PynPoint Documentation

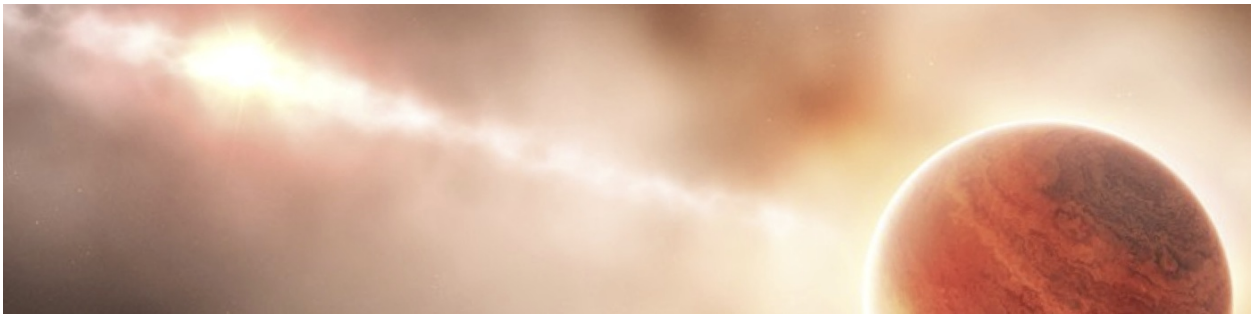
Release 0.8.1

Tomas Stolker, Markus Bonse, Sascha Quanz, and Adam Amara

Jul 17, 2019

Getting Started

PynPoint is a pipeline for processing and analysis of high-contrast imaging data of exoplanets and circumstellar disks. The Python package has been developed at the of ETH Zurich in a collaboration between the and the .



CHAPTER 1

Installation

PynPoint is compatible with Python 3.6 and 3.7. Earlier versions (up to v0.7.0) are also compatible with Python 2.7. We highly recommend using Python 3 since several key Python projects have already Python 2.

1.1 Virtual Environment

PynPoint is available in the and on . We recommend using a Python virtual environment to install and run PynPoint such that the correct versions of the dependencies can be installed without affecting other installed Python packages. First install *virtualenv*, for example with the :

```
$ pip install virtualenv
```

Then create a virtual environment for Python 3:

```
$ virtualenv -p python3 folder_name
```

And activate the environment with:

```
$ source folder_name/bin/activate
```

A virtual environment can be deactivated with:

```
$ deactivate
```

Important: Make sure to adjust the path where the virtual environment is installed and activated.

1.2 Installation from PyPI

PynPoint can now be installed with pip:

```
$ pip install pynpoint
```

If you do not use a virtual environment then you may have to add the `--user` argument:

```
$ pip install --user pynpoint
```

PynPoint is actively being developed. To update the installation to the latest version:

```
$ pip install --upgrade PynPoint
```

1.3 Installation from Github

The repository can also be cloned from Github, which contains the most recent implementations:

```
$ git clone git@github.com:PynPoint/PynPoint.git
```

In that case, the dependencies can be installed from the PynPoint folder:

```
$ pip install -r requirements.txt
```

By adding the path of the repository to the `PYTHONPATH` environment variable enables PynPoint to be imported from any location:

```
$ echo "export PYTHONPATH='$PYTHONPATH:/path/to/pynpoint'" >> folder_name/bin/activate
```

Important: Make sure to adjust local path in which PynPoint will be cloned from the Github repository.

Do you want to makes changes to the code? Then please fork the PynPoint repository on the Github page and clone your own fork instead of the main repository. We very much welcome active contributions and pull requests (see *Contributing* section).

1.4 Testing Pynpoint

The installation can be tested by starting Python in interactive mode and printing the PynPoint version:

```
>>> import pynpoint
>>> pynpoint.__version__
```

Tip: If the PynPoint package is not find by Python then possibly the path was not set correctly. The list of folders that are searched by Python for modules can be printed in interactive mode as:

```
>>> import sys
>>> sys.path
```

The result should contain the folder in which the Github repository was cloned or the folder in which Python modules are installed with pip.

CHAPTER 2

Running PynPoint

2.1 Introduction

As a first example, we provide a preprocessed dataset of beta Pic in the M' filter (4.8 μm). This archival dataset was obtained with NACO at the Very Large Telescope under the ESO program ID . The exposure time of the individual images was 65 ms and the total field rotation about 50 degrees. To limit the size of the dataset, every 200 images have been mean-collapsed. The data is stored in an HDF5 database (see *HDF5 Files*) which contains 263 images of 80x80 pixels, the parallactic angles, and the pixel scale. The dataset is stored under the tag name `stack`.

2.2 First Example

The following script downloads the data (13 MB), runs the PSF subtraction with PynPoint, and plots an image of the median-collapsed residuals:

```
import os
import urllib
import matplotlib.pyplot as plt

from pynpoint import Pypeline, \
                     Hdf5ReadingModule, \
                     PSFpreparationModule, \
                     PcaPsfSubtractionModule

working_place = "/path/to/working_place/"
input_place = "/path/to/input_place/"
output_place = "/path/to/output_place/"

data_url = "https://people.phys.ethz.ch/~stolkert/pynpoint/betapic_naco_mp.hdf5"
data_loc = os.path.join(input_place, "betapic_naco_mp.hdf5")

urllib.request.urlretrieve(data_url, data_loc)
```

(continues on next page)

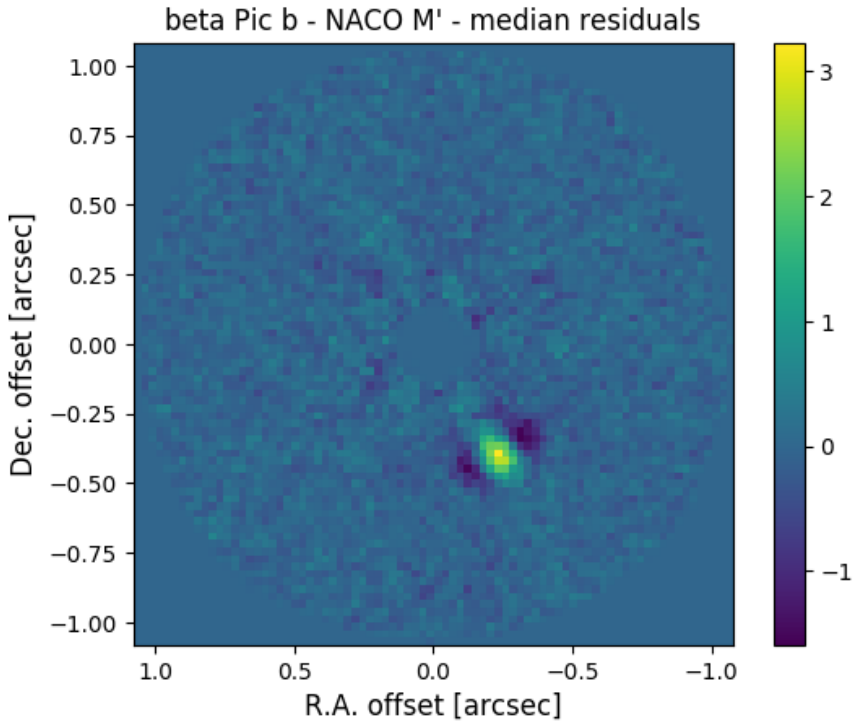
(continued from previous page)

```
pipeline = Pypeline(working_place_in=working_place,
                    input_place_in=input_place,
                    output_place_in=output_place)

module = Hdf5ReadingModule(name_in="read",
                           input_filename="betapic_naco_mp.hdf5",
                           input_dir=None,
                           tag_dictionary={"stack":"stack"})

pipeline.add_module(module)

module = PSFpreparationModule(name_in="prep",
                              image_in_tag="stack",
                              image_out_tag="prep",
                              mask_out_tag=None,
                              norm=False,
                              resize=None,
                              cent_size=0.15,
                              edge_size=1.1)

pipeline.add_module(module)

module = PcaPsfSubtractionModule(pca_numbers=(20, ),
                                 name_in="pca",
                                 images_in_tag="prep",
                                 reference_in_tag="prep",
                                 res_median_tag="residuals")

pipeline.add_module(module)

pipeline.run()

residuals = pipeline.get_data("residuals")
pixscale = pipeline.get_attribute("stack", "PIXSCALE")

size = pixscale*residuals.shape[-1]/2.

plt.imshow(residuals[0, ], origin='lower', extent=[size, -size, -size, size])
plt.title("beta Pic b - NACO M' - median residuals")
plt.xlabel('R.A. offset [arcsec]', fontsize=12)
plt.ylabel('Dec. offset [arcsec]', fontsize=12)
plt.colorbar()
plt.savefig(os.path.join(output_place, "residuals.png"), bbox_inches='tight')
```

Important: In the example, make sure to change the path of the `working place`, `input place`, and `output place`.

2.3 Detection of beta Pic b

That's it! The residuals of the PSF subtraction are stored in the database under the tag name `residuals` and the plotted image is located in the `output_place_in` folder. The image shows the detection of the exoplanet :



The star of this planetary system is located in the center of the image (which is masked here) and the orientation of the image is such that North is up and East is left. The bright yellow feature in the bottom right direction is the planet beta Pic b at an angular separation of 0.46 arcseconds.

CHAPTER 3

Overview

Here you find a list of all available pipeline modules with a very short description of what each module does. Reading modules import data into the database, writing modules export data from the database, and processing modules run a specific task of the data reduction and analysis. More details on the design of the pipeline can be found in the *Architecture* section.

Note: All PynPoint classes ending with `Module` in their name (e.g. *`FitsReadingModule`*) are pipeline modules that can be added to an instance of *`Pypeline`* (see *Pypeline* section).

3.1 Reading Modules

- *`FitsReadingModule`*: Import FITS files and relevant header information into the database.
- *`Hdf5ReadingModule`*: Import datasets and attributes from an HDF5 file (as created by PynPoint).
- *`ParangReadingModule`*: Import a list of parallactic angles as dataset attribute.
- *`AttributeReadingModule`*: Import a list of values as dataset attribute.
- *`NearReadingModule`* (CPU): Import VLT/VISIR data for the NEAR experiment.

3.2 Writing Modules

- *`FitsWritingModule`*: Export a dataset from the database to a FITS file.
- *`Hdf5WritingModule`*: Export part of the database to a new HDF5 file.
- *`TextWritingModule`*: Export a dataset to an ASCII file.
- *`ParangWritingModule`*: Export the parallactic angles of a dataset to an ASCII file.
- *`AttributeWritingModule`*: Export a list of attribute values to an ASCII file.

3.3 Processing Modules

3.3.1 Background Subtraction

- *SimpleBackgroundSubtractionModule*: Simple background subtraction for dithering datasets.
- *MeanBackgroundSubtractionModule*: Mean background subtraction for dithering datasets.
- *LineSubtractionModule* (CPU): Subtraction of striped detector artifacts.
- *NoddingBackgroundModule*: Background subtraction for nodding datasets.

3.3.2 Bad Pixel Cleaning

- *BadPixelSigmaFilterModule* (CPU): Find and replace bad pixels with a sigma filter.
- *BadPixelInterpolationModule* (CPU): Interpolate bad pixels with a spectral deconvolution technique.
- *BadPixelMapModule*: Create a bad pixel map from dark and flat images.
- *BadPixelTimeFilterModule* (CPU): Sigma clipping of bad pixels along the time dimension.
- *ReplaceBadPixelsModule* (CPU): Replace bad pixels based on a bad pixel map.

3.3.3 Basic Processing

- *SubtractImagesModule*: Subtract two stacks of images.
- *AddImagesModule*: Add two stacks of images
- *RotateImagesModule*: Rotate a stack of images.
- *RepeatImagesModule*: Repeat a stack of images.

3.3.4 Centering

- *StarAlignmentModule* (CPU): Align the images with a cross-correlation.
- *FitCenterModule* (CPU): Fit the PSF with a 2D Gaussian or Moffat function.
- *ShiftImagesModule*: Shift a stack of images.
- *WaffleCenteringModule*: Use the waffle spots to center the images.

3.3.5 Dark and Flat Correction

- *DarkCalibrationModule*: Dark frame subtraction.
- *FlatCalibrationModule*: Flat field correction.

3.3.6 Denoising

- *WaveletTimeDenoisingModule* (CPU): Wavelet-based denoising in the time domain.
- *TimeNormalizationModule* (CPU): Normalize a stack of images.

3.3.7 Detection Limits

- `ContrastCurveModule` (CPU): Compute a contrast curve.
- `MassLimitsModule`: Calculate mass limits from a contrast curve and an isochrones model grid.

3.3.8 Extract Star

- `StarExtractionModule` (CPU): Locate and crop the position of the star.
- `ExtractBinaryModule` (CPU): Extract a PSF which rotates across a stack of images.

3.3.9 Flux and Position

- `FakePlanetModule`: Inject an artificial planet in a dataset.
- `SimplexMinimizationModule`: Determine the flux and position with a simplex minimization.
- `FalsePositiveModule`: Compute the signal-to-noise ratio and false positive fraction.
- `MCMCsamplingModule` (CPU): Estimate the flux and position of a planet with MCMC sampling.
- `AperturePhotometryModule` (CPU): Compute the integrated flux at a position.

3.3.10 Frame Selection

- `RemoveFramesModule`: Remove images by their index number.
- `FrameSelectionModule`: Frame selection to remove low-quality image.
- `RemoveLastFrameModule`: Remove the last image of a VLT/NACO dataset.
- `RemoveStartFramesModule`: Remove images at the beginning of each original data cube.
- `ImageStatisticsModule` (CPU): Compute statistics of the pixel values for each image.
- `FrameSimilarityModule` (CPU): Compute different similarity measures of a set of images.
- `SelectByAttributeModule`: Select images by the ascending/descending attribute values.

3.3.11 Image Resizing

- `CropImagesModule`: Crop the images.
- `ScaleImagesModule` (CPU): Resample the images (spatially and/or in flux).
- `AddLinesModule`: Add pixel lines on the sides of the images.
- `RemoveLinesModule`: Remove pixel lines from the sides of the images.

3.3.12 PCA Background Subtraction

- `PCABackgroundPreparationModule`: Preparation for the PCA-based background subtraction.
- `PCABackgroundSubtractionModule`: PCA-based background subtraction.
- `DitheringBackgroundModule`: Wrapper for background subtraction of dithering datasets.

3.3.13 PSF Preparation

- *PSFpreparationModule*: Mask the images before the PSF subtraction.
- *AngleInterpolationModule*: Interpolate the parallactic angles between the start and end values.
- *AngleCalculationModule*: Calculate the parallactic angles.
- *SortParangModule*: Sort the images by parallactic angle.
- *SDIpreparationModule*: Prepare the images for SDI.

3.3.14 PSF Subtraction

- *PcaPsfSubtractionModule* (CPU): PSF subtraction with PCA.
- *ClassicalADIModule* (CPU): PSF subtraction with classical ADI.

3.3.15 Stacking

- *StackAndSubsetModule*: Stack and/or select a random subset of the images.
- *StackCubesModule*: Collapse each original data cube separately.
- *DerotateAndStackModule*: Derotate and/or stack the images.
- *CombineTagsModule*: Combine multiple database tags into a single dataset.

Note: The pipeline modules with multiprocessing functionalities are indicated with "CPU" in parentheses. The number of parallel processes can be set with the `CPU` parameter in the central configuration file and the number of images that is simultaneously loaded into the memory with the `MEMORY` parameter. Pipeline modules that apply (in parallel) a function to subsets of images use a number of images per subset equal to `MEMORY` divided by `CPU`.

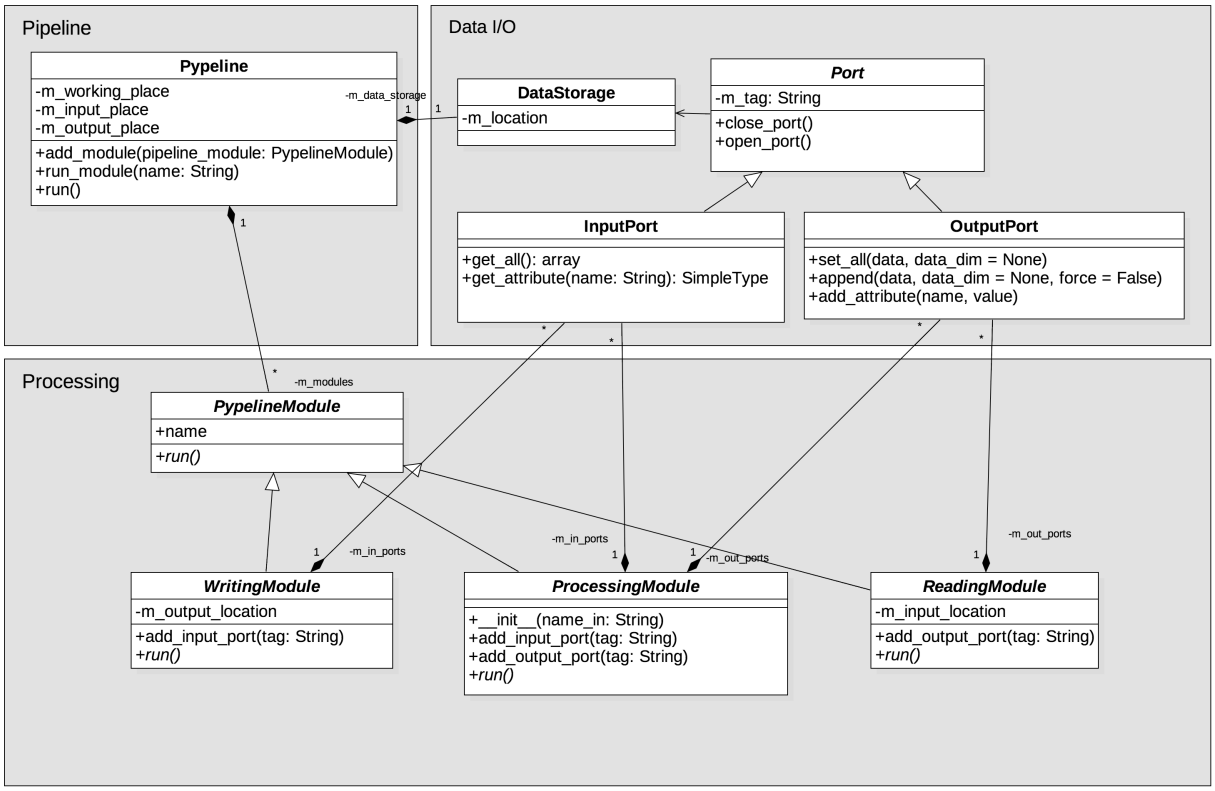
CHAPTER 4

Architecture

4.1 Introduction

PynPoint has evolved from a PSF subtraction toolkit to an end-to-end pipeline for processing and analysis of high-contrast imaging data. The architecture of PynPoint was redesigned in v0.3.0 with the goal to create a generic, modular, and open-source data reduction pipeline, which is extendable to new data processing techniques and data types in the future.

The actual pipeline and the processing modules have been separated in a different subpackages. Therefore, it is possible to extend the processing functionalities without intervening with the core of the pipeline. The UML class diagram below illustrates the pipeline architecture of PynPoint:



The diagram shows that the architecture is subdivided in three components:

- Data management - *pynpoint.core.dataio*
- Pipeline modules for reading, writing, and processing of data - *pynpoint.core.processing*
- The actual pipeline - *pynpoint.core.pypeline*

4.2 Central Database

In the new architecture, the data management has been separated from the data processing for the following reasons:

1. Raw datasets can be very large, in particular in the 3–5 μm wavelength regime, which challenges the processing on a computer with a small amount of memory (RAM). A central database is used to store the data on a computer's hard drive.
2. Some data is used in different steps of the pipeline. A central database makes it easy to access that data without making a copy.
3. The central data storage on the hard drive will remain updated after each step. Therefore, processing steps that already finished remain unaffected if an error occurs or the data reduction is interrupted by the user.

Understanding the central data storage classes can be helpful if you plan to write your own Pipeline modules (see *Coding a New Module*). When running the pipeline, it is enough to understand the concept of database tags.

Each pipeline module has input and/or output tags which point to specific dataset in the central database. A module with `image_in_tag=im_arr` will look for a stack of input images in the central database under the tag name *im_arr*. Similarly, a module with `image_out_tag=im_arr_processed` will a stack of processed images to the central database under the tag *im_arr_processed*. Note that input tags will never change the data in the database.

Accessing the data storage occurs through instances of *Port* which allow pipeline modules to read data from and write data to central database.

4.3 Pipeline Modules

A pipeline module has a specific task that is appended to the internal queue of a *Pypeline* instance. Pipeline modules can read and write data tags from and to the central database through dedicated input and output connections. There are three types of pipeline modules:

1. *pynpoint.core.processing.ReadingModule* - A module with only output tags/ports, used to read data to the central database.
2. *pynpoint.core.processing.WritingModule* - A module with only input tags/ports, used to export data from the central database.
3. *pynpoint.core.processing.ProcessingModule* - A module with both input and output tags/ports, used for processing of the data.

Typically, a *ProcessingModule* reads one or multiple datasets from the database, applies a specific processing task with user-defined parameter values, and stores the results as a new dataset in the database.

In order to create a valid data reduction cascade, one should check that the required input tags are linked to data which were previously created by another pipeline module. In other words, there needs to be a previous module which has stored output under that same tag name.

4.4 Pypeline

The *pypeline* module is the central component which manages the order and execution of the different pipeline modules. Each *Pypeline* instance has an `working_place_in` path which is where the central database and configuration file are stored, an `input_place_in` path which is the default data location for reading modules, and an `output_place_in` path which is the default output path where the data will be saved by the writing modules:

```
from pynpoint import Pypeline, FitsReadingModule

pipeline = Pypeline(working_place_in="/path/to/working_place",
                    input_place_in="/path/to/input_place",
                    output_place_in="/path/to/output_place")
```

A pipeline module is created from any of the classes listed in the *Overview* section, for example:

```
module = FitsReadingModule(name_in="read", image_tag="input")
```

The module is appended to the pipeline queue as:

```
pipeline.add_module(module)
```

And can be removed from the queue with the following method:

```
pipeline.remove_module("read")
```

The names and order of the pipeline modules can be listed with:

```
pipeline.get_module_names()
```

Running all modules attached to the pipeline is achieved with:

```
pipeline.run()
```

Or a single module is executed as:

```
pipeline.run_module("read")
```

Both run methods will check if the pipeline has valid input and output tags.

An instance of `Pypeline` can be used to directly access data from the central database. See the *HDF5 Files* section for more information.

CHAPTER 5

Configuration

5.1 Introduction

A configuration file has to be stored in the `working_place_in` with the name `PynPoint_config.ini`. The file will be created with default values in case it does not exist when the pipeline is initiated. The values of the configuration file are stored in a separate group of the central database, each time the pipeline is initiated.

5.2 Config File

The file contains two different sections of configuration parameters. The `header` section is used to link attributes in PynPoint with header values in the FITS files that will be imported into the database. For example, some of the pipeline modules require values for the dithering position. These attributes are stored as `DITHER_X` and `DITHER_Y` in the central database and are for example provided by the `ESO SEQ CUMOFFSETX` and `ESO SEQ CUMOFFSETY` values in the FITS header. Setting `DITHER_X: ESO SEQ CUMOFFSETX` in the `header` section of the configuration file makes sure that the relevant FITS header values are imported when *`FitsReadingModule`* is executed. Therefore, FITS files have to be imported again if values in the `header` section are changed. Values can be set to `None` since `header` values are only required for some of the pipeline modules.

The second section of the configuration values contains the central settings that are used by the pipeline modules. These values are stored in the `settings` section of the configuration file. The pixel scale can be provided in arcsec per pixel (e.g. `PIXSCALE: 0.027`), the number of images that will be simultaneously loaded into the memory (e.g. `MEMORY: 1000`), and the number of cores that are used for pipeline modules that have multiprocessing capabilities (e.g. `CPU: 8`) such as *`PcaPsfSubtractionModule`* and *`MCMCsamplingModule`*. A complete overview of the pipeline modules that support multiprocessing is available in the *Overview* section.

Note that some of the pipeline modules provide also multithreading support, which by default runs on all available CPUs. The multithreading can be controlled from the command line by setting the `OMP_NUM_THREADS` environment variable:

```
$ export OMP_NUM_THREADS=8
```

In this case a maximum of 8 threads is used. So, if a modules provide both multiprocessing and multithreading support, then the total number of used cores is equal to the product of the values chosen for `CPU` in the configuration file and `OMP_NUM_THREADS` from the command line.

5.3 Examples

In this section, several examples are provided of the configuration content for the following instruments:

- *VLT/NACO*
- *VLT/SPHERE/IRDIS*
- *VLT/VISIR*

5.3.1 VLT/NACO

```
[header]

INSTRUMENT: INSTRUME
NFRAMES: NAXIS3
EXP_NO: ESO DET EXP NO
DIT: ESO DET DIT
NDIT: ESO DET NDIT
PARANG_START: ESO ADA POSANG
PARANG_END: ESO ADA POSANG END
DITHER_X: ESO SEQ CUMOFFSETX
DITHER_Y: ESO SEQ CUMOFFSETY
PUPIL: ESO ADA PUPILPOS
DATE: DATE-OBS
LATITUDE: ESO TEL GEOLAT
LONGITUDE: ESO TEL GEOLON
RA: RA
DEC: DEC

[settings]

PIXSCALE: 0.027
MEMORY: 1000
CPU: 1
```

5.3.2 VLT/SPHERE/IRDIS

```
[header]

INSTRUMENT: INSTRUME
NFRAMES: NAXIS3
EXP_NO: ESO DET EXP ID
DIT: EXPTIME
NDIT: ESO DET NDIT
PARANG_START: ESO TEL PARANG START
PARANG_END: ESO TEL PARANG END
DITHER_X: ESO INS1 DITH POSX
DITHER_Y: ESO INS1 DITH POSY
```

(continues on next page)

(continued from previous page)

```
PUPIL: None
DATE: DATE-OBS
LATITUDE: ESO TEL GEOLAT
LONGITUDE: ESO TEL GEOLON
RA: ESO INS4 DROT2 RA
DEC: ESO INS4 DROT2 DEC

[settings]

PIXSCALE: 0.01226
MEMORY: 1000
CPU: 1
```

5.3.3 VLT/VISIR

```
[header]

INSTRUMENT: INSTRUME
NFRAMES: NAXIS3
EXP_NO: ESO TPL EXPNO
DIT: ESO DET SEQ1 DIT
NDIT: ESO DET CHOP NCYCLES
PARANG_START: ESO ADA POSANG
PARANG_END: ESO ADA POSANG END
DITHER_X: None
DITHER_Y: None
PUPIL: ESO ADA PUPILPOS
DATE: DATE-OBS
LATITUDE: ESO TEL GEOLAT
LONGITUDE: ESO TEL GEOLON
RA: RA
DEC: DEC

[settings]

PIXSCALE: 0.045
MEMORY: 1000
CPU: 1
```

CHAPTER 6

Tutorial

6.1 Introduction

The pipeline can be executed with a Python script, in interactive mode of Python, or with a Jupyter Notebook. The pipeline works with two different components:

1. Pipeline modules which read, write, and process data:

    1.1 *pynpoint.core.processing.ReadingModule* - Reading of the data and relevant header information.

    1.2 *pynpoint.core.processing.WritingModule* - Exporting of results from the database.

    1.3 *pynpoint.core.processing.ProcessingModule* - Processing and analysis of the data.

2. The actual pipeline *pynpoint.core.pypeline.Pypeline* which capsules a list of pipeline modules.

Important: Pixel coordinates are zero-indexed, meaning that (x, y) = (0, 0) corresponds to the center of the pixel in the bottom-left corner of the image. The coordinate of the bottom-left corner is therefore (x, y) = (-0.5, -0.5). This means that there is an offset of -1.0 in both directions with respect to the pixel coordinates of DS9, for which the bottom-left corner is (x, y) = (0.5, 0.5).

6.2 Data Types

PynPoint currently works with three types of input and output data:

- FITS files
- HDF5 files
- ASCII files

PynPoint creates an HDF5 database called `PynPoin_database.hdf5` in the `working_place_in` of the pipeline. This is the central data storage in which the results of the processing steps are saved. The advantage of the HDF5 data format is that reading of data is much faster compared to the FITS data format and it is possible to quickly read subsets from very large datasets.

Input data is read into the central database with a *ReadingModule*. By default, PynPoint will read data from the `input_place_in` but setting a manual folder is possible to read data to separate database tags (e.g., dark frames, flat fields, and science data). Here we show an example of how to read FITS files and a list of parallactic angles.

First, we need to create an instance of *Pypeline*:

```
pipeline = Pypeline(working_place_in="/path/to/working_place",
                    input_place_in="/path/to/input_place",
                    output_place_in="/path/to/output_place")
```

Next, we read the science data from the the default input location:

```
module = FitsReadingModule(name_in="read_science",
                           input_dir=None,
                           image_tag="science")

pipeline.add_module(module)
```

And we read the flat fields from a separate location:

```
module = FitsReadingModule(name_in="read_flat",
                           input_dir="/path/to/flat",
                           image_tag="flat")

pipeline.add_module(module)
```

The parallactic angles are read from a text file in the default input folder and attached as attribute to the science data:

```
module = ParangReadingModule(file_name="parang.dat",
                             name_in="parang",
                             input_dir=None,
                             data_tag="science")

pipeline.add_module(module)
```

Finally, we run all pipeline modules:

```
pipeline.run()
```

Alternatively, it is also possible to run the modules individually by their `name_in` value:

```
pipeline.run_module("read_science")
pipeline.run_module("read_flat")
pipeline.run_module("parang")
```

The FITS files of the science data and flat fields are read and stored into the central HDF5 database. The data is labelled with a tag which is used by other pipeline module to access data from the database.

Restoring data from an already existing pipeline database can be done by creating an instance of *Pypeline* with the `working_place_in` pointing to the path of the `PynPoint_database.hdf5` file.

PynPoint can also handle the HDF5 format as input and output data. Data and corresponding attributes can be exported as HDF5 file with *Hdf5WritingModule*. This data format can be imported into the central database with *Hdf5ReadingModule*. Have a look at the *pynpoint package* section for more information.

6.3 HDF5 Files

There are several options to access data from the central HDF5 database:

- Use *FitsWritingModule* to export data to a FITS file, as shown in the *Examples* section.
- Use the easy access functions of the *pynpoint.core.pypeline* module to retrieve data and attributes from the database:
  - `pipeline.get_data(tag='tag_name')`
  - `pipeline.get_attribute(data_tag='tag_name', attr_name='attr_name')`
- Use an external tool such as or to read, inspect, and visualize data and attributes in the HDF5 database. We recommend using HDFCompass because it is easy to use and has a basic plotting functionality, allowing the user to quickly inspect images from a particular database tag. In HDFCompass, the static attributes can be opened with the *Reopen as HDF5 Attributes* option.

CHAPTER 7

Examples

7.1 VLT/SPHERE H-alpha data

An end-to-end example of a SPHERE/ZIMPOL H-alpha data set of the accreting M dwarf companion of HD 142527 (see Cugno et al. 2019) can be downloaded here.

7.2 VLT/NACO M' dithering data

Here we show an end-to-end processing example of a pupil-stabilized data set of beta Pic from Stolker et al. (2019) (see also *Running PynPoint*). This archival data set was obtained with VLT/NACO in the M' band. A dithering pattern was applied to sample the sky background.

First we need to import the Pypeline, as well as the I/O and processing modules. These can be directly imported from the package, for example:

```
from pynpoint import Pypeline, FitsReadingModule
```

Next, we create an instance of `Pypeline` with the `working_place_in` pointing to a path where PynPoint has enough space to create its database, `input_place_in` pointing to the path with the raw FITS files, and `output_place_in` to a folder for the output:

```
pipeline = Pypeline(working_place_in='/path/to/working_place',
                    input_place_in='/path/to/input_place',
                    output_place_in'/path/to/output_place')
```

The FWHM of the PSF is defined for simplicity:

```
fwhm = 0.134 # [arcsec]
```

Now we are ready to add all the pipeline modules that we need. Have a look at the documentation in the *pynpoint package* section for a detailed description of the individual modules and their parameters.

1. Import the raw science, flat, and dark images into the database:

```
module = FitsReadingModule(name_in='read1',
                           input_dir='/path/to/science/',
                           image_tag='science',
                           overwrite=True,
                           check=True)

pipeline.add_module(module)

module = FitsReadingModule(name_in='read2',
                           input_dir='/path/to/flat/',
                           image_tag='flat',
                           overwrite=True,
                           check=False)

pipeline.add_module(module)

module = FitsReadingModule(name_in='read4',
                           input_dir='/path/to/dark/',
                           image_tag='dark',
                           overwrite=True,
                           check=False)

pipeline.add_module(module)
```

2. Import the image with the (separately processed) unsaturated PSF of the star:

```
module = Hdf5ReadingModule(name_in='read4',
                           input_filename='flux.hdf5',
                           input_dir='/path/to/flux/',
                           tag_dictionary={'flux': 'flux'})

pipeline.add_module(module)
```

3. Remove NDIT+1 frames which contain the average of the FITS cube (NACO specific):

```
module = RemoveLastFrameModule(name_in='last',
                               image_in_tag='science',
                               image_out_tag='last')

pipeline.add_module(module)
```

4. Calculate the parallactic angles which each image:

```
module = AngleCalculationModule(name_in='angle',
                                data_tag='last',
                                instrument='NACO')

pipeline.add_module(module)
```

5. Remove the top and bottom line to make the images square:

```
module = RemoveLinesModule(lines=(0, 0, 1, 1),
                           name_in='cut',
                           image_in_tag='last',
                           image_out_tag='cut')

pipeline.add_module(module)
```

6. Subtract the dark current from the flat field:

```
module = DarkCalibrationModule(name_in='dark',
                               image_in_tag='flat',
                               dark_in_tag='dark',
                               image_out_tag='flat_cal')

pipeline.add_module(module)
```

7. Divide the science data by the master flat:

```
module = FlatCalibrationModule(name_in='flat',
                               image_in_tag='science',
                               flat_in_tag='flat_cal',
                               image_out_tag='science_cal')

pipeline.add_module(module)
```

8. Remove the first 5 frames from each FITS cube because of the systematically higher background emission:

```
module = RemoveStartFramesModule(frames=5,
                                 name_in='first',
                                 image_in_tag='science_cal',
                                 image_out_tag='first')

pipeline.add_module(module)
```

9. PCA based background subtraction:

```
module = DitheringBackgroundModule(name_in='background',
                                   image_in_tag='first',
                                   image_out_tag='background',
                                   center=((263, 263), (116, 263), (116, 116),␣
↪(263, 116)),
                                   cubes=None,
                                   size=3.5,
                                   gaussian=fwhm,
                                   subframe=10.*fwhm,
                                   pca_number=60,
                                   mask_star=4.*fwhm,
                                   mask_planet=None,
                                   subtract_mean=True,
                                   bad_pixel=(9, 5., 3),
                                   crop=True,
                                   prepare=True,
                                   pca_background=True,
                                   combine='pca')

pipeline.add_module(module)
```

10. Bad pixel correction:

```
module = BadPixelSigmaFilterModule(name_in='bad',
                                   image_in_tag='background',
                                   image_out_tag='bad',
                                   map_out_tag='bpmap',
                                   box=9,
                                   sigma=5.,
```

(continues on next page)

(continued from previous page)

```
                                    iterate=3)

pipeline.add_module(module)
```

11. Frame selection:

```
module = FrameSelectionModule(name_in='select',
                              image_in_tag='bad',
                              selected_out_tag='selected',
                              removed_out_tag='removed',
                              index_out_tag=None,
                              method='median',
                              threshold=2.,
                              fwhm=fwhm,
                              aperture=('circular', fwhm),
                              position=(None, None, 4.*fwhm))

pipeline.add_module(module)
```

12. Extract the star position and center with pixel precision:

```
module = StarExtractionModule(name_in='extract',
                              image_in_tag='selected',
                              image_out_tag='extract',
                              index_out_tag='index',
                              image_size=3.,
                              fwhm_star=fwhm,
                              position=(None, None, 4.*fwhm))

pipeline.add_module(module)
```

13. Align the images with a cross-correlation of the central 800 mas:

```
module = StarAlignmentModule(name_in='align',
                             image_in_tag='odd',
                             ref_image_in_tag=None,
                             image_out_tag='align',
                             interpolation='spline',
                             accuracy=10,
                             resize=None,
                             num_references=10,
                             subframe=0.8)

pipeline.add_module(module)
```

14. Center the images with subpixel precision by applying a constant shift:

```
module = FitCenterModule(name_in='center',
                         image_in_tag='align',
                         fit_out_tag='fit',
                         mask_out_tag=None,
                         method='mean',
                         radius=5.*fwhm,
                         sign='positive',
                         model='gaussian',
                         filter_size=None,
```

(continues on next page)

(continued from previous page)

```
                          guess=(0., 0., 1., 1., 100., 0., 0.))

pipeline.add_module(module)

module = ShiftImagesModule(name_in='shift',
                           image_in_tag='align',
                           image_out_tag='center',
                           shift_xy='fit',
                           interpolation='spline')

pipeline.add_module(module)
```

15. Stack by 100 images:

```
module = StackAndSubsetModule(name_in='stack',
                              image_in_tag='center',
                              image_out_tag='stack',
                              random=None,
                              stacking=100)

pipeline.add_module(stack)
```

16. Prepare the data for PSF subtraction:

```
module = PSFpreparationModule(name_in='prep',
                              image_in_tag='stack',
                              image_out_tag='prep',
                              mask_out_tag=None,
                              norm=False,
                              resize=None,
                              cent_size=fwhm,
                              edge_size=1.)

pipeline.add_module(module)
```

17. PSF subtraction with PCA:

```
module = PcaPsfSubtractionModule(pca_numbers=range(1, 51),
                                 name_in='pca',
                                 images_in_tag='prep',
                                 reference_in_tag='prep',
                                 res_mean_tag='pca_mean',
                                 res_median_tag='pca_median',
                                 res_weighted_tag=None,
                                 res_arr_out_tag=None,
                                 res_rot_mean_clip_tag=None,
                                 extra_rot=0.)

pipeline.add_module(module)
```

18. Measure the signal-to-noise ratio and false positive fraction:

```
module = FalsePositiveModule(position=(50.5, 26.5),
                             aperture=fwhm/2.,
                             ignore=True,
                             name_in='fpf',
```

(continues on next page)

(continued from previous page)

```
                              image_in_tag='pca_median',
                              snr_out_tag='fpf')

pipeline.add_module(module)
```

19. Write the median residuals to a FITS file:

```
module = FitsWritingModule(name_in='write',
                           file_name='residuals.fits',
                           output_dir=None,
                           data_tag='pca_median',
                           data_range=None)

pipeline.add_module(module)
```

20. And finally, run the pipeline:

```
pipeline.run()
```

21. Or, to run a specific pipeline module individually:

```
pipeline.run_module('pca')
```

CHAPTER 8

API Documentation

8.1 pynpoint package

8.1.1 Subpackages

pynpoint.core package

Submodules

pynpoint.core.attributes module

Module to obtain information about the implemented attributes.

pynpoint.core.attributes.**get_attributes**()
Function to get a dictionary with all attributes.

Returns Attribute information.

Return type dict

pynpoint.core.dataio module

Modules for accessing data and attributes in the central database.

class pynpoint.core.dataio.**ConfigPort**(*tag*, *data_storage_in=None*)
Bases: *pynpoint.core.dataio.Port*

ConfigPort can be used to read the 'config' tag from a (HDF5) database. This tag contains the central settings used by PynPoint, as well as the relevant FITS header keywords. You can use a ConfigPort instance to access a single attribute of the dataset using get_attribute().

Constructor of the ConfigPort class which creates the config port instance which can read the settings stored in the central database under the tag *config*. An instance of the ConfigPort is created in the constructor of

PypelineModule such that the attributes in the ConfigPort can be accessed from within all type of modules. For example:

```
memory = self._m_config_port.get_attribute('MEMORY')
```

Parameters

- **tag** (*str*) – The tag name of the port. The port can be used to get data from the dataset with the key *config*.
- **data_storage_in** (pynpoint.core.dataio.DataStorage) – The input DataStorage. It is possible to give the constructor of an ConfigPort a DataStorage instance which will link the port to that DataStorage. Usually the DataStorage is set later by calling *set_database_connection()*.

Returns None

Return type NoneType

get_attribute(*name*)

Returns a static attribute which is connected to the dataset of the ConfigPort.

Parameters **name** (*str*) – The name of the attribute.

Returns The attribute value. Returns None if the attribute does not exist.

Return type str, float, or int

class pynpoint.core.dataio.**DataStorage**(*location_in*)

Bases: object

Instances of DataStorage manage to open and close the Pypeline HDF5 databases. They have an internal h5py data bank (self.m_data_bank) which gives direct access to the data if the storage is open (self.m_open == True).

Constructor of a DataStorage instance. It needs the location of the HDF5 file (Pypeline database) as input. If the file already exists it is opened and extended, if not a new File will be created.

Parameters **location_in** (*str*) – Location (directory + filename) of the HDF5 database.

Returns None

Return type NoneType

close_connection()

Closes the connection to the HDF5 file. All entries of the data bank will be stored on the hard drive and the memory is cleaned.

Returns None

Return type NoneType

open_connection()

Opens the connection to the HDF5 file by opening an old file or creating a new one.

Returns None

Return type NoneType

class pynpoint.core.dataio.**InputPort**(*tag*, *data_storage_in=None*)

Bases: *pynpoint.core.dataio.Port*

InputPorts can be used to read datasets with a specific tag from the HDF5 database. This type of port can be used to access:

- A complete dataset using the get_all() method.

- A single attribute of the dataset using get_attribute().
- All attributes of the dataset using get_all_static_attributes() and get_all_non_static_attributes().
- A part of a dataset using slicing. For example:

```
in_port = InputPort('tag')
data = in_port[0, :, :] # returns the first 2D image of a 3D image stack.
```

(More information about how 1D, 2D, and 3D data is organized can be found in the documentation of OutputPort (*append()* and *set_all()*)

InputPorts can load two types of attributes which give additional information about a dataset the port is linked to:

- Static attributes: contain global information about a dataset which is not changing through a dataset in the database (e.g. the instrument name or pixel scale).
- Non-static attributes: contain information which changes for different parts of the dataset (e.g. the parallactic angles or dithering positions).

Constructor of InputPort. An input port can read data from the central database under the key *tag*. Instances of InputPort should not be created manually inside a PypelineModule but should be created with the add_input_port() function.

Parameters

- **tag** (*str*) – The tag of the port. The port can be used in order to get data from the dataset with the key *tag*.
- **data_storage_in** (pynpoint.core.dataio.DataStorage) – It is possible to give the constructor of an InputPort a DataStorage instance which will link the port to that DataStorage. Usually the DataStorage is set later by calling *set_database_connection()*.

Returns None

Return type NoneType

get_all()

Returns the whole dataset stored in the data bank under the tag of the Port. Be careful using this function for loading large datasets. The data type is inferred from the data with numpy.asarray. A 32 bit array will be returned in case the input data is a combination of float32 and float64 arrays.

Returns The full dataset. Returns None if the data does not exist.

Return type numpy.ndarray

get_all_non_static_attributes()

Returns a list of all non-static attribute keys. More information about static and non-static attributes can be found in the class documentation of *InputPort*.

Returns List of all existing non-static attribute keys.

Return type list(str,)

get_all_static_attributes()

Get all static attributes of the dataset which are linked to the Port tag.

Returns Dictionary of all attributes, as *{attr_name:attr_value}*.

Return type dict

get_attribute(*name*)
Returns an attribute which is connected to the dataset of the port. The function can return static and non-static attributes (static attributes have priority). More information about static and non-static attributes can be found in the class documentation of *InputPort*.

Parameters **name** (*str*) – The name of the attribute.

Returns The attribute value. Returns None if the attribute does not exist.

Return type str, float, int, or numpy.ndarray

get_ndim()
Returns the number of dimensions of the dataset the port is linked to.

Returns Number of dimensions of the dataset. Returns None if the dataset does not exist.

Return type int

get_shape()
Returns the shape of the dataset the port is linked to. This can be useful if you need the shape without loading the whole data.

Returns Shape of the dataset. Returns None if the dataset does not exist.

Return type tuple(int,)

class pynpoint.core.dataio.**OutputPort**(*tag*, *data_storage_in=None*, *activate_init=True*)
Bases: *pynpoint.core.dataio.Port*

Output ports can be used to save results under a given tag to the HDF5 DataStorage. An instance of OutputPort with self.tag='tag' can store data under the key *tag* by using one of the following methods:

- set_all(...) - replaces and sets the whole dataset
- append(...) - appends data to the existing data set. For more information see function documentation (*append()*).
- slicing - sets a part of the actual dataset. Example:

```
out_port = OutputPort('Some_tag')
data = np.ones(200, 200) # 2D image filled with ones
out_port[0,:,:] = data # Sets the first 2D image of a 3D image stack
```

- add_attribute(...) - modifies or creates a attribute of the dataset
- del_attribute(...) - deletes a attribute
- del_all_attributes(...) - deletes all attributes
- append_attribute_data(...) - appends information to non-static attributes. See add_attribute() (*add_attribute()*) for more information about static and non-static attributes.
- check_static_attribute(...) - checks if a static attribute exists and if it is equal to a given value
- other functions listed below

For more information about how data is organized inside the central database have a look at the function documentation of the function *set_all()* and *append()*.

Furthermore it is possible to deactivate a OutputPort to stop him saving data.

Constructor of the OutputPort class which creates an output port instance which can write data to the the central database under the tag *tag*. If you write a PypelineModule you should not create instances manually! Use the add_output_port() function instead.

Parameters

- **tag** (*str*) – The tag of the port. The port can be used in order to write data to the dataset with the key = *tag*.
- **data_storage_in** (pynpoint.core.dataio.DataStorage) – It is possible to give the constructor of an OutputPort a DataStorage instance which will link the port to that DataStorage. Usually the DataStorage is set later by calling *set_database_connection()*.

Returns None

Return type NoneType

activate()

Activates the port. A non activated port will not save data.

Returns None

Return type NoneType

add_attribute(*name*, *value*, *static=True*)

Adds an attribute to the dataset of the Port with the attribute name = *name* and the value = *value*. If the attribute already exists it will be overwritten. Two different types of attributes are supported:

1. **static attributes**: Contain a single value or name (e.g. The name of the used Instrument).
2. **non-static attributes**: Contain a dataset which is connected to the actual data set (e.g. Instrument temperature). It is possible to append additional information to non-static attributes later (*append_attribute_data()*). This is not supported by static attributes.

Static and non-static attributes are stored in a different way using the HDF5 file format. Static attributes will be direct attributes while non-static attributes are stored in a group with the name *header_* + name of the dataset.

Parameters

- **name** (*str*) – Name of the attribute.
- **value** (*str, float, or int*) – Value of the attribute.
- **static** (*bool*) – Indicate if the attribute is static (True) or non-static (False).

Returns None

Return type NoneType

add_history(*module*, *history*)

Adds an attribute with history information about the pipeline module.

Parameters

- **module** (*str*) – Name of the pipeline module which was executed.
- **history** (*str*) – History information.

Returns None

Return type NoneType

append(*data*, *data_dim=None*, *force=False*)

Appends data to an existing dataset with the tag of the Port along the first dimension. If no data exists with the tag of the Port a new data set is created. For more information about how the dimensions are organized see documentation of the function *set_all()*. Note it is not possible to append data with a different shape or data type to the existing dataset.

Example: An internal data set is 3D (storing a stack of 2D images) with shape (233, 300, 300) which mean it contains 233 images with a resolution of 300 x 300 pixel. Thus it is only possible to extend along the first dimension by appending new images with a size of (300, 300) or by appending a stack of images (:, 300, 300). Everything else will raise exceptions.

It is possible to force the function to overwrite the existing data set if and only if the shape or type of the input data does not match the existing data. **Warning**: This can delete the existing data.

Parameters

- **data** (*numpy.ndarray*) – The data which will be appended.
- **data_dim** (*int*) – Number of data dimensions used if a new data set is created. The dimension of the *data* is used if set to None.
- **force** (*bool*) – The existing data will be overwritten if shape or type does not match if set to True.

Returns None

Return type NoneType

append_attribute_data(*name*, *value*)
Function which appends a single data value to non-static attributes.

Parameters

- **name** (*str*) – Name of the attribute.
- **value** (*str, float, or int*) – Value which will be appended to the attribute dataset.

Returns None

Return type NoneType

check_non_static_attribute(*name*, *comparison_value*)
Checks if a non-static attribute exists and if it is equal to a comparison value.

Parameters

- **name** (*str*) – Name of the non-static attribute.
- **comparison_value** (*numpy.ndarray*) – Comparison values

Returns Status: 1 if the non-static attribute does not exist, 0 if the non-static attribute exists and is equal, and -1 if the non-static attribute exists but is not equal.

Return type int

check_static_attribute(*name*, *comparison_value*)
Checks if a static attribute exists and if it is equal to a comparison value.

Parameters

- **name** (*str*) – Name of the static attribute.
- **comparison_value** (*str, float, or int*) – Comparison value.

Returns Status: 1 if the static attribute does not exist, 0 if the static attribute exists and is equal, and -1 if the static attribute exists but is not equal.

Return type int

copy_attributes(*input_port*)
Copies all static and non-static attributes from a given InputPort. Attributes which already exist will be

overwritten. Non-static attributes will be linked not copied. If the InputPort tag = OutputPort tag (self.tag) nothing will be changed. Use this function in all modules to keep the header information.

Parameters **input_port** (pynpoint.core.dataio.InputPort) – The InputPort with the header information.

Returns None

Return type NoneType

deactivate()
Deactivates the port. A non activated port will not save data.

Returns None

Return type NoneType

del_all_attributes()
Deletes all static and non-static attributes of the dataset.

Returns None

Return type NoneType

del_all_data()
Delete all data belonging to the database tag.

del_attribute(*name*)
Deletes the attribute of the dataset with the given name. Finds and removes static and non-static attributes.

Parameters **name** (*str*) – Name of the attribute.

Returns None

Return type NoneType

flush()
Forces the *DataStorage* to save all data from the memory to the hard drive without closing the *OutputPort*.

Returns None

Return type NoneType

set_all(*data*, *data_dim=None*, *keep_attributes=False*)
Set the data in the database by replacing all old values with the values of the input data. If no old values exists the data is just stored. Since it is not possible to change the number of dimensions of a data set later in the processing history one can choose a dimension different to the input data. The following cases are implemented:

- (#dimension of the first input data#, #desired data_dim#)
- (1, 1) 1D input or single value will be stored as list in HDF5
- (1, 2) 1D input, but 2D array stored inside (i.e. a list of lists with a fixed size).
- (2, 2) 2D input (single image) and 2D array stored inside (i.e. a list of lists with a fixed size).
- (2, 3) 2D input (single image) but 3D array stored inside (i.e. a stack of images with a fixed size).
- (3, 3) 3D input and 3D array stored inside (i.e. a stack of images with a fixed size).

For 2D and 3D data the first dimension always represents the list / stack (variable size) while the second (or third) dimension has a fixed size. After creation it is possible to extend a data set using *append()* along the first dimension.

Example 1:

Input 2D array with size (200, 200). Desired dimension 3D. The result is a 3D dataset with the dimension (1, 200, 200). It is possible to append other images with the size (200, 200) or other stacks of images with the size (:, 200, 200).

Example 2:

Input 2D array with size (200, 200). Desired dimension 2D. The result is a 2D dataset with the dimension (200, 200). It is possible to append other list with the length 200 or other stacks of lines with the size (:, 200). However it is not possible to append other 2D images along a third dimension.

Parameters

- **data** (*numpy.ndarray*) – The data to be saved.
- **data_dim** (*int*) – Number of data dimensions. The dimension of the *first_data* is used if set to None.
- **keep_attributes** (*bool*) – All attributes of the old dataset will remain the same if set to True.

Returns None

Return type NoneType

class pynpoint.core.dataio.**Port**(*tag*, *data_storage_in=None*)

Bases: object

Abstract interface and implementation of common functionality of the InputPort, OutputPort, and ConfigPort. Each Port has a internal tag which is its key to a dataset in the DataStorage. If for example data is stored under the entry `im_arr` in the central data storage only a port with the tag (`self._m_tag = im_arr`) can access and change that data. A port knows exactly one DataStorage instance, whether it is active or not (`self._m_data_base_active`).

Abstract constructor of a Port. As input the tag / key is expected which is needed to build the connection to the database entry with the same tag / key. It is possible to give the Port a DataStorage. If this storage is not given the Pypeline module has to set it or the connection needs to be added manually using *set_database_connection()*.

Parameters

- **tag** (*str*) – Input Tag.
- **data_storage_in** (pynpoint.core.dataio.DataStorage) – The data storage to which the port is connected.

Returns None

Return type NoneType

close_port()

Closes the connection to the *DataStorage* and forces it to save the data to the hard drive. All data that was accessed using the port is cleaned from the memory.

Returns None

Return type NoneType

open_port()

Opens the connection to the *DataStorage* and activates its data bank.

Returns None

Return type NoneType

set_database_connection(*data_base_in*)
Sets the internal DataStorage instance.

Parameters **data_base_in** (pynpoint.core.dataio.DataStorage) – The input DataStorage.

Returns None

Return type NoneType

tag
Getter for the internal tag (no setter).

Returns Database tag name.

Return type str

pynpoint.core.processing module

Interfaces for pipeline modules.

class pynpoint.core.processing.**ProcessingModule**(*name_in*)
Bases: *pynpoint.core.processing.PypelineModule*

The abstract class ProcessingModule is an interface for all processing steps in the pipeline which read, process, and store data. Hence processing modules have read and write access to the central database through a dictionary of output ports (self._m_output_ports) and a dictionary of input ports (self._m_input_ports).

Abstract constructor of a ProcessingModule which needs the unique name identifier as input (more information: *pynpoint.core.processing.PypelineModule*). Call this function in all __init__() functions inheriting from this class.

Parameters **name_in** (*str*) – The name of the ProcessingModule.

add_input_port(*tag*)
Function which creates an InputPort for a ProcessingModule and appends it to the internal InputPort dictionary. This function should be used by classes inheriting from ProcessingModule to make sure that only input ports with unique tags are added. The new port can be used as:

```
port = self._m_input_ports[tag]
```

or by using the returned Port.

Parameters **tag** (*str*) – Tag of the new input port.

Returns The new InputPort for the ProcessingModule.

Return type *pynpoint.core.dataio.InputPort*

add_output_port(*tag*, *activation=True*)
Function which creates an OutputPort for a ProcessingModule and appends it to the internal OutputPort dictionary. This function should be used by classes inheriting from ProcessingModule to make sure that only output ports with unique tags are added. The new port can be used as:

```
port = self._m_output_ports[tag]
```

or by using the returned Port.

Parameters

- **tag** (*str*) – Tag of the new output port.

- **activation** (*bool*) – Activation status of the Port after creation. Deactivated ports will not save their results until they are activated.

Returns The new OutputPort for the ProcessingModule.

Return type *pynpoint.core.dataio.OutputPort*

apply_function_in_time(*func*, *image_in_port*, *image_out_port*, *func_args=None*)
Applies a function to all pixel lines in time.

Parameters

- **func** (*function*) – The input function.
- **image_in_port** (pynpoint.core.dataio.InputPort) – Input port which is linked to the input data.
- **image_out_port** (pynpoint.core.dataio.OutputPort) – Output port which is linked to the results.
- **func_args** (*tuple, None*) – Additional arguments which are required by the input function. Not used if set to None.

Returns None

Return type NoneType

apply_function_to_images(*func*, *image_in_port*, *image_out_port*, *message*, *func_args=None*)
Function which applies a function to all images of an input port. Stacks of images are processed in parallel if the CPU and MEMORY attribute are set in the central configuration. The number of images per process is equal to the value of MEMORY divided by the value of CPU. Note that the function *func* is not allowed to change the shape of the images if the input and output port have the same tag and MEMORY is not set to None.

Parameters

- **func** (*function*) – The function which is applied to all images. Its definitions should be similar to:

```
def function(image_in,
             parameter1,
             parameter2,
             parameter3)
```

- **image_in_port** (pynpoint.core.dataio.InputPort) – Input port which is linked to the input data.
- **image_out_port** (pynpoint.core.dataio.OutputPort) – Output port which is linked to the results.
- **message** (*str*) – Progress message that is printed.
- **func_args** (*tuple*) – Additional arguments that are required by the input function.

Returns None

Return type NoneType

connect_database(*data_base_in*)
Function used by a ProcessingModule to connect all ports in the internal input and output port dictionaries to the database. The function is called by Pypeline and connects the DataStorage object to all module ports.

Parameters **data_base_in** (pynpoint.core.dataio.DataStorage) – The central database.

Returns None

Return type NoneType

get_all_input_tags()
Returns a list of all input tags to the ProcessingModule.

Returns List of input tags.

Return type list(str,)

get_all_output_tags()
Returns a list of all output tags to the ProcessingModule.

Returns List of output tags.

Return type list(str,)

run()
Abstract interface for the run method of a *pynpoint.core.processing.ProcessingModule* which inheres the actual algorithm behind the module.

class pynpoint.core.processing.**PypelineModule**(*name_in*)
Bases: object

Abstract interface for the PypelineModule:

- Reading Module (*pynpoint.core.processing.ReadingModule*)
- Writing Module (*pynpoint.core.processing.WritingModule*)
- Processing Module (*pynpoint.core.processing.ProcessingModule*)

Each PypelineModule has a name as a unique identifier in the Pypeline and requires the *connect_database* and *run* methods.

Abstract constructor of a PypelineModule. Needs a name as identifier.

Parameters **name_in** (*str*) – The name of the PypelineModule.

Returns None

Return type NoneType

connect_database(*data_base_in*)
Abstract interface for the function *connect_database* which is needed to connect the Ports of a PypelineModule with the DataStorage.

Parameters **data_base_in** (pynpoint.core.dataio.DataStorage) – The central database.

name
Returns the name of the PypelineModule. This property makes sure that the internal module name can not be changed.

Returns The name of the PypelineModule.

Return type str

run()
Abstract interface for the run method of a PypelineModule which inheres the actual algorithm behind the module.

class pynpoint.core.processing.**ReadingModule**(*name_in*, *input_dir=None*)

Bases: *pynpoint.core.processing.PypelineModule*

The abstract class ReadingModule is an interface for processing steps in the Pypeline which have only read access to the central data storage. One can specify a directory on the hard drive where the input data for the module is located. If no input directory is given then default Pypeline input directory is used. Reading modules have a dictionary of output ports (self._m_out_ports) but no input ports.

Abstract constructor of ReadingModule which needs the unique name identifier as input (more information: *pynpoint.core.processing.PypelineModule*). An input directory can be specified for the location of the data or else the Pypeline default directory is used. This function is called in all *__init__()* functions inheriting from this class.

Parameters

- **name_in** (*str*) – The name of the ReadingModule.
- **input_dir** (*str*) – Directory where the input files are located.

Returns None

Return type NoneType

add_output_port(*tag*, *activation=True*)

Function which creates an OutputPort for a ReadingModule and appends it to the internal OutputPort dictionary. This function should be used by classes inheriting from ReadingModule to make sure that only output ports with unique tags are added. The new port can be used as:

```
port = self._m_output_ports[tag]
```

or by using the returned Port.

Parameters

- **tag** (*str*) – Tag of the new output port.
- **activation** (*bool*) – Activation status of the Port after creation. Deactivated ports will not save their results until they are activated.

Returns The new OutputPort for the ReadingModule.

Return type *pynpoint.core.dataio.OutputPort*

connect_database(*data_base_in*)

Function used by a ReadingModule to connect all ports in the internal input and output port dictionaries to the database. The function is called by Pypeline and connects the DataStorage object to all module ports.

Parameters **data_base_in** (pynpoint.core.dataio.DataStorage) – The central database.

Returns None

Return type NoneType

get_all_output_tags()

Returns a list of all output tags to the ReadingModule.

Returns List of output tags.

Return type list(str,)

run()

Abstract interface for the run method of a ReadingModule which inheres the actual algorithm behind the module.

class pynpoint.core.processing.**WritingModule**(*name_in*, *output_dir=None*)

Bases: *pynpoint.core.processing.PypelineModule*

The abstract class WritingModule is an interface for processing steps in the pipeline which do not change the content of the internal DataStorage. They only have reading access to the central data base. WritingModules can be used to export data from the HDF5 database. WritingModules know the directory on the hard drive where the output of the module can be saved. If no output directory is given the default Pypeline output directory is used. WritingModules have a dictionary of input ports (self._m_input_ports) but no output ports.

Abstract constructor of a WritingModule which needs the unique name identifier as input (more information: *pynpoint.core.processing.PypelineModule*). In addition one can specify a output directory where the module will save its results. If no output directory is given the Pypeline default directory is used. This function is called in all *__init__()* functions inheriting from this class.

Parameters

- **name_in** (*str*) – The name of the WritingModule.
- **output_dir** (*str*) – Directory where the results will be saved.

Returns None

Return type NoneType

add_input_port(*tag*)

Function which creates an InputPort for a WritingModule and appends it to the internal InputPort dictionary. This function should be used by classes inheriting from WritingModule to make sure that only input ports with unique tags are added. The new port can be used as:

```
port = self._m_input_ports[tag]
```

or by using the returned Port.

Parameters **tag** (*str*) – Tag of the new input port.

Returns The new InputPort for the WritingModule.

Return type *pynpoint.core.dataio.InputPort*

connect_database(*data_base_in*)

Function used by a WritingModule to connect all ports in the internal input and output port dictionaries to the database. The function is called by Pypeline and connects the DataStorage object to all module ports.

Parameters **data_base_in** (pynpoint.core.dataio.DataStorage) – The central database.

Returns None

Return type NoneType

get_all_input_tags()

Returns a list of all input tags to the WritingModule.

Returns List of input tags.

Return type list(str,)

run()

Abstract interface for the run method of a WritingModule which inheres the actual algorithm behind the module.

pynpoint.core.pypeline module

Module which capsules the methods of the Pypeline.

class pynpoint.core.pypeline.**Pypeline**(*working_place_in=None*, *input_place_in=None*, *output_place_in=None*)

Bases: object

A Pypeline instance can be used to manage various processing steps. It inheres an internal dictionary of Pypeline steps (modules) and their names. A Pypeline has a central DataStorage on the hard drive which can be accessed by various modules. The order of the modules depends on the order the steps have been added to the pypeline. It is possible to run all modules attached to the Pypeline at once or run a single modules by name.

Constructor of Pypeline.

Parameters

- **working_place_in** (*str*) – Working location of the Pypeline which needs to be a folder on the hard drive. The given folder will be used to save the central PynPoint database (an HDF5 file) in which all the intermediate processing steps are saved. Note that the HDF5 file can become very large depending on the size and number of input images.
- **input_place_in** (*str*) – Default input directory of the Pypeline. All ReadingModules added to the Pypeline use this directory to look for input data. It is possible to specify a different location for the ReadingModules using their constructors.
- **output_place_in** (*str*) – Default result directory used to save the output of all WritingModules added to the Pypeline. It is possible to specify a different locations for the WritingModules by using their constructors.

Returns None

Return type NoneType

add_module(*module*)

Adds a Pypeline module to the internal Pypeline dictionary. The module is appended at the end of this ordered dictionary. If the input module is a reading or writing module without a specified input or output location then the Pypeline default location is used. Moreover, the given module is connected to the Pypeline internal data storage.

Parameters **module** (ReadingModule, WritingModule, or ProcessingModule) – Input pipeline module.

Returns None

Return type NoneType

delete_data(*tag*)

Function for deleting a dataset and related attributes from the central database. Disk space does not seem to free up when using this function.

Parameters **tag** (*str*) – Database tag.

Returns None

Return type NoneType

get_attribute(*data_tag*, *attr_name*, *static=True*)

Function for accessing attributes in the central database.

Parameters

- **data_tag** (*str*) – Database tag.

- **attr_name** (*str*) – Name of the attribute.
- **static** (*bool*) – Static or non-static attribute.

Returns The attribute values.

Return type numpy.ndarray

get_data(*tag*)
Function for accessing data in the central database.

Parameters **tag** (*str*) – Database tag.

Returns The selected dataset from the database.

Return type numpy.ndarray

get_module_names()
Function which returns a list of all module names.

Returns Ordered list of all Pypeline modules.

Return type list(str,)

get_shape(*tag*)
Function for getting the shape of a database entry.

Parameters **tag** (*str*) – Database tag.

Returns Dataset shape.

Return type tuple(int,)

get_tags()
Function for listing the database tags, ignoring header and config tags.

Returns Database tags.

Return type numpy.asarray

remove_module(*name*)
Removes a Pypeline module from the internal dictionary.

Parameters **name** (*str*) – Name of the module that has to be removed.

Returns Confirmation of removal.

Return type bool

run()
Walks through all saved processing steps and calls their run methods. The order in which the steps are called depends on the order they have been added to the Pypeline.

Returns None

Return type NoneType

run_module(*name*)
Runs a single processing module.

Parameters **name** (*str*) – Name of the pipeline module.

Returns None

Return type NoneType

set_attribute(*data_tag*, *attr_name*, *attr_value*, *static=True*)
Function for writing attributes to the central database. Existing values will be overwritten.

Parameters

- **data_tag** (*str*) – Database tag.
- **attr_name** (*str*) – Name of the attribute.
- **attr_value** (*float, int, str, tuple, or numpy.ndarray*) – Attribute value.
- **static** (*bool*) – Static or non-static attribute.

Returns None

Return type NoneType

validate_pipeline()
Function which checks if all input ports of the Pypeline are pointing to previous output ports.

Returns

- *bool* – Confirmation of pipeline validation.
- *str* – Module name that is not valid.

validate_pipeline_module(*name*)
Checks if the data exists for the module with label *name*.

Parameters **name** (*str*) – Name of the module that is checked.

Returns

- *bool* – Confirmation of pipeline module validation.
- *str* – Module name that is not valid.

Module contents

pynpoint.readwrite package

Submodules

pynpoint.readwrite.fitsreading module

Module for reading FITS files.

class pynpoint.readwrite.fitsreading.**FitsReadingModule**(*name_in: str*, *input_dir: str = None*, *image_tag: str = 'im_arr'*, *overwrite: bool = True*, *check: bool = True*, *filenames: Union[str, List[str]] = None*)

Bases: *pynpoint.core.processing.ReadingModule*

Reads FITS files from the given *input_dir* or the default directory of the Pypeline. The FITS files need to contain either single images (2D) or cubes of images (3D). Individual images should have the same shape and type. The header of the FITS is scanned for the required static attributes (should be identical for each FITS file) and non-static attributes. Static entries will be saved as HDF5 attributes while non-static attributes will be saved as separate data sets in a subfolder of the database named *header_* + image_tag. If the FITS files in the input directory have changing static attributes or the shape of the input images is changing a warning appears. FitsReadingModule overwrites by default all existing data with the same tags in the central database.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **input_dir** (*str, None*) – Input directory where the FITS files are located. If not specified the Pypeline default directory is used.
- **image_tag** (*str*) – Tag of the read data in the HDF5 database. Non static header information is stored with the tag: *header_* + image_tag / header_entry_name.
- **overwrite** (*bool*) – Overwrite existing data and header in the central database.
- **check** (*bool*) – Print a warning if certain attributes from the configuration file are not present in the FITS header. If set to *False*, attributes are still written to the dataset but there will be no warning if a keyword is not found in the FITS header.
- **filenames** (*str, list(str,), None*) – If a string, then a path of a text file should be provided. This text file should contain a list of FITS files. If a list, then the paths of the FITS files should be provided directly. If set to None, the FITS files in the *input_dir* are read. All paths should be provided either relative to the Python working folder (i.e., the folder where Python is executed) or as absolute paths.

Returns None

Return type NoneType

read_single_file(*fits_file: str*, *overwrite_tags: list*) → Tuple[astropy.io.fits.header.Header, tuple]
Function which reads a single FITS file and appends it to the database. The function gets a list of *overwriting_tags*. If a new key (header entry or image data) is found that is not on this list the old entry is overwritten if *self.m_overwrite* is active. After replacing the old entry the key is added to the *overwriting_tags*. This procedure guaranties that all previous database information, that does not belong to the new data set that is read by FitsReadingModule is replaced and the rest is kept.

Parameters

- **fits_file** (*str*) – Absolute path and filename of the FITS file.
- **overwrite_tags** (*list(str,)*) – The list of database tags that will not be overwritten.

Returns

- *astropy.io.fits.header.Header* – FITS header.
- *tuple(int,)* – Image shape.

run() → None
Run method of the module. Looks for all FITS files in the input directory and imports the images into the database. Note that previous database information is overwritten if `overwrite=True`. The filenames are stored as attributes.

Returns None

Return type NoneType

pynpoint.readwrite.fitswriting module

Module for writing data as FITS file.

class pynpoint.readwrite.fitswriting.**FitsWritingModule**(*name_in: str*, *data_tag: str*, *file_name: str*, *output_dir: str = None*, *data_range: Tuple[int, int] = None*, *overwrite: bool = True*)

Bases: *pynpoint.core.processing.WritingModule*

Module for writing a data set of the central HDF5 database as FITS file. The data and all attached attributes will be saved. Besides typical image stacks it is possible to export for example non-static header information. To choose the data set from the database its tag / key has to be specified. FitsWritingModule is a Writing Module and supports to use the Pypeline default output directory as well as a own location. See *pynpoint.core.processing.WritingModule* for more information. Note that per default this module will overwrite an existing FITS file with the same filename.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **data_tag** (*str*) – Tag of the database entry the module has to export as FITS file.
- **file_name** (*str*) – Name of the FITS output file. Requires the FITS extension.
- **output_dir** (*str, None*) – Output directory where the FITS file will be stored. If no folder is specified the Pypeline default is chosen.
- **data_range** (*tuple, None*) – A two element tuple which specifies a begin and end frame of the export. This can be used to save a subsets of huge dataset. If None the whole dataset will be exported.
- **overwrite** (*bool*) – Overwrite existing FITS file with identical filename.

Returns None

Return type NoneType

run() → None

Run method of the module. Creates a FITS file and saves the data as well as the corresponding attributes.

Returns None

Return type NoneType

pynpoint.readwrite.hdf5reading module

Module for reading HDF5 files that were created with the *Hdf5WritingModule*.

class pynpoint.readwrite.hdf5reading.**Hdf5ReadingModule**(*name_in: str*, *input_filename: str = None*, *input_dir: str = None*, *tag_dictionary: dict = None*)

Bases: *pynpoint.core.processing.ReadingModule*

Reads an HDF5 file from the given *input_dir* or the default directory of the Pypeline. A tag dictionary has to be set in order to choose the datasets which will be imported into the database. Also the static and non-static attributes are read from the HDF5 file and stored in the database with the corresponding data set. This module should only be used for reading HDF5 files that are created with the Hdf5WritingModule. Reading different type of HDF5 files may lead to inconsistencies in the central database.

Parameters

- **name_in** (*str*) – Unique name of the module instance.

- **input_filename** (*str, None*) – The file name of the HDF5 input file. All files inside the input location will be imported if no filename is provided.
- **input_dir** (*str, None*) – The directory of the input HDF5 file. If no location is given, the default input location of the Pypeline is used.
- **tag_dictionary** (*dict, None*) – Dictionary of all data sets that will be imported. The dictionary format is {*tag_name_in_input_file*:*tag_name_in_database*, }. All data sets in the input HDF5 file that match one of the *tag_name_in_input_file* will be imported. The tag name inside the internal Pypeline database will be changed to *tag_name_in_database*.

Returns None

Return type NoneType

read_single_hdf5 (*file_in: str*) → None
Function which reads a single HDF5 file.

Parameters **file_in** (*str*) – Path and name of the HDF5 file.

Returns None

Return type NoneType

run () → None
Run method of the module. Looks for all HDF5 files in the input directory and reads the datasets that are provided in the tag dictionary.

Returns None

Return type NoneType

pynpoint.readwrite.hdf5writing module

Module for writing a list of tags from the database to a separate HDF5 file.

class pynpoint.readwrite.hdf5writing.**Hdf5WritingModule**(*name_in: str*, *file_name: str*, *output_dir: str = None*, *tag_dictionary: dict = None*, *keep_attributes: bool = True*, *overwrite: bool = False*)

Bases: *pynpoint.core.processing.WritingModule*

Module which exports a part of the PynPoint internal database to a separate HDF5 file. The datasets of the database can be chosen using the *tag_dictionary*. The module will also export the static and non-static attributes.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **file_name** (*str*) – Name of the file which will be created by the module.
- **output_dir** (*str, None*) – Location where the HDF5 file will be stored. The Pypeline default output location is used when no location is given.
- **tag_dictionary** (*dict, None*) – Directory containing all tags / keys of the datasets which will be exported from the PynPoint internal database. The datasets will be exported as {*input_tag*:*output_tag*, }.
- **keep_attributes** (*bool*) – If True all static and non-static attributes will be exported.
- **overwrite** (*bool*) – Overwrite an existing HDF5 file.

Returns None

Return type NoneType

run() → None
Run method of the module. Exports all datasets defined in the *tag_dictionary* to an external HDF5 file.

Returns None

Return type NoneType

pynpoint.readwrite.nearreading module

Module for reading FITS files obtained with VLT/VISIR for the NEAR experiment.

class pynpoint.readwrite.nearreading.**NearReadingModule**(*name_in: str, input_dir: str = None, chopa_out_tag: str = 'chopa', chopb_out_tag: str = 'chopb', subtract: bool = False, crop: Union[Tuple[int, int, float], Tuple[None, None, float]] = None, combine: str = None*)
Bases: *pynpoint.core.processing.ReadingModule*

Pipeline module for reading VLT/VISIR data of the NEAR experiment. The FITS files and required header information are read from the input directory and stored in two datasets, corresponding to chop A and chop B. The primary HDU of the FITS files should contain the main header information, while the subsequent HDUs contain each a single image (alternated for chop A and chop B) and some additional header information for that image. The last HDU is ignored as it contains the average of all images.

Parameters

- **name_in** (*str*) – Unique name of the instance.
- **input_dir** (*str, None*) – Input directory where the FITS files are located. The default input folder of the Pypeline is used if set to None.
- **chopa_out_tag** (*str*) – Database entry where the chop A images will be stored. Should be different from `chop_b_out_tag`.
- **chopb_out_tag** (*str*) – Database entry where the chop B images will be stored. Should be different from `chop_a_out_tag`.
- **subtract** (*bool*) – If True, the other chop position is subtracted before saving out the chop A and chop B images.
- **crop** (*tuple(int, int, float), None*) – The pixel position (x, y) around which the chop A and chop B images are cropped and the new image size (arcsec), together provided as (pos_x, pos_y, size). The same size will be used for both image dimensions. It is recommended to crop the images around the approximate coronagraph position. No cropping is applied if set to None.
- **combine** (*str, None*) – Method ('mean' or 'median') for combining (separately) the chop A and chop B frames from each cube into a single frame. All frames are stored if set to None.

Returns None

Return type NoneType

check_header(*header: astropy.io.fits.header.Header*) → None
Method to check the header information and prompt a warning if a value is not as expected.

Parameters **header** (*astropy.io.fits.header.Header*) – Header information from the FITS file that is read.

Returns None

Return type NoneType

read_header(*filename: str*) → Tuple[astropy.io.fits.header.Header, Tuple[int, int, int]]
Function that opens a FITS file and separates the chop A and chop B images. The primary HDU contains only a general header. The subsequent HDUs contain a single image with a small extra header. The last HDU is the average of all images, which will be ignored.

Parameters **filename** (*str*) – Absolute path and filename of the FITS file.

Returns

- *astropy.io.fits.header.Header* – Primary header, which is valid for all images.
- *tuple(int, int, int)* – Shape of a stack of images for chop A or B.

read_images(*filename: str, im_shape: Tuple[int, int, int]*) → Tuple[numpy.ndarray, numpy.ndarray]
Function that opens a FITS file and separates the chop A and chop B images. The primary HDU contains only a general header. The subsequent HDUs contain a single image with a small extra header. The last HDU is the average of all images, which will be ignored.

Parameters

- **filename** (*str*) – Absolute path and filename of the FITS file.
- **im_shape** (*tuple(int, int, int)*) – Shape of a stack of images for chop A or B.

Returns

- *numpy.array* – Array containing the images of chop A.
- *numpy.array* – Array containing the images of chop B.

run() → None
Run the module. The FITS files are collected from the input directory and uncompressed if needed. The images are then sorted by the two chop positions (chop A and chop B). The required FITS header keywords (which should be set in the configuration file) are also imported and stored as attributes to the two output datasets in the HDF5 database.

Returns None

Return type NoneType

uncompress() → None
Method to check if the input directory contains compressed files ending with .fits.Z. If this is the case, the files will be uncompressed using multithreading. The number of threads can be set with the `CPU` parameter in the configuration file.

Returns None

Return type NoneType

pynpoint.readwrite.textreading module

Modules for reading data from a text file.

class pynpoint.readwrite.textreading.**AttributeReadingModule**(*name_in: str*, *data_tag: str*, *file_name: str*, *attribute: str*, *input_dir: str = None*, *overwrite: bool = False*)

Bases: *pynpoint.core.processing.ReadingModule*

Module for reading a list of values from a text file and appending them as a non-static attributes to a dataset.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **data_tag** (*str*) – Tag of the database entry to which the attribute is written.
- **file_name** (*str*) – Name of the input file with a list of values.
- **attribute** (*str*) – Name of the attribute as to be written in the database.
- **input_dir** (*str, None*) – Input directory where the text file is located. If not specified the Pypeline default directory is used.
- **overwrite** (*bool*) – Overwrite if the attribute is already exists.

Returns None

Return type NoneType

run() → None

Run method of the module. Reads a list of values from a text file and writes them as non-static attribute to a dataset.

Returns None

Return type NoneType

class pynpoint.readwrite.textreading.**ParangReadingModule**(*name_in: str*, *data_tag: str*, *file_name: str*, *input_dir: str = None*, *overwrite: bool = False*)

Bases: *pynpoint.core.processing.ReadingModule*

Module for reading a list of parallactic angles from a text file.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **data_tag** (*str*) – Tag of the database entry to which the PARANG attribute is written.
- **file_name** (*str*) – Name of the input file with a list of parallactic angles (deg). Should be equal in size to the number of images in *data_tag*.
- **input_dir** (*str, None*) – Input directory where the text file is located. If not specified the Pypeline default directory is used.
- **overwrite** (*bool*) – Overwrite if the PARANG attribute already exists.

Returns None

Return type NoneType

run () → None
Run method of the module. Reads the parallactic angles from a text file and writes the values as non-static attribute (PARANG) to the database tag.

Returns None

Return type NoneType

pynpoint.readwrite.textwriting module

Modules for writing data as text file.

class pynpoint.readwrite.textwriting.**AttributeWritingModule**(*name_in: str*, *data_tag: str*, *attribute: str*, *file_name: str = 'attributes.dat'*, *output_dir: str = None*, *header: str = None*)

Bases: *pynpoint.core.processing.WritingModule*

Module for writing a 1D or 2D array of non-static attributes to a text file.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **data_tag** (*str*) – Tag of the database entry from which the PARANG attribute is read.
- **attribute** (*str*) – Name of the non-static attribute as given in the central database (e.g., 'INDEX' or 'STAR_POSITION').
- **file_name** (*str*) – Name of the output file.
- **output_dir** (*str, None*) – Output directory where the text file will be stored. If no path is specified then the Pypeline default output location is used.
- **header** (*str, None*) – Header that is written at the top of the text file.

Returns None

Return type NoneType

run () → None
Run method of the module. Writes the non-static attributes (1D or 2D) to a a text file.

Returns None

Return type NoneType

class pynpoint.readwrite.textwriting.**ParangWritingModule**(*name_in: str*, *data_tag: str*, *file_name: str = 'parang.dat'*, *output_dir: str = None*, *header: str = None*)

Bases: *pynpoint.core.processing.WritingModule*

Module for writing a list of parallactic angles to a text file.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **data_tag** (*str*) – Tag of the database entry from which the PARANG attribute is read.

- **file_name** (*str*) – Name of the output file.
- **output_dir** (*str, None*) – Output directory where the text file will be stored. If no path is specified then the Pypeline default output location is used.
- **header** (*str, None*) – Header that is written at the top of the text file.

Returns None

Return type NoneType

run() → None
Run method of the module. Writes the parallactic angles from the PARANG attribute of the specified database tag to a a text file.

Returns None

Return type NoneType

class pynpoint.readwrite.textwriting.**TextWritingModule**(*name_in: str*, *data_tag: str*, *file_name: str*, *output_dir: str = None*, *header: str = None*)
Bases: *pynpoint.core.processing.WritingModule*

Module for writing a 1D or 2D data set from the central HDF5 database as text file. TextWritingModule is a *pynpoint.core.processing.WritingModule* and supports the use of the Pypeline default output directory as well as a specified location.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **data_tag** (*str*) – Tag of the database entry from which data is exported.
- **file_name** (*str*) – Name of the output file.
- **output_dir** (*str, None*) – Output directory where the text file will be stored. If no path is specified then the Pypeline default output location is used.
- **header** (*str, None*) – Header that is written at the top of the text file.

Returns None

Return type NoneType

run() → None
Run method of the module. Saves the specified data from the database to a text file.

Returns None

Return type NoneType

Module contents

pynpoint.processing package

Submodules

pynpoint.processing.background module

Pipeline modules for subtraction of the background emission.

class pynpoint.processing.background.**LineSubtractionModule**(*name_in: str*, *image_in_tag: str*, *image_out_tag: str*, *combine: str = 'median'*, *mask=None*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for subtracting the background emission from each pixel by computing the mean or median of all values in the row and column of the pixel. The module can for example be used if no background data is available or to remove a detector bias.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output.
- **combine** (*str*) – The method by which the column and row pixel values are combined ('median' or 'mean'). Using a mean-combination is computationally faster than a median-combination.
- **mask** (*float, None*) – The radius of the mask within which pixel values are ignored. No mask is used if set to None.

Returns None

Return type NoneType

run() → None

Run method of the module. Selects the pixel values in the column and row at each pixel position, computes the mean or median value while excluding pixels within the radius of the mask, and subtracts the mean or median value from each pixel separately.

Returns None

Return type NoneType

class pynpoint.processing.background.**MeanBackgroundSubtractionModule**(*name_in: str*, *image_in_tag: str*, *image_out_tag: str*, *shift: int = None*, *cubes: int = 1*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for mean background subtraction. Only applicable on data obtained with dithering.

name_in [str] Unique name of the module instance.

image_in_tag [str] Tag of the database entry that is read as input.

image_out_tag [str] Tag of the database entry that is written as output. Should be different from *image_in_tag*.

shift [int, None] Image index offset for the background subtraction. Typically equal to the number of frames per dither location. If set to None, the NFRAMES attribute will be used to select the background frames automatically. The *cubes* parameters should be set when *shift* is set to None.

cubes [int] Number of consecutive cubes per dithering position.

Returns None

Return type NoneType

run() → None
Run method of the module. Mean background subtraction which uses either a constant index offset or the NFRAMES attributes. The mean background is calculated from the cubes before and after the science cube.

Returns None

Return type NoneType

class pynpoint.processing.background.**NoddingBackgroundModule**(*name_in: str*, *science_in_tag: str*, *sky_in_tag: str*, *image_out_tag: str*, *mode: str = 'both'*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for background subtraction of data obtained with nodding (e.g., NACO AGPM data). Before using this module, the sky images should be stacked with the StackCubesModule such that each image in the stack of sky images corresponds to the mean combination of a single FITS data cube.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **science_in_tag** (*str*) – Tag of the database entry with science images that are read as input.
- **sky_in_tag** (*str*) – Tag of the database entry with sky images that are read as input. The *StackCubesModule* should be used on the sky images beforehand.
- **image_out_tag** (*str*) – Tag of the database entry with sky subtracted images that are written as output.
- **mode** (*str*) – Sky images that are subtracted, relative to the science images. Either the next, previous, or average of the next and previous cubes of sky frames can be used by choosing 'next', 'previous', or 'both', respectively.

Returns None

Return type NoneType

calc_sky_frame(*index_of_science_data*)
Method for finding the required sky frame (next, previous, or the mean of next and previous) by comparing the time stamp of the science frame with preceding and following sky frames.

run() → None
Run method of the module. Create list of time stamps, get sky and science images, and subtract the sky images from the science images.

Returns None

Return type NoneType

class pynpoint.processing.background.**SimpleBackgroundSubtractionModule**(*name_in: str*, *image_in_tag: str*, *image_out_tag: str*, *shift: int*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for simple background subtraction. Only applicable on data obtained with dithering.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output.
- **shift** (*int*) – Frame index offset for the background subtraction. Typically equal to the number of frames per dither location.

Returns None

Return type NoneType

run() → None

Run method of the module. Simple background subtraction with a constant index offset.

Returns None

Return type NoneType

pynpoint.processing.badpixel module

Pipeline modules for the detection and interpolation of bad pixels.

class pynpoint.processing.badpixel.**BadPixelInterpolationModule**(*name_in: str*, *image_in_tag: str*, *bad_pixel_map_tag: str*, *image_out_tag: str*, *iterations: int = 1000*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module to interpolate bad pixels with spectral deconvolution.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry with the images that are read as input.
- **bad_pixel_map_tag** (*str*) – Tag of the database entry with the bad pixel map that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output.

- **iterations** (*int*) – Number of iterations of the spectral deconvolution.

Returns None

Return type NoneType

run() → None
Run method of the module. Interpolates bad pixels with an iterative spectral deconvolution.

Returns None

Return type NoneType

class pynpoint.processing.badpixel.**BadPixelMapModule**(*name_in: str, dark_in_tag: Optional[str], flat_in_tag: Optional[str], bp_map_out_tag: str, dark_threshold: float = 0.2, flat_threshold: float = 0.2*)
Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module to create a bad pixel map from the dark frames and flat fields.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **dark_in_tag** (*str, None*) – Tag of the database entry with the dark frames that are read as input. Not read if set to None.
- **flat_in_tag** (*str, None*) – Tag of the database entry with the flat fields that are read as input. Not read if set to None.
- **bp_map_out_tag** (*str*) – Tag of the database entry with the bad pixel map that is written as output.
- **dark_threshold** (*float*) – Fractional threshold with respect to the maximum pixel value in the dark frame to flag bad pixels. Pixels *brighter* than the fractional threshold are flagged as bad.
- **flat_threshold** (*float*) – Fractional threshold with respect to the maximum pixel value in the flat field to flag bad pixels. Pixels *fainter* than the fractional threshold are flagged as bad.

Returns None

Return type NoneType

run() → None
Run method of the module. Collapses a cube of dark frames and flat fields if needed, flags bad pixels by comparing the pixel values with the threshold times the maximum value, and writes a bad pixel map to the database. For the dark frame, pixel values larger than the threshold will be flagged while for the flat frame pixel values smaller than the threshold will be flagged.

Returns None

Return type NoneType

class pynpoint.processing.badpixel.**BadPixelSigmaFilterModule**(*name_in: str*, *image_in_tag: str*, *image_out_tag: str*, *map_out_tag: str = None*, *box: int = 9*, *sigma: float = 5.0*, *iterate: int = 1*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for finding bad pixels with a sigma filter and replacing them with the mean value of the surrounding pixels.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output.
- **map_out_tag** (*str, None*) – Tag of the database entry with the bad pixel map that is written as output. No data is written if set to None. This output port can not be used if CPU > 1.
- **box** (*int*) – Size of the sigma filter. The area of the filter is equal to the squared value of *box*.
- **sigma** (*float*) – Sigma threshold.
- **iterate** (*int*) – Number of iterations.

Returns None

Return type NoneType

run() → None

Run method of the module. Finds bad pixels with a sigma filter, replaces bad pixels with the mean value of the surrounding pixels, and writes the cleaned images to the database.

Returns None

Return type NoneType

class pynpoint.processing.badpixel.**BadPixelTimeFilterModule**(*name_in: str*, *image_in_tag: str*, *image_out_tag: str*, *sigma: Tuple[float, float] = (5.0, 5.0)*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for finding bad pixels with a sigma filter along a pixel line in time. This module is suitable for removing bad pixels that are only present at a position in a small number of images, for example because a dither pattern has been applied. Pixel lines can be processed in parallel by setting the CPU keyword in the configuration file.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output.
- **sigma** (*tuple(float, float)*) – Lower and upper sigma threshold as (lower, upper).

Returns None

Return type NoneType

run() → None
Run method of the module. Finds bad pixels along a pixel line, replaces the bad pixels with the mean value of the pixels (excluding the bad pixels), and writes the cleaned images to the database.

Returns None

Return type NoneType

class pynpoint.processing.badpixel.**ReplaceBadPixelsModule**(*name_in: str, image_in_tag: str, map_in_tag: str, image_out_tag: str, size: int = 2, replace: str = 'median'*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for replacing bad pixels with the mean are median value of the surrounding pixels. The bad pixels are selected from the input bad pixel map.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output.
- **sigma** (*tuple(float, float)*) – Lower and upper sigma threshold as (lower, upper).
- **size** (*int*) – Number of pixel lines around the bad pixel that are used to calculate the median or mean replacement value. For example, a 5x5 window is used if `size=2`.
- **replace** (*str*) – Replace the bad pixel with the 'median', 'mean' or 'nan'.

Returns None

Return type NoneType

run() → None
Run method of the module. Masks the bad pixels with NaN and replaces the bad pixels with the mean or median value (excluding the bad pixels) within a window centered on the bad pixel. The original value is used if there are only NaNs within the window.

Returns None

Return type NoneType

pynpoint.processing.basic module

Pipeline modules for basic image operations.

class pynpoint.processing.basic.**AddImagesModule**(*name_in: str, image_in_tags: Tuple[str, str], image_out_tag: str, scaling: float = 1.0*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for adding two sets of images.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tags** (*tuple(str, str)*) – Tuple with two tags of the database entry that are read as input.
- **image_out_tag** (*str*) – Tag of the database entry with the added images that are written as output.
- **scaling** (*float*) – Additional scaling factor.

Returns None

Return type NoneType

run() → None
Run method of the module. Add the images from the two database tags on a frame-by-frame basis.

Returns None

Return type NoneType

class pynpoint.processing.basic.**RepeatImagesModule**(*name_in: str*, *image_in_tag: str*, *image_out_tag: str*, *repeat: int*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for repeating the images from a dataset.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry with the added images that are written as output.
- **repeat** (*int*) – The number of times the input images get repeated.

Returns None

Return type NoneType

run() → None
Run method of the module. Repeats the stack of input images a specified number of times.

Returns None

Return type NoneType

class pynpoint.processing.basic.**RotateImagesModule**(*name_in: str*, *image_in_tag: str*, *image_out_tag: str*, *angle: float*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for rotating images.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output.
- **angle** (*float*) – Rotation angle (deg). Rotation is clockwise for positive values.

Returns None

Return type NoneType

run() → None
Run method of the module. Rotates all images by a constant angle.

Returns None

Return type NoneType

class pynpoint.processing.basic.**SubtractImagesModule**(*name_in: str, image_in_tags: Tuple[str, str], image_out_tag: str, scaling: float = 1.0*)
Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for subtracting two sets of images.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tags** (*tuple(str, str)*) – Tuple with two tags of the database entry that are read as input.
- **image_out_tag** (*str*) – Tag of the database entry with the subtracted images that are written as output.
- **scaling** (*float*) – Additional scaling factor.

Returns None

Return type NoneType

run() → None
Run method of the module. Subtracts the images from the second database tag from the images of the first database tag, on a frame-by-frame basis.

Returns None

Return type NoneType

pynpoint.processing.centering module

Pipeline modules for aligning and centering of the star.

class pynpoint.processing.centering.**FitCenterModule**(*name_in: str, image_in_tag: str, fit_out_tag: str, mask_out_tag: str = None, method: str = 'full', radius: float = 0.1, sign: str = 'positive', model: str = 'gaussian', filter_size: float = None, **kwargs*)
Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for fitting the PSF with a 2D Gaussian or Moffat function.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry with images that are read as input.
- **fit_out_tag** (*str*) – Tag of the database entry with the best-fit results of the model fit and the 1-sigma errors. Data is written in the following format: x offset (pix), x offset error (pix) y offset (pix), y offset error (pix), FWHM major axis (arcsec), FWHM major axis error (arcsec), FWHM minor axis (arcsec), FWHM minor axis error (arcsec), amplitude (counts),

amplitude error (counts), angle (deg), angle error (deg) measured in counterclockwise direction with respect to the upward direction (i.e., East of North), offset (counts), offset error (counts), power index (only for Moffat function), and power index error (only for Moffat function). Not used if set to None.

- **mask_out_tag** (*str, None*) – Tag of the database entry with the masked images that are written as output. The unmasked part of the images is used for the fit. The effect of the smoothing that is applied by setting the *fwhm* parameter is also visible in the data of the *mask_out_tag*. Data is not written when set to None.
- **method** (*str*) – Fit and shift all the images individually ('full') or only fit the mean of the cube and shift all images to that location ('mean'). The 'mean' method could be used after running the *StarAlignmentModule*.
- **radius** (*float*) – Radius (arcsec) around the center of the image beyond which pixels are neglected with the fit. The radius is centered on the position specified in *guess*, which is the center of the image by default.
- **sign** (*str*) – Fit a 'positive' or 'negative' Gaussian/Moffat. A negative model can be used to center coronagraphic data in which a dark hole is present.
- **model** (*str*) – Type of 2D model used to fit the PSF ('gaussian' or 'moffat'). Both models are elliptical in shape.
- **filter_size** (*float, None*) – Standard deviation (arcsec) of the Gaussian filter that is used to smooth the images before fitting the model. No filter is applied if set to None.

Keyword Arguments **guess** (*tuple(float, float, float, float, float, float, float, float)*) – The initial parameter values for the least squares fit: x offset with respect to center (pix), y offset with respect to center (pix), FWHM x (pix), FWHM y (pix), amplitude (counts), angle (deg), offset (counts), and power index (only for Moffat function).

Returns None

Return type NoneType

run() → None

Run method of the module. Uses a non-linear least squares (Levenberg-Marquardt) to fit the the individual images or the mean of the stack with a 2D Gaussian or Moffat function, and stores the best fit results. The fitting results contain zeros in case the algorithm could not converge. The *fit_out_tag* can be directly used as input for the *shift_xy* argument of the *ShiftImagesModule*.

Returns None

Return type NoneType

class pynpoint.processing.centering.**ShiftImagesModule**(*name_in: str, image_in_tag: str, image_out_tag: str, shift_xy: Union[Tuple[float, float], str], interpolation: str = 'spline'*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for shifting a stack of images.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output.

- **shift_xy** (*tuple(float, float), str*) – The shift (pix) in x and y direction as (delta_x, delta_y). Or, a database tag with the fit results from the *FitCenterModule*.
- **interpolation** (*str*) – Interpolation type for shifting of the images ('spline', 'bilinear', or 'fft').

Returns None

Return type NoneType

run() → None
Run method of the module. Shifts an image with a fifth order spline, bilinear, or a Fourier shift interpolation.

Returns None

Return type NoneType

class pynpoint.processing.centering.**StarAlignmentModule**(*name_in: str, image_in_tag: str, image_out_tag: str, ref_image_in_tag: str = None, interpolation: str = 'spline', accuracy: float = 10.0, resize: float = None, num_references: int = 10, subframe: float = None*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module to align the images with a cross-correlation in Fourier space.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry with the stack of images that is read as input.
- **ref_image_in_tag** (*str, None*) – Tag of the database entry with the reference image(s) that are read as input. If it is set to None, a random subsample of *num_references* elements of *image_in_tag* is taken as reference images.
- **image_out_tag** (*str*) – Tag of the database entry with the images that are written as output.
- **interpolation** (*str*) – Type of interpolation that is used for shifting the images (spline, bilinear, or fft).
- **accuracy** (*float*) – Upsampling factor for the cross-correlation. Images will be registered to within 1/accuracy of a pixel.
- **resize** (*float, None*) – Scaling factor for the up/down-sampling before the images are shifted. Not used if set to None.
- **num_references** (*int*) – Number of reference images for the cross-correlation.
- **subframe** (*float, None*) – Size (arcsec) of the subframe around the image center that is used for the cross-correlation. The full image is used if set to None.

Returns None

Return type NoneType

run() → None
Run method of the module. Applies a cross-correlation of the input images with respect to a stack of reference images, rescales the image dimensions, and shifts the images to a common center.

Returns None

Return type NoneType

class pynpoint.processing.centering.**WaffleCenteringModule**(*name_in: str*, *image_in_tag: str*, *center_in_tag: str*, *image_out_tag: str*, *size: float = None*, *center: Tuple[float, float] = None*, *radius: float = 45.0*, *pattern: str = 'x'*, *sigma: float = 0.06*, *dither: bool = False*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for centering of SPHERE data obtained with a Lyot coronagraph for which center frames with satellite spots are available.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry with science images that are read as input.
- **center_in_tag** (*str*) – Tag of the database entry with the center frame that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry with the centered images that are written as output. Should be different from *image_in_tag*.
- **size** (*float, None*) – Image size (arcsec) for both dimensions. Original image size is used if set to None.
- **center** (*tuple(float, float), None*) – Approximate position (x0, y0) of the coronagraph. The center of the image is used if set to None.
- **radius** (*float*) – Approximate separation (pix) of the waffle spots from the star.
- **pattern** (*str*) – Waffle pattern that is used ('x' or '+').
- **sigma** (*float*) – Standard deviation (arcsec) of the Gaussian kernel that is used for the unsharp masking.
- **dither** (*bool*) – Apply dithering correction based on the DITHER_X and DITHER_Y attributes.

Returns None

Return type NoneType

run() → None
Run method of the module. Locates the position of the calibration spots in the center frame. From the four spots, the position of the star behind the coronagraph is fitted, and the images are shifted and cropped.

Returns None

Return type NoneType

pynpoint.processing.darkflat module

Pipeline modules for dark frame and flat field calibrations.

class pynpoint.processing.darkflat.**DarkCalibrationModule**(*name_in: str*, *image_in_tag: str*, *dark_in_tag: str*, *image_out_tag: str*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module to subtract a master dark from the science data.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry with the science images that are read as input.
- **dark_in_tag** (*str*) – Tag of the database with the dark frames that are read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output.

Returns None

Return type NoneType

run() → None
Run method of the module. Creates a master dark with the same shape as the science data and subtracts the dark frame from the science data.

Returns None

Return type NoneType

class pynpoint.processing.darkflat.**FlatCalibrationModule**(*name_in: str*, *image_in_tag: str*, *flat_in_tag: str*, *image_out_tag: str*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module to apply a flat field correction to the science data.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the science database that is read as input.
- **dark_in_tag** (*str*) – Tag of the flat field database that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output.

Returns None

Return type NoneType

run() → None
Run method of the module. Creates a master flat with the same shape as the science image and divides the science images by the flat field.

Returns None

Return type NoneType

pynpoint.processing.extract module

Pipeline modules for locating and extracting the position of a star.

class pynpoint.processing.extract.**ExtractBinaryModule**(*name_in: str, image_in_tag: str, image_out_tag: str, pos_center: Tuple[float, float], pos_binary: Tuple[float, float], image_size: float = 2.0, search_size: float = 0.1, filter_size: float = None*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module to extract a binary star (or another point source) which is rotating across the image stack.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the dataset with the input images.
- **image_out_tag** (*str*) – Tag of the dataset that is stored as output, containing the extracted images.
- **pos_center** (*tuple(float, float)*) – Approximate position (x, y) of the center of rotation (pix).
- **pos_binary** (*tuple(float, float)*) – Approximate position (x, y) of the binary star in the first image (pix).
- **image_size** (*float*) – Cropped image size (arcsec).
- **search_size** (*float*) – Window size (arcsec) in which the brightest pixel is selected as position of the binary star. The search window is centered on the position that for each image is calculated from the `pos_center`, `pos_binary`, and parallactic angle (`PARANG`) of the image.
- **filter_size** (*float, None*) – Full width at half maximum (arcsec) of the Gaussian kernel that is used to smooth the images to lower contributions of bad pixels.

Returns None

Return type NoneType

run() → None

Run method of the module. Locates the position of a binary star (or some other point source) which rotates across the stack of images due to parallactic rotation. The approximate position of the binary star is calculated by taking into account the parallactic angle of each image separately. The brightest pixel is then selected as center around which the image is cropped.

Returns None

Return type NoneType

class pynpoint.processing.extract.**StarExtractionModule**(*name_in: str, image_in_tag: str, image_out_tag: str, index_out_tag: str = None, image_size: float = 2.0, fwhm_star: float = 0.2, position: Union[Tuple[int, int, float], Tuple[None, None, float]] = None*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module to locate the position of the star in each image and to crop all the images around this position.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the dataset with the input images.
- **image_out_tag** (*str*) – Tag of the dataset that is stored as output, containing the extracted images.
- **index_out_tag** (*str, None*) – List with image indices for which the image size is too large to be cropped around the brightest pixel. No data is written if set to None. This tag name can be provided to the `frames` parameter in *RemoveFramesModule*. This argument is ignored if `CPU` is set to a value larger than 1.
- **image_size** (*float*) – Cropped image size (arcsec).
- **fwhm_star** (*float*) – Full width at half maximum (arcsec) of the Gaussian kernel that is used to smooth the images to lower contributions of bad pixels.
- **position** (*tuple(int, int, float), None*) – Subframe that is selected to search for the star. The tuple should contain a position (pix) and size (arcsec) as (pos_x, pos_y, size). The full image is used if set to None. The center of the image will be used with `position=(None, None, size)`.

Returns None

Return type NoneType

run() → None
Run method of the module. Locates the position of the star (only pixel precision) by selecting the highest pixel value. A Gaussian kernel with a FWHM similar to the PSF is used to lower the contribution of bad pixels which may have higher values than the peak of the PSF. Images are cropped and written to an output port. The position of the star is attached to the input images (only with `CPU == 1`) as the non-static attribute `STAR_POSITION` (y, x).

Returns None

Return type NoneType

pynpoint.processing.fluxposition module

Pipeline modules for photometric and astrometric measurements.

class pynpoint.processing.fluxposition.**AperturePhotometryModule**(*name_in: str*, *image_in_tag: str*, *phot_out_tag: str*, *radius: float = 0.1*, *position: Tuple[float, float] = None*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for calculating the counts within a circular area.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **phot_out_tag** (*str*) – Tag of the database entry with the photometry values that are written as output.
- **radius** (*float*) – Radius (arcsec) of the circular aperture.
- **position** (*tuple(float, float), None*) – Center position (pix) of the aperture, (x, y), with subpixel precision. The center of the image will be used if set to None. Python indexing starts at zero so the bottom left corner of the image has coordinates (-0.5, -0.5).

Returns None

Return type NoneType

run() → None
Run method of the module. Computes the flux within a circular aperture for each frame and saves the values in the database.

Returns None

Return type NoneType

class pynpoint.processing.fluxposition.**FakePlanetModule**(*name_in: str, image_in_tag: str, psf_in_tag: str, image_out_tag: str, position: Tuple[float, float], magnitude: float, psf_scaling: float = 1.0, interpolation: str = 'spline'*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module to inject a positive or negative artificial planet into a stack of images.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry with images that are read as input.
- **psf_in_tag** (*str*) – Tag of the database entry that contains the reference PSF that is used as fake planet. Can be either a single image (2D) or a cube (3D) with the dimensions equal to *image_in_tag*.
- **image_out_tag** (*str*) – Tag of the database entry with images that are written as output.
- **position** (*tuple(float, float)*) – Angular separation (arcsec) and position angle (deg) of the fake planet. Angle is measured in counterclockwise direction with respect to the upward direction (i.e., East of North).
- **magnitude** (*float*) – Magnitude of the fake planet with respect to the star.
- **psf_scaling** (*float*) – Additional scaling factor of the planet flux (e.g., to correct for a neutral density filter). A negative value will inject a negative planet signal.
- **interpolation** (*str*) – Type of interpolation that is used for shifting the images (spline, bilinear, or fft).

Returns None

Return type NoneType

run() → None
Run method of the module. Shifts the PSF template to the location of the fake planet with an additional correction for the parallactic angle and an optional flux scaling. The stack of images with the injected planet signal is stored.

Returns None

Return type NoneType

class pynpoint.processing.fluxposition.**FalsePositiveModule**(*name_in: str, image_in_tag: str, snr_out_tag: str, position: Tuple[float, float], aperture: float = 0.1, ignore: bool = False, optimize: bool = False, **kwargs*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module to calculate the signal-to-noise ratio (SNR) and false positive fraction (FPF) at a specified location in an image by using the Student's t-test (Mawet et al. 2014). Optionally, the SNR can be optimized with the aperture position as free parameter.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry with the images that are read as input.
- **snr_out_tag** (*str*) – Tag of the database entry that is written as output. The output format is: (x position (pix), y position (pix), separation (arcsec), position angle (deg), SNR, FPF). The position angle is measured in counterclockwise direction with respect to the upward direction (i.e., East of North).
- **position** (*tuple(float, float)*) – The x and y position (pix) where the SNR and FPF is calculated. Note that the bottom left of the image is defined as (-0.5, -0.5) so there is a -1.0 offset with respect to the DS9 coordinate system. Aperture photometry corrects for the partial inclusion of pixels at the boundary.
- **aperture** (*float*) – Aperture radius (arcsec).
- **ignore** (*bool*) – Ignore the two neighboring apertures that may contain self-subtraction from the planet.
- **optimize** (*bool*) – Optimize the SNR. The aperture position is written in the history. The size of the aperture is kept fixed.

Keyword Arguments

- **tolerance** (*float*) – The fractional tolerance on the position for the optimization to end. Default is set to 1e-3.
- **bounds** (*tuple(tuple(float, float), tuple(float, float))*) – Boundaries (pix) for the horizontal and vertical offset with respect to the *position*. The default is set to (-3, 3) for both directions.

Returns None

Return type NoneType

run() → None
Run method of the module. Calculates the SNR and FPF for a specified position in a post- processed

image with the Student's t-test (Mawet et al. 2014). This approach assumes Gaussian noise but accounts for small sample statistics.

Returns None

Return type NoneType

class pynpoint.processing.fluxposition.**MCMCsamplingModule**(*name_in: str, image_in_tag: str, psf_in_tag: str, chain_out_tag: str, param: Tuple[float, float, float], bounds: Tuple[Tuple[float, float], Tuple[float, float], Tuple[float, float]], nwalkers: int = 100, nsteps: int = 200, psf_scaling: float = -1.0, pca_number: int = 20, aperture: Union[float, Tuple[int, int, float]] = 0.1, mask: Union[Tuple[float, float], Tuple[None, float], Tuple[float, None], Tuple[None, None]] = None, extra_rot: float = 0.0, merit: str = 'gaussian', residuals: str = 'median', **kwargs*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module to measure the separation, position angle, and contrast of a planet with injection of negative artificial planets and sampling of the posterior distributions with emcee, an affine invariant Markov chain Monte Carlo (MCMC) ensemble sampler.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry with images that are read as input.
- **psf_in_tag** (*str*) – Tag of the database entry with the reference PSF that is used as fake planet. Can be either a single image (2D) or a cube (3D) with the dimensions equal to *image_in_tag*.
- **chain_out_tag** (*str*) – Tag of the database entry with the Markov chain that is written as output. The shape of the array is (nwalkers*nsteps, 3).
- **param** (*tuple(float, float, float)*) – The approximate separation (arcsec), angle (deg), and contrast (mag), for example obtained with the *SimplexMinimizationModule*. The angle is measured in counterclockwise direction with respect to the upward direction (i.e., East of North). The specified separation and angle are also used as fixed position for the aperture if *aperture* contains a float value.
- **bounds** (*tuple(tuple(float, float), tuple(float, float), tuple(float, float))*) – The boundaries of the separation (arcsec), angle (deg), and contrast (mag). Each set of boundaries is specified as a tuple.

- **nwalkers** (*int*) – Number of ensemble members.
- **nsteps** (*int*) – Number of steps to run per walker.
- **psf_scaling** (*float*) – Additional scaling factor of the planet flux (e.g., to correct for a neutral density filter). Should be negative in order to inject negative fake planets.
- **pca_number** (*int*) – Number of principal components used for the PSF subtraction.
- **aperture** (*float, tuple(int, int, float)*) – Either the aperture radius (arcsec) at the position of *param* or tuple with the position and aperture radius (arcsec) as (pos_x, pos_y, radius).
- **mask** (*tuple(float, float), None*) – Inner and outer mask radius (arcsec) for the PSF subtraction. Both elements of the tuple can be set to None. Masked pixels are excluded from the PCA computation, resulting in a smaller runtime.
- **extra_rot** (*float*) – Additional rotation angle of the images (deg).
- **merit** (*str*) – Figure of merit that is used for the likelihood function ('gaussian' or 'poisson'). Pixels are assumed to be independent measurements which are expected to be equal to zero in case the best-fit negative PSF template is injected. With 'gaussian', the variance is estimated from the pixel values within an annulus at the separation of the aperture (but excluding the pixels within the aperture). With 'poisson', a Poisson distribution is assumed for the variance of each pixel value (see Wertz et al. 2017).
- **residuals** (*str*) – Method used for combining the residuals ('mean', 'median', 'weighted', or 'clipped').

Keyword Arguments

- **scale** (*float*) – The proposal scale parameter (Goodman & Weare 2010). The default is set to 2.
- **sigma** (*tuple(float, float, float)*) – The standard deviations that randomly initializes the start positions of the walkers in a small ball around the a priori preferred position. The tuple should contain a value for the separation (arcsec), position angle (deg), and contrast (mag). The default is set to (1e-5, 1e-3, 1e-3).

Returns None

Return type NoneType

run() → None

Run method of the module. The posterior distributions of the separation, position angle, and flux contrast are sampled with the affine invariant Markov chain Monte Carlo (MCMC) ensemble sampler emcee. At each step, a negative copy of the PSF template is injected and the likelihood function is evaluated at the approximate position of the planet.

Returns None

Return type NoneType

class pynpoint.processing.fluxposition.**SimplexMinimizationModule**(*name_in: str, image_in_tag: str, psf_in_tag: str, res_out_tag: str, flux_position_tag: str, position: Tuple[int, int], magnitude: float, psf_scaling: float = -1.0, merit: str = 'hessian', aperture: float = 0.1, sigma: float = 0.0, tolerance: float = 0.1, pca_number: int = 20, cent_size: float = None, edge_size: float = None, extra_rot: float = 0.0, residuals: str = 'median', reference_in_tag: str = None*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module to measure the flux and position of a planet by injecting negative fake planets and minimizing a figure of merit.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry with the science images that are read as input.
- **psf_in_tag** (*str*) – Tag of the database entry with the reference PSF that is used as fake planet. Can be either a single image or a stack of images equal in size to `image_in_tag`.
- **res_out_tag** (*str*) – Tag of the database entry with the image residuals that are written as output. The residuals are stored for each step of the minimization. The last image contains the best-fit residuals.
- **flux_position_tag** (*str*) – Tag of the database entry with the flux and position results that are written as output. Each step of the minimization stores the x position (pix),

y position (pix), separation (arcsec), angle (deg), contrast (mag), and the chi-square value. The last row contains the best-fit results.

- **position** (*tuple(int, int)*) – Approximate position (x, y) of the planet (pix). This is also the location where the figure of merit is calculated within an aperture of radius `aperture`.
- **magnitude** (*float*) – Approximate magnitude of the planet relative to the star.
- **psf_scaling** (*float*) – Additional scaling factor of the planet flux (e.g., to correct for a neutral density filter). Should be negative in order to inject negative fake planets.
- **merit** (*str*) – Figure of merit for the minimization. Can be 'hessian', to minimize the sum of the absolute values of the determinant of the Hessian matrix, or 'poisson', to minimize the sum of the absolute pixel values, assuming a Poisson distribution for the noise (Wertz et al. 2017), or 'gaussian', to minimize the ratio of the squared pixel values and the variance of the pixels within an annulus but excluding the aperture area.
- **aperture** (*float*) – Aperture radius (arcsec) at the position specified at *position*.
- **sigma** (*float*) – Standard deviation (arcsec) of the Gaussian kernel which is used to smooth the images before the figure of merit is calculated (in order to reduce small pixel-to-pixel variations).
- **tolerance** (*float*) – Absolute error on the input parameters, position (pix) and contrast (mag), that is used as acceptance level for convergence. Note that only a single value can be specified which is used for both the position and flux so tolerance=0.1 will give a precision of 0.1 mag and 0.1 pix. The tolerance on the output (i.e., the chi-square value) is set to np.inf so the condition is always met.
- **pca_number** (*int*) – Number of principal components used for the PSF subtraction.
- **cent_size** (*float*) – Radius of the central mask (arcsec). No mask is used when set to None.
- **edge_size** (*float*) – Outer radius (arcsec) beyond which pixels are masked. No outer mask is used when set to None. The radius will be set to half the image size if the argument is larger than half the image size.
- **extra_rot** (*float*) – Additional rotation angle of the images in clockwise direction (deg).
- **residuals** (*str*) – Method for combining the residuals ('mean', 'median', 'weighted', or 'clipped').
- **reference_in_tag** (*str, None*) – Tag of the database entry with the reference images that are read as input. The data of the `image_in_tag` itself is used as reference data for the PSF subtraction if set to None. Note that the mean is not subtracted from the data of `image_in_tag` and `reference_in_tag` in case the `reference_in_tag` is used, to allow for flux and position measurements in the context of RDI.

Returns None

Return type NoneType

run() → None

Run method of the module. The position and contrast of a planet is measured by injecting negative copies of the PSF template and applying a simplex method (Nelder-Mead) for minimization of a figure of merit at the planet location.

Returns None

Return type NoneType

pynpoint.processing.frameselection module

Pipeline modules for frame selection.

class pynpoint.processing.frameselection.**FrameSelectionModule**(*name_in: str, image_in_tag: str, selected_out_tag: str, removed_out_tag: str, index_out_tag: str = None, method='median', threshold: float = 4.0, fwhm: float = 0.1, aperture: Union[Tuple[str, float], Tuple[str, float, float]] = ('circular', 0.2), position: Union[Tuple[int, int, float], Tuple[None, None, float]] = None*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for frame selection.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **selected_out_tag** (*str*) – Tag of the database entry with the selected images that are written as output. Should be different from *image_in_tag*. No data is written when set to None.
- **removed_out_tag** (*str*) – Tag of the database entry with the removed images that are written as output. Should be different from *image_in_tag*. No data is written when set to None.
- **index_out_tag** (*str, None*) – Tag of the database entry with the list of frames indices that are removed with the frames selection. No data is written when set to *None*.
- **method** (*str*) – Perform the sigma clipping with respect to the median or maximum aperture flux by setting the *method* to 'median' or 'max'.
- **threshold** (*float*) – Threshold in units of sigma for the frame selection. All images that are a *threshold* number of sigmas away from the median photometry will be removed.
- **fwhm** (*float, None*) – The full width at half maximum (FWHM) of the Gaussian kernel (arcsec) that is used to smooth the images before the brightest pixel is located. Should be similar in size to the FWHM of the stellar PSF. A fixed position, specified by *position*, is used when *fwhm* is set to None.
- **aperture** (*tuple(str, float, float)*) – Tuple with the aperture properties for measuring the photometry around the location of the brightest pixel. The first element contains the aperture type ('circular', 'annulus', or 'ratio'). For a circular aperture, the second

element contains the aperture radius (arcsec). For the other two types, the second and third element are the inner and outer radii (arcsec) of the aperture. The position of the aperture has to be specified with *position* when *fwhm* is set to None.

- **position** (*tuple(int, int, float), None*) – Subframe that is selected to search for the star. The tuple contains the center (pix) and size (arcsec) (pos_x, pos_y, size). Setting *position* to None will use the full image to search for the star. If *position=(None, None, size)* then the center of the image will be used.

Returns None

Return type NoneType

run() → None
Run method of the module. Smooths the images with a Gaussian kernel, locates the brightest pixel in each image, measures the integrated flux around the brightest pixel, calculates the median and standard deviation of the photometry, and applies sigma clipping to remove low quality images.

Returns None

Return type NoneType

class pynpoint.processing.frameselection.**FrameSimilarityModule**(*name_in: str, image_tag: str, method: str = 'MSE', mask_radius: Tuple[float, float] = (0.0, 5.0), window_size: float = 0.1, temporal_median: str = 'full'*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for measuring the similarity between frames.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_tag** (*str*) – Tag of the database entry that is read as input.
- **method** (*str*) – Method for the similarity measure. There are three measures available:
  - *MSE* - Mean Squared Error
  - *PCC* - Pearson Correlation Coefficient
  - *SSIM* - Structural Similarity

  These measures compare each image to the temporal median of the image set.
- **mask_radius** (*tuple(float, float)*) – Inner and outer radius (arcsec) of the mask that is applied to the images.
- **window_size** (*float*) – Size (arcsec) of the sliding window that is used when the SSIM similarity is calculated.
- **temporal_median** (*str*) – Option to calculate the temporal median for each position ('full') or as a constant value ('constant') for the entire set. The latter is computationally less expensive.

Returns None

Return type NoneType

run() → None
Run method of the module. Computes the similarity between frames based on the Mean Squared Error (MSE), the Pearson Correlation Coefficient (PCC), or the Structural Similarity (SSIM).

Returns None

Return type NoneType

class pynpoint.processing.frameselection.**ImageStatisticsModule**(*name_in: str, image_in_tag: str, stat_out_tag: str, position: Union[Tuple[int, int, float], Tuple[None, None, float]] = None*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for calculating image statistics for the full images or a subsection of the images.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry with the images that are read as input.
- **stat_out_tag** (*str*) – Tag of the database entry with the statistical results that are written as output. The result is stored in the following order: minimum, maximum, sum, mean, median, and standard deviation.
- **position** (*tuple(int, int, float)*) – Position (x, y) (pix) and radius (arcsec) of the circular area in which the statistics are calculated. The full image is used if set to None.

Returns None

Return type NoneType

run() → None
Run method of the module. Calculates the minimum, maximum, sum, mean, median, and standard deviation of the pixel values of each image separately. NaNs are ignored for each calculation. The values are calculated for either the full images or a circular subsection of the images.

Returns None

Return type NoneType

class pynpoint.processing.frameselection.**RemoveFramesModule**(*name_in: str, image_in_tag: str, selected_out_tag: str, removed_out_tag: str, frames: Union[str, range, list, numpy.ndarray]*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for removing images by their index number.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **selected_out_tag** (*str*) – Tag of the database entry with the remaining images after removing the specified images. Should be different from *image_in_tag*. No data is written when set to *None*.
- **removed_out_tag** (*str*) – Tag of the database entry with the images that are removed. Should be different from *image_in_tag*. No data is written when set to *None*.
- **frames** (*str, list, range, numpy.ndarray*) – A tuple or array with the frame indices that have to be removed or a database tag pointing to a list of frame indices.

Returns None

Return type NoneType

run() → None
Run method of the module. Removes the frames and corresponding attributes, updates the NFRAMES attribute, and saves the data and attributes.

Returns None

Return type NoneType

class pynpoint.processing.frameselection.**RemoveLastFrameModule**(*name_in: str, image_in_tag: str, image_out_tag: str*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for removing every NDIT+1 frame from NACO data obtained in cube mode. This frame contains the average pixel values of the cube.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output. Should be different from *image_in_tag*.

Returns None

Return type NoneType

run() → None
Run method of the module. Removes every NDIT+1 frame and saves the data and attributes.

Returns None

Return type NoneType

class pynpoint.processing.frameselection.**RemoveStartFramesModule**(*name_in: str, image_in_tag: str, image_out_tag: str, frames: int = 1*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for removing a fixed number of images at the beginning of each cube. This can be useful for NACO data in which the background is significantly higher in the first several frames of a data cube.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output. Should be different from *image_in_tag*.
- **frames** (*int*) – Number of frames that are removed at the beginning of each cube.

Returns None

Return type NoneType

run() → None
Run method of the module. Removes a constant number of images at the beginning of each cube and saves the data and attributes.

Returns None

Return type NoneType

class pynpoint.processing.frameselection.**SelectByAttributeModule**(*name_in: str*, *image_in_tag: str*, *selected_out_tag: str*, *removed_out_tag: str*, *attribute_tag: str*, *number_frames: int = 100*, *order: str = 'descending'*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for selecting frames based on attribute values.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_tag** (*str*) – Tag of the database entry that is read as input.
- **selected_out_tag** (*str*) – Tag of the database entry to which the selected frames are written.
- **removed_out_tag** (*str*) – Tag of the database entry to which the removed frames are written.
- **attribute_tag** (*str*) – Name of the attribute which is used to sort and select the frames.
- **number_frames** (*int*) – Number of frames that are selected.
- **order** (*str*) – Order in which the frames are selected. Can be either 'descending' (will select the lowest attribute values) or 'ascending' (will select the highest attribute values).

Returns None

Return type NoneType

Examples

The example below selects the first 100 frames with an ascending order of the `INDEX` values that are stored to the 'im_arr' dataset:

```
SelectByAttributeModule(name_in='frame_selection',
                        image_in_tag='im_arr',
                        attribute_tag='INDEX',
                        selected_frames=100,
                        order='ascending',
                        selected_out_tag='im_arr_selected',
                        removed_out_tag='im_arr_removed'))
```

The example below selects the 200 frames with the largest `SSIM` values that are stored to the 'im_arr' dataset:

```
SelectByAttributeModule(name_in='frame_selection',
                        image_in_tag='im_arr',
                        attribute_tag='SSIM',
                        selected_frames=200,
                        order='descending',
                        selected_out_tag='im_arr_selected',
                        removed_out_tag='im_arr_removed'))
```

run() → None
Run method of the module. Selects images according to a specified attribute tag and ordering, e.g. the highest 150 `INDEX` frames, or the lowest 50 `PCC` frames.

Returns None

Return type NoneType

pynpoint.processing.limits module

Pipeline modules for estimating detection limits.

class pynpoint.processing.limits.**ContrastCurveModule**(*name_in: str*, *image_in_tag: str*, *psf_in_tag: str*, *contrast_out_tag: str*, *separation: Tuple[float, float, float] = (0.1, 1.0, 0.01)*, *angle: Tuple[float, float, float] = (0.0, 360.0, 60.0)*, *threshold: Tuple[str, float] = ('sigma', 5.0)*, *psf_scaling: float = 1.0*, *aperture: float = 0.05*, *pca_number: int = 20*, *cent_size: float = None*, *edge_size: float = None*, *extra_rot: float = 0.0*, *residuals: str = 'median'*, *snr_inject: float = 100.0*, ***kwargs*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module to calculate contrast limits for a given sigma level or false positive fraction, with a correction for small sample statistics. Positions are processed in parallel if `CPU` is set to a value larger than 1 in the configuration file.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that contains the stack with images.
- **psf_in_tag** (*str*) – Tag of the database entry that contains the reference PSF that is used as fake planet. Can be either a single image or a stack of images equal in size to *image_in_tag*.
- **contrast_out_tag** (*str*) – Tag of the database entry that contains the separation, azimuthally averaged contrast limits, the azimuthal variance of the contrast limits, and the threshold of the false positive fraction associated with sigma.
- **separation** (*tuple(float, float, float)*) – Range of separations (arcsec) where the contrast is calculated. Should be specified as (lower limit, upper limit, step size). Apertures that fall within the mask radius or beyond the image size are removed.
- **angle** (*tuple(float, float, float)*) – Range of position angles (deg) where the contrast is calculated. Should be specified as (lower limit, upper limit, step size), measured counterclockwise with respect to the vertical image axis, i.e. East of North.
- **threshold** (*tuple(str, float)*) – Detection threshold for the contrast curve, either in terms of 'sigma' or the false positive fraction (FPF). The value is a tuple, for example provided as ('sigma', 5.) or ('fpf', 1e-6). Note that when sigma is fixed, the false positive fraction will change with separation. Also, sigma only corresponds to the standard deviation of a normal distribution at large separations (i.e., large number of samples).
- **psf_scaling** (*float*) – Additional scaling factor of the planet flux (e.g., to correct for a neutral density filter). Should have a positive value.
- **aperture** (*float*) – Aperture radius (arcsec).
- **pca_number** (*int*) – Number of principal components used for the PSF subtraction.
- **cent_size** (*float, None*) – Central mask radius (arcsec). No mask is used when set to None.
- **edge_size** (*float, None*) – Outer edge radius (arcsec) beyond which pixels are masked. No outer mask is used when set to None. If the value is larger than half the image size then it will be set to half the image size.
- **extra_rot** (*float*) – Additional rotation angle of the images in clockwise direction (deg).
- **residuals** (*str*) – Method used for combining the residuals ('mean', 'median', 'weighted', or 'clipped').
- **snr_inject** (*float*) – Signal-to-noise ratio of the injected planet signal that is used to measure the amount of self-subtraction.

Returns None

Return type NoneType

run() → None

Run method of the module. An artificial planet is injected (based on the noise level) at a given separation and position angle. The amount of self-subtraction is then determined and the contrast limit is calculated for a given sigma level or false positive fraction. A correction for small sample statistics is applied for both

cases. Note that if the sigma level is fixed, the false positive fraction changes with separation, following the Student's t-distribution (see Mawet et al. 2014 for details).

Returns None

Return type NoneType

class pynpoint.processing.limits.**MassLimitsModule**(*name_in: str*, *contrast_in_tag: str*, *mass_out_tag: str*, *model_file: str*, *star_prop: dict*, *instr_filter: str = "L'"*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module to calculate mass limits from the contrast limits and any isochrones model grid downloaded from https://phoenix.ens-lyon.fr/Grids/.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **contrast_in_tag** (*str*) – Tag of the database entry that contains the contrast curve data, as computed with the *ContrastCurveModule*.
- **mass_out_tag** (*str*) – Tag of the database entry with the output data containing the separation, the mass limits, and the upper and lower one sigma deviation as calculated for the azimuthal variance on the contrast limits.
- **model_file** (*str*) – Path to the file containing the model data. Must be in the same format as the grids found on https://phoenix.ens-lyon.fr/Grids/. Any of the isochrones files from this website can be used.
- **star_prop** (*dict*) – Dictionary containing host star properties. Must have the following keys:
  - `magnitude` - Apparent magnitude, in the same band as the *instr_filter*.
  - `distance` - Distance in parsec.
  - `age` - Age of the system in the Myr.
- **instr_filter** (*str*) – Instrument filter in the same format as listed in the *model_file*.

Returns None

Return type NoneType

static interpolate_model(*age_eval: numpy.ndarray, mag_eval: numpy.ndarray, filter_index: int, model_age: List[float], model_data: List[numpy.ndarray]*) → numpy.ndarray

Interpolates the grid based model data.

Parameters

- **age_eval** (*numpy.ndarray*) – Age at which the system is evaluated. Must be of the same shape as *mag_eval*.
- **mag_eval** (*numpy.ndarray*) – Absolute magnitude for which the system is evaluated. Must be of the same shape as *age_eval*.
- **filter_index** (*int*) – Column index where the filter is located.
- **model_age** (*list(float,)*) – List of ages which are given by the model.
- **model_data** (*list(numpy.ndarray,)*) – List of arrays containing the model data.

Returns griddata – Interpolated values for the given evaluation points (age_eval, mag_eval). Has the same shape as age_eval and mag_eval.

Return type numpy.ndarray

static read_model(*model_file_path: str*) → Tuple[List[float], List[numpy.ndarray], List[str]]
Reads the data from the model file and structures it. Returns an array of available model ages and a list of model data for each age.

Parameters model_file (*str*) – Path to the file containing the model data.

Returns

- *list(float,)* – List with all the ages from the model grid.
- *list(numpy.ndarray,)* – List with all the isochrone data, so the length is the same as the number of ages.
- *list(str,)* – List with all the column names from the model grid.

run() → None
Run method of the module. Calculates the mass limits from the contrast limits (as calculated with the *ContrastCurveModule*) and the isochrones of an evolutionary model. The age and the absolute magnitude of the isochrones are linearly interpolated such that the mass limits can be calculated for a given contrast limits (which is converted in an absolute magnitude with the apparent magnitude and distance of the central star).

Returns None

Return type NoneType

pynpoint.processing.pcabackground module

Pipeline modules for PCA-based background subtraction.

class pynpoint.processing.pcabackground.**DitheringBackgroundModule**(*name_in: str, image_in_tag: str, image_out_tag: str, center: tuple = None, cubes: int = None, size: float = 2.0, gaussian: float = 0.15, subframe: float = None, pca_number: int = 60, mask_star: float = 0.7, subtract_mean: bool = False, **kwargs*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for PCA-based background subtraction of data with dithering. This is a wrapper that applies the processing modules required for the PCA background subtraction.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output. Not written if set to None.
- **center** (*tuple(tuple(int, int),), None*) – Tuple with the centers of the dithering positions, e.g. ((x0,y0), (x1,y1)). The order of the coordinates should correspond to the order in which the star is present. If *center* and *cubes* are both set to None then sorting and subtracting of the background frames is based on DITHER_X and DITHER_Y. If *center* is specified and *cubes* is set to None then the DITHER_X and DITHER_Y attributes will be used for sorting and subtracting of the background but not for selecting the dither positions.
- **cubes** (*int, None*) – Number of consecutive cubes per dither position. If *cubes* is set to None then sorting and subtracting of the background frames is based on DITHER_X and DITHER_Y.
- **size** (*float*) – Image size (arsec) that is cropped at the specified dither positions.
- **gaussian** (*float*) – Full width at half maximum (arcsec) of the Gaussian kernel that is used to smooth the image before the star is located.
- **subframe** (*float, None*) – Size (arcsec) of the subframe that is used to search for the star. Cropping of the subframe is done around the center of the dithering position. If set to None then the full frame size (*size*) will be used.

- **pca_number** (*int*) – Number of principal components.
- **mask_star** (*float*) – Radius of the central mask (arcsec).
- **subtract_mean** (*bool*) – The mean of the background images is subtracted from both the star and background images before the PCA basis is constructed.

Keyword Arguments

- **crop** (*bool*) – Skip the step of selecting and cropping of the dithering positions if set to False.
- **prepare** (*bool*) – Skip the step of preparing the PCA background subtraction if set to False.
- **pca_background** (*bool*) – Skip the step of the PCA background subtraction if set to False.
- **combine** (*str*) – Combine the mean background subtracted ('mean') or PCA background subtracted ('pca') frames. This step is ignored if set to None.

Returns None

Return type NoneType

run() → None

Run method of the module. Cuts out the detector sections at the different dither positions, prepares the PCA background subtraction, locates the star in each image, runs the PCA background subtraction, combines the output from the different dither positions is written to a single database tag.

Returns None

Return type NoneType

class pynpoint.processing.pcabackground.**PCABackgroundPreparationModule**(*name_in: str, image_in_tag: str, star_out_tag: str, subtracted_out_tag: str, background_out_tag: str, dither: Union[Tuple[int, int, int], Tuple[int, None, Tuple[float, float]]], combine: str = 'mean', **kwargs*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for preparing the PCA background subtraction.

Parameters

- **name_in** (*str*) – Unique name of the pipeline module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **star_out_tag** (*str*) – Output tag with the images containing the star.
- **subtracted_out_tag** (*str*) – Output tag with the mean/median background subtracted images with the star.
- **background_out_tag** (*str*) – Output tag with the images containing only background emission.
- **dither** (*tuple(int, int, int), tuple(int, None, tuple(float, float))*) – Tuple with the parameters for separating the star and background frames. The tuple should contain three values (positions, cubes, first) with *positions* the number of unique dithering position, *cubes* the number of consecutive cubes per dithering position, and *first* the index value of the first cube which contains the star (Python indexing starts at zero). Sorting is based on the DITHER_X and DITHER_Y attributes when *cubes* is set to None. In that case, the *first* value should be a tuple with the DITHER_X and DITHER_Y values in which the star appears first.
- **combine** (*str*) – Method to combine the background images ('mean' or 'median').

Returns None

Return type NoneType

run() → None
Run method of the module. Separates the star and background frames, subtracts the mean or median background from both the star and background frames, and writes the star and background frames separately.

Returns None

Return type NoneType

class pynpoint.processing.pcabackground.**PCABackgroundSubtractionModule**(*name_in: str*, *star_in_tag: str*, *background_in_tag: str*, *residuals_out_tag: str*, *fit_out_tag: str = None*, *mask_out_tag: str = None*, *pca_number: int = 60*, *mask_star: float = 0.7*, *subtract_mean: bool = False*, *subframe: float = None*, *gaussian: float = 0.15*, ***kwargs*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for PCA based background subtraction. See Hunziker et al. 2018 for details.

Parameters

- **name_in** (*str*) – Tag of the database entry with the star images.
- **star_in_tag** (*str*) – Tag of the database entry with the star images.
- **background_in_tag** (*str*) – Tag of the database entry with the background images.
- **residuals_out_tag** (*str*) – Tag of the database entry with the residuals of the star images after the background subtraction.
- **fit_out_tag** (*str, None*) – Tag of the database entry with the fitted background. No data is written when set to None.
- **mask_out_tag** (*str, None*) – Tag of the database entry with the mask. No data is written when set to None.
- **pca_number** (*int*) – Number of principal components.
- **mask_star** (*float*) – Radius of the central mask (arcsec).
- **subtract_mean** (*bool*) – The mean of the background images is subtracted from both the star and background images before the PCA basis is constructed.
- **gaussian** (*float*) – Full width at half maximum (arcsec) of the Gaussian kernel that is used to smooth the image before the star is located.
- **subframe** (*float, None*) – Size (arcsec) of the subframe that is used to find the star. Cropping of the subframe is done around the center of the image. The full images is used if set to None.

Returns None

Return type NoneType

run() → None
Run method of the module. Creates a PCA basis set of the background frames, masks the PSF in the star frames and optionally an off-axis point source, fits the star frames with a linear combination of the principal components, and writes the residuals of the background subtracted images.

Returns None

Return type NoneType

pynpoint.processing.psfpreparation module

Pipeline modules to prepare the data for the PSF subtraction.

class pynpoint.processing.psfpreparation.**AngleCalculationModule**(*name_in: str*, *data_tag: str*, *instrument: str = 'NACO'*)

Bases: *pynpoint.core.processing.ProcessingModule*

Module for calculating the parallactic angles. The start time of the observation is taken and multiples of the exposure time are added to derive the parallactic angle of each frame inside the cube. Instrument specific overheads are included.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **data_tag** (*str*) – Tag of the database entry for which the parallactic angles are written as attributes.

- **instrument** (*str*) – Instrument name ('NACO', 'SPHERE/IRDIS', or 'SPHERE/IFS').

Returns None

Return type NoneType

run() → None
Run method of the module. Calculates the parallactic angles from the position of the object on the sky and the telescope location on earth. The start of the observation is used to extrapolate for the observation time of each individual image of a data cube. The values are written as PARANG attributes to *data_tag*.

Returns None

Return type NoneType

class pynpoint.processing.psfpreparation.**AngleInterpolationModule**(*name_in: str*, *data_tag: str*)

Bases: *pynpoint.core.processing.ProcessingModule*

Module for calculating the parallactic angle values by interpolating between the begin and end value of a data cube.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **data_tag** (*str*) – Tag of the database entry for which the parallactic angles are written as attributes.

Returns None

Return type NoneType

run() → None
Run method of the module. Calculates the parallactic angles of each frame by linearly interpolating between the start and end values of the data cubes. The values are written as attributes to *data_tag*. A correction of 360 deg is applied when the start and end values of the angles change sign at +/-180 deg.

Returns None

Return type NoneType

class pynpoint.processing.psfpreparation.**PSFpreparationModule**(*name_in: str*, *image_in_tag: str*, *image_out_tag: str*, *mask_out_tag: str = None*, *norm: bool = False*, *resize: float = None*, *cent_size: float = None*, *edge_size: float = None*)

Bases: *pynpoint.core.processing.ProcessingModule*

Module to prepare the data for PSF subtraction with PCA. The preparation steps include masking and image normalization.

Parameters

- **name_in** (*str*) – Unique name of the module instance.

- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry with images that is written as output.
- **mask_out_tag** (*str, None, optional*) – Tag of the database entry with the mask that is written as output. If set to None, no mask array is saved.
- **norm** (*bool*) – Normalize each image by its Frobenius norm.
- **resize** (*float, None*) – DEPRECATED. This parameter is currently ignored by the module and will be removed in a future version of PynPoint.
- **cent_size** (*float, None, optional*) – Radius of the central mask (in arcsec). No mask is used when set to None.
- **edge_size** (*float, None, optional*) – Outer radius (in arcsec) beyond which pixels are masked. No outer mask is used when set to None. If the value is larger than half the image size then it will be set to half the image size.

Returns None

Return type NoneType

run() → None
Run method of the module. Masks and normalizes the images.

Returns None

Return type NoneType

class pynpoint.processing.psfpreparation.**SDIpreparationModule**(*name_in: str, image_in_tag: str, image_out_tag: str, wavelength: Tuple[float, float], width: Tuple[float, float]*)

Bases: *pynpoint.core.processing.ProcessingModule*

Module for preparing continuum frames for SDI subtraction.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output. Should be different from *image_in_tag*.
- **wavelength** (*tuple(float, float)*) – The central wavelengths of the line and continuum filter, (line, continuum), in arbitrary but identical units.
- **width** (*tuple(float, float)*) – The equivalent widths of the line and continuum filter, (line, continuum), in arbitrary but identical units.

Returns None

Return type NoneType

run() → None
Run method of the module. Normalizes the images for the different filter widths, upscales the images, and crops the images to the initial image shape in order to align the PSF patterns.

Returns None

Return type NoneType

class pynpoint.processing.psfpreparation.**SortParangModule**(*name_in: str, image_in_tag: str, image_out_tag: str*)

Bases: *pynpoint.core.processing.ProcessingModule*

Module to sort the images and non-static attributes with increasing INDEX.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry with images that is written as output. Should be different from *image_in_tag*.

Returns None

Return type NoneType

run() → None

Run method of the module. Sorts the images and relevant non-static attributes.

Returns None

Return type NoneType

pynpoint.processing.psfsubtraction module

Pipeline modules for PSF subtraction.

class pynpoint.processing.psfsubtraction.**ClassicalADIModule**(*name_in: str, image_in_tag: str, res_out_tag: str, stack_out_tag: str, threshold: Optional[Tuple[float, float, float]], nreference: int = None, residuals: str = 'median', extra_rot: float = 0.0*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for PSF subtraction with classical ADI by subtracting a median-combined reference image. A rotation threshold can be set for a fixed separation to prevent self-subtraction.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry with the science images that are read as input.
- **res_out_tag** (*str*) – Tag of the database entry with the residuals of the PSF subtraction that are written as output.
- **stack_out_tag** (*str*) – Tag of the database entry with the stacked residuals that are written as output.

- **threshold** (*tuple(float, float, float), None*) – Tuple with the separation for which the angle threshold is optimized (arcsec), FWHM of the PSF (arcsec), and the threshold (FWHM) for the selection of the reference images. No threshold is used if set to None.
- **nreference** (*int, None*) – Number of reference images, closest in line to the science image. All images are used if *threshold* is None or *nreference* is None.
- **residuals** (*str*) – Method used for combining the residuals ('mean', 'median', 'weighted', or 'clipped').
- **extra_rot** (*float*) – Additional rotation angle of the images (deg).

Returns None

Return type NoneType

run() → None

Run method of the module. Selects for each image the reference images closest in line while taking into account a rotation threshold for a fixed separation, median-combines the references images, and subtracts the reference image from each image separately. Alternatively, a single, median-combined reference image can be created and subtracted from all images. All images are used if the rotation condition can not be met. Both the individual residuals (before derotation) and the stacked residuals are stored.

Returns None

Return type NoneType

class pynpoint.processing.psfsubtraction.**PcaPsfSubtractionModule**(*name_in: str, images_in_tag: str, reference_in_tag: str, res_mean_tag: str = None, res_median_tag: str = None, res_weighted_tag: str = None, res_rot_mean_clip_tag: str = None, res_arr_out_tag: str = None, basis_out_tag: str = None, pca_numbers: Union[range, List[int], numpy.ndarray] = range(1, 21), extra_rot: float = 0.0, subtract_mean: bool = True*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for PSF subtraction with principal component analysis (PCA). The residuals are calculated in parallel for the selected numbers of principal components. This may require a large amount of memory in case the stack of input images is very large. The number of processes can be set with the CPU keyword in the configuration file.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **images_in_tag** (*str*) – Tag of the database entry with the science images that are read as input
- **reference_in_tag** (*str*) – Tag of the database entry with the reference images that are read as input.
- **res_mean_tag** (*str, None*) – Tag of the database entry with the mean collapsed residuals. Not calculated if set to None.
- **res_median_tag** (*str, None*) – Tag of the database entry with the median collapsed residuals. Not calculated if set to None.
- **res_weighted_tag** (*str, None*) – Tag of the database entry with the noise-weighted residuals (see Bottom et al. 2017). Not calculated if set to None.
- **res_rot_mean_clip_tag** (*str, None*) – Tag of the database entry of the clipped mean residuals. Not calculated if set to None.
- **res_arr_out_tag** (*str, None*) – Tag of the database entry with the image residuals from the PSF subtraction. If a list of PCs is provided in *pca_numbers* then multiple tags will be created in the central database. Not calculated if set to None. Not supported with multiprocessing.
- **basis_out_tag** (*str, None*) – Tag of the database entry with the basis set. Not stored if set to None.
- **pca_numbers** (*range, list(int,), numpy.ndarray*) – Number of principal components used for the PSF model. Can be a single value or a tuple with integers.
- **extra_rot** (*float*) – Additional rotation angle of the images (deg).
- **subtract_mean** (*bool*) – The mean of the science and reference images is subtracted from the corresponding stack, before the PCA basis is constructed and fitted.

Returns None

Return type NoneType

run() → None

Run method of the module. Subtracts the mean of the image stack from all images, reshapes the stack of images into a 2D array, uses singular value decomposition to construct the orthogonal basis set, calculates the PCA coefficients for each image, subtracts the PSF model, and writes the residuals as output.

Returns None

Return type NoneType

pynpoint.processing.resizing module

Pipeline modules for resizing of images.

class pynpoint.processing.resizing.**AddLinesModule**(*name_in: str, image_in_tag: str, image_out_tag: str, lines: Tuple[int, int, int, int]*)

Bases: *pynpoint.core.processing.ProcessingModule*

Module to add lines of pixels to increase the size of an image.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output, including the images with increased size. Should be different from *image_in_tag*.
- **lines** (*tuple(int, int, int, int)*) – The number of lines that are added in left, right, bottom, and top direction.

Returns None

Return type NoneType

run() → None

Run method of the module. Adds lines of zero-value pixels to increase the size of an image.

Returns None

Return type NoneType

class pynpoint.processing.resizing.**CropImagesModule**(*name_in: str, image_in_tag: str, image_out_tag: str, size: float, center: Optional[Tuple[int, int]]*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for cropping of images around a given position.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output. Should be different from *image_in_tag*.
- **size** (*float*) – New image size (arcsec). The same size will be used for both image dimensions.
- **center** (*tuple(int, int), None*) – Tuple (x0, y0) with the new image center. Python indexing starts at 0. The center of the input images will be used when *center* is set to *None*. Note that if the image is even-sized, it is not possible to a uniquely define a pixel position in the center of the image. The image center is determined (with pixel precision) with the *center_pixel()* function.

Returns None

Return type NoneType

run() → None

Run method of the module. Decreases the image size by cropping around an given position. The module always returns odd-sized images.

Returns None

Return type NoneType

class pynpoint.processing.resizing.**RemoveLinesModule**(*name_in: str, image_in_tag: str, image_out_tag: str, lines: Tuple[int, int, int, int]*)

Bases: *pynpoint.core.processing.ProcessingModule*

Module to decrease the dimensions of an image by removing lines of pixels.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output, including the images with decreased size. Should be different from *image_in_tag*.
- **lines** (*tuple(int, int, int, int)*) – The number of lines that are removed in left, right, bottom, and top direction.

Returns None

Return type NoneType

run() → None

Run method of the module. Removes the lines given by *lines* from each frame.

Returns None

Return type NoneType

class pynpoint.processing.resizing.**ScaleImagesModule**(*name_in: str, image_in_tag: str, image_out_tag: str, scaling: Union[Tuple[float, float, float], Tuple[None, None, float], Tuple[float, float, None]], pixscale: bool = False*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for rescaling of an image.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output. Should be different from *image_in_tag*.
- **scaling** (*tuple(float, float, float)*) – Tuple with the scaling factors for the image size and flux, (scaling_x, scaling_y, scaling_flux). Upsampling and downsampling of the image corresponds to `scaling_x/y` > 1 and 0 < `scaling_x/y` < 1, respectively.
- **pixscale** (*bool*) – Adjust the pixel scale by the average scaling in x and y direction.

Returns None

Return type NoneType

run() → None

Run method of the module. Rescales an image with a fifth order spline interpolation and a reflecting boundary condition.

Returns None

Return type NoneType

pynpoint.processing.stacksubset module

Pipeline modules for stacking and subsampling of images.

class pynpoint.processing.stacksubset.**CombineTagsModule**(*name_in: str, image_in_tags: List[str], image_out_tag: str, check_attr: bool = True, index_init: bool = False*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for combining tags from multiple database entries into a single tag.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tags** (*list(str,)*) – Tags of the database entries that are read as input and combined.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output. Should not be present in *image_in_tags*.
- **check_attr** (*bool*) – Compare non-static attributes between the tags or combine all non-static attributes into the new database tag.
- **index_init** (*bool*) – Reinitialize the `INDEX` attribute. The frames are indexed in the order of tags names that are provided in *image_in_tags*.

Returns None

Return type NoneType

run() → None

Run method of the module. Combines the frames of multiple tags into a single dataset and adds the static and non-static attributes. The values of the attributes are compared between the input tags to make sure that the input tags descent from the same data set.

Returns None

Return type NoneType

class pynpoint.processing.stacksubset.**DerotateAndStackModule**(*name_in: str, image_in_tag: str, image_out_tag: str, derotate: bool = True, stack: str = None, extra_rot: float = 0.0*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for derotating and/or stacking (i.e., taking the median or average) of the images.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output. The output is either 2D (*stack=False*) or 3D (*stack=True*).

- **derotate** (*bool*) – Derotate the images with the PARANG attribute.
- **stack** (*str*) – Type of stacking applied after optional derotation ('mean', 'median', or None for no stacking).
- **extra_rot** (*float*) – Additional rotation angle of the images in clockwise direction (deg).

Returns None

Return type NoneType

run() → None
Run method of the module. Uses the PARANG attributes to derotate the images (if *derotate* is set to True) and applies an optional mean or median stacking afterwards.

Returns None

Return type NoneType

class pynpoint.processing.stacksubset.**MeanCubeModule**(*name_in: str*, *image_in_tag: str*, *image_out_tag: str*)
Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for calculating the mean of each individual cube associated with a database tag.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry with the mean collapsed images that are written as output. Should be different from *image_in_tag*.

Returns None

Return type NoneType

run() → None
Run method of the module. Uses the NFRAMES attribute to select the images of each cube, calculates the mean of each cube, and saves the data and attributes.

Returns None

Return type NoneType

class pynpoint.processing.stacksubset.**StackAndSubsetModule**(*name_in: str*, *image_in_tag: str*, *image_out_tag: str*, *random: int = None*, *stacking: int = None*, *combine: str = 'mean'*)
Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for stacking subsets of images and/or selecting a random sample of images.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output. Should be different from *image_in_tag*.

- **random** (*int*) – Number of random images. All images are used if set to None.
- **stacking** (*int*) – Number of stacked images per subset. No stacking is applied if set to None.
- **combine** (*str*) – Method for combining images ('mean' or 'median'). The angles are always mean-combined.

Returns None

Return type NoneType

run() → None
Run method of the module. Stacks subsets of images and/or selects a random subset. Also the parallactic angles are mean-combined if images are stacked.

Returns None

Return type NoneType

class pynpoint.processing.stacksubset.**StackCubesModule**(*name_in: str*, *image_in_tag: str*, *image_out_tag: str*, *combine: str = 'mean'*)
Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for calculating the mean or median of each original data cube associated with a database tag.

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry with the mean or median collapsed images that are written as output. Should be different from *image_in_tag*.
- **combine** (*str*) – Method to combine the images ('mean' or 'median').

Returns None

Return type NoneType

run() → None
Run method of the module. Uses the NFRAMES attribute to select the images of each cube, calculates the mean or median of each cube, and saves the data and attributes.

Returns None

Return type NoneType

pynpoint.processing.timedenoising module

Continuous wavelet transform (CWT) and discrete wavelet transform (DWT) denoising for speckle suppression in the time domain. The module can be used as additional preprocessing step. See Bonse et al. 2018 more information.

class pynpoint.processing.timedenoising.**CwtWaveletConfiguration**(*wavelet: str = 'dog'*, *wavelet_order: int = 2*, *keep_mean: bool = False*, *resolution: float = 0.5*)

Bases: object

Configuration capsule for a CWT based time denoising. Standard configuration as in the original paper.

Parameters

- **wavelet** (*str*) – Wavelet.
- **wavelet_order** (*int*) – Wavelet order.
- **keep_mean** (*bool*) – Keep mean.
- **resolution** (*float*) – Resolution.

Returns None

Return type NoneType

class pynpoint.processing.timedenoising.**DwtWaveletConfiguration**(*wavelet: str = 'db8'*)

Bases: object

Configuration capsule for a DWT based time denoising. A cheap alternative of the CWT based wavelet denoising. However, the supported wavelets should perform worse compared to the CWT DOG wavelet.

Parameters **wavelet** (*str*) – Wavelet.

Returns None

Return type NoneType

class pynpoint.processing.timedenoising.**TimeNormalizationModule**(*name_in: str*, *image_in_tag: str*, *image_out_tag: str*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for normalization of global brightness variations of the detector (see Bonse et al. 2018).

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output.

Returns None

Return type NoneType

run() → None

Run method of the module.

Returns None

Return type NoneType

class pynpoint.processing.timedenoising.**WaveletTimeDenoisingModule**(*name_in: str, image_in_tag: str, image_out_tag: str, wavelet_configuration: Union[pynpoint.processing.timeden pynpoint.processing.timedenoising.Dw padding: str = 'zero', median_filter: bool = False, threshold_function: str = 'soft'*)

Bases: *pynpoint.core.processing.ProcessingModule*

Pipeline module for speckle subtraction in the time domain by using CWT or DWT wavelet shrinkage (see Bonse et al. 2018).

Parameters

- **name_in** (*str*) – Unique name of the module instance.
- **image_in_tag** (*str*) – Tag of the database entry that is read as input.
- **image_out_tag** (*str*) – Tag of the database entry that is written as output.
- **wavelet_configuration** (pynpoint.processing.timedenoising.CwtWaveletConfiguration *or* pynpoint.processing.timedenoising.DwtWaveletConfiguration) – Instance of DwtWaveletConfiguration or CwtWaveletConfiguration which gives the parameters of the wavelet transformation to be used.
- **padding** (*str*) – Padding method ('zero', 'mirror', or 'none').
- **median_filter** (*bool*) – If true a median filter in time is applied which removes outliers in time like cosmic rays.
- **threshold_function** (*str*) – Threshold function used for wavelet shrinkage in the wavelet space ('soft' or 'hard').

Returns None

Return type NoneType

run() → None

Run method of the module. Applies the time denoising for the lines in time in parallel.

Returns None

Return type NoneType

Module contents

pynpoint.util package

Submodules

pynpoint.util.analysis module

Functions for point source analysis.

pynpoint.util.analysis.**fake_planet**(*images: numpy.ndarray, psf: numpy.ndarray, parang: numpy.ndarray, position: Tuple[float, float], magnitude: float, psf_scaling: float, interpolation: str = 'spline'*) → numpy.ndarray

Function to inject artificial planets in a dataset.

Parameters

- **images** (*numpy.ndarray*) – Input images (3D).
- **psf** (*numpy.ndarray*) – PSF template (3D).
- **parang** (*numpy.ndarray*) – Parallactic angles (deg).
- **position** (*tuple(float, float)*) – Separation (pix) and position angle (deg) measured in counterclockwise with respect to the upward direction.
- **magnitude** (*float*) – Magnitude difference used to scale input PSF.
- **psf_scaling** (*float*) – Extra factor used to scale input PSF.
- **interpolation** (*str*) – Interpolation type ('spline', 'bilinear', or 'fft').

Returns Images with artificial planet injected.

Return type numpy.ndarray

pynpoint.util.analysis.**false_alarm**(*image: numpy.ndarray, x_pos: float, y_pos: float, size: float, ignore: bool*) → Tuple[float, float, float, float]

Function for the formal t-test for high-contrast imaging at small working angles and the related false positive fraction (Mawet et al. 2014).

Parameters

- **image** (*numpy.ndarray*) – Input image (2D).
- **x_pos** (*float*) – Position (pix) along the horizontal axis. The pixel coordinates of the bottom-left corner of the image are (-0.5, -0.5).
- **y_pos** (*float*) – Position (pix) along the vertical axis. The pixel coordinates of the bottom-left corner of the image are (-0.5, -0.5).
- **size** (*float*) – Aperture radius (pix).
- **ignore** (*bool*) – Ignore the neighboring apertures for the noise estimate.

Returns

- *float* – Signal.
- *float* – Noise level.
- *float* – Signal-to-noise ratio.
- *float* – False positive fraction (FPF).

pynpoint.util.analysis.**merit_function**(*residuals: numpy.ndarray, merit: str, aperture: Tuple[int, int, float], sigma: float*) → float

Function to calculate the figure of merit at a given position in the image residuals.

Parameters

- **residuals** (*numpy.ndarray*) – Residuals of the PSF subtraction (2D).
- **merit** (*str*) – Figure of merit for the chi-square function ('hessian', 'poisson', or 'gaussian').
- **aperture** (*tuple(int, int, float)*) – Position (y, x) of the aperture center (pix) and aperture radius (pix).
- **sigma** (*float*) – Standard deviation (pix) of the Gaussian kernel which is used to smooth the residuals before the chi-square is calculated.

Returns Chi-square ('poisson' and 'gaussian') or sum of the absolute values ('hessian').

Return type float

pynpoint.util.analysis.**student_t**(*t_input: Tuple[str, float], radius: float, size: float, ignore: bool*) → float

Function to calculate the false positive fraction for a given sigma level (Mawet et al. 2014).

Parameters

- **t_input** (*tuple(str, float)*) – Tuple with the input type ('sigma' or 'fpf') and the input value.
- **radius** (*float*) – Aperture radius (pix).
- **size** (*float*) – Separation of the aperture center (pix).
- **ignore** (*bool*) – Ignore neighboring apertures of the point source to exclude the self-subtraction lobes.

Returns False positive fraction (FPF).

Return type float

pynpoint.util.attributes module

Functions for adding attributes to a dataset in the central database.

pynpoint.util.attributes.**set_extra_attr**(*fits_file: str, nimages: int, config_port: pynpoint.core.dataio.ConfigPort, image_out_port: pynpoint.core.dataio.OutputPort, first_index: int*) → int

Function which adds extra attributes to the central database.

Parameters

- **fits_file** (*str*) – Absolute path and filename of the FITS file.
- **nimages** (*int*) – Number of images.
- **config_port** (pynpoint.core.dataio.ConfigPort) – Configuration port.
- **image_out_port** (pynpoint.core.dataio.OutputPort) – Output port of the images to which the attributes are stored.
- **first_index** (*int*) – First image index of the current subset.

Returns First image index for the next subset.

Return type int

pynpoint.util.attributes.**set_nonstatic_attr**(*header: astropy.io.fits.header.Header*, *config_port: pynpoint.core.dataio.ConfigPort*, *image_out_port: pynpoint.core.dataio.OutputPort*, *check: bool = True*) → None

Function which adds the non-static attributes to the central database.

Parameters

- **header** (*astropy.io.fits.header.Header*) – Header information from the FITS file that is read.
- **config_port** (pynpoint.core.dataio.ConfigPort) – Configuration port.
- **image_out_port** (pynpoint.core.dataio.OutputPort) – Output port of the images to which the non-static attributes are stored.

Returns None

Return type NoneType

pynpoint.util.attributes.**set_static_attr**(*fits_file: str*, *header: astropy.io.fits.header.Header*, *config_port: pynpoint.core.dataio.ConfigPort*, *image_out_port: pynpoint.core.dataio.OutputPort*, *check: bool = True*) → None

Function which adds the static attributes to the central database.

Parameters

- **fits_file** (*str*) – Name of the FITS file.
- **header** (*astropy.io.fits.header.Header*) – Header information from the FITS file that is read.
- **config_port** (pynpoint.core.dataio.ConfigPort) – Configuration port.
- **image_out_port** (pynpoint.core.dataio.OutputPort) – Output port of the images to which the static attributes are stored.
- **check** (*bool*) – Print a warning if certain attributes from the configuration file are not present in the FITS header. If set to *False*, attributes are still written to the dataset but there will be no warning if a keyword is not found in the FITS header.

Returns None

Return type NoneType

pynpoint.util.continuous module

pynpoint.util.continuous.**angularfreq**(*N*, *dt*)

Compute angular frequencies.

Parameters

- **N** (*int*) – Number of data samples.
- **dt** (*int*) – Time step.

Returns Angular frequencies (1D).

Return type numpy.ndarray

pynpoint.util.continuous.**autoscales**(*N*, *dt*, *dj*, *wf*, *p*)
Compute scales as fractional power of two.

Parameters

- **N** (*int*) – Number of data samples.
- **dt** (*int*) – Time step.
- **dj** (*float*) – Scale resolution (smaller values of give finer resolution).
- **wf** (*str*) – Wavelet function ("morlet", "paul", or "dog").
- **p** (*int*) – omega0 ("morlet") or order ("paul", "dog").

Returns Scales (1D).

Return type numpy.ndarray

pynpoint.util.continuous.**cwt**(*x*, *dt*, *scales*, *wf='dog'*, *p=2*)
Continuous Wavelet Transform.

Parameters

- **x** (*numpy.ndarray*) – Data (1D).
- **dt** (*int*) – Time step.
- **scales** (*numpy.ndarray*) – Scales (1D).
- **wf** (*str*) – Wavelet function ("morlet", "paul", or "dog").
- **p** (*int*) – omega0 ("morlet") or order ("paul", "dog").

Returns Transformed data (2D).

Return type numpy.ndarray

pynpoint.util.continuous.**dogft**(*s*, *w*, *order*, *dt*)
Fourier transformed DOG function.

Parameters

- **s** (*numpy.ndarray*) – Scales.
- **w** (*numpy.ndarray*) – Angular frequencies.
- **order** (*int*) – Wavelet order.
- **dt** (*int*) – Time step.

Returns Normalized Fourier transformed DOG function.

Return type numpy.ndarray

pynpoint.util.continuous.**icwt**(*X*, *scales*)
Inverse Continuous Wavelet Transform. The reconstruction factor is not applied.

Parameters

- **X** (*numpy.ndarray*) – Transformed data (2D).
- **scales** (*numpy.ndarray*) – Scales (1D).

Returns 1D data.

Return type numpy.ndarray

pynpoint.util.continuous.**morletft**(*s*, *w*, *w0*, *dt*)
" Fourier transformed morlet function.

Parameters

- **s** (*numpy.ndarray*) – Scales.
- **w** (*numpy.ndarray*) – Angular frequencies.
- **w0** (*int*) – Omega0 frequency.
- **dt** (*int*) – Time step.

Returns Normalized Fourier transformed morlet function

Return type numpy.ndarray

pynpoint.util.continuous.**normalization**(*s*, *dt*)
" :param s: Scales. :type s: numpy.ndarray :param dt: Time step. :type dt: int

Returns Normalized data.

Return type numpy.ndarray

pynpoint.util.image module

Functions for image processing.

pynpoint.util.image.**cartesian_to_polar**(*center: Tuple[float, float], y_pos: float, x_pos: float*) → Tuple[float, float]
Function to convert pixel coordinates to polar coordinates.

Parameters

- **center** (*tuple(float, float)*) – Image center (y, x) from *center_subpixel()*.
- **y_pos** (*float*) – Pixel coordinate along the vertical axis. The bottom left corner of the image is (-0.5, -0.5).
- **x_pos** (*float*) – Pixel coordinate along the horizontal axis. The bottom left corner of the image is (-0.5, -0.5).

Returns Separation (pix) and position angle (deg). The angle is measured counterclockwise with respect to the positive y-axis.

Return type tuple(float, float)

pynpoint.util.image.**center_pixel**(*image: numpy.ndarray*) → Tuple[int, int]
Function to get the pixel position of the image center. Note that this position can not be unambiguously defined for an even-sized image. Python indexing starts at 0 so the coordinates of the pixel in the bottom-left corner are (0, 0).

Parameters **image** (*numpy.ndarray*) – Input image (2D or 3D).

Returns Pixel position (y, x) of the image center.

Return type tuple(int, int)

pynpoint.util.image.**center_subpixel**(*image: numpy.ndarray*) → Tuple[float, float]
Function to get the precise position of the image center. The center of the pixel in the bottom left corner of the image is defined as (0, 0), so the bottom left corner of the image is located at (-0.5, -0.5).

Parameters **image** (*numpy.ndarray*) – Input image (2D or 3D).

Returns Subpixel position (y, x) of the image center.

Return type tuple(float, float)

pynpoint.util.image.**create_mask**(*im_shape: Tuple[int, int], size: Union[Tuple[float, float], Tuple[float, None], Tuple[None, float], Tuple[None, None]]*) → numpy.ndarray

Function to create a mask for the central and outer image regions.

Parameters

- **im_shape** (*tuple(int, int)*) – Image size in both dimensions.
- **size** (*tuple(float, float)*) – Size (pix) of the inner and outer mask.

Returns Image mask.

Return type numpy.ndarray

pynpoint.util.image.**crop_image**(*image: numpy.ndarray, center: Optional[tuple], size: int, copy: bool = True*) → numpy.ndarray

Function to crop square images around a specified position.

Parameters

- **image** (*numpy.ndarray*) – Input image (2D or 3D).
- **center** (*tuple(int, int), None*) – The new image center (y, x). The center of the image is used if set to None.
- **size** (*int*) – Image size (pix) for both dimensions. Increased by 1 pixel if size is an even number.
- **copy** (*bool*) – Whether or not to return a copy (instead of a view) of the cropped image (default: True).

Returns Cropped odd-sized image (2D or 3D).

Return type numpy.ndarray

pynpoint.util.image.**locate_star**(*image: numpy.ndarray, center: Optional[tuple], width: Optional[int], fwhm: Optional[int]*) → numpy.ndarray

Function to locate the star by finding the brightest pixel.

Parameters

- **image** (*numpy.ndarray*) – Input image (2D).
- **center** (*tuple(int, int), None*) – Pixel center (y, x) of the subframe. The full image is used if set to None.
- **width** (*int, None*) – The width (pix) of the subframe. The full image is used if set to None.
- **fwhm** (*int, None*) – Full width at half maximum (pix) of the Gaussian kernel. Not used if set to None.

Returns Position (y, x) of the brightest pixel.

Return type numpy.ndarray

pynpoint.util.image.**pixel_distance**(*im_shape: Tuple[int, int], position: Tuple[int, int] = None*) → numpy.ndarray

Function to calculate the distance of each pixel with respect to a given pixel position.

Parameters

- **im_shape** (*tuple(int, int)*) – Image shape (y, x).

- **position** (*tuple(int, int)*) – Pixel center (y, x) from which the distance is calculated. The image center is used if set to None. Python indexing starts at zero so the bottom left pixel is (0, 0).

Returns 2D array with the distances of each pixel from the provided pixel position.

Return type numpy.ndarray

pynpoint.util.image.**polar_to_cartesian**(*image: numpy.ndarray*, *sep: float*, *ang: float*) → Tuple[float, float]

Function to convert polar coordinates to pixel coordinates.

Parameters

- **image** (*numpy.ndarray*) – Input image (2D or 3D).
- **sep** (*float*) – Separation (pix).
- **ang** (*float*) – Position angle (deg), measured counterclockwise with respect to the positive y-axis.

Returns Cartesian coordinates (y, x). The bottom left corner of the image is (-0.5, -0.5).

Return type tuple(float, float)

pynpoint.util.image.**rotate_coordinates**(*center: Tuple[float, float], position: Tuple[float, float], angle: float*) → Tuple[float, float]

Function to rotate coordinates around the image center.

Parameters

- **center** (*tuple(float, float)*) – Image center (y, x).
- **position** (*tuple(float, float)*) – Position (y, x) in the image.
- **angle** (*float*) – Angle (deg) to rotate in counterclockwise direction.

Returns New position (y, x).

Return type tuple(float, float)

pynpoint.util.image.**rotate_images**(*images: numpy.ndarray*, *angles: numpy.ndarray*) → numpy.ndarray

Function to rotate all images in clockwise direction.

Parameters

- **images** (*numpy.ndarray*) – Stack of images (3D).
- **angles** (*numpy.ndarray*) – Rotation angles (deg).

Returns Rotated images.

Return type numpy.ndarray

pynpoint.util.image.**scale_image**(*image: numpy.ndarray*, *scaling_y: float*, *scaling_x: float*) → numpy.ndarray

Function to spatially scale an image.

Parameters

- **image** (*numpy.ndarray*) – Input image (2D).
- **scaling_y** (*float*) – Scaling factor y.
- **scaling_x** (*float*) – Scaling factor x.

Returns Shifted image (2D).

Return type numpy.ndarray

pynpoint.util.image.**select_annulus**(*image_in: numpy.ndarray*, *radius_in: float*, *radius_out: float*, *mask_position: Tuple[float, float] = None*, *mask_radius: float = None*) → numpy.ndarray

image_in [numpy.ndarray] Input image.

radius_in [float] Inner radius of the annulus (pix).

radius_out [float] Outer radius of the annulus (pix).

mask_position [tuple(float, float), None] Center (pix) position (y, x) in of the circular region that is excluded. Not used if set to None.

mask_radius [float, None] Radius (pix) of the circular region that is excluded. Not used if set to None.

pynpoint.util.image.**shift_image**(*image: numpy.ndarray, shift_yx: Union[Tuple[float, float], numpy.ndarray], interpolation: str, mode: str = 'constant'*) → numpy.ndarray

Function to shift an image.

Parameters

- **image** (*numpy.ndarray*) – Input image (2D or 3D). If 3D the image is not shifted along the 0th axis.
- **shift_yx** (*tuple(float, float), np.ndarray*) – Shift (y, x) to be applied (pix). An additional shift of zero pixels will be added for the first dimension in case the input image is 3D.
- **interpolation** (*str*) – Interpolation type ('spline', 'bilinear', or 'fft').
- **mode** (*str*) – Interpolation mode.

Returns Shifted image.

Return type numpy.ndarray

pynpoint.util.image.**subpixel_distance**(*im_shape: Tuple[int, int], position: Tuple[float, float]*) → numpy.ndarray

Function to calculate the distance of each pixel with respect to a given subpixel position.

Parameters

- **im_shape** (*tuple(int, int)*) – Image shape (y, x).
- **position** (*tuple(float, float)*) – Pixel center (y, x) from which the distance is calculated. Python indexing starts at zero so the bottom left image corner is (-0.5, -0.5).

Returns 2D array with the distances of each pixel from the provided pixel position.

Return type numpy.ndarray

pynpoint.util.limits module

Functions for calculating detection limits.

pynpoint.util.limits.**contrast_limit**(*path_images: str, path_psf: str, noise: numpy.ndarray, mask: numpy.ndarray, parang: numpy.ndarray, psf_scaling: float, extra_rot: float, pca_number: int, threshold: Tuple[str, float], aperture: float, residuals: str, snr_inject: float, position: Tuple[float, float]*) → Tuple[float, float, float, float]

Function for calculating the contrast limit at a specified position for a given sigma level or false positive fraction, both corrected for small sample statistics.

Parameters

- **path_images** (*str*) – System location of the stack of images (3D).
- **path_psf** (*str*) – System location of the PSF template for the fake planet (3D). Either a single image or a stack of images equal in size to science data.
- **noise** (*numpy.ndarray*) – Residuals of the PSF subtraction (3D) without injection of fake planets. Used to measure the noise level with a correction for small sample statistics.
- **mask** (*numpy.ndarray*) – Mask (2D).
- **parang** (*numpy.ndarray*) – Derotation angles (deg).
- **psf_scaling** (*float*) – Additional scaling factor of the planet flux (e.g., to correct for a neutral density filter). Should have a positive value.
- **extra_rot** (*float*) – Additional rotation angle of the images in clockwise direction (deg).
- **pca_number** (*int*) – Number of principal components used for the PSF subtraction.
- **threshold** (*tuple(str, float)*) – Detection threshold for the contrast curve, either in terms of 'sigma' or the false positive fraction (FPF). The value is a tuple, for example provided as ('sigma', 5.) or ('fpf', 1e-6). Note that when sigma is fixed, the false positive fraction will change with separation. Also, sigma only corresponds to the standard deviation of a normal distribution at large separations (i.e., large number of samples).
- **aperture** (*float*) – Aperture radius (pix) for the calculation of the false positive fraction.
- **residuals** (*str*) – Method used for combining the residuals ('mean', 'median', 'weighted', or 'clipped').
- **snr_inject** (*float*) – Signal-to-noise ratio of the injected planet signal that is used to measure the amount of self-subtraction.
- **position** (*tuple(float, float)*) – The separation (pix) and position angle (deg) of the fake planet.

Returns

- *float* – Separation (pix).
- *float* – Position angle (deg).
- *float* – Contrast (mag).
- *float* – False positive fraction.

pynpoint.util.mcmc module

Functions for MCMC sampling.

pynpoint.util.mcmc.**lnprob**(*param: numpy.ndarray, bounds: Tuple[Tuple[float, float], Tuple[float, float], Tuple[float, float]], images: numpy.ndarray, psf: numpy.ndarray, mask: numpy.ndarray, parang: numpy.ndarray, psf_scaling: float, pixscale: float, pca_number: int, extra_rot: float, aperture: Tuple[int, int, float], indices: numpy.ndarray, merit: str, residuals: str*) → float

Function for the log posterior function. Should be placed at the highest level of the Python module to be pickable

for the multiprocessing.

Parameters

- **param** (*numpy.ndarray*) – The separation (arcsec), angle (deg), and contrast (mag). The angle is measured in counterclockwise direction with respect to the positive y-axis.
- **bounds** (*tuple(tuple(float, float), tuple(float, float), tuple(float, float))*) – The boundaries of the separation (arcsec), angle (deg), and contrast (mag). Each set of boundaries is specified as a tuple.
- **images** (*numpy.ndarray*) – Stack with images.
- **psf** (*numpy.ndarray*) – PSF template, either a single image (2D) or a cube (3D) with the dimensions equal to *images*.
- **mask** (*numpy.ndarray*) – Array with the circular mask (zeros) of the central and outer regions.
- **parang** (*numpy.ndarray*) – Array with the angles for derotation.
- **psf_scaling** (*float*) – Additional scaling factor of the planet flux (e.g., to correct for a neutral density filter). Should be negative in order to inject negative fake planets.
- **pixscale** (*float*) – Additional scaling factor of the planet flux (e.g., to correct for a neutral density filter). Should be negative in order to inject negative fake planets.
- **pca_number** (*int*) – Number of principal components used for the PSF subtraction.
- **extra_rot** (*float*) – Additional rotation angle of the images (deg).
- **aperture** (*tuple(int, int, float)*) – Position (y, x) of the aperture center (pix) and aperture radius (pix).
- **indices** (*numpy.ndarray*) – Non-masked image indices.
- **merit** (*str*) – Figure of merit that is used for the likelihood function ('gaussian' or 'poisson'). Pixels are assumed to be independent measurements which are expected to be equal to zero in case the best-fit negative PSF template is injected. With 'gaussian', the variance is estimated from the pixel values within an annulus at the separation of the aperture (but excluding the pixels within the aperture). With 'poisson', a Poisson distribution is assumed for the variance of each pixel value.
- **residuals** (*str*) – Method used for combining the residuals ('mean', 'median', 'weighted', or 'clipped').

Returns Log posterior probability.

Return type float

pynpoint.util.module module

Functions for Pypeline modules.

pynpoint.util.module.**memory_frames**(*memory: Union[int, numpy.int64], nimages: int*) → numpy.ndarray

Function to subdivide the input images is in quantities of MEMORY.

Parameters

- **memory** (*int*) – Number of images that is simultaneously loaded into the memory.
- **nimages** (*int*) – Number of images in the stack.

Returns

Return type numpy.ndarray

pynpoint.util.module.**progress**(*current: int*, *total: int*, *message: str*, *start_time: float = None*) → None

Function to show and update the progress as standard output.

Parameters

- **current** (*int*) – Current index.
- **total** (*int*) – Total index number.
- **message** (*str*) – Message that is printed.
- **start_time** (*float, None, optional*) – Start time in seconds. Not used if set to None.

Returns None

Return type NoneType

pynpoint.util.module.**update_arguments**(*index: int, nimages: int, args_in: Optional[tuple]*) → Optional[tuple]

Function to update the arguments of an input function. Specifically, arguments which contain an array with the first dimension equal in size to the total number of images will be substituted by the array element of the image index.

Parameters

- **index** (*int*) – Image index in the stack.
- **nimages** (*int*) – Total number of images in the stack.
- **args_in** (*tuple, None*) – Function arguments that have to be updated.

Returns Updated function arguments.

Return type tuple, None

pynpoint.util.multiline module

Utilities for multiprocessing of lines in time with the poison pill pattern.

class pynpoint.util.multiline.**LineProcessingCapsule**(*image_in_port: pynpoint.core.dataio.InputPort*, *image_out_port: pynpoint.core.dataio.OutputPort*, *num_proc: numpy.int64*, *function: Callable*, *function_args: Optional[tuple]*, *data_length: int*)

Bases: *pynpoint.util.multiproc.MultiprocessingCapsule*

Capsule for parallel processing of lines in time with the poison pill pattern. A function is applied in parallel to each line in time, for example as in *WaveletTimeDenoisingModule*.

Parameters

- **image_in_port** (pynpoint.core.dataio.InputPort) – Input port.
- **image_out_port** (pynpoint.core.dataio.OutputPort) – Output port.
- **num_proc** (*int*) – Number of processors.

- **function** (*function*) – Input function that is applied to the lines.
- **function_args** (*tuple, None, optional*) – Function arguments.
- **data_length** (*int*) – Length of the processed data.

Returns None

Return type NoneType

create_processors() → List[pynpoint.util.multiline.LineTaskProcessor]

Returns List with instances of `LineTaskProcessor`

Return type list(pynpoint.util.multiproc.LineTaskProcessor,)

init_creator(*image_in_port: pynpoint.core.dataio.InputPort*) → pynpoint.util.multiline.LineReader

Parameters **image_in_port** (`pynpoint.core.dataio.InputPort`) – Input port from where the subsets of lines are read.

Returns Line reader object.

Return type *pynpoint.util.multiline.LineReader*

class pynpoint.util.multiline.**LineReader**(*data_port_in: pynpoint.core.dataio.InputPort, tasks_queue_in: multiprocessing.context.BaseContext.JoinableQueue, data_mutex_in: multiprocessing.context.BaseContext.Lock, num_proc: numpy.int64, data_length: int*)

Bases: *pynpoint.util.multiproc.TaskCreator*

Reader of task inputs for *LineProcessingCapsule*. Continuously read all rows of a dataset and puts them into a task queue.

Parameters

- **data_port_in** (`pynpoint.core.dataio.InputPort`) – Input port.
- **tasks_queue_in** (*multiprocessing.queues.JoinableQueue*) – Tasks queue.
- **data_mutex_in** (*multiprocessing.synchronize.Lock*) – A mutex shared with the writer to ensure that no read and write operations happen at the same time.
- **num_proc** (*int*) – Number of processors.
- **data_length** (*int*) – Length of the processed data.

Returns None

Return type NoneType

run() → None

Returns None

Return type NoneType

class pynpoint.util.multiline.**LineTaskProcessor**(*tasks_queue_in: multiprocessing.context.BaseContext.JoinableQueue, result_queue_in: multiprocessing.context.BaseContext.JoinableQueue, function: Callable, function_args: Optional[tuple]*)

Bases: *pynpoint.util.multiproc.TaskProcessor*

Processor of task inputs for *LineProcessingCapsule*. A processor applies a function on a row of lines in time.

Parameters

- **tasks_queue_in** (*multiprocessing.queues.JoinableQueue*) – Tasks queue.
- **result_queue_in** (*multiprocessing.queues.JoinableQueue*) – Results queue.
- **function** (*function*) – Input function.
- **function_args** (*tuple, None*) – Optional function arguments.

Returns None

Return type NoneType

run_job(*tmp_task: pynpoint.util.multiproc.TaskInput*) → pynpoint.util.multiproc.TaskResult

Parameters **tmp_task** (pynpoint.util.multiproc.TaskInput) – Task input with the subsets of lines and the job parameters.

Returns Task result.

Return type *pynpoint.util.multiproc.TaskResult*

pynpoint.util.multipca module

Capsule for multiprocessing of the PSF subtraction with PCA. Residuals are created in parallel for a range of principal components for which the PCA basis is required as input.

class pynpoint.util.multipca.**PcaMultiprocessingCapsule**(*mean_out_port: Optional[pynpoint.core.dataio.OutputPort], median_out_port: Optional[pynpoint.core.dataio.OutputPort], weighted_out_port: Optional[pynpoint.core.dataio.OutputPort], clip_out_port: Optional[pynpoint.core.dataio.OutputPort], num_proc: numpy.int64, pca_numbers: numpy.ndarray, pca_model: sklearn.decomposition.pca.PCA, star_reshape: numpy.ndarray, angles: numpy.ndarray, im_shape: Tuple[int, int, int], indices: numpy.ndarray*)

Bases: *pynpoint.util.multiproc.MultiprocessingCapsule*

Capsule for PCA multiprocessing with the poison pill pattern.

Constructor of PcaMultiprocessingCapsule.

Parameters

- **mean_out_port** (pynpoint.core.dataio.OutputPort) – Output port for the mean residuals.

- **median_out_port** (pynpoint.core.dataio.OutputPort) – Output port for the median residuals.
- **weighted_out_port** (pynpoint.core.dataio.OutputPort) – Output port for the noise-weighted residuals.
- **clip_out_port** (pynpoint.core.dataio.OutputPort) – Output port for the mean clipped residuals.
- **num_proc** (*int*) – Number of processors.
- **pca_numbers** (*numpy.ndarray*) – Number of principal components.
- **pca_model** (*sklearn.decomposition.pca.PCA*) – PCA object with the basis.
- **star_reshape** (*numpy.ndarray*) – Reshaped (2D) input images.
- **angles** (*numpy.ndarray*) – Derotation angles (deg).
- **im_shape** (*tuple(int, int, int)*) – Original shape of the input images.
- **indices** (*numpy.ndarray*) – Non-masked pixel indices.

Returns None

Return type NoneType

create_processors() → List[pynpoint.util.multipca.PcaTaskProcessor]
Method to create a list of instances of PcaTaskProcessor.

Returns PCA task processors.

Return type list(*pynpoint.util.multipca.PcaTaskProcessor*,)

create_writer(*image_out_port: None*) → pynpoint.util.multipca.PcaTaskWriter
Method to create an instance of PcaTaskWriter.

Parameters **image_out_port** (*None*) – Output port, not used.

Returns PCA task writer.

Return type *pynpoint.util.multipca.PcaTaskWriter*

init_creator(*image_in_port: None*) → pynpoint.util.multipca.PcaTaskCreator
Method to create an instance of PcaTaskCreator.

Parameters **image_in_port** (*None*) – Input port, not used.

Returns PCA task creator.

Return type *pynpoint.util.multipca.PcaTaskCreator*

class pynpoint.util.multipca.**PcaTaskCreator**(*tasks_queue_in: multiprocessing.context.BaseContext.JoinableQueue*, *num_proc: numpy.int64*, *pca_numbers: numpy.ndarray*)
Bases: *pynpoint.util.multiproc.TaskCreator*

The TaskCreator of the PCA multiprocessing. Creates one task for each principal component number. Does not require an input port since the data is directly given to the task processors.

Parameters

- **tasks_queue_in** (*multiprocessing.queues.JoinableQueue*) – Input task queue.
- **num_proc** (*int*) – Number of processors.

- **pca_numbers** (*numpy.ndarray*) – Principal components for which the residuals are computed.

Returns None

Return type NoneType

run() → None
Run method of PcaTaskCreator.

Returns None

Return type NoneType

class pynpoint.util.multipca.**PcaTaskProcessor**(*tasks_queue_in: multiprocessing.context.BaseContext.JoinableQueue, result_queue_in: multiprocessing.context.BaseContext.JoinableQueue, star_reshape: numpy.ndarray, angles: numpy.ndarray, pca_model: sklearn.decomposition.pca.PCA, im_shape: Tuple[int, int, int], indices: numpy.ndarray, requirements: Tuple[bool, bool, bool, bool]*)

Bases: *pynpoint.util.multiproc.TaskProcessor*

The TaskProcessor of the PCA multiprocessing is the core of the parallelization. An instance of this class will calculate one forward and backward PCA transformation given the pre-trained scikit-learn PCA model. It does not get data from the TaskCreator but uses its own copy of the input data, which are the same and independent for each task. The following residuals can be created:

- Mean residuals – requirements[0] = True
- Median residuals – requirements[1] = True
- Noise-weighted residuals – requirements[2] = True
- Clipped mean of the residuals – requirements[3] = True

Parameters

- **tasks_queue_in** (*multiprocessing.queues.JoinableQueue*) – Input task queue.
- **result_queue_in** (*multiprocessing.queues.JoinableQueue*) – Input result queue.
- **star_reshape** (*numpy.ndarray*) – Reshaped (2D) stack of images.
- **angles** (*numpy.ndarray*) – Derotation angles (deg).
- **pca_model** (*sklearn.decomposition.pca.PCA*) – PCA object with the basis.
- **im_shape** (*tuple(int, int, int)*) – Original shape of the stack of images.
- **indices** (*numpy.ndarray*) – Non-masked image indices.
- **requirements** (*tuple(bool, bool, bool, bool)*) – Required output residuals.

Returns None

Return type NoneType

run_job(*tmp_task: pynpoint.util.multiproc.TaskInput*) → pynpoint.util.multiproc.TaskResult
Run method of PcaTaskProcessor.

Parameters **tmp_task** (pynpoint.util.multiproc.TaskInput) – Input task.

Returns Output residuals.

Return type *pynpoint.util.multiproc.TaskResult*

class pynpoint.util.multipca.**PcaTaskWriter**(*result_queue_in: multiprocessing.context.BaseContext.JoinableQueue, mean_out_port: Optional[pynpoint.core.dataio.OutputPort], median_out_port: Optional[pynpoint.core.dataio.OutputPort], weighted_out_port: Optional[pynpoint.core.dataio.OutputPort], clip_out_port: Optional[pynpoint.core.dataio.OutputPort], data_mutex_in: multiprocessing.context.BaseContext.Lock, requirements: Tuple[bool, bool, bool, bool]*)

Bases: *pynpoint.util.multiproc.TaskWriter*

The TaskWriter of the PCA parallelization. Four different ports are used to save the results of the task processors (mean, median, weighted, and clipped).

Constructor of PcaTaskWriter.

Parameters

- **result_queue_in** (*multiprocessing.queues.JoinableQueue*) – Input result queue.
- **mean_out_port** (pynpoint.core.dataio.OutputPort) – Output port with the mean residuals. Not used if set to None.
- **median_out_port** (pynpoint.core.dataio.OutputPort) – Output port with the median residuals. Not used if set to None.
- **weighted_out_port** (pynpoint.core.dataio.OutputPort) – Output port with the noise-weighted residuals. Not used if set to None.
- **clip_out_port** (pynpoint.core.dataio.OutputPort) – Output port with the clipped mean residuals. Not used if set to None.
- **data_mutex_in** (*multiprocessing.synchronize.Lock*) – A mutual exclusion variable which ensure that no read and write simultaneously occur.
- **requirements** (*tuple(bool, bool, bool, bool)*) – Required output residuals.

Returns None

Return type NoneType

run() → None
Run method of PcaTaskWriter. Writes the residuals to the output ports.

Returns None

Return type NoneType

pynpoint.util.multiproc module

Abstract interfaces for multiprocessing applications with the poison pill pattern.

class pynpoint.util.multiproc.**MultiprocessingCapsule**(*image_in_port: Optional[pynpoint.core.dataio.InputPort], image_out_port: Optional[pynpoint.core.dataio.OutputPort], num_proc: numpy.int64*)

Bases: object

Abstract interface for multiprocessing capsules based on the poison pill pattern.

Parameters

- **image_in_port** (pynpoint.core.dataio.InputPort, *None*) – Port to the input data.
- **image_out_port** (pynpoint.core.dataio.OutputPort, *None*) – Port to the place where the output data will be stored.
- **num_proc** (*int*) – Number of task processors.

Returns None

Return type NoneType

create_processors() → None
Function that is called from the constructor to create a list of instances of *TaskProcessor*.

Returns None

Return type NoneType

create_writer(*image_out_port: Optional[pynpoint.core.dataio.OutputPort]*) → pynpoint.util.multiproc.TaskWriter
Function that is called from the constructor to create the *TaskWriter*.

Parameters **image_out_port** (pynpoint.core.dataio.OutputPort, *None*) – Output port for the creator.

Returns Task writer.

Return type *pynpoint.util.multiproc.TaskWriter*

init_creator(*image_in_port: Optional[pynpoint.core.dataio.InputPort]*) → None
Function that is called from the constructor to create a *TaskCreator*.

Parameters **image_in_port** (pynpoint.core.dataio.InputPort, *None*) – Input port for the task creator.

Returns None

Return type NoneType

run() → None
Run method that starts the *TaskCreator*, the instances of *TaskProcessor*, and the *TaskWriter*. They will be shut down when all tasks have finished.

Returns None

Return type NoneType

class pynpoint.util.multiproc.**TaskCreator**(*data_port_in: Optional[pynpoint.core.dataio.InputPort], tasks_queue_in: multiprocessing.context.BaseContext.JoinableQueue, data_mutex_in: Optional[multiprocessing.context.BaseContext.Lock], num_proc: numpy.int64*)

Bases: multiprocessing.context.Process

Abstract interface for *TaskCreator* classes. A *TaskCreator* creates instances of *TaskInput*, which will be processed by the *TaskProcessor*, and appends them to the central task queue. In general there is only one *TaskCreator* running for a poison pill multiprocessing application. A *TaskCreator* communicates with to the *TaskWriter* in order to avoid simultaneously access to the central database.

Parameters

- **data_port_in** (pynpoint.core.dataio.InputPort, *None*) – An input port which links to the data that has to be processed.
- **tasks_queue_in** (*multiprocessing.queues.JoinableQueue*) – The central task queue.
- **data_mutex_in** (*multiprocessing.synchronize.Lock, None*) – A mutex shared with the writer to ensure that no read and write operations happen at the same time.
- **num_proc** (*int*) – Maximum number of instances of *TaskProcessor* that run simultaneously.

Returns None

Return type NoneType

create_poison_pills() → None

Creates poison pills for the *TaskProcessor* and *TaskWriter*. A process will shut down if it receives a poison pill as a new task. This method should be executed at the end of the *run()* method.

Returns None

Return type NoneType

run() → None

Creates objects of the *TaskInput* until all tasks are placed in the task queue.

Returns None

Return type NoneType

class pynpoint.util.multiproc.**TaskInput**(*input_data: Union[numpy.ndarray, numpy.int64], job_parameter: tuple*)

Bases: object

Class for tasks that are processed by the *TaskProcessor*.

Parameters

- **input_data** (*int, float, numpy.ndarray*) – Input data for by the *TaskProcessor*.
- **job_parameter** (*tuple*) – Additional data or parameters.

Returns None

Return type NoneType

class pynpoint.util.multiproc.**TaskProcessor**(*tasks_queue_in: multiprocessing.context.BaseContext.JoinableQueue*, *result_queue_in: multiprocessing.context.BaseContext.JoinableQueue*)

Bases: multiprocessing.context.Process

Abstract interface for *TaskProcessor* classes. The number of instances of *TaskProcessor* that run simultaneously in a poison pill multiprocessing application can be set with CPU parameter in the central configuration file. A *TaskProcessor* takes tasks from a task queue, processes the task, and stores the results back into a result queue. The process will shut down if the next task is a poison pill. The order in which process finish is not fixed.

Parameters

- **tasks_queue_in** (*multiprocessing.queues.JoinableQueue*) – The input task queue with instances of *TaskInput*.
- **result_queue_in** (*multiprocessing.queues.JoinableQueue*) – The result task queue with instances of *TaskResult*.

Returns None

Return type NoneType

check_poison_pill(*next_task: Union[pynpoint.util.multiproc.TaskInput, int, None]*) → bool

Function to check if the next task is a poison pill.

Parameters **next_task** (*int, None,* pynpoint.util.multiproc.TaskInput) – The next task.

Returns True if the next task is a poison pill, False otherwise.

Return type bool

run() → None

Run method to start the *TaskProcessor*. The run method will continue to process tasks from the input task queue until it receives a poison pill.

Returns None

Return type NoneType

run_job(*tmp_task: pynpoint.util.multiproc.TaskInput*) → None

Abstract interface for the *run_job()* method which is called from the *run()* method for each task individually.

Parameters **tmp_task** (pynpoint.util.multiproc.TaskInput) – Input task.

Returns None

Return type NoneType

class pynpoint.util.multiproc.**TaskResult**(*data_array: numpy.ndarray*, *position: tuple*)

Bases: object

Class for results that can be stored by the *TaskWriter*.

Parameters

- **data_array** (*numpy.ndarray*) – Array with the results for a given position.
- **position** (*tuple(tuple(int, int, int), tuple(int, int, int), tuple(int, int, int))*) – The position where the results will be stored.

Returns None

Return type NoneType

class pynpoint.util.multiproc.**TaskWriter**(*result_queue_in: multiprocessing.context.BaseContext.JoinableQueue, data_out_port_in: Optional[pynpoint.core.dataio.OutputPort], data_mutex_in: multiprocessing.context.BaseContext.Lock*)

Bases: multiprocessing.context.Process

The *TaskWriter* receives results from the result queue, which have been computed by a *TaskProcessor*, and stores the results in the central database. The position parameter of the *TaskResult* is used to slice the result to the correct position in the complete output dataset.

Parameters

- **result_queue_in** (*multiprocessing.queues.JoinableQueue*) – The result queue.
- **data_out_port_in** (pynpoint.core.dataio.OutputPort, *None*) – The output port where the results will be stored.
- **data_mutex_in** (*multiprocessing.synchronize.Lock*) – A mutex that is shared with the *TaskWriter* which ensures that read and write operations to the database do not occur simultaneously.

Returns None

Return type NoneType

check_poison_pill(*next_result: Optional[pynpoint.util.multiproc.TaskResult]*) → int

Function to check if the next result is a poison pill.

Parameters **next_result** (*None*, pynpoint.util.multiproc.TaskResult) – The next result.

Returns 0 -> no poison pill, 1 -> poison pill, 2 -> poison pill but still results in the queue (rare error case).

Return type int

run() → None

Run method of the *TaskWriter*. It is called once when it has to start storing the results until it receives a poison pill.

Returns None

Return type NoneType

pynpoint.util.multiproc.**apply_function**(*tmp_data: numpy.ndarray, func: Callable, func_args: Optional[tuple]*) → numpy.ndarray

Apply a function with optional arguments to the input data.

Parameters

- **tmp_data** (*numpy.ndarray*) – Input data.
- **func** (*function*) – Function.
- **func_args** (*tuple*, *None*) – Function arguments.

Returns The results of the function.

Return type numpy.ndarray

pynpoint.util.multiproc.**to_slice**(*tuple_slice: tuple*) → tuple
Function to convert tuples into slices for a multiprocessing queue.

Parameters **tuple_slice** (*tuple*) – Tuple to be converted into a slice.

Returns Tuple with three slices.

Return type tuple(slice, slice, slice)

pynpoint.util.multistack module

Utilities for multiprocessing of stacks of images.

class pynpoint.util.multistack.**StackProcessingCapsule**(*image_in_port: pynpoint.core.dataio.InputPort, image_out_port: pynpoint.core.dataio.OutputPort, num_proc: numpy.int64, function: Callable, function_args: Optional[tuple], stack_size: int, result_shape: tuple, nimages: int*)
Bases: *pynpoint.util.multiproc.MultiprocessingCapsule*

Capsule for parallel processing of stacks of images with the poison pill pattern. A function is applied in parallel to each stack of images.

Parameters

- **image_in_port** (pynpoint.core.dataio.InputPort) – Input port.
- **image_out_port** (pynpoint.core.dataio.OutputPort) – Output port.
- **num_proc** (*int*) – Number of processors.
- **function** (*function*) – Input function.
- **function_args** (*tuple, None*) – Function arguments.
- **stack_size** (*int*) – Number of images per stack.
- **result_shape** (*tuple(int,)*) – Shape of the array with output results (usually a stack of images).
- **nimages** (*int*) – Total number of images.

Returns None

Return type NoneType

create_processors() → List[pynpoint.util.multistack.StackTaskProcessor]

Returns List with instances of StackTaskProcessor.

Return type list(pynpoint.util.multiproc.StackTaskProcessor,)

init_creator(*image_in_port: pynpoint.core.dataio.InputPort*) → pynpoint.util.multistack.StackReader

Parameters **image_in_port** (pynpoint.core.dataio.InputPort) – Input port from where the subsets of images are read.

Returns Reader of stacks of images.

Return type *pynpoint.util.multistack.StackReader*

class pynpoint.util.multistack.**StackReader**(*data_port_in: pynpoint.core.dataio.InputPort*, *tasks_queue_in: multiprocessing.context.BaseContext.JoinableQueue*, *data_mutex_in: multiprocessing.context.BaseContext.Lock*, *num_proc: numpy.int64*, *stack_size: int*, *result_shape: tuple*)

Bases: *pynpoint.util.multiproc.TaskCreator*

Reader of task inputs for *StackProcessingCapsule*. Reads continuously stacks of images of a dataset and puts them into a task queue.

Parameters

- **data_port_in** (*pynpoint.core.dataio.InputPort*) – Input port.
- **tasks_queue_in** (*multiprocessing.queues.JoinableQueue*) – Tasks queue.
- **data_mutex_in** (*multiprocessing.synchronize.Lock*) – A mutex shared with the writer to ensure that no read and write operations happen at the same time.
- **num_proc** (*int*) – Number of processors.
- **stack_size** (*int*) – Number of images per stack.
- **result_shape** (*tuple(int,)*) – Shape of the array with the output results (usually a stack of images).

Returns None

Return type NoneType

run() → None

Returns None

Return type NoneType

class pynpoint.util.multistack.**StackTaskProcessor**(*tasks_queue_in: multiprocessing.context.BaseContext.JoinableQueue*, *result_queue_in: multiprocessing.context.BaseContext.JoinableQueue*, *function: Callable*, *function_args: Optional[tuple]*, *nimages: int*)

Bases: *pynpoint.util.multiproc.TaskProcessor*

Processor of task inputs for *StackProcessingCapsule*. A processor applies a function on a stack of images.

Parameters

- **tasks_queue_in** (*multiprocessing.queues.JoinableQueue*) – Tasks queue.
- **result_queue_in** (*multiprocessing.queues.JoinableQueue*) – Results queue.
- **function** (*function*) – Input function that is applied to the images.
- **function_args** (*tuple, None*) – Function arguments.
- **nimages** (*int*) – Total number of images.

Returns None

Return type NoneType

run_job(*tmp_task: pynpoint.util.multiproc.TaskInput*) → pynpoint.util.multiproc.TaskResult

Parameters **tmp_task** (pynpoint.util.multiproc.TaskInput) – Task input with the subsets of images and the job parameters.

Returns Task result.

Return type *pynpoint.util.multiproc.TaskResult*

pynpoint.util.psf module

Functions for PSF subtraction.

pynpoint.util.psf.**pca_psf_subtraction**(*images: numpy.ndarray*, *angles: numpy.ndarray*, *pca_number: int*, *pca_sklearn: sklearn.decomposition.pca.PCA = None*, *im_shape: Tuple[int, int, int] = None*, *indices: numpy.ndarray = None*) → numpy.ndarray

Function for PSF subtraction with PCA.

Parameters

- **images** (*numpy.ndarray*) – Stack of images. Also used as reference images if *pca_sklearn* is set to None. Should be in the original 3D shape if *pca_sklearn* is set to None or in the 2D reshaped format if *pca_sklearn* is not set to None.
- **parang** (*numpy.ndarray*) – Derotation angles (deg).
- **pca_number** (*int*) – Number of principal components used for the PSF model.
- **pca_sklearn** (*sklearn.decomposition.pca.PCA, None*) – PCA object with the basis if not set to None.
- **im_shape** (*tuple(int, int, int), None*) – Original shape of the stack with images. Required if *pca_sklearn* is not set to None.
- **indices** (*numpy.ndarray, None*) – Non-masked image indices. All pixels are used if set to None.

Returns

- *numpy.ndarray* – Residuals of the PSF subtraction.
- *numpy.ndarray* – Derotated residuals of the PSF subtraction.

pynpoint.util.remove module

Functions to write selected data and attributes to the central database.

pynpoint.util.remove.**write_selected_attributes**(*indices: numpy.ndarray*, *port_input: pynpoint.core.dataio.InputPort*, *port_selected: pynpoint.core.dataio.OutputPort*, *port_removed: pynpoint.core.dataio.OutputPort*) → None

Function to write the attributes of a selected number of images.

Parameters

- **indices** (*numpy.ndarray*) – Indices that are removed.
- **port_input** (pynpoint.core.dataio.InputPort) – Port to the input data.
- **port_selected** (pynpoint.core.dataio.OutputPort) – Port to store the attributes of the selected images.
- **port_removed** (pynpoint.core.dataio.OutputPort) – Port to store the attributes of the removed images. Not written if set to None.

Returns None

Return type NoneType

pynpoint.util.remove.**write_selected_data**(*images: numpy.ndarray*, *indices: numpy.ndarray*, *port_selected: pynpoint.core.dataio.OutputPort*, *port_removed: pynpoint.core.dataio.OutputPort*) → None

Function to write a selected number of images from a data set.

Parameters

- **images** (*numpy.ndarray*) – Stack of images.
- **indices** (*numpy.ndarray*) – Indices that are removed.
- **port_selected** (pynpoint.core.dataio.OutputPort) – Port to store the selected images.
- **port_removed** (pynpoint.core.dataio.OutputPort) – Port to store the removed images.

Returns None

Return type NoneType

pynpoint.util.residuals module

Functions for combining the residuals of the PSF subtraction.

pynpoint.util.residuals.**combine_residuals**(*method: str*, *res_rot: numpy.ndarray*, *residuals: numpy.ndarray = None*, *angles: numpy.ndarray = None*) → numpy.ndarray

Function for combining the derotated residuals of the PSF subtraction.

Parameters

- **method** (*str*) – Method used for combining the residuals ('mean', 'median', 'weighted', or 'clipped').
- **res_rot** (*numpy.ndarray*) – Derotated residuals of the PSF subtraction (3D).
- **residuals** (*numpy.ndarray, None*) – Non-derotated residuals of the PSF subtraction (3D). Only required for the noise-weighted residuals.
- **angles** (*numpy.ndarray, None*) – Derotation angles (deg). Only required for the noise-weighted residuals.

Returns Combined residuals (3D).

Return type numpy.ndarray

pynpoint.util.tests module

Functions for testing the pipeline and modules.

pynpoint.util.tests.**create_config**(*filename: str*) → None
Create a configuration file.

Parameters **filename** (*str*) – Configuration filename.

Returns None

Return type NoneType

pynpoint.util.tests.**create_fake**(*path: str, ndit: List[int], nframes: List[int], exp_no: List[int], npix: Tuple[int, int], fwhm: Optional[float], x0: List[float], y0: List[float], angles: List[List[float]], sep: Optional[float], contrast: Optional[float]*) → None
Create ADI test data with a fake planet.

Parameters

- **path** (*str*) – Working folder.
- **ndit** (*list(int,)*) – Number of exposures.
- **nframes** (*list(int,)*) – Number of images.
- **exp_no** (*list(int,)*) – Exposure numbers.
- **npix** (*tupe(int, int)*) – Number of pixels in x and y direction.
- **fwhm** (*float, None*) – Full width at half maximum (pix) of the PSF.
- **x0** (*list(float,)*) – Horizontal positions of the star.
- **y0** (*list(float,)*) – Vertical positions of the star.
- **angles** (*list(list(float, float),)*) – Derotation angles (deg).
- **sep** (*float, None*) – Separation of the planet.
- **contrast** (*float, None*) – Brightness contrast of the planet.

Returns None

Return type NoneType

pynpoint.util.tests.**create_fits**(*path: str, filename: str, image: numpy.ndarray, ndit: int, exp_no: int = 0, parang: List[float] = [0.0, 0.0], x0: float = 0.0, y0: float = 0.0*) → None
Create a FITS file with images and header information.

Parameters

- **path** (*str*) – Working folder.
- **filename** (*str*) – FITS filename.
- **image** (*numpy.ndarray*) – Images.
- **ndit** (*int*) – Number of integrations.
- **exp_no** (*int*) – Exposure number.
- **parang** (*list(float, float)*) – Start and end parallactic angle.
- **x0** (*float*) – Horizontal dither position.

- **y0** (*float*) – Vertical dither position.

Returns None

Return type NoneType

pynpoint.util.tests.**create_near_data**(*path: str*) → None

Create a stack of images with Gaussian distributed pixel values.

Parameters **path** (*str*) – Working folder.

Returns None

Return type NoneType

pynpoint.util.tests.**create_random**(*path: str*, *ndit: int = 10*, *parang: Optional[numpy.ndarray] = array([1., 2., 3., 4., 5., 6., 7., 8., 9., 10.])*) → None

Create a stack of images with Gaussian distributed pixel values.

Parameters

- **path** (*str*) – Working folder.
- **ndit** (*int*) – Number of images.
- **parang** (*numpy.ndarray, None*) – Parallactic angles.

Returns None

Return type NoneType

pynpoint.util.tests.**create_star_data**(*path: str, npix_x: int = 100, npix_y: int = 100, x0: List[float] = [50.0, 50.0, 50.0, 50.0], y0: List[float] = [50.0, 50.0, 50.0, 50.0], parang_start: List[float] = [0.0, 5.0, 10.0, 15.0], parang_end: List[float] = [5.0, 10.0, 15.0, 20.0], exp_no: List[int] = [1, 2, 3, 4], ndit: int = 10, nframes: int = 10, noise: bool = True*) → None

Create data with a stellar PSF and Gaussian noise.

Parameters

- **path** (*str*) – Working folder.
- **npix_x** (*int*) – Number of pixels in horizontal direction.
- **npix_y** (*int*) – Number of pixels in vertical direction.
- **x0** (*list(float,)*) – Positions of the PSF in horizontal direction.
- **y0** (*list(float,)*) – Positions of the PSF in vertical direction.
- **parang_start** (*list(float,)*) – Start values of the parallactic angle (deg).
- **parang_end** (*list(float,)*) – End values of the parallactic angle (deg).
- **exp_no** (*list(int,)*) – Exposure numbers.
- **ndit** (*int*) – Number of exposures.
- **nframes** (*int*) – Number of frames.
- **noise** (*bool*) – Adding noise to the images.

Returns None

Return type NoneType

pynpoint.util.tests.**create_waffle_data**(*path: str, npix: int, x_spot: List[float], y_spot: List[float]*) → None

Create data with satellite spots and Gaussian noise.

Parameters

- **path** (*str*) – Working folder.
- **npix** (*int*) – Number of pixels in both dimensions.
- **x_spot** (*list(float,)*) – Pixel positions in horizontal direction of the satellite spots.
- **y_spot** (*list(float,)*) – Pixel positions in vertical direction of the satellite spots.

Returns None

Return type NoneType

pynpoint.util.tests.**remove_test_data**(*path: str, folders: List[str] = None, files: List[str] = None*) → None

Function to remove data created by the test cases.

Parameters

- **path** (*str*) – Working folder.
- **folders** (*list(str,)*) – Folders to remove.
- **files** (*list(str,)*) – Files to removes.

Returns None

Return type NoneType

pynpoint.util.wavelets module

Wrapper utils for the wavelet functions for the mlpy cwt implementation (see continous.py)

class pynpoint.util.wavelets.**WaveletAnalysisCapsule**(*signal_in: numpy.ndarray, wavelet_in: str = 'dog', order: int = 2, padding: str = 'none', frequency_resolution: float = 0.5*)

Bases: object

Capsule class to process one 1d time series using the CWT and wavelet de-nosing by wavelet shrinkage.

Parameters

- **signal_in** (*numpy.ndarray*) – 1D input signal.
- **wavelet_in** (*str*) – Wavelet function ('dog' or 'morlet').
- **order** (*int*) – Order of the wavelet function.
- **padding** (*str*) – Padding method ('zero', 'mirror', or 'none').
- **frequency_resolution** (*float*) – Wavelet space resolution in scale/frequency.

Returns None

Return type NoneType

compute_cwt() → None

Compute the wavelet space of the given input signal.

Returns None

Return type NoneType

denoise_spectrum(*soft: bool = False*) → None
Applies wavelet shrinkage on the current wavelet space (m_spectrum) by either a hard of soft threshold function.

Parameters **soft** (*bool*) – If True a soft threshold is used, hard otherwise.

Returns None

Return type NoneType

get_signal() → numpy.ndarray
Returns the current version of the 1d signal. Use update_signal() in advance in order to get the current reconstruction of the wavelet space. Removes padded values as well.

Returns Current version of the 1D signal.

Return type numpy.ndarray

median_filter() → None
Applies a median filter on the internal 1d signal. Can be useful for cosmic ray correction after temporal de-noising

Returns None

Return type NoneType

update_signal() → None
Updates the internal signal by the reconstruction of the current wavelet space.

Returns None

Return type NoneType

Module contents

8.1.2 Module contents

CHAPTER 9

Data Reduction

9.1 Introduction

The documentation on this page contains an introduction into data reduction of the modified instrument for the (New Earths in the Alpha Cen Region) experiment. All data are available in the ESO archive under program ID .

The basic processing steps with PynPoint are described in the example below while a complete overview of all available pipeline modules can be found in the *Overview* section. Further details about the pipeline architecture and data processing are also available in . More in-depth information of the input parameters for individual PynPoint modules can be found in the *API Documentation*.

Please also have a look at the *Attribution* section when using PynPoint results in a publication.

9.2 Example

In this example, we will process the images of chop A (i.e., frames in which alpha Cen A was centered behind the AGPM coronagraph). Note that the same procedure can be applied on the images of chop B (i.e., with alpha Cen B centered behind the coronagraph).

9.2.1 Setup

To get started, use the instructions available in the *Installation* section to install PynPoint.

The results shown below are based on 1 hour of commissioning data of alpha Cen. There is a available to download all the FITS files (126 Gb). First make the bash script executable:

```
$ chmod +x near_files.sh
```

And then execute it as:

```
$ ./near_files.sh
```

You can also start by downloading only a few files by running a subset of the bash script lines (useful for validating the pipeline installation because analyzing the full data set takes hours).

Now that we have the data, we can start the data reduction with PynPoint!

The *Pypeline* of PynPoint requires a folder for the working_place, input_place, and output_place. Before we start running PynPoint, we have to put the raw NEAR data in the default input folder or the location that will provided as input_dir in the *NearReadingModule*.

Then we create a configuration file which contains the global pipeline settings and is used to select the required FITS header keywords. Create a text file called PynPoint_config.ini in the working_place folder with the following content:

```
[header]

INSTRUMENT: INSTRUME
NFRAMES: ESO DET CHOP NCYCLES
EXP_NO: ESO TPL EXPNO
DIT: ESO DET SEQ1 DIT
NDIT: None
PARANG_START: ESO ADA POSANG
PARANG_END: ESO ADA POSANG END
DITHER_X: None
DITHER_Y: None
PUPIL: ESO ADA PUPILPOS
DATE: DATE-OBS
LATITUDE: ESO TEL GEOLAT
LONGITUDE: ESO TEL GEOLON
RA: RA
DEC: DEC

[settings]

PIXSCALE: 0.045
MEMORY: 1000
CPU: 1
```

The MEMORY and CPU setting can be adjusted. They define the number of images that is simultaneously loaded into the computer memory and the number of parallel processes that are used by some of the pipeline modules.

Note that in addition to the config file above, the working_place directory is also used to store the database file (*PynPoint_database.hdf5*). This database stores all intermediate results (typically a stack of images), which allows the user to rerun particular processing steps without having to rerun the complete pipeline.

9.2.2 Running PynPoint

Example code snippets for the different steps to reduce NEAR data with PynPoint are included below. These code snippets can be executed in Python interactive mode, as a Jupyter notebook.py file, or combined into a python script and executed from the command line.

The first steps are to initialize the pipeline and read in the data contained in the given input_place_in directory. Data are automatically divided into the chop A and chop B data sets. Here we also use the *AngleInterpolationModule* to calculate the parallactic angle for each individual frame, which is necessary for derotating and combining the frames after PSF subtraction:

```
# Import the Pypeline and the modules that we will use in this example
```

(continues on next page)

(continued from previous page)

```
from pynpoint import Pypeline, NearReadingModule, AngleInterpolationModule, \
                     CropImagesModule, SubtractImagesModule, ExtractBinaryModule, \
                     StarAlignmentModule, FitCenterModule, ShiftImagesModule, \
                     FakePlanetModule, PSFpreparationModule, PcaPsfSubtractionModule,␣
↪\
                     ContrastCurveModule, FitsWritingModule, TextWritingModule

# Create a Pypeline instance (change the directories to the correct paths)

pipeline = Pypeline(working_place_in='working_folder/',  # directory for database and␣
↪config files
                    input_place_in='input_folder/',      # default directory for␣
↪reading in input data
                    output_place_in='output_folder/')    # default directory for␣
↪saving output files
                                                         #   (i.e., with␣
↪FitsWritingModule used below)

# Read the raw data (i.e., all the fits files contained in the input_place_in folder␣
↪above)
# and separate the chop A and chop B images

module = NearReadingModule(name_in='read',
                           input_dir=None,
                           chopa_out_tag='chopa',
                           chopb_out_tag='chopb')

pipeline.add_module(module)

# Interpolate the parallactic angles between the start and end value of each FITS file
# The angles will be added as PARANG attribute to the chop A and chop B datasets

module = AngleInterpolationModule(name_in='angle1',
                                  data_tag='chopa')

pipeline.add_module(module)

module = AngleInterpolationModule(name_in='angle2',
                                  data_tag='chopb')

pipeline.add_module(module)

# Run each of the above modules using their 'name_in' tags

pipeline.run_module('read')
pipeline.run_module('angle1')
pipeline.run_module('angle2')

# Note that you can also run all the added modules using this function:
# pipeline.run()
```

The next step is to reduce the chop A frames with alpha Cen A behind the corognagraph. Here we crop the chop A and chop B images around the coronagraph position, subtract chop B from chop A to remove the sky background, and center the subtracted chop A frames:

```
# Crop the chop A and chop B images around the approximate coronagraph position
```

(continues on next page)

(continued from previous page)

```
module = CropImagesModule(size=5.,
                          center=(432, 287),
                          name_in='crop1',
                          image_in_tag='chopa',
                          image_out_tag='chopa_crop')

pipeline.add_module(module)

module = CropImagesModule(size=5.,
                          center=(432, 287),
                          name_in='crop2',
                          image_in_tag='chopb',
                          image_out_tag='chopb_crop')

pipeline.add_module(module)

# Subtract frame-by-frame chop B from chop A

module = SubtractImagesModule(name_in='subtract_aminusb',
                              image_in_tags=('chopa_crop', 'chopb_crop'),
                              image_out_tag='chopa_sub',
                              scaling=1.)

pipeline.add_module(module)

# Fit the center position of chop A, using the images from before the chop-subtraction
# For simplicity, only the mean of all images is fitted

module = FitCenterModule(name_in='center1',
                         image_in_tag='chopa_crop',
                         fit_out_tag='chopa_fit',
                         mask_out_tag=None,
                         method='mean',
                         radius=1.,
                         sign='positive',
                         model='moffat',
                         filter_size=None,
                         guess=(0., 0., 10., 10., 1e4, 0., 0., 1.))

pipeline.add_module(module)


# Center the chop-subtracted images

module = ShiftImagesModule(shift_xy='chopa_fit',
                           name_in='shift1',
                           image_in_tag='chopa_sub',
                           image_out_tag='chopa_center',
                           interpolation='spline')

pipeline.add_module(module)

# Run each of the above modules

pipeline.run_module('crop1')
pipeline.run_module('crop2')
```

(continues on next page)

(continued from previous page)

```
pipeline.run_module('subtract_aminusb')
pipeline.run_module('center1')
pipeline.run_module('shift1')
```

Next, we use the chop B frames where alpha Cen A if off of the coronagraph to extract a reference PSF. This reference PSF will later be used for calculating the detection limits:

```
# Subtract chop A from chop B before extracting the non-coronagraphic PSF

module = SubtractImagesModule(name_in='subtract_bminusa',
                              image_in_tags=('chopb', 'chopa'),
                              image_out_tag='chopb_sub',
                              scaling=1.)

pipeline.add_module(module)

# Crop out the non-coronagraphic PSF for chop A from the chop B images

module = ExtractBinaryModule(pos_center=(432., 287.),
                             pos_binary=(430., 175.),
                             name_in='extract_refpsf',
                             image_in_tag='chopb_sub',
                             image_out_tag='psfa',
                             image_size=5.,
                             search_size=1.,
                             filter_size=None)

pipeline.add_module(module)

# Align the non-coronagraphic PSF images

module = StarAlignmentModule(name_in='align_refpsf',
                             image_in_tag='psfa',
                             ref_image_in_tag=None,
                             image_out_tag='psfa_align',
                             interpolation='spline',
                             accuracy=10,
                             resize=None,
                             num_references=10,
                             subframe=1.)

pipeline.add_module(module)

# Fit the center position of the mean, non-coronagraphic PSF

module = FitCenterModule(name_in='center_refpsf',
                         image_in_tag='psfa',
                         fit_out_tag='psfa_fit',
                         mask_out_tag=None,
                         method='mean',
                         radius=1.,
                         sign='positive',
                         model='moffat',
                         filter_size=None,
                         guess=(0., 0., 10., 10., 1e4, 0., 0., 1.))
```

(continues on next page)

(continued from previous page)

```
pipeline.add_module(module)

# Center the non-coronagraphic PSF images

module = ShiftImagesModule(shift_xy='psfa_fit',
                           name_in='shift_refpsf',
                           image_in_tag='psfa',
                           image_out_tag='psfa_center',
                           interpolation='spline')

pipeline.add_module(module)

# Mask the non-coronagraphic PSF beyond 1 arsec

module = PSFpreparationModule(name_in='prep_refpsf',
                              image_in_tag='psfa_center',
                              image_out_tag='psfa_mask',
                              mask_out_tag=None,
                              norm=False,
                              cent_size=None,
                              edge_size=1.)

pipeline.add_module(module)

# Run each of the above modules

pipeline.run_module('subtract_bminusa')
pipeline.run_module('extract_refpsf')
pipeline.run_module('align_refpsf')
pipeline.run_module('center_refpsf')
pipeline.run_module('shift_refpsf')
pipeline.run_module('prep_refpsf')
```

Finally, we use PCA to subtract the stellar PSF of alpha Cen A. For testing purposes, we first use the reference PSF created above to inject a fake planet into the chop A data. The median combination of the PSF-subtracted and derotated frames is saved in its own tag and then written out to a fits file:

```
# Inject a fake planet at a separation of 1 arcsec and a contrast of 10 mag

module = FakePlanetModule(position=(1., 0.),
                          magnitude=10.,
                          psf_scaling=1.,
                          interpolation='spline',
                          name_in='fake',
                          image_in_tag='chopa_center',
                          psf_in_tag='psfa_mask',
                          image_out_tag='chopa_fake')

pipeline.add_module(module)

# Mask the central and outer part of the chop A images

module = PSFpreparationModule(name_in='prep_data',
                              image_in_tag='chopa_fake',
                              image_out_tag='chopa_prep',
                              mask_out_tag=None,
```

(continues on next page)

(continued from previous page)

```
                                norm=False,
                                cent_size=0.3,
                                edge_size=3.)

pipeline.add_module(module)

# Subtract a PSF model with PCA and median-combine the residuals

module = PcaPsfSubtractionModule(pca_numbers=range(1, 51),
                                 name_in='pca',
                                 images_in_tag='chopa_prep',
                                 reference_in_tag='chopa_prep',
                                 res_median_tag='chopa_pca',
                                 extra_rot=0.0)

pipeline.add_module(module)

# Datasets can be exported to FITS files by their tag name in the database
# Here we will export the median-combined residuals of the PSF subtraction

module = FitsWritingModule(name_in='write_result_psfsub',
                           file_name='chopa_pca.fits',
                           output_dir=None,
                           data_tag='chopa_pca',
                           data_range=None,
                           overwrite=True)

pipeline.add_module(module)

# Run each of the above modules

pipeline.run_module('fake')
pipeline.run_module('prep_data')
pipeline.run_module('pca')
pipeline.run_module('write_result_psfsub')
```

PynPoint also includes a module to calculate the detection limits of the final image:

```
# Calculate detection limits between 0.8 and 2.0 arcsec
# The false positive fraction is fixed to 2.87e-6 (i.e. 5 sigma for Gaussian␣
↪statistics)

module = ContrastCurveModule(name_in='limits',
                             image_in_tag='chopa_center',
                             psf_in_tag='psfa_mask',
                             contrast_out_tag='limits',
                             separation=(0.3, 2., 0.1),
                             angle=(0., 360., 60.),
                             threshold=('fpf', 2.87e-6),
                             psf_scaling=1.,
                             aperture=0.15,
                             pca_number=10,
                             cent_size=0.3,
                             edge_size=3.,
                             extra_rot=0.,
                             residuals='median')
```

(continues on next page)

(continued from previous page)

```
pipeline.add_module(module)

# And we write the detection limits to a text file

header = 'Separation [arcsec] - Contrast [mag] - Variance [mag] - FPF'

module = TextWritingModule(name_in='write_result_limits',
                           file_name='contrast_curve.dat',
                           output_dir=None,
                           data_tag='limits',
                           header=header)

pipeline.add_module(module)

# Run each of the above modules

pipeline.run_module('limits')
pipeline.run_module('write_result_limits')
```

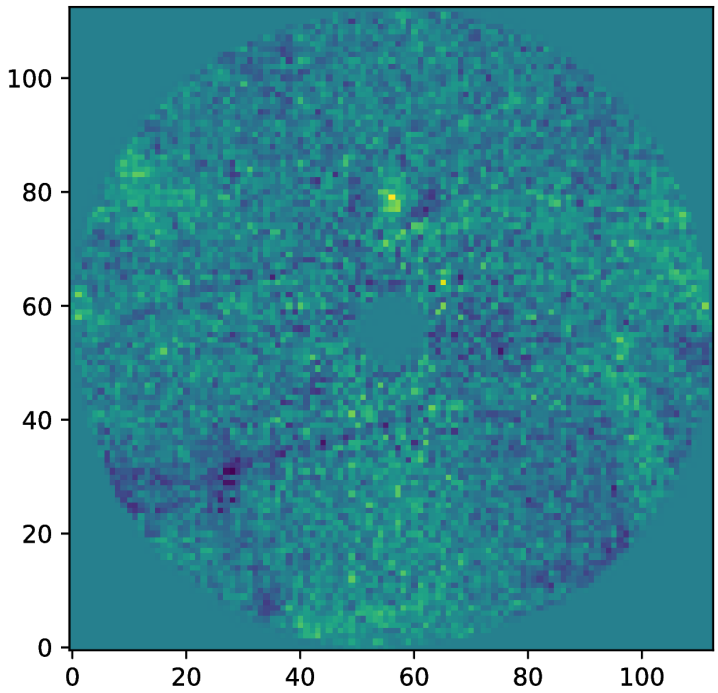
9.3 Results

The images that were exported to a FITS file can be visualized with a tool such as . We can also use the *Pypeline* functionalities to get the data from the database (without having to rerun the pipeline). For example, to get the residuals of the PSF subtraction:

```
data = pipeline.get_data('chopa_pca')
```

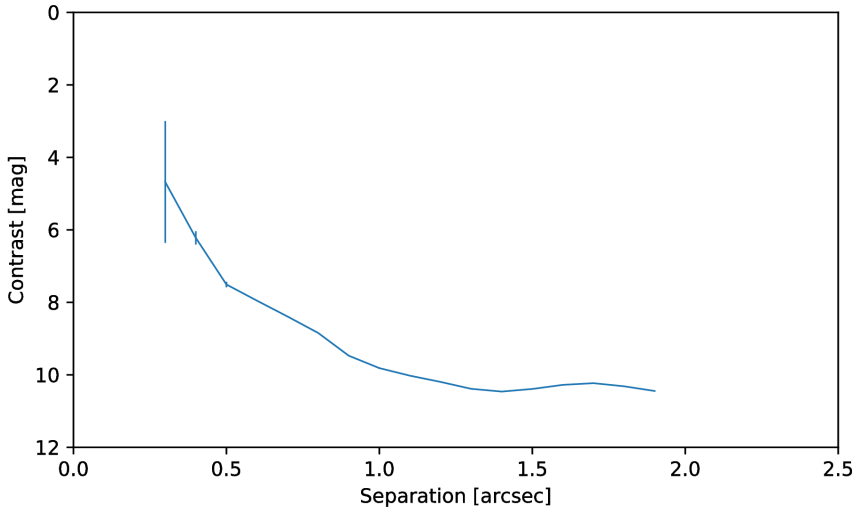
And to plot the residuals for 10 principal components (Python indexing starts at zero):

```
import matplotlib.pyplot as plt

plt.imshow(data[9, ], origin='lower')
plt.show()
```



Or to plot the detection limits with the error bars showing the variance of the six azimuthal positions that were tested:

```
data = pipeline.get_data('limits')

plt.figure(figsize=(7, 4))
plt.errorbar(data[:, 0], data[:, 1], data[:, 2])
plt.xlim(0., 2.5)
plt.ylim(12., 0.)
plt.xlabel('Separation [arcsec]')
plt.ylabel('Contrast [mag]')
plt.show()
```



CHAPTER 10

Python Guidelines

10.1 Getting Started

The modular architecture of PynPoint allows for easy implementation of new pipeline modules and we welcome contributions from users. Before writing a new PynPoint module, it is helpful to have a look at the *Architecture* section. In addition, some basic knowledge on Python is required and some understanding on the following items can be helpful:

- Python such as lists, tuples, and dictionaries.
- , in particular the concept of inheritance.
- as interfaces.

There are three different types of pipeline modules: *ReadingModule*, *WritingModule*, and *ProcessingModule*. The concept is similar for the three types of modules so here we will explain only how to code a processing module.

10.2 Conventions

Before we start writing a new PynPoint module, please take notice of the following style conventions:

- – style guide for Python code
- We recommend using and to analyze newly written code in order to keep PynPoint well structured, readable, and documented.
- Names of class member should start with `m_`.
- Images should ideally not be read from and written to the central database at once but in amounts of `MEMORY`.

Now we are ready to code!

CHAPTER 11

Coding a New Module

11.1 Class Constructor

First, we need to import the interface (i.e. abstract class) *ProcessingModule*:

```
from pynpoint.core.processing import ProcessingModule
```

All pipeline modules are classes which contain the parameters of the pipeline step, input ports and/or output ports. So let's create a simple `ExampleModule` class using the ProcessingModule interface (inheritance):

```
class ExampleModule(ProcessingModule):
```

When an IDE like PyCharm is used, a warning will appear that all abstract methods must be implemented in the `ExampleModule` class. The abstract class ProcessingModule has some abstract methods which have to be implemented by its children classes (e.g., `__init__()` and `run()`). Thus we have to implement the `__init__()` function (i.e., the constructor of our module):

```
def __init__(self,
             name_in="example",
             in_tag_1="in_tag_1",
             in_tag_2="in_tag_2",
             out_tag_1="out_tag_1",
             out_tag_2="out_tag_2",
             parameter_1=0,
             parameter_2="value"):
```

Each `__init__()` function of a *PypelineModule* requires a `name_in` argument (and default value) which is used by the pipeline to run individual modules by name. Furthermore, the input and output tags have to be defined which are used to to access data from the central database. The constructor starts with a call of the *ProcessingModule* interface:

```
super(ExampleModule, self).__init__(name_in)
```

Next, the input and output ports behind the database tags have to be defined:

```
self.m_in_port_1 = self.add_input_port(in_tag_1)
self.m_in_port_2 = self.add_input_port(in_tag_2)

self.m_out_port_1 = self.add_output_port(out_tag_1)
self.m_out_port_2 = self.add_output_port(out_tag_2)
```

Reading to and writing from the central database should always be done with the `add_input_port` and `add_output_port` functionalities and not by manually creating an instance of *InputPort* or *OutputPort*.

Finally, the module parameters should be saved to the `ExampleModule` instance:

```
self.m_parameter_1 = parameter_1
self.m_parameter_2 = parameter_2
```

That's it! The constructor of the `ExampleModule` is ready.

11.2 Run Method

We can now add the functionalities of the module in the `run()` method which will be called by the pipeline:

```
def run(self):
```

The input ports of the module are used to load data from the central database into the memory with slicing or the `get_all()` function:

```
data1 = self.m_in_port_1.get_all()
data2 = self.m_in_port_2[0:4]
```

We want to avoid using the `get_all()` function because data sets in 3–5 μm range typically consists of thousands of images. Therefore, loading all images at once in the computer memory might not be possible, in particular early in the data reduction chain when the images have their original size. Instead, it is recommended to use the `MEMORY` attribute that is specified in the configuration file.

Attributes of the input port are accessed in the following:

```
parang = self.m_in_port_1.get_attribute("PARANG")
pixscale = self.m_in_port_2.get_attribute("PIXSCALE")
```

And attributes of the central configuration are accessed through the *ConfigPort*:

```
memory = self._m_config_port.get_attribute("MEMORY")
cpu = self._m_config_port.get_attribute("CPU")
```

More information on importing of data can be found in the package documentation of *InputPort*.

Next, the processing steps are implemented:

```
result1 = 10.*self.m_parameter_1
result2 = 20.*self.m_parameter_1
result3 = [1, 2, 3]

attribute = self.m_parameter_2
```

The output ports are used to write the results to the central database:

```
self.m_out_port_1.set_all(result1)
self.m_out_port_1.append(result2)

self.m_out_port_2[0:2] = result2
self.m_out_port_2.add_attribute(name="new_attribute", value=attribute)
```

More information on storing of data can be found in the package documentation of *OutputPort*.

The attribute information has to be copied from the input port and history information has to be added. This step should be repeated for all the output ports:

```
self.m_out_port_1.copy_attributes(self.m_in_port_1)
self.m_out_port_1.add_history("ExampleModule", "history text")

self.m_out_port_2.copy_attributes(self.m_in_port_1)
self.m_out_port_2.add_history("ExampleModule", "history text")
```

Finally, the central database and all the open ports should be closed:

```
self.m_out_port_1.close_port()
```

Important: It is enough to close only one port because all other ports will be closed automatically.

Warning: It is not recommended to use the same tag name for the input and output port because that would only be possible when data is read and written at once with the `get_all()` and `set_all()` functionalities, respectively. Instead image should be read and written in amounts of `MEMORY` so an error should be raised when `in_tag=out_tag`.

11.3 Example Module

The full code for the `ExampleModule` from above is:

```
from pynpoint.core.processing import ProcessingModule

class ExampleModule(ProcessingModule):

    def __init__(self,
                 name_in="example",
                 in_tag_1="in_tag_1",
                 in_tag_2="in_tag_2",
                 out_tag_1="out_tag_1",
                 out_tag_2="out_tag_2”,
                 parameter_1=0,
                 parameter_2="value"):

        super(ExampleModule, self).__init__(name_in)

        self.m_in_port_1 = self.add_input_port(in_tag_1)
        self.m_in_port_2 = self.add_input_port(in_tag_2)

        self.m_out_port_1 = self.add_output_port(out_tag_1)
```

(continues on next page)

(continued from previous page)

```
        self.m_out_port_2 = self.add_output_port(out_tag_2)

        self.m_parameter_1 = parameter_1
        self.m_parameter_2 = parameter_2

    def run(self):

        data1 = self.m_in_port_1.get_all()
        data2 = self.m_in_port_2[0:4]

        parang = self.m_in_port_1.get_attribute("PARANG")
        pixscale = self.m_in_port_2.get_attribute("PIXSCALE")

        memory = self._m_config_port.get_attribute("MEMORY")
        cpu = self._m_config_port.get_attribute("CPU")

        result1 = 10.*self.m_parameter_1
        result2 = 20.*self.m_parameter_1
        result3 = [1, 2, 3]

        self.m_out_port_1.set_all(result1)
        self.m_out_port_1.append(result2)

        self.m_out_port_2[0:2] = result2
        self.m_out_port_2.add_attribute(name="new_attribute", value=attribute)

        self.m_out_port_1.copy_attributes(self.m_in_port_1)
        self.m_out_port_1.add_history("ExampleModule", "history text")

        self.m_out_port_2.copy_attributes(self.m_in_port_1)
        self.m_out_port_2.add_history("ExampleModule", "history text")

        self.m_out_port_1.close_port()
```

11.4 Apply Function To Images

A processing module often applies a specific method to each image of an input port. Therefore, the *apply_function_to_images()* function has been implemented to apply a function to all images of an input port. This function uses the `CPU` and `MEMORY` parameter from the configuration file to automatically process subsets of images in parallel. An example of the implementation can be found in the code of the bad pixel cleaning with a sigma filter: *BadPixelSigmaFilterModule*.

CHAPTER 12

Mailing List

The PynPoint mailing list is used to announce releases, new functionalities, pipeline modules, and other updates. The mailing list can be joined by sending a blank email to pynpoint-join@lists.phys.ethz.ch.

The mailing list can be consulted for suggestions and questions about PynPoint by sending an email to pynpoint@lists.phys.ethz.ch.

Further information about the mailing list can be found on the .

CHAPTER 13

Contributing

If you encounter problems when using PynPoint then please contact (see *Development Team* section). Bug reports and functionality requests can be provided by creating an on the Github page.

We also welcome active help with bug fixing and the development of new functionalities and pipeline modules. Please consider forking the Github repository and creating a for implementations that could be of interest for other users.

CHAPTER 14

About

14.1 Development Team

- Tomas Stolker <tomas.stolker@phys.ethz.ch>
- Markus Bonse <markus.bonse@stud.tu-darmstadt.de>
- Sascha Quanz <sascha.quanz@phys.ethz.ch>
- Adam Amara <adam.amara@phys.ethz.ch>

14.2 Attribution

If you use PynPoint in your publication then please cite Stolker et al. (2019). Please also cite Amara & Quanz (2012) as the origin of PynPoint, which focused initially on the use of principal component analysis (PCA) as a PSF subtraction method. In case you use specifically the PCA-based background subtraction module or the wavelet based speckle suppression module, please give credit to Hunziker et al. (2018) or Bonse, Quanz & Amara (2018), respectively.

14.3 Acknowledgements

We would like to thank several people who provided contributions and helped testing the package before its release:

- Anna Boehle (ETH Zurich)
- Alexander Bohn (Leiden University)
- Gabriele Cugno (ETH Zurich)
- Silvan Hunziker (ETH Zurich)

The PynPoint logo was designed by Atlas Infographics and is available for use in presentations.

Python Module Index

p

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W