error*, *y_error*, *n_points*)
Resamples the y(x) function to np points for plotting. This is useful for plotting large arrays in a way that short outlier pulses are still indicated. The original function is divided into np equal size blocks in x and in each block the minimum and maximum is determined. The output will contain 2*np number or points. Each consecutive point pair contains the minimum and maximum in a block, the time is the centre time of the block.

x: Input x array. y: input y array. np: Desired number of blocks. This would be a number larger than the number of pixels in the plot in the horizontal direction.

Return values: x_out, y_out If the box length is less than 5 the original data will be returned.

flap.plot.**set_plot_id**(*plot_id*)
Set the current plot.

2.5 Select

Created on Fri Apr 12 19:23:25 2019

@author: Zoletnik

`flap.select.`**`select_intervals`**(*object_descr*, *coordinate=None*, *exp_id='*'*, *intervals=None*, *options=None*, *plot_options=None*, *output_name=None*)

Select intervals from data interactively. INPUT:

object_descr: If a data object: Plot this data object to select. String: Will be interpreted as a data object name in flap storage.

exp_id: The exp_id if dobject_descr is a string

intervals: Information of processing intervals.

If dictionary with a single key: {selection coordinate: description}) Key is a coordinate name which can be different from the calculation coordinate. Description can be flap.Intervals, flap.DataObject or a list of two numbers. If it is a data object with data name identical to the coordinate the error ranges of the data object will be used for interval. If the data name is not the same as coordinate a coordinate with the same name will be searched for in the data object and the value_ranges will be used fromm it to set the intervals.

If not a dictionary and not None it is interpreted as the interval description, the selection coordinate is taken the same as coordinate.

If None, the whole data interval will be used as a single interval.

coordinate: The name of the coordinate to use for x axis. (string) This will be the coordinate of the selection.

plot_options: Passed to plot(). options: Dictionary of options:

'Select': 'Start': Select start of intervals. (Needs Lenght to be set.) 'End': Select end of intervals. (Needs Lenght to be set.) 'Full': Select start and end of interval. 'Center': Select center of interval. None: No interactive selection

'Length': Length of intervals. 'Event' : Dictionary describing events to search for. A reference time will be

determined for each event and a Length interval will be selected symmetrically around it. Trend removal and/or filtering should be done before calling this function. 'Type': 'Maximum' or 'Minimum':

Will look for signal pieces above/below threshold and calculate maximum place of signal in this piece.

'Max-weight' or 'Min-weight': Same as Maximum and Minimum but selects center of gravity for signal piece.

In each of the above cases the interval will be 'Length' long around the event. 'Above', 'Below':

The intervals will be where the signal is above or below the threshold.

'Start delay', 'End delay': For the Above and Below events the start and end delay of the interval in the coordinate units.

'Threshold': The threshold for the event. 'Thr-type' Threshold type:

'Absolute': Absolute signal value 'Sigma': Threshold times sigma

output_name: Output object name in flap storage.

Return value: A data object. Data name is the coordinate name. The error gives the intervals.

2.6 Spectral analysis

Created on Thu Mar 7 10:34:49 2019

@author: Zoletnik

Spectral analysis tools fro FLAP

flap.spectral_analysis.**trend_removal_func**(*d*, *ax*, *trend*, *x=None*, *return_trend=False*, *return_poly=False*)

This function makes the _trend_removal internal function public

2.7 Test data

2.8 Tools

Created on Wed Jan 23 13:18:25 2019

@author: Zoletnik

Tools for the FLAP module

flap.tools.**chlist**(*chlist=None*, *chrange=None*, *prefix=''*, *postfix=''*)

Creates a channel (signal) list name from a prefix, postfix a channel list and a list of channel ranges

flap.tools.**del_list_elements**(*input_list*, *indices*)

delete elements from a list

flap.tools.**expand_matrix**(*mx*, *new_shape*, *dim_list*)

Insert new dimensions to a matrix so as it has <new shape> shape. The original dimensions are at dim_list dimensions

Input: mx: The matrix with arbitrary dimensions. new_shape: This will be the new shape dim_list: This is a list of dimensions where mx is in the output

matrix. len(dim_list) == mx.ndim

flap.tools.**find_str_match**(*value*, *options*)

Given value string and a list of possibilities in the list of strings option find matches assuming value is an abbreviation. Return ValueError if no match or multiple match is found. If one match is found return the matching string

flap.tools.**flatten_multidim**(*mx*, *dim_list*)

Flatten the dimensions in dim_list to dim_list[0] Returns the modified matrix and a mapping from the original to the new dimension list. The mapping will be None for the flattened dimension in dim_list even if flattening was not done. The dimension numbers in the dimension list assume that the flattened dimensions are removed.

flap.tools.**grid_to_box**(*xdata*, *ydata*)

Given 2D x and y coordinate matrices create box coordinates around the points as needed by matplotlib.pcolomesh. xdata: X coordinates. ydata: Y coordinates. In both arrays x direction is along first dimension, y direction along second dimension. Returns xbox, ybox.

flap.tools.**move_axes_to_end**(*mx_orig*, *axes*)
Moves the listed axes to the end.

flap.tools.**move_axes_to_start**(*mx_orig*, *axes*)
Moves the listed axes to the start axes.

flap.tools.**multiply_along_axes**(*a1_orig*, *a2_orig*, *axes*, *keep_a1_dims=True*)
Multiplies two arrays along given axes. INPUT:
a1_orig: Array 1. a2_orig: Array 2. axes: List of two axis numbers or list of two lists of axis numbers keep_1_dims: (bool)

If True: The output array has dimensions of a1 followed by a2 with the common dims removed If False: The output array has the a1 dimensions without common dims then the common dims

followed by a2 with the common dims removed

Return values:

a, axis_source, axis_number a: An array with dimension number a1.dim+a2.dim-1. axis_source: List of integers telling the source array for each output axis (0 or 1) axis_number: Axis numbers in the arrays listed in axes_source

flap.tools.**select_signals**(*signal_list*, *signal_spec*)
Selects signals from a signal list following signal specifications.

signal_list: List of strings of possible signal names
signal_spec: List of strings with signal specifications including wildcards Normal Unix file name wildcards are accepted and extended with [<num>-<num>] type expressions so as e.g. a channel range can be selected.
Returs select_list, select_index select_list: List of strings with selected signal names select_index: List of indices to signal list of the selected signals
Raises ValueError if there is no match for one specification

flap.tools.**submatrix_index**(*mx_shape*, *index*)
Given an arbitrary dimension matrix with shape mx_shape the tuple to extract a submatrix is created and returned. The elements in each dimension are selected by index.
Input: mx_shape: Shape of the matrix index: Tuple or list of 1D numpy arrays. The length should be equal to the length of mx_shape. Each array contains the indices for the

corresponding dimension.
Return value: A tuple of index matrices. Each index matrix has the same shape as described by index. Each matrix contains the indices for one dimension of the matrix. This tuple can be directly used for indexing the matrix.

flap.tools.**unify_list**(*list1*, *list2*)
Returns list with elements present in any of the two lists. Output list is sorted.

CHAPTER

THREE

USAGE OF FLAP (TIPS/TRICKS/FAQ)

This page contains all the techniques the FLAP developers or other FLAP users find useful for newcomers and even experienced FLAP users.

Contents

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f