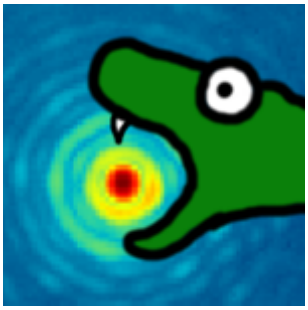
pyKLIP Documentation

Release 1.0

pyKLIP Developers

May 18, 2017

Setup



pyKLIP is a python library for direct imaging of exoplanets and disks. It uses an implmentation of KLIP and KLIP-FM to perform point spread function (PSF) subtraction. KLIP is based off of principal component analysis to model and subtract off the stellar PSF to look for faint exoplanets and disks and are around it.

pyKLIP is open source, BSD-licened, and available at this Bitbucket repo. You can use the issue tracker there to submit issues and all contributions are welcome!

CHAPTER 1

Features

- Capable of running ADI, SDI, ADI+SDI with spectral templates to optimize the PSF subtraction
- Library of KLIP-FM capabilties including forward-modelling a PSF, detection algorithms, and spectral extraction.
- A Forward Model Matched Filter of KLIP is available for GPI as well as post-processing planet detection algorithms.
- Parallelized with both a quick memory-intensive mode and a slower memory-lite mode
- Modularized to support data from multiple instruments. Currently there are interfaces to P1640, GPI, SPHERE, MagAO/VisAO, and Keck/NIRC2.
- If confused about what a function is doing, read the docstring for it. We have tried our best to document everything
- Version 1.1 - see *Release Notes* for update notes

CHAPTER 2

Bugs/Feature Requests

Please use the Issue Tracker on Bitbucket to submit bugs and new feature requests. Anyone is able to open issues.

CHAPTER 3

Attribution

The development of pyKLIP is led by Jason Wang with contributions made by Jonathan Aguilar, JB Ruffio, Rob de Rosa, Schuyler Wolff, Abhijith Rajan, Zack Briesemeister, Kate Follette, Maxwell Millar-Blanchaer, Alexandra Greenbaum, Simon Ko, Tom Esposito, Elijah Spiro, and Laurent Pueyo. If you use this code, please cite cite the Astrophysical Source Code Library record of it (ASCL or ADS)

Wang, J. J., Ruffio, J.-B., De Rosa, R. J., et al. 2015, Astrophysics Source Code Library, ascl:1506.001

CHAPTER 4

Contents

Installation

Dependencies

Before you install pyKLIP, you will need to install the following packages, which are useful for most astronomical data analysis situations anyways. The main pyKLIP code is cross-compatible with both python2.7 and python3.5.

- numpy
- scipy
- astropy
- Optional: matplotlib, mkl-service

For the optional packages, matplotlib is useful to actually plot the images. For mkl-service, pyKLIP autmoatically toggles off MKL parallelism during parallelized KLIP if the mkl-service package is installed. Otherwise, you will need to toggle them off yourself for optimal performance. See notes on parallelized performance below.

For *Bayesian KLIP-FM Astrometry (BKA)* specifically, you'll also want to install the following packages:

- emcee
- corner

As pyKLIP is computationally expensive, we recommend a powerful computer to optimize the computation. As direct imaging data comes in many different forms, we cannot say right here what the hardware requirements are for your data reduction needs. For data from the Gemini Planet Imager (GPI), a computer with 20+ GB of memory is optimal for an 1 hour sequence taken with the integral field spectrograph and reduced using ADI+SDI. For broadband polarimetry data from GPI, any laptop can reduce the data.

Install

Due to the continually developing nature of pyKLIP, we recommend you use the current version of the code on Bitbucket and keep it updated. To install the most up to date developer version, clone this repository if you haven't

already:

```
$ git clone git@bitbucket.org:pyKLIP/pyklip.git
```

This clones the repoistory using SSH authentication. If you get an authentication error, you will want to follow this guide to setup SSH authentication, or clone using the HTTPS option instead, which just requires a password.

Once the repository is cloned onto your computer, `cd` into it and run the setup file:

```
$ python setup.py develop
```

If you use multiple versions of python, you will need to run `setup.py` with each version of python (this should not apply to most people).

Note on parallelized performance

Due to the fact that numpy compiled with BLAS and MKL also parallelizes linear algebra routines across multiple cores, performance can actually sharply decrease when multiprocessing and BLAS/MKL both try to parallelize the KLIP math. If you are noticing your load averages greatly exceeding the number of threads/CPUs, try disabling the BLAS/MKL optimization when running pyKLIP.

To disable OpenBLAS, just set the following environment variable before running pyKLIP:

```
$ export OPENBLAS_NUM_THREADS=1
```

A recent update to anaconda included some MKL optimizations which may cause load averages to greatly exceed the number of threads specified in pyKLIP. As with the OpenBLAS optimizations, this can be avoided by setting the maximum number of threads the MKL-enabled processes can use:

```
$ export MKL_NUM_THREADS=1
```

As these optimizations may be useful for other python tasks, you may only want MKL_NUM_THREADS=1 only when pyKLIP is called, rather than on a system-wide level. By defaulf in `parallelized.py`, if `mkl-service` is installed, the original maximum number of threads for MKL is saved, and restored to its original value after pyKLIP has finished. You can also modify the number of threads MKL uses on a per-code basis by running the following piece of code (assuming `mkl-service` is installed):

```
import mkl
mkl.set_num_threads(1)
```

Release Notes

Version 1.1

- Updated installation to be much easier
- Reorganized repo structure to match standard python repos
- Improvements to automatic planet detection code

Version 1.0

- Initial Release
- Fully-functional KLIP implementation for ADI and SDI
- Interface for GPI data in both spectral and polarimetry mode

- Utility functions like fake injection and contrast calculation

Basic KLIP Tutorial with GPI

Here, we will explain how to run a simple PSF subtraciton using the KLIP algorithm in pyKLIP. If you are not familiar with KLIP, we suggest you first read the KLIP paper which describes the algorithm in detail. In this tutorial, we assume you are familiar with the terminology in KLIP. We will use GPI data to explain the process, but other than reading in the data, all the PSF subtraction steps are the same for any other dataset.

Reading in GPI Data

First, you'll need some reduced GPI datacubes to run KLIP one since pyKLIP does not reduce raw data. If you have raw GPI data you need to reduce, the GPI Data Reduction Pipeline Documentation page has all of the instructions and tutorials to reduce GPI data. After reducing the data, you should have a series of 3-D datacubes where the third dimension is either wavelength or polarization depending if you are working with spectral or polarimetric data respectively. Regardless, the data should have the satellite spot fluxes and locations measured and stored in the header as we will need these to register and calibrate the datacubes. If you don't have any GPI data or are simply too lazy to reduce some yourself, you can use the reduced Beta Pic datacubes from the GPI Public Data Release.

Once you have reduced some data, we need to identify and parse through the GPI data from GPI specific information to standardized information for pyKLIP

```
import glob
import pyklip.instruments.GPI as GPI

filelist = glob.glob("path/to/dataset/*.fits")
dataset = GPI.GPIData(filelist, highpass=True)
```

This returns `dataset`, an implementation of the abstract class *`pyklip.instruments.Instrument.Data`* with standardized fields that are needed to perform the KLIP subtraction, none of which are instrument specific. Please read the docstring for `pyklip.instruments.GPI.GPIData` to more information on the the fields for GPI data.

Note: If you get an error here, you likely did not reduce the raw GPI data correctly, so please check that the satellite spots were measured and stored in the header.

Note: When reading in the GPI data, the data are no longer automatically high-pass filtered. You should explictly high pass filter the data if desired (we find it is typically good for planet SNR using the optional keyword `highpass=True`. You can also apply the high-pass filter as pre-processing step before KLIP in *pyklip.parallelized.klip_dataset* if you don't want to do it here as it is slower.

Running KLIP

Next, we will perform the actual KLIP ADI+SDI subtraction. To take advantage of the easily parallelizable computation, we will use the *`pyklip.parallelized`* module to perform the KLIP subtraction, which uses the python `multiprocessing` library to parallelize the code

```
import pyklip.parallelized as parallelized

parallelized.klip_dataset(dataset, outputdir="path/to/save/dir/", fileprefix="myobject
↪",
                          annuli=9, subsections=4, movement=1, numbasis=[1,20,50,100],
                          calibrate_flux=True, mode="ADI+SDI")
```

This will save the processed KLIP images in the field `dataset.output` and as FITS files saved using the directory and fileprefix specified. The FITS files contain two different kinds of outputs. The first is a "KL-mode cube", a single 3D datacube where the z-axis is all the different KL mode cutoffs used to model the stellar PSF. Here is an example KL-mode cube using GPI public data on beta Pic, where the planet is quite visible.

The second is a series of spectral datacubes with the z-axis is wavelength and each datacube uses a different KL mode cutoff as specified by its filename. Here is an example of a 20 KL-mode cutoff cube using the same GPI data on beta Pic.

Picking KLIP Parameters for Point Sources

There are a lot of ways to tune the reduction, so check out the docstring of *pyklip.parallelized.klip_dataset()* for all the keywords you can use. Here, we have provided the keywords which we use the most and should be sufficient for most cases.

Geometry

We have divided the image into 9 annuli and each annuli into 4 sectors (which do not rotate with the sky) and run KLIP independently on each sector. Picking the geometry depends on the structure of the PSF, but we have found this to be pretty good for GPI data.

annuli_spacing

By default we break up the image into equal sized annuli (except for the last one that emcompasses the rest of the image), but sometimes we want smaller annuli closer in, since the stellar PSF changes rapidly there. In that case, we suggest setting `annuli_spacing="log"` so the widths of the annuli increases logarithmatically.

"Aggressiveness"

"Aggressiveness" is a key parameter to tune in the PSF subtraction. Increasing the aggressiveness of the PSF subtraction typically allows you to better model and subtract the stellar PSF. However, doing so also typically causes any astrophysical flux (e.g. planets, disks) to also be subtracted to a higher degree. Typically, there is a sweet spot that balances subtracting the stellar PSF and maintaining the signal of planets and disks. The aggressiveness of the subtraction is tuned via a combination of the "movement" or "minrot" parameters and "numbasis" keywords, as described below.

movement

In our exmaple, to build the reference library to build our principal components, we picked PSFs where any potential astrophysical source will have moved by 1 pixel due to ADI (azimuthal motion) and SDI (radial motion). Decreasing

this number increases the aggressiveness of the reduction as it will allow you to pick PSFs that are closer in time and wavelength. However, you will also suffer more self-subtraction of potential astrophysical sources. We find for GPI data, 1 pixel is good for maximizing the SNR of potential planets in the data.

numbasis

We don't pick just one KL basis cutoff for KLIP, but rather an array so we can play aroud with the optimal number. Increasing the number of KL modes also increases the aggressiveness of the reduction. For GPI data, we find between 20-50 KL modes for planet data and 1-10 KL modes for disk data is optimal. However, with both the `movement` and `numbasis` parameters, it requires a bit of searching to find the optimal configuration.

mode

The `mode` keyword specifies whether to use ADI, SDI, or both.

spectrum

A parameter not specified in this tutorial is the spectral template. Since we know exoplanet spectra should follow the models (at least roughly), we can use that to better choose reference PSFs to subtract out the stellar PSF. Currently, the only option is to optimze for T-dwarfs which have sharp methane absorption features. This can be turned on by setting `spectrum='methane'`. By doing this, in channels without methane absorption (i.e. where the planet signal is strong), we will use reference PSFs from channels where with methane abosrption (i.e. where the planet signal is weak). The aggressiveness of this is tuned with the `movement` keyword (i.e. by decreasing `movement`, we will allow into the reference PSFs images at wavelengths where the ratio of "no methane abospriton"/"some methane absorption" is smaller). When this keyword is set, we also do a weighted mean collapse in wavelength for the outputted KL-mode cubes.

Other

We have also choosen to flux calibrate the data to convert it into contrast units to work in more physical units.

Note: The `calibrate_flux` keyword does **not** correct for algorithm throughput, which is a loss of flux due to the PSF subtraction process. It merely provides the calibration to convert to contrast units. You will then need to correct for algorithm throughput by methods such as fake planet injection. See *Calibrating Algoirthm Throughput & Generating Contrast Curves* which explains how to do this in the context of contrast curves.

There are more parameters that can be tweaked. Read the docstring of *pyklip.parallelized.klip_dataset()* for the full details.

Picking KLIP Parameters for Disks

Using KLIP for disks can be difficult since the optimal parameters will depend on the geometry of the disk and the amount of field rotation in the sequence. Below, we describe some starting points for tuning the subtraction. Note that for disks it is suggested to only use mode="ADI" as SDI can severely distort the disk signal.

Geometry

PyKLIP splits divides the image into a number of annuli centered around the center of the image as defined by the `dataset.centers` attribute, and splits each of those annuli into a number of subsections, set by the `annuli` and `subsection` keywords, respectively. For disks, we find `subsections=1` to be effective. The number of annuli can also depend on the geometry of the disk, but we find that `annuli=1` is sufficient for most cases and produces smoother looking reductions.

Aggressiveness

The aggressiveness of a PSF subtraction is influenced by a number of parameters described below. There is often no one optimal aggressiveness, and there is much to be gained from both more aggressive and less aggressive reductions. A more aggressive reduction will allows you to probe features at closer inner working angles at the cost of killing fainter or more extended features. The aggressiveness and the parameters you choose can also be affected by the geometry and strength of the detection. Edge-on disks are more resilient to more aggressive reductions while face-on disks will need less aggressive reductions due to the self-subtraction associated with ADI.

Numbasis

Changing the number of basis vectors subtracted will show different sets of features. More basis vectors will self-subtract more of the extended PSF structure, showing features in closer inner working angles while subtracting fewer basis vectors will show more extended features of the disk.

Minrot

Given the structure of debris disks, it is preferable to use the minrot criterion to select basis vectors rather than the movement parameters as is used in psf subtraction. The choise for this paraeter will depend on the geometry. For thin disks, a smaller minrot is desireable as it will allow for a cleaner subtraction while thicker disks will require a larger minrot to avoid self-subtraction.

Project 1640 PyKLIP tutorial

Usage instructions for Project 1640 are similar to those for GPI with the exception that the grid spot positions must be found before KLIP can be run. Import the P1640 instrument class instead of the GPI instrument class.

The grid spots locations only need to be found once, after which they can be read from a file. Instructions for use are found below.

Overview

P1640 Instrument class and support code to interface with PyKLIP PSF subtraction.

Author: Jonathan Aguilar

The code here defines the instrument class for Project 1640 that interacts with the rest of the PyKLIP module. The Instrument class contains the information that is needed to scale and align the datacubes, and to select the reference slicess.

Dependencies

Required

- numpy
- scipy
- astropy
- python 2.7 or 3.4
- photutils #### Recommended (required to run the cube and spot verifier tools)
- matplotlib

Installing photutils

Instructions for installing photutils can be found here: http://photutils.readthedocs.io/en/latest/photutils/install.html. Note that the conda instructions may not work - in that case, you can try `conda install -c https://conda.anaconda.org/astropy photutils`

Steps

The general steps are:

1. Collect datacubes
2. Vet datacubes
3. Fit grid spots
4. Vet grid spots
5. Run KLIP

A set of tools built into PyKLIP makes this easier to do.

The trickiest part is setting up the grid spot fitting and making sure it succeeds. Once that's done, the grid spot positions can simply be read in from a file. This is described in more detail below.

TODO: Contrast curves and fake injections require unocculted cubes. Currently there is no way to hook these in. Yeah, I want it too. If you want it so bad, do it yourself.

Tutorial

Important This tutorial assumes you are inside the following directory:

`pyklip/pyklip/instruments/P1640_support/tutorial`

A couple datacubes (with all but the essential information stripped from them) are available by clicking this link here or from the command line with `wget https://sites.google.com/site/aguilarja/otherstuff/pyklip-tutorial-data/P1640_tutorial_data.tar.gz` Download the tarball and unpack the fits files into the `tutorial/data` folder with the command `tar -xvf P1640_tutorial_data.tar.gz`

Living On The Edge Version

If you trust me, you can do only steps "Collect the datacubes", "Fit the gridspots", and "Run KLIP". This skips visual inspection of the datacubes and spot fitting.

The P1640Data class *will* automatically check for the presence of the spot files and, if it doesn't find them, will attempt to do the fitting itself. You're then trusting that the fitting succeeds. It normally does, but generally I like to fit the grid spots first, visually inspect them, and then move on to the KLIP step. If you don't think you need to do this - or you already have done the grid spot fitting and vetting - then you can move right on to the Run KLIP step. Otherwise, proceed below to fit the grid spots.

Collect the datacubes

Easy-peasy.

```
:::python
    import glob
    filelist=glob.glob("*Occulted*fits")
```

Vet the datacubes

This uses the cube checker, a separate command-line tool that lets you quickly decide whether or not you should include a particular cube in your reduction.

Note: there is a new version called `P1640_cube_checker_interactive` that is way easier to use, replace `P1640_cube_checker` with this in the lines below if you want to use it. We have noticed that it can take a long time to load over ssh on Macs (for some reason this doesn't affect Linux). A workaround is to enable ssh compression with ssh -C.

From an IPython terminal, do: (the syntax here is weird because telling python to evaluate python variables)

```
:::python
    import sys
    sys.path.append("..")
    import P1640_cube_checker
    good_cubes = P1640_cube_checker.run_checker(filelist)
  or
    %run ../P1640_cube_checker.py --files {" ".join(filelist)}
```

Alternatively, from a bash terminal, do:

```
:::bash
    filelist=`ls data/*Occulted*fits`
    python ../P1640_cube_checker.py --files ${filelist}
```

An animation of each cube, along with observing conditions and a comparison to the other cubes in the set, will pop up and the terminal will prompt you Y/N to keep it in the "good cubes" list. These are the files that you will keep for KLIP. If you like the cube, press Y. If you don't, press N. All the Y's will be spit out in a copy-pasteable format at the end, and stored in memory (in this case, in the variable *good_cubes*). After you've looped through all the cubes, you'll be prompted to quit or re-inspect the cubes. If you're happy with your selection, go ahead and quit (Y), but if you want to revisit your choices, press N to restart the loop. You'll have redo all of your decisions.

Fit grid spots

Note: you should only need to do this once, after which you can just read in the grid spot positions from a file.

First, re-assemble your handy list of P1640 data.

Grid spots MUST exist, and (for now) they MUST be in the normal orientation. If this isn't true, then the code will hang.

In order to fit the spots, we need the P1640spots module:

```
:::python
    import sys
    sys.path.append("..")
    import P1640spots
    # if the variables below are not set, default values will be read from P1640.ini
    # for the tutorial, let's set them explicitly
    spot_filepath = 'shared_spot_folder/'
    spot_filesuffix = '-spot'
    spot_fileext = 'csv'
    for test_file in good_cubes:
        spot_positions = P1640spots.get_single_file_spot_positions(test_file, rotated_
↪spots=False)
        P1640spots.write_spots_to_file(test_file, spot_positions, spot_filepath,
                                       spotid=spot_filesuffix, ext=spot_fileext, ␣
↪overwrite=False)
```

(For now, only normally-oriented gridspots can be used, but in the future you should be able to set `rotated_spots=True` to fit 45deg-rotated grid spots).

The default values for the spot file filenames and directories (on Dnah at AMNH) can be found in the `P1640.ini` config file. I tend to write a separate config file specifically for the reduction and define them again there, with a custom directory if I want. An example reduction config file will eventually be added to the repo.

Vet grid spots

We can run `P1640_cube_checker` in "spots" mode to check the spots. Usage is similar to before except now you need to use the `--spots` flag and specify the location of the spot file folder.

From IPython, there are two ways:

```
:::python
    import sys
    sys.path.append("..")
    import P1640_cube_checker
    good_spots = P1640_cube_checker.run_spot_checker(good_cubes, spot_path='shared_
↪spot_folder/')
  or
    %run ../P1640_cube_checker.py --files {" ".join(good_cubes)} --spots --spot_path␣
↪shared_spot_folder/
```

From bash, do: (note: check the value of good_cubes before you pass it, make sure it got set properly)

```
:::bash
    good_cubes="copy names of vetted files here"
    python ../P1640_cube_checker --files ${good_cubes} --spots --spot_path shared_
↪spot_folder
```

Again, you will be prompted `Y/n` for each cube. Y = keep it, N = throw it out. At the end, you will be told all the files for which the spot fitting FAILED and for which it succeeded. For these files, you can either try to re-run the fitting, or (more likely) remove that cube from the datacubes that get sent to PyKLIP.

When running in python mode, the variable `good_spots` stores the file names for which you said the spot fitting succeeeded. These are the files which you will use to run KLIP, and can be used to initialize the P1640Data object (more below).

Run KLIP

Running KLIP on P1640 data is nearly identical to running it on GPI, with the exception that you have to be careful to only use cubes that have corresponding grid spot files. We'll start off by assuming that the variable `filelist` stores a list of the files that you want to include in your reduction (i.e. they passed all the vetting stages above).

```
:::python
    import sys
    sys.path.append("../../../../")
    import pyklip.instruments.P1640 as P1640
    dataset = P1640.P1640Data(filelist, spot_directory="shared_spot_folder/")
    import pyklip.parallelized as parallelized
    parallelized.klip_dataset(dataset, outputdir="output/", fileprefix="woohoo",␣
↪annuli=5, subsections=4, movement=3, numbasis=[1,20,100], calibrate_flux=False,␣
↪mode="SDI")
```

This will run the KLIP PSF subtraction algorithm. The resulting images are stored in the `dataset.output` field and written as FITS files to the output directory with the file prefix you provided. The P1640 output header format is that the first header stores the KLIP parameters, and the subsequent headers store copies of the headers from the original FITS files that were combined in this analysis. One file containing a datacube is written for each KL cutoff specified.

Calibrating Algoirthm Throughput & Generating Contrast Curves

Due to oversubtraction and selfsubtraction (see Pueyo (2016) for a good explaination), the shape, flux, and spectrum of the signal of a planet or disk is distoed by PSF subtraction. To calibrate algorithm throughput after KLIP in this tutorial, we will use the standard fake injection technique, which basically is injecting fake planets/disks in the data of known shape, flux, and spectrum to measure the algorithm throughput.

In this tutorial, we will calibrate the throughput of the previous exmaple (*Basic KLIP Tutorial with GPI*) for the purpose of generating a contrast curve. Note that this same general process can be used to character a planet or disk (e.g. measure astrometry and spectrum of an imaged exoplanet).

Contrast Curves

To measure the contrast (ignoring algorithm throughput), we use *`pyklip.klip.meas_contrast()`*, which assumes azimuthally symmetric noise and computes the 5σ noise level at each separation. It uses a Gaussian cross correlation to compute the noise as a small optimization to smooth out high frequency noise (since we know our planet is not going to be smaller than on λ/D scales). It also corrects for small number statistics (i.e. by assuming a Student's t-distribution rather than a Gaussian). This will give us a sense of the contrast to inject fake planets into the data (algorithm throughput is ~50%). We are calculating broadband contrast so we want to spectrally-collapsed data (if applicable). You can do this by reading back in the KL mode cube and picking a KL mode cutoff. (The KL mode cutoff is chosen to maximize planet SNR, which we won't discuss here, but can be determined with fake planet injection.)

Here we will show an example using the pyKLIP output of GPI data, and using KL modes.

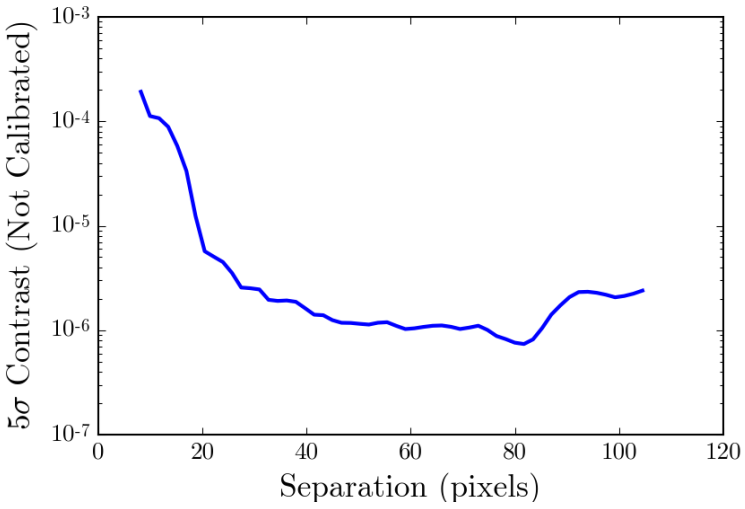
```
import astropy.io.fits as fits
hdulist = fits.open("myobject-KLmodes-all.fits")
# pick the 20 KL mode cutoff frame out of [1,20,50,100]
kl20frame = hdulist[1].data[1]
dataset_center = [klip_hdulist[1].header['PSFCENTX'], klip_hdulist[1].header['PSFCENTY
↪'] ]
```

Then, a convenient pyKLIP function will calculate the contrast, accounting for small sample statistics. We are picking 1.1 arcseconds as the outer boundary of our contrast curve. The *low_pass_filter* option specifies the size of the Gaussian to use in our cross correlation to smooth low frequency noise. It is typically smaller than the size of the PSF since self-subtraction from KLIP decreases the PSF size. We also need to specify the FWHM of the PSF in order to account for small sample statistics. It also determines the spacing the contrast curve returns. The function samples the noise with a spacing of FWHM/2.

For our example dataset on Beta Pic, we also need to mask out the planet, Beta Pic b, so that it doesn't bias our noise estimate.

```
import numpy as np
import pyklip.klip as klip
import pyklip.instruments.GPI as GPI

dataset_iwa = GPI.GPIData.fpm_diam['J']/2 # radius of occulter
dataset_owa = 1.5/GPI.GPIData.lenslet_scale # 1.5" is the furtherest out we will go
dataset_fwhm = 3.5 # fwhm of PSF roughly
low_pass_size = 1. # pixel, corresponds to the sigma of the Gaussian

# mask beta Pic b
# first get the location of the planet from Wang+ (2016)
betapicb_sep = 30.11 # pixels
betapicb_pa = 212.2 # degrees
betapicb_x = betapicb_sep * -np.sin(np.radians(betapicb_pa)) + dataset_center[0]
betapicb_y = betapicb_sep * np.cos(np.radians(betapicb_pa)) + dataset_center[1]
# now mask the data
ydat, xdat = np.indices(kl20frame.shape)
distance_from_planet = np.sqrt((xdat - betapicb_x)**2 + (ydat - betapicb_y)**2)
kl20frame[np.where(distance_from_planet <= 2*dataset_fwhm)] = np.nan


contrast_seps, contrast = klip.meas_contrast(kl20frame, dataset_iwa, dataset_owa,␣
↪dataset_fwhm, center=dataset_center, low_pass_filter=low_pass_size)
```

Now we can plot what the contrast curve (missing a calibration for algorithm throughput) looks like.



Injecting Fake Planets

KLIP naturally subtracts out planet flux due to over-subtraction and self-subtraction. To calibrate our sensitivity to planets, we need to inject some fake planets at known brightness into our data to calibrate KLIP attenuation. In this tutorial, we only only inject a few fakes once into the data just to demonstrate the technique with pyKLIP. For your data, it is suggested you inject many planets to explore the attenuation factor as a function of brightness, separation, and KLIP parameters (more aggressive reductions increase attenuation of flux due to KLIP).Fake planets are free, so the more the merrier!

First, let's read in the data again. This is the same dataset as you read in to run KLIP the first time.

```
import glob

filelist = glob.glob("path/to/dataset/*.fits")
dataset = GPI.GPIData(filelist, highpass=True)
```

Now we'll inject 12 fake planets in each cube. We'll do this one fake planet at a time using *pyklip.fakes.inject_planet()*. As we get further out in the image, we will inject fainter planets, since the throughput does vary with planet flux, so we want the fake planets to be just around the detection threshold (slightly above is preferably to reduce noise). Since we specify a fake planet's location by it's separation and position angle, we need to know the orientation of the sky on the image using the frames' WCS headers. The planets also are injected in raw data units, we need to convert the planet flux from contrast to DN for GPI. For other instruments, each should have its flux calibration and thus own method to convert between data units and contrast.

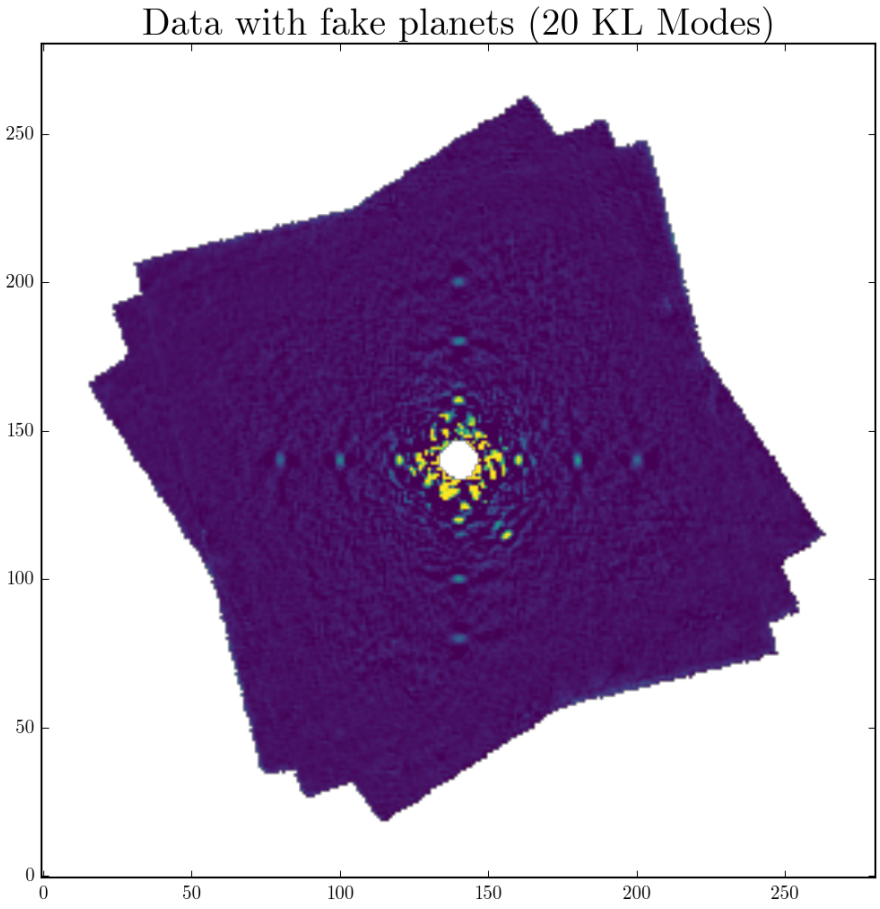
```
import pyklip.fakes as fakes

# three sets, planets get fainter as contrast gets better further out
input_planet_fluxes = [1e-4, 1e-5, 5e-6]
seps = [20, 40, 60]
fwhm = 3.5 # pixels, approximate for GPI

for input_planet_flux, sep in zip(input_planet_fluxes, seps):
    # inject 4 planets at that separation to improve noise
    # fake planets are injected in data number, not contrast units, so we need to␣
↪convert the flux
    # for GPI, a convenient field dn_per_contrast can be used to convert the planet␣
↪flux to raw data numbers
```

```
    injected_flux = input_planet_flux * dataset.dn_per_contrast
    for pa in [0, 90, 180, 270]:
        fakes.inject_planet(dataset.input, dataset.centers, injected_flux, dataset.
↪wcs, sep, pa, fwhm=fwhm)
```

Now we'll run KLIP using the example same parameters on this dataset with fake planets.

```
import pyklip.parallelized as parallelized

parallelized.klip_dataset(dataset, outputdir="path/to/save/dir/", fileprefix=
↪"myobject-withfakes",
                          annuli=9, subsections=4, movement=1, numbasis=[1,20,50,100],
                          calibrate_flux=True, mode="ADI+SDI")
```

Now, the resulting KLIP dataset should have 12 more planets in it! For the Beta Pic dataset, we actually have 13 planets ;).



We now will read in the output of the KLIP reducation with fake planets. Since we're using the 20 KL mode cutoff frame for our contrast curve, we want the same cutoff for our reduction with fake planets.

```
kl_hdulist = fits.open("myobject-withfakes-KLmodes-all.fits")
dat_with_fakes = kl_hdulist[1].data[1]
dat_with_fakes_centers = [kl_hdulist[1].header['PSFCENTX'], kl_hdulist[1].header[
↪'PSFCENTY'] ]
```

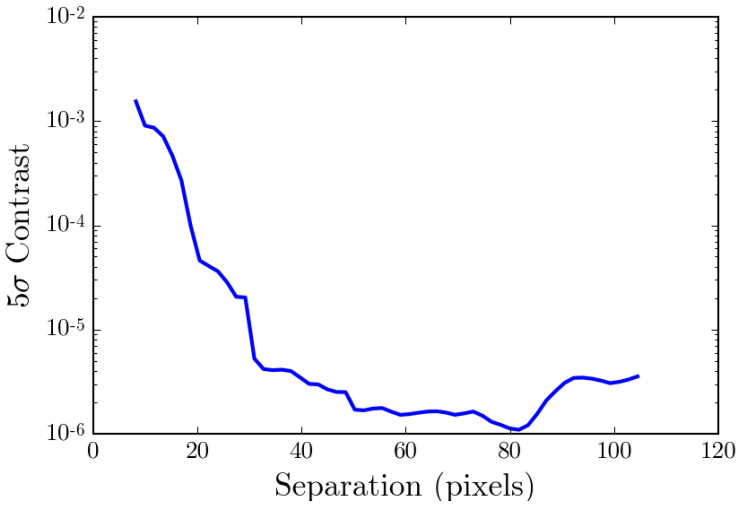
We will measure the flux of each fake in the reduced image using *pyklip.fakes.retrieve_planet_flux()*. Our strategy here is to assume the throughput is constant azimuthally, and for each 4 planets at a separation, average their fluxes together to reduce noise. Note that we need to again specify a WCS header to tell the code where to look for the planet in the image. You can grab that from the header of the reduced image, or we will be lazy here are use the `dataset.wcs` field from our fake dataset, which automatically gets rotated after KLIP.

```
retrieved_fluxes = [] # will be populated, one for each separation

for input_planet_flux, sep in zip(input_planet_fluxes, seps):
    fake_planet_fluxes = []
    for pa in [0, 90, 270, 360]:
        fake_flux = fakes.retrieve_planet_flux(dat_with_fakes, dat_with_fakes_centers,
↪ dataset.wcs[0], sep, pa, searchrad=7)
        fake_planet_fluxes.append(fake_flux)
    retrieved_fluxes.append(np.mean(fake_planet_fluxes))
```

Now we can calibrate the contrast curves. We know what flux level we injected the planets into the data at. We now have measured the flux value of the planets at each separation, so we can now calculate the "algorithm throughput" which measures how much KLIP attenuates flux. Then for each location on the contrast curve, we will just use the closest fake planet injection separation to assume an algorithm throughput correction. This is why it is good in practice in inject fakes in as many places as possible, so that the fake planets better model the algorithm throughput at each separation.

```
# fake planet output / fake planet input = throughput of KLIP
algo_throughput = np.array(retrieved_fluxes)/np.array(input_planet_fluxes) # a number␣
↪less than 1 probably

corrected_contrast_curve = np.copy(contrast)
for i, sep in enumerate(contrast_seps):
    closest_throughput_index = np.argmin(np.abs(sep - seps))
    corrected_contrast_curve[i] /= algo_throughput[closest_throughput_index]
```

Finally, we get a calibrated contrast curve!



Bayesian KLIP-FM Astrometry (BKA)

This tutorial will provide the necessary steps to run the Bayesian KLIP-FM Astorometry technique (BKA) that is described in Wang et al. (2016) to obtain one milliarcsecond astrometry on β Pictoris b.

Why BKA?

Astrometry of directly imaged exoplanets is challenging since PSF subtraction algorithms (like pyKLIP) distort the PSF of the planet. Pueyo (2016) provide a technique to forward model the PSF of a planet through KLIP. Taking this forward model, you could fit it to the data with MCMC, but you would underestimate your errors because the noise in direct imaging data is correlated (i.e. each pixel is not independent). To account for the correlated nature of the noise, we use Gaussian process regression to model and account for the correlated nature of the noise. This allows us to obtain accurate astrometry and accurate uncertainties on the astrometry.

BKA Requirements

To run BKA, you need the additional packages installed, which should be available readily:

- emcee
- corner

You also need the following pieces of data to forward model the data.

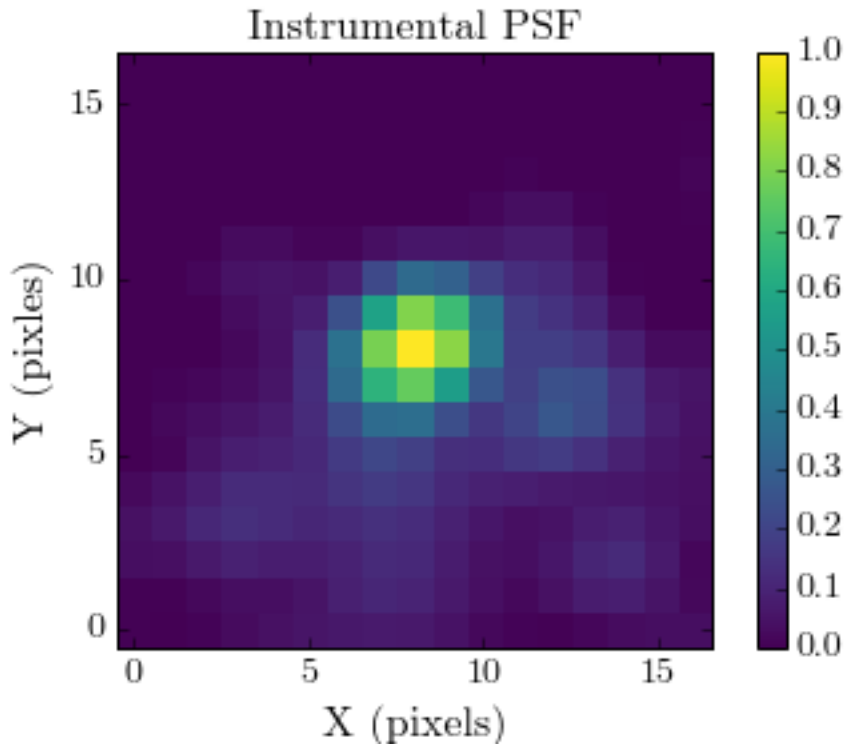
- Data to run PSF subtraction on
- A model or data of the instrumental PSF
- A good guess of the position of the planet (a center of light centroid routine should get the astrometry to a pixel)
- For IFS data, an estimate of the spectrum of the planet (it does not need to be very accurate, and 20% errors are fine)

Generating instrumental PSFs for GPI

A quick aside for GPI spectral mode data, here is how to generate the instrumental PSF from the satellite spots.

```
import glob
import numpy as np
import pyklip.instruments.GPI as GPI

# read in the data into a dataset
filelist = glob.glob("path/to/dataset/*.fits")
dataset = GPI.GPIData(filelist)

# generate instrumental PSF
boxsize = 25 # we want a 25x25 pixel box centered on the instrumental PSF
dataset.generate_psfs(boxrad=boxsize//2) # this function extracts the satellite spots␣
↪from the data
# now dataset.psfs contains a 37x25x25 spectral cube with the instrumental PSF
# normalize the instrumental PSF so the peak flux is unity
dataset.psfs /= (np.mean(dataset.spot_flux.reshape([dataset.spot_flux.shape[0] / 37,␣
↪37]),
                         axis=0)[:, None, None])
```

Here is an exmaple using three datacubes from the publicly available GPI data on beta Pic. Note that the wings of the PSF are somewhat noisy, due to the fact the speckle noise in J-band is high near the satellite spots. However, this should still give us an acceptable instrumental PSF.



Forward Modelling the PSF with KLIP-FM

With an estimate of the planet position, the instrumental PSF, and, if applicable, an estimate of the spectrum, we can use the *pyklip.fm* implementation of KLIP-FM and *pyklip.fmlib.fmpsf.FMPlanetPSF* extension to forward model the PSF of a planet through KLIP.

First, let us initalize *pyklip.fmlib.fmpsf.FMPlanetPSF* to forward model the planet in our data.

For GPI, we are using normalized copies of the satellite spots as our input PSFs, and because of that, we need to pass in a flux conversion value, `dn_per_contrast`, that allows us to scale our `guessflux` in contrast units to data units. If you are not using normalized PSFs, `dn_per_contrast` should be the factor that scales your input PSF to the flux of the unocculted star. If your input PSF is already scaled to the flux of the stellar PSF, `dn_per_contrast` is optional and should not actually be passed into the function.

```
# setup FM guesses
# You should change these to be suited to your data!
numbasis = np.array([1, 7, 100]) # KL basis cutoffs you want to try
guesssep = 30.1 # estimate of separation in pixels
guesspa = 212.2 # estimate of position angle, in degrees
guessflux = 5e-5 # estimated contrast
dn_per_contrast = your_flux_conversion # factor to scale PSF to star PSF. For GPI,␣
↪this is dataset.dn_per_contrast
guessspec = your_spectrum # should be 1-D array with number of elements = np.size(np.
↪unique(dataset.wvs))

# initialize the FM Planet PSF class
```

```
import pyklip.fmlib.fmpsf as fmpsf
fm_class = fmpsf.FMPlanetPSF(dataset.input.shape, numbasis, guesssep, guesspa,␣
↪guessflux, dataset.psfs,
                             np.unique(dataset.wvs), dn_per_contrast, star_spt='A6',
                             spectrallib=[guessspec])
```

Note: When executing the initializing of FMPlanetPSF, you will get a warning along the lines of "The coefficients of the spline returned have been computed as the minimal norm least-squares solution of a (numerically) rank deficient system." This is completeness normal and expected, and should not be an issue.

Next we will run KLIP-FM with the *pyklip.fm* module. Before we run it, we will need to pick our PSF subtraction parameters (see the *Basic KLIP Tutorial with GPI* for more details on picking KLIP parameters). For our zones, we will run KLIP only on one zone: an annulus centered on the guessed location of the planet with a width of 30 pixels. The width just needs to be big enough that you see the entire planet PSF.

```
# PSF subtraction parameters
# You should change these to be suited to your data!
outputdir = "." # where to write the output files
prefix = "betpic-131210-j-fmpsf" # fileprefix for the output files
annulus_bounds = [[guesssep-15, guesssep+15]] # one annulus centered on the planet
subsections = 1 # we are not breaking up the annulus
padding = 0 # we are not padding our zones
movement = 4 # we are using an conservative exclusion criteria of 4 pixels

# run KLIP-FM
import pyklip.fm as fm
fm.klip_dataset(dataset, fm_class, outputdir=outputdir, fileprefix=prefix,␣
↪numbasis=numbasis,
                annuli=annulus_bounds, subsections=subsections, padding=padding,␣
↪movement=movement)
```

This will now run KLIP-FM, producing both a PSF subtracted image of the data and a forward-modelled PSF of the planet at the gussed location of the planet. The PSF subtracted image as the "-klipped-" string in its filename, while the forward-modelled planet PSF has the "-fmpsf-" string in its filename.

Fitting the Astrometry using MCMC and Gaussian Processes

Now that we have the forward-modeled PSF and the data, we can fit them in a Bayesian framework using Gaussian processes to model the correlated noise and MCMC to sample the posterior distribution.

First, let's read in the data from our previous forward modelling. We will take the collapsed KL mode cubes, and select the KL mode cutoff we want to use. For the example, we will use 7 KL modes to model and subtract off the stellar PSF.

```
import os
import astropy.io.fits as fits
# read in outputs
output_prefix = os.path.join(outputdir, prefix)
fm_hdu = fits.open(output_prefix + "-fmpsf-KLmodes-all.fits")
data_hdu = fits.open(output_prefix + "-klipped-KLmodes-all.fits")

# get FM frame, use KL=7
fm_frame = fm_hdu[1].data[1]
fm_centx = fm_hdu[1].header['PSFCENTX']
```

```
fm_centy = fm_hdu[1].header['PSFCENTY']

# get data_stamp frame, use KL=7
data_frame = data_hdu[1].data[1]
data_centx = data_hdu[1].header["PSFCENTX"]
data_centy = data_hdu[1].header["PSFCENTY"]

# get initial guesses
guesssep = fm_hdu[0].header['FM_SEP']
guesspa = fm_hdu[0].header['FM_PA']
```

We will generate a *pyklip.fitpsf.FMAstrometry* object that we handle all of the fitting processes. The first thing we will do is create this object, and feed it in the data and forward model. It will use them to generate stamps of the data and forward model which can be accessed using `fma.data_stmap` and `fma.fm_stamp` respectively. When reading in the data, it will also generate a noise map for the data stamp by computing the standard deviation around an annulus, with the planet masked out

```
import pyklip.fitpsf as fitpsf
# create FM Astrometry object
fma = fitpsf.FMAstrometry(guesssep, guesspa, 13)

# generate FM stamp
# padding should be greater than 0 so we don't run into interpolation problems
fma.generate_fm_stamp(fm_frame, [fm_centx, fm_centy], padding=5)

# generate data_stamp stamp
# not that dr=4 means we are using a 4 pixel wide annulus to sample the noise for␣
↪each pixel
# exclusion_radius excludes all pixels less than that distance from the estimated␣
↪location of the planet
fma.generate_data_stamp(data_frame, [data_centx, data_centy], dr=4, exclusion_
↪radius=10)
```

Next we need to choose the Gaussian process kernel. We currently only support the Matern (ν=3/2) and square exponential kernel, so we will pick the Matern kernel here. Note that there is the option to add a diagonal (i.e. read/photon noise) term to the kernel, but we have chosen not to use it in this example. If you are not dominated by speckle noise (i.e. around fainter stars or further out from the star), you should enable the read noies term.

```
# set kernel, no read noise
corr_len_guess = 3.
corr_len_label = r"$l$"
fma.set_kernel("matern32", [corr_len_guess], [corr_len_label])
```

Now we need to set bounds on our priors for our MCMC. We are going to be simple and use uninformative priors. The priors in the x/y posible will be flat in linear space, and the priors on the flux scaling and kernel parameters will be flat in log space, since they are scale paramters. In the following function below, we will set the boundaries of the priors. The first two values are for x/y and they basically say how far away (in pixels) from the guessed position of the planet can the chains wander. For the rest of the parameters, the values say how many ordres of magnitude can the chains go from the guessed value (e.g. a value of 1 means we allow a factor of 10 variation in the value).

```
# set bounds
x_range = 1.5 # pixels
y_range = 1.5 # pixels
flux_range = 1. # flux can vary by an order of magnitude
corr_len_range = 1. # between 0.3 and 30
fma.set_bounds(x_range, y_range, flux_range, [corr_len_range])
```

Finally, we are set to run the MCMC sampler (using the emcee package). Here we have provided a wrapper to already set up the likelihood and prior. All we want to do is specify the number of walkers, number of steps each walker takes, and the number of production steps the walkers take. We also can specify the number of threads to use. If you have not turned BLAS and MKL off, you probably only want one or a few threads, as MKL/BLAS automatically parallelizes the likelihood calculation, and trying to parallelize on top of that just creates extra overhead.

```
# run MCMC fit
fma.fit_astrometry(nwalkers=100, nburn=200, nsteps=800, numthreads=1)
```

When the MCMC finishes running, we have our answer for the location of the planet in the data. Here are some fields to access this information:

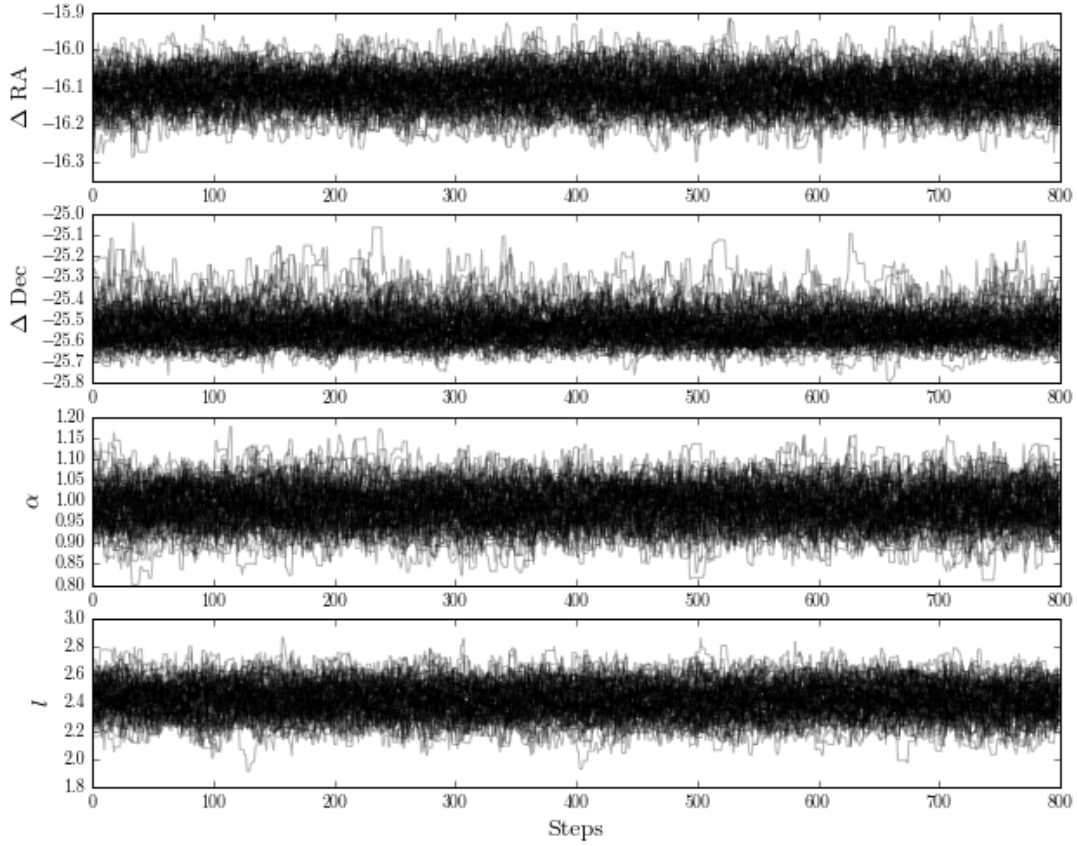
- `fma.RA_offset`: RA offset of the planet from the star as determined by the median of the marginalized posterior
- `fma.Dec_offset`: Dec offset of the plnaet from the star as determined by the median of the marginalized posterior
- `fma.RA_offset_1sigma`: 16th and 84th percentile values for the RA offset of the planet
- `fma.Dec_offset_1sigma`: 16th and 84th percentile values for the Dec offset of the planet
- `fma.flux`, `fma.flux_1sigma`: same thing except for the flux of the planet
- `fma.covar_param_bestfits`, `fma.covar_param_1sigma`: same thing for the hyperparameters on the Gaussian process kernel. These are both kept in a list with length equal to the number of hyperparameters.
- `fma.sampler`: this is the `emcee.EnsembleSampler` object which contains the full chains and other MCMC fitting information

The RA offset and Dec offset are what we are interested in for the purposes of astrometry. The flux scaling paramter (α) and the correlation length (l) are hyperparameters we marginalize over. First, we want to check to make sure all of our chains have converged by plotting them. As long as they have settled down (no large scale movements), then the chains have probably converged.

```
import matplotlib.pylab as plt
fig = plt.figure(figsize=(10,8))

# grab the chains from the sampler
chain = fma.sampler.chain

# plot RA offset
ax1 = fig.add_subplot(411)
ax1.plot(chain[:,:,0].T, '-', color='k', alpha=0.3)
ax1.set_xlabel("Steps")
ax1.set_ylabel(r"$\Delta$ RA")

# plot Dec offset
ax2 = fig.add_subplot(412)
ax2.plot(chain[:,:,1].T, '-', color='k', alpha=0.3)
ax2.set_xlabel("Steps")
ax2.set_ylabel(r"$\Delta$ Dec")

# plot flux scaling
ax3 = fig.add_subplot(413)
ax3.plot(chain[:,:,2].T, '-', color='k', alpha=0.3)
ax3.set_xlabel("Steps")
ax3.set_ylabel(r"$\alpha$")

# plot hyperparameters.. we only have one for this example: the correlation length
```

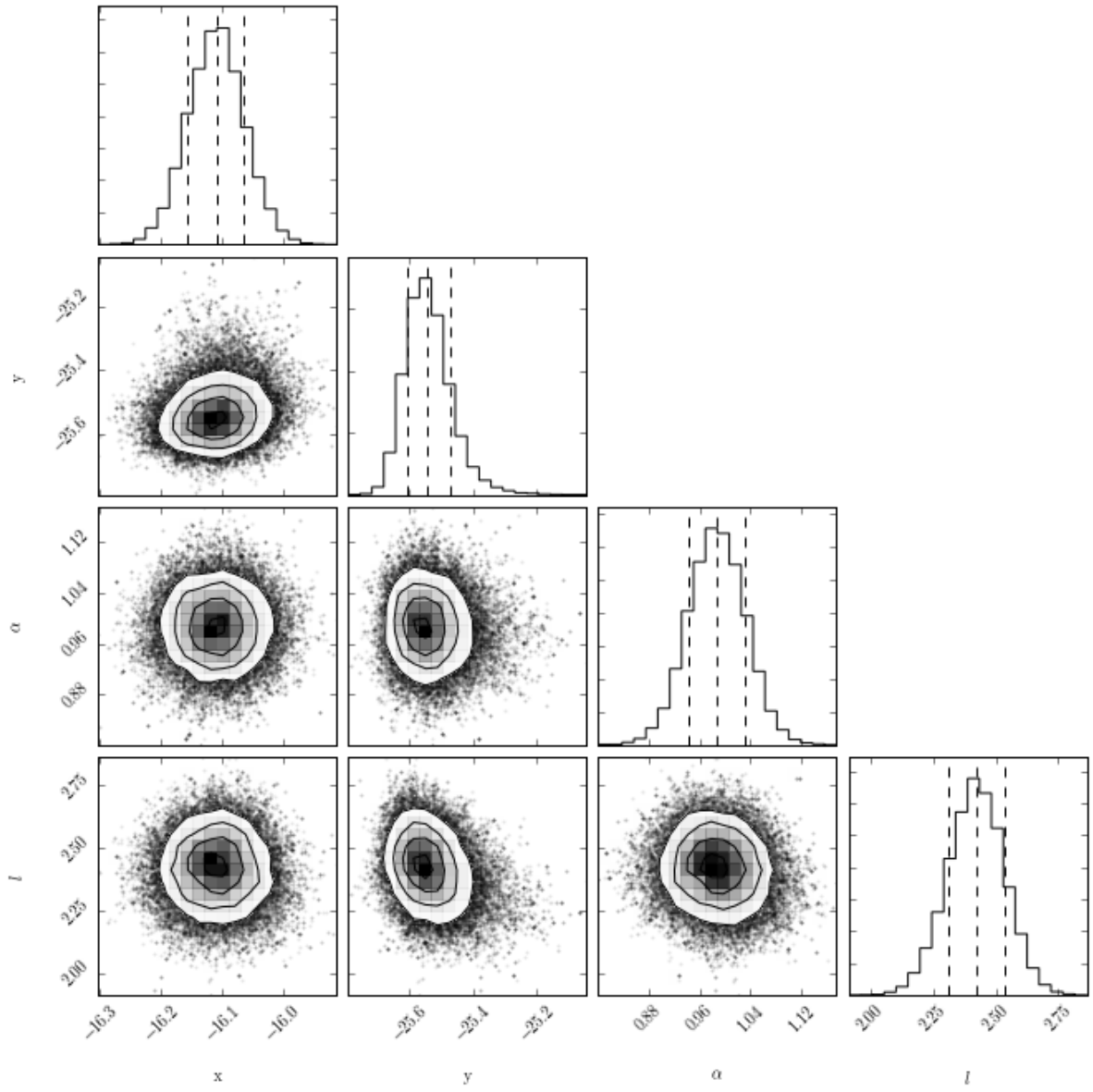
```
ax4 = fig.add_subplot(414)
ax4.plot(chain[:,:,3].T, '-', color='k', alpha=0.3)
ax4.set_xlabel("Steps")
ax4.set_ylabel(r"$l$")
```

Here is an example using three cubes of public GPI data on beta Pic.



We can also plot the corner plot to look at our posterior distribution and correlation between parameters.

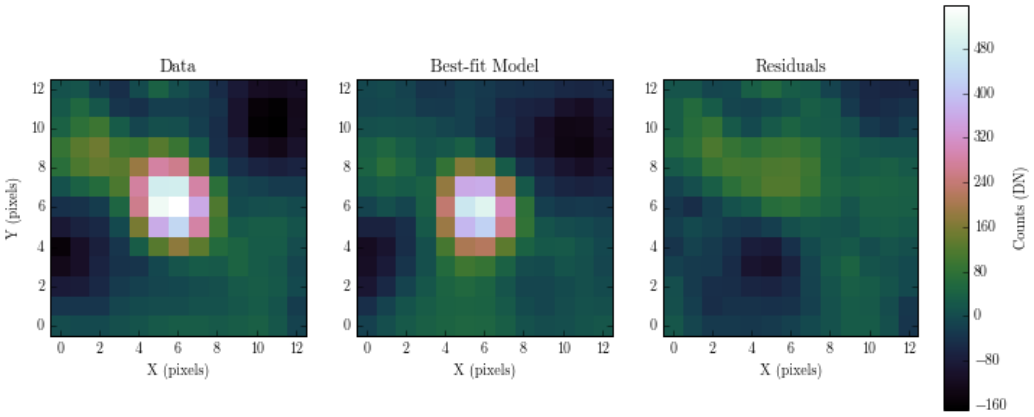
```
fig = plt.figure()
fig = fma.make_corner_plot(fig=fig)
```



Hopefully the corner plot does not contain too much structure (the posteriors should be roughly Gaussian). In the example figure from three cubes of GPI data on beta Pic, the residual speckle noise has not been very whitened, so there is some asymmetry in the posterior, which represents the local strucutre of the speckle noise. These posteriors should become more Gaussian as we add more data and whiten the speckle noise. And finally, we can plot the visual comparison of our data, best fitting model, and residuals to the fit.

```
fig = plt.figure()
fig = fma.best_fit_and_residuals(fig=fig)
```

And here is the example from the three frames of beta Pic b J-band GPI data:



The data and best fit model should look pretty close, and the residuals hopefully do not show any obvious strcuture that was missed in the fit. The residual ampltidue should also be consistent with noise. If that is the case, we can use the best fit values for the astrometry of this epoch. Remember that the 1-sigma values given here are just the statistical uncertainity on the location of the planet. You will need to include more uncertainties such as the location of the star and astrometric calibration uncertainties to obtain your full astrometric error bar. The flux values should in theory measure the flux of the planet, but that is out of the scope of this tutorial. Here, we print out our confidence on just the location of the planet in the image.

```
print("Planet RA offset is at {0} with a 1-sigma range of {1}".format(fma.RA_offset,␣
↪fma.RA_offset_1sigma))
print("Planet Dec offset is at {0} with a 1-sigma range of {1}".format(fma.Dec_offset,
↪ fma.Dec_offset_1sigma))
```

Forward Model Matched Filter (FMMF) Tutorial with GPI

The Forward Model Matched Filter (FMMF) is an algorithm aimed at improved exo-planet detection for direct imaging. The current implementation only works for GPI and this tutorial will explain how to use it. A reference paper J.-B. Ruffio et al. 2016/2017 with the description of the method is currently in preparation.

Note: If you ask me enough, I will try to make the code more general to work for different instruments.

Input Data

What are the input data needed to run the FMMF pyklip implementation.

Running FMMF

How to run the code.

Disk Foward Modelling Tutorial with GPI

Disk forward modelling is intended for use in cases where you would like to model a variety of different model disks on the same dataset. This can be used with an MCMC that is fitting for model parameters. It works by saving the KLIP basis vectors in a file so they do not have to be recomputed every time.

Running

How to use:

```
import glob
import pyklip.parallelized.GPI as GPI
from pyklip.fmlib.diskfm import DiskFM
import pyklip.fm as FM

filelist = glob.glob("path/to/dataset/*.fits")
dataset = GPI.GPIData(filelist)
model = [some 2D image array]
```

For a single run:

```
diskobj = DiskFM([n_files, data_xshape, data_yshape], numbasis, dataset, model_disk,␣
↪annuli = 2, subsections = 1)
```

If you would like to forward model multiple models on a dataset, then you will save the eigenvalues and eigenvectors:

```
diskobj = DiskFM([n_files, data_xshape, data_yshape], numbasis, dataset, model_disk,␣
↪annuli = 2, subsections = 1, basis_file_name = 'klip-basis.p', save_basis = True,␣
↪load_from_basis = False)
```

In both cases you then run:

```
fmout = fm.klip_dataset(dataset, diskobj, numbasis = numbasis,
annuli = 2, subsections = 1, mode = 'ADI')
```

Note that in the case that annuli = 1, you will need to set padding = 0 in klip_dataset

In order to forward model another disk:

```
diskobj = DiskFM([n_files, data_xshape, data_yshape], numbasis, dataset, model_disk,␣
↪annuli = 2, subsections = 1, basis_file_name = 'klip-basis.p', load_from_basis =␣
↪True)
diskobj.update_disk(newmodel)
fmout = diskobj.fm_parallelized()
```

Current Works in Progress

- Does not support SDI mode
- Is not parallized

Developing for pyKLIP

Docker

One very useful tool to have is a local build environment of the pyKLIP package for testing and validation purposes. We will be using a software container platform called Docker and this tutorial will provide a brief overview on what it is, how to set it up, and how to use it for pyKLIP.

Here you will find everything you need to know about Docker for pyKLIP.

Setup

Here you will learn how to install and setup your Docker.

Installation

We will be using the community edition of Docker.

For Ubuntu Linux

```
sudo apt-get update
sudo apt-get install docker-ce
```

For all other OSes, installation instructions and requirements can be found here.

Setup

From a fresh install, there are a few steps to getting your container up and running.

1. Download and run the pyKLIP image. You can do this by pulling and running, but simply running the pyKLIP image will do both steps in one. Executing the run command will first check your local machine for the appropriate images and use them if Docker finds them, or download them from Docker Hub if it fails. For now we'll start with the pull command:

```
$ docker pull simonko/pyklip
```

2. From here, to check if the appropriate image has been set up use the `docker images` command, and you should get something similar to the following

```
$ docker images

REPOSITORY              TAG                 IMAGE ID            CREATED                 ␣
↪SIZE
simonko/pyklip          latest              e9a584c685bb        4 hours ago          2.
↪37 GB
```

3. Running the command below creates a container of the pyklip image and gives us an interactive shell to interact with container. The `-i -t` flags allows for interactive mode and allocates a pseudo-tty for the container respectively. This is usually combined into the flag `-it`. If you don't specify a tag, it'll generate some random name for you. (ex. sad_lovelace, agitated_saha, ecstatic_pare, etc)

```
$ docker run -it simonko/pyklip:latest /bin/bash
```

4. When you're done with the container, simply type `exit` and your session will end. If you get the message that states there is a process running, simply type `exit` again and it'll exit the session. 5. After you've made your container you should be able to see it with

```
$ docker ps -a

CONTAINER ID        IMAGE                    COMMAND                  CREATED
↪    STATUS                      PORTS               NAMES
c6695e4d9a63        simonko/pyklip:latest    "/usr/bin/tini -- ..."   6 seconds ago
↪    Exited (0) 3 seconds ago                        zealous_goldwasser
```

6. To get into the container, you have to first start the container again, then use the attach command to get back into the interactive shell.

```
$ docker start <container name>
$ docker attach <container name>
```

For a very basic tutorial on Docker and how to use it, check out the Docker docs and tutorials here. There are a lot of helpful tutorials and information there.

Working With Docker

Here you will learn how some basics on working with Docker.

Using Local Files

Once you have your image, you can *cp* over local files into the container. To do this you have to use the `attach` command and `-d` flag like so

```
$ docker run -it -d simonko/pyklip:latest

exit

$ docker cp <source file/directory> <container name>:<destination>

$ docker start <container name>

$ docker attach <container name>
```

It should be noted that if the specified destination does not exist, it will create the destination for you. For example if I were to do the following

```
$ docker cp <somefile/directory> zealous_goldwasser:/pyklip
```

inside the *zealous_goldwasser* container and it did not already have a pyklip directory, docker would create the directory for me and place the file in it, just like the normal *cp* command.

Deleting Images and Containers

You may find that your docker is getting a bit cluttered after playing around with it. The following section will show you how to delete images and containers. You can also refer to this cheat sheet for more on deleting images and containers. The below is just a few basic and useful commands.

Deleting Containers

To delete a container, first locate the container(s) you wish to delete, then use `docker rm <ID or NAME>` to delete:

```
$ docker ps -a

CONTAINER ID        IMAGE                  COMMAND                  CREATED
↪    STATUS                     PORTS               NAMES
c6695e4d9a63        simonko/pyklip:latest   "/usr/bin/tini -- ..."   6 seconds ago
↪    Exited (0) 3 seconds ago                       zealous_goldwasser

$ docker rm <container ID (c6695e4d9a63) or Name (zealous goldwasser)>
```

To delete multiple containers at once use the filter flag. For example, if you want to delete all exited containers

```
$ docker rm $(docker ps -a -f status=exited -q)
```

You can also find all containers all exited containers using just the command in the parenthesis without the *-q* flag. This is particularly useful if there are many exited containers and you don't remember which ones you wanted to delete.

Deleting Images

To delete your images first you must find which ones you wish to delete. It should also be noted that to delete an image, there can be no containers associated with it. You must delete all containers from the image before deleting the image.

```
$ docker images

REPOSITORY              TAG                 IMAGE ID            CREATED
↪SIZE
pyklip-pipeline         latest              e9a584c685bb        13 days ago         2.
↪37 GB
simonko/pyklip          latest              e9a584c685bb        13 days ago         2.
↪37 GB
localrepo               latest              dc74a96e5ef0        2 weeks ago         2.
↪25 GB
ubuntu                  latest              0ef2e08ed3fa        3 weeks ago
↪130 MB
continuumio/anaconda3   latest              26043756c44f        6 weeks ago         2.
↪23 GB

$ docker rmi <repository name>
```

Note: Before you delete an image, all containers using the image must be DELETED, not exited.

To delete ALL of your images

```
$ docker rmi $(docker images -a -q)
```

Sharing Images

Here you will learn how to create and upload your own images onto Docker Hub for others to use.

Creating Images

In this section, you will learn how to create and upload your own image. To do this you need to make a dockerfile. If you wish to share the image for others to use, you need to create a Docker Hub account and push your image into a repository. This section will go over all of these steps. For a more detailed tutorial use this link. Otherwise here are the very basics.

Docker images are created from a set of commands in a dockerfile. What goes on this file is entirely up to you. Docker uses these commands to create an image, and it can be an entirely new one or an image based off of another existing image.

1. **Create a file and name it dockerfile. There are three basic commands that go on a dockerfile.**
    - FROM <Repository>:<Build> - This command will tell docker that this image is based off of another image. You can specify which build to use. To use the most up-to-date version of the image, use "latest" for build.
    - RUN <Command> - This will run commands in a new layer and creates a new image. Typically used for installing necessary packages. You can have multiple RUN statements.
    - CMD <Command> - This is the default command that will run once the image environment has been set up. You can only have ONE CMD statement.

    For more information on RUN vs CMD here is a useful link.
2. After you've made your file run the following command to create your image

```
$ docker build -t <Image Name> <Path to Directory of Dockerfile>
```

The `-t` flag lets you name the image.

For example, the docker file used for the pyklip image I set up above (under the "Using Docker" section) is made using a dockerfile with the following content:

```
FROM continuumio/anaconda3:latest
RUN git clone https://bitbucket.org/pyKLIP/pyklip.git \
 && pip install coveralls \
 && pip install emcee \
 && pip install corner \
 && conda install -c https://conda.anaconda.org/astropy photutils
```

Uploading Images

1. If you haven't already, create a Docker Hub account.
2. After you've made your account, sign in and click on "Create Repository" and fill out the details. Make sure visibility is set to PUBLIC. Press create.
3. Find your image ID. Using a previous example

```
$ docker images

REPOSITORY              TAG                 IMAGE ID            CREATED
↪  SIZE
pyklip-pipeline         latest              e9a584c685bb        13 days ago
↪  2.37 GB
```

The image ID would be e9a584c685bb.

4. Tag the image using

```
$ docker tag <Image ID> <DockerHub Account Name>/<Image Name>:<Version or Tag>
```

So for the pyklip pipeline image my command would be:

```
$ docker tag e9a584c685bb simonko/pyklip:latest
```

Check that the image has been tagged

```
$ docker images

REPOSITORY              TAG                 IMAGE ID            CREATED                 ␣
↪SIZE
pyklip-pipeline         latest              e9a584c685bb        13 days ago          2.
↪37 GB
simonko/pyklip          latest              e9a584c685bb        13 days ago          2.
↪37 GB
```

5. Login to Docker on terminal

```
$ docker login

Username: *****
Password: *****
Login Succeeded
```

6. Push your tagged image to docker hub

```
$ docker push <Repository Name>
```

7. To pull from the repo now, all you have to do is run the repo. Docker will automatically pull from docker hub if it cannot find it locally.

Tests

Here we will lay out the testing infrastructure used for pyKLIP.

Testing

Here we will go over how we test and what our testing infrastructure looks like for pyKLIP.

All of our tests can be found in the `tests` folder located in the base directory. In the folder, each module or feature gets it's own test file, and inside every file, each function is a different test for the module/feature.

The testing workflow for pyKLIP can be broken down into the following steps:

- Creating the tests
- Documenting the tests
- Running the tests

Creating Tests

All tests for pyKLIP can be found in the `tests` directory. We use pytest to run all of our tests in this directory. All tests should be named "test_<module/purpose>", and within the test files, each function should be named "test_<function name>" to give an idea of what the test is for. The docstring for the function will go into detail as to what the test is testing and a summary of how it works.

Our testing framework is organized so that each file tests an individual module or feature, and each function inside each test file tests different aspects of the module/feature.

During the test, you may find it necessary to look at input or output files. In this case, all pathing should be agnostic of the directory structure outside of the pyKLIP folder. It is suggested you first construct relative paths with respect to the current test file or the pyklip base directory, and then convert it to an absolute path before using it in the function.

Some commands you may find helpful to find files:

- **os.path.abspath(path)** - returns the absolute path of the path provided
- **os.path.dirname(path)** - returns the name of the directory of the filepath provided.
- **os.path.exists(path)** - returns True if the path exists, False otherwise.
- **os.path.sep** = path separator. This is important because different OSes can have different path separators. For example Ubuntu Linux uses "/" while Windows uses "\". This will take care of that.
- **os.path.join(args)** - returns a string with all the args separated by the appropriate path separator. For exmaple `os.path.join("this", "is", "a", "path")` would return `"this/is/a/path"` in Ubuntu Linux.
- **__file__** - When used on its own, filepath of this python file. All python modules should also have this as an attirbute (e.g. `pyklip.__file__`)

Documenting Tests

Docstring for tests should follow the Google Python stylguide. Here is an exmaple of a function docstring:

```
"""
Summary of what your tests does goes here.

Args:
    param1: First param.
    param2: Second param.
    etc: etc

Returns:
    A description of what is returned.

Raises:
    Error: Exception.
"""
```

Use the following link for more details on docstrings as well as Python style in general.

Running Tests

All of our tests are run automatically using `pytest` on a Docker image using a continuous integration build system (Bitbucket Pipelines). This allows us to test pyKIP against a fresh and updated Python installation to ensure function-

ality is not broken and is comptable with the newest Python version. If these terms seem unfamiliar, please refer to our *Developing for pyKLIP* page under the "Docker" section for more information on Docker.

Here is a simple overview of the steps invovled in our automated testing framework:

1. Bitbucket Pipelines reads our pipeline yml file to build the pipeline.
2. Creates a docker image of the latest continuum anaconda3.
3. Git clones the pyklip repository inside image.
4. Installs all necessary packages.
5. Runs tests using pytest on the test directory.
6. Runs coverage analysis on our tests.
7. Submits coverage report.

You can also run tests locally. This is typically useful when you make changes and want to check that the changes does not break any functionality. It can also be useful if you write a test before writing the function code, and debug your code as you develop your function. That way, you will have validation code from the start. In this case, you may not want to run the full suite of tests.

To simply run a single test you can either call the file directly using:

```
$ python <path/to/test file name>.py
```

To run all tests simply call:

```
$ pytest
```

The general command for pytest is as follows and there are two ways to invoke it:

```
$ python -m pytest [args]
$ pytest [args]
```

The above line will invoke pytest through the Python interpreter and add the current directory to sys.path. Otherwise the second line is equivalent to the first. There are many arguments and many different ways to use pytest. To run a single test simply enter the path to the test file to run, to test all files in a directory use the path to the directory instead of a single file. For more information on how to use pytest and some of its various usages, visit this link.

Code Coverage

Here we will go over code coverage, the analysis of what lines of code are tested in our tests.

Our code coverage is set up using two different tools - Coverage and Coveralls. Coverage is what we use to report the coverage statistics on our code and tests, while Coveralls is the service we use to hook our reports to our pipeline, giving us a website to read coverage reports for each build.

Coverage

The documentation for the coverage package can be found here.

There are several different ways of reporting code coverage. I highly recommend reading the How Coverage.py Works section to learn what it means to say your tests have x% code coverage. Basically, there are three phases to the code coverage we use:

- **Execution**: Executes code and records information.

- **Analysis**: Analyzes codebase for total executable lines.
- **Reporting**: Combines execution and analysis phases to give coverage report.

When tests are run, coverage.py runs a trace function that records each file and line that is executed when a test is run. It records this information in a JSON file (usually) named `.coverage`. This is called the execution phase.

During "Analysis," coverage looks at the compiled python files to get the set of executable lines, and filters through to leave out lines that shouldn't be considered (e.g. blank lines, docstrings).

Finally, the Reporting phase handles the format in which to report its findings. There are several different outputs for the reports that you can use.

Configuration

Coverage also has a configuration file that allows the user to specify different options for coverage to handle, such as multi-threading. The coverage configuration file is named `.coveragerc` by default. Information on the syntax for the file can be found here. Through the configuration file you can specify lines to skip, ignoring specific errors, where to output the coverage report, etc.

Note: When running multiple coverage reports or using the multi-thread option, the command `coverage combine` is useful in that it will combine all the reports into one. Multi-threading will spawn multiple processes which will each have their own report so combining is very important for getting an accurate report. Note that all the reports must be in the same directory when running the command.

As a final note, it is important to note that, although code coverage is a great tool to have and use, it is not by itself enough to say the code is bug free. 100% code coverage, in the end, does not mean much. It simply means all the executable lines of code have been run in one way or another, but there is no real way to test *ALL* possible branches and situations your code can take, especially for larger code bases. Read this article for more on why code coverage can be flawed as well as a few examples. Just know that code coverage is a useful tool but not fool-proof.

Coveralls

Coveralls is the web service used to track our code coverage and report on our automated pipeline builds. Every time code is pushed to our Bitbucket repo and our tests are run, we first obtain our report using coverage, then we send the report to coveralls which in turn organizes our report with each build and displays the information for us on both the coverage website and a badge on the bitbucket repo.

For information on how to setup a coveralls hook to a repo, look here. For our pipeline, we use Bitbucket Pipelines, so use the "Usage (another CI)" section.

pyklip package

Subpackages

pyklip.fmlib package

Submodules

pyklip.fmlib.diskfm module

class pyklip.fmlib.diskfm.**DiskFM**(*inputs_shape*, *numbasis*, *dataset*, *model_disk*, *basis_filename='klip-basis.p'*, *load_from_basis=False*, *save_basis=False*, *annuli=None*, *subsections=None*, *OWA=None*, *numthreads=None*, *mode='ADI'*)

Bases: *pyklip.fmlib.nofm.NoFM*

alloc_fmout(*output_img_shape*)
Allocates shared memory for output image

cleanup_fmout(*fmout*)
After running KLIP-FM, we need to reshape fmout so that the numKL dimension is the first one and not the last

Parameters **fmout** – numpy array of ouput of FM

Returns same but cleaned up if necessary

Return type fmout

fm_from_eigen(*klmodes=None*, *evals=None*, *evecs=None*, *input_img_shape=None*, *input_img_num=None*, *ref_psfs_indicies=None*, *section_ind=None*, *section_ind_nopadding=None*, *aligned_imgs=None*, *pas=None*, *wvs=None*, *radstart=None*, *radend=None*, *phistart=None*, *phiend=None*, *padding=None*, *IOWA=None*, *ref_center=None*, *parang=None*, *ref_wv=None*, *numbasis=None*, *fmout=None*, *perturbmag=None*, *klipped=None*, *covar_files=None*, ***kwargs*)
FIXME

fm_parallelized()
Functions like klip_parallelized, but doesn't find new evals and evecs.

load_basis_files(*basis_file_pattern*)
Loads in previously saved basis files and sets variables for fm_from_eigen

save_fmout(*dataset*, *fmout*, *outputdir*, *fileprefix*, *numbasis*, *klipparams=None*, *calibrate_flux=False*, *spectrum=None*)
Uses self.dataset parameters to save fmout, the output of fm_paralellized or klip_dataset

update_disk(*model_disk*)
Takes model disk and rotates it to the PAs of the input images for use as reference PSFS

Parameters

- **model_disk** – Disk to be forward modeled, can be either a 2D array ([N,N], where N is the width and height of your image)
- **which case, if the dataset is multiwavelength then the same model is used for all wavelenths. Otherwise, the model_disk is** (*in*) –
- **as a 3D arary, [nwvs, N,N], where nwvs is the number of wavelength channels)** (input) –

Returns None

pyklip.fmlib.extractSpec module

class pyklip.fmlib.extractSpec.**ExtractSpec**(*inputs_shape*, *numbasis*, *sep*, *pa*, *input_psfs*, *input_psfs_wvs*, *datatype='float'*, *stamp_size=None*)

Bases: *pyklip.fmlib.nofm.NoFM*

Planet Characterization class. Goal to characterize the astrometry and photometry of a planet

alloc_fmout(*output_img_shape*)

Allocates shared memory for the output of the shared memory

Parameters **output_img_shape** – shape of output image (usually N,y,x,b)

Returns mp.array to store FM data in fmout_shape: shape of FM data array

Return type fmout

cleanup_fmout(*fmout*)

After running KLIP-FM, we need to reshape fmout so that the numKL dimension is the first one and not the last

Parameters **fmout** – numpy array of ouput of FM

Returns same but cleaned up if necessary

Return type fmout

fm_from_eigen(*klmodes=None*, *evals=None*, *evecs=None*, *input_img_shape=None*, *input_img_num=None*, *ref_psfs_indicies=None*, *section_ind=None*, *section_ind_nopadding=None*, *aligned_imgs=None*, *pas=None*, *wvs=None*, *radstart=None*, *radend=None*, *phistart=None*, *phiend=None*, *padding=None*, *IOWA=None*, *ref_center=None*, *parang=None*, *ref_wv=None*, *numbasis=None*, *fmout=None*, *perturbmag=None*, *klipped=None*, ***kwargs*)

Generate forward models using the KL modes, eigenvectors, and eigenvectors from KLIP. Calls fm.py functions to perform the forward modelling

Parameters

- **klmodes** – unpertrubed KL modes
- **evals** – eigenvalues of the covariance matrix that generated the KL modes in ascending order (lambda_0 is the 0 index) (shape of [nummaxKL])
- **evecs** – corresponding eigenvectors (shape of [p, nummaxKL])
- **input_image_shape** – 2-D shape of inpt images ([ysize, xsize])
- **input_img_num** – index of sciece frame
- **ref_psfs_indicies** – array of indicies for each reference PSF
- **section_ind** – array indicies into the 2-D x-y image that correspond to this section. Note needs be called as section_ind[0]
- **pas** – array of N parallactic angles corresponding to N reference images [degrees]
- **wvs** – array of N wavelengths of those referebce images
- **radstart** – radius of start of segment
- **radend** – radius of end of segment
- **phistart** – azimuthal start of segment [radians]
- **phiend** – azimuthal end of segment [radians]

- **padding** – amount of padding on each side of sector
- **IOWA** – tuple (IWA,OWA) where IWA = Inner working angle and OWA = Outer working angle both in pixels. It defines the separation interva in which klip will be run.
- **ref_center** – center of image
- **numbasis** – array of KL basis cutoffs
- **parang** – parallactic angle of input image [DEGREES]
- **ref_wv** – wavelength of science image
- **fmout** – numpy output array for FM output. Shape is (N, y, x, b)
- **perturbmag** – numpy output for size of linear perturbation. Shape is (N, b)
- **klipped** – PSF subtracted image. Shape of (size(section), b)
- **kwargs** – any other variables that we don't use but are part of the input

generate_models(*input_img_shape*, *section_ind*, *pas*, *wvs*, *radstart*, *radend*, *phistart*, *phiend*, *padding*, *ref_center*, *parang*, *ref_wv*, *stamp_size=None*)

Generate model PSFs at the correct location of this segment for each image denoated by its wv and parallactic angle

Parameters

- **pas** – array of N parallactic angles corresponding to N images [degrees]
- **wvs** – array of N wavelengths of those images
- **radstart** – radius of start of segment
- **radend** – radius of end of segment
- **phistart** – azimuthal start of segment [radians]
- **phiend** – azimuthal end of segment [radians]
- **padding** – amount of padding on each side of sector
- **ref_center** – center of image
- **parang** – parallactic angle of input image [DEGREES]
- **ref_wv** – wavelength of science image
- **stamp_size** – size of the stamp for spectral extraction

Returns array of size (N, p) where p is the number of pixels in the segment

Return type models

pyklip.fmlib.extractSpec.**calculate_annuli_bounds**(*num_annuli*, *annuli_index*, *iwa*, *firstframe*, *firstframe_centers*)

Calculate annulus boundaries of a particular annuli. Useful for figuring out annuli boundaries when just giving an integer as the parameter to pyKLIP

Parameters

- **num_annuli** – integer for number of annuli requested
- **annuli_index** – integer for which annuli (innermost annulus is 0)
- **iwa** – inner working angle
- **firstframe** – data of first frame of the sequence. dataset.inputs[0]
- **firstframe_centers** – [x,y] center for the first frame. i.e. dataset.centers[0]

Returns

radial separation of annuli. [annuli_start, annuli_end] This is a single 2 element list [annuli_start, annuli_end]

Return type rad_bounds[annuli_index]

pyklip.fmlib.fmpsf module

class pyklip.fmlib.fmpsf.**FMPlanetPSF**(*inputs_shape*, *numbasis*, *sep*, *pa*, *dflux*, *input_psfs*, *input_wvs*, *flux_conversion=None*, *spectrallib=None*, *spectrallib_units='flux'*, *star_spt=None*, *refine_fit=False*)

Bases: *pyklip.fmlib.nofm.NoFM*

Forward models the PSF of the planet through KLIP. Returns the forward modelled planet PSF

alloc_fmout(*output_img_shape*)

Allocates shared memory for the output of the shared memory

Parameters **output_img_shape** – shape of output image (usually N,y,x,b)

Returns mp.array to store FM data in fmout_shape: shape of FM data array

Return type fmout

alloc_perturbmag(*output_img_shape*, *numbasis*)

Allocates shared memory to store the fractional magnitude of the linear KLIP perturbation Stores a number for each frame = max(oversub + selfsub)/std(PCA(image))

Parameters

- **output_img_shape** – shape of output image (usually N,y,x,b)
- **numbasis** – array/list of number of KL basis cutoffs requested

Returns mp.array to store linaer perturbation magnitude perturbmag_shape: shape of linear perturbation magnitude

Return type perturbmag

cleanup_fmout(*fmout*)

After running KLIP-FM, we need to reshape fmout so that the numKL dimension is the first one and not the last

Parameters **fmout** – numpy array of ouput of FM

Returns same but cleaned up if necessary

Return type fmout

fm_from_eigen(*klmodes=None*, *evals=None*, *evecs=None*, *input_img_shape=None*, *input_img_num=None*, *ref_psfs_indicies=None*, *section_ind=None*, *section_ind_nopadding=None*, *aligned_imgs=None*, *pas=None*, *wvs=None*, *radstart=None*, *radend=None*, *phistart=None*, *phiend=None*, *padding=None*, *IOWA=None*, *ref_center=None*, *parang=None*, *ref_wv=None*, *numbasis=None*, *fmout=None*, *perturbmag=None*, *klipped=None*, *covar_files=None*, *flipx=True*, ***kwargs*)

Generate forward models using the KL modes, eigenvectors, and eigenvectors from KLIP. Calls fm.py functions to perform the forward modelling

Parameters

- **klmodes** – unpertrubed KL modes

- **evals** – eigenvalues of the covariance matrix that generated the KL modes in ascending order (lambda_0 is the 0 index) (shape of [nummaxKL])
- **evecs** – corresponding eigenvectors (shape of [p, nummaxKL])
- **input_image_shape** – 2-D shape of inpt images ([ysize, xsize])
- **input_img_num** – index of sciece frame
- **ref_psfs_indicies** – array of indicies for each reference PSF
- **section_ind** – array indicies into the 2-D x-y image that correspond to this section. Note needs be called as section_ind[0]
- **pas** – array of N parallactic angles corresponding to N reference images [degrees]
- **wvs** – array of N wavelengths of those referebce images
- **radstart** – radius of start of segment
- **radend** – radius of end of segment
- **phistart** – azimuthal start of segment [radians]
- **phiend** – azimuthal end of segment [radians]
- **padding** – amount of padding on each side of sector
- **IOWA** – tuple (IWA,OWA) where IWA = Inner working angle and OWA = Outer working angle both in pixels. It defines the separation interva in which klip will be run.
- **ref_center** – center of image
- **numbasis** – array of KL basis cutoffs
- **parang** – parallactic angle of input image [DEGREES]
- **ref_wv** – wavelength of science image
- **fmout** – numpy output array for FM output. Shape is (N, y, x, b)
- **perturbmag** – numpy output for size of linear perturbation. Shape is (N, b)
- **klipped** – PSF subtracted image. Shape of (size(section), b)
- **kwargs** – any other variables that we don't use but are part of the input

generate_models(*input_img_shape*, *section_ind*, *pas*, *wvs*, *radstart*, *radend*, *phistart*, *phiend*, *padding*, *ref_center*, *parang*, *ref_wv*, *flipx*)

Generate model PSFs at the correct location of this segment for each image denoated by its wv and parallactic angle

Parameters

- **pas** – array of N parallactic angles corresponding to N images [degrees]
- **wvs** – array of N wavelengths of those images
- **radstart** – radius of start of segment
- **radend** – radius of end of segment
- **phistart** – azimuthal start of segment [radians]
- **phiend** – azimuthal end of segment [radians]
- **padding** – amount of padding on each side of sector
- **ref_center** – center of image

- **parang** – parallactic angle of input image [DEGREES]
- **ref_wv** – wavelength of science image
- **flipx** – if True, flip x coordinate in final image

Returns array of size (N, p) where p is the number of pixels in the segment

Return type models

save_fmout(*dataset*, *fmout*, *outputdir*, *fileprefix*, *numbasis*, *klipparams=None*, *calibrate_flux=False*, *spectrum=None*)

Saves the FM planet PSFs to disk. Saves both a KL mode cube and spectral cubes

Parameters

- **dataset** – Instruments.Data instance. Will use its dataset.savedata() function to save data
- **fmout** – the fmout data passed from fm.klip_parallelized which is passed as the output of cleanup_fmout
- **outputdir** – output directory
- **fileprefix** – the fileprefix to prepend the file name
- **numbasis** – KL mode cutoffs used
- **klipparams** – string with KLIP-FM parameters
- **calibrate_flux** – if True, flux calibrate the data in the same way as the klipped data
- **spectrum** – if not None, spectrum to weight the data by. Length same as dataset.wvs

pyklip.fmlib.fmpsf.**calculate_annuli_bounds**(*num_annuli*, *annuli_index*, *iwa*, *firstframe*, *firstframe_centers*)

Calculate annulus boundaries of a particular annuli. Useful for figuring out annuli boundaries when just giving an integer as the parameter to pyKLIP

Parameters

- **num_annuli** – integer for number of annuli requested
- **annuli_index** – integer for which annuli (innermost annulus is 0)
- **iwa** – inner working angle
- **firstframe** – data of first frame of the sequence. dataset.inputs[0]
- **firstframe_centers** – [x,y] center for the first frame. i.e. dataset.centers[0]

Returns

radial separation of annuli. [annuli_start, annuli_end] This is a single 2 element list [annuli_start, annuli_end]

Return type rad_bounds[annuli_index]

pyklip.fmlib.matchedFilter module

class pyklip.fmlib.matchedFilter.**MatchedFilter**(*inputs_shape*, *numbasis*, *input_psfs*, *input_psfs_wvs*, *spot_flux*, *spectrallib=None*, *mute=False*, *star_type=None*, *filter_name=None*, *save_per_sector=None*, *datatype='float'*, *fakes_sepPa_list=None*, *disable_FM=None*, *true_fakes_pos=None*)

Bases: *pyklip.fmlib.nofm.NoFM*

Matched filter with forward modelling.

alloc_fmout(*output_img_shape*)

Allocates shared memory for the output of the shared memory

Parameters **output_img_shape** – shape of output image (usually N,y,x,b)

Returns mp.array to store auxilliary data in fmout_shape: shape of auxilliary array

Return type fmout

fm_end_sector(*interm_data=None*, *fmout=None*, *sector_index=None*, *section_indicies=None*)

Does some forward modelling at the end of a sector after all images have been klipped for that sector.

fm_from_eigen(*klmodes=None*, *evals=None*, *evecs=None*, *input_img_shape=None*, *input_img_num=None*, *ref_psfs_indicies=None*, *section_ind=None*, *section_ind_nopadding=None*, *aligned_imgs=None*, *pas=None*, *wvs=None*, *radstart=None*, *radend=None*, *phistart=None*, *phiend=None*, *padding=None*, *IOWA=None*, *ref_center=None*, *parang=None*, *ref_wv=None*, *numbasis=None*, *fmout=None*, *perturbmag=None*, *klipped=None*, ***kwargs*)

Parameters

- **klmodes** – unpertrubed KL modes
- **evals** – eigenvalues of the covariance matrix that generated the KL modes in ascending order (lambda_0 is the 0 index) (shape of [nummaxKL])
- **evecs** – corresponding eigenvectors (shape of [p, nummaxKL])
- **input_image_shape** – 2-D shape of inpt images ([ysize, xsize])
- **input_img_num** – index of sciece frame
- **ref_psfs_indicies** – array of indicies for each reference PSF
- **section_ind** – array indicies into the 2-D x-y image that correspond to this section. Note needs be called as section_ind[0]
- **pas** – array of N parallactic angles corresponding to N reference images [degrees]
- **wvs** – array of N wavelengths of those referebce images
- **radstart** – radius of start of segment
- **radend** – radius of end of segment
- **phistart** – azimuthal start of segment [radians]
- **phiend** – azimuthal end of segment [radians]
- **padding** – amount of padding on each side of sector
- **IOWA** – tuple (IWA,OWA) where IWA = Inner working angle and OWA = Outer working angle both in pixels. It defines the separation interva in which klip will be run.

- **ref_center** – center of image
- **numbasis** – array of KL basis cutoffs
- **parang** – parallactic angle of input image [DEGREES]
- **ref_wv** – wavelength of science image
- **fmout** – numpy output array for FM output. Shape is (N, y, x, b)
- **klipped** – array of shape (p,b) that is the PSF subtracted data for each of the b KLIP basis cutoffs. If numbasis was an int, then sub_img_row_selected is just an array of length p
- **kwargs** – any other variables that we don't use but are part of the input

generate_model_sci(*input_img_shape*, *section_ind*, *pa*, *wv*, *radstart*, *radend*, *phistart*, *phiend*, *padding*, *ref_center*, *parang*, *ref_wv*, *sep_fk*, *pa_fk*)

Generate model PSFs at the correct location of this segment for each image denoated by its wv and parallactic angle

Parameters

- **pas** – array of N parallactic angles corresponding to N images [degrees]
- **wvs** – array of N wavelengths of those images
- **radstart** – radius of start of segment
- **radend** – radius of end of segment
- **phistart** – azimuthal start of segment [radians]
- **phiend** – azimuthal end of segment [radians]
- **padding** – amount of padding on each side of sector
- **ref_center** – center of image
- **parang** – parallactic angle of input image [DEGREES]
- **ref_wv** – wavelength of science image

Returns array of size (N, p) where p is the number of pixels in the segment

Return type models

generate_model_sci_nearestNeigh(*input_img_shape*, *section_ind*, *pa*, *wv*, *radstart*, *radend*, *phistart*, *phiend*, *padding*, *ref_center*, *parang*, *ref_wv*, *sep_fk*, *pa_fk*)

Generate model PSFs at the correct location of this segment for each image denoated by its wv and parallactic angle

Parameters

- **pas** – array of N parallactic angles corresponding to N images [degrees]
- **wvs** – array of N wavelengths of those images
- **radstart** – radius of start of segment
- **radend** – radius of end of segment
- **phistart** – azimuthal start of segment [radians]
- **phiend** – azimuthal end of segment [radians]
- **padding** – amount of padding on each side of sector

- **ref_center** – center of image
- **parang** – parallactic angle of input image [DEGREES]
- **ref_wv** – wavelength of science image

Returns array of size (N, p) where p is the number of pixels in the segment

Return type models

generate_models(*input_img_shape*, *section_ind*, *pas*, *wvs*, *radstart*, *radend*, *phistart*, *phiend*, *padding*, *ref_center*, *parang*, *ref_wv*, *sep_fk*, *pa_fk*)

Generate model PSFs at the correct location of this segment for each image denoated by its wv and parallactic angle

Parameters

- **pas** – array of N parallactic angles corresponding to N images [degrees]
- **wvs** – array of N wavelengths of those images
- **radstart** – radius of start of segment
- **radend** – radius of end of segment
- **phistart** – azimuthal start of segment [radians]
- **phiend** – azimuthal end of segment [radians]
- **padding** – amount of padding on each side of sector
- **ref_center** – center of image
- **parang** – parallactic angle of input image [DEGREES]
- **ref_wv** – wavelength of science image

Returns array of size (N, p) where p is the number of pixels in the segment

Return type models

generate_models_nearestNeigh(*input_img_shape*, *section_ind*, *pas*, *wvs*, *radstart*, *radend*, *phistart*, *phiend*, *padding*, *ref_center*, *parang*, *ref_wv*, *sep_fk*, *pa_fk*)

Generate model PSFs at the correct location of this segment for each image denoated by its wv and parallactic angle

Parameters

- **pas** – array of N parallactic angles corresponding to N images [degrees]
- **wvs** – array of N wavelengths of those images
- **radstart** – radius of start of segment
- **radend** – radius of end of segment
- **phistart** – azimuthal start of segment [radians]
- **phiend** – azimuthal end of segment [radians]
- **padding** – amount of padding on each side of sector
- **ref_center** – center of image
- **parang** – parallactic angle of input image [DEGREES]
- **ref_wv** – wavelength of science image

Returns array of size (N, p) where p is the number of pixels in the segment

Return type models

skip_section(*radstart*, *radend*, *phistart*, *phiend*)
Returns a boolean indicating if the section defined by (radstart, radend, phistart, phiend) should be skipped. When True is returned the current section in the loop in klip_parallelized() is skipped.

Parameters

- **radstart** – minimum radial distance of sector [pixels]
- **radend** – maximum radial distance of sector [pixels]
- **phistart** – minimum azimuthal coordinate of sector [radians]
- **phiend** – maximum azimuthal coordinate of sector [radians]

Returns False so by default it never skips.

Return type Boolean

pyklip.fmlib.nofm module

class pyklip.fmlib.nofm.**NoFM**(*inputs_shape*, *numbasis*)
Bases: object

Super class for all forward modelling classes. Has fall-back functions for all fm dependent calls so that each FM class does not need to implement functions it doesn't want to. Should do no forward modelling and just do regular KLIP by itself

alloc_fmout(*output_img_shape*)
Allocates shared memory for the output of the forward modelling

Parameters **output_img_shape** – shape of output image (usually N,y,x,b)

Returns mp.array to store FM data in fmout_shape: shape of FM data array

Return type fmout

alloc_interm(*max_sector_size*, *numsciframes*)
Allocates shared memory array for intermediate step

Intermediate step is allocated for a sector by sector basis

Parameters **max_sector_size** – number of pixels in this sector. Max because this can be variable. Stupid rotating sectors

Returns mp.array to store intermediate products from one sector in interm_shape:shape of interm array (used to convert to numpy arrays)

Return type interm

alloc_output()
Allocates shared memory array for final output

Only use multiprocessing data structors as we are using the multiprocessing class

Args:

Returns mp.array to store final outputs in (shape of (N*wv, y, x, numbasis)) outputs_shape: shape of outputs array to convert to numpy arrays

Return type outputs

alloc_perturbmag(*output_img_shape*, *numbasis*)
Allocates shared memory to store the fractional magnitude of the linear KLIP perturbation

Parameters

- **output_img_shape** – shape of output image (usually N,y,x,b)
- **numbasis** – array/list of number of KL basis cutoffs requested

Returns mp.array to store linaer perturbation magnitude perturbmag_shape: shape of linear perturbation magnitude

Return type perturbmag

cleanup_fmout(*fmout*)
After running KLIP-FM, if there's anything to do to the fmout array (such as reshaping it), now's the time to do that before outputting it

Parameters **fmout** – numpy array of ouput of FM

Returns same but cleaned up if necessary

Return type fmout

fm_end_sector(*selfself*, ***kwargs*)
Does some forward modelling at the end of a sector after all images have been klipped for that sector.

fm_from_eigen(***kwargs*)
Generate forward models using the KL modes, eigenvectors, and eigenvectors from KLIP This is called immediately after regular KLIP

save_fmout(*dataset*, *fmout*, *outputdir*, *fileprefix*, *numbasis*, *klipparams=None*, *calibrate_flux=False*, *spectrum=None*)
Saves the fmout data to disk following the instrument's savedata function

Parameters

- **dataset** – Instruments.Data instance. Will use its dataset.savedata() function to save data
- **fmout** – the fmout data passed from fm.klip_parallelized which is passed as the output of cleanup_fmout
- **outputdir** – output directory
- **fileprefix** – the fileprefix to prepend the file name
- **numbasis** – KL mode cutoffs used
- **klipparams** – string with KLIP-FM parameters
- **calibrate_flux** – if True, flux calibrate the data (if applicable)
- **spectrum** – if not None, the spectrum to weight the data by. Length same as dataset.wvs

skip_section(*radstart*, *radend*, *phistart*, *phiend*)
Returns a boolean indicating if the section defined by (radstart, radend, phistart, phiend) should be skipped. When True is returned the current section in the loop in klip_parallelized() is skipped.

Parameters

- **radstart** – minimum radial distance of sector [pixels]
- **radend** – maximum radial distance of sector [pixels]
- **phistart** – minimum azimuthal coordinate of sector [radians]
- **phiend** – maximum azimuthal coordinate of sector [radians]

Returns False so by default it never skips.

Return type Boolean

Module contents

pyklip.instruments package

Subpackages

pyklip.instruments.P1640_support package

Submodules

pyklip.instruments.P1640_support.P1640_cube_checker module

Given a datacube, find the four corresponding spot files. Plot the calculated positions on top of the original cube.

Run from an ipython terminal with: %run spot_checker.py full/path/to/cube.fits

class pyklip.instruments.P1640_support.P1640_cube_checker.**ConfigAction**(*option_strings*, *dest*, *nargs=None*, ***kwargs*)

Bases: argparse.Action

Create a custom action to parse the

pyklip.instruments.P1640_support.P1640_cube_checker.**dnah_spot_directory = '/data/p1640/data/users/s**

Pseudocode –

1.Load list of files

2.Create the "good files" dictionary

3. For each file: 3a. Split offt a thread for drawing the cube 3b. Ask for user input 4. When the user provides 'y' or 'n', update the dictionary and kill the drawing thread 5. Move on to the next file

pyklip.instruments.P1640_support.P1640_cube_checker.**draw_cube**(*cube*, *cube_name*, *header*, *seeing*, *airmass*, *cube_ix*)

Make a figure and draw cube slices on it

pyklip.instruments.P1640_support.P1640_cube_checker.**draw_spot_cube**(*cube*, *cube_name*, *spots*)

Make a figure and draw cube slices on it spots are a list of [row, col] positions for each spot

pyklip.instruments.P1640_support.P1640_cube_checker.**get_total_exposure_time**(*fitsfiles*, *unit=Unit("min")*)

Accept a list of fits files and return the total exposure time Input:

fitsfiles: single fits file or list of files with keyword 'EXPTIME' in the header units: [minute] astropy.units unit for the output

Output: tot_exp_time: the sum of the exposure times for each cube, in minutes

pyklip.instruments.P1640_support.P1640_cube_checker.**plot_airmass_and_seeing**(*fitsfiles*)

pyklip.instruments.P1640_support.P1640_cube_checker.**run_checker**(*fitsfiles*)
Run the checker

pyklip.instruments.P1640_support.P1640_cube_checker.**run_spot_checker**(*files=None*, *config=None*, *spot_path=None*)
Supply ONE OF: files: list of files config: config file with a list of files

pyklip.instruments.P1640_support.P1640_cube_checker.**usage**()

pyklip.instruments.P1640_support.P1640_cube_checker_interactive module

Observational SNR and exposure time calculator Jonathan Aguilar Nov. 15, 2013

class pyklip.instruments.P1640_support.P1640_cube_checker_interactive.**ConfigAction**(*option_strings*, *dest*, *nargs=None*, ***kwargs*)
Bases: argparse.Action

Create a custom action to parse the command line arguments

class pyklip.instruments.P1640_support.P1640_cube_checker_interactive.**CubeChecker**(*master*, *fitsfiles*, *spot_mode=Fal*, *spot_path='/da*

activate_printing(*event*)

bind_buttons_to_frame(*frame_ref*)
These are the buttons you want to be recognized by all the frames

current_cube

draw_spots_on_cube()

keep_button_pushed(*event*)

load_selected_cube()
Read the highlighted fitsfile, get the datacube from it, and display it This function does the heavy lifting and calls the other update functions

- •check for spot mode
- •set the current file index
- •set the current file using the current file index
- •load the cube from the current file

load_spot_files()
based on the current cube and the spot path, get the spot files

next_button_pushed(*event*)

next_cube()

prev_button_pushed(*event*)

prev_cube()

print_good_cubes()

quit_button_pushed(*event*)

scroll_cube_left(*event*)

scroll_cube_right(*event*)

toggle_autoscroll()

toggle_check()

update_cube_stats()

update_cubeax_display(*event=None*)

update_seeing_and_airmass()

pyklip.instruments.P1640_support.P1640_cube_checker_interactive.**get_total_exposure_time**(*fitsf uni*

Accept a list of fits files and return the total exposure time Input:

fitsfiles: single fits file or list of files with keyword 'EXPTIME' in the header units: [minute] astropy.units unit for the output

Output: tot_exp_time: the sum of the exposure times for each cube, in minutes

pyklip.instruments.P1640_support.P1640_spot_checker module

Given a datacube, find the four corresponding spot files. Plot the calculated positions on top of the original cube.

Run from an ipython terminal with: %run spot_checker.py full/path/to/cubes.fits

class pyklip.instruments.P1640_support.P1640_spot_checker.**ConfigAction**(*option_strings*, *dest*, *nargs=None*, ***kwargs*)

Bases: argparse.Action

Create a custom action to parse the

pyklip.instruments.P1640_support.P1640_spot_checker.**draw_cube**(*cube*, *cube_name*, *spots*)

Make a figure and draw cube slices on it spots are a list of [row, col] positions for each spot

pyklip.instruments.P1640_support.P1640_spot_checker.**run_checker**(*files=None*, *config=None*, *spot_path=None*)

Supply ONE OF: files: list of files config: config file with a list of files

pyklip.instruments.P1640_support.P1640contrast module

Utilities specific to generating contrast curves

pyklip.instruments.P1640_support.P1640contrast.**calc_contrast_multifile**(*core_files*, *datacube*)

Assemble a median core PSF out of the list of core_files, and return a datacube scaled by the core flux

pyklip.instruments.P1640_support.P1640contrast.**calc_contrast_single_file**(*filename*, *core_info=None*, *chans='all'*)

Calculate the radiall-averaged variance for a pyklip-reduced file for a single channel Input:

filename: full path to a pyklip-processed datacube core_info: pandas DataFrame with 'radius' and 'flux' columns chans: iterative type list of channels

pyklip.instruments.P1640_support.P1640contrast.**make_contrast_plot**(*contrast_map*, *title=None*, *contrast_range=None*, *plate_scale=19.2*, *pckwargs=None*)

Plot contrast against separation and channel number. Input:

contrast_map: Pandas DataFrame. See required columns below. name: plot title (preferably the file name corresponding to the contrast map) plate_scale (19.2 mas/pix): convert between pixels and mas contrast_range (None): tuple of upper and lower bounds for contrast plot. If none, min-max. pckwargs: dictionary of arguments that can be passed to plt.pcolor

Returns plt.figure() object

Return type fig

contrast_map: one column is titled 'rad', this is the separation in pixels The rest of the columns are titled 'chan##', where ## is the channel number.

pyklip.instruments.P1640_support.P1640contrast.**make_contrast_summary_plot**(*contrast_map_dict*, *chan='chan23'*, *title=None*, *plate_scale=19.2*, *kwargs=None*)

Plot the mean, min, and max contrast in the given channel for all the reductions

pyklip.instruments.P1640_support.P1640cores module

This library has method for operating on P1640 core images

pyklip.instruments.P1640_support.P1640cores.**aperture_convolve_cube**(*orig_cube*, *aperture_radii*, *apkwargs={'subpixels': 4, 'method': 'subpixel'}*)

Perform apeture photometry on every pixel in a datacube or image Wrapper for _aperture_convolve_img to handle 2-D and 3-D data Input:

orig_cube: [Nlambda x] Npix x Npix datacube aperture_radii: Nlambda array of aperture radii apkwargs: dictionary of arguments to pass to aperture_photometry

Default: {'method':'subpixel','subpixels':4}

Returns cube with shape of orig_cube of the aperture photometry

Return type phot_cube

pyklip.instruments.P1640_support.P1640cores.**centroid_image**(*orig_img*)
Centroid an image - weighted sum of pixels Input:

orig_img: 2D array

Returns [y,x] floating-point coordinates of the centroid

pyklip.instruments.P1640_support.P1640cores.**combine_multiple_cores**(*multiple_core_info*)
Combine the stellar flux and radii information from multiple cores in the proper way.

pyklip.instruments.P1640_support.P1640cores.**get_PSF_center**(*orig_cube*, *refchan=26*, *fine=False*)
Return the PSF center at the pixel level (default) or subpixel level (fine=True) Input:

orig_cube: [Nlambda x] Npix x Npix datacube (or image) refchan(=26): Reference channel for the initial center estimate fine_centering(=False): After getting a rough estimate of the center, centroid the image

Returns Nlamdba x 2 array of pixel indices for the PSF center

pyklip.instruments.P1640_support.P1640cores.**get_cube_xsection**(*orig_cube*, *center*, *width*)
Select the cross-section of a cube centered in center with 1/2-width width Input:

orig_cube: Nlambda x Npix x Npix datacube center: [row, col] index width: 1/2-width of cross-section

Returns Nlambda x (2*width+1) x (2*width+1) cube cross-section

Return type cube_cutout

pyklip.instruments.P1640_support.P1640cores.**get_encircled_energy_cube**(*orig_cube*, *frac=0.5*)
Get the fractional encircled energy of a PSF in each channel of a datacube. Basically a wrapper for _get_encircled_energy_image. Accepts 2-D and 3-D input. Input:

orig_cube: unocculted core cube [Nlambda x]Npix x Npix frac: encircled energy cutoff

Returns [starx, stary, radius, flux]

Return type Pandas Dataframe with Nlambda columns

pyklip.instruments.P1640_support.P1640cores.**get_injection_core**(*core_cubes*)
Remember, the injected PSF needs to be the SAME as the reference PSF, except for a scaling factor! Combine multiple cubes into a single core file for injection. Make sure that the total injected flux is the sum of the pixels! Outline: 1. Get the encircled energy fraction for all the cores (frac=1) 2. For each core, prepare a zero-cube with width of the largest radius + 1 3. Add the core from each channel to the zero-cube

pyklip.instruments.P1640_support.P1640cores.**make_median_core**(*core_cubes*)
Take a set of core cubes and assemble a median cube out of them. Set all non-PSF pixels to 0 Input:

core_cubes: Ncubes x Nlambda x Npix x Npix set of cores

Returns Nlambda x Npix x Npix

Return type median_core

pyklip.instruments.P1640_support.P1640cores.**zero_pad_core_box**(*core_cutout*, *centers*, *radii*)

Get a cube of core cutouts with a center and a radius for each channel Inside/outside the radius is determined by the center of the pixel. Negative pixels are set to 0.

pyklip.instruments.P1640_support.P1640spots module

class pyklip.instruments.P1640_support.P1640spots.**P1640params**

aperture_refchan = 3.5

channels = array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31

nchan = 32

num_spots = 4

refchan = 26

reflambda = 1.663451612903226

scale_factors = array([0.58252371, 0.59858049, 0.61463727, 0.63069405, 0.64675083, 0.66280761, 0.67886439, 0.6949

wlsol = array([0.969 , 0.99570968, 1.02241935, 1.04912903, 1.07583871, 1.10254839, 1.12925806, 1.15596774, 1.18267742

pyklip.instruments.P1640_support.P1640spots.**check_bad_channels**(*rad_spot*)

Check that the spot positions increase monotonically in radius. If they don't, return the positions that do not follow monotonically. Input:

rad_spot: Nchan array of radial sep of a spot from star

Output: bad_chans: a list of channel pairs that fail the check and need fixing

pyklip.instruments.P1640_support.P1640spots.**check_bad_spots**(*spot*, *centers*)

Input: spot: y,x positions for a spot centers: y,x positions for the center

Output: fixed_spot: a fixed spot position y,x

pyklip.instruments.P1640_support.P1640spots.**fit_grid_spot**(*img*, *center*, *loc=None*)

Fit spot with a 2-D Gaussian Inputs:

img: 2-D masked_array to fit center: center of the image in (row, col) order loc (optional): initial position guess in (row, col) order

pyklip.instruments.P1640_support.P1640spots.**fit_grid_spots**(*masked_cubes*, *centers*, *spots_guesses*)

Wrapper for fit_grid_spot to loop over all four spots Input:

masked_cubes: Nspot x Nchan x Npix x Npix array of masks centers: Nchan x 2 array of (row, col) guesses for spot centers spot_guesses: Nspot x Nchan x 2 (row, col) guesses for spot locations

Output: spot_fits: List of astropy.model fits spot_locs: Nspot x Nchan x 2 array of spot locations from fitting

pyklip.instruments.P1640_support.P1640spots.**fit_poly**(*ind*, *dep*, *order*)

Takes in an array of positions in row, col format and returns a polynomial that fits them to col = b + C*row, where C is a vector of coefficients Fitting is done by least squares Input:

ind: dependent variable (prob channel number) dep: independent variable (prob x or y spot position)

Returns array of polynomial coefficients

pyklip.instruments.P1640_support.P1640spots.**fix_bad_channels**(*spot*, *centers*, *bad_chans*)

Two cases:

1. a spot jumps inwards
2. a spot jumps outwards

In either case, remove both the failing spot and the one before it and fit a cubic to the remaining points. Then, fix the spot that is further from the fit. Input:

spot: Nchan x 2 array of positions for one spot centers: y,x positions for the star in each channel bad_chan: index of a spot that does not monotonically increase in radius

Output: fixed_spot: Nchan x 2 array of fixed positions for the spot

pyklip.instruments.P1640_support.P1640spots.**get_centered_grid**(*img_shape*, *center*)

Return a coordinate grid shifted to the center

pyklip.instruments.P1640_support.P1640spots.**get_initial_spot_guesses**(*cube*, *rotated_spots=False*)

pyklip.instruments.P1640_support.P1640spots.**get_points_from_poly**(*ind*, *coeffs*)

Get a polynomials coefficients and return the y-values given the independent values

pyklip.instruments.P1640_support.P1640spots.**get_rotated_grid**(*img_shape*, *center*, *angle*)

Rotate a coordinate grid by some angle around a center

pyklip.instruments.P1640_support.P1640spots.**get_scaling**(*spot_array*, *star_array=None*, *return_mean=True*)

Wrapper for get_single_cube_scaling_factors, to handle multiple cubes Input:

spot_array: (Nfiles) x Nspots x Nchan x 2 array in (row, col) order star_array: (Nfiles) x Nchan x 2 array of (row, column) star positions.

If not supplied, spot_array will be calculated from spot_array

return_mean: (default False) If true, return mean scaling factor for each channel (useful for multiple cubes)

Output: scaling_array: (Nfiles) x Nchan array of scaling fators

pyklip.instruments.P1640_support.P1640spots.**get_scaling_and_centering_from_files**(*files*, *mean_scaling*

Take some csv spot files, and return the star positions and scaling factors for each datacube Wrapper for get_scaling_and_centering_from_spots Input:

files: a list of fits files with data cubes if the files end in fits or csv, call appropriate routines

mean_scaling: [True] return the mean scaling of the 4 spots

Output: scaling_factors: scaling factors for each slice of each cube star_positions: star positions in each slice of each cube

pyklip.instruments.P1640_support.P1640spots.**get_scaling_and_centering_from_spots**(*spot_positions*, *mean_scaling*

Accepts an array of spots, and returns the scaling factors and centers. See also: get_scaling_and_centering_from_files Input:

spot_positions: Ncube x Nspot x Nchan x 2 array of (row, col) spot positions mean_scaling: [True] return the average scaling of the 4 spots

Output: scaling_factors: Ncube x Nchan array of scaling factors star_positions: Ncube x Nchan x 2 array of (row, col) star positions

pyklip.instruments.P1640_support.P1640spots.**get_single_cube_scaling_factors**(*spot_array*, *star_array=None*)

Get the scaling factors for a single cube Input:

spot_array: Nspots x Nchan x 2 array of (row, column) spot positions star_array: Nchan x 2 array of (row, column) star positions.

If not supplied, spot_array will be calculated from spot_array

Output:

scaling: Nspots x Nchan array of scaling factors, normalized to P1640params.refchan

pyklip.instruments.P1640_support.P1640spots.**get_single_cube_spot_photometry**(*cube*, *spot_positions*)

Do aperture photometry on the spots. Will need to be careful about aperture size for future comparison Input:

cube: Nchan x Npix x Npix data cube to do photometry spot_positions: Nspot x Nchan x 2 spot positions for apertures scaling_factors: Nchan array for scaling apertures with wavelength

Output: spot_phot: Nspot x Nchan array of spot photometry and spot errors

pyklip.instruments.P1640_support.P1640spots.**get_single_cube_spot_positions**(*cube*, *rotated_spots=False*)

Return the spot positions for a single cube Input:

cube: a data cube from P1640 rotated_spots: (False) if True, use the rotated masks

Output: spot_array: Nspots x Nchan x 2 array of spot positions.

pyklip.instruments.P1640_support.P1640spots.**get_single_cube_spot_positions_and_photometry**(

Wrapper that combines get_single_cube_spot_positions and get_single_cube_spot_photometry Input:

cube: a datacube in P1640 format

Output: spot_positions: Nspots x Nchan x 2 array of spot positions. spot_photometry: Nspots x Nchan x 1 array of spot fluxes

pyklip.instruments.P1640_support.P1640spots.**get_single_cube_star_positions**(*spot_array*)

Using the spot positions for a single cube, find the star position at each wavelength. Input:

spot_array: Nspots x Nchan x 2 array of (row, column) spot positions

Output: star_array: Nchan x 2 array of (row, column) star positions

pyklip.instruments.P1640_support.P1640spots.**get_single_file_scaling_and_centering**(*fitsfile*)
Take a single fits file, and return the star positions and scaling factors See also: get_scalign_and_centering_from_spots Input:

fitsfile: a single fits file with a P1640 cube

Output: scaling_factors: scaling factors for each slice of the cube star_positions: star positions in each slice of the cube

pyklip.instruments.P1640_support.P1640spots.**get_single_file_spot_positions**(*fitsfile*, *rotated_spots=False*)
Wrapper for get_single_cube_spot_positions

pyklip.instruments.P1640_support.P1640spots.**get_spot_positions**(*fitsfiles*)
Return the spot positions for a set of data cubes. Really just a wrapper for get_single_cube_spot_positions Input:

fitsfiles: a list of P1640 fits files

Output: spot_array: Nfile x 4 x Nchan x 2 array of spot positions

pyklip.instruments.P1640_support.P1640spots.**get_spot_positions_and_photometry**(*fitsfiles*)
Wrapper that combines get_single_cube_spot_positions and get_single_cube_spot_photometry Accept a list of fits files and returns the spot positions and spot photometry Input:

fitsfiles: a list of P1640 fits files

Output: spot_array: Nfiles x Nspots x Nchan x 2 array of (row, col) positions spot_phot: Nfils x Nspots x Nchan array of spot photometry

pyklip.instruments.P1640_support.P1640spots.**get_star_positions**(*spot_array*)
Get the center of a set of 4 spots for a single cube Input:

spot_locations: Nspot x Nlambda x 2 array of [row, col] spot positions

Returns Nlambda x 2 array of [row, col] star positions

Return type star_array

pyklip.instruments.P1640_support.P1640spots.**guess_grid_spot_loc**(*img*)
get max pixel as initial guess of location

pyklip.instruments.P1640_support.P1640spots.**make_mask_bar**(*img_shape*, *center*, *angle*, *width*)
Make a bar mask where all the pixels inside a bar through the center of the image within some width are 1 and everything outside is 0 Inputs:

img_shape: the shape of the image in (row, col) center: the center of the mask, in (row, col) angle: angle measured counterclockwise from vertical, default in deg width: with of bar in pixels

Returns

a masked array where the values inside the bar are False and outside the bar are True

Return type mask

pyklip.instruments.P1640_support.P1640spots.**make_mask_circle**(*img_shape*, *center*, *R*)
Make a circular mask, where everything inside a radius R around the center is False and outside is True Input:

img_shape: the shape of the image in (row, col) center: the center of the mask, in (row, col) R: the radius of the circle

Returns a masked array of shape img_shape

Return type mask

pyklip.instruments.P1640_support.P1640spots.**make_mask_donut**(*img_shape*, *center*, *R0*, *R1*)

Make a donut mask centered on 'center' where the inside of the donut is False and the outside of the donut is True

pyklip.instruments.P1640_support.P1640spots.**make_mask_grid_spots**(*img_shape*, *centers*, *rotated_spots=False*, *nchan=32*)

Make a mask that shows only the grid spots Input:

img_shape: the shape of the image to mask in (row, col) centers: (Nchan x 2) array of centers of the mask in (row, col) rotated_spots: [False] make mask for normal (False) or rotated (True) grid spots nchan: number of spectral channels in the cube

Returns Nspot x Nchan cube of masks

Return type masks

pyklip.instruments.P1640_support.P1640spots.**make_mask_half_img**(*img_shape*, *center*, *angle*)

Mask half the image, cutting it through the center at an arbitrary angle. Angle is measured *counterclockwise* from vertical, and should be an astropy units object, otherwise assume degrees. Input:

img_shape: shape of image in (row, col) center: the center of the mask in (row, col) angle: angle measured counterclockwise from vertical, default in deg

Output:

mask: masked_array with a plane running through point (center) at angle (angle)

pyklip.instruments.P1640_support.P1640spots.**make_mask_refined_grid_spots**(*img_shape*, *centers*, *spots*, *nchan=32*)

Make a new set of masks that are centered on the interpolated grid spot locations Input:

img_shape: x- and y-dimensions of image centers: nchan x 2 array of star positions spots: num_spots x nchan x 2 array of spot positions

pyklip.instruments.P1640_support.P1640spots.**write_spots_to_file**(*data_filepath*, *spot_positions*, *output_dir=None*, *spotid=None*, *ext=None*, *overwrite=True*)

Write one file for each spot to the directory defined at the top of this file. Output file name is data_filename -fits +spoti.csv. Format is (row, col). Will overwrite existing files. Input:

data_filename: the base name of the file with the spots spot_positions: Nspot x Nchan x 2 array of spot positions output_dir: directory to write the output files overwrite: (True) overwrite existing spot files spotid: (-spoti) identifier for the 4 different spot files ext: (csv) file extension

Returns None writes a file to the output dir whose name corresponds to the cube used to generate the spots + spotidN.ext (N is 0-3)

pyklip.instruments.P1640_support.P1640spots.**write_spots_to_header**(*spots*, *fitsfile*)

Write the spot positions to a fits header Input:

spots: 4 x Nchan x 2 array fitsfile: full path to a fits file whose header you want to modify

pyklip.instruments.P1640_support.P1640utils module

pyklip.instruments.P1640_support.P1640utils.**centroid_image**(*orig_img*)

Centroid an image - weighted sum of pixels Input:

orig_img: 2D array

Returns [y,x] floating-point coordinates of the centroid

pyklip.instruments.P1640_support.P1640utils.**clean_bad_pixels**(*img*, *boxrad=2*, *thresh=3*)

Clean the image of outlier pixels using a median filter. Input:

img: 2-d array boxrad: 1/2 the fitler size (2*boxrad+1) thresh: threshold (in stdev) for deciding a hot pixel

Returns 2_d array where hot pixels have been replaced by median values

Return type cleaned_img

pyklip.instruments.P1640_support.P1640utils.**clean_bad_pixels_cube**(*cube*, *boxrad=2*, *thresh=10*)

Clean the image of outlier pixels using a median filter. Input:

cube: 3-D data cube boxrad: 1/2 the fitler size (2*boxrad+1) thresh: threshold (in stdev) for deciding a hot pixel

pyklip.instruments.P1640_support.P1640utils.**find_bad_pix**(*img*, *median_img*, *std_img*, *thresh=3*)

Find the bad pixels

pyklip.instruments.P1640_support.P1640utils.**get_PSF_center**(*cube*, *refchan=26*, *fine=False*)

Return the PSF center at the pixel level (default) or subpixel level (fine=True) Input:

cube: Nlambda x Npix x Npix datacube refchan(=26): Reference channel for the initial center estimate fine_centering(=False): After getting a rough estimate of the center, centroid the image

Returns Nlamdba x 2 array of pixel indices for the PSF center

pyklip.instruments.P1640_support.P1640utils.**get_cube_xsection**(*orig_cube*, *center*, *width*)

Select the cross-section of a cube centered in center with 1/2-width width Input:

orig_cube: Nlambda x Npix x Npix datacube center: [row, col] index width: 1/2-width of cross-section

Returns Nlambda x (2*width+1) x (2*width+1) cube cross-section

Return type cube_cutout

pyklip.instruments.P1640_support.P1640utils.**get_encircled_energy_cube**(*cube*, *frac=0.5*, *re-fchan=26*)

Get the fractional encircled energy of a PSF in each channel of a datacube. Basically a wrapper for get_encircled_energy_image Input:

core_cube: unocculted core cube Nlambda x Npix x Npix frac: encircled energy cutoff

Returns [starx, stary, radius, flux]

Return type Pandas Dataframe with Nlambda columns

pyklip.instruments.P1640_support.P1640utils.**get_encircled_energy_image**(*im*, *cen-ter*, *frac=0.5*)

Given an image, find the fraction of encircled energy around the center. Input:

im: unocculted core cube Npix x Npix frac: encircled energy cutoff

Returns [starx, stary, radius, flux, bgnd_mean, bgnd_std, bgnd_npix]

Return type Pandas Series with the following indices

pyklip.instruments.P1640_support.P1640utils.**set_zeros_to_nan**(*data*)

PyKLIP expects values outside the detector to be set to nan. P1640 sets these (and also saturated pixels) to identically 0. Find all the zeros and convert them to nans Input:

data: N x Npix x Npix datacube or appended set of datacubes

Returns data with nans instead of zeros

Return type nandata

pyklip.instruments.P1640_support.P1640utils.**table_to_TableHDU**(*table*, *kwargs={}*)

Accept a table with a .colnames element and return it as an astropy fits.TableHDU object. Only works with floating-point data atm. Input:

table: astropy.table.Table object kwargs: dict of keywords and arguments to pass to the HDU

Returns fits TableHDU with an empty header

Return type TableHDU

Module contents

pyklip.instruments.utils package

Submodules

pyklip.instruments.utils.nair module

pyklip.instruments.utils.nair.**GetCoeff**(*i*, *P*, *T*, *H*)
Calculate the coefficients for the polynomial series expansion of index of refraction (Mathar (2008)) ***Only valid for between 1.3 and 2.5 microns!

Inputs: i: degree of expansion in wavenumber P: Pressure in Pa T: Temperature in Kelvin H: relative humiditiy in % (i.e. between 0 and 100)

Returns Coefficient [cm^-i]

Return type coeff

pyklip.instruments.utils.nair.**nMathar**(*wv*, *P*, *T*, *H=10*)
Calculate the index of refraction as given by Mathar (2008): http://arxiv.org/pdf/physics/0610256v2.pdf ***Only valid for between 1.3 and 2.5 microns!

Inputs: wv: wavelength in microns P: Pressure in Pa T: Temperature in Kelvin H: relative humiditiy in % (i.e. between 0 and 100)

Returns index of refratoin

Return type n

pyklip.instruments.utils.nair.**nRoe**(*wv*, *P*, *T*, *fh20=0.0*)
Compute n for air from the formula in Henry Roe's PASP paper: http://arxiv.org/pdf/astro-ph/0201273v1.pdf which in turn is referenced from Allen's Astrophysical Quantities.

Inputs: wv: wavelength in microns P: pressure in Pascal T: temperature in Kelvin fh20:fractional partial pressure of water (typically between 0 and 4%)

Returns index of refraction of air

Return type n

Module contents

Submodules

pyklip.instruments.GPI module

pyklip.instruments.Instrument module

class pyklip.instruments.Instrument.**Data**
Bases: object

Abstract Class with the required fields and methods that need to be implemented

input
Array of shape (N,y,x) for N images of shape (y,x)

centers
Array of shape (N,2) for N centers in the format [x_cent, y_cent]

filenums
Array of size N for the numerical index to map data to file that was passed in

filenames
Array of size N for the actual filepath of the file that corresponds to the data

PAs
Array of N for the parallactic angle rotation of the target (used for ADI) [in degrees]

wvs
Array of N wavelengths of the images (used for SDI) [in microns]. For polarization data, defaults to "None"

wcs
Array of N wcs astormetry headers for each image.

IWA
a floating point scalar (not array). Specifies to inner working angle in pixels

OWA
(optional) specifies outer working angle in pixels

output
Array of shape (b, len(files), len(uniq_wvs), y, x) where b is the number of different KL basis cutoffs

creator
(optional) string for creator of the data (used to identify pipelines that call pyklip)

klipparams
(optional) a string that saves the most recent KLIP parameters

flipx
(optional) True by default. Determines whether a relfection about the x axis is necessary to rotate image North-up East left

readdata()
reread in the dadta

savedata()
save a specified data in the GPI datacube format (in the 1st extension header)

calibrate_output()
flux calibrate the output data

IWA
a floating point scalar (not array). Specifies to inner working angle in pixels

PAs
Array of N for the parallactic angle rotation of the target (used for ADI) [in degrees]

calibrate_output(*img*, *spectral=False*)
Calibrates the flux of an output image. Can either be a broadband image or a spectral cube depending on if the spectral flag is set.

Assumes the broadband flux calibration is just multiplication by a single scalar number whereas spectral datacubes may have a separate calibration value for each wavelength

Parameters

- **img** – unclaibrated image. If spectral is not set, this can either be a 2-D or 3-D broadband image where the last two dimensions are [y,x] If specetral is True, this is a 3-D spectral cube with shape [wv,y,x]
- **spectral** – if True, this is a spectral datacube. Otherwise, it is a broadband image.

Returns calibrated image of the same shape

Return type calib_img

centers
Image centers. Shape of (N, 2) where the 2nd dimension is [x,y] pixel coordinate (in that order)

filenames
Array of size N for the actual filepath of the file that corresponds to the data

filenums
Array of size N for the numerical index to map data to file that was passed in

input
Input Data. Shape of (N, y, x)

output
Array of shape (b, len(files), len(uniq_wvs), y, x) where b is the number of different KL basis cutoffs

readdata(*filepaths*)
Reads in the data from the files in the filelist and writes them to fields

static savedata(*filepath*, *data*, *klipparams=None*, *filetype=''*, *zaxis=None*, *more_keywords=None*)
Saves data for this instrument

Parameters

- **filepath** – filepath to save to
- **data** – data to save
- **klipparams** – a string of KLIP parameters. Write it to the 'PSFPARAM' keyword
- **filtype** – type of file (e.g. "KL Mode Cube", "PSF Subtracted Spectral Cube"). Wrriten to 'FILETYPE' keyword
- **zaxis** – a list of values for the zaxis of the datacub (for KL mode cubes currently)
- **more_keywords** (*dictionary*) – a dictionary {key: value, key:value} of header keywords and values which will written into the primary header

wcs
Array of N wcs astormetry headers for each image.

wvs
Array of N wavelengths (used for SDI) [in microns]. For polarization data, defaults to "None"

class pyklip.instruments.Instrument.**GenericData**(*input_data*, *centers*, *parangs=None*, *wvs=None*, *IWA=0*, *filenames=None*)
Bases: *pyklip.instruments.Instrument.Data*

Basic class to interface with a basic direct imaging dataset

input
Array of shape (N,y,x) for N images of shape (y,x)

centers
Array of shape (N,2) for N centers in the format [x_cent, y_cent]

filenums
Array of size N for the numerical index to map data to file that was passed in

filenames
Array of size N for the actual filepath of the file that corresponds to the data

PAs
Array of N for the parallactic angle rotation of the target (used for ADI) [in degrees]

wvs
Array of N wavelengths of the images (used for SDI) [in microns]. For polarization data, defaults to "None"

wcs
Array of N wcs astormetry headers for each image.

IWA
a floating point scalar (not array). Specifies to inner working angle in pixels

output
Array of shape (b, len(files), len(uniq_wvs), y, x) where b is the number of different KL basis cutoffs

Parameters

- **input_data** – either a 1-D list of filenames to read in, or a 3-D cube of all data (N, y, x)
- **centers** – array of shape (N,2) for N centers in the format [x_cent, y_cent]
- **parangs** – Array of N for the parallactic angle rotation of the target (used for ADI) [in degrees]
- **wvs** – Array of N wavelengths of the images (used for SDI) [in microns]. For polarization data, defaults to "None"
- **IWA** – a floating point scalar (not array). Specifies to inner working angle in pixels
- **filenames** – Array of size N for the actual filepath of the file that corresponds to the data

IWA

PAs

calibrate_output(*img*, *spectral=False*)
Calibrates the flux of an output image. Can either be a broadband image or a spectral cube depending on if the spectral flag is set.

Assumes the broadband flux calibration is just multiplication by a single scalar number whereas spectral datacubes may have a separate calibration value for each wavelength

Parameters

- **img** – unclaibrated image. If spectral is not set, this can either be a 2-D or 3-D broadband image where the last two dimensions are [y,x] If specetral is True, this is a 3-D spectral cube with shape [wv,y,x]
- **spectral** – if True, this is a spectral datacube. Otherwise, it is a broadband image.

Returns calibrated image of the same shape

Return type calib_img

centers

filenames

filenums

input

output

readdata (*filepaths*)
Reads in the data from the files in the filelist and writes them to fields.

savedata (*filepath*, *data*, *klipparams=None*, *filetype=''*, *zaxis=None*, *more_keywords=None*)
Saves data for this instrument

Parameters

- **filepath** – filepath to save to
- **data** – data to save
- **klipparams** – a string of KLIP parameters. Write it to the 'PSFPARAM' keyword
- **filtype** – type of file (e.g. "KL Mode Cube", "PSF Subtracted Spectral Cube"). Wrriten to 'FILETYPE' keyword
- **zaxis** – a list of values for the zaxis of the datacub (for KL mode cubes currently)
- **more_keywords** (*dictionary*) – a dictionary {key: value, key:value} of header keywords and values which will written into the primary header

wcs

wvs

pyklip.instruments.NIRC2 module

pyklip.instruments.P1640 module

pyklip.instruments.SPHERE module

class pyklip.instruments.SPHERE.**Ifs**(*data_cube*, *psf_cube*, *info_fits*, *wavelength_info*, *psf_cube_size=21*, *nan_mask_boxsize=9*, *IWA=0.15*)
Bases: *pyklip.instruments.Instrument.Data*

A sequence of SPHERE IFS Data.

Parameters

- **data_cube** – FITS file with a 4D-cube (Nfiles, Nwvs, Ny, Nx) with all IFS coronagraphic data
- **psf_cube** – FITS file with a 3-D (Nwvs, Ny, Nx) PSF cube
- **info_fits** – FITS file with a table in the 1st ext hdr with parallactic angle info
- **wavelenegth_info** – FITS file with a 1-D array (Nwvs) of the wavelength sol'n of a cube
- **psf_cube_size** – size of the psf cube to save (length along 1 dimension)
- **nan_mask_boxsize** – size of box centered around any pixel <= 0 to mask as NaNs
- **IWA** – inner working angle of the data in arcsecs

input
Array of shape (N,y,x) for N images of shape (y,x)

centers
Array of shape (N,2) for N centers in the format [x_cent, y_cent]

filenums
Array of size N for the numerical index to map data to file that was passed in

filenames
Array of size N for the actual filepath of the file that corresponds to the data

PAs
Array of N for the parallactic angle rotation of the target (used for ADI) [in degrees]

wvs
Array of N wavelengths of the images (used for SDI) [in microns]. For polarization data, defaults to "None"

IWA
a floating point scalar (not array). Specifies to inner working angle in pixels

output
Array of shape (b, len(files), len(uniq_wvs), y, x) where b is the number of different KL basis cutoffs

psfs
Spectral cube of size (Nwv, psfy, psfx) where psf_cube_size defines the size of psfy, psfx.

psf_center
[x, y] location of the center of the PSF for a frame in self.psfs

flipx
True by default. Determines whether a relfection about the x axis is necessary to rotate image North-up East left

nfiles
number of datacubes

nwvs
number of wavelengths

IWA

PAs

calibrate_output (*img*, *spectral=False*, *units='contrast'*)

Calibrates the flux of an output image. Can either be a broadband image or a spectral cube depending on if the spectral flag is set.

Args:

img: unclaibrated image. If spectral is not set, this can either be a 2-D or 3-D broadband image where the last two dimensions are [y,x] If specetral is True, this is a 3-D spectral cube with shape [wv,y,x]

spectral: if True, this is a spectral datacube. Otherwise, it is a broadband image. units: currently only support "contrast" w.r.t central star

Return: img: calibrated image of the same shape (this is the same object as the input!!!)

centers

filenames

filenums

input

north_offset = -102.18

output

platescale = 0.007462

readdata (*filepaths*)
Reads in the data from the files in the filelist and writes them to fields

savedata (*filepath*, *data*, *klipparams=None*, *filetype=''*, *zaxis=None*, *more_keywords=None*)

Save SPHERE Data.

Args:

filepath: path to file to output data: 2D or 3D data to save klipparams: a string of klip parameters filetype: filetype of the object (e.g. "KL Mode Cube", "PSF Subtracted Spectral Cube") zaxis: a list of values for the zaxis of the datacub (for KL mode cubes currently) more_keywords (dictionary) : a dictionary {key: value, key:value} of header keywords and values which will

written into the primary header

wcs

wvs

class pyklip.instruments.SPHERE.**Irdis**(*data_cube*, *psf_cube*, *info_fits*, *wavelength_str*, *psf_cube_size=21*, *IWA=0.2*)
Bases: *pyklip.instruments.Instrument.Data*

A sequence of SPHERE IRDIS Data.

Parameters

- **data_cube** – FITS file with a 4D-cube (Nfiles, Nwvs, Ny, Nx) with all IFS coronagraphic data
- **psf_cube** – FITS file with a 3-D (Nwvs, Ny, Nx) PSF cube
- **info_fits** – FITS file with a table in the 1st ext hdr with parallactic angle info
- **wavelength_str** – string to specifiy the band (e.g. "H2H3", "K1K2")
- **psf_cube_size** – size of the psf cube to save (length along 1 dimension)
- **IWA** – inner working angle of the data in arcsecs

input
Array of shape (N,y,x) for N images of shape (y,x)

centers
Array of shape (N,2) for N centers in the format [x_cent, y_cent]

filenums
Array of size N for the numerical index to map data to file that was passed in

filenames
Array of size N for the actual filepath of the file that corresponds to the data

PAs
Array of N for the parallactic angle rotation of the target (used for ADI) [in degrees]

wvs
Array of N wavelengths of the images (used for SDI) [in microns]. For polarization data, defaults to "None"

IWA
a floating point scalar (not array). Specifies to inner working angle in pixels

output
Array of shape (b, len(files), len(uniq_wvs), y, x) where b is the number of different KL basis cutoffs

psfs
Spectral cube of size (2, psfy, psfx) where psf_cube_size defines the size of psfy, psfx.

psf_center
[x, y] location of the center of the PSF for a frame in self.psfs

flipx
True by default. Determines whether a relfection about the x axis is necessary to rotate image North-up East left

nfiles
number of datacubes

nwvs
number of wavelengths (i.e. 2 for dual band imaging)

IWA

PAs

calibrate_output (*img*, *spectral=False*, *units='contrast'*)

Calibrates the flux of an output image. Can either be a broadband image or a spectral cube depending on if the spectral flag is set.

Args:

img: unclaibrated image. If spectral is not set, this can either be a 2-D or 3-D broadband image where the last two dimensions are [y,x] If specetral is True, this is a 3-D spectral cube with shape [wv,y,x]

spectral: if True, this is a spectral datacube. Otherwise, it is a broadband image. units: currently only support "contrast" w.r.t central star

Return: img: calibrated image of the same shape (this is the same object as the input!!!)

centers

filenames

filenums

input

north_offset = -1.75

output

platescale = 0.012255

readdata (*filepaths*)
Reads in the data from the files in the filelist and writes them to fields

savedata (*filepath*, *data*, *klipparams=None*, *filetype=''*, *zaxis=None*, *more_keywords=None*)

Save SPHERE Data.

Args:

filepath: path to file to output data: 2D or 3D data to save klipparams: a string of klip parameters filetype: filetype of the object (e.g. "KL Mode Cube", "PSF Subtracted Spectral Cube") zaxis: a list of values for the zaxis of the datacub (for KL mode cubes currently) more_keywords (dictionary) : a dictionary {key: value, key:value} of header keywords and values which will

written into the primary header

wavelengths = {'K1K2': (2.1, 2.244), 'Y2Y3': (1.02, 1.073), 'H3H4': (1.667, 1.731), 'J2J3': (1.19, 1.27), 'H2H3': (1.587,

wcs

wvs

Module contents

pyklip.kpp package

Subpackages

pyklip.kpp.detection package

Submodules

pyklip.kpp.detection.CADIQuicklook module

class pyklip.kpp.detection.CADIQuicklook.**CADIQuicklook**(*inputDir=None*, *outputDir=None*, *mute=None*, *label=None*, *GOI_list_folder=None*, *overwrite=False*, *copy_save=None*)

Bases: *pyklip.kpp.utils.kppSuperClass.KPPSuperClass*

Class for CADI quicklook.

calculate()

Parameters **N** – Defines the width of the ring by the number of pixels it has to contain

Returns self.image the imput fits file.

check_existence()

Returns False

check_existence_noInit(*outputDir=None*, *folderName=None*)

Returns False

initialize(*inputDir=None*, *outputDir=None*, *folderName=None*, *compact_date=None*, *label=None*)

Initialize the non general inputs that are needed for the metric calculation and load required files.

For this super class it simply reads the input file including fits headers and store it in self.image. One can also overwrite inputDir, outputDir which is basically the point of this function. The file is assumed here to be a fits containing a 2D image or a GPI 3D cube (assumes 37 spectral slice).

Example for inherited classes: It can read the PSF cube or define the hat function. It can also read the template spectrum in a 3D scenario. It could also overwrite this function in case it needs to read multiple files or non fits file.

Parameters

- **inputDir** – If defined it allows filename to not include the whole path and just the filename. Files will be read from inputDir. Note tat inputDir might be redefined using initialize at any point. If inputDir is None then filename is assumed to have the absolute path.
- **outputDir** – Directory where to create the folder containing the outputs. Note tat inputDir might be redefined using initialize at any point. If outputDir is None:

  If inputDir is defined: outputDir = inputDir+os.path.sep+"**planet_detec_**"
- **folderName** – Name of the folder containing the outputs. It will be located in outputDir. Default folder name is "default_out". The convention is to have one folder per spectral template. If the keyword METFOLDN is available in the fits file header then the keyword value is used no matter the input.
- **label** – Define the suffix to the output folder when it is not defined. cf outputDir. Default is "default".

Returns None

load()

Returns None

save()

Returns None

pyklip.kpp.detection.ROC module

class pyklip.kpp.detection.ROC.**ROC**(*filename*, *filename_detec*, *inputDir=None*, *outputDir=None*, *mute=None*, *N_threads=None*, *label=None*, *detec_distance=None*, *ignore_distance=None*, *GOI_list_folder=None*, *threshold_sampling=None*, *overwrite=False*, *IWA=None*, *OWA=None*)

Bases: *pyklip.kpp.utils.kppSuperClass.KPPSuperClass*

Class for SNR calculation.

calculate()

Parameters **N** – Defines the width of the ring by the number of pixels it has to contain

Returns self.image the imput fits file.

check_existence()

Returns False

initialize(*inputDir=None*, *outputDir=None*, *folderName=None*, *compact_date=None*, *label=None*)

Initialize the non general inputs that are needed for the metric calculation and load required files.

For this super class it simply reads the input file including fits headers and store it in self.image. One can also overwrite inputDir, outputDir which is basically the point of this function. The file is assumed here to be a fits containing a 2D image or a GPI 3D cube (assumes 37 spectral slice).

Example for inherited classes: It can read the PSF cube or define the hat function. It can also read the template spectrum in a 3D scenario. It could also overwrite this function in case it needs to read multiple files or non fits file.

Parameters

- **inputDir** – If defined it allows filename to not include the whole path and just the filename. Files will be read from inputDir. Note tat inputDir might be redefined using initialize at any point. If inputDir is None then filename is assumed to have the absolute path.
- **outputDir** – Directory where to create the folder containing the outputs. Note tat inputDir might be redefined using initialize at any point. If outputDir is None:

  If inputDir is defined: outputDir = inputDir+os.path.sep+"**planet_detec_**"
- **folderName** – Name of the folder containing the outputs. It will be located in outputDir. Default folder name is "default_out". The convention is to have one folder per spectral template. If the keyword METFOLDN is available in the fits file header then the keyword value is used no matter the input.
- **label** – Define the suffix to the output folder when it is not defined. cf outputDir. Default is "default".

Returns None

load()

Returns None

save()

Returns None

pyklip.kpp.detection.ROC.**gather_ROC**(*filename_filter*, *mute=False*)

Build the combined ROC curve from individual frame ROC curve. It looks for all the file matching filename_filter using glob.glob and then add each individual ROC to build the master ROC.

Plot master_N_false_pos vs master_N_true_detec to get a ROC curve.

Parameters

- **filename_filter** – Filename filter with wild characters indicating which files to pick
- **mute** – If True, mute prints. Default is False.

Returns threshold_sampling,master_N_false_pos,master_N_true_detec: threshold_sampling: The metric sampling. It is the curve parametrization. master_N_false_pos: Number of false positives as a function of threshold_sampling master_N_true_detec: Number of true positives as a function of threshold_sampling

pyklip.kpp.detection.ROC.**gather_multiple_ROCs**(*base_dir*, *filename_filter_list*, *mute=False*, *epoch_suffix=None*, *stars2ignore=None*, *band=None*)

Build the multiple combined ROC curve from individual frame ROC curve while making sure they have the same inputs. If the folders are organized following the convention below then it will make sure there is a ROC file for each filename_filter in each epoch. Otherwise it skips the epoch.

The folders need to be organized as: base_dir/TARGET/autoreduced/EPOCH_Spec/filename_filter

In the function TARGET and EPOCH are wild characters.

It looks for all the file matching filename_filter using glob.glob and then add each individual ROC to build the master ROC.

Plot master_N_false_pos vs master_N_true_detec to get a ROC curve.

Parameters

- **base_dir** – Base directory from which the file search go.
- **filename_filter** – Filename filter with wild characters indicating which files to pick.

- **mute** – If True, mute prints. Default is False.

Returns threshold_sampling,master_N_false_pos,master_N_true_detec: threshold_sampling: The metric sampling. It is the curve parametrization. master_N_false_pos: Number of false positives as a function of threshold_sampling master_N_true_detec: Number of true positives as a function of threshold_sampling

pyklip.kpp.detection.ROC.**get_all_false_pos**(*base_dir*, *filename_filter_list*, *threshold*, *mute=False*, *epoch_suffix=None*, *stars2ignore=None*, *IFSfilter=None*)

Build the multiple combined ROC curve from individual frame ROC curve while making sure they have the same inputs. If the folders are organized following the convention below then it will make sure there is a ROC file for each filename_filter in each epoch. Otherwise it skips the epoch.

The folders need to be organized as: base_dir/TARGET/autoreduced/EPOCH_Spec/filename_filter

In the function TARGET and EPOCH are wild characters.

It looks for all the file matching filename_filter using glob.glob and then add each individual ROC to build the master ROC.

Plot master_N_false_pos vs master_N_true_detec to get a ROC curve.

Parameters

- **base_dir** – Base directory from which the file search go.
- **filename_filter** – Filename filter with wild characters indicating which files to pick.
- **mute** – If True, mute prints. Default is False.

Returns threshold_sampling,master_N_false_pos,master_N_true_detec: threshold_sampling: The metric sampling. It is the curve parametrization. master_N_false_pos: Number of false positives as a function of threshold_sampling master_N_true_detec: Number of true positives as a function of threshold_sampling

pyklip.kpp.detection.ROC.**get_candidates**(*base_dir*, *filename_filter_list*, *threshold*, *mute=False*, *epoch_suffix=None*, *IWA=None*, *OWA=None*, *GOI_list_folder=None*, *ignore_distance=None*, *detec_distance=None*, *stars2ignore=None*)

Build the multiple combined ROC curve from individual frame ROC curve while making sure they have the same inputs. If the folders are organized following the convention below then it will make sure there is a ROC file for each filename_filter in each epoch. Otherwise it skips the epoch.

The folders need to be organized as: base_dir/TARGET/autoreduced/EPOCH_Spec/filename_filter

In the function TARGET and EPOCH are wild characters.

It looks for all the file matching filename_filter using glob.glob and then add each individual ROC to build the master ROC.

Plot master_N_false_pos vs master_N_true_detec to get a ROC curve.

Parameters

- **base_dir** – Base directory from which the file search go.
- **filename_filter** – Filename filter with wild characters indicating which files to pick.
- **mute** – If True, mute prints. Default is False.

Returns threshold_sampling,master_N_false_pos,master_N_true_detec: threshold_sampling: The metric sampling. It is the curve parametrization. master_N_false_pos: Number of false positives as a function of threshold_sampling master_N_true_detec: Number of true positives as a function of threshold_sampling

pyklip.kpp.detection.ROC.**get_metrics_stat**(*base_dir*, *filename_filter_list*, *IOWA*, *bins*, *GOI_list_folder*, *mute=False*, *epoch_suffix=None*, *stars2ignore=None*, *IFSfilter=None*)

Build the multiple combined ROC curve from individual frame ROC curve while making sure they have the same inputs. If the folders are organized following the convention below then it will make sure there is a ROC file for each filename_filter in each epoch. Otherwise it skips the epoch.

The folders need to be organized as: base_dir/TARGET/autoreduced/EPOCH_Spec/filename_filter

In the function TARGET and EPOCH are wild characters.

It looks for all the file matching filename_filter using glob.glob and then add each individual ROC to build the master ROC.

Plot master_N_false_pos vs master_N_true_detec to get a ROC curve.

Parameters

- **base_dir** – Base directory from which the file search go.
- **filename_filter** – Filename filter with wild characters indicating which files to pick.
- **mute** – If True, mute prints. Default is False.

Returns threshold_sampling,master_N_false_pos,master_N_true_detec: threshold_sampling: The metric sampling. It is the curve parametrization. master_N_false_pos: Number of false positives as a function of threshold_sampling master_N_true_detec: Number of true positives as a function of threshold_sampling

pyklip.kpp.detection.detection module

class pyklip.kpp.detection.detection.**Detection**(*filename*, *inputDir=None*, *outputDir=None*, *mute=None*, *N_threads=None*, *label=None*, *mask_radius=None*, *threshold=None*, *maskout_edge=None*, *overwrite=False*, *IWA=None*, *OWA=None*)

Bases: *pyklip.kpp.utils.kppSuperClass.KPPSuperClass*

Class for SNR calculation.

calculate()

Parameters **N** – Defines the width of the ring by the number of pixels it has to contain

Returns self.image the imput fits file.

check_existence()

Returns False

initialize(*inputDir=None*, *outputDir=None*, *folderName=None*, *compact_date=None*, *label=None*)

Initialize the non general inputs that are needed for the metric calculation and load required files.

For this super class it simply reads the input file including fits headers and store it in self.image. One can also overwrite inputDir, outputDir which is basically the point of this function. The file is assumed here to be a fits containing a 2D image or a GPI 3D cube (assumes 37 spectral slice).

Example for inherited classes: It can read the PSF cube or define the hat function. It can also read the template spectrum in a 3D scenario. It could also overwrite this function in case it needs to read multiple files or non fits file.

Parameters

- **inputDir** – If defined it allows filename to not include the whole path and just the filename. Files will be read from inputDir. Note tat inputDir might be redefined using initialize at any point. If inputDir is None then filename is assumed to have the absolute path.
- **outputDir** – Directory where to create the folder containing the outputs. Note tat inputDir might be redefined using initialize at any point. If outputDir is None:

  If inputDir is defined: outputDir = inputDir+os.path.sep+"**planet_detec_**"
- **folderName** – Name of the folder containing the outputs. It will be located in outputDir. Default folder name is "default_out". The convention is to have one folder per spectral template. If the keyword METFOLDN is available in the fits file header then the keyword value is used no matter the input.
- **label** – Define the suffix to the output folder when it is not defined. cf outputDir. Default is "default".

Returns None

load()

Returns None

save()

Returns None

pyklip.kpp.detection.quicklook module

class pyklip.kpp.detection.quicklook.**Quicklook**(*filename_proba*, *filename_detec*, *inputDir=None*, *outputDir=None*, *mute=None*, *label=None*, *GOI_list_folder=None*, *overwrite=False*, *copy_save=None*, *SNR=None*)

Bases: *pyklip.kpp.utils.kppSuperClass.KPPSuperClass*

Class for CADI quicklook.

calculate()

Parameters **N** – Defines the width of the ring by the number of pixels it has to contain

Returns self.image the imput fits file.

check_existence()

Returns False

check_existence_noInit(*outputDir=None*, *folderName=None*)

Returns False

initialize(*inputDir=None*, *outputDir=None*, *folderName=None*, *compact_date=None*, *label=None*)

Initialize the non general inputs that are needed for the metric calculation and load required files.

For this super class it simply reads the input file including fits headers and store it in self.image. One can also overwrite inputDir, outputDir which is basically the point of this function. The file is assumed here to be a fits containing a 2D image or a GPI 3D cube (assumes 37 spectral slice).

Example for inherited classes: It can read the PSF cube or define the hat function. It can also read the template spectrum in a 3D scenario. It could also overwrite this function in case it needs to read multiple files or non fits file.

Parameters

- **inputDir** – If defined it allows filename to not include the whole path and just the filename. Files will be read from inputDir. Note tat inputDir might be redefined using initialize at any point. If inputDir is None then filename is assumed to have the absolute path.
- **outputDir** – Directory where to create the folder containing the outputs. Note tat inputDir might be redefined using initialize at any point. If outputDir is None:

  If inputDir is defined: outputDir = inputDir+os.path.sep+"**planet_detec_**"
- **folderName** – Name of the folder containing the outputs. It will be located in outputDir. Default folder name is "default_out". The convention is to have one folder per spectral template. If the keyword METFOLDN is available in the fits file header then the keyword value is used no matter the input.
- **label** – Define the suffix to the output folder when it is not defined. cf outputDir. Default is "default".

Returns None

load()

Returns None

save()

Returns None

Module contents

pyklip.kpp.metrics package

Submodules

pyklip.kpp.metrics.FMMF module

pyklip.kpp.metrics.crossCorr module

class pyklip.kpp.metrics.crossCorr.**CrossCorr**(*filename*, *inputDir=None*, *outputDir=None*, *folderName=None*, *mute=None*, *N_threads=None*, *label=None*, *overwrite=False*, *kernel_type=None*, *kernel_width=None*, *collapse=None*, *weights=None*, *nans2zero=None*)

Bases: *pyklip.kpp.utils.kppSuperClass.KPPSuperClass*

Class for SNR calculation.

calculate()

Parameters **N** – Defines the width of the ring by the number of pixels it has to contain

Returns self.image the imput fits file.

check_existence()

Returns False

initialize(*inputDir=None*, *outputDir=None*, *folderName=None*, *compact_date=None*, *label=None*)

Initialize the non general inputs that are needed for the metric calculation and load required files.

For this super class it simply reads the input file including fits headers and store it in self.image. One can also overwrite inputDir, outputDir which is basically the point of this function. The file is assumed here to be a fits containing a 2D image or a GPI 3D cube (assumes 37 spectral slice).

Example for inherited classes: It can read the PSF cube or define the hat function. It can also read the template spectrum in a 3D scenario. It could also overwrite this function in case it needs to read multiple files or non fits file.

Parameters

- **inputDir** – If defined it allows filename to not include the whole path and just the filename. Files will be read from inputDir. Note tat inputDir might be redefined using initialize at any point. If inputDir is None then filename is assumed to have the absolute path.
- **outputDir** – Directory where to create the folder containing the outputs. Note tat inputDir might be redefined using initialize at any point. If outputDir is None:

  If inputDir is defined: outputDir = inputDir+os.path.sep+"**planet_detec_**"
- **folderName** – Name of the folder containing the outputs. It will be located in outputDir. Default folder name is "default_out". The convention is to have one folder per spectral template. If the keyword METFOLDN is available in the fits file header then the keyword value is used no matter the input.
- **label** – Define the suffix to the output folder when it is not defined. cf outputDir. Default is "default".

Returns None

load()

Returns None

save()

Returns None

pyklip.kpp.metrics.shapeOrMF module

Module contents

pyklip.kpp.stat package

Submodules

pyklip.kpp.stat.contrast module

pyklip.kpp.stat.contrastFMMF module

class pyklip.kpp.stat.contrastFMMF.**ContrastFMMF**(*filename*, *filename_fakes=None*, *inputDir=None*, *outputDir=None*, *mute=None*, *N_threads=None*, *label=None*, *mask_radius=None*, *IOWA=None*, *GOI_list_folder=None*, *overwrite=False*, *contrast_filename=None*)

Bases: *pyklip.kpp.utils.kppSuperClass.KPPSuperClass*

Class for SNR calculation.

calculate()

Parameters **N** – Defines the width of the ring by the number of pixels it has to contain

Returns self.image the imput fits file.

check_existence()

Returns False

initialize(*inputDir=None*, *outputDir=None*, *folderName=None*, *compact_date=None*, *label=None*)

Initialize the non general inputs that are needed for the metric calculation and load required files.

For this super class it simply reads the input file including fits headers and store it in self.image. One can also overwrite inputDir, outputDir which is basically the point of this function. The file is assumed here to be a fits containing a 2D image or a GPI 3D cube (assumes 37 spectral slice).

Example for inherited classes: It can read the PSF cube or define the hat function. It can also read the template spectrum in a 3D scenario. It could also overwrite this function in case it needs to read multiple files or non fits file.

Parameters

- **inputDir** – If defined it allows filename to not include the whole path and just the filename. Files will be read from inputDir. Note tat inputDir might be redefined using initialize at any point. If inputDir is None then filename is assumed to have the absolute path.
- **outputDir** – Directory where to create the folder containing the outputs. Note tat inputDir might be redefined using initialize at any point. If outputDir is None:

  If inputDir is defined: outputDir = inputDir+os.path.sep+"**planet_detec_**"
- **folderName** – Name of the folder containing the outputs. It will be located in outputDir. Default folder name is "default_out". The convention is to have one folder per spectral template. If the keyword METFOLDN is available in the fits file header then the keyword value is used no matter the input.
- **label** – Define the suffix to the output folder when it is not defined. cf outputDir. Default is "default".

Returns None

load()

Returns None

save()

Returns None

pyklip.kpp.stat.contrastFMMF.**gather_contrasts**(*base_dir*, *filename_filter_list*, *mute=False*, *epoch_suffix=None*, *cont_name_list=None*, *band=None*, *stars2ignore=None*)

Build the multiple combined ROC curve from individual frame ROC curve while making sure they have the same inputs. If the folders are organized following the convention below then it will make sure there is a ROC file for each filename_filter in each epoch. Otherwise it skips the epoch.

The folders need to be organized as: base_dir/TARGET/autoreduced/EPOCH_Spec/filename_filter

In the function TARGET and EPOCH are wild characters.

It looks for all the file matching filename_filter using glob.glob and then add each individual ROC to build the master ROC.

Plot master_N_false_pos vs master_N_true_detec to get a ROC curve.

Parameters

- **base_dir** – Base directory from which the file search go.
- **filename_filter** – Filename filter with wild characters indicating which files to pick.
- **mute** – If True, mute prints. Default is False.

Returns threshold_sampling,master_N_false_pos,master_N_true_detec: threshold_sampling: The metric sampling. It is the curve parametrization. master_N_false_pos: Number of false positives as a function of threshold_sampling master_N_true_detec: Number of true positives as a function of threshold_sampling

pyklip.kpp.stat.stat module

class pyklip.kpp.stat.stat.**Stat**(*filename*, *filename_noPlanets=None*, *inputDir=None*, *outputDir=None*, *folderName=None*, *mute=None*, *N_threads=None*, *label=None*, *mask_radius=None*, *IOWA=None*, *N=None*, *Dr=None*, *r_step=None*, *type=None*, *rm_edge=None*, *GOI_list_folder=None*, *overwrite=False*, *kernel_type=None*, *kernel_width=None*, *image_wide=None*, *collapse=None*, *weights=None*, *nans2zero=None*)

Bases: *pyklip.kpp.utils.kppSuperClass.KPPSuperClass*

Class for SNR calculation.

calculate()

Parameters **N** – Defines the width of the ring by the number of pixels it has to contain

Returns self.image the imput fits file.

check_existence()

Returns False

initialize(*inputDir=None*, *outputDir=None*, *folderName=None*, *compact_date=None*, *label=None*)

Initialize the non general inputs that are needed for the metric calculation and load required files.

For this super class it simply reads the input file including fits headers and store it in self.image. One can also overwrite inputDir, outputDir which is basically the point of this function. The file is assumed here to be a fits containing a 2D image or a GPI 3D cube (assumes 37 spectral slice).

Example for inherited classes: It can read the PSF cube or define the hat function. It can also read the template spectrum in a 3D scenario. It could also overwrite this function in case it needs to read multiple files or non fits file.

Parameters

- **inputDir** – If defined it allows filename to not include the whole path and just the filename. Files will be read from inputDir. Note tat inputDir might be redefined using initialize at any point. If inputDir is None then filename is assumed to have the absolute path.
- **outputDir** – Directory where to create the folder containing the outputs. Note tat inputDir might be redefined using initialize at any point. If outputDir is None:

  If inputDir is defined: outputDir = inputDir+os.path.sep+"**planet_detec_**"
- **folderName** – Name of the folder containing the outputs. It will be located in outputDir. Default folder name is "default_out". The convention is to have one folder per spectral template. If the keyword METFOLDN is available in the fits file header then the keyword value is used no matter the input.
- **label** – Define the suffix to the output folder when it is not defined. cf outputDir. Default is "default".

Returns None

load()

Returns None

save()

Returns None

pyklip.kpp.stat.statPerPix module

class pyklip.kpp.stat.statPerPix.**StatPerPix**(*filename*, *inputDir=None*, *outputDir=None*, *mute=None*, *N_threads=None*, *label=None*, *mask_radius=None*, *IOWA=None*, *N=None*, *Dr=None*, *Dth=None*, *type=None*, *rm_edge=None*, *GOI_list_folder=None*, *overwrite=False*, *kernel_type=None*, *kernel_width=None*, *filename_noPlanets=None*, *collapse=None*, *weights=None*, *resolution=None*, *folderName=None*)

Bases: *pyklip.kpp.utils.kppSuperClass.KPPSuperClass*

Class for SNR calculation.

calculate()

Parameters **N** – Defines the width of the ring by the number of pixels it has to contain

Returns self.image the imput fits file.

check_existence()

Returns False

initialize(*inputDir=None*, *outputDir=None*, *folderName=None*, *compact_date=None*, *label=None*)

Initialize the non general inputs that are needed for the metric calculation and load required files.

For this super class it simply reads the input file including fits headers and store it in self.image. One can also overwrite inputDir, outputDir which is basically the point of this function. The file is assumed here to be a fits containing a 2D image or a GPI 3D cube (assumes 37 spectral slice).

Example for inherited classes: It can read the PSF cube or define the hat function. It can also read the template spectrum in a 3D scenario. It could also overwrite this function in case it needs to read multiple files or non fits file.

Parameters

- **inputDir** – If defined it allows filename to not include the whole path and just the filename. Files will be read from inputDir. Note tat inputDir might be redefined using initialize at any point. If inputDir is None then filename is assumed to have the absolute path.
- **outputDir** – Directory where to create the folder containing the outputs. Note tat inputDir might be redefined using initialize at any point. If outputDir is None:

  If inputDir is defined: outputDir = inputDir+os.path.sep+"**planet_detec_**"
- **folderName** – Name of the folder containing the outputs. It will be located in outputDir. Default folder name is "default_out". The convention is to have one folder per spectral template. If the keyword METFOLDN is available in the fits file header then the keyword value is used no matter the input.
- **label** – Define the suffix to the output folder when it is not defined. cf outputDir. Default is "default".

Returns None

load()

Returns None

save()

Returns None

pyklip.kpp.stat.statPerPix_utils module

pyklip.kpp.stat.statPerPix_utils.**get_image_stat_map_perPixMasking**(*image*, *image_without_planet*, *mask_radius=7*, *IOWA=None*, *N=None*, *centroid=None*, *mute=True*, *N_threads=None*, *Dr=None*, *Dth=None*, *type='SNR'*, *resolution=None*)

Calculate the SNR or probability (tail distribution) of a given image on a per pixel basis.

Parameters

- **image** – The image or cubes for which one wants the statistic.

- **image_without_planet** – Same as image but where real signal has been masked out. The code will actually use map to calculate the standard deviation or the density function.
- **mask_radius** – Radius of the mask used around the current pixel when use_mask_per_pixel = True.
- **IOWA** – (IWA,OWA) inner working angle, outer working angle. It defines boundary to the zones in which the statistic is calculated.
- **N** – Defines the width of the ring by the number of pixels it has to include. The width of the annuli will therefore vary with sepration.
- **centroid** – Define the cente rof the image. Default is x_cen = np.ceil((nx-1)/2) ; y_cen = np.ceil((ny-1)/2)
- **mute** – Won't print any logs.
- **N_threads** – Number of threads to be used. If None run sequentially.
- **Dr** – If not None defines the width of the ring as Dr. N is then ignored if Dth is defined as well.
- **Dth** – Define the angular size of a sector in degree (will apply for either Dr or N)
- **type** – Indicate the type of statistic to be calculated. If "SNR" (default) simple stddev calculation and returns SNR. If "stddev" returns the pure standard deviation map. If "proba" triggers proba calculation with pdf fitting.

Returns The statistic map for image.

pyklip.kpp.stat.statPerPix_utils.**get_image_stat_map_perPixMasking_threadTask**(*row_indices*, *col_indices*, *image*, *image_without_planet*, *x_grid*, *y_grid*, *N*, *mask_radius*, *firstZone_radii*, *lastZone_radii*, *Dr=None*, *Dth=None*, *type='SNR'*, *resolution=None*)

Calculate the SNR or probability (tail distribution) for some pixels in image on a per pixel basis. The pixels are defined by row_indices and col_indices.

This function is used for parallelization

Parameters

- **row_indices** – The row indices of images for which we want the statistic.
- **col_indices** – The column indices of images for which we want the statistic.

- **image** – The image or cubes for which one wants the statistic.
- **image_without_planet** – Same as image but where real signal has been masked out. The code will actually use map to calculate the standard deviation or the density function.
- **mask_radius** – Radius of the mask used around the current pixel when use_mask_per_pixel = True.
- **IOWA** – (IWA,OWA) inner working angle, outer working angle. It defines boundary to the zones in which the statistic is calculated.
- **N** – Defines the width of the ring by the number of pixels it has to include. The width of the annuli will therefore vary with sepration.
- **firstZone_radii** – (DISABLED) When N is not None it contains the meam_radius, the min radius and the max radius defining the first sector. The first sector in that case has includes roughly N pixels. For pixel too close to the inner edge this sector is taken by default.
- **lastZone_radii** – (DISABLED) Same as firstZone_radii for the outer edge.
- **centroid** – Define the cente rof the image. Default is x_cen = np.ceil((nx-1)/2) ; y_cen = np.ceil((ny-1)/2)
- **mute** – Won't print any logs.
- **N_threads** – Number of threads to be used. If None run sequentially.
- **Dr** – If not None defines the width of the ring as Dr. N is then ignored.
- **Dth** – Define the angular size of a sector in degree (will apply for either Dr or N)
- **type** – Indicate the type of statistic to be calculated. If "SNR" (default) simple stddev calculation and returns SNR. If "stddev" returns the pure standard deviation map. If "proba" triggers proba calculation with pdf fitting.

Returns The statistic map for image.

pyklip.kpp.stat.statPerPix_utils.**get_image_stat_map_perPixMasking_threadTask_star**(*params*)
Convert *f([1,2])* to *f(1,2)* call. It allows one to call get_image_probability_map_perPixMasking_threadTask() with a tuple of parameters.

pyklip.kpp.stat.stat_utils module

pyklip.kpp.stat.stat_utils.**get_cdf_model**(*data*, *interupt_plot=False*, *pure_gauss=False*)
Calculate a model CDF for some data.

/!This function is for some reason still a work in progress. JB could never decide what the best option was. But it should work even if the code is a mess.

Parameters

- **data** – arrays of samples from a random variable
- **interupt_plot** – Plot the histogram and model fit. It
- **pure_gauss** – Assume gaussian statistic. Do not fit exponential tails.

Returns (cdf_model,new_sampling,im_histo, center_bins) with: cdf_model: The cdf model = np.cumsum(pdf_model) pdf_model: The pdf model sampling: sampling of pdf/cdf_model im_histo: histogram from original data center_bins: bin centers for im_histo

pyklip.kpp.stat.stat_utils.**get_cube_stddev**(*cube*, *IOWA*, *N=2000*, *centroid=None*, *r_step=None*, *Dr=None*)

pyklip.kpp.stat.stat_utils.**get_image_PDF**(*image*, *IOWA*, *N=2000*, *centroid=None*, *r_step=None*, *Dr=None*, *image_wide=None*)

pyklip.kpp.stat.stat_utils.**get_image_stat_map**(*image*, *image_without_planet*, *IOWA=None*, *N=3000*, *centroid=None*, *r_step=5*, *mute=True*, *Dr=None*, *type='SNR'*, *image_wide=None*)

pyklip.kpp.stat.stat_utils.**get_image_stddev**(*image*, *IOWA=None*, *N=None*, *centroid=None*, *r_step=2*, *Dr=2*, *image_wide=None*, *resolution=None*)

pyklip.kpp.stat.stat_utils.**get_pdf_model**(*data*, *interupt_plot=False*, *pure_gauss=False*)

Calculate a model PDF for some data.

/!This function is for some reason still a work in progress. JB could never decide what the best option was. But it should work even if the code is a mess.

Parameters

- **data** – arrays of samples from a random variable
- **interupt_plot** – Plot the histogram and model fit. It
- **pure_gauss** – Assume gaussian statistic. Do not fit exponential tails.

Returns (pdf_model,new_sampling,im_histo, center_bins) with: pdf_model: The pdf model new_sampling: sampling of pdf_model im_histo: histogram from original data center_bins: bin centers for im_histo

Module contents

pyklip.kpp.utils package

Submodules

pyklip.kpp.utils.GOI module

pyklip.kpp.utils.GOI.**get_pos_known_objects**(*prihdr*, *exthdr*, *GOI_list_folder=None*, *xy=False*, *pa_sep=False*, *ignore_fakes=False*, *fakes_only=False*, *include_speckles=False*, *IWA=None*, *OWA=None*)

pyklip.kpp.utils.GOI.**make_GOI_list**(*outputDir*, *GOI_list_csv*, *GPI_TID_csv*)

Generate the GOI files from the GOI table and the TID table (queried from the database).

Parameters

- **outputDir** – Output directory in which to save the GOI files.
- **GOI_list_csv** – Table with the list of GOIs (including separation, PA...)
- **GPI_TID_csv** – Table giving the TID code for a given object name.

Returns One .csv file per target for which at list one GOI exists. The filename follows: [object]_GOI.csv. For e.g. c_Eri_GOI.csv.

pyklip.kpp.utils.GOI.**mask_known_objects**(*cube*, *prihdr*, *exthdr*, *GOI_list_folder=None*, *mask_radius=7*, *include_speckles=False*)

pyklip.kpp.utils.GPIimage module

pyklip.kpp.utils.GPIimage.**as2pix**(*sep_as*)

pyklip.kpp.utils.GPIimage.**get_IOWA**(*image*, *centroid=None*)
Get the IWA (inner working angle) of the central disk of nans and return the mask corresponding to the inner disk.

Parameters

- **image** – A GPI image with a disk full of nans at the center.
- **centroid** – center of the nan disk

Returns

pyklip.kpp.utils.GPIimage.**get_occ**(*image*, *centroid=None*)
Get the IWA (inner working angle) of the central disk of nans and return the mask corresponding to the inner disk.

Parameters

- **image** – A GPI image with a disk full of nans at the center.
- **centroid** – center of the nan disk

Returns

pyklip.kpp.utils.GPIimage.**pix2as**(*sep_pix*)

pyklip.kpp.utils.kppSuperClass module

class pyklip.kpp.utils.kppSuperClass.**KPPSuperClass**(*filename*, *inputDir=None*, *outputDir=None*, *folderName=None*, *mute=None*, *N_threads=None*, *label=None*, *overwrite=False*)
Bases: object

Super class for all kpop classes (ie FMMF, matched filter, shape, weighted collapse...). Has fall-back functions for all metric dependent calls so that each metric class does not need to implement functions it doesn't want to. It is not a completely empty function and includes features that are probably useful to most inherited class though one might decide to overwrite them. Here it simply returns the input fits file as read.

I should remove the option to set output dir in the class definition

calculate()
Calculate the metric map.

For this super class it returns the input fits file read in initialize().

Inherited classes: It could check at the output folder if the file with the right extension already exist.

Returns self.image the imput fits file.

check_existence()
Check if this metric has already been calculated for this file.

For this super class it returns False.

Inherited classes: It could check at the output folder if the file with the right extension already exist.

Returns False

init_new_spectrum(*spectrum*)
Function allowing the reinitialization of the class with a new spectrum without reinitializing everything.

Parameters **spectrum** – spectrum path relative to pykliproot + os.path.sep + "spectra" with pykliproot the directory in which pyklip is installed. It that case it should be a spectrum from Mark Marley. Instead of a path it can be a simple ndarray with the right dimension. Or by default it is a completely flat spectrum.

Returns None

initialize(*inputDir=None*, *outputDir=None*, *folderName=None*, *label=None*, *read=True*)
Initialize the non general inputs that are needed for the metric calculation and load required files.

For this super class it simply reads the input file including fits headers and store it in self.image. One can also overwrite inputDir, outputDir which is basically the point of this function. The file is assumed here to be a fits containing a 2D image or a GPI 3D cube (assumes 37 spectral slice).

Example for inherited classes: It can read the PSF cube or define the hat function. It can also read the template spectrum in a 3D scenario. It could also overwrite this function in case it needs to read multiple files or non fits file.

Parameters

- **inputDir** – If defined it allows filename to not include the whole path and just the filename. Files will be read from inputDir. Note tat inputDir might be redefined using initialize at any point. If inputDir is None then filename is assumed to have the absolute path.
- **outputDir** – Directory where to create the folder containing the outputs. Note tat inputDir might be redefined using initialize at any point. If outputDir is None:

  If inputDir is defined: outputDir = inputDir+os.path.sep+"**planet_detec_**"
- **folderName** – Name of the folder containing the outputs. It will be located in outputDir. Default folder name is "default_out". The convention is to have one folder per spectral template. Usually this folderName should be defined by the class itself and not by the user.
- **label** – Define the suffix to the output folder when it is not defined. cf outputDir. Default is "default".
- **read** – If true (default) read the fits file according to inputDir and filename.

Returns None

load()
Load the metric map if it already exist from self.outputDir+os.path.sep+self.folderName

For this super class it doesn't do anything.

Returns None

save()
Save the metric map as a fits file in self.outputDir+os.path.sep+self.folderName

For this super class it doesn't do anything.

Inherited classes: It should probably include new fits keywords with the metric parameters before saving the outputs.

Returns None

spectrum_iter_available()
Should indicate wether or not the class is equipped for iterating over spectra. If the metric requires a

spectrum one might one to iterate over several spectra without rereading the input files. In order to iterate over spectra the function init_new_spectrum() should be defined. spectrum_iter_available is a utility function for campaign data processing to know wether or not spectra the metric class should be iterated over different spectra.

In the case of this super class an therefore by default it returns False.

Returns False

pyklip.kpp.utils.mathfunc module

pyklip.kpp.utils.mathfunc.**LSQ_model_exp**(*x*, *y*, *m*, *alpha*)

pyklip.kpp.utils.mathfunc.**gauss2d**(*x*, *y*, *amplitude=1.0*, *xo=0.0*, *yo=0.0*, *sigma_x=1.0*, *sigma_y=1.0*, *theta=0*, *offset=0*)

pyklip.kpp.utils.mathfunc.**hat**(*x*, *y*, *radius*)

pyklip.kpp.utils.mathfunc.**model_exp**(*x*, *m*, *alpha*)

pyklip.kpp.utils.multiproc module

class pyklip.kpp.utils.multiproc.**NoDaemonPool**(*processes=None*, *initializer=None*, *initargs=()*, *maxtasksperchild=None*)

Bases: multiprocessing.pool.Pool

Process
alias of *NoDaemonProcess*

class pyklip.kpp.utils.multiproc.**NoDaemonProcess**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*)

Bases: multiprocessing.process.Process

daemon

Module contents

Submodules

pyklip.kpp.kppPerDir module

pyklip.kpp.kppPerDir.**kppPerDir**(*inputDir*, *obj_list*, *spec_path_list=None*, *outputDir=None*, *mute_error=True*, *compact_date_convention=None*)

pyklip.kpp.kppPerDir.**run**(*obj*)

Module contents

Submodules

pyklip.covars module

pyklip.covars.**matern32**(*x*, *y*, *sigmas*, *corr_len*)
Generates a Matern (nu=3/2) covariance matrix that assumes x/y has the same correlation length

C_ij = sigma_i sigma_j (1 + sqrt(3) r_ij / l) exp(-sqrt(3) r_ij / l)

Parameters

- **x** (*np.array*) – 1-D array of x coordinates
- **y** (*np.array*) – 1-D array of y coordinates
- **sigmas** (*np.array*) – 1-D array of errors on each pixel
- **corr_len** (*float*) – correlation length of the Matern function

Returns 2-D covariance matrix parameterized by the Matern function

Return type cov (np.array)

pyklip.covars.**sq_exp**(*x*, *y*, *sigmas*, *corr_len*)

Generates square exponential covariance matrix that assumes x/y has the same correlation length

C_ij = sigma_i sigma_j exp(-r_ij^2/[2 l^2])

Parameters

- **x** (*np.array*) – 1-D array of x coordinates
- **y** (*np.array*) – 1-D array of y coordinates
- **sigmas** (*np.array*) – 1-D array of errors on each pixel
- **corr_len** (*float*) – correlation length (i.e. standard deviation of Gaussian)
- **mode** (*string*) – either "numpy", "cython", or None, specifying the implementation of the kernel.

Returns 2-D covariance matrix parameterized by the Matern function

Return type cov (np.array)

pyklip.fakes module

pyklip.fakes.**LSQ_gauss2d**(*planet_image*, *x_grid*, *y_grid*, *a*, *x_cen*, *y_cen*, *sig*)

Calculate the squared norm of the residuals of the model with the data. Helper function for least square fit. The model is a 2d symmetric gaussian.

Parameters

- **planet_image** – stamp image (y,x) of the satellite spot.
- **x_grid** – x samples grid as given by meshgrid.
- **y_grid** – y samples grid as given by meshgrid.
- **a** – amplitude of the 2d gaussian
- **x_cen** – x center of the gaussian
- **y_cen** – y center of the gaussian
- **sig** – standard deviation of the gaussian

Returns Squared norm of the residuals

pyklip.fakes.**PSFcubefit**(*frame*, *xguess*, *yguess*, *searchrad=10*, *psfs_func_list=None*, *wave_index=None*, *residuals=False*)

Estimate satellite spot amplitude (peak value) by fitting a symmetric 2d gaussian. Fit parameters: x,y position, amplitude, standard deviation (same in x and y direction)

Parameters

- **frame** – the data - Array of size (y,x)
- **xguess** – x location to fit the 2d guassian to.
- **yguess** – y location to fit the 2d guassian to.
- **searchrad** – 1/2 the length of the box used for the fit
- **psfs_func_list** – List of spline fit function for the PSF_cube.
- **wave_index** – Index of the current wavelength. In [0,36] for GPI. Only used when psfs_func_list is not None.
- **residuals** – If True (Default = False) then calculate the residuals of the sat spot fit (gaussian or PSF cube).

Returns

scalar, Estimation of the peak flux of the satellite spot. ie Amplitude of the fitted gaussian.

Return type returned_flux

pyklip.fakes.**convert_pa_to_image_polar**(*pa*, *astr_hdr*)

Given a position angle (angle to North through East), calculate what polar angle theta (angle from +X CCW towards +Y) it corresponds to

Parameters

- **pa** – position angle in degrees
- **astr_hdr** – wcs astrometry header (astropy.wcs)

Returns polar angle in degrees

Return type theta

pyklip.fakes.**convert_polar_to_image_pa**(*theta*, *astr_hdr*)

Reversed engineer from covert_pa_to_image_polar by JB. Actually JB doesn't quite understand how it works...

Parameters

- **theta** – parallactic angle in degrees
- **astr_hdr** – wcs astrometry header (astropy.wcs)

Returns polar angle in degrees

Return type theta

pyklip.fakes.**gauss2d**(*x0*, *y0*, *peak*, *sigma*)

2d symmetric guassian function for guassfit2d

Parameters

- **x0,y0** – center of gaussian
- **peak** – peak amplitude of guassian
- **sigma** – stddev in both x and y directions

pyklip.fakes.**gaussfit2d**(*frame*, *xguess*, *yguess*, *searchrad=5*, *guessfwhm=3*, *guesspeak=1*, *refinefit=True*)

Fits a 2d gaussian to the data at point (xguess, yguess)

Parameters

- **frame** – the data - Array of size (y,x)

- **xguess,yguess** – location to fit the 2d guassian to (should be pretty accurate)
- **searchrad** – 1/2 the length of the box used for the fit
- **guessfwhm** – approximate fwhm to fit to
- **guesspeak** – approximate flux
- **refinefit** – whether to refine the fit of the position of the guess

Returns the peakflux of the gaussian fwhm: fwhm of the PFS in pixels xfit: x position (only chagned if refinefit is True) yfit: y position (only chagned if refinefit is True)

Return type peakflux

pyklip.fakes.**gaussfit2dLSQ**(*frame*, *xguess*, *yguess*, *searchrad=5*, *fit_centroid=False*, *residuals=False*)

Estimate satellite spot amplitude (peak value) by fitting a symmetric 2d gaussian. Fit parameters: x,y position, amplitude, standard deviation (same in x and y direction)

Parameters

- **frame** – the data - Array of size (y,x)
- **xguess** – x location to fit the 2d guassian to.
- **yguess** – y location to fit the 2d guassian to.
- **searchrad** – 1/2 the length of the box used for the fit
- **fit_centroid** – If False (default), disable the centroid fit and only fit the amplitude and the standard deviation
- **residuals** – If True (Default = False) then calculate the residuals of the sat spot fit (gaussian or PSF cube).

Returns

scalar, estimation of the peak flux of the satellite spot. ie Amplitude of the fitted gaussian.

Return type returned_flux

pyklip.fakes.**inject_disk**(*frames*, *centers*, *inputfluxes*, *astr_hdrs*, *pa*, *fwhm=3.5*)

Injects a fake disk into a dataset

Parameters

- **frames** – array of (N,y,x) for N is the total number of frames
- **centers** – array of size (N,2) of [x,y] coordiantes of the image center
- **intputfluxes** – array of size N of the peak flux of the fake disk in each frame OR array of 2-D models (North up East left) to inject into the data.

  (Disk is assumed to be centered at center of image)
- **astr_hdrs** – array of size N of the WCS headers
- **pa** – position angles angle (in degrees) of disk plane
- **fwhm** – if injecting a Gaussian disk (i.e inputfluxes is an array of floats), fwhm of Gaussian

Returns saves result in input "frames" variable

pyklip.fakes.**inject_planet**(*frames*, *centers*, *inputflux*, *astr_hdrs*, *radius*, *pa*, *fwhm=3.5*, *thetas=None*, *stampsize=None*)

Injects a fake planet into a dataset either using a Gaussian PSF or an input PSF

Parameters

- **frames** – array of (N,y,x) for N is the total number of frames
- **centers** – array of size (N,2) of [x,y] coordiantes of the image center
- **inputflux** – EITHER array of size N of the peak flux of the fake planet in each frame (will inject a Gaussian PSF) OR array of size (N,psfy,psfx) of template PSFs. The brightnesses should be scaled and the PSFs should be centered at the center of each of the template images
- **astr_hdrs** – array of size N of the WCS headers
- **radius** – separation of the planet from the star
- **pa** – position angle (in degrees) of planet
- **fwhm** – fwhm (in pixels) of gaussian
- **thetas** – ignore PA, supply own thetas (CCW angle from +x axis toward +y) array of size N
- **stampsize** – in pixels, the width of the square stamp to inject the image into. Defaults to 3*fwhm if None

Returns saves result in input "frames" variable

pyklip.fakes.**retrieve_planet**(*frames*, *centers*, *astr_hdrs*, *sep*, *pa*, *searchrad=7*, *guessfwhm=3.0*, *guesspeak=1*, *refinefit=True*, *thetas=None*)

Retrives the planet properties from a series of frames given a separation and PA

Parameters

- **frames** – frames of data to retrieve planet. Can be a single 2-D image ([y,x]) for a series/cube ([N,y,x])
- **centers** – coordiantes of the image center. Can be [2]-element lst or an array that matches array of frames [N,2]
- **astr_hdrs** – astr_hdrs, can be a single one or an array of N of them
- **sep** – radial distance in pixels
- **PA** – parallactic angle in degrees
- **searchrad** – 1/2 the length of the box used for the fit
- **guessfwhm** – approximate fwhm to fit to
- **guesspeak** – approximate flux
- **refinefit** – whether or not to refine the positioning of the planet
- **thetas** – ignore PA, supply own thetas (CCW angle from +x axis toward +y) single number or array of size N

Returns (peakflux, x, y, fwhm). A single tuple if one frame passed in. Otherwise an array of tuples

Return type measured

pyklip.fakes.**retrieve_planet_flux**(*frames*, *centers*, *astr_hdrs*, *sep*, *pa*, *searchrad=7*, *guessfwhm=3.0*, *guesspeak=1*, *refinefit=False*, *thetas=None*)

Retrives the planet flux from a series of frames given a separation and PA

Parameters

- **frames** – frames of data to retrieve planet. Can be a single 2-D image ([y,x]) for a series/cube ([N,y,x])

- **centers** – coordiantes of the image center. Can be [2]-element lst or an array that matches array of frames [N,2]
- **astr_hdrs** – astr_hdrs, can be a single one or an array of N of them
- **sep** – radial distance in pixels
- **PA** – parallactic angle in degrees
- **searchrad** – 1/2 the length of the box used for the fit
- **guessfwhm** – approximate fwhm to fit to
- **guesspeak** – approximate flux
- **refinefit** – whether or not to refine the positioning of the planet
- **thetas** – ignore PA, supply own thetas (CCW angle from +x axis toward +y) single number or array of size N

Returns

either a single peak flux or an array depending on whether a single frame or multiple frames where passed in

Return type peakflux

pyklip.fitpsf module

class pyklip.fitpsf.**FMAstrometry**(*guess_sep*, *guess_pa*, *fitboxsize*)

Bases: object

Base class to perform astrometry on direct imaging data_stamp using MCMC and GP regression

best_fit_and_residuals(*fig=None*)

Generate a plot of the best fit FM compared with the data_stamp and also the residuals :param fig: if not None, a matplotlib Figure object :type fig: matplotlib.Figure

Returns the Figure object. If input fig is None, function will make a new one

Return type fig (matplotlib.Figure)

fit_astrometry(*nwalkers=100*, *nburn=200*, *nsteps=800*, *save_chain=True*, *chain_output='bka-chain.pkl'*, *numthreads=None*)

Run a Bayesian fit of the astrometry using MCMC Saves to self.chian

Parameters

- **nwalkers** – number of walkers
- **nburn** – numbe of samples of burn-in for each walker
- **nsteps** – number of samples each walker takes
- **save_chain** – if True, save the output in a pickled file
- **chain_output** – filename to output the chain to
- **numthreads** – number of threads to use

Returns:

generate_data_stamp(*data*, *data_center*, *data_wcs=None*, *noise_map=None*, *dr=4*, *exclusion_radius=10*)

Generate a stamp of the data_stamp ~centered on planet and also corresponding noise map :param data: the

final collapsed data_stamp (2-D) :param data_center: location of star in the data_stamp :param data_wcs: sky angles WCS object. To rotate the image properly [NOT YET IMPLMETNED]

if None, data_stamp is already rotated North up East left

Parameters

- **noise_map** – if not None, noise map for each pixel in the data_stamp (2-D). if None, one will be generated assuming azimuthal noise using an annulus widthh of dr
- **dr** – width of annulus in pixels from which the noise map will be generated
- **exclusion_radius** – radius around the guess planet location which doens't get factored into noise estimate

Returns:

generate_fm_stamp(*fm_image*, *fm_center=None*, *fm_wcs=None*, *extract=True*, *padding=5*)

Generates a stamp of the forward model and stores it in self.fm_stamp :param fm_image: full imgae containing the fm_stamp :param fm_center: [x,y] center of image (assuing fm_stamp is located at sep/PA) corresponding to guess_sep and guess_pa :param fm_wcs: if not None, specifies the sky angles in the image. If None, assume image is North up East left :param extract: if True, need to extract the forward model from the image. Otherwise, assume the fm_stamp is already

centered in the frame (fm_image.shape // 2)

Parameters **padding** – number of pixels on each side in addition to the fitboxsize to extract to pad the fm_stamp (should be >= 1)

Returns:

make_corner_plot(*fig=None*)

Generate a corner plot of the posteriors from the MCMC :param fig: if not None, a matplotlib Figure object

Returns the Figure object. If input fig is None, function will make a new one

Return type fig

set_bounds(*dRA*, *dDec*, *df*, *covar_param_bounds*, *read_noise_bounds=None*)

Set bounds on Bayesian priors. All paramters can be a 2 element tuple/list/array that specifies the lower and upper bounds x_min < x < x_max. Or a single value whose interpretation is specified below If you are passing in both lower and upper bounds, both should be in linear scale! :param dRA: Distance from initial guess position in pixels. For a single value, this specifies the largest distance

form the initial guess (i.e. RA_guess - dRA < x < RA_guess + dRA)

Parameters

- **dDec** – Same as dRA except with Dec
- **df** – Flux range. If single value, specifies how many orders of 10 the flux factor can span in one direction (i.e. log_10(guess_flux) - df < log_10(guess_flux) < log_10(guess_flux) + df
- **covar_param_bounds** – Params for covariance matrix. Like df, single value specifies how many orders of magnitude parameter can span. Otherwise, should be a list of 2-elem touples
- **read_noise_bounds** – Param for read noise term. If single value, specifies how close to 0 it can go based on powers of 10 (i.e. log_10(-read_noise_bound) < read_noise < 1)

Returns:

set_kernel(*covar*, *covar_param_guesses*, *covar_param_labels*, *include_readnoise=False*, *read_noise_fraction=0.01*)
Set the Gaussian process kernel used in our astrometric fit

Parameters

- **covar** – Covariance kernel for GP regression. If string, can be "matern32" or "sqexp" Can also be a function: cov = cov_function(x_indices, y_indices, sigmas, cov_params)
- **covar_param_guesses** – a list of guesses on the hyperparmeteres (size of N_hyperparams)
- **covar_param_labels** – a list of strings labelling each covariance parameter
- **include_readnoise** – if True, part of the noise is a purely diagonal term (i.e. read/photon noise)
- **read_noise_fraction** – fraction of the total measured noise is read noise (between 0 and 1)

Returns:

pyklip.fitpsf.**lnlike**(*fitparams*, *fma*, *cov_func*)
Likelihood function :param fitparams: array of params (size N). First three are [dRA,dDec,f]. Additional parameters are GP hyperparams

dRA,dDec: RA,Dec offsets from star. Also coordianaes in self.data_{RA,Dec}_offset f: flux scale factor to normalizae the flux of the data_stamp to the model

Parameters

- **fma** (FMAstrometry) – a FMAstrometry object that has been fully set up to run
- **cov_func** (*function*) – function that given an input [x,y] coordinate array returns the covariance matrix e.g. cov = cov_function(x_indices, y_indices, sigmas, cov_params)

Returns log of likelihood function (minus a constant factor)

Return type likeli

pyklip.fitpsf.**lnprior**(*fitparams*, *bounds*)
Bayesian prior

Parameters

- **fitparams** – array of params (size N)
- **bounds** – array of (N,2) with corresponding lower and upper bound of params bounds[i,0] <= fitparams[i] < bounds[i,1]

Returns 0 if inside bound ranges, -inf if outside

Return type prior

pyklip.fitpsf.**lnprob**(*fitparams*, *fma*, *bounds*, *cov_func*)
Function to compute the relative posterior probabiltiy. Product of likelihood and prior :param fitparams: array of params (size N). First three are [dRA,dDec,f]. Additional parameters are GP hyperparams

dRA,dDec: RA,Dec offsets from star. Also coordianaes in self.data_{RA,Dec}_offset f: flux scale factor to normalizae the flux of the data_stamp to the model

Parameters

- **fma** – a FMAstrometry object that has been fully set up to run
- **bounds** – array of (N,2) with corresponding lower and upper bound of params bounds[i,0] <= fitparams[i] < bounds[i,1]
- **cov_func** – function that given an input [x,y] coordinate array returns the covariance matrix e.g. cov = cov_function(x_indices, y_indices, sigmas, cov_params)

Returns:

pyklip.fm module

pyklip.fm.**calculate_fm**(*delta_KL_nospec*, *original_KL*, *numbasis*, *sci*, *model_sci*, *inputflux=None*)

Calculate what the PSF looks up post-KLIP using knowledge of the input PSF, assumed spectrum of the science target, and the partially calculated KL modes (Delta Z_k^lambda in Laurent's paper). If inputflux is None, the spectral dependence has already been folded into delta_KL_nospec (treat it as delta_KL).

Note: if inputflux is None and delta_KL_nospec has three dimensions (ie delta_KL_nospec was calculated using pertrurb_nospec() or perturb_nospec_modelsBased()) then only klipped_oversub and klipped_selfsub are returned. Besides they will have an extra first spectral dimension.

Parameters

- **delta_KL_nospec** – perturbed KL modes but without the spectral info. delta_KL = spectrum x delta_Kl_nospec. Shape is (numKL, wv, pix). If inputflux is None, delta_KL_nospec = delta_KL
- **orignal_KL** – unpertrubed KL modes (array of size [numbasis, numpix])
- **numbasis** – array of KL mode cutoffs If numbasis is [None] the number of KL modes to be used is automatically picked based on the eigenvalues.
- **sci** – array of size p representing the science data
- **model_sci** – array of size p corresponding to the PSF of the science frame
- **input_spectrum** – array of size wv with the assumed spectrum of the model

If delta_KL_nospec does NOT include a spectral dimension or if inputflux is not None: :returns:

array of shape (b,p) showing the forward modelled PSF Skipped if inputflux = None, and delta_KL_nospec has 3 dimensions.

klipped_oversub: array of shape (b, p) showing the effect of oversubtraction as a function of KL modes klipped_selfsub: array of shape (b, p) showing the effect of selfsubtraction as a function of KL modes Note: psf_FM = model_sci - klipped_oversub - klipped_selfsub to get the FM psf as a function of K Lmodes

(shape of b,p)

Return type fm_psf

If inputflux = None and if delta_KL_nospec include a spectral dimension: :returns: Sum(<S|KL>KL) with klipped_oversub.shape = (size(numbasis),Npix)

klipped_selfsub: Sum(<N|DKL>KL) + Sum(<N|KL>DKL) with klipped_selfsub.shape = (size(numbasis),N_lambda or N_ref,N_pix)

Return type klipped_oversub

pyklip.fm.**calculate_fm_singleNumbasis**(*delta_KL_nospec*, *original_KL*, *numbasis*, *sci*, *model_sci*, *inputflux=None*)

Same function as calculate_fm() but faster when numbasis has only one element. It doesn't do the mutliplication with the triangular matrix.

Calculate what the PSF looks up post-KLIP using knowledge of the input PSF, assumed spectrum of the science target, and the partially calculated KL modes (Delta Z_k^lambda in Laurent's paper). If inputflux is None, the spectral dependence has already been folded into delta_KL_nospec (treat it as delta_KL).

Note: if inputflux is None and delta_KL_nospec has three dimensions (ie delta_KL_nospec was calculated using pertrurb_nospec() or perturb_nospec_modelsBased()) then only klipped_oversub and klipped_selfsub are returned. Besides they will have an extra first spectral dimension.

Parameters

- **delta_KL_nospec** – perturbed KL modes but without the spectral info. delta_KL = spectrum x delta_Kl_nospec. Shape is (numKL, wv, pix). If inputflux is None, delta_KL_nospec = delta_KL
- **orignal_KL** – unpertrubed KL modes (array of size [numbasis, numpix])
- **numbasis** – array of (ONE ELEMENT ONLY) KL mode cutoffs If numbasis is [None] the number of KL modes to be used is automatically picked based on the eigenvalues.
- **sci** – array of size p representing the science data
- **model_sci** – array of size p corresponding to the PSF of the science frame
- **input_spectrum** – array of size wv with the assumed spectrum of the model

If delta_KL_nospec does NOT include a spectral dimension or if inputflux is not None: :returns:

array of shape (b,p) showing the forward modelled PSF Skipped if inputflux = None, and delta_KL_nospec has 3 dimensions.

klipped_oversub: array of shape (b, p) showing the effect of oversubtraction as a function of KL modes klipped_selfsub: array of shape (b, p) showing the effect of selfsubtraction as a function of KL modes Note: psf_FM = model_sci - klipped_oversub - klipped_selfsub to get the FM psf as a function of K Lmodes

(shape of b,p)

Return type fm_psf

If inputflux = None and if delta_KL_nospec include a spectral dimension: :returns: Sum(<S|KL>KL) with klipped_oversub.shape = (size(numbasis),Npix)

klipped_selfsub: Sum(<N|DKL>KL) + Sum(<N|KL>DKL) with klipped_selfsub.shape = (size(numbasis),N_lambda or N_ref,N_pix)

Return type klipped_oversub

pyklip.fm.**calculate_validity**(*covar_perturb*, *models_ref*, *numbasis*, *evals_orig*, *covar_orig*, *evecs_orig*, *KL_orig*, *delta_KL*)

Calculate the validity of the perturbation based on the eigenvalues or the 2nd order term compared to the 0th order term of the covariance matrix expansion

Parameters

- **evals_perturb** – linear expansion of the perturbed covariance matrix (C_AS). Shape of N x N

- **models_ref** – N x p array of the N models corresponding to reference images. Each model should contain spectral information
- **numbasis** – array of KL mode cutoffs
- **evevs_orig** – size of [N, maxKL]

Returns perturbed KL modes. Shape is (numKL, wv, pix)

Return type delta_KL_nospec

pyklip.fm.**find_id_nearest**(*array*, *value*)

Find index of the closest value in input array to input value :param array: 1D array :param value: scalar value :return: Index of the nearest value in array

pyklip.fm.**klip_dataset**(*dataset*, *fm_class*, *mode='ADI+SDI'*, *outputdir='.'*, *fileprefix='pyklipfm'*, *annuli=5*, *subsections=4*, *OWA=None*, *N_pix_sector=None*, *movement=None*, *flux_overlap=0.1*, *PSF_FWHM=3.5*, *minrot=0*, *padding=3*, *numbasis=None*, *maxnumbasis=None*, *numthreads=None*, *calibrate_flux=False*, *aligned_center=None*, *spectrum=None*, *highpass=False*, *save_klipped=True*, *mute_progression=False*)

Run KLIP-FM on a dataset object

Parameters

- **dataset** – an instance of Instrument.Data (see instruments/ subfolder)
- **fm_class** – class that implements the the forward modelling functionality
- **mode** – one of ['ADI', 'SDI', 'ADI+SDI'] for ADI, SDI, or ADI+SDI
- **anuuli** – Annuli to use for KLIP. Can be a number, or a list of 2-element tuples (a, b) specifying the pixel bondaries (a <= r < b) for each annulus
- **subsections** – Sections to break each annuli into. Can be a number [integer], or a list of 2-element tuples (a, b) specifying the positon angle boundaries (a <= PA < b) for each section [radians]
- **OWA** – if defined, the outer working angle for pyklip. Otherwise, it will pick it as the cloest distance to a nan in the first frame
- **N_pix_sector** – Rough number of pixels in a sector. Overwriting subsections and making it sepration dependent. The number of subsections is defined such that the number of pixel is just higher than N_pix_sector. I.e. subsections = floor(pi*(r_max^2-r_min^2)/N_pix_sector) Warning: There is a bug if N_pix_sector is too big for the first annulus. The annulus is defined from

  0 to 2pi which create a bug later on. It is probably in the way pa_start and pa_end are defined in fm_from_eigen(). (I am taking about matched filter by the way)
- **movement** – minimum amount of movement (in pixels) of an astrophysical source to consider using that image for a refernece PSF
- **flux_overlap** – Maximum fraction of flux overlap between a slice and any reference frames included in the covariance matrix. Flux_overlap should be used instead of "movement" when a template spectrum is used. However if movement is not None then the old code is used and flux_overlap is ignored. The overlap is estimated for 1D gaussians with FWHM defined by PSF_FWHM. So note that the overlap is not exactly the overlap of two real 2D PSF for a given instrument but it will behave similarly.
- **PSF_FWHM** – FWHM of the PSF used to calculate the overlap (cf flux_overlap). Default is FWHM = 3.5 corresponding to sigma ~ 1.5.

- **minrot** – minimum PA rotation (in degrees) to be considered for use as a reference PSF (good for disks)
- **padding** – for each sector, how many extra pixels of padding should we have around the sides.
- **numbasis** – number of KL basis vectors to use (can be a scalar or list like). Length of b If numbasis is [None] the number of KL modes to be used is automatically picked based on the eigenvalues.
- **maxnumbasis** – Number of KL modes to be calculated from whcih numbasis modes will be taken.
- **numthreads** – number of threads to use. If none, defaults to using all the cores of the cpu
- **calibrate_flux** – if true, flux calibrates the regular KLIP subtracted data. DOES NOT CALIBRATE THE FM
- **aligned_center** – array of 2 elements [x,y] that all the KLIP subtracted images will be centered on for image registration
- **spectrum** – if not None, a array of length N with the flux of the template spectrum at each wavelength. Uses minmove to determine the separation from the center of the segment to determine contamination and the size of the PSF (TODO: make PSF size another quanitity) (e.g. minmove=3, checks how much containmination is within 3 pixels of the hypothetical source) if smaller than 10%, (hard coded quantity), then use it for reference PSF
- **highpass** – if True, run a Gaussian high pass filter (default size is sigma=imgsize/10) can also be a number specifying FWHM of box in pixel units
- **save_klipped** – if True, will save the regular klipped image. If false, it wil not and sub_imgs will return None
- **mute_progression** – Mute the printing of the progression percentage. Indeed sometimes the overwriting feature doesn't work and one ends up with thousands of printed lines. Therefore muting it can be a good idea.

pyklip.fm.**klip_math**(*sci*, *refs*, *numbasis*, *covar_psfs=None*, *model_sci=None*, *models_ref=None*, *spec_included=False*, *spec_from_model=False*)

linear algebra of KLIP with linear perturbation disks and point source

Parameters

- **sci** – array of length p containing the science data
- **refs** – N x p array of the N reference images that characterizes the extended source with p pixels
- **numbasis** – number of KLIP basis vectors to use (can be an int or an array of ints of length b) If numbasis is [None] the number of KL modes to be used is automatically picked based on the eigenvalues.
- **covar_psfs** – covariance matrix of reference images (for large N, useful). Normalized following numpy normalization in np.cov documentation
- **The following arguments must all be passed in, or none of them for klip_math to work** (*#*) –
- **models_ref** – N x p array of the N models corresponding to reference images. Each model should be normalized to unity (no flux information)
- **model_sci** – array of size p corresponding to the PSF of the science frame
- **Sel_wv** – wv x N array of the the corresponding wavelength for each reference PSF

- **input_spectrum** – array of size wv with the assumed spectrum of the model

Returns

array of shape (p,b) that is the PSF subtracted data for each of the b KLIP basis cutoffs. If numbasis was an int, then sub_img_row_selected is just an array of length p

KL_basis: array of KL basis (shape of [numbasis, p]) If models_ref is passed in (not None):

delta_KL_nospec: array of shape (b, wv, p) that is the almost perturbed KL modes just missing spectral info

Otherwise: evals: array of eigenvalues (size of max number of KL basis requested aka nummaxKL) evecs: array of corresponding eigenvectors (shape of [p, nummaxKL])

Return type sub_img_rows_selected

pyklip.fm.**klip_parallelized**(*imgs*, *centers*, *parangs*, *wvs*, *IWA*, *fm_class*, *OWA=None*, *mode='ADI+SDI'*, *annuli=5*, *subsections=4*, *movement=None*, *flux_overlap=0.1*, *PSF_FWHM=3.5*, *numbasis=None*, *maxnumbasis=None*, *aligned_center=None*, *numthreads=None*, *minrot=0*, *maxrot=360*, *spectrum=None*, *padding=3*, *save_klipped=True*, *flipx=True*, *N_pix_sector=None*, *mute_progression=False*)

multithreaded KLIP PSF Subtraction

Parameters

- **imgs** – array of 2D images for ADI. Shape of array (N,y,x)
- **centers** – N by 2 array of (x,y) coordinates of image centers
- **parangs** – N length array detailing parallactic angle of each image
- **wvs** – N length array of the wavelengths
- **IWA** – inner working angle (in pixels)
- **fm_class** – class that implements the the forward modelling functionality
- **OWA** – if defined, the outer working angle for pyklip. Otherwise, it will pick it as the cloest distance to a nan in the first frame
- **mode** – one of ['ADI', 'SDI', 'ADI+SDI'] for ADI, SDI, or ADI+SDI
- **anuuli** – Annuli to use for KLIP. Can be a number, or a list of 2-element tuples (a, b) specifying the pixel bondaries (a <= r < b) for each annulus
- **subsections** – Sections to break each annuli into. Can be a number [integer], or a list of 2-element tuples (a, b) specifying the positon angle boundaries (a <= PA < b) for each section [radians]
- **N_pix_sector** – Rough number of pixels in a sector. Overwriting subsections and making it sepration dependent. The number of subsections is defined such that the number of pixel is just higher than N_pix_sector. I.e. subsections = floor(pi*(r_max^2-r_min^2)/N_pix_sector) Warning: There is a bug if N_pix_sector is too big for the first annulus. The annulus is defined from

  0 to 2pi which create a bug later on. It is probably in the way pa_start and pa_end are defined in fm_from_eigen(). (I am taking about matched filter by the way)
- **movement** – minimum amount of movement (in pixels) of an astrophysical source to consider using that image for a refernece PSF

- **flux_overlap** – Maximum fraction of flux overlap between a slice and any reference frames included in the covariance matrix. Flux_overlap should be used instead of "movement" when a template spectrum is used. However if movement is not None then the old code is used and flux_overlap is ignored. The overlap is estimated for 1D gaussians with FWHM defined by PSF_FWHM. So note that the overlap is not exactly the overlap of two real 2D PSF for a given instrument but it will behave similarly.
- **PSF_FWHM** – FWHM of the PSF used to calculate the overlap (cf flux_overlap). Default is FWHM = 3.5 corresponding to sigma ~ 1.5.
- **numbasis** – number of KL basis vectors to use (can be a scalar or list like). Length of b If numbasis is [None] the number of KL modes to be used is automatically picked based on the eigenvalues.
- **maxnumbasis** – Number of KL modes to be calculated from whcih numbasis modes will be taken.
- **aligned_center** – array of 2 elements [x,y] that all the KLIP subtracted images will be centered on for image registration
- **numthreads** – number of threads to use. If none, defaults to using all the cores of the cpu
- **minrot** – minimum PA rotation (in degrees) to be considered for use as a reference PSF (good for disks)
- **maxrot** – maximum PA rotation (in degrees) to be considered for use as a reference PSF (temporal variability)
- **spectrum** – if not None, a array of length N with the flux of the template spectrum at each wavelength. Uses minmove to determine the separation from the center of the segment to determine contamination and the size of the PSF (TODO: make PSF size another quanitity) (e.g. minmove=3, checks how much containmination is within 3 pixels of the hypothetical source) if smaller than 10%, (hard coded quantity), then use it for reference PSF
- **padding** – for each sector, how many extra pixels of padding should we have around the sides.
- **save_klipped** – if True, will save the regular klipped image. If false, it wil not and sub_imgs will return None
- **flipx** – if True, flips x axis after rotation to get North up East left
- **mute_progression** – Mute the printing of the progression percentage. Indeed sometimes the overwriting feature doesn't work and one ends up with thousands of printed lines. Therefore muting it can be a good idea.

Returns

array of [array of 2D images (PSF subtracted)] using different number of KL basis vectors as

specified by numbasis. Shape of (b,N,y,x).

Note: this will be None if save_klipped is False

fmout_np: output of forward modelling. perturbmag: output indicating the magnitude of the linear perturbation to assess validity of KLIP FM

Return type sub_imgs

pyklip.fm.**pertrurb_nospec**(*evals*, *evecs*, *original_KL*, *refs*, *models_ref*)

Perturb the KL modes using a model of the PSF but with no assumption on the spectrum. Useful for planets.

By no assumption on the spectrum it means that the spectrum has been factored out of Delta_KL following equation (4) of Laurent Pueyo 2016 noted bold "Delta Z_k^lambda (x)". In order to get the actual perturbed KL modes one needs to multpily it by a spectrum.

This function fits each cube's spectrum independently. So the effective spectrum size is N_wavelengths * N_cubes.

Parameters

- **evals** – array of eigenvalues of the reference PSF covariance matrix (array of size numbasis)
- **evecs** – corresponding eigenvectors (array of size [p, numbasis])
- **orignal_KL** – unpertrubed KL modes (array of size [numbasis, p])
- **Sel_wv** – wv x N array of the the corresponding wavelength for each reference PSF
- **refs** – N x p array of the N reference images that characterizes the extended source with p pixels
- **models_ref** – N x p array of the N models corresponding to reference images. Each model should be normalized to unity (no flux information)
- **model_sci** – array of size p corresponding to the PSF of the science frame

Returns

perturbed KL modes but without the spectral info. delta_KL = spectrum x delta_Kl_nospec. Shape is (numKL, wv, pix)

Return type delta_KL_nospec

pyklip.fm.**perturb_nospec_modelsBased**(*evals*, *evecs*, *original_KL*, *refs*, *models_ref_list*)

Perturb the KL modes using a model of the PSF but with no assumption on the spectrum. Useful for planets.

By no assumption on the spectrum it means that the spectrum has been factored out of Delta_KL following equation (4) of Laurent Pueyo 2016 noted bold "Delta Z_k^lambda (x)". In order to get the actual perturbed KL modes one needs to multpily it by a spectrum.

Effectively does the same thing as pertrurb_nospec() but in a different way. It injects models with dirac spectrum (all but one vanishing wavelength) and because of the linearity of the problem allow one de get reconstruct the perturbed KL mode for any spectrum. The difference however in the pertrurb_nospec() case is that the spectrum here is the asummed to be the same for all

cubes while pertrurb_nospec() fit each cube independently.

Parameters

- **evals** –
- **evecs** –
- **original_KL** –
- **refs** –
- **models_ref** –

Returns

pyklip.fm.**perturb_specIncluded**(*evals*, *evecs*, *original_KL*, *refs*, *models_ref*, *return_perturb_covar=False*)

Perturb the KL modes using a model of the PSF but with the spectrum included in the model. Quicker than the others

Parameters

- **evals** – array of eigenvalues of the reference PSF covariance matrix (array of size numbasis)
- **evecs** – corresponding eigenvectors (array of size [p, numbasis])
- **orignal_KL** – unpertrubed KL modes (array of size [numbasis, p])
- **refs** – N x p array of the N reference images that characterizes the extended source with p pixels
- **models_ref** – N x p array of the N models corresponding to reference images. Each model should contain spectral informatoin
- **model_sci** – array of size p corresponding to the PSF of the science frame

Returns perturbed KL modes. Shape is (numKL, wv, pix)

Return type delta_KL_nospec

pyklip.klip module

pyklip.klip.**align_and_scale**(*img*, *new_center*, *old_center=None*, *scale_factor=1*, *dtype=<type 'float'>*)

Helper function that realigns and/or scales the image

Parameters

- **img** – 2D image to perform manipulation on
- **new_center** – 2 element tuple (xpos, ypos) of new image center
- **old_center** – 2 element tuple (xpos, ypos) of old image center
- **scale_factor** – how much the stretch/contract the image. Will we scaled w.r.t the new_center (done after relaignment). We will adopt the convention

  >1: stretch image (shorter to longer wavelengths) <1: contract the image (longer to shorter wvs) This means scale factor should be lambda_0/lambda where lambda_0 is the wavelength you want to scale to

Returns shifted and/or scaled 2D image

Return type resampled_img

pyklip.klip.**align_and_scale_JB**(*img*, *new_center*, *old_center=None*, *scale_factor=1*, *dtype=<type 'float'>*)

Helper function that realigns and/or scales the image

>>JB's version<<

I think the Nan management is better but there is still a bug to be fixed. (grid pattern in the nan background)

Parameters

- **img** – 2D image to perform manipulation on
- **new_center** – 2 element tuple (xpos, ypos) of new image center
- **old_center** – 2 element tuple (xpos, ypos) of old image center
- **scale_factor** – how much the stretch/contract the image. Will we scaled w.r.t the new_center (done after relaignment). We will adopt the convention

>1: stretch image (shorter to longer wavelengths) <1: contract the image (longer to shorter wvs) This means scale factor should be lambda_0/lambda where lambda_0 is the wavelength you want to scale to

Returns shifted and/or scaled 2D image

Return type resampled_img

pyklip.klip.**calc_scaling**(*sats*, *refwv=18*)

Helper function that calculates the wavelength scaling factor from the satellite spot locations. Uses the movement of spots diagonally across from each other, to calculate the scaling in a (hopefully? tbd.) centering-independent way. This method is definitely temporary and will be replaced by better scaling strategies as we come up with them. Scaling is calculated as the average of (1/2 * sqrt((x_1-x_2)**2+(y_1-y_2))), over the two pairs of spots.

Parameters

- **sats** – [4 x Nlambda x 2] array of x and y positions for the 4 satellite spots
- **refwv** – reference wavelength for scaling (optional, default = 20)

Returns Nlambda array of scaling factors

Return type scaling_factors

pyklip.klip.**define_annuli_bounds**(*annuli*, *IWA*, *OWA*, *annuli_spacing='constant'*)

Defines the annuli boundaries radially

Parameters

- **annuli** – number of annuli
- **IWA** – inner working angle (pixels)
- **OWA** – outer working anglue (pixels)
- **annuli_spacing** – how to distribute the annuli radially. Currently three options. Constant (equally spaced), log (logarithmical expansion with r), and linear (linearly expansion with r)

Returns array of 2-element tuples that specify the beginning and end radius of that annulus

Return type rad_bounds

pyklip.klip.**estimate_movement**(*radius*, *parang0=None*, *parangs=None*, *wavelength0=None*, *wavelengths=None*, *mode=None*)

Estimates the movement of a hypothetical astrophysical source in ADI and/or SDI at the given radius and given reference parallactic angle (parang0) and reference wavelegnth (wavelength0)

Parameters

- **radius** – the radius from the star of the hypothetical astrophysical source
- **parang0** – the parallactic angle of the reference image (in degrees)
- **parangs** – array of length N of the parallactic angle of all N images (in degrees)
- **wavelength0** – the wavelength of the reference image
- **wavelengths** – array of length N of the wavelengths of all N images
- **NOTE** – we expect parang0 and parangs to be either both defined or both None. Same with wavelength0 and wavelengths
- **mode** – one of ['ADI', 'SDI', 'ADI+SDI'] for ADI, SDI, or ADI+SDI

Returns

array of length N of the distance an astrophysical source would have moved from the reference image

Return type moves

pyklip.klip.**high_pass_filter**(*img*, *filtersize=10*)
A FFT implmentation of high pass filter.

Parameters

- **img** – a 2D image
- **filtersize** – size in Fourier space of the size of the space. In image space, size=img_size/filtersize

Returns the filtered image

Return type filtered

pyklip.klip.**klip_math**(*sci*, *ref_psfs*, *numbasis*, *covar_psfs=None*, *return_basis=False*, *return_basis_and_eig=False*)
Helper function for KLIP that does the linear algebra

Parameters

- **sci** – array of length p containing the science data
- **ref_psfs** – N x p array of the N reference PSFs that characterizes the PSF of the p pixels
- **numbasis** – number of KLIP basis vectors to use (can be an int or an array of ints of length b)
- **covar_psfs** – covariance matrix of reference psfs passed in so you don't have to calculate it here
- **return_basis** – If true, return KL basis vectors (used when onesegment==True)
- **return_basis_and_eig** – If true, return KL basis vectors as well as the eigenvalues and eigenvectors of the covariance matrix. Used for KLIP Forward Modelling of Laurent Pueyo.

Returns

array of shape (p,b) that is the PSF subtracted data for each of the b KLIP basis cutoffs. If numbasis was an int, then sub_img_row_selected is just an array of length p

KL_basis: array of shape (max(numbasis),p). Only if return_basis or return_basis_and_eig is True. evals: Eigenvalues of the covariance matrix. The covariance matrix is assumed NOT to be normalized by (p-1).

Only if return_basis_and_eig is True.

evecs: Eigenvectors of the covariance matrix. The covariance matrix is assumed NOT to be normalized by (p-1). Only if return_basis_and_eig is True.

Return type sub_img_rows_selected

pyklip.klip.**meas_contrast**(*dat*, *iwa*, *owa*, *resolution*, *center=None*, *low_pass_filter=True*)
Measures the contrast in the image. Image must already be in contrast units and should be corrected for algorithm thoughput.

Parameters

- **dat** – 2D image - already flux calibrated
- **iwa** – inner working angle

- **owa** – outer working angle
- **resolution** – size of resolution element in pixels (FWHM or lambda/D)
- **center** – location of star (x,y). If None, defaults the image size // 2.
- **low_pass_filter** – if True, run a low pass filter. Can also be a float which specifices the width of the Gaussian filter (sigma). If False, no Gaussian filter is run

Returns tuple of separations in pixels and corresponding 5 sigma FPF

Return type (seps, contrast)

pyklip.klip.**nan_gaussian_filter**(*img*, *sigma*)

Gaussian low-pass filter that handles nans

Parameters

- **img** – 2-D image
- **sigma** – float specifiying width of Gaussian

Returns 2-D image that has been smoothed with a Gaussian

Return type filtered

pyklip.klip.**rotate**(*img*, *angle*, *center*, *new_center=None*, *flipx=True*, *astr_hdr=None*)

Rotate an image by the given angle about the given center. Optional: can shift the image to a new image center after rotation. Also can reverse x axis for those left

handed astronomy coordinate systems

Parameters

- **img** – a 2D image
- **angle** – angle CCW to rotate by (degrees)
- **center** – 2 element list [x,y] that defines the center to rotate the image to respect to
- **new_center** – 2 element list [x,y] that defines the new image center after rotation
- **flipx** – default is True, which reverses x axis.
- **astr_hdr** – wcs astrometry header for the image

Returns new 2D image

Return type resampled_img

pyklip.parallelized module

pyklip.parallelized.**high_pass_filter_imgs**(*imgs*, *numthreads=None*, *filtersize=10*)

filters a sequences of images using a FFT

Inputs: imgs: array of shape (N,y,x) containing N images numthreads: number of threads to be used filtersize: size in Fourier space of the size of the space. In image space, size=img_size/filtersize

Output: filtered: array of shape (N,y,x) containing the filtered images

pyklip.parallelized.**klip_dataset**(*dataset*, *mode='ADI+SDI'*, *outputdir='.'*, *fileprefix=''*, *annuli=5*, *subsections=4*, *movement=3*, *numbasis=None*, *numthreads=None*, *minrot=0*, *calibrate_flux=False*, *aligned_center=None*, *annuli_spacing='constant'*, *maxnumbasis=None*, *spectrum=None*, *psf_library=None*, *highpass=False*, *lite=False*, *save_aligned=False*, *restored_aligned=None*, *dtype=<type 'numpy.float32'>*)

run klip on a dataset class outputted by an implementation of Instrument.Data

Parameters

- **dataset** – an instance of Instrument.Data (see instruments/ subfolder)
- **mode** – some combination of ADI, SDI, and RDI (e.g. "ADI+SDI", "RDI")
- **outputdir** – directory to save output files
- **fileprefix** – filename prefix for saved files
- **anuuli** – number of annuli to use for KLIP
- **subsections** – number of sections to break each annuli into
- **movement** – minimum amount of movement (in pixels) of an astrophysical source to consider using that image for a refernece PSF
- **numbasis** – number of KL basis vectors to use (can be a scalar or list like). Length of b
- **numthreads** – number of threads to use. If none, defaults to using all the cores of the cpu
- **minrot** – minimum PA rotation (in degrees) to be considered for use as a reference PSF (good for disks)
- **calibrate_flux** – if True calibrate flux of the dataset, otherwise leave it be
- **aligned_center** – array of 2 elements [x,y] that all the KLIP subtracted images will be centered on for image registration
- **annuli_spacing** – how to distribute the annuli radially. Currently three options. Constant (equally spaced), log (logarithmical expansion with r), and linear (linearly expansion with r)
- **maxnumbasis** – if not None, maximum number of KL basis/correlated PSFs to use for KLIP. Otherwise, use max(numbasis)
- **spectrum** – (only applicable for SDI) if not None, optimizes the choice of the reference PSFs based on the spectrum shape. Currently only supports "methane" between 1 and 10 microns.
- **psf_library** – if not None, a rdi.PSFLibrary object with a PSF Library for RDI
- **highpass** – if True, run a Gaussian high pass filter (default size is sigma=imgsize/10) can also be a number specifying FWHM of box in pixel units
- **lite** – if True, run a low memory version of the alogirhtm
- **Save the aligned and scaled images** (*save_aligned*) –
- **The aligned and scaled images from a previous run of klip_dataset** (*restore_aligned*) – (usually restored_aligned = dataset.aligned_and_scaled)

Returns Saved files in the output directory Returns: nothing, but saves to dataset.output: (b, N, wv, y, x) 5D cube of KL cutoff modes (b), number of images

(N), wavelengths (wv), and spatial dimensions. Images are derotated. For ADI only, the wv is omitted so only 4D cube

pyklip.parallelized.**klip_parallelized**(*imgs*, *centers*, *parangs*, *wvs*, *IWA*, *OWA=None*, *mode='ADI+SDI'*, *annuli=5*, *subsections=4*, *movement=3*, *numbasis=None*, *aligned_center=None*, *numthreads=None*, *minrot=0*, *maxrot=360*, *annuli_spacing='constant'*, *maxnumbasis=None*, *spectrum=None*, *psf_library=None*, *psf_library_good=None*, *psf_library_corr=None*, *save_aligned=False*, *restored_aligned=None*, *dtype=<type 'float'>*)

multithreaded KLIP PSF Subtraction

Parameters

- **imgs** – array of 2D images for ADI. Shape of array (N,y,x)
- **centers** – N by 2 array of (x,y) coordinates of image centers
- **parangs** – N length array detailing parallactic angle of each image
- **wvs** – N length array of the wavelengths
- **IWA** – inner working angle (in pixels)
- **mode** – one of ['ADI', 'SDI', 'ADI+SDI'] for ADI, SDI, or ADI+SDI
- **anuuli** – number of annuli to use for KLIP
- **subsections** – number of sections to break each annuli into
- **movement** – minimum amount of movement (in pixels) of an astrophysical source to consider using that image for a refernece PSF
- **numbasis** – number of KL basis vectors to use (can be a scalar or list like). Length of b
- **aligned_center** – array of 2 elements [x,y] that all the KLIP subtracted images will be centered on for image registration
- **numthreads** – number of threads to use. If none, defaults to using all the cores of the cpu
- **minrot** – minimum PA rotation (in degrees) to be considered for use as a reference PSF (good for disks)
- **maxrot** – maximum PA rotation (in degrees) to be considered for use as a reference PSF (temporal variability)
- **annuli_spacing** – how to distribute the annuli radially. Currently three options. Constant (equally spaced), log (logarithmical expansion with r), and linear (linearly expansion with r)
- **maxnumbasis** – if not None, maximum number of KL basis/correlated PSFs to use for KLIP. Otherwise, use max(numbasis)
- **spectrum** – if not None, a array of length N with the flux of the template spectrum at each wavelength. Uses minmove to determine the separation from the center of the segment to determine contamination and the size of the PSF (TODO: make PSF size another quanitity) (e.g. minmove=3, checks how much containmination is within 3 pixels of the hypothetical source) if smaller than 10%, (hard coded quantity), then use it for reference PSF
- **psf_library** – array of (N_lib, y, x) with N_lib PSF library PSFs

- **psf_library_good** – array of size N_lib indicating which N_good are good are selected in the passed in corr matrix
- **psf_library_corr** – matrix of size N_sci x N_good with correlation between the target franes and the good RDI PSFs
- **save_aligned** – Save the aligned and scaled images (as well as various wcs information), True/False
- **restore_aligned** – The aligned and scaled images from a previous run of klip_dataset (usually restored_aligned = dataset.aligned_and_scaled)
- **dtype** – data type of the arrays. Should be either float (meaning double) or np.float32.

Returns

array of [array of 2D images (PSF subtracted)] using different number of KL basis vectors as specified by numbasis. Shape of (b,N,y,x).

Return type sub_imgs

pyklip.parallelized.**klip_parallelized_lite**(*imgs*, *centers*, *parangs*, *wvs*, *IWA*, *OWA=None*, *mode='ADI+SDI'*, *annuli=5*, *subsections=4*, *movement=3*, *numbasis=None*, *aligned_center=None*, *numthreads=None*, *minrot=0*, *maxrot=360*, *annuli_spacing='constant'*, *maxnumbasis=None*, *spectrum=None*, *dtype=<type 'float'>*, ***kwargs*)

multithreaded KLIP PSF Subtraction, has a smaller memory foot print than the original

Parameters

- **imgs** – array of 2D images for ADI. Shape of array (N,y,x)
- **centers** – N by 2 array of (x,y) coordinates of image centers
- **parangs** – N length array detailing parallactic angle of each image
- **wvs** – N length array of the wavelengths
- **IWA** – inner working angle (in pixels)
- **OWA** – outer working angle (in pixels)
- **mode** – one of ['ADI', 'SDI', 'ADI+SDI'] for ADI, SDI, or ADI+SDI
- **anuuli** – number of annuli to use for KLIP
- **subsections** – number of sections to break each annuli into
- **movement** – minimum amount of movement (in pixels) of an astrophysical source to consider using that image for a refernece PSF
- **numbasis** – number of KL basis vectors to use (can be a scalar or list like). Length of b
- **annuli_spacing** – how to distribute the annuli radially. Currently three options. Constant (equally spaced), log (logarithmical expansion with r), and linear (linearly expansion with r)
- **maxnumbasis** – if not None, maximum number of KL basis/correlated PSFs to use for KLIP. Otherwise, use max(numbasis)
- **aligned_center** – array of 2 elements [x,y] that all the KLIP subtracted images will be centered on for image registration

- **numthreads** – number of threads to use. If none, defaults to using all the cores of the cpu
- **minrot** – minimum PA rotation (in degrees) to be considered for use as a reference PSF (good for disks)
- **maxrot** – maximum PA rotation (in degrees) to be considered for use as a reference PSF (temporal variability)
- **spectrum** – if not None, a array of length N with the flux of the template spectrum at each wavelength. Uses minmove to determine the separation from the center of the segment to determine contamination and the size of the PSF (TODO: make PSF size another quanitity) (e.g. minmove=3, checks how much containmination is within 3 pixels of the hypothetical source) if smaller than 10%, (hard coded quantity), then use it for reference PSF
- **kwargs** – in case you pass it stuff that we don't use in the lite version
- **dtype** – data type of the arrays. Should be either float (meaning double) or np.float32.

Returns

array of [array of 2D images (PSF subtracted)] using different number of KL basis vectors as specified by numbasis. Shape of (b,N,y,x).

Return type sub_imgs

pyklip.parallelized.**rotate_imgs**(*imgs*, *angles*, *centers*, *new_center=None*, *numthreads=None*, *flipx=True*, *hdrs=None*, *disable_wcs_rotation=False*)

derotate a sequences of images by their respective angles

Parameters

- **imgs** – array of shape (N,y,x) containing N images
- **angles** – array of length N with the angle to rotate each frame. Each angle should be CW in degrees. (TODO: fix this angle convention)
- **centers** – array of shape N,2 with the [x,y] center of each frame
- **new_centers** – a 2-element array with the new center to register each frame. Default is middle of image
- **numthreads** – number of threads to be used
- **flipx** – flip the x axis to get a left handed coordinate system (oh astronomers...)
- **hdrs** – array of N wcs astrometry headers

Returns array of shape (N,y,x) containing the derotated images

Return type derotated

pyklip.rdi module

class pyklip.rdi.**PSFLibrary**(*data*, *aligned_center*, *filenames*, *correlation_matrix=None*, *wvs=None*, *compute_correlation=False*)

Bases: object

This is an PSF Library to use for reference differential imaging

master_library
np.ndarray – aligned library of PSFs. 3-D cube of dim = [N, y, x]. Where N is ALL files

aligned_center
array-like – a (x,y) coordinate specifying common center the library is aligned to

master_filenames
np.ndarray – array of N filenames for each frame in the library. Should match with pyklip.instruments.Data.filenames for cross-matching

master_correlation
np.ndarray – N x N array of correlations between each 2 frames

master_wvs
np.ndarray – N wavelengths for each frame

nfiles
int – the number of files in the PSF library

dataset
pyklip.instruments.Instrument.Data

correlation
np.ndarray – N_data x M array of correlations between each 2 frames where M are the selected frames and N_data is the number of files in the dataset. Along the N_data dimension, files are ordered in the same way as the dataset object

isgoodpsf
np.ndarray – array of N indicating which M PSFs are good for this dataset

prepare_library(*dataset*, *badfiles=None*)
Prepare the PSF Library for an RDI reduction of a specific dataset by only taking the part of the library we need.

Parameters

- **dataset** (pyklip.instruments.Instrument.Data) –
- **badfiles** (*np.ndarray*) – a list of filenames corresponding to bad files we want to also exclude

Returns:

save_correlation(*filename*, *clobber=False*, *format='fits'*)
Saves self.correlation to a file specified by filename :param filename: filepath to store the correlation matrix :type filename: str :param format: type of file to store the correlation matrix as. Supports numpy?/fits?/pickle? (TBD) :type format: str

pyklip.spectra_management module

pyklip.spectra_management.**LSQ_place_model_PSF**(*PSF_template*, *x_cen*, *y_cen*, *planet_image*, *x_grid=None*, *y_grid=None*)

pyklip.spectra_management.**LSQ_scale_model_PSF**(*PSF_template*, *planet_image*, *a*)

pyklip.spectra_management.**extract_planet_centroid**(*cube*, *position*, *PSF_cube*)

pyklip.spectra_management.**extract_planet_spectrum**(*cube_para*, *position*, *PSF_cube_para*, *method=None*, *filter=None*, *mute=True*)

pyklip.spectra_management.**find_lower_nearest**(*array*, *value*)
Find the lower nearest element to value in array.

Parameters

- **array** – Array of value

- **value** – Value for which one wants the lower value.

Returns (low_value, id) with low_value the closest lower value and id its index.

pyklip.spectra_management.**find_nearest**(*array*, *value*)
Find the nearest element to value in array.

Parameters

- **array** – Array of value
- **value** – Value for which one wants the closest value.

Returns (closest_value, id) with closest_value the closest lower value and id its index.

pyklip.spectra_management.**find_upper_nearest**(*array*, *value*)
Find the upper nearest element to value in array.

Parameters

- **array** – Array of value
- **value** – Value for which one wants the upper value.

Returns (up_value, id) with up_value the closest upper value and id its index.

pyklip.spectra_management.**get_gpi_filter**(*filter_name*)
Extract the spectrum of a given gpi filter with the sampling of pipeline reduced cubes.

Inputs: filter_name: 'H', 'J', 'K1', 'K2', 'Y'

Output:

(wavelengths, spectrum) where wavelengths: is the gpi sampling of the considered band in mum. spectrum: is the transmission spectrum of the filter for the given band.

pyklip.spectra_management.**get_gpi_wavelength_sampling**(*filter_name*)
Return GPI wavelength sampling for a given band.

Parameters **filter_name** – 'H', 'J', 'K1', 'K2', 'Y'. Wavelength samples are linearly spaced between the first and the last wavelength of the band.

Returns is the gpi sampling of the considered band in micrometer.

Return type *wavelengths*

pyklip.spectra_management.**get_planet_spectrum**(*filename*, *wavelength*)
Get the normalized spectrum of a planet for a GPI spectral band or any wavelengths array. Spectra are extraced from .flx files from Mark Marley et al's models.

Parameters

- **filename** – Path of the .flx file containing the spectrum.
- **wavelength** – 'H', 'J', 'K1', 'K2', 'Y' or array of wavelenths in microns. When using GPI spectral band, wavelength samples are linearly spaced between the first and the last wavelength of the band.

Returns is the gpi sampling of the considered band in micrometer. spectrum: is the spectrum of the planet for the given band or wavelength array and normalized to unit mean.

Return type *wavelengths*

pyklip.spectra_management.**get_specType**(*object_name*, *SpT_file_csv=None*)
Return the spectral type for a target based on the table in SpT_file

Parameters

- **object_name** – Name of the target: ie "c_Eri"
- **SpT_file** – Filename (.csv) of the table containing the target names and their spectral type. Can be generated from bu quering Simbad. If None (default), the function directly tries to query Simbad.

Returns Spectral type

pyklip.spectra_management.**get_star_spectrum**(*wvs_or_filter_name*, *star_type=None*, *temperature=None*, *mute=None*)

Get the spectrum of a star with given spectral type interpolating in the pickles database. The spectrum is normalized to unit mean. Work only for type V star.

Inputs: wvs_or_filter_name: list of wavelengths or GPI filter 'H', 'J', 'K1', 'K2', 'Y'. star_type: 'A5','F4',... Is ignored if temperature is defined.

If star_type is longer than 2 characters it is truncated.

temperature: temperature of the star. Overwrite star_type if defined.

Output:

(wavelengths, spectrum) where wavelengths: Sampling in mum. spectrum: is the spectrum of the star for the given band.

pyklip.spectra_management.**place_model_PSF**(*PSF_template*, *x_cen*, *y_cen*, *output_shape*, *x_grid=None*, *y_grid=None*)

Module contents

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

W

Z