opticalib

Release 1.0.0

opticalib contributors

Sep 29, 2025

CONTENTS

Welcome to the opticalib docs.

CHAPTER

ONE

ABOUT

This site documents the `opticalib` Python library. The docs are generated from the code's docstrings using Sphinx.

- Start at the API Reference for module/class/function docs.
- Contribute by improving docstrings in the source code.

CHAPTER

TWO

API REFERENCE

The API docs are generated automatically from the source using autodoc and autosummary.

opticalib	Author(s) Pietro Ferraiuolo : written in 2025

2.1 opticalib

2.1.1 Author(s)

- Pietro Ferraiuolo : pietro.ferraiuolo@inaf.it

2.1.2 Description

opticalib is a package for the control of laboratory instrumentations, like Interferometers and Deformable Mirrors. It also provides tools for the analysis of wavefronts and images.

2.1.3 How to Use:

`` `python > import opticalib > interf = opticalib.PhaseCam('193.206.155.218', 8011) > img = interf.acquire_map() ` ``

opticalib.**create_configuration_file**(*path=''*, *data_path=False*)

Create a configuration file in the specified path.

Parameters

- **path** (*str*) – The path to the configuration file.
- **data_path** (*str | bool*) – The path to the data folder. If True, it will be set to the same directory as the configuration file. If False, it will not be set. If a string, a path must be provided, and the *data_path* will be set to that path.
- **load** (*bool*) – If True, the configuration file will be loaded after creation, the folder tree created (if not already) and all the paths updated.

Return type

None

opticalib.**getFileList**(*tn=None*, *fold=None*, *key=None*)

Search for files in a given tracking number or complete path, sorts them and puts them into a list.

Parameters

- **tn** (*str*) – Tracking number of the data in the OPDImages folder.

- **fold** (*str, optional*) – Folder in which searching for the tracking number. If None, the default folder is the OPD_IMAGES_ROOT_FOLDER.
- **key** (*str, optional*) – A key which identify specific files to return

Returns

- **fl** (*list of str*) – List of sorted files inside the folder.
- *How to Use it*
- *———-*
- *If the complete path for the files to retrieve is available, then this function*
- *should be called with the 'fold' argument set with the path, while 'tn' is*
- *defaulted to None.*
- **In any other case, the tn must be given** (*it will search for the tracking*)
- *number into the OPDImages folder, but if the search has to point another*
- *folder, then the fold argument comes into play again. By passing both the*
- *tn (with a tracking number) and the fold argument (with only the name of the*
- *folder) then the search for files will be done for the tn found in the*
- *specified folder. Hereafter there is an example, with the correct use of the*
- *key argument too.*

Return type

list[str]

Examples

Here are some examples regarding the use of the 'key' argument. Let's say w e need a list of files inside ''tn = '20160516_114916' '' in the IFFunctions folder.

```
>>> iffold = 'IFFunctions'
>>> tn = '20160516_114916'
>>> getFileList(tn, fold=iffold)
['.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/cmdMatrix.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0000.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0001.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0002.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0003.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/modesVector.fits']
```

Let's suppose we want only the list of 'mode_000x.fits' files:

```
>>> getFileList(tn, fold=iffold, key='mode_')
['.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0000.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0001.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0002.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0003.fits']
```

Notice that, in this specific case, it was necessary to include the undersc ore after 'mode' to exclude the 'modesVector.fits' file from the list.

`opticalib.load_fits(filepath, return_header=False)`

Loads a FITS file.

Parameters

- **filepath** (*str*) – Path to the FITS file.
- **return_header** (*bool*) – Wether to return the header of the loaded fits file. Default is False.

Returns

- **fit** (*np.ndarray or np.ma.MaskedArray*) – FITS file data.
- **header** (*dict | fits.Header, optional*) – The header of the loaded fits file.

Return type

tuple[*ImageData* | *CubeData* | *MatrixLike* | *Buffer* | *_SupportsArray*[*dtype*[*Any*]] | *_NestedSequence*[*_SupportsArray*[*dtype*[*Any*]]] | bool | int | float | complex | str | bytes | *_NestedSequence*[bool | int | float | complex | str | bytes], *Any*]

`opticalib.read_phasemap(file_path)`

Function to read interferometric data, in the three possible formats (FITS, 4D, H5)

Parameters

file_path (*str*) – Complete filepath of the file to load.

Returns

image – Image as a masked array.

Return type

ImageData

`opticalib.save_fits(filepath, data, overwrite=True, header=None)`

Saves a FITS file.

Parameters

- **filepath** (*str*) – Path to the FITS file.
- **data** (*np.array*) – Data to be saved.
- **overwrite** (*bool, optional*) – Whether to overwrite an existing file. Default is True.
- **header** (*dict[str, any] | fits.Header, optional*) – Header information to include in the FITS file. Can be a dictionary or a fits.Header object.

Return type

None

Modules

alignment	ALIGNMENT module 2024
analyzer	ANALYZER module 2020-2024
core	CORE module 2024
devices	DEVICES module 2025
dmutils	DMUTILS subpackage 2024

continues on next page

Table 2 – continued from previous page

ground	GROUND module 2024
typings	TYPINGS module 2025

2.1.4 opticalib.alignment

ALIGNMENT module

2024

Author(s):

- Pietro Ferraiuolo : pietro.ferraiuolo@inaf.it

Description

This module provides the *Alignment* class and related functions for performing alignment procedures, including calibration and correction.

How to Use it

This module contains the class *Alignment*, which manages, alone, both the calibration and the correction of the alignment of the system. The class is initialized with the mechanical and acquisition devices used for alignment. These devices, which, for example, in the case of the M4 project are the OTT and the interferometer, are passed as arguments and configured through the *configuration.yaml* file, under the *ALIGNMENT* section.

Usage Example

Given the OTT (with Parabola, Reference Mirror and M4 Hexapode) as mechanical device and the interferometer as acquisition device, we can initialize the class as follows:

```python
from opticalib.alignment import Alignment align = Alignment(ott, interf) # At this point the alignment is ready to be calibrated, given the command amplitude amps = [0,7, 10, 10, 6, 6, 4, 4] # example, verosimilar, amplitudes align.calibrate_alignment(amps) [...] "Ready for Alignment..."
```

At this point, the calibration is complete and and *InteractionMatrix.fits* file was created, saved and stored in the Alignment class. It is ready to compute and apply corrections.

```python
modes2correct = [3,4] # Reference Mirror DoF zern2correct = [0,1] # tip $ tilt align.correct_alignment(modes2correct, zern2correct, apply=True)
```

If we already have an *InteractionMatrix.fits* file, we can load it and apply corrections based off the loaded calibration. All to do is to load the calibration to the class:

```python
tn_cal = '20241122_160000' # example, tracking number align.load_calibration(tn_cal) # load the calibration align.correct_alignment(modes2correct, zern2correct, apply=True)
```

It can also be instanced with a calibration:

```python
tn_cal = '20241122_160000' # example, tracking number align = Alignment(ott, interf, calibtn=tn_cal) align.correct_alignment(modes2correct, zern2correct, apply=True)
```

Notes

Note that the calibration process can be done uploading to the class a *calibrated cavity*, so that a different algorithm for the Zernike fitting is performed. This can be done through the *load_fitting_surface* method.

```python
cavity_tn = '20241122_160000' # example, tracking number align.load_fitting_surface(cavity_tn) # load the calibrated cavity
```

When working with segmented system (e.g. a segmented mirror), the Zernike modes shall be computed as global coefficients, which are basically the average of the local amplitude measured on each of the segment.

Classes

Alignment(mechanical_devices, ...[, calibtn])	Class for the alignment procedure: calibration and correction.

class `opticalib.alignment.`**`Alignment`**(*mechanical_devices*, *acquisition_devices*, *calibtn=None*)

Bases: `object`

Class for the alignment procedure: calibration and correction.

This class provides methods to perform alignment procedures using mechanical and acquisition devices. It handles the initialization of devices, reading calibration data, and executing alignment commands.

Parameters

- **mechanical_devices** (*GenericDevice | list[GenericDevice]*)
- **acquisition_devices** (*InterferometerDevice | list[InterferometerDevice]*)
- **calibtn** (*str | None*)

mdev

The mechanical devices used for alignment. Can be either a single object which calls more devices or a list of single devices.

Type

object or list

ccd

The acquisition devices used for alignment.

Type

object

cmdMat

The command matrix read from a FITS file, used for alignment commands.

Type

numpy.ndarray

intMat

The interaction matrix, initialized as None.

Type
numpy.ndarray or None

recMat

The reconstruction matrix, initialized as None.

Type
numpy.ndarray or None

correct_alignment(*modes2correct*, *zern2correct*, *tn=None*, *apply=False*, *n_frames=15*)

Corrects the alignment of the system based on Zernike coefficients.

Parameters

- **modes2correct** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)
- **zern2correct** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)
- **apply** (*bool*)
- **n_frames** (*int*)

Return type
str | *Buffer* | *_SupportsArray*[*dtype*[*Any*]] | *_NestedSequence*[*_SupportsArray*[*dtype*[*Any*]]] | bool | int | float | complex | bytes | *_NestedSequence*[bool | int | float | complex | str | bytes]

calibrate_alignment(*cmdAmp*, *n_frames=15*, *template=None*, *n_repetitions=1*, *save=True*)

Calibrates the alignment of the system using the provided command amplitude and template.

Parameters

- **cmdAmp** (*int | float | Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)
- **n_frames** (*int*)
- **template** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)
- **n_repetitions** (*int*)
- **save** (*bool*)

Return type
str

read_positions(*show=True*)

Reads the current positions of the devices.

Parameters
`show` (*bool*)

Return type
Buffer | *_SupportsArray*[*dtype*[*Any*]] | *_NestedSequence*[*_SupportsArray*[*dtype*[*Any*]]] | bool | int | float | complex | str | bytes | *_NestedSequence*[bool | int | float | complex | str | bytes]

reload_calibrated_parabola(*tn*)

Reloads the calibrated parabola from the given tracking number.

__init__(*mechanical_devices*, *acquisition_devices*, *calibtn=None*)

Initializes the Alignment class with mechanical and acquisition devices.

Parameters

- **mechanical_devices** (*object or list of objects*) – The mechanical devices used for alignment. Can be either a single object which calls more devices or a list of single devices.
- **acquisition_devices** (*object*) – The acquisition devices used for alignment.
- **calibtn** (*str, optional*) – The tracking number of the alignment calibration to be used.

calibrate_alignment(*cmdAmp*, *n_frames=15*, *template=None*, *n_repetitions=1*, *save=True*)

Calibrate the alignment of the system using the provided command amplitude and template.

Parameters

- **cmdAmp** (*int | float | arrayLike*) – The command amplitude used for calibration.
- **n_frames** (*int, optional*) – The number of frames acquired and averaged for calibration. Default is 15.
- **template** (*list, optional*) – A list representing the template for calibration. If not provided, the default template will be used.
- **n_repetitions** (*int, optional*) – The number of repetitions for the calibration process. Default is 1.
- **save** (*bool, optional*) – If True, the resulting internal matrix will be saved to a FITS file. Default is False.

Returns
A message indicating that the system is ready for alignment.

Return type
str

Notes

This method performs the following steps: 1. Sets the command amplitude. 2. Uses the provided template or the default template if none is provided. 3. Produces a list of images based on the template and number of repetitions. 4. Executes a Zernike routine on the image list to generate an internal matrix. 5. Optionally saves the internal matrix to a FITS file.

correct_alignment(*modes2correct*, *zern2correct*, *apply=False*, *n_frames=15*)

Corrects the alignment of the system based on Zernike coefficients.

Parameters

- **modes2correct** (*array-like*) – Indices of the modes to correct.
- **zern2correct** (*array-like*) – Indices of the Zernike coefficients to correct.

- **tn** (*str, optional*) – Tracking number of the intMat.fits to be used
- **apply** (*bool, optional*) – If True, the correction command will be applied to the system. If False (default), the correction command will be returned.
- **n_frames** (*int, optional*) – Number of frames acquired and averaged the alignment correction. Default is 15.

Returns
If *apply* is False, returns the correction command as a numpy array. If *apply* is True, applies the correction command and returns a string indicating that the alignment has been corrected along with the current positions.

Return type
numpy.ndarray or str

Notes

This method acquires an image, calculates the Zernike coefficients, reads the interaction matrix from a FITS file, reduces the interaction matrix and command matrix based on the specified modes and Zernike coefficients, creates a reconstruction matrix, calculates the reduced command, and either applies the correction command or returns it.

load_calibration(*tn*)

Loads the alignment calibration InteractionMatrix.fits based on the provided tracking number.

Parameters
tn (*str*) – The tracking number of the calibration to be loaded.

Return type
None

load_fitting_surface(*filepath*)

This function let you load the mask to use for zernike fitting. In the case of M$, for example, here the calibrated parabola is loaded, so that zernike modes are fitted using the parabola surface (right) instead of the Reference Mirror one (smaller, wrong)

Parameters
filepath (*str*) – The file path to the parabola file.

Returns
A message indicating the successful loading of the file.

Return type
str

read_positions(*show=True*)

Reads the current positions of the devices.

Returns
pos – The list of current positions of the devices.

Return type
ArrayLike

Parameters
show (*bool*)

2.1.5 opticalib.analyzer

ANALYZER module

2020-2024

Author(s)

- Runa Briguglio: runa.briguglio@inaf.it
- Pietro Ferraiuolo: pietro.ferraiuolo@inaf.it

Description

Functions

averageFrames(tn[, first, last, ...])	Perform the average of a list of images, retrievable through a tracking number.
comp_filtered_image(imgin[, verbose, disp, ...])	
comp_psd(imgin[, nbins, norm, verbose, ...])	
createCube(filelist[, register])	Creates a cube of images from an images file list
cubeRebinner(cube, rebin[, method])	Cube rebinner
frame(idx, mylist)	Returns a single frame from a list of files or from a cube.
frame2ottFrame(img, croppar[, flipOffset])	
getDataFileList(tn)	Returns a list of data files for the given tracking number.
integrate_psd(y, img)	
modeRebinner(img, rebin[, method])	Image rebinner
openAverage(tn)	Loads an averaged frame from an 'average.fits' file, found inside the input tracking number
readTemperatures(tn)	
readZernike(tn)	
runningDiff(tn[, gap])	Computes the running difference of the frames in a given tracking number.
runningMean(vec, npoints)	
saveAverage(tn[, average_img, overwrite])	Saves an averaged frame, in the same folder as the original frames.
spectrum(signal[, dt, show])	
strfunct(vect, gapvect)	vect shall be npoints x m the strfunct is calculate m times over the npoints time series returns stf(n_timeseries x ngaps)
timevec(tn)	
track2date(tni)	Converts a tracing number into a list containing year, month, day, hour, minutes and seconds, divied.
track2jd(tni)	

continues on next page

Table 4 – continued from previous page

`zernikePlot`(mylist[, modes])	

`opticalib.analyzer.`**`averageFrames`**(*tn*, *first=None*, *last=None*, *file_selector=None*, *thresh=False*)

Perform the average of a list of images, retrievable through a tracking number.

Parameters

- **`tn`** (*`str`*) – Data Tracking Number.
- **`first`** (*`int, optional`*) – Index number of the first file to consider. If None, the first file in the list is considered.
- **`last`** (*`int, optional`*) – Index number of the last file to consider. If None, the last file in list is considered.
- **`file_selector`** (*`list, optional`*) – A list of integers, representing the specific files to load. If None, the range (first->last) is considered.
- **`thresh`** (*`bool, optional`*) – DESCRIPTION. The default is None.

Returns
aveimg – Final image of averaged frames.

Return type
ndarray

`opticalib.analyzer.`**`comp_filtered_image`**(*imgin*, *verbose=False*, *disp=False*, *d=1*, *freq2filter=None*)

Parameters

- **`imgin`** (*`TYPE`*) – DESCRIPTION.
- **`verbose`** (*`TYPE, optional`*) – DESCRIPTION. The default is False.
- **`disp`** (*`TYPE, optional`*) – DESCRIPTION. The default is False.
- **`d`** (*`TYPE, optional`*) – DESCRIPTION. The default is 1.
- **`freq2filter`** (*`TYPE, optional`*) – DESCRIPTION. The default is None.

Returns
imgout – DESCRIPTION.

Return type
TYPE

`opticalib.analyzer.`**`comp_psd`**(*imgin*, *nbins=None*, *norm='backward'*, *verbose=False*, *disp=False*, *d=1*, *sigma=None*, *crop=True*)

Parameters

- **`imgin`** (*`TYPE`*) – DESCRIPTION.
- **`nbins`** (*`TYPE, optional`*) – DESCRIPTION. The default is None.
- **`norm`** (*`TYPE, optional`*) – DESCRIPTION. The default is “backward”.
- **`verbose`** (*`TYPE, optional`*) – DESCRIPTION. The default is False.
- **`disp`** (*`TYPE, optional`*) – DESCRIPTION. The default is False.
- **`d`** (*`TYPE, optional`*) – DESCRIPTION. The default is 1.

- **sigma** (*TYPE, optional*) – DESCRIPTION. The default is None.
- **crop** (*TYPE, optional*) – DESCRIPTION. The default is True.

Returns

- **fout** (*TYPE*) – DESCRIPTION.
- **Aout** (*TYPE*) – DESCRIPTION.

`opticalib.analyzer.`**`createCube`**(*filelist*, *register=False*)

Creates a cube of images from an images file list

Parameters

- **filelist** (*list of str*) – List of file paths to the images/frames to be stacked into a cube.
- **register** (*int or tuple, optional*) – If not False, and int or a tuple of int must be passed as value, and the registration algorithm is performed on the images before stacking them into the cube. Default is False.

Returns
cube – Data cube containing the images/frames stacked.

Return type
ndarray

`opticalib.analyzer.`**`cubeRebinner`**(*cube*, *rebin*, *method='averaging'*)

Cube rebinner

Parameters

- **cube** (*ndarray*) – Cube to rebin.
- **rebin** (*int*) – Rebinning factor.
- **method** (*str, optional*) – Rebinning method, either 'averaging' or 'sampling'. The default is 'averaging'.

Returns
newCube – Rebinned cube.

Return type
ndarray

`opticalib.analyzer.`**`frame`**(*idx*, *mylist*)

Returns a single frame from a list of files or from a cube.

Parameters

- **id** (*TYPE*) – DESCRIPTION.
- **mylist** (*TYPE*) – DESCRIPTION.
- **idx** (*int*)

Returns
img – DESCRIPTION.

Return type
TYPE

`opticalib.analyzer.`**`frame2ottFrame`**(*img*, *croppar*, *flipOffset=True*)

Parameters

- **img** (*TYPE*) – DESCRIPTION.
- **croppar** (*TYPE*) – DESCRIPTION.
- **flipOffset** (*TYPE, optional*) – DESCRIPTION. The default is True.

Returns
fullimg – DESCRIPTION.

Return type
TYPE

`opticalib.analyzer.getDataFileList(tn)`

Returns a list of data files for the given tracking number.

Parameters
tn (*str*) – Tracking number.

Returns
filelist – List of file paths to the data files.

Return type
list of str

`opticalib.analyzer.integrate_psd(y, img)`

Parameters

- **y** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)
- **img** (*ImageData*)

Return type
Buffer | *_SupportsArray*[*dtype*[*Any*]] | *_NestedSequence*[*_SupportsArray*[*dtype*[*Any*]]] | bool | int | float | complex | str | bytes | *_NestedSequence*[bool | int | float | complex | str | bytes]

`opticalib.analyzer.modeRebinner(img, rebin, method='averaging')`

Image rebinner

Rebins a masked array image by a factor rebin.

Parameters

- **img** (*masked_array*) – Image to rebin.
- **rebin** (*int*) – Rebinning factor.
- **method** (*str, optional*) – Rebinning method, either 'averaging' or 'sampling'. The default is 'averaging'.

Returns
newImg – Rebinned image.

Return type
masked_array

`opticalib.analyzer.openAverage(tn)`

Loads an averaged frame from an 'average.fits' file, found inside the input tracking number

Parameters
tn (*str*) – Tracking number of the averaged frame.

Returns
: **image** – Averaged image.

Return type
: ndarray

Raises
: `FileNotFoundError` – Raised if the file does not exist.

`opticalib.analyzer.readTemperatures(tn)`

Parameters
: **tn** (*TYPE*) – DESCRIPTION.

Returns
: **temperatures** – DESCRIPTION.

Return type
: TYPE

`opticalib.analyzer.readZernike(tn)`

Parameters
: **tn** (*TYPE*) – DESCRIPTION.

Returns
: **temperatures** – DESCRIPTION.

Return type
: TYPE

`opticalib.analyzer.runningDiff(tn, gap=2)`

Computes the running difference of the frames in a given tracking number.

Parameters
- **tn** (`str`) – Tracking number of the frames to process.
- **gap** (`int, optional`) – Number of frames to skip between each difference calculation. The default is 2.

Returns
: **svec** – Array of standard deviations for each frame difference.

Return type
: ndarray

`opticalib.analyzer.runningMean(vec, npoints)`

Parameters
- **vec** (*TYPE*) – DESCRIPTION.
- **npoints** (*TYPE*) – DESCRIPTION.

Returns
: DESCRIPTION.

Return type
: TYPE

`opticalib.analyzer.`**`saveAverage`**`(`*tn*, *average_img=None*, *overwrite=False*, ***kwargs*`)`

Saves an averaged frame, in the same folder as the original frames. If no averaged image is passed as argument, it will create a new average for the specified tracking number, and additional arguments, the same as ''average-Frames'' can be specified.

Parameters

- **`tn`** (*`str`*) – Tracking number where to save the average frame file. If average_img is None, it is the tracking number of the data that will be averaged
- **`average_img`** (*`ndarray, optional`*) – Result average image of multiple frames. If it's None, it will be generated from data found in the tracking number folder. Additional arguments can be passed on
- **`**kwargs`** (*`dict[str, Any]`*) – The same arguments as *averageFrames*, to specify the averaging method. - first : int, optional

  Index number of the first file to consider. If None, the first file in the list is considered.

  - **last**
    [int, optional] Index number of the last file to consider. If None, the last file in list is considered.
  - **file_selector**
    [list of ints, optional] A list of integers, representing the specific files to load. If None, the range (first->last) is considered.
  - **thresh**
    [bool, optional] DESCRIPTION. The default is None.
- **`overwrite`** (*`bool`*)
- **`**kwargs`**

`opticalib.analyzer.`**`spectrum`**`(`*signal*, *dt=1*, *show=None*`)`

Parameters

- **`signal`** (*`ndarray`*) – DESCRIPTION.
- **`dt`** (*`float, optional`*) – DESCRIPTION. The default is 1.
- **`show`** (*`bool, optional`*) – DESCRIPTION. The default is None.

Returns

- **spe** (*float* | *ndarray*) – DESCRIPTION.
- **freq** (*float* | *ArrayLike*) – DESCRIPTION.

Return type

tuple[*Buffer* | *_SupportsArray*[*dtype*[*Any*]] | *_NestedSequence*[*_SupportsArray*[*dtype*[*Any*]]] | bool | int | float | complex | str | bytes | *_NestedSequence*[bool | int | float | complex | str | bytes], *Buffer* | *_SupportsArray*[*dtype*[*Any*]] | *_NestedSequence*[*_SupportsArray*[*dtype*[*Any*]]] | bool | int | float | complex | str | bytes | *_NestedSequence*[bool | int | float | complex | str | bytes]]

`opticalib.analyzer.`**`strfunct`**`(`*vect*, *gapvect*`)`

vect shall be npoints x m the strfunct is calculate m times over the npoints time series returns stf(n_timeseries x ngaps)

Parameters

- **vect** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)
- **gapvect** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)

Return type

Buffer | *_SupportsArray*[*dtype*[*Any*]] | *_NestedSequence*[*_SupportsArray*[*dtype*[*Any*]]] | bool | int | float | complex | str | bytes | *_NestedSequence*[bool | int | float | complex | str | bytes]

`opticalib.analyzer.`**`timevec`**(*tn*)

Parameters

tn (*str*) – Tracking number of the frames to process.

Returns

timevector – Array of time values for each frame.

Return type

_np.ndarray

`opticalib.analyzer.`**`track2date`**(*tni*)

Converts a tracing number into a list containing year, month, day, hour, minutes and seconds, divied.

Parameters

tni (*str*) – Tracking number to be converted.

Returns

time – List containing the date element by element. [0] y : str

Year.

[1] mo
[str] Month.

[2] d
[str] Day.

[3] h
[float] Hour.

[4] mi
[float] Minutes.

[5] s
[float] Seconds.

Return type

list

`opticalib.analyzer.`**`track2jd`**(*tni*)

Parameters

tni (*TYPE*) – DESCRIPTION.

Returns

jdi – DESCRIPTION.

Return type
TYPE

`opticalib.analyzer.zernikePlot(mylist, modes=None)`

Parameters

- **mylist** (*TYPE*) – DESCRIPTION.
- **modes** (*TYPE, optional*) – DESCRIPTION. The default is _np.array(range(1, 11)).

Returns
zcoeff – DESCRIPTION.

Return type
TYPE

2.1.6 opticalib.core

CORE module

2024

Author(s):

- Pietro Ferraiuolo: pietro.ferraiuolo@inaf.it

Description:

This module contains the core functionalities of the opticalib package, such as path management, configuration handling and custom exceptions.

Contents:

- *root.py*: Module for managing the *opticalib* file paths.
- *read_config.py*: Module for handling the *opticalib* main configuration file.
- *exceptions.py*: Module for defining custom exceptions.

Modules

exceptions	
read_config	Module: read_config This module provides utilities for reading, writing, and updating YAML configuration files used in the opticalib system. It supports configuration management for devices such as deformable mirrors and interferometers, as well as acquisition and alignment settings.
root	

opticalib.core.exceptions

This module defines custom exceptions used in the opticalib system.

Exceptions

CommandError(message)	Exception raised when a command is not valid.
DeviceError(device_name, device_type)	Exception raised when a device is not found in the configuration file.
DeviceNotFoundError(device_name)	Exception raised when a device is not found in the configuration file.
MatrixError(message)	Exception raised when a matrix is not valid.

exception `opticalib.core.exceptions.`**`CommandError`**(*message*)

Bases: `Exception`

Exception raised when a command is not valid.

Parameters
message (*str*)

add_note()

Exception.add_note(note) – add a note to the exception

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `opticalib.core.exceptions.`**`DeviceError`**(*device_name*, *device_type*)

Bases: `Exception`

Exception raised when a device is not found in the configuration file.

Parameters

- **device_name** (*str*)
- **device_type** (*str*)

add_note()

Exception.add_note(note) – add a note to the exception

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `opticalib.core.exceptions.`**`DeviceNotFoundError`**(*device_name*)

Bases: `Exception`

Exception raised when a device is not found in the configuration file.

Parameters
device_name (*str*)

add_note()

Exception.add_note(note) – add a note to the exception

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `opticalib.core.exceptions.`**`MatrixError`**(*message*)

Bases: `Exception`

Exception raised when a matrix is not valid.

Parameters

message (*str*)

add_note()

Exception.add_note(note) – add a note to the exception

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

opticalib.core.read_config

This module provides utilities for reading, writing, and updating YAML configuration files used in the opticalib system. It supports configuration management for devices such as deformable mirrors and interferometers, as well as acquisition and alignment settings.

Features

- Load and dump YAML configuration files.
- Retrieve and update configuration blocks for IFF acquisition, DM devices, and interferometers.
- Copy configuration files for record keeping.
- Parse and convert configuration values, including numpy arrays.
- Access alignment and stitching settings as structured objects.

Author(s)

- Pietro Ferraiuolo: written in 2025
- Runa Briguglio

Functions

copyIffConfigFile(tn[, old_path])	Copies the YAML configuration file to the new folder for record keeping of the configuration used on data acquisition.
dump_yaml_config(config[, path])	Writes the configuration dictionary back to the YAML file.
getAlignmentConfig()	Reads the alignment settings in the configuration file.
getCmdDelay([bpath])	Retrieves the command delay from the YAML configuration file.
getDmConfig(device_name)	Retrieves the DM address from the YAML configuration file.

continues on next page

Table 7 – continued from previous page

getDmIffConfig([bpath])	Retrieves the DM configuration from the YAML file.
getIffConfig(key[, bpath])	Reads the configuration from the YAML file for the IFF acquisition.
getInterfConfig(device_name)	Retrieves the interferometer address from the YAML configuration file.
getNActs([bpath])	Retrieves the number of actuators from the YAML configuration file.
getStitchingConfig()	Reads the stitching settings in the configuration file.
getTiming([bpath])	Retrieves timing information from the YAML configuration file.
load_yaml_config([path])	Loads the YAML configuration file.
updateConfigFile(key, item, value[, bpath])	Updates the YAML configuration file for the IFF acquisition.
updateIffConfig(tn, item, value)	Updates the YAML configuration file for the IFF acquisition.

opticalib.core.read_config.**copyIffConfigFile**(*tn*, *old_path='/home/pietrof/.tmp_opticalibData/SysConfig'*)

Copies the YAML configuration file to the new folder for record keeping of the configuration used on data acquisition.

Parameters

- **tn** (*str*) – Tracking number for the new data.
- **old_path** (*str, optional*) – Base path where the YAML configuration file resides.

Returns

res – Path where the file was copied.

Return type

str

opticalib.core.read_config.**dump_yaml_config**(*config*, *path=None*)

Writes the configuration dictionary back to the YAML file.

Parameters

- **config** (*dict*) – The configuration dictionary to write.
- **bpath** (*str, optional*) – Base path of the file to write. Default points to the configuration root folder.
- **path** (*str*)

opticalib.core.read_config.**getAlignmentConfig**()

Reads the alignment settings in the configuration file.

Returns

config – The alignment configuration as a class, for backwards compatibility.

Return type

class

opticalib.core.read_config.**getCmdDelay**(*bpath='/home/pietrof/.tmp_opticalibData/SysConfig'*)

Retrieves the command delay from the YAML configuration file.

Parameters

bpath (*str, optional*) – Base path of the configuration file.

Returns
: **cmdDelay** – Command delay for the interferometer synchronization.

Return type
: float

`opticalib.core.read_config.getDmConfig`(*device_name*)

Retrieves the DM address from the YAML configuration file.

Parameters
: `device_name` (`str`) – Name of the DM device.

Returns

- **ip** (*str*) – DM ip address.
- **port** (*int*) – DM port.

`opticalib.core.read_config.getDmIffConfig`(*bpath='/home/pietrof/.tmp_opticalibData/SysConfig'*)

Retrieves the DM configuration from the YAML file.

Parameters
: `bpath` (`str, optional`) – Base path of the configuration file.

Returns
: **config** – The DM configuration dictionary.

Return type
: dict

`opticalib.core.read_config.getIffConfig`(*key*, *bpath='/home/pietrof/.tmp_opticalibData/SysConfig'*)

Reads the configuration from the YAML file for the IFF acquisition. The key passed is the block of information retrieved within the INFLUENCE.FUNCTIONS section.

Parameters

- `key` (`str`) –

  Key value of the block of information to read. Can be

  - 'TRIGGER'
  - 'REGISTRATION'
  - 'IFFUNC'
- `bpath` (`str, optional`) – Base path of the file to read. Default points to the configuration root folder.

Returns

info –

A dictionary containing the configuration info:

- zeros
- modes
- amplitude
- template
- modalBase

Return type
: dict

`opticalib.core.read_config.getInterfConfig(device_name)`

Retrieves the interferometer address from the YAML configuration file.

Returns

- **ip** (*str*) – Interferometer ip address.
- **port** (*int*) – Interferometer port.

Parameters

`device_name` (*str*)

`opticalib.core.read_config.getNActs(bpath='/home/pietrof/.tmp_opticalibData/SysConfig')`

Retrieves the number of actuators from the YAML configuration file.

Parameters

`bpath` (*str, optional*) – Base path of the configuration file.

Returns

nacts – Number of DM actuators.

Return type

int

`opticalib.core.read_config.getStitchingConfig()`

Reads the stitching settings in the configuration file.

Returns

config – The defined stitching parameters.

Return type

dict

`opticalib.core.read_config.getTiming(bpath='/home/pietrof/.tmp_opticalibData/SysConfig')`

Retrieves timing information from the YAML configuration file.

Parameters

`bpath` (*str, optional*) – Base path of the configuration file.

Returns

timing – Timing used for synchronization.

Return type

int

`opticalib.core.read_config.load_yaml_config(path=None)`

Loads the YAML configuration file.

Parameters

`path` (*str, optional*) – Base path of the file to read. Default points to the configuration root folder.

Returns

config – The configuration dictionary.

Return type

dict

`opticalib.core.read_config.updateConfigFile(key, item, value, bpath='/home/pietrof/.tmp_opticalibData/SysConfig')`

Updates the YAML configuration file for the IFF acquisition. The key passed is within the INFLUENCE.FUNCTIONS section.

Parameters

- **key** (*str*) – Key of the block to update (e.g., 'TRIGGER', 'REGISTRATION', 'IFFUNC').
- **item** (*str*) – The configuration item to update.
- **value** (*any*) – New value to update.
- **bpath** (*str, optional*) – Base path of the configuration file.

opticalib.core.read_config.**updateIffConfig**(*tn*, *item*, *value*)

Updates the YAML configuration file for the IFF acquisition. The item passed is within the INFLUENCE.FUNCTIONS/IFFUNC section.

Parameters

- **tn** (*str*) – Tracking number of the *iffConfig.yaml* copied from the original *configuration.yaml* file.
- **item** (*str*) – The configuration item to update.
- **value** (*any*) – New value to update.

opticalib.core.root

this module is at the heart of the package, as it defines its folder structure and the configuration file reader and writer. Also, it is fundamental for the *calpy* custom entry point functionalities.

Functions

create_configuration_file([path, data_path])	Create a configuration file in the specified path.
create_folder_tree(BASE_DATA_PATH)	Create the folder tree for the package.

Classes

ConfSettingReader4D(file_path)	Class which reads an interferometer configuration settings file '4DSettings.ini'

class opticalib.core.root.**ConfSettingReader4D**(*file_path*)

Bases: *object*

Class which reads an interferometer configuration settings file '4DSettings.ini'

getFrameRate() :

Gets the camera frame rate in Hz.

getImageWidthInPixels() :

Get the width of the frame in pixel units.

getImageHeightInPixels() :

Get the height of the frame in pixel units.

getOffsetX() :

Get the frame offset in x-axis.

getOffsetY() :

Get the frame offset in y-axis.

getPixelFormat() :
Get the format of the pixels.

getUserSettingFilePath() :
Get the path of the configuration file.

How to Use it

After initializing the class with a file path, just call methods on the defined

`object()`

```
>>> cr = ConfSettingReader(file_path)
>>> cr.getImageWidhtInPixels()
2000()
>>> cr.getImageHeightInPixels()
2000()
```

Notes

Note that there is no need to directly use this module, as the settings information retrievement is handled by m4.urils.osutils, with its functions ''getConf4DSettingsPath" and ''getCameraSettings".

`getFrameRate()`

Returns the acquisition frame rate of the interferometer in Hz

Returns
frame_rate – The frame rate.

Return type
float

`getImageHeightInPixels()`

Returns the image height in pixel scale

Returns
image_height_in_pixels – Image pixel height.

Return type
int

`getImageWidhtInPixels()`

Returns the image widht in pixel scale

Returns
image_wight_in_pixels – Image pixel width.

Return type
int

`getOffsetX()`

Returns the camera offset, in pixels, along the x-axis.

Returns
offset_x – Pixel offset in the x-axis.

Return type
int

`getOffsetY()`

Returns the camera offset, in pixels, along the y-axis.

Returns
offset_y – Pixel offset in the y-axis.

Return type
int

`getPixelFormat()`

Returns the format of the pixel.

Returns
pixel_format – Pixel format.

Return type
str

`getUserSettingFilePath()`

Returns the complete filepath of the settings configuration file.

Returns
user_setting_file_path – Settings file path.

Return type
str

`opticalib.core.root.create_configuration_file(path='', data_path=False)`

Create a configuration file in the specified path.

Parameters

- **path** (*str*) – The path to the configuration file.
- **data_path** (*str | bool*) – The path to the data folder. If True, it will be set to the same directory as the configuration file. If False, it will not be set. If a string, a path must be provided, and the *data_path* will be set to that path.
- **load** (*bool*) – If True, the configuration file will be loaded after creation, the folder tree created (if not already) and all the paths updated.

Return type
None

`opticalib.core.root.create_folder_tree(BASE_DATA_PATH)`

Create the folder tree for the package.

Parameters
BASE_DATA_PATH (*str*)

Return type
None

2.1.7 opticalib.devices

DEVICES module

2025

Author(s):

- Pietro Ferraiuolo: pietro.ferraiuolo@inaf.it
- Runa Briguglio: runa.briguglio@inaf.it

Description:

This module contains the classes for interfacing the devices used in optical benches, or in general optical devices.

Contents:

- deformable_mirrors.py: Contains classes for different deformable mirrors. The definitions and low level interfaces to these devices are handled in the *_API* submodule.
- interferometer.py: Contains classes for different interferometers. The definitions and low level interfaces to these devices are handled in the *_API* submodule.

class `opticalib.devices.`**`AccuFiz`**(*model=None*, *ip=None*, *port=None*)

Bases: `BaseInterferometer`

Class for the AccuFiz Laser Interferometer.

Parameters

- **`model`** (*str | int | None*)
- **`ip`** (*str*)
- **`port`** (*int*)

`__init__`(*model=None*, *ip=None*, *port=None*)

The constructor

Parameters

- **`model`** (*str | int | None*)
- **`ip`** (*str*)
- **`port`** (*int*)

`acquireFullFrame`(***kwargs*)

Wrapper for the consecutive execution of *acquire_mapo* and *intoFullFrame*.

Parameters

`**kwargs` (*dict*) – Additional keyword arguments to be passed to *acquire_map*.

Returns

The full frame image data.

Return type

_ot.ImageData

`acquire_detector`(*nframes=1*, *delay=0*)

Parameters

- **`nframes`** (*int*) – number of frames
- **`delay`** (*int | flaot [s]*) – delay between images

Returns

data2d – detector interferometer image

Return type
numpy masked array

`acquire_map`(*nframes=1*, *delay=0*, *rebin=1*)

Acquires the interferometer image and returns it as a masked array.

Parameters

- **nframes** (*int*) – Number of frames to be averaged that produce the measurement.
- **delay** (*int*) – Delay between images in seconds.
- **rebin** (*int*) – Rebin factor for the image.

Returns
masked_ima – Interferometer image.

Return type
ImageData

`capture`(*numberOfFrames*, *folder_name=None*)

Parameters

- **numberOfFrames** (*int*) – number of frames to acquire
- **folder_name** (*string*) – if None a tacking number is generate

Returns
folder_name – name of folder measurements

Return type
string

`copy4DSettings`(*destination*)

Copies the interferometer settings file to the specified destination.

Parameters
destination (*str*)

Return type
None

`getCameraSettings`()

Reads che actual interferometer settings from its configuration file.

Returns

- **output** (*list*)
- **list of camera settings** (*[width_pixel, height_pixel, offset_x, offset_y]*)

Return type
list[int]

`getFrameRate`()

Reads the frame rate the interferometer is working at.

Returns
frame_rate – Frame rate of the interferometer

Return type
float

`intoFullFrame(img)`

The function fits the passed frame (expected cropped) into the full interferometer frame (2048x2048), after reading the cropping parameters.

Parameters
`img` (*ImageData*) – The image to be fitted into the full frame.

Returns
output – The output image, in the interferometer full frame.

Return type
ImageData

`loadConfiguration(conffile)`

Read and loads the configuration file of the interferometer.

Parameters
`conffile` (*string*) – name of the configuration file to load

Return type
None

`produce(tn)`

Parameters

- `folder_name` (*string*) – name of folder measurements to convert
- `tn` (*str | list[str]*)

Return type
None

`setTriggerMode(enable)`

Parameters

- `folder_name` (*string*) – name of folder measurements to convert
- `enable` (*bool*)

Return type
None

`class opticalib.devices.AdOpticaDm(tn=None)`

Bases: `BaseAdOpticaDm`, `BaseDeformableMirror`

AdOptica Deformable Mirror interface.

Used with the AdOptica AO Client. In use for the DP, and will later be used for M4.

Parameters
`tn` (*str | None*)

`__init__(tn=None)`

The Constructor

Parameters
`tn` (*str | None*)

`getCounter()`

Function which returns the current shape of the mirror.

Returns
shape – Current shape of the mirror.

Return type
numpy.ndarray

`get_force()`

Function which returns the current force applied to the mirror.

Returns
force – Current force applied to the mirror actuators.

Return type
numpy.ndarray

`get_shape()`

Retrieve the actuators positions

`plot_acts(amp=None, **kwargs)`

Function which plots the actuators.

Parameters

- **amp** (`ot.ArrayLike`) – Amplitude to be plotted.
- ****kwargs** (`dict`) – Additional keyword arguments for plotting.

`runCmdHistory(interf=None, differential=False, save=None)`

Runs the loaded command history on the DM. If *triggered* is not False, it must be a dictionary containing the low level arguments for the *aoClient.timeHistoryRun* function.

Parameters

- **interf** (`_ot.InterferometerDevice`) – The interferometer device to be used for acquiring images during the command history run.
- **differential** (`bool, optional`) – If True, the commands will be applied as differential commands (default is False).
- **triggered** (`bool | dict[str, _ot.Any], optional`) – If False, the command history will be run in a sequential mode. If not False, a dictionary must be provided, where it should contain the keys 'freq', 'wait', and 'delay' for the triggered mode.
- **sequential_delay** (`int | float, optional`) – The delay between each command execution in seconds (only if not in triggered mode).
- **save** (`str, optional`) – If provided, the command history will be saved with this name as a timestamp.

Return type
None

`set_shape(cmd)`

Applies the given command to the DM actuators.

Parameters
cmd (`list[float]`) – The command to be applied to the DM actuators, of lenght equal the number of actuators.

`uploadCmdHistory(tcmdhist)`

Uploads the (timed) command history in the DM. if *for_triggered* is true, then it is loaded direclty in the AO client for the triggere mode run.

Parameters

- **tcmdhist** (*_ot.MatrixLike*) – The command history to be uploaded, of shape (used_acts, nmodes).
- **tfor_triggered** (*bool, optional*) – If True, the command history will be uploaded directly to the AO client for the triggered mode run. If False, it will be stored in the *cmdHistory* attribute of the DM instance (default is False).

Return type

None

class `opticalib.devices.AlpaoDm`(*nacts=None*, *ip=None*, *port=None*)

Bases: `BaseAlpaoMirror`, `BaseDeformableMirror`

Alpao Deformable Mirror interface.

Parameters

- **nacts** (*str | int | None*)
- **ip** (*str | None*)
- **port** (*int | None*)

`__init__`(*nacts=None*, *ip=None*, *port=None*)

The Contructor

Parameters

- **nacts** (*str | int | None*)
- **ip** (*str | None*)
- **port** (*int | None*)

`get_shape()`

Abstract method to get the shape of the deformable mirror. Must be implemented by subclasses.

Return type

Buffer | *_SupportsArray*[*dtype*[*Any*]] | *_NestedSequence*[*_SupportsArray*[*dtype*[*Any*]]] | bool | int | float | complex | str | bytes | *_NestedSequence*[bool | int | float | complex | str | bytes]

property `nActuators`: int

`runCmdHistory`(*interf=None*, *delay=0.2*, *save=None*, *differential=True*)

Abstract method to run the command history on the deformable mirror. Must be implemented by subclasses.

Parameters

- **interf** (*InterferometerDevice*)
- **delay** (*int | float*)
- **save** (*str*)
- **differential** (*bool*)

Return type

str

`setReferenceActuator`(*refAct*)

Parameters

refAct (*int*)

Return type
None

`setZeros2Acts()`

`set_shape(cmd, differential=False)`

Abstract method to set the shape of the deformable mirror. Must be implemented by subclasses.

Parameters

- **cmd** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)
- **differential** (*bool*)

Return type
None

`uploadCmdHistory(tcmdhist)`

Abstract method to upload the command history to the deformable mirror. Must be implemented by subclasses.

Parameters
tcmdhist (*MatrixLike*)

Return type
None

`class opticalib.devices.PhaseCam(model=None, ip=None, port=None)`

Bases: `BaseInterferometer`

Class for the 4D Twyman-Green PhaseCam Laser Interferometer.

Parameters

- **model** (*str | int | None*)
- **ip** (*str*)
- **port** (*int*)

`__init__(model=None, ip=None, port=None)`

The constructor

Parameters

- **model** (*str | int | None*)
- **ip** (*str*)
- **port** (*int*)

`acquireFullFrame(**kwargs)`

Wrapper for the consecutive execution of *acquire_mapo* and *intoFullFrame*.

Parameters
****kwargs** (*dict*) – Additional keyword arguments to be passed to *acquire_map*.

Returns
The full frame image data.

Return type
_ot.ImageData

`acquire_detector`(*nframes=1*, *delay=0*)

Parameters

- **nframes** (*int*) – number of frames
- **delay** (*int | flaot [s]*) – delay between images

Returns
data2d – detector interferometer image

Return type
numpy masked array

`acquire_map`(*nframes=1*, *delay=0*, *rebin=1*)

Acquires the interferometer image and returns it as a masked array.

Parameters

- **nframes** (*int*) – Number of frames to be averaged that produce the measurement.
- **delay** (*int*) – Delay between images in seconds.
- **rebin** (*int*) – Rebin factor for the image.

Returns
masked_ima – Interferometer image.

Return type
ImageData

`capture`(*numberOfFrames*, *folder_name=None*)

Parameters

- **numberOfFrames** (*int*) – number of frames to acquire
- **folder_name** (*string*) – if None a tacking number is generate

Returns
folder_name – name of folder measurements

Return type
string

`copy4DSettings`(*destination*)

Copies the interferometer settings file to the specified destination.

Parameters
destination (*str*)

Return type
None

`getCameraSettings`()

Reads che actual interferometer settings from its configuration file.

Returns

- **output** (*list*)
- **list of camera settings** (*[width_pixel, height_pixel, offset_x, offset_y]*)

Return type
list[int]

getFrameRate()

Reads the frame rate the interferometer is working at.

Returns
frame_rate – Frame rate of the interferometer

Return type
float

intoFullFrame(*img*)

The function fits the passed frame (expected cropped) into the full interferometer frame (2048x2048), after reading the cropping parameters.

Parameters
img (*ImageData*) – The image to be fitted into the full frame.

Returns
output – The output image, in the interferometer full frame.

Return type
ImageData

loadConfiguration(*conffile*)

Read and loads the configuration file of the interferometer.

Parameters
conffile (*string*) – name of the configuration file to load

Return type
None

produce(*tn*)

Parameters

- **folder_name** (*string*) – name of folder measurements to convert
- **tn** (*str | list[str]*)

Return type
None

setTriggerMode(*enable*)

Parameters

- **folder_name** (*string*) – name of folder measurements to convert
- **enable** (*bool*)

Return type
None

class opticalib.devices.**SplattDm**(*ip=None*, *port=None*)

Bases: `BaseDeformableMirror`

SPLATT deformable mirror interface.

Parameters

- **ip** (*str*)
- **port** (*int*)

`__init__`(*ip=None*, *port=None*)

The Constructor

Parameters

- **ip** (*str*)
- **port** (*int*)

`get_shape`()

Abstract method to get the shape of the deformable mirror. Must be implemented by subclasses.

`integratePosition`(*Nits=3*)

Parameters

Nits (*int*)

property nActuators: int

`plot_command`(*cmd*)

Parameters

cmd (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)

Return type

None

`runCmdHistory`(*interf=None*, *delay=0.2*, *save=None*, *differential=True*, *read_buffers=False*)

Abstract method to run the command history on the deformable mirror. Must be implemented by subclasses.

Parameters

- **interf** (*InterferometerDevice | None*)
- **delay** (*int | float*)
- **save** (*str | None*)
- **differential** (*bool*)
- **read_buffers** (*bool*)

Return type

str

`sendBufferCommand`(*cmd*, *differential=False*, *delay=1.0*)

Parameters

- **cmd** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)
- **differential** (*bool*)
- **delay** (*int | float*)

Return type

str

set_shape(*cmd*, *differential=False*)

Abstract method to set the shape of the deformable mirror. Must be implemented by subclasses.

Parameters

- **cmd** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)
- **differential** (*bool*)

Return type

None

uploadCmdHistory(*tcmdhist*)

Abstract method to upload the command history to the deformable mirror. Must be implemented by subclasses.

Parameters

tcmdhist (*MatrixLike*)

Return type

None

Modules

deformable_mirrors	This module contains the classes for the high-level use of deformable mirrors.
interferometer	This module contains the high-level classes for the use of interferometer devices.

opticalib.devices.deformable_mirrors

This module contains the classes for the high-level use of deformable mirrors.

Author(s)

- Pietro Ferraiuolo : written in 2025

Description

Classes

AdOpticaDm([tn])	AdOptica Deformable Mirror interface.
AlpaoDm([nacts, ip, port])	Alpao Deformable Mirror interface.
SplattDm([ip, port])	SPLATT deformable mirror interface.

class opticalib.devices.deformable_mirrors.**AdOpticaDm**(*tn=None*)

Bases: BaseAdOpticaDm, BaseDeformableMirror

AdOptica Deformable Mirror interface.

Used with the AdOptica AO Client. In use for the DP, and will later be used for M4.

Parameters

tn (*str | None*)

`__init__`(*tn=None*)

The Constructor

Parameters

tn (*str | None*)

`getCounter`()

Function which returns the current shape of the mirror.

Returns

shape – Current shape of the mirror.

Return type

numpy.ndarray

`get_force`()

Function which returns the current force applied to the mirror.

Returns

force – Current force applied to the mirror actuators.

Return type

numpy.ndarray

`get_shape`()

Retrieve the actuators positions

`plot_acts`(*amp=None*, ***kwargs*)

Function which plots the actuators.

Parameters

- **amp** (*ot.ArrayLike*) – Amplitude to be plotted.
- ****kwargs** (*dict*) – Additional keyword arguments for plotting.

`runCmdHistory`(*interf=None*, *differential=False*, *save=None*)

Runs the loaded command history on the DM. If *triggered* is not False, it must be a dictionary containing the low level arguments for the *aoClient.timeHistoryRun* function.

Parameters

- **interf** (*_ot.InterferometerDevice*) – The interferometer device to be used for acquiring images during the command history run.
- **differential** (*bool, optional*) – If True, the commands will be applied as differential commands (default is False).
- **triggered** (*bool | dict[str, _ot.Any], optional*) – If False, the command history will be run in a sequential mode. If not False, a dictionary must be provided, where it should contain the keys 'freq', 'wait', and 'delay' for the triggered mode.
- **sequential_delay** (*int | float, optional*) – The delay between each command execution in seconds (only if not in triggered mode).
- **save** (*str, optional*) – If provided, the command history will be saved with this name as a timestamp.

Return type

None

`set_shape`(*cmd*)

Applies the given command to the DM actuators.

Parameters

cmd (*list[float]*) – The command to be applied to the DM actuators, of lenght equal the number of actuators.

`uploadCmdHistory`(*tcmdhist*)

Uploads the (timed) command history in the DM. if *for_triggered* is true, then it is loaded direclty in the AO client for the triggere mode run.

Parameters

- **tcmdhist** (*_ot.MatrixLike*) – The command history to be uploaded, of shape (used_acts, nmodes).
- **tfor_triggered** (*bool, optional*) – If True, the command history will be uploaded directly to the AO client for the triggered mode run. If False, it will be stored in the *cmdHistory* attribute of the DM instance (default is False).

Return type

None

class opticalib.devices.deformable_mirrors.**AlpaoDm**(*nacts=None*, *ip=None*, *port=None*)

Bases: `BaseAlpaoMirror`, `BaseDeformableMirror`

Alpao Deformable Mirror interface.

Parameters

- **nacts** (*str | int | None*)
- **ip** (*str | None*)
- **port** (*int | None*)

`__init__`(*nacts=None*, *ip=None*, *port=None*)

The Contructor

Parameters

- **nacts** (*str | int | None*)
- **ip** (*str | None*)
- **port** (*int | None*)

`get_shape`()

Abstract method to get the shape of the deformable mirror. Must be implemented by subclasses.

Return type

Buffer | *_SupportsArray*[*dtype*[*Any*]] | *_NestedSequence*[*_SupportsArray*[*dtype*[*Any*]]] | bool | int | float | complex | str | bytes | *_NestedSequence*[bool | int | float | complex | str | bytes]

property nActuators: int

`runCmdHistory`(*interf=None*, *delay=0.2*, *save=None*, *differential=True*)

Abstract method to run the command history on the deformable mirror. Must be implemented by subclasses.

Parameters

- **interf** (*InterferometerDevice*)
- **delay** (*int | float*)

- **save** (*str*)
- **differential** (*bool*)

Return type
str

`setReferenceActuator`(*refAct*)

Parameters
refAct (*int*)

Return type
None

`setZeros2Acts`()

`set_shape`(*cmd*, *differential=False*)

Abstract method to set the shape of the deformable mirror. Must be implemented by subclasses.

Parameters

- **cmd** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)
- **differential** (*bool*)

Return type
None

`uploadCmdHistory`(*tcmdhist*)

Abstract method to upload the command history to the deformable mirror. Must be implemented by subclasses.

Parameters
tcmdhist (*MatrixLike*)

Return type
None

class `opticalib.devices.deformable_mirrors.SplattDm`(*ip=None*, *port=None*)

Bases: `BaseDeformableMirror`

SPLATT deformable mirror interface.

Parameters

- **ip** (*str*)
- **port** (*int*)

`__init__`(*ip=None*, *port=None*)

The Constructor

Parameters

- **ip** (*str*)
- **port** (*int*)

`get_shape`()

Abstract method to get the shape of the deformable mirror. Must be implemented by subclasses.

`integratePosition`(*Nits=3*)

Parameters
`Nits` (*int*)

property nActuators: int

`plot_command`(*cmd*)

Parameters
`cmd` (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)

Return type
None

`runCmdHistory`(*interf=None, delay=0.2, save=None, differential=True, read_buffers=False*)

Abstract method to run the command history on the deformable mirror. Must be implemented by subclasses.

Parameters

- **interf** (*InterferometerDevice | None*)
- **delay** (*int | float*)
- **save** (*str | None*)
- **differential** (*bool*)
- **read_buffers** (*bool*)

Return type
str

`sendBufferCommand`(*cmd, differential=False, delay=1.0*)

Parameters

- **cmd** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)
- **differential** (*bool*)
- **delay** (*int | float*)

Return type
str

`set_shape`(*cmd, differential=False*)

Abstract method to set the shape of the deformable mirror. Must be implemented by subclasses.

Parameters

- **cmd** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)
- **differential** (*bool*)

Return type
None

uploadCmdHistory(*tcmdhist*)

Abstract method to upload the command history to the deformable mirror. Must be implemented by subclasses.

Parameters
tcmdhist (*MatrixLike*)

Return type
None

opticalib.devices.interferometer

This module contains the high-level classes for the use of interferometer devices.

Author(s)

- Pietro Ferraiuolo : pietro.ferraiuolo@inaf.it

Classes

AccuFiz([model, ip, port])	Class for the AccuFiz Laser Interferometer.
PhaseCam([model, ip, port])	Class for the 4D Twyman-Green PhaseCam Laser Interferometer.

class opticalib.devices.interferometer.**AccuFiz**(*model=None*, *ip=None*, *port=None*)

Bases: BaseInterferometer

Class for the AccuFiz Laser Interferometer.

Parameters

- **model** (*str* | *int* | *None*)
- **ip** (*str*)
- **port** (*int*)

__init__(*model=None*, *ip=None*, *port=None*)

The constructor

Parameters

- **model** (*str* | *int* | *None*)
- **ip** (*str*)
- **port** (*int*)

acquireFullFrame(***kwargs*)

Wrapper for the consecutive execution of *acquire_mapo* and *intoFullFrame*.

Parameters
****kwargs** (*dict*) – Additional keyword arguments to be passed to *acquire_map*.

Returns
The full frame image data.

Return type
_ot.ImageData

`acquire_detector`(*nframes=1*, *delay=0*)

Parameters

- **nframes** (*int*) – number of frames
- **delay** (*int | flaot [s]*) – delay between images

Returns
data2d – detector interferometer image

Return type
numpy masked array

`acquire_map`(*nframes=1*, *delay=0*, *rebin=1*)

Acquires the interferometer image and returns it as a masked array.

Parameters

- **nframes** (*int*) – Number of frames to be averaged that produce the measurement.
- **delay** (*int*) – Delay between images in seconds.
- **rebin** (*int*) – Rebin factor for the image.

Returns
masked_ima – Interferometer image.

Return type
ImageData

`capture`(*numberOfFrames*, *folder_name=None*)

Parameters

- **numberOfFrames** (*int*) – number of frames to acquire
- **folder_name** (*string*) – if None a tacking number is generate

Returns
folder_name – name of folder measurements

Return type
string

`copy4DSettings`(*destination*)

Copies the interferometer settings file to the specified destination.

Parameters
destination (*str*)

Return type
None

`getCameraSettings`()

Reads che actual interferometer settings from its configuration file.

Returns

- **output** (*list*)
- **list of camera settings** (*[width_pixel, height_pixel, offset_x, offset_y]*)

Return type
list[int]

`getFrameRate()`

Reads the frame rate the interferometer is working at.

Returns
frame_rate – Frame rate of the interferometer

Return type
float

`intoFullFrame(img)`

The function fits the passed frame (expected cropped) into the full interferometer frame (2048x2048), after reading the cropping parameters.

Parameters
img (*ImageData*) – The image to be fitted into the full frame.

Returns
output – The output image, in the interferometer full frame.

Return type
ImageData

`loadConfiguration(conffile)`

Read and loads the configuration file of the interferometer.

Parameters
conffile (*string*) – name of the configuration file to load

Return type
None

`produce(tn)`

Parameters

- **folder_name** (*string*) – name of folder measurements to convert
- **tn** (*str | list[str]*)

Return type
None

`setTriggerMode(enable)`

Parameters

- **folder_name** (*string*) – name of folder measurements to convert
- **enable** (*bool*)

Return type
None

`class opticalib.devices.interferometer.PhaseCam(model=None, ip=None, port=None)`

Bases: `BaseInterferometer`

Class for the 4D Twyman-Green PhaseCam Laser Interferometer.

Parameters

- **model** (*str | int | None*)
- **ip** (*str*)
- **port** (*int*)

`__init__`(*model=None*, *ip=None*, *port=None*)

The constructor

Parameters

- **model** (*str | int | None*)
- **ip** (*str*)
- **port** (*int*)

`acquireFullFrame`(***kwargs*)

Wrapper for the consecutive execution of *acquire_mapo* and *intoFullFrame*.

Parameters

****kwargs** (*dict*) – Additional keyword arguments to be passed to *acquire_map*.

Returns

The full frame image data.

Return type

_ot.ImageData

`acquire_detector`(*nframes=1*, *delay=0*)

Parameters

- **nframes** (*int*) – number of frames
- **delay** (*int | flaot [s]*) – delay between images

Returns

data2d – detector interferometer image

Return type

numpy masked array

`acquire_map`(*nframes=1*, *delay=0*, *rebin=1*)

Acquires the interferometer image and returns it as a masked array.

Parameters

- **nframes** (*int*) – Number of frames to be averaged that produce the measurement.
- **delay** (*int*) – Delay between images in seconds.
- **rebin** (*int*) – Rebin factor for the image.

Returns

masked_ima – Interferometer image.

Return type

ImageData

`capture`(*numberOfFrames*, *folder_name=None*)

Parameters

- **numberOfFrames** (*int*) – number of frames to acquire
- **folder_name** (*string*) – if None a tacking number is generate

Returns

folder_name – name of folder measurements

Return type
string

`copy4DSettings`(*destination*)

Copies the interferometer settings file to the specified destination.

Parameters
destination (*str*)

Return type
None

`getCameraSettings`()

Reads che actual interferometer settings from its configuration file.

Returns

- **output** (*list*)
- **list of camera settings** (*[width_pixel, height_pixel, offset_x, offset_y]*)

Return type
list[int]

`getFrameRate`()

Reads the frame rate the interferometer is working at.

Returns
frame_rate – Frame rate of the interferometer

Return type
float

`intoFullFrame`(*img*)

The function fits the passed frame (expected cropped) into the full interferometer frame (2048x2048), after reading the cropping parameters.

Parameters
img (*ImageData*) – The image to be fitted into the full frame.

Returns
output – The output image, in the interferometer full frame.

Return type
ImageData

`loadConfiguration`(*conffile*)

Read and loads the configuration file of the interferometer.

Parameters
conffile (*string*) – name of the configuration file to load

Return type
None

`produce`(*tn*)

Parameters

- **folder_name** (*string*) – name of folder measurements to convert
- **tn** (*str | list[str]*)

Return type
None

`setTriggerMode`(*enable*)

Parameters

- **folder_name** (*string*) – name of folder measurements to convert
- **enable** (*bool*)

Return type
None

2.1.8 opticalib.dmutils

DMUTILS subpackage

2024

Author(s):

- Pietro Ferraiuolo: pietro.ferraiuolo@inaf.it
- Runa Briguglio: runa.briguglio@inaf.it

Description:

This subpackage contains all the utility modules concerning a Deformable Mirror, which are its calibration and flattening.

Contents:

- *iff_acquisition_preparation.py*: Module for preparing the acquisition of the Influence Functions.
- *iff_processing.py*: Module for processing the Influence Functions.
- *iff_module.py*: high level module for managing the acquisition of IFFs.
- *flattening.py*: module containing the procedures for flattening a DM.

`class opticalib.dmutils.Flattening`(*tn*)

Bases: `object`

Class for computing and applying flattening commands to deformable mirrors.

Overview

This class manages the process of flattening a deformable mirror using an interaction cube and a reference shape (typically acquired from an interferometer). It supports loading and filtering interaction cubes, aligning and processing images, computing reconstruction matrices, and generating the appropriate command to flatten the mirror surface.

Key Features

- Loads and filters interaction cubes based on Zernike modes.
- Aligns input images to the interaction cube mask for accurate command computation.
- Computes the reconstruction matrix using SVD, with options to discard modes or set thresholds.
- Calculates the flattening command for a given shape and applies it to the deformable mirror.

- Saves all relevant data (commands, images, metadata) for traceability and reproducibility.

Public Methods

- **applyFlatCommand(dm, interf, modes2flat, nframes=5, modes2discard=None):**
  Acquires images, computes and applies the flattening command, and saves results.
- **computeFlatCmd(n_modes):**
  Computes the flattening command for the loaded shape and selected modes.
- **loadImage2Shape(img, compute=None):**
  Loads a new image to flatten and optionally computes the reconstruction matrix.
- **computeRecMat(threshold=None):**
  Computes the reconstruction matrix for the loaded image.
- **filterIntCube(zernModes=None):**
  Filters the interaction cube by removing specified Zernike modes.
- **loadNewTn(tn):**
  Loads a new tracking number and updates internal data.

Usage Example

```
>>> f = Flattening('20240906_110000')
>>> img = interf.acquire_map()
>>> f.loadImage2Shape(img)
>>> f.computeRecMat()
>>> flatCmd = f.computeFlatCmd(10)
>>> f.applyFlatCommand(dm, interf, modes2flat=10)
```

`__init__`(*tn*)

The Constructor

Parameters
`tn` (*str*)

`applyFlatCommand`(*dm*, *interf*, *modes2flat*, *modes2discard=None*, *cmdOffset=None*, *nframes=5*)

Parameters

- **`dm`** (*DeformableMirrorDevice*)
- **`interf`** (*InterferometerDevice*)
- **`modes2flat`** (*int | Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)
- **`modes2discard`** (*int | None*)
- **`cmdOffset`** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None*)
- **`nframes`** (*int*)

Return type
None

`computeFlatCmd`(*n_modes*)

Compute the command to apply to flatten the input shape.

Parameters
`n_modes` (*int | ArrayLike*) – Number of modes used to compute the flat command. If int, it will compute the first n_modes of the command matrix. If list, it will compute the flat command for the given modes.

Returns
flat_cmd – Flat command.

Return type
ndarray

`computeRecMat`(*threshold=None*)

Compute the reconstruction matrix for the loaded image.

Parameters
`threshold` (*int | float, optional*) – If not None, it can be either the number of modes to discard from the reconstruction matrix computation (int) or the threshold value to discard computed eigenvalues for the reconstruction (float). Default is None.

`filterIntCube`(*zernModes=None*)

Filter the interaction cube with the given zernike modes

Parameters
`zernModes` (*list of int | ArrayLike, optional*) – Zernike modes to filter out this cube (if it's not already filtered). Default modes are [1,2,3] -> piston/tip/tilt.

Return type
Flattening

`loadImage2Shape`(*img*, *compute=None*)

(Re)Loader for the image to flatten.

Parameters

- `img` (*ImageData*) – Image to flatten.
- `compute` (*int | float, optional*) – If not None, it can be either the number of modes to discard from the reconstruction matrix computation (int) or the threshold value to discard computed eigenvalues for the reconstruction (float). Default is None.

Return type
None

`loadNewTn`(*tn*)

Load a new tracking number for the flattening.

Parameters
`tn` (*str*) – Tracking number of the new data.

Return type
None

Parameters
`tn` (*str*)

class opticalib.dmutils.**IFFCapturePreparation**(*dm*)

Bases: object

Class containing all the functions necessary to create the final timed command matrix history to be executed by M4

Import and Initialization

Import the module and initialize the class with a deformable mirror object

```
>>> from opticalib.dmutils.iff_acquisition_preparation import IFFCapturePreparation
>>> from opticalib.devices import AlpaoDm
>>> dm = AlpaoDm(88)
>>> ifa = IFFCapturePreparation(dm)
```

createTimedCmdHistory()

Creates the final timed command matrix history. Takes 4 positional optional arguments, which will be read from a configuration file if not passed

Parameters

- **modesList** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None*)
- **modesAmp** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None*)
- **template** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None*)
- **shuffle** (*bool*)

Return type

MatrixLike

createCmdMatrixhistory()

Takes the modal base loaded into the class (which can be updated using the sub-method _updateModalBase) and returns the wanted command matrix with the dedired modes and amplitudes, which can be either passed on as arguments or read automatically from a configuration file.

```
>>> # As example, wanting to update the modal base using a zonal one
>>> ifa._updateModalBase('zonal')
'Using zonal modes'
```

createAuxCmdHistory()

Creates the auxiliary command matrix to attach to the command matrix history. This auxiliary matrix comprehends the trigger padding and the registration padding schemes. the parameters on how to create these schemes is written in a configuration file.

Return type

MatrixLike

`getInfoToSave()`

A function that returns a dictionary containing all the useful information to save, such as the command matrix used, the used mode list, the indexing the amplitudes, the used tamplate and the shuffle option.

Return type
dict[str, *Any*]

`__init__`(*dm*)

The Constructor

Parameters
`dm` (*DeformableMirrorDevice*)

`createAuxCmdHistory()`

Creates the initial part of the final command history matrix that will be passed to M4. This includes the Trigger Frame, the first frame to have a non-zero command, and the Padding Frame, two frames with high rms, useful for setting a start to the real acquisition.

Result

aus_cmdHistory
[MatrixLike] The auxiliary command history, which includes the trigger padding and the registration pattern. This matrix is used to create the final command history to be passed to the DM.

Return type
MatrixLike

`createCmdMatrixHistory`(*mlist=None*, *modesAmp=None*, *template=None*, *shuffle=False*)

Creates the command matrix history for the IFF acquisition.

Parameters

- **mlist** (*ArrayLike*) – List of selected modes to use. If no argument is passed, it will
- **modesAmp** (*float | ArrayLike*) – Amplitude of the modes to be commanded. If no argument is passed, it will be loaded from the configuration file iffConfig.ini
- **template** (*ArrayLike*) – Template for the push-pull application of the modes. If no argument is passed, it will be loaded from the configuration file iffConfig.ini
- **shuffle** (*bool*) – Decides to wether shuffle or not the order in which the modes are applied. Default is False

Returns
cmd_matrixHistory – Command matrix history to be applied, with the correct push-pull application, following the desired template.

Return type
MatrixLike

`createTimedCmdHistory`(*modesList=None*, *modesAmp=None*, *template=None*, *shuffle=False*)

Function that creates the final timed command history to be applied

Parameters

- **modesList** (*int | ArrayLike*) – List of selected modes to use. Default is None, that means all modes of the base command matrix are used.
- **modesAmp** (*float*) – Amplitude of the modes. Default is None, that means the value is loaded from the 'iffconfig.ini' file

- **template** (*int | ArrayLike*) – Template for the push-pull measures. List of 1 and -1. Default is None, which means the template is loaded from the 'iffcongig.ini' file.
- **shuffle** (*boolean*) – Decide wether to shuffle or not the modes order. Default is False

Returns
timedCmdHist – Final timed command history, including the trigger padding, the registration pattern and the command matrix history.

Return type
float | ArrayLike

`getInfoToSave()`

Return the data to save as fits files, arranged in a dictionary

Returns
info – Dictionary containing all the vectors and matrices needed

Return type
dict

Parameters
dm (*DeformableMirrorDevice*)

Modules

actuator_identification_lib	
flattening	Module containing the class which computes the flattening command for a deformable mirror, given an imput shape and a (filtered) interaction cube.
iff_acquisition_preparation	This module contains the IFFCapturePreparation class, a class which serves as a preparator for the Influence Function acquisition, creating the timed command matrix history that will be ultimately used.
iff_module	This module contains the necessary high/user-leve functions to acquire the IFF data, given a deformable mirror and an interferometer.
iff_processing	Module containing all the functions necessary to process the data acquired for the Influence Function measurements.
pupil_calibration	Author(s) Pietro Ferraiuolo : written in 2025 Matteo Menessini
stitching	

opticalib.dmutils.actuator_identification_lib

Functions

combineMasks(imglist)	combine masks layers of masked arrays, or a list of masks, to produce the intersection masks: not masked here AND not mnaked there masks are expected as in the np.ma convention: True when not masked :returns: intersection mask
extractPeak(img[, radius])	Extract a circular area around the peak in the image
findActuator(img)	Finds the coordinates of an actuator, given the image with the InfFunction masked around the act. img: masked array image where the act is to be searched Return imgout: array coordinates of the act.
findFrameCoord(imglist, actlist, actcoord)	returns the position of given actuators from a list of frames
marker_general_remap(cghf, ottf, pos2t)	transforms the pos2t coordinates, using the cghf and ottf coordinates to create the trnasformation

`opticalib.dmutils.actuator_identification_lib.`**`combineMasks`**(*imglist*)

combine masks layers of masked arrays, or a list of masks, to produce the intersection masks: not masked here AND not mnaked there masks are expected as in the np.ma convention: True when not masked :returns: intersection mask

`opticalib.dmutils.actuator_identification_lib.`**`extractPeak`**(*img*, *radius=50*)

Extract a circular area around the peak in the image

`opticalib.dmutils.actuator_identification_lib.`**`findActuator`**(*img*)

Finds the coordinates of an actuator, given the image with the InfFunction masked around the act. img: masked array

image where the act is to be searched

Return imgout: array

coordinates of the act

`opticalib.dmutils.actuator_identification_lib.`**`findFrameCoord`**(*imglist*, *actlist*, *actcoord*)

returns the position of given actuators from a list of frames

`opticalib.dmutils.actuator_identification_lib.`**`marker_general_remap`**(*cghf*, *ottf*, *pos2t*)

transforms the pos2t coordinates, using the cghf and ottf coordinates to create the trnasformation

opticalib.dmutils.flattening

Module containing the class which computes the flattening command for a deformable mirror, given an imput shape and a (filtered) interaction cube.

Author(s)

- Pietro Ferraiuolo : written in 2024

Description

From the loaded tracking number (tn) the interaction cube will be loaded (and filtered, if it's not already) from which the interaction matrix will be computed. If an image to shape is provided on class instance, then the reconstructor will be automatically computed, while if not, the load_img2shape methos is available to upload a shape from which compute the reconstructor.

How to Use it

Instancing the class only with the tn of the interaction cube

`python from opticalib.dmutils import flattening as flt tn = '20240906_110000' # example tn f = flt.Flattening(tn) # say we have acquired an image img = interf.acquire_map() f.load_image2shape(img) f.computeRecMat() 'Computing reconstruction matrix...' `

all is ready to compute the flat command, by simply running the method

`python flatCmd = f.computeFlatCmd() `

Update : all the steps above have been wrapped into the *applyFlatCommand* method, which will also save the flat command and the images used for the computation in a dedicated folder in the flat root folder.

Classes

Flattening(tn)	Class for computing and applying flattening commands to deformable mirrors.

class opticalib.dmutils.flattening.**Flattening**(*tn*)

Bases: object

Class for computing and applying flattening commands to deformable mirrors.

Overview

This class manages the process of flattening a deformable mirror using an interaction cube and a reference shape (typically acquired from an interferometer). It supports loading and filtering interaction cubes, aligning and processing images, computing reconstruction matrices, and generating the appropriate command to flatten the mirror surface.

Key Features

- Loads and filters interaction cubes based on Zernike modes.
- Aligns input images to the interaction cube mask for accurate command computation.
- Computes the reconstruction matrix using SVD, with options to discard modes or set thresholds.
- Calculates the flattening command for a given shape and applies it to the deformable mirror.
- Saves all relevant data (commands, images, metadata) for traceability and reproducibility.

Public Methods

- **applyFlatCommand(dm, interf, modes2flat, nframes=5, modes2discard=None):**
  Acquires images, computes and applies the flattening command, and saves results.
- **computeFlatCmd(n_modes):**
  Computes the flattening command for the loaded shape and selected modes.
- **loadImage2Shape(img, compute=None):**
  Loads a new image to flatten and optionally computes the reconstruction matrix.
- **computeRecMat(threshold=None):**
  Computes the reconstruction matrix for the loaded image.

- **filterIntCube(zernModes=None):**
  Filters the interaction cube by removing specified Zernike modes.
- **loadNewTn(tn):**
  Loads a new tracking number and updates internal data.

Usage Example

```
>>> f = Flattening('20240906_110000')
>>> img = interf.acquire_map()
>>> f.loadImage2Shape(img)
>>> f.computeRecMat()
>>> flatCmd = f.computeFlatCmd(10)
>>> f.applyFlatCommand(dm, interf, modes2flat=10)
```

`__init__`(*tn*)

The Constructor

Parameters
`tn` (*str*)

`applyFlatCommand`(*dm*, *interf*, *modes2flat*, *modes2discard=None*, *cmdOffset=None*, *nframes=5*)

Parameters

- `dm` (*DeformableMirrorDevice*)
- `interf` (*InterferometerDevice*)
- `modes2flat` (*int | Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)
- `modes2discard` (*int | None*)
- `cmdOffset` (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None*)
- `nframes` (*int*)

Return type
None

`computeFlatCmd`(*n_modes*)

Compute the command to apply to flatten the input shape.

Parameters
`n_modes` (*int | ArrayLike*) – Number of modes used to compute the flat command. If int, it will compute the first n_modes of the command matrix. If list, it will compute the flat command for the given modes.

Returns
flat_cmd – Flat command.

Return type
ndarray

`computeRecMat`(*threshold=None*)

Compute the reconstruction matrix for the loaded image.

Parameters

`threshold` (*int | float, optional*) – If not None, it can be either the number of modes to discard from the reconstruction matrix computation (int) or the threshold value to discard computed eigenvalues for the reconstruction (float). Default is None.

`filterIntCube`(*zernModes=None*)

Filter the interaction cube with the given zernike modes

Parameters

`zernModes` (*list of int | ArrayLike, optional*) – Zernike modes to filter out this cube (if it's not already filtered). Default modes are [1,2,3] -> piston/tip/tilt.

Return type

Flattening

`loadImage2Shape`(*img*, *compute=None*)

(Re)Loader for the image to flatten.

Parameters

- `img` (*ImageData*) – Image to flatten.
- `compute` (*int | float, optional*) – If not None, it can be either the number of modes to discard from the reconstruction matrix computation (int) or the threshold value to discard computed eigenvalues for the reconstruction (float). Default is None.

Return type

None

`loadNewTn`(*tn*)

Load a new tracking number for the flattening.

Parameters

`tn` (*str*) – Tracking number of the new data.

Return type

None

Parameters

`tn` (*str*)

opticalib.dmutils.iff_acquisition_preparation

This module contains the IFFCapturePreparation class, a class which serves as a preparator for the Influence Function acquisition, creating the timed command matrix history that will be ultimately used.

Author(s):

- Pietro Ferraiuolo: pietro.ferraiuolo@inaf.it

Classes

IFFCapturePreparation(dm)	Class containing all the functions necessary to create the final timed command matrix history to be executed by M4

class opticalib.dmutils.iff_acquisition_preparation.**IFFCapturePreparation**(*dm*)

Bases: object

Class containing all the functions necessary to create the final timed command matrix history to be executed by M4

Import and Initialization

Import the module and initialize the class with a deformable mirror object

```
>>> from opticalib.dmutils.iff_acquisition_preparation import IFFCapturePreparation
>>> from opticalib.devices import AlpaoDm
>>> dm = AlpaoDm(88)
>>> ifa = IFFCapturePreparation(dm)
```

createTimedCmdHistory()

Creates the final timed command matrix history. Takes 4 positional optional arguments, which will be read from a configuration file if not passed

Parameters

- **modesList** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None*)
- **modesAmp** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None*)
- **template** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None*)
- **shuffle** (*bool*)

Return type

MatrixLike

createCmdMatrixhistory()

Takes the modal base loaded into the class (which can be updated using the sub-method _updateModalBase) and returns the wanted command matrix with the dedired modes and amplitudes, which can be either passed on as arguments or read automatically from a configuration file.

```
>>> # As example, wanting to update the modal base using a zonal one
>>> ifa._updateModalBase('zonal')
'Using zonal modes'
```

createAuxCmdHistory()

Creates the auxiliary command matrix to attach to the command matrix history. This auxiliary matrix comprehends the trigger padding and the registration padding schemes. the parameters on how to create these schemes is written in a configuration file.

Return type
MatrixLike

`getInfoToSave()`

A function that returns a dictionary containing all the useful information to save, such as the command matrix used, the used mode list, the indexing the amplitudes, the used tamplate and the shuffle option.

Return type
dict[str, *Any*]

`__init__`(*dm*)

The Constructor

Parameters
`dm` (*DeformableMirrorDevice*)

`createAuxCmdHistory()`

Creates the initial part of the final command history matrix that will be passed to M4. This includes the Trigger Frame, the first frame to have a non-zero command, and the Padding Frame, two frames with high rms, useful for setting a start to the real acquisition.

Result

aus_cmdHistory
[MatrixLike] The auxiliary command history, which includes the trigger padding and the registration pattern. This matrix is used to create the final command history to be passed to the DM.

Return type
MatrixLike

`createCmdMatrixHistory`(*mlist=None*, *modesAmp=None*, *template=None*, *shuffle=False*)

Creates the command matrix history for the IFF acquisition.

Parameters

- **mlist** (*ArrayLike*) – List of selected modes to use. If no argument is passed, it will
- **modesAmp** (*float | ArrayLike*) – Amplitude of the modes to be commanded. If no argument is passed, it will be loaded from the configuration file iffConfig.ini
- **template** (*ArrayLike*) – Template for the push-pull application of the modes. If no argument is passed, it will be loaded from the configuration file iffConfig.ini
- **shuffle** (*bool*) – Decides to wether shuffle or not the order in which the modes are applied. Default is False

Returns
cmd_matrixHistory – Command matrix history to be applied, with the correct push-pull application, following the desired template.

Return type
MatrixLike

`createTimedCmdHistory`(*modesList=None*, *modesAmp=None*, *template=None*, *shuffle=False*)

Function that creates the final timed command history to be applied

Parameters

- **modesList** (*int | ArrayLike*) – List of selected modes to use. Default is None, that means all modes of the base command matrix are used.
- **modesAmp** (*float*) – Amplitude of the modes. Default is None, that means the value is loaded from the 'iffconfig.ini' file
- **template** (*int | ArrayLike*) – Template for the push-pull measures. List of 1 and -1. Default is None, which means the template is loaded from the 'iffcongig.ini' file.
- **shuffle** (*boolean*) – Decide wether to shuffle or not the modes order. Default is False

Returns
timedCmdHist – Final timed command history, including the trigger padding, the registration pattern and the command matrix history.

Return type
float | ArrayLike

`getInfoToSave()`

Return the data to save as fits files, arranged in a dictionary

Returns
info – Dictionary containing all the vectors and matrices needed

Return type
dict

Parameters
dm (*DeformableMirrorDevice*)

opticalib.dmutils.iff_module

This module contains the necessary high/user-leve functions to acquire the IFF data, given a deformable mirror and an interferometer.

Author(s):

- Pietro Ferraiuolo: pietro.ferraiuolo@inaf.it
- Runa Briguglio: runa.briguglio@inaf.it

Functions

iffDataAcquisition(dm, interf[, modesList, ...])	This is the user-lever function for the acquisition of the IFF data, given a deformable mirror and an interferometer.

`opticalib.dmutils.iff_module.iffDataAcquisition`(*dm*, *interf*, *modesList=None*, *amplitude=None*, *cmdOffset=None*, *template=None*, *shuffle=False*)

This is the user-lever function for the acquisition of the IFF data, given a deformable mirror and an interferometer.

Except for the devices, all the arguments are optional, as, by default, the values are taken from the *iffConfig.ini* configuration file.

Parameters

- **dm** (*DeformableMirrorDevice*) – The inizialized deformable mirror object

- **interf** (*InterferometerDevice*) – The initialized interferometer object to take measurements
- **modesList** (*ArrayLike , optional*) – list of modes index to be measured, relative to the command matrix to be used
- **amplitude** (*float | ArrayLike, optional*) – command amplitude
- **template** (*ArrayLike , oprional*) – template file for the command matrix
- **shuffle** (*bool , optional*) – if True, shuffle the modes before acquisition
- **cmdOffset** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None*)

Returns
tn – The tracking number of the dataset acquired, saved in the OPDImages folder

Return type
str

opticalib.dmutils.iff_processing

Module containing all the functions necessary to process the data acquired for the Influence Function measurements.

Author(s):

- Pietro Ferraiuolo: pietro.ferraiuolo@inaf.it
- Runa Briguglio: runa.briguglio@inaf.it

High-level Functions

process(tn, registration=False, roi=None, save=False, rebin=1)
Function that processes the data contained in the OPDImages/tn folder. By performing the differential algorithm, it produces fits images for each commanded mode into the IFFunctions/tn folder, and creates a cube from these into INTMatrices/tn. If 'registration is not False', upon createing the cube, the registration algorithm is performed.

stackCubes(tnlist)
Function that, given as imput a tracking number list containing cubes data, will stack the found cubes into a new one with a new tracking number, into INTMatrices/new_tn. A 'flag.txt' file will be created to give more information on the process.

Example

```
`python tn1 = '20160516_114916' tn2 = '20160516_114917' # A copy of tn1 (simulated) data ifp.process(tn1, save=True) Cube saved in '.../path/to/data/OPTData/INTMatrices/20160516_114916/IMcube.fits' ifp.process(tn2, save=True) Cube saved in '.../path/to/data/OPTData/INTMatrices/20160516_114917/IMcube.fits' tnlist = [tn1, tn2] ifp.stackCubes(tnlist) Stacked cube and matrices saved in '.../path/to/data/OPTData/INTMatrices/'new_tn'/IMcube.fits' `
```

Functions

createMasterMask(cube)	Function which creates a master mask for the cube, by performing a logical OR operation between the masks of each image in the cube.
filterZernikeCube(tn[, zern_modes, save, ...])	Function which filters out the desired zernike modes from a cube.
findFrameOffset(tn, imglist, actlist)	This function computes the position difference between the current frame and a reference one.
getIffFileMatrix(tn[, roi])	Creates the iffMat
getRegFileMatrix(tn[, roi])	Search for the registration frames in the images file list, and creates the registration file matrix.
getTriggerFrame(tn[, amplitude, roi])	Analyze the tracking number's images list and search for the trigger frame.
iffRedux(tn, fileMat, ampVect, modeList, ...)	Reduction function that performs the push-pull analysis on each mode, saving out the final processed image for each mode. The differential algorithm for each mode is the sum over the push-pull realizations of the images, and it is performed as follows:
process(tn[, register, roi, save, rebin])	High level function with processes the data contained in the given tracking number OPDimages folder, performing the differential algorithm and saving the final cube.
pushPullRedux(fileVec, template[, shuffle])	Performs the basic operation of processing PushPull data.
registrationRedux(tn, fileMat)	Reduction function that performs the push-pull analysis on the registration data.
saveCube(tn[, rebin, register, cube_header])	Creates and save a cube from the fits files contained in the tn folder, along with the command matrix and the modes vector fits.
stackCubes(tnlist)	Stack the cubes sontained in the corresponding tracking number folder, creating a new cube, along with stacked command matrix and modes vector.

opticalib.dmutils.iff_processing.**createMasterMask**(*cube*)

Function which creates a master mask for the cube, by performing a logical OR operation between the masks of each image in the cube.

Parameters
cube (*masked_array*) – Cube from which create the master mask.

Returns
master_mask – Master mask created from the cube.

Return type
masked_array

opticalib.dmutils.iff_processing.**filterZernikeCube**(*tn*, *zern_modes=None*, *save=True*, *cube_header=None*)

Function which filters out the desired zernike modes from a cube.

Parameters

- **tn** (*str*) – Tracking number of the cube to filter.
- **zern_modes** (*list, optional*) – List of zernike modes to filter out. The default is [1,2,3] (piston, tip and tilt).

- **save** (*bool*)
- **cube_header** (*dict[str, Any] | Header | None*)

Returns

- **ffcube** (*masked array*) – Filtered cube.
- **new_tn** (*str*) – Tracking Number of the new folder where the filtered cube is saved.

Return type
tuple[*CubeData*, str]

`opticalib.dmutils.iff_processing.`**`findFrameOffset`**`(`*tn*, *imglist*, *actlist*`)`

This function computes the position difference between the current frame and a reference one.

Parameters

- **tn** (*str*) – Tracking number
- **imglist** (*list | masked arrays*) – List of the actuator images to be used
- **actlist** (*int | array*) – List of actuators (index)

Returns
dp – Position difference

Return type
float

`opticalib.dmutils.iff_processing.`**`getIffFileMatrix`**`(`*tn*, *roi=None*`)`

Creates the iffMat

Parameters

- **tn** (*str*) – Tracking number of the data in the OPDImages folder.
- **roi** (*int*)

Returns
iffMat – A matrix of images in string format, conatining all the images for the IFF acquisition, that is all the modes with each push-pull realization. It has shape (modes, n_push_pull)

Return type
ndarray

`opticalib.dmutils.iff_processing.`**`getRegFileMatrix`**`(`*tn*, *roi=None*`)`

Search for the registration frames in the images file list, and creates the registration file matrix.

Parameters

- **tn** (*str*) – Tracking number of the data in the OPDImages folder.
- **roi** (*int*)

Returns

- **regEnd** (*int*) – Index which identifies the last registration frame in the images file list.
- **regMat** (*ndarray*) – A matrix of images in string format, containing the registration frames. It has shape (registration_modes, n_push_pull).

Return type
tuple[int, *Buffer* | *_SupportsArray*[*dtype*[*Any*]] | *_NestedSequence*[*_SupportsArray*[*dtype*[*Any*]]] | bool | int | float | complex | str | bytes | *_NestedSequence*[bool | int | float | complex | str | bytes]]

opticalib.dmutils.iff_processing.**getTriggerFrame**(*tn*, *amplitude=None*, *roi=None*)

Analyze the tracking number's images list and search for the trigger frame.

Parameters

- **tn** (*str*) – Tracking number of the data in the OPDImages folder.
- **amplitude** (*int or float, optional*) – Amplitude of the commanded trigger mode, which serves as the check value for finding the frame. If no value is passed it is loaded from the iffConfig.ini file.
- **roi** (*int*)

Returns

trigFrame – Index which identifies the trigger frame in the images folder file list.

Return type

int

Raises

RuntimeError – Error raised if the file iteration goes beyon the expected trigger frame wich can be inferred through the number of trigger zeros in the iffConfig.ini file.

opticalib.dmutils.iff_processing.**iffRedux**(*tn*, *fileMat*, *ampVect*, *modeList*, *template*, *shuffle=0*)

Reduction function that performs the push-pull analysis on each mode, saving out the final processed image for each mode.
 The differential algorithm for each mode is the sum over the push-pull realizations of the images, and it is performed as follows:

Math:

sum_i dfrac{I_i cdot t_i - I_{i-1}cdot t_{i-1}}{Acdot(n-1)}

Parameters

- **tn** (*str*) – Tracking number of the data in the OPDImages folder.
- **fileMat** (*ndarray*) – A matrix of images in string format, in which each row is a mode and the columns are its template realization.
- **ampVect** (*float | ArrayLike*) – Vector containing the amplitude for each commanded mode.
- **modeList** (*int | ArrayLike*) – Vector conaining the list of commanded modes.
- **template** (*int | ArrayLike*) – Template for the push-pull command actuation.
- **shuffle** (*int, optional*) – A value different from 0 activates the shuffle option, and the imput value is the number of repetition for each mode's push-pull packet. The default is 0, which means the shuffle is OFF.

Return type

None

opticalib.dmutils.iff_processing.**process**(*tn*, *register=False*, *roi=None*, *save=False*, *rebin=1*)

High level function with processes the data contained in the given tracking number OPDimages folder, performing the differential algorithm and saving the final cube.

Parameters

- **tn** (*str*) – Tracking number of the data in the OPDImages folder.
- **register** (*bool, optional*) – Parameter which enables the registration option. The default is False.

- **save_and_rebin_cube** (*bool | int | tuple, optional*) – If a bool is passed, the value is used to save the cube. If an int is passed, the value is used to rebin and save the cube. If a tuple is passed, the first value is used to save the cube, and the second to rebin it. The default is (False, 1).
- **roi** (*int*)
- **save** (*bool*)
- **rebin** (*int*)

Return type
None

`opticalib.dmutils.iff_processing.pushPullRedux(fileVec, template, shuffle=0)`

Performs the basic operation of processing PushPull data.

Parameters

- **fileVec** (*string | array*) – It is a row in the fileMat (the organized matrix of the images filename), corresponding to all the realizations of the same mode (or act), with a given template. If shuffle option has been used, the fileMat (and fileVec) shall be reorganized before running the script.
- **template** (*int | ArrayLike*) – Template for the PushPull acquisition.
- **shuffle** (*int, optional*) – A value different from 0 activates the shuffle option, and the imput value is the number of repetition for each mode's templated sampling. The default value is 0, which means the shuffle option is OFF.

Returns
image – Final processed mode's image.

Return type
masked_array

`opticalib.dmutils.iff_processing.registrationRedux(tn, fileMat)`

Reduction function that performs the push-pull analysis on the registration data.

Parameters

- **fileMat** (*ndarray*) – A matrix of images in string format, in which each row is a mode and the columns are its template realization.
- **tn** (*str*)

Returns
imgList – List of the processed registration images.

Return type
ArrayLike

`opticalib.dmutils.iff_processing.saveCube(tn, rebin=1, register=False, cube_header=None)`

Creates and save a cube from the fits files contained in the tn folder, along with the command matrix and the modes vector fits.

Parameters

- **tn** (*str*) – Tracking number of the IFFunctions data folder from which create the cu be.
- **rebin** (*int*) – Rebinning factor to apply to the images before stacking them into the cube.

- **register** (*int or tuple, optional*) – If not False, and int or a tuple of int must be passed as value, and the registration algorithm is performed on the images before stacking them into the cube. Default is False.
- **cube_header** (*dict | Header, optional*) – Header to be used for the cube. If None, a default header is created.

Returns
cube – Data cube of the images, with shape (npx, npx, nmodes).

Return type
masked_array

`opticalib.dmutils.iff_processing.stackCubes(tnlist)`

Stack the cubes sontained in the corresponding tracking number folder, creating a new cube, along with stacked command matrix and modes vector.

Parameters
tnlist (*list of str*) – List containing the tracking numbers of the cubes to stack.

Returns
stacked_cube – Final cube, stacked along the 3th axis.

Return type
masked_array

opticalib.dmutils.pupil_calibration

Author(s)

- Pietro Ferraiuolo : written in 2025
- Matteo Menessini

Description

Classes

PupilCalibrator(tn, dm)	Class to calibrate a DM given a pupil diofferent from that of the calibration data loaded.

`class opticalib.dmutils.pupil_calibration.PupilCalibrator(tn, dm)`

Bases: `object`

Class to calibrate a DM given a pupil diofferent from that of the calibration data loaded.

Parameters

- **tn** (*str*)
- **dm** (*DeformableMirrorDevice*)

`__init__(tn, dm)`

The Initiator

Parameters

- **tn** (*str*)
- **dm** (*DeformableMirrorDevice*)

Return type
None

`act_coordinates_tranformation`(*dm*, *img=None*)

Parameters

- `dm` (*DeformableMirrorDevice*)
- `img` (*ImageData | None*)

Return type
MatrixLike

`property dmCoords: MatrixLike`

Returns the actuator coordinates of the DM.

`fitShape2Command`(*target_shape*, *mask*, *remapped_IFF*)

Computes the command to obtain target_shape on the input mask using the IFF

Parameters

- `target_shape` (*ImageData*) – The shape to be commanded to the mirror
- `mask` (*ImageData of booleans*) – The mask of the commanded shape
- `remapped_IFF` (*MatrixLike*) – The masked influence functions matrix

Returns
raw_cmd – Vector of actuator commands.

Return type
ArrayLike

`maskTransform`(*mask*, *geometry*)

Transforms the given mask to the given geometry

Parameters

- `mask` (*ImageData*) – The mask to be transformed
- `geometry` (*Geometry*) – The geometry to which the mask should be transformed

Returns
transformed_mask – The transformed mask in the given geometry

Return type
ImageData

`remapIff`(*mask*, *geometry*)

Fits the IFFs to the given mask and geometry

Parameters

- `mask` (*ImageData*) – The mask to be used for remapping the IFFs
- `geometry` (*Geometry*) – The geometry to which the IFFs should be remapped

Returns
remapped_IFF – The remapped influence functions matrix

Return type
MatrixLike

slaveCoords(*raw_cmd*, *slave_ids*, *slaving_method='zero'*)

Computes the command to obtain target_shape on the input mask using the IFF

Parameters

- **raw_cmd** (*ArrayLike*) – Vector of actuator commands.
- **slave_ids** (*list*) – The list of slave actuator ids.
- **slaving_method** (*str*) – String for the slaving method to use: - 'spline' : thin plate spline interpolation - 'nearest' : nearest grid interpolation - 'zero' : set slaves to zero Default is 'zero'

Returns

cmd – Slaved actuator commands.

Return type

ArrayLike

opticalib.dmutils.stitching

Classes

StitchAcquire(dm, interf, motors)	Class to acquire images in sub-aperture mode of a mirror, to be later processed and analyzed with the *StitchAnalysis* class.
StitchAnalysis([tn])	Class to process and analyze acquisitions in sub-aperture mode of a mirror, to perform the stitching algorithm and produce stitched images.

class opticalib.dmutils.stitching.**StitchAcquire**(*dm*, *interf*, *motors*)

Bases: `object`

Class to acquire images in sub-aperture mode of a mirror, to be later processed and analyzed with the *StitchAnalysis* class.

Parameters

- **dm** (*_ot.DeformableMirrorDevice*) – The deformable mirror device used for the acquisition.
- **interf** (*_ot.InterferometerDevice*) – The interferometer device used for the acquisition.
- **motors** (*_ot.GenericDevice*) – The motor device controlling the axis of the acquisition.

acquireSingleScan(*coord_vec*, *nframes=1*, *homing=True*)

Acquire a single scan at each position in the coordinate vector.

Parameters

- **coord_vec** (*list[float]*) – A list of tuples with the coordinates (x, z) where the scan will be acquired. e.g. [(x1, z1), (x2, z2), . . .]
- **nframes** (*int, optional*) – The number of frames to acquire at each position. Default is 1.
- **homing** (*bool, optional*) – If True, the axis will be homed after the acquisition. Default is True.

Returns
tn – The tracking number (TN) of the scan, where is located the cube of acquired images at each position.

Return type
str

`acquireSubApertureIFF`(*coord_vec*)

Acquire the IFF at each position in the coordinate vector.

Parameters
`coord_vec` (*list*) – A list of tuples with the coordinates (x, z) where the IFF will be acquired.

Returns
tnvec – A list of lists, where each list contains the TN and the coordinates.

Return type
list

`getAxisPosition`()

Get the current position of the motor's axis.

Returns
A dictionary with the current position of the axis in the format: {"x": x_position, "y": y_position, "z": z_position}

Return type
dict[str, float]

`getCoordinatesVector`(*nstep*, *step_in_mm=(3, 3)*, *live_pos=False*)

Get the coordinates vector of the grid for scanning.

Parameters

- `nstep` (*int*) – The number of steps in each direction (x and z).
- `step_in_mm` (*tuple[int, int], optional*) – The step size in millimeters for the x and z directions. Default is (3, 3).
- `live_pos` (*bool, optional*) – If True, the starting position will be the current position of the axis. If False, it will use the starting coordinates defined in the constants. Default is False.

Returns
coord_vector – A list of tuples containing the coordinates (x, z) for each step in the grid. The coordinates are calculated based on the starting position and the step size.

Return type
list[tuple[float, float]]

`setAxisPosition`(*coord*)

Set the position of the motor's axis to the specified coordinates.

Parameters
`coord` (*list[float]*) – A list containing the x and z coordinates to set the axis position. e.g. [x, z]

class `opticalib.dmutils.stitching.StitchAnalysis`(*tn=None*)

Bases: `object`

Class to process and analyze acquisitions in sub-aperture mode of a mirror, to perform the stitching algorithm and produce stitched images.

Parameters
`tn` (*str* | *None*)

`__init__`(*tn=None*)

The Initiation

Parameters
`tn` (*str* | *None*)

`getCubeAndHeader`(*filepath*)

Load a cube and its header from a FITS file.

Parameters
`filepath` (*str*) – The path to the FITS file.

Returns
A tuple containing the transposed cube data (shape *[n_img,n_px,n_px]*) and the header.

Return type
tuple

`processTns`(*tnvec*)

Process the IFF obtained during the acquisition, and produces the modes and cubes for each position, and produces a cube for each IFF in different positions.

Parameters
`tnvec` (*tuple*) – A tuple of tuples, where each inner tuple contains the scan name and the coordinates e.g. ((“scan1”, (x1, z1)), (“scan2”, (x2, z2)), …)

Returns
The new TN where everything is saved

Return type
str

`reloadConstants`()

Reload the constants from the configuration file

Return type
None

`remaskCube`(*mask_radius*, *cube*, *coords*, *threshold=0.2*)

Remask all the images in the cube by intersecting a circular mask with a specified radius with the already existing one.

Parameters

- `mask_radius` (*float*) – The radius of the circular mask in millimeters.
- `cube` (*_ot.CubeData*) – The input image cube to be remasked.
- `coords` (*ArrayLike*) – The transformed coordinates written on the cube header.
- `threshold` (*float, optional*) – The pixel threshold used to reject images.Percentage of useful pixels.

Returns

- **new_cube** (*_ot.CubeData*) – The remasked image cube.
- **new_coords** (*ArrayLike*) – The updated transformed coordinates of the remasked images.

Return type
CubeData

`retrieveCubeCoords`(*n_positions*, *header*)

Returns the coordinates written in the cube's header

Parameters

- **n_positions** (*int*) – The number of pair coordinate positions in the cube.
- **header** (*dict or astropy.Header*) – The header of the cube containing the coordinates.

Returns

coords – An array of coordinates in the form of (x, z) for each position.

Return type

_ot.ArrayLike

`stitchAllIffCubes`(*tn*, ***stitchargs*)

Stitch the IFF cubes obtained during the acquisition, and produces a single cube for each IFF in different positions.

Parameters

- **tn** (*str*) – The tracking number where the IFF modes cubes are stored.
- **Parameters** (*Additional*)
- **---------------------**
- ****stitchargs** (*dict[str, _ot.Any]*) – Additional arguments for stitching, including:
- **remask** (*-*) – The new mask radius, in mm, to apply to the cube images. Default is None, meaning no remask.
- **step_size** (*-*) – The step size of the re-sampling of the iff. Default is None, meaning no re-sampling.
- **mask_threshold** (*-*) – Pixel threshold to trim images. Default is 0.2, meaning that images with less than 20% of pixels masked will be discarded.
- **deg** (*-*) – The rotation angle in degrees to apply to the coordinates. Default is None, meaning no rotation.
- **average** (*-*) – The average image to subtract from the cube images. Default is None, meaning no subtraction.

Returns

The tracking number of the stitched IFF cube.

Return type

newtn = str

`stitchSingleIffCube`(*cube*, *header*, *remask=None*, *mask_threshold=0.2*, *step_size=None*, *deg=None*, *average=None*)

Analyze a single scan and performs the stitching

Parameters

- **cube** (*_ot.CubeData*) – The cube data to be stitched.
- **header** (*dict or astropy.Header*) – The header of the cube containing the coordinates.
- **remask** (*float, optional*) – The new mask radius, in mm, to apply to the cube images. Default is False, meaning no remask.

- **mask_threshold** (*float, optional*) – Pixel threshold to trim images.
- **step_size** (*float | int, optional*) – The step size of the re-sampling of the iff.
- **average** (*np.ndarray, optional*) – The average image to subtract from the cube images.
- **deg** (*float, optional*) – The rotation angle in degrees to apply to the coordinates.

Returns
The stitched image

Return type
np.MaskedArray

`stitchSingleScansionCube`(*tn*, *deg=None*, *average=None*, *chunk_size=128*)

Analyze a single scansion cube and performs the stitching.

Parameters

- **str** (*tn;*) – The tracking number of the scansion to analyze.
- **average** (*np.ndarray, optional*) – The average image to subtract from the cube images.
- **deg** (*float, optional*) – The rotation angle in degrees to apply to the coordinates.
- **tn** (*str*)
- **chunk_size** (*int*)

Returns
The stitched image.

Return type
np.MaskedArray

2.1.9 opticalib.ground

GROUND module

2024

Author(s)

- Pietro Ferraiuolo : pietro.ferraiuolo@inaf.it
- Marco Xompero : marco.xompero@inaf.it
- Runa Briguglio : runa.briguglio@inaf.it

Description

This module provides various utility functionalities for the opticalib package, including functions for wavefront reconstruction, interaction matrix computation, and device management.

Contents

- **computerec.py**
  [module for computing and using the reconstructor from a DM] calibration. Used in *dmutils.flattening*
- **osutils.py**
  [module with various OS utilities, like reading/writing FITS] files and interferometer maps.

- logger.py : module for logging utilities.
- geo.py : module for geometric utilities, like circular masks creation.
- roi.py : module for region of interest (ROI) management.
- zernike.py : module for Zernike polynomials computation and fitting on images.

Modules

computerec	Author(s): Chiara Selmi : written in 2019 Marco Xompero : modified in 2024 Pietro Ferraiuolo : modified in 2024
geo	Autor(s) Runa Briguglio : created Mar 2020 Federico Miceli : added funcitonality on 2022 Pietro Ferraiuolo : polished on 2024
logger	Author(s) Chiara Selmi : written in 2020 Pietro Ferraiuolo : modified in 2024
osutils	Author(s) Chiara Selmi: written in 2019 Pietro Ferraiuolo: updated in 2025
roi	Author(s): Chiara Selmi: written in 2019 | rewritten in 2022 Pietro Ferraiuolo: modified in 2024
zernike	Zernike Generation Library Author(s) Tim van Werkhoven (t.i.m.vanwerkhoven@xs4all.nl) Original Author Created in 2011-10-12 Pietro Ferraiuolo (pietro.ferraiuolo@inaf.it) : Adapted in 2024

opticalib.ground.computerec

Module containing the ComputeReconstructor class, which, from a dm calibration, computes the reconstruction matrix.

Author(s):

- Marco Xompero : written in 2024
- Pietro Ferraiuolo : modified in 2024

Classes

ComputeReconstructor(interaction_matrix_cube)	This class analyzes the measurements made through the IFF class and calculates the reconstructor to be used in the control loop.

class opticalib.ground.computerec.**ComputeReconstructor**(*interaction_matrix_cube*, *mask2intersect=None*)

Bases: `object`

This class analyzes the measurements made through the IFF class and calculates the reconstructor to be used in the control loop.

HOW TO USE IT:

```
tn = "YYYYMMDD_HHMMSS"
cr = ComputeReconstructor.loadInteractionMatrix(tn)
rec = cr.run()
```

OR

cr = ComputeReconstructor(interation_matrix_cube) rec = cr.run(Interactive=True)

where the interaction_matrix_cube is a masked_array dstack of shape [pixels, pixels, n_images]

Parameters

- **interaction_matrix_cube** (*MatrixLike*)
- **mask2intersect** (*MatrixLike | None*)

__init__(*interaction_matrix_cube*, *mask2intersect=None*)

The constructor

Parameters

- **interaction_matrix_cube** (*MatrixLike*)
- **mask2intersect** (*MatrixLike | None*)

loadInteractionCube(*intCube=None*, *tn=None*)

Function intended as a reloader for the interaction matrix cube, to use a different IFF for recontructor creation.

Parameters

- **intCube** (*ndarray, optional*) – The data cube itself.
- **tn** (*str, optional*) – The tracking number where to find the data cube. Default is None.

Return type

ComputeReconstructor

loadShape2Flat(*img*)

Function intended as a reloader for the image mask to intersect, in order to create a new recontructor matrix.

Parameters

img (*ImageData*) – The image to compute the new recontructor.

Return type

ComputeReconstructor

static make_interactive_plot(*singular_values*, *current_threshold=None*)

run(*Interactive=False*, *sv_threshold=None*)

Compute the reconstruction matrix from the interaction matrix and the image to flatten.

Parameters

- **Interactive** (*bool, optional*) – If True, the function will show an interactive plot to choose the threshold for the singular values. Default is False.

- **sv_threshold** (*int | float, optional*) – The threshold for the singular values. If None, the function will compute the pseudo-inverse of the interaction matrix. If an integer is provided, it will be used as the threshold. If a float is provided, it will be used as the threshold. Default is None.

Returns
recMat – Reconstruction matrix.

Return type
MatrixLike

opticalib.ground.geo

This module contains functions for geometric operations on images.

Autor(s)

- Runa Briguglio : created Mar 2020
- Federico Miceli : added funcitonality on 2022
- Pietro Ferraiuolo : polished on 2024

Functions

draw_mask(img, cx, cy, r[, out])	Function to create circular mask Created by Runa
qpupil(mask[, xx, yy, nocircle])	Function for.
qpupil_circle(image[, pixel_dir])	Function for.

opticalib.ground.geo.**draw_mask**(*img*, *cx*, *cy*, *r*, *out=0*)

Function to create circular mask Created by Runa

Parameters

- **img** (*numpy array*) – image to mask
- **cx** (*int [pixel]*) – center x of the mask
- **cy** (*int [pixel]*) – center y of the mask
- **r** (*int [pixel]*) – radius of the mask

Returns
img1 – start image mask whit circular new mask

Return type
numpy array

opticalib.ground.geo.**qpupil**(*mask*, *xx=None*, *yy=None*, *nocircle=0*)

Function for. . . . created by Runa

Parameters
mask (*numpy array*)

Returns

- *x0*
- *y0*
- *r*

- **xx** (*numpy array*) – grid of coordinates of the same size as input mask
- **yy** (*numpy array*) – grid of coordinates of the same size as input mask

opticalib.ground.geo.**qpupil_circle**(*image*, *pixel_dir=0*)

Function for… Created by Federico NOTA: la funzione usa come standard la direzione y per determinare la dimensione dei pixel

pixel_dir: int
indicates which direction to use for counting the number of pixels in the image. Y direction as standard

opticalib.ground.logger

This module provides an easy interface for setting up scripts logging within the Opticalib framework. It includes functions to configure a rotating file logger and a simple text file logger class.

Author(s)

- Chiara Selmi : written in 2020
- Pietro Ferraiuolo : rewritten in 2024

Example Usage

To set up logging for your script, use the `set_up_logger` function to configure a rotating file logger. You can then log messages using the standard logging interface or the provided `log` function. For simple text logging, instantiate the `txtLogger` class.

Example:

```
# Set up a rotating file logger
logger = set_up_logger('my_script.log', logging.INFO)

# Log messages using the log function
log("This is an informational message.", "INFO")
log("This is a debug message.", "DEBUG")

# Use the txtLogger for simple text logging
txt_log = txtLogger('simple_log.txt')
txt_log.log("This is a message written to a text file.")
```

Functions

log(message[, level])	Log a message at the specified level.
set_up_logger(filename[, logging_level])	Set up a rotating file logger.

Classes

txtLogger(file_path)	Simple logger class for writing log messages to a text file.

opticalib.ground.logger.**log**(*message*, *level='INFO'*)

Log a message at the specified level.

Parameters

- **message** (*str*) – The message to log.
- **level** (*str, optional*) – The logging level to use for the message. This should be one of the following strings: 'DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL'. (can use lowercase too). The default is 'DEBUG'.

Return type
None

Notes

- The message will be logged using the logger configured by *set_up_logger*.
- The message will be logged with the specified level.
- If the specified level is not recognized, the message will be logged at the 'DEBUG' level.

`opticalib.ground.logger.set_up_logger(filename, logging_level=10)`

Set up a rotating file logger.

This function configures a logger to write log messages to a file with rotation. The log file will be encoded in UTF-8 and will rotate when it reaches a specified size, keeping a specified number of backup files.

Parameters

- **filename** (*str*) – The path to the log file where log messages will be written.
- **logging_level** (*int*) – The logging level to set for the logger. This should be one of the logging level constants defined in the `logging` module (e.g., `DEBUG=10, INFO=20, WARNING=30, ERROR=40, CRITICAL=50`).

Return type
Logger

Notes

- The log file will rotate when it reaches 10,000,000 bytes (10 MB).
- Up to 3 backup log files will be kept.
- The log format includes the timestamp, log level, logger name, and message.
- The logger is configured at the root level, affecting all loggers in the application.
- The handler will perform an initial rollover when set up.

Examples

```
set_up_logger('/path/to/logfile.log', logging.DEBUG)
```

`class opticalib.ground.logger.txtLogger(file_path)`

Bases: `object`

Simple logger class for writing log messages to a text file.

Parameters
file_path (*str*) – Path to the log file, including the file name.

file_path

Path to the log file.

Type

str

__init__(*file_path*)

Initializes the txtLogger with the specified file path.

Parameters

file_path (*str*) – The path to the log file.

log(*message*)

Writes the log message to the *.txt* file.

Parameters

message (*str*) – The log message to be written to the file.

Return type

None

opticalib.ground.osutils

Module containing various utility functions for handling files and directories, especially related to tracking numbers and interferometric data, within the Opticalib framework.

Author(s)

- Chiara Selmi: written in 2019
- Pietro Ferraiuolo: updated in 2025

Functions

findTracknum(tn[, complete_path])	Search for the tracking number given in input within all the data path subfolders.
getCameraSettings(tn)	Reads the interferometer settings from a given configuration file.
getFileList([tn, fold, key])	Search for files in a given tracking number or complete path, sorts them and puts them into a list.
getFrameRate(tn)	Reads the frame rate of the camera from a given configuration file.
is_tn(string)	Check if a given string is a valid tracking number or the full path of a tracking number.
load_fits(filepath[, return_header])	Loads a FITS file.
newtn()	Returns a timestamp in a string of the format *YYYYMMDD_HHMMSS*.
read_phasemap(file_path)	Function to read interferometric data, in the three possible formats (FITS, 4D, H5)
rename4D(folder)	Renames the produced 'x.4D' files into '0000x.4D'
save_fits(filepath, data[, overwrite, header])	Saves a FITS file.
tnRange(tn0, tn1)	Returns the list of tracking numbers between tn0 and tn1, within the same folder, if they both exist in it.

`opticalib.ground.osutils.findTracknum(tn, complete_path=False)`

Search for the tracking number given in input within all the data path subfolders.

Parameters

- **tn** (*str*) – Tracking number to be searched.
- **complete_path** (*bool, optional*) – Option for wheter to return the list of full paths to the folders which contain the tracking number or only their names.

Returns

tn_path – List containing all the folders (within the OPTData path) in which the tracking number is present, sorted in alphabetical order.

Return type

list of str

`opticalib.ground.osutils.getCameraSettings(tn)`

Reads the interferometer settings from a given configuration file.

Returns

output – List of camera settings: [width_pixel, height_pixel, offset_x, offset_y]

Return type

list of int

Parameters

tn (*str*)

`opticalib.ground.osutils.getFileList(tn=None, fold=None, key=None)`

Search for files in a given tracking number or complete path, sorts them and puts them into a list.

Parameters

- **tn** (*str*) – Tracking number of the data in the OPDImages folder.
- **fold** (*str, optional*) – Folder in which searching for the tracking number. If None, the default folder is the OPD_IMAGES_ROOT_FOLDER.
- **key** (*str, optional*) – A key which identify specific files to return

Returns

- **fl** (*list of str*) – List of sorted files inside the folder.
- *How to Use it*
- ———-
- *If the complete path for the files to retrieve is available, then this function*
- *should be called with the 'fold' argument set with the path, while 'tn' is*
- *defaulted to None.*
- **In any other case, the tn must be given** (*it will search for the tracking*)
- *number into the OPDImages folder, but if the search has to point another*
- *folder, then the fold argument comes into play again. By passing both the*
- *tn (with a tracking number) and the fold argument (with only the name of the*
- *folder) then the search for files will be done for the tn found in the*
- *specified folder. Hereafter there is an example, with the correct use of the*

- *key argument too.*

Return type
list[str]

Examples

Here are some examples regarding the use of the 'key' argument. Let's say w e need a list of files inside ''tn = '20160516_114916' '' in the IFFunctions folder.

```
>>> iffold = 'IFFunctions'
>>> tn = '20160516_114916'
>>> getFileList(tn, fold=iffold)
['.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/cmdMatrix.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0000.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0001.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0002.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0003.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/modesVector.fits']
```

Let's suppose we want only the list of 'mode_000x.fits' files:

```
>>> getFileList(tn, fold=iffold, key='mode_')
['.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0000.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0001.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0002.fits',
 '.../M4/m4/data/M4Data/OPTData/IFFunctions/20160516_114916/mode_0003.fits']
```

Notice that, in this specific case, it was necessary to include the undersc ore after 'mode' to exclude the 'modesVector.fits' file from the list.

`opticalib.ground.osutils.`**`getFrameRate`**`(`*tn*`)`

Reads the frame rate of the camera from a given configuration file.

Returns
frame_rate – Frame rate of the interferometer

Return type
float

Parameters
`tn` (*`str`*)

`opticalib.ground.osutils.`**`is_tn`**`(`*string*`)`

Check if a given string is a valid tracking number or the full path of a tracking number.

Parameters
`string` (*`str`*) – The string to check.

Returns
True if the string is a valid tracking number, False otherwise.

Return type
bool

`opticalib.ground.osutils.`**`load_fits`**`(`*filepath*, *return_header=False*`)`

Loads a FITS file.

Parameters

- **filepath** (*str*) – Path to the FITS file.
- **return_header** (*bool*) – Wether to return the header of the loaded fits file. Default is False.

Returns

- **fit** (*np.ndarray or np.ma.MaskedArray*) – FITS file data.
- **header** (*dict | fits.Header, optional*) – The header of the loaded fits file.

Return type

tuple[*ImageData* | *CubeData* | *MatrixLike* | *Buffer* | *_SupportsArray*[*dtype*[*Any*]] | *_NestedSequence*[*_SupportsArray*[*dtype*[*Any*]]] | bool | int | float | complex | str | bytes | *_NestedSequence*[bool | int | float | complex | str | bytes], *Any*]

`opticalib.ground.osutils.newtn()`

Returns a timestamp in a string of the format *YYYYMMDD_HHMMSS*.

Returns

Current time in a string format.

Return type

str

`opticalib.ground.osutils.read_phasemap(file_path)`

Function to read interferometric data, in the three possible formats (FITS, 4D, H5)

Parameters

file_path (*str*) – Complete filepath of the file to load.

Returns

image – Image as a masked array.

Return type

ImageData

`opticalib.ground.osutils.rename4D(folder)`

Renames the produced 'x.4D' files into '0000x.4D'

Parameters

folder (*str*) – The folder where the 4D data is stored.

Return type

None

`opticalib.ground.osutils.save_fits(filepath, data, overwrite=True, header=None)`

Saves a FITS file.

Parameters

- **filepath** (*str*) – Path to the FITS file.
- **data** (*np.array*) – Data to be saved.
- **overwrite** (*bool, optional*) – Whether to overwrite an existing file. Default is True.
- **header** (*dict[str, any] | fits.Header, optional*) – Header information to include in the FITS file. Can be a dictionary or a fits.Header object.

Return type

None

`opticalib.ground.osutils.`**`tnRange`**`(`*tn0*, *tn1*`)`

Returns the list of tracking numbers between tn0 and tn1, within the same folder, if they both exist in it.

Parameters

- **tn0** (*str*) – Starting tracking number.
- **tn1** (*str*) – Finish tracking number.

Returns

tnMat – A list or a matrix of tracking number in between the start and finish ones.

Return type

list of str

Raises

FileNotFoundError – An exception is raised if the two tracking numbers are not found in the same folder

opticalib.ground.roi

Module containing functions for region of interest (ROI) generation and other image utilities within the Opticalib framework.

Author(s):

- Pietro Ferraiuolo: pietro.ferraiuolo@inaf.it

Functions

imgCut(img)	Cuts the image to the bounding box of the finite (non-NaN) pixels in the masked image.
roiGenerator(img)	This function generates a list of *n_masks* roi from the input image.

`opticalib.ground.roi.`**`imgCut`**`(`*img*`)`

Cuts the image to the bounding box of the finite (non-NaN) pixels in the masked image.

Parameters

- **image** (*np.ma.maskedArray*) – The original masked image array.
- **img** (*ImageData*)

Returns

The cut image within the bounding box of finite pixels.

Return type

cutImg = np.ma.maskedArray

`opticalib.ground.roi.`**`roiGenerator`**`(`*img*`)`

This function generates a list of *n_masks* roi from the input image.

Parameters

img (*ImageData | np.ma.maskedArray*) – input image from which the roi are generated.

Returns

roiList – List of the first *n_masks* roi found in the image.

Return type
list

opticalib.ground.zernike

Zernike Generation Library

This module provides functions and utilities for generating Zernike polynomials, which are a sequence of polynomials that are orthogonal on the unit disk. These polynomials are commonly used in optics and wavefront analysis.

Author(s)

- Tim van Werkhoven (t.i.m.vanwerkhoven@xs4all.nl) : Original Author, Created in 2011-10-12
- Pietro Ferraiuolo (pietro.ferraiuolo@inaf.it) : Adapted in 2024

Functions

- removeZernike(ima, modes=np.array([1, 2, 3, 4])): Remove Zernike modes from an image.
- removeZernikeAuxMask(img, mm, zlist): Remove Zernike modes from an image using an auxiliary mask.
- zernikeFit(img, zernike_index_vector, qpupil=True): Fit Zernike modes to an image.
- zernikeFitAuxmask(img, auxmask, zernike_index_vector): Fit Zernike modes to an image using an auxiliary mask.
- zernikeSurface(img, coef, mat): Generate Zernike surface from coefficients and matrix.
- _surf_fit(xx, yy, zz, zlist, ordering='noll'): Fit surface using Zernike polynomials.
- _getZernike(xx, yy, zlist, ordering='noll'): Get Zernike polynomials.
- _zernike_rad(m, n, rho): Calculate the radial component of Zernike polynomial (m, n).
- _zernike(m, n, rho, phi): Calculate Zernike polynomial (m, n).
- _zernikel(j, rho, phi): Calculate Zernike polynomial with Null coordinate j.
- _l2mn_ansi(j): Convert ANSI index to Zernike polynomial indices.
- _l2mn_noll(j): Convert Noll index to Zernike polynomial indices.

Example

Example usage of the module:

`python import numpy.ma as ma # Create a sample image with a mask image_data = np.random.random((100, 100)) mask = np.zeros((100, 100), dtype=bool) mask[30:70, 30:70] = True masked_image = ma.masked_array(image_data, mask=mask) # Define Zernike modes to be removed zernike_modes = np.array([1, 2, 3, 4]) # Remove Zernike modes from the image cleaned_image = zernike.removeZernike(masked_image, zernike_modes) # Display the original and cleaned images import matplotlib.pyplot as plt plt.figure() plt.subplot(1, 2, 1) plt.title("Original Image") plt.imshow(masked_image, cmap='gray') plt.subplot(1, 2, 2) plt.title("Cleaned Image") plt.imshow(cleaned_image, cmap='gray') plt.show() `

Functions

generateZernMat(noll_ids, img_mask[, ...])	Generates the interaction matrix of the Zernike modes with Noll index in noll_ids on the mask in input
removeZernike(image[, modes])	Remove Zernike modes from an image.
removeZernikeAuxMask(image, mask, zlist)	Remove Zernike modes from an image using an auxiliary mask.
zernikeFit(image, zernike_index_vector[, qpupil])	Fit Zernike modes to an image.
zernikeFitAuxmask(image, auxmask, ...)	Fit Zernike modes to an image using an auxiliary mask.
zernikeSurface(image, coeff, mat)	Generate Zernike surface from coefficients and matrix.

opticalib.ground.zernike.**generateZernMat**(*noll_ids*, *img_mask*, *scale_length=None*)

Generates the interaction matrix of the Zernike modes with Noll index in noll_ids on the mask in input

Parameters

- **noll_ids** (*ArrayLike*) – List of (Noll) mode indices to fit.
- **img_mask** (*matrix bool*) – Mask of the desired image.
- **scale_length** (*float, optional*) – The scale length to use for the Zernike fit. The default is the maximum of the image mask shape.

Returns

ZernMat – The Zernike interaction matrix of the given indices on the given mask.

Return type

MatrixLike [n_pix,n_zern]

opticalib.ground.zernike.**removeZernike**(*image*, *modes=None*)

Remove Zernike modes from an image.

Parameters

- **image** (*numpy masked array*) – Image from which Zernike modes are to be removed.
- **modes** (*numpy array, optional*) – Zernike modes to be removed. Default is np.array([1, 2, 3, 4]).

Returns

new_ima – Image with Zernike modes removed.

Return type

numpy masked array

opticalib.ground.zernike.**removeZernikeAuxMask**(*image*, *mask*, *zlist*)

Remove Zernike modes from an image using an auxiliary mask.

Parameters

- **image** (*numpy masked array*) – Image from which Zernike modes are to be removed.
- **mask** (*numpy array*) – Auxiliary mask.
- **zlist** (*numpy array*) – List of Zernike modes to be removed.

Returns

new_ima – Image with Zernike modes removed.

Return type

numpy masked array

`opticalib.ground.zernike.`**`zernikeFit`**(*image*, *zernike_index_vector*, *qpupil=True*)

Fit Zernike modes to an image.

Parameters

- **`img`** (*`numpy masked array`*) – Image for Zernike fit.
- **`zernike_index_vector`** (*`numpy array`*) – Vector containing the index of Zernike modes to be fitted starting from 1.
- **`qpupil`** (*`bool, optional`*) – If True, use a pupil mask; otherwise, use a circular pupil. Default is True.
- **`image`** (*`ImageData`*)

Returns

- **coeff** (*numpy array*) – Vector of Zernike coefficients.
- **mat** (*numpy array*) – Matrix of Zernike polynomials.

Return type

tuple[*Buffer* | *_SupportsArray*[*dtype*[*Any*]] | *_NestedSequence*[*_SupportsArray*[*dtype*[*Any*]]] | bool | int | float | complex | str | bytes | *_NestedSequence*[bool | int | float | complex | str | bytes], *Buffer* | *_SupportsArray*[*dtype*[*Any*]] | *_NestedSequence*[*_SupportsArray*[*dtype*[*Any*]]] | bool | int | float | complex | str | bytes | *_NestedSequence*[bool | int | float | complex | str | bytes]]

`opticalib.ground.zernike.`**`zernikeFitAuxmask`**(*image*, *auxmask*, *zernike_index_vector*)

Fit Zernike modes to an image using an auxiliary mask.

Parameters

- **`img`** (*`numpy masked array`*) – Image for Zernike fit.
- **`auxmask`** (*`numpy array`*) – Auxiliary mask.
- **`zernike_index_vector`** (*`numpy array`*) – Vector containing the index of Zernike modes to be fitted starting from 1.
- **`image`** (*`ImageData`*)

Returns

- **coeff** (*numpy array*) – Vector of Zernike coefficients.
- **mat** (*numpy array*) – Matrix of Zernike polynomials.

Return type

tuple[*Buffer* | *_SupportsArray*[*dtype*[*Any*]] | *_NestedSequence*[*_SupportsArray*[*dtype*[*Any*]]] | bool | int | float | complex | str | bytes | *_NestedSequence*[bool | int | float | complex | str | bytes], *Buffer* | *_SupportsArray*[*dtype*[*Any*]] | *_NestedSequence*[*_SupportsArray*[*dtype*[*Any*]]] | bool | int | float | complex | str | bytes | *_NestedSequence*[bool | int | float | complex | str | bytes]]

`opticalib.ground.zernike.`**`zernikeSurface`**(*image*, *coeff*, *mat*)

Generate Zernike surface from coefficients and matrix.

Parameters

- **`img`** (*`numpy masked array`*) – Image for Zernike fit.
- **`coeff`** (*`numpy array`*) – Vector of Zernike coefficients.
- **`mat`** (*`numpy array`*) – Matrix of Zernike polynomials.
- **`image`** (*`ImageData`*)

Returns
surf – Zernike surface generated by coefficients.

Return type
numpy masked array

2.1.10 opticalib.typings

TYPINGS module

2025

Author(s)

- Pietro Ferraiuolo : pietro.ferraiuolo@inaf.it

Description

This module defines custom type aliases and protocols for type hinting within the *opticalib* package. It includes protocols for matrix-like objects, image data, cube data, interferometer devices, and deformable mirror devices. Additionally, it provides a custom *isinstance_* function to check if an object conforms to these protocols.

Functions

`array_str_formatter`(array)	Formats an array-like object into a string representation.
`isinstance_`(obj, class_name)	Custom *isinstance* wrapper: checks if the object is an instance of a specific class.

Classes

`InstanceCheck`()	A class to check if an object is an instance of a specific type.

class `opticalib.typings.InstanceCheck`

Bases: `object`

A class to check if an object is an instance of a specific type.

static generic_check(*obj*, *class_name*)

Generic check for any object type. Returns True if obj is an instance of the specified class, otherwise False.

Parameters

- **obj** (*Any*)
- **class_name** (*str*)

Return type
bool

static is_cube_like(*obj*)

Check if the object is a cube-like object. Returns True if obj is a 3D cube ArrayLike object with a mask, otherwise False.

Parameters
obj (*Any*)

Return type
bool

`static is_image_like(obj, ndim=2)`

Check if the object is an image-like object. Returns True if obj is a 2D image ArrayLike object with a mask, otherwise False.

Parameters

- **obj** (*Any*)
- **ndim** (*int*)

Return type
bool

`static is_matrix_like(obj)`

Check if the object is a matrix-like object. Returns True if obj is a 2D matrix-like object, otherwise False.

Parameters
obj (*Any*)

Return type
bool

`classmethod isinstance_(obj, class_name)`

Custom *isinstance* wrapper: checks if the object is an instance of a specific class.

Parameters

- **class_name** (*str*) – The name of the class to check against.
- **obj** (*Any*) – The object to check.

Returns
True if obj is an instance of the specified class, otherwise False.

Return type
bool

`opticalib.typings.array_str_formatter(array)`

Formats an array-like object into a string representation.

Parameters

- **arr** (*ArrayLike os list[ArrayLike]*) – The array-like object to be formatted.
- **array** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | list[Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]]*)

Returns
array_strs – The string representation of the array(s).

Return type
str

`opticalib.typings.isinstance_`(*obj*, *class_name*)

Custom *isinstance* wrapper: checks if the object is an instance of a specific class.

Parameters

- **class_name** (*str*) – The name of the class to check against.
- **obj** (*Any*) – The object to check.

Returns

True if obj is an instance of the specified class, otherwise False.

Return type

bool

PYTHON MODULE INDEX

O

INDEX