Whisky Documentation

Release 0.0.1

Interferometric Monkeys

November 07, 2014

CONTENTS

CHAPTER

ONE

LIBS PACKAGE

1.1 `libs` Package

1.2 `font` Module

CHAPTER

TWO

PYMASK PACKAGE

2.1 pymask Package

2.1.1 PYMASK: Python aperture masking analysis pipeline

— pymask is a python module for fitting models to aperture masking data reduced to oifits format by the IDL masking pipeline.

It consists of a class, cpo, which stores all the relevant information from the oifits file, and a set of functions, cp_tools, for manipulating these data and fitting models.

Fitting is based on the MCMC Hammer algorithm (aka ensemble affine invariant MCMC) or the MultiNest algorithm (aka multimodal nested sampling). Both of these must be installed correctly or else pymask won't work! See readme.txt for more details.

- Ben

—

2.2 cp_tools Module

pymask.cp_tools.**bin_fit_residuals**(*params*, *cpo*)
Function for binary_fit without errorbars

pymask.cp_tools.**binary_fit**(*cpo*, *p0*)
p0 is the initial guess for the parameters 3 parameter vector typical example would be : [100.0, 0.0, 5.0]. returns the full solution of the least square fit: - soluce[0] : best-fit parameters - soluce[1] : covariance matrix ————————————————————————-

pymask.cp_tools.**brute_force_chi2_grid**(*everything*)
Function for multiprocessing, fills in part of the big 3d chi2 grid in a way that minimises repeated calculations (i.e. each model only calculated once).

pymask.cp_tools.**brute_force_detec_limits**(*cpo*, *nsim=100*, *nsep=32*, *nth=20*, *ncon=32*, *smin='Default'*, *smax='Default'*, *cmin=10.0*, *cmax=500.0*, *addederror=0*, *threads=0*, *save=False*, *include_cov=False*, *icpo=False*, *projected=False*, *errscale=1.0*)
uses a Monte Carlo simulation to establish contrast-separation detection limits given an array of

standard deviations per closure phase.

Because different separation-contrast grid points are entirely separate, this task is embarrassingly parallel. If you want to speed up the calculation, use multiprocessing with a threads argument equal to the number of available cores.

Make nseps a multiple of threads! This uses the cores most efficiently.

Hyperthreading (2x processes per core) in my experience gets a ~20% improvement in speed.

Written by F. Martinache and B. Pope.

This version was modified by ACC to use a brute force chi2 grid that emulates the idl program binary_grid.pro

pymask.cp_tools.**chi2_grid**(*everything*)
Function for multiprocessing, does 2d chi2 grid for coarse_grid

pymask.cp_tools.**chi2_grid_proj**(*everything*)
Function for multiprocessing, does 2d chi2 grid for coarse_grid, with projected data

pymask.cp_tools.**coarse_grid**(*cpo*, *nsep=32*, *nth=20*, *ncon=32*, *smin='Default'*, *smax='Default'*, *cmin=10.0*, *cmax=500.0*, *threads=0*, *projected=False*)
Does a coarse grid search for the best fit parameters. This is helpful for finding a good initial point for hammer or multinest.

Because different separation-contrast grid points are entirely separate, this task is embarrassingly parallel. If you want to speed up the calculation, use multiprocessing with a threads argument equal to the number of available cores.

Make nseps a multiple of threads! This uses the cores most efficiently.

Hyperthreading (2x processes per core) in my experience gets a ~20% improvement in speed.

Written by A Cheetham, with some parts stolen from other pysco/pymask routines.

pymask.cp_tools.**cp_loglikelihood**(*params*, *u*, *v*, *wavel*, *t3data*, *t3err*, *model='constant'*)
Calculate loglikelihood for closure phase data. Used both in the MultiNest and MCMC Hammer implementations.

pymask.cp_tools.**cp_loglikelihood_multiple**(*params*, *u*, *v*, *wavel*, *t3data*, *t3err*, *model='constant'*, *ncomp=1*)
Calculate loglikelihood for closure phase data and multiple companions. Used both in the MultiNest and MCMC Hammer implementations.

pymask.cp_tools.**cp_loglikelihood_proj**(*params*, *u*, *v*, *wavel*, *proj_t3data*, *proj_t3err*, *proj*)
Calculate loglikelihood for projected closure phase data. Used both in the MultiNest and MCMC Hammer implementations.

pymask.cp_tools.**cp_loglikelihood_spectrum**(*spec_params*, *bin_params*, *u*, *v*, *wavel*, *t3data*, *t3err*, *model='free'*)
Calculate loglikelihood for closure phase data, with fixed sep, pa and con so you can measure a spectrum. Used in hammer_spectrum.

pymask.cp_tools.**cp_model**(*params*, *u*, *v*, *wavels*, *model='constant'*)
Function to model closure phases. Takes a parameter list, u,v triangles and a single wavelength. Allows fitting of a model to contrast vs wavelength. Models: constant

linear (params[2,3]=contrast ratios at end wavelengths), free (params[2:]=contrast ratios).

NOTE: This doesn't allow for nonzero size of each component!

pymask.cp_tools.**cp_model_old**(*params*, *u*, *v*, *wavels*)
Function to model closure phases. Takes a parameter list, u,v triangles and a single wavelength.

pymask.cp_tools.**detec_limits**(*cpo*, *nsim=2000*, *nsep=32*, *nth=20*, *ncon=32*, *smin='Default'*, *smax='Default'*, *cmin=1.0001*, *cmax=500.0*, *addederror=0*, *threads=0*, *save=False*, *projected=False*, *include_cov=False*, *icpo=False*, *errscale=1.0*, *no_plot=False*)
uses a Monte Carlo simulation to establish contrast-separation detection limits given an array of standard deviations per closure phase.

Because different separation-contrast grid points are entirely separate, this task is embarrassingly parallel. If you want to speed up the calculation, use multiprocessing with a threads argument equal to the number of available cores.

Make nseps a multiple of threads! This uses the cores most efficiently.

Hyperthreading (2x processes per core) in my experience gets a ~20% improvement in speed.

Written by F. Martinache and B. Pope. ACC added option for projected data and a few tweaks.

Note also that the calculation of the model closure phases could be done outside the big loop, which would be efficient on CPU but not RAM. However, ACC tried adding this and ran out of RAM (8GB) on GPI data (8880 clps), so removed it.

pymask.cp_tools.**detec_sim_loopfit**(*everything*)
Function for multiprocessing in detec_limits. Takes a single separation and full angle, contrast lists.

pymask.cp_tools.**detec_sim_loopfit_proj**(*everything*)
Function for multiprocessing in detec_limits. Takes a single separation and full angle, contrast lists. Made for projected data

pymask.cp_tools.**hammer**(*cpo, ivar=[52.0, 192.0, 1.53], ndim='Default', nwalcps=50, plot=False, projected=False, niters=1000, threads=1, model='constant', sep_prior=None, pa_prior=None, crat_prior=None, err_scale=1.0*)
Default implementation of emcee, the MCMC Hammer, for closure phase fitting. Requires a closure phase object cpo, and is best called with ivar chosen to be near the peak - it can fail to converge otherwise. Also allows fitting of a contrast vs wavlength model. See cp_model for details! Prior ranges introduce a flat (tophat) prior between the two values specified

pymask.cp_tools.**hammer_spectrum**(*cpo, params, ivar=[1.53], nwalcps=50, plot=False, niters=1000, threads=1, crat_prior=None, err_scale=1.0*)
ACC's modified version of hammer that allows fitting to a spectrum only, by fixing the position angle and separation of the companion. Made for HD142527 Requires a closure phase object cpo, and is best called with ivar chosen to be near the peak - it can fail to converge otherwise. Also allows fitting of a contrast vs wavlength model. See cp_model for details! Prior ranges introduce a flat (tophat) prior between the two values specified.

Ndim is the number of d.o.f in the spectral channels you want to fit to. It only works if ndim=nwav at the moment. Whoops. Otherwise need a polynomial model for cp_model

pymask.cp_tools.**lmfit**(*everything*)
Unpacks a cpo and calls the proper binary L-M fitting function

pymask.cp_tools.**mas2rad**(*x*)
Convenient little function to convert milliarcsec to radians

pymask.cp_tools.**multiple_companions_hammer**(*cpo, ivar=[[50.0, 0.0, 2.0], [50.0, 90.0, 2.0]], ndim='Default', nwalcps=50, plot=False, projected=False, niters=1000, threads=1, model='constant', sep_prior=None, pa_prior=None, crat_prior=None, err_scale=1.0*)

Implementation of emcee, the MCMC Hammer, for closure phase fitting to multiple companions. Requires a closure phase object cpo, and is best called with ivar chosen to be near the peak - it can fail to converge otherwise. See cp_model for details! Prior ranges introduce a flat (tophat) prior between the two values specified NOTE: This doesn't work with the wavelength model yet!

pymask.cp_tools.**nest**(*cpo, paramlimits=[20.0, 250.0, 0.0, 360.0, 1.0001, 10], ndim=3, resume=False, eff=0.3, multi=True*)

Default implementation of a MultiNest fitting routine for closure phase data. Requires a closure phase cpo object, parameter limits and sensible keyword arguments for the multinest parameters.

This function does very naughty things creating functions inside this function because PyMultiNest is very picky about how you pass it data.

Optional parameter eff tunes sampling efficiency, and multi toggles multimodal nested sampling on and off. Turning off multimodal sampling results in a speed boost of ~ 20-30%.

pymask.cp_tools.**phase_binary**(*u*, *v*, *wavel*, *p*)

p: 3-component vector (+2 optional), the binary "parameters": - p[0] = sep (mas) - p[1] = PA (deg) E of N. - p[2] = contrast ratio (primary/secondary)

optional: - p[3] = angular size of primary (mas) - p[4] = angular size of secondary (mas)

- u,v: baseline coordinates (meters)
- wavel: wavelength (meters)

pymask.cp_tools.**rad2mas**(*x*)

Convenient little function to convert radians to milliarcseconds

pymask.cp_tools.**test_significance**(*cpo*, *params*, *projected=False*)

Tests whether a certain set of binary parameters is a better fit than a single star model

2.3 cpo Module

class pymask.cpo.**cpo**(*oifits*)

Class used to manipulate multiple closure phase datasets

extract_from_oifits(*filename*)

Extract closure phase data from an oifits file.

2.4 oifits Module

A module for reading/writing OIFITS files

This module is NOT related to the OIFITS Python module provided at http://www.mrao.cam.ac.uk/research/OAS/oi_data/oifits.html It is a (better) alternative.

To open an existing OIFITS file, use the oifits.open(filename) function. This will return an oifits object with the following members (any of which can be empty dictionaries or numpy arrays):

array: a dictionary of interferometric arrays, as defined by the OI_ARRAY tables. The dictionary key is the name of the array (ARRNAME).

target: a numpy array of targets, as defined by the rows of the OI_TARGET table.

wavelength: a dictionary of wavelength tables (OI_WAVELENGTH). The dictionary key is the name of the instrument/settings (INSNAME).

vis, vis2 and t3: numpy arrays of objects containing all the measurement information. Each list member corresponds to a row in an OI_VIS/OI_VIS2/OI_T3 table.

This module makes an ad-hoc, backwards-compatible change to the OIFITS revision 1 standard originally described by Pauls et al., 2005, PASP, 117, 1255. The OI_VIS and OI_VIS2 tables in OIFITS files produced by this file contain two additional columns for the correlated flux, CFLUX and CFLUXERR , which are arrays with a length corresponding to the number of wavelength elements (just as VISAMP/VIS2DATA).

The main purpose of this module is to allow easy access to your OIFITS data within Python, where you can then analyze it in any way you want. As of version 0.3, the module can now be used to create OIFITS files from scratch without serious pain. Be warned, creating an array table from scratch is probably like nailing jelly to a tree. In a future verison this will become easier.

The module also provides a simple mechanism for combining multiple oifits objects, achieved by using the '+' operator on two oifits objects: result = a + b. The result can then be written to a file using result.save(filename).

Many of the parameters and their meanings are not specifically documented here. However, the nomenclature mirrors that of the OIFITS standard, so it is recommended to use this module with the PASP reference above in hand.

Beginning with version 0.3, the OI_VIS/OI_VIS2/OI_T3 classes now use masked arrays for convenience, where the mask is defined via the 'flag' member of these classes. Beware of the following subtlety: as before, the array data are accessed via (for example) OI_VIS.visamp; however, OI_VIS.visamp is just a method which constructs (on the fly) a masked array from OI_VIS._visamp, which is where the data are actually stored. This is done transparently, and the data can be accessed and modified transparently via the "visamp" hidden attribute. The same goes for correlated fluxes, differential/closure phases, triple products, etc. See the notes on the individual classes for a list of all the "hidden" attributes.

For further information, contact Paul Boley (boley@mpia-hd.mpg.de).

class pymask.oifits.**OI_ARRAY**(*frame*, *arrxyz*, *stations=()*)

Contains all the data for a single OI_ARRAY table. Note the hidden convenience attributes latitude, longitude, and altitude.

get_station_by_name(*name*)

info(*verbose=0*)

Print the array's center coordinates. If verbosity >= 1, print information about each station.

class pymask.oifits.**OI_STATION**(*tel_name=None, sta_name=None, diameter=None, staxyz=[None, None, None]*)

This class corresponds to a single row (i.e. single station/telescope) of an OI_ARRAY table.

class pymask.oifits.**OI_T3**(*timeobs*, *int_time*, *t3amp*, *t3amperr*, *t3phi*, *t3phierr*, *flag*, *u1coord*, *v1coord*, *u2coord*, *v2coord*, *wavelength*, *target*, *array=None*, *station=(None*, *None*, *None)*)

Class for storing triple product and closure phase data. To access the data, use the following hidden attributes:

t3amp, t3amperr, t3phi, t3phierr

info()

class pymask.oifits.**OI_TARGET**(*target*, *raep0*, *decep0*, *equinox=2000.0*, *ra_err=0.0*, *dec_err=0.0*, *sysvel=0.0*, *veltyp='TOPCENT'*, *veldef='OPTICAL'*, *pmra=0.0*, *pmdec=0.0*, *pmra_err=0.0*, *pmdec_err=0.0*, *parallax=0.0*, *para_err=0.0*, *spectyp='UNKNOWN'*)

info()

class pymask.oifits.**OI_VIS**(*timeobs*, *int_time*, *visamp*, *visamperr*, *visphi*, *visphierr*, *flag*, *ucoord*, *vcoord*, *wavelength*, *target*, *array=None*, *station=(None*, *None)*, *cflux=None*, *cfluxerr=None*)

Class for storing visibility amplitude and differential phase data. To access the data, use the following hidden attributes:

visamp, visamperr, visphi, visphierr, flag; and possibly cflux, cfluxerr.

info()

class pymask.oifits.**OI_VIS2**(*timeobs*, *int_time*, *vis2data*, *vis2err*, *flag*, *ucoord*, *vcoord*, *wavelength*, *target*, *array=None*, *station=(None*, *None)*)

Class for storing squared visibility amplitude data. To access the data, use the following hidden attributes:

vis2data, vis2err

info()

class pymask.oifits.**OI_WAVELENGTH**(*eff_wave*, *eff_band=None*)

info()

class pymask.oifits.**oifits**

info(*recursive=True*, *verbose=0*)

Print out a summary of the contents of the oifits object. Set recursive=True to obtain more specific information about each of the individual components, and verbose to an integer to increase the verbosity level.

isconsistent()

Returns True if the object is entirely self-contained, i.e. all cross-references to wavelength tables, arrays, stations etc. in the measurements refer to elements which are stored in the oifits object. Note that an oifits object can be 'consistent' in this sense without being 'valid' as checked by isvalid().

isvalid()

Returns True of the oifits object is both consistent (as determined by isconsistent()) and conforms to the OIFITS standard (according to Pauls et al., 2005, PASP, 117, 1255).

save(*filename*)

Write the contents of the oifits object to a file in OIFITS format.

pymask.oifits.**open**(*filename*, *quiet=False*)

Open an OIFITS file.

CHAPTER

THREE

WHISKY PACKAGE

CHAPTER

FOUR

WRITE_DOCSTRINGS MODULE

This is some introductionary text that is here for the whole module.

This module shows examples how to write good docstrings and make links between modules and functions

write_docstrings.**another_func**()
Just for fun

write_docstrings.**write_docstrings**(*param1*, *param2='haha'*)
This function does something.

I have a docstring, but **won't** be imported if you don't imported it.

Parameters

- **param1** (*str.*) – A first parameter.
- **param2** (*bool.*) – Another parameter, depending on `param1`

Returns int – the return code.

Raises AttributeError, KeyError

You can use that magic function if:

- you are awesome
- your are really awesome
- you hate IDL

You never call this class before calling `another_func()`.

Note: An example of intersphinx is this: you **cannot** use `misc` on this class.

```
>>> print 'this is some code sample followed by the result'
'this is some code sample followed by the result'
```

source: http://codeandchaos.wordpress.com/2012/07/30/sphinx-autodoc-tutorial-for-dummies/

and: https://pythonhosted.org/an_example_pypi_project/sphinx.html#full-code-example

PYTHON MODULE INDEX

p

w

INDEX

R

S

T

W