Whisky Documentation

Release 0.0.1

Interferometric Monkeys

November 07, 2014

CONTENTS

CHAPTER

ONE

LIBS PACKAGE

1.1 libs Package

A package that includes miscellaneous functions of various utility.

1.2 astro Module

libs.astro.**cal2JD**(*time=None*)

libs.astro.**fr_to_nu**(*fr*, *eccentricity*, *prec=1e-10*, *degrees=False*)
input: fr [% of period] (can be vector), eccentricity [0-1]; output: nu [deg] (same vector-size as fr)

libs.astro.**julian_to_calendar**(*julian*)
Calculates the calendar date from a julian date.

Parameters **julian** (*real*) – the data to pad, can be 1 to 3 dimensions

Returns anti-padded data as ndarray

```
>>> dvsef
```

libs.astro.**next_date**(*initdate*, *period*, *date_after=False*)
ads 'period' to 'initdate' as many times as required to have the new date right before the current gm time (or right after if 'date_after' is True)

libs.astro.**now**()

Returns the current UT time

libs.astro.**nu_to_fr**(*nu*, *eccentricity*, *degrees=False*)
input: nu N-dim, eccentricity [0-1); output: fr [% of period] (same shape as nu)

libs.astro.**orbital_pos**(*nu*, *e*, *i*, *w*, *a=149597871000.0*, *o=0.0*, *degrees=False*)
Calculates the orbital positions of a body with Keplerian orbit

Parameters

- **nu** (*N-dim array*) – true anomaly of the body (position on its orbit)
- **e** (*real*) – eccentricity of the orbit, must be [0,1)
- **i** (*real, radian*) – inclination of the orbit, must be [0,180). i>90 means retrograde body
- **w** (*real, radian*) – argument at periapsis of the orbit, should be [0,360)

- **a** (*real*) – semi-major axis of the orbit
- **o** (*real, radian*) – longitude of the ascending node of the orbit, should be [0,360)
- **degrees** (*bool*) – if True, `i`, `w`, `o` will be expected in degrees

Returns (radius, projected radius, north angle, phase angle), each having the same shape as `nu`. radius and projected radius have the same unit as `a`; north angle and phase angle are radians unless `degrees` is `True`.

Raises

- Exception, if e is outside [0,1)
- Exception, if i is outside [0,180]

```
>>> import misc.funcs as funcs
>>> print funcs.orbital_pos([10,20,30], 0.1, 89.6, 56, 1, 90, degrees=True)
(array([ 0.90124472,  0.90496144,  0.91109671]),
 array([ 0.36661431,  0.21901579,  0.06387085]),
 array([ 90.89833379,  91.60388027,  95.70132773]),
 array([ 24.00313585,  14.00559899,   4.01991791]))
```

1.3 `flux` Module

`libs.flux.`**`blackbody_spectral_irr`**(*teff*, *wl*)

Give an effective temerature Returns the emitted (at surface!) flux in W/m2/m assuming black body behaviour

UNITS: teff: K wl: m - can be vector output: W/m2/m

EXAMPLE: >>> blackbody_spectral_irr(5778, np.linspace(550-44, 550+44)*e-9)

class `libs.flux.`**`c_filter`**(*band*, *data*)

`libs.flux.`**`import_bands`**(*filename='data/UBVRIJHK.txt'*, *skiprows=2*)

Reads filename skiping the skiprows first rows Returns a dictionnary of filters

EXAMPLE: filters=import_bands('UBVRIJHK.txt', 2) filters['V'] <enter> shows all properties

`libs.flux.`**`magref2bbflux`**(*ref_mag*, *ref_band*, *band_or_wl*, *teff*, *photometry_file='data/UBVRIJHK.txt'*, *unit='W'*)

Give a reference magnitude, a reference band and wavelength or band for which you want to know the flux given an effective temperature of the star and the black body assumption. Returns a magnitude in the band_or_wl band, or a flux in W/m2/m for each of the wavelength in band_or_wl.

UNITS: ref_mag/ref_band: U, B, V, R, I, J, H, K band_or_wl: band or meter - can be a vector teff: K output: depending on unit 'Phm': Ph/s/m2/m (m being the walength grain) 'Wm': W/m2/m (m being the walength grain) 'Ph': Ph/s/m2 'W': W/m2 if band_or_wl is a photometric band, unit is disregarded and a mag. is returned

EXAMPLE: >>> magref2bbflux(4.83, 'V', 'R', 5778)

http://www.stsci.edu/hst/nicmos/tools/conversion_form.html

1.4 `font` Module

This module provides the user with a set of string constants to be concatenated, in order to add colors and/or formatting to terminal display.

```
>>> import misc.font as font
>>> print 'Hello, '+font.red+font.underlined+' I am in underlined red'+font.normal

>>> font. <tab>
Will give you the list of available colors and formatting options
```

Colors and font attributes taken from: http://misc.flogisoft.com/bash/tip_colors_and_formatting

1.5 `funcs` Module

libs.funcs.**apodizer**(*sizeframe*, *no_power_radius*, *full_power_radius=None*, *order=2.0*, *no_power_limit=0.01*, *full_power_limit=0.99*)

libs.funcs.**array_to_bins**(*arr*)

TBD

Parameters **data** (*array*) – the data to pad, can be 1 to 3 dimensions

Returns anti-padded data as ndarray

```
>>> import misc.funcs as funcs
>>> print func.array_to_bins([1,4,5,7])
[-0.5  2.5  4.5  6.   8. ]
```

libs.funcs.**array_to_binsize**(*arr*)

TBD

Give the bin size between elements of arr so that the whole range from min(arr) to max(arr) is filled. the i-th binsize = (arr[i+1]-arr[i-1])/2

```
>>> import misc.funcs as funcs
>>> print funcs.array_to_binsize([1,4,5,7])
[ 3.   2.   1.5  2. ]
```

libs.funcs.**aslist**(*data*, *numpy=False*, *integer=False*)

Transforms data into a 1 dimensional list

Parameters

- **data** – the data to transform into a list, can be any dimension and size
- **numpy** (*bool*) – returns the list as nparray
- **integer** (*bool*) – returns the list as integer

Returns a 1 dimensional list of `data`

```
>>> import misc.funcs as funcs
>>> print funcs.aslist(1)
[1]
>>> print funcs.aslist([(1.0,4.0),(2.0,3.0)], numpy=True, integer=True)
[1, 4, 2, 3]
```

libs.funcs.**bin_array**(*data*, *binning*, *mode=<function sum at 0x91a1534>*)

Groups together adjacent cells of an array

Parameters

- **data** – the N-dim array to process
- **binning** (*int or tuple of int*) – the shape of the N-dim cude that defines adjacent cells

- **mode** (*callable function*) – the function to apply on the adjacent cells

Returns the data, where adjacent cells in a cube of shape `binning` got collapsed together following `mode` method. The shape of the new data is _np.asarray(data.shape)/_np.asarray(binning)

Raises Exception, if binning `data`.shape is not a multiple of `binning`, or if their size disagree

```
>>> import misc.funcs as funcs
>>> vector = _np.array(_np.matrix(_np.arange(4)).T*_np.arange(6))
>>> data=_np.array([vector, vector+1])
>>> print data
[[[ 0  0  0  0  0  0]
  [ 0  1  2  3  4  5]
  [ 0  2  4  6  8 10]
  [ 0  3  6  9 12 15]]
 [[ 1  1  1  1  1  1]
  [ 1  2  3  4  5  6]
  [ 1  3  5  7  9 11]
  [ 1  4  7 10 13 16]]]
>>> print data.shape
(2, 4, 6)
>>> binned_data = funcs.bin_array(data, (2, 2, 1))
>>> print binned_data
[[[ 2  4  6  8 10 12]
  [ 2 12 22 32 42 52]]]
>>> print binned_data.shape
(1, 2, 6)
```

Note: `mode` must be a callable function that collapses 2D data along its axis 1 to a 1D data.

This callable function must take:

- some 2D data as first parameter
- a parameter `axis` that will be set to 1

You may use _np.sum, _np.mean, _np.median, _np.max, _np.min, ... functions.

`libs.funcs.`**`cat_cond`**(*cond*, *iftrue=''*, *iffalse=''*)

Returns iftrue as string is cond is True or not None, or iffalse as string if cond if false or None

`libs.funcs.`**`center_array`**(*data*, *newshape*)

Extracts the central part of an array

Parameters

- **data** – the N-dim array to process
- **newshape** (*int or tuple of int*) – the shape of the final subarray to extract

Returns central portion of the array, with shape `newshape`

```
>>> import misc.funcs as funcs
>>> print funcs.center_array([0,1,1,0], 2)
[1 1]
>>> print funcs.center_array([[0,1,1,0],[0,2,2,0],[0,3,3,0]], (1,2))
[[2 2]]
```

`libs.funcs.`**`check_str`**(*text*, *rem_char=None*, *auth_char=None*, *rem_words=None*, *auth_words=None*, *ignore_case=False*)

libs.funcs.**cmd_line**(*command*, *ret=True*, *clean=True*)
Executes a command line in shell environment

Parameters

- **command** (*string*) – the command to execute
- **ret** (*bool.*) – if `True`, the function will wait until `command` is executed. Else, the python script will go on while `command` is executed in the background
- **clean** (*bool.*) – if `True`, the 'empty lines' and the 'end of line characters' will be deleted from the output list

Returns `None` if `ret` is `False`, or if it is `True` a list of the standard output lines given back by `command`

```
>>> import misc.funcs as funcs
>>> print funcs.cmd_line('sleep 1')
[] (after 1 sec wait)
>>> print funcs.cmd_line('sleep 1', False)
will output nothing and jump to the next python instruction
>>> print funcs.cmd_line('sleep 1;ls *.dum', True, False)
['dummy.dum\n', 'dummy2.dum\n'] (after 1 sec wait)
```

libs.funcs.**colorbar**(*cmap='jet'*, *cm_min=0*, *cm_max=1*)

libs.funcs.**euler_rot**(*x*, *y*, *z*, *z1*, *x2*, *z3*, *degrees=False*)
Calculates the Euler rotation (z1, x2, z3) of the input coordinates

Parameters

- **x-y-z** (*N-dim array*) – the cooridnates to rotate
- **z1-x2-z3** (*real, radian*) – the Euler angles
- **degrees** (*bool*) – if True, the Euler angles `z1`, `x2`, `z3` will be expected in degrees

Returns (x', y', z'), the Euler angle rotated values of `(x, y, z)`

```
>>> import misc.funcs as funcs
>>> print funcs.euler_rot(1, 2, 3, 0.1, 0.2, -0.6)
(1.898096374053891, 0.66116903171326336, 3.1559603397867377)
```

libs.funcs.**gauss**(*x*, *a=1.0*, *x0=0.0*, *sigma=1.0*, *foot=0.0*)

libs.funcs.**gauss2D**(*x*, *y*, *a=1.0*, *x0=0.0*, *y0=0.0*, *sigma=1.0*, *foot=0.0*)

libs.funcs.**gauss2D_sup**(*x*, *y*, *a=1.0*, *x0=0.0*, *y0=0.0*, *sigma=1.0*, *foot=0.0*, *n=2*)

libs.funcs.**gauss_fit**(*x*, *y*, *a=None*, *x0=None*, *sigma=None*, *foot=None*)

libs.funcs.**gauss_sup**(*x*, *a=1.0*, *x0=0.0*, *sigma=1.0*, *foot=0.0*, *n=2*)

libs.funcs.**gen_generator**(*seed=None*)
Generates a random generator with a seed based on the micro second time and the process id. This makes it multi-threading safe.

Returns a random number generator object

```
>>> import misc.funcs as funcs
>>> rnd = funcs.gen_generator()
>>> print rnd.uniform()
0.472645
```

Note: This function calls `gen_seed()` if seed is not given.

---

libs.funcs.**gen_seed**()

Generates a seed based on the micro second time and the process id. This makes it multi-threading safe.

Returns an integer seed

```
>>> import misc.funcs as funcs
>>> print funcs.gen_seeed()
1825296981
```

class libs.funcs.**idlvar_read**(*filename*)

Reads an IDLVAR file and converts it to a python object

Parameters **filename** (*string*) – (path+)filename to the IDLVAR file

Returns an object whose attributes are the different variables found in the IDLVAR file

```
>>> import misc.funcs as funcs
>>> myvar = funcs.idlvar('/folder/folder/file.idlvar')
>>> myvar. <tab>
This will display all variables stored in the IDLVAR file
```

libs.funcs.**integrate_array**(*x*, *y*, *axis=0*)

Calculates the trapezoidal integral Y(x) of y(x) such as Y[i] = integral of y from x[0] to x[i], along one specific axis of y if it is multi-dimensional

Parameters

- **x-y** (*real*) – arrays that define a function f such as y=f(x). x and y along `axis` axis must have the same dimension
- **axis** (*int*) – if y is multi-dimensional, it tells the axis along which the integral must be carried

Returns Y(x), a numpy array having the same dimention as y

```
>>> import misc.funcs as funcs
>>> y = _np.tile(_np.array(_np.matrix(_np.ones(3)).T*_np.arange(4)), (2,1,1))
>>> print y
[[[ 0.  1.  2.  3.]
  [ 0.  1.  2.  3.]
  [ 0.  1.  2.  3.]]
 [[ 0.  1.  2.  3.]
  [ 0.  1.  2.  3.]
  [ 0.  1.  2.  3.]]]
>>> print y.shape
(2, 3, 4)
>>> print funcs.integrate_array(_np.arange(4), y, axis=2)
[[[ 0.   0.5  2.   4.5]
  [ 0.   0.5  2.   4.5]
  [ 0.   0.5  2.   4.5]]
 [[ 0.   0.5  2.   4.5]
  [ 0.   0.5  2.   4.5]
  [ 0.   0.5  2.   4.5]]]
```

libs.funcs.**nextp2**(*data*)

libs.funcs.**nextpn**(*data*, *power*)

libs.funcs.**pad_array**(*data*, *fill_value=0*, *axes=None*, *newshape=None*)
Pads data to the double of its size (unless specified) in each specified axis

Parameters

- **data** – the N-dim array to pad
- **fill_value** (*real*) – the value to fill the padding with
- **axes** (*int or tuple of int*) – the axes along which to perform anti-padding, default is all axes
- **newshape** (*"fft", int or tuple of int*) –
  - if set to "fft", the data will be padded to the next power of 2 along each specified axis
  - if array.shape-like, `axes` is ignored and data returned as the center of a an array with shape `newshape`

Returns padded data as ndarray

Raises Exception, if dimension of `newshape` does not agree with data.shape

```
>>> import misc.funcs as funcs
>>> print funcs.pad_array([1,1])
[ 0.   1.   1.   0. ]
>>> print funcs.pad_array([[1,1],[2,2]],3, axes=-1)
[[ 3.   1.   1.   3. ]
 [ 3.   2.   2.   3. ]]
```

Note: If `newshape` is set to "fft", all specified axes will be returned with the same size, which will be determined by the greatest next power of 2

If `newshape` is not specified, the size of each specified axis will be multiplied by 2

```
>>> print funcs.pad_array([[1,2]], 0, axes=(1), newshape="fft")
[[ 0.  1.  2.  0.]]
>>> print funcs.pad_array([[1,2]], 0, axes=(0,1), newshape="fft")
[[ 0.  0.  0.  0.]
 [ 0.  1.  2.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

libs.funcs.**padrev_array**(*data*, *axes=None*, *newshape=None*)
Anti-pads data to the half of its size in each specified axis, or to newshape is specified

Parameters

- **data** – the N-dim array to unpad
- **axes** (*int or tuple of int*) – the axes along which to perform anti-padding, default is all axes
- **newshape** (*int or tuple of int*) – if not None, `axes` is ignored and the inner region of data is returned, with shape `newshape`

Returns unpadded data as ndarray

```
>>> import misc.funcs as funcs
>>> print funcs.padrev_data([0,1,1,0])
[ 1.   1. ]
```

```
>>> print funcs.padrev_data([[0,1,1,0],[0,2,2,0]], axes=1)
[[1 1]
 [2 2]]
>>> print funcs.padrev_data([[0,1,1,0],[0,2,2,0]], newshape=(2,2))
[[1 1]
 [2 2]]
```

Note: This function calls `center_array()`.

If `newshape` is not specified, the size of each specified axis will be divided by 2

`libs.funcs.`**`percentile_view`**(*data*, *bins=None*, *log=False*, *plot=None*)

Calculates the percentile of a data and plots the result if required

Parameters

- **data** (*bool*) – the data, any dimension possible as it will be flattened
- **bins** (*int or array*) – the number of percentile values to be calculated, they are linearly spaced between 0 to 100 (included). If `bins` is an array, the percentile values are calculated for each of these values
- **log** (*bool*) – if true, shows the plot with log scale on y
- **plot** (*int or False*) – the number of the figure on which the curve is (over)plotted; if False, the function returns a vector; else or None, a new figure is created

Returns a vector of the calculated percentile if `plot` is False, otherwise None

```
>>> import misc.funcs as funcs
>>> print funcs.percentile_view(_np.random.random(1000),[25,50,75])
[0.23487384236763059, 0.48188344917304932, 0.76106477406241557]
```

`libs.funcs.`**`random_custom_cdf`**(*x*, *cdf*, *size=1*, *renorm=False*)

Generates random numbers (or the generating function) following a custom cdf

Parameters

- **x** (*array*) – the x data relative to the pdf
- **cdf** (*array*) – the cdf, must be the same size as x
- **size** (*bool, int, tupple*) – the size of the output: * if False, this fuction returns a function * if int, this function returns an array of size `size` * if array, this function returns an array whose shape is `size`
- **renorm** (*bool*) – forces first and last element of the cdf (obtained from the pdf) to be 0 and 1

Returns randomly generated numbers (or the generating function) following the pdf distribution

Raises

- Exception, if cdf is not monotonic increasing
- Exception, if first and last element of cdf are not 0 and 1 (and renorm is False)

```
>>> import misc.funcs as funcs
>>> x = _np.r_[_np.linspace(0,0.5,100), _np.linspace(0.5,1,101)[1:]]
>>> y = _np.r_[_np.linspace(0,0.5,100), 0.5-_np.linspace(0,0.5,100)]
>>> y = integrate_array(x, y)
```

```
>>> print func.random_custom_cdf(x, y, size=(3,2), renorm=True)
[[ 0.34601846  0.57638203]
 [ 0.46625474  0.52558282]
 [ 0.28020879  0.14224155]]
```

Note: This function calls `gen_generator()`.

---

libs.funcs.**random_custom_pdf**(*x*, *pdf*, *size=1*, *renorm=False*)

Generates random numbers (or the generating function) following a custom pdf

Parameters

- **x** (*array*) – the x data relative to the pdf
- **pdf** (*array*) – the pdf, must be the same size as x
- **size** (*bool, int or tuple of int*) – the size of the output: * if False, this fuction returns a function * if int, this function returns an array of size `size` * if array, this function returns an array whose shape is `size`
- **renorm** (*bool*) – forces first and last element of the cdf (obtained from the pdf) to be 0 and 1

Returns randomly generated numbers (or the generating function) following the pdf distribution

Raises Exception, if any value of the pdf is negative

```
>>> import misc.funcs as funcs
>>> x = _np.r_[_np.linspace(0,0.5,100), _np.linspace(0.5,1,101)[1:]]
>>> y = _np.r_[_np.linspace(0,0.5,100), 0.5-_np.linspace(0,0.5,100)]
>>> print funcs.random_custom_pdf(x, y, size=(3,2), renorm=True)
[[ 0.48273149  0.76731952]
 [ 0.8438356   0.32353295]
 [ 0.44305302  0.5361907 ]]
```

Note: This function calls `random_custom_cdf()`, which calls `gen_generator()`.

---

libs.funcs.**rebin_2D**(*x*, *y*, *data*, *bin_x*, *bin_y*, *offset_x=0.0*, *offset_y=0.0*, *halfway=False*)

libs.funcs.**rebin_2D_smart**(*x*, *y*, *data*, *bin_x*, *bin_y*)

libs.funcs.**replace_multi**(*text*, *reps*, *ignore_case=False*)

Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

libs.funcs.**resample_1d**(*data*, *pts*, *norm=True*)

libs.funcs.**resample_2d**(*data*, *shape*, *norm=True*)

libs.funcs.**setval**(*cond*, *iftrue*, *iffalse*, *toobject=None*, *attr=''*)

Returns (or adds) a value (or an attribute to an object) depending on a condition

Parameters

- **cond** (*bool or None*) – the condition. If `cond` is not already a boolean, then `cond` is checked to be None: `cond = cond is not None`
- **iftrue** (*any*) – the value that will be returned or added to the object if `cond` is `True`

- **iffalse** (*any*) – the value that will be returned or added to the object if `cond` is `False`
- **toobject** (*non-write protected object*) – if not None, setval will not return a value, but add it to this object
- **attr** (*string*) – the name of the attribute under which the value will be added to `toobject`. Ignored if `toobject` is None

Returns `iftrue` value if `cond` is `True` or not `None`, `iffalse` value if `cond` is `False` or `None`, nothing if the value is added to `toobject`

```
>>> import misc.funcs as funcs
>>> a = 12
>>> print funcs.setval(a>10, "Hello", "Goodbye")
Hello
```

`libs.funcs.`**shiftmicro_1d**(*data*, *delta*)

`libs.funcs.`**shiftmicro_2d**(*data*, *dx*, *dy*)

`libs.funcs.`**timestamp**(*gm=True*, *fmt='%Y%m%dT%H%M%S'*)

`libs.funcs.`**timestamp_name**(*filename*, *ext=None*, *gm=True*, *fmt='%Y%m%dT%H%M%S'*, *rem_char='sp'*, *auth_char=None*)

`libs.funcs.`**unique_timeid**(*unicity=None*, *freq=10*, *len_output=None*)

Generates a short unique ID based on the UT timestamp

Parameters

- **unicity** (*real, in years or None*) – the duration over which the unique ID will be unique. Leave `None` for truly unique ID
- **freq** (*real, in Hz*) – the unique ID update frequency. Systems usually provide timestamps down to the micro-second (freq=1e6)
- **len_output** (*int or None*) – the length of the string output, which will be padded with 0 if necessary. Leave `None` for no padding.

Returns a string representing the unique ID based on the UT timestamp

```
>>> import misc.funcs as funcs
>>> print funcs.unique_timeid(10, 1)
1zxtk
```

Note: The alphabet used for the string generation is `abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789`.

1.6 `getoption` Module

This module wraps getopt library to return a convenient object whose properties are lists containing the attributes and options provided in the command-line call.

class `libs.getoption.`**getoption**(*sysargv*, *opt=''*, *optval=''*, *param=[]*, *paramval=[]*, *man=''*)

Wraps getopt library to return a convenient object whose properties are lists containing the attributes and options provided in the command-line call.

Parameters

- **sysargv** (*list of string*) – the command-line call, as it is provided by sys.argv
- **opt** (*string*) – the concatenation of all authorized one-char options i.e. "-e"
- **optval** (*string*) – the concatenation of all authorized one-char options with input value i.e. "-e 3"
- **param** (*list of string*) – the list of all authorized multi-char options i.e. "–check"
- **paramval** (*list of string*) – the list of all authorized multi-char options with input value i.e. "–name=test"
- **man** (*string*) – the manual page of the script

Returns a python object whose attributes are lists of options and attributes, see note below for details

Raises Exception, if the getopt library cannot parse properly the command-line call

---

Note:

The returned object possesses as attributes:

- `command`: a string that shows the full command-line
- `nargs`: an integer showing the total number of arguments provided
- `optkey`: a list of all options provided in the command-line call, in the order of the command-line call
- `opt`: a dictionnary of the option provided and their associated value. Value-less options return True
- `attrs`: a list of all the attributes provided after the options
- `man`: a string that contains the manual of how to use this script

---

```
>>> import getoption, sys
>>> scriptcall = getoption.getoption(sys.argv, opt='wes', optval='r',
    man='This is the manual page. Thanks you for using getoption')
>>> if scriptcall['r'] is not None: print "r option was given"
>>> if 'h' in scriptcall.optkey:
>>>     print scriptcall.man
>>>     sys.exit()
```

Note: outputObject['option_name'] will return the value of the option `option_name` (or `True` is the option has no value), or `None` if the `option_name` was not provided in the command line call

The option `-h` is automatically added to the list of one-char authorized options

The syntax of the script is automatically generated from authorized input given to getoption and added at the beginning of the provided manual `man`

This module uses `misc.font` and `misc.funcs.aslist()`

```
>>> from getoption import getoption
>>> thecall = "conex -w --logfile=~/log.txt 1 TP 234".split(' ')
>>> # the .split(' ') is use to emulate the parsing of sys.argv
>>> scriptcall = getoption(thecall, opt='wes', paramval=['logfile'],
```

```
    man='Manual page of that script')
>>> scriptcall. <tab>
a.attrs    a.command  a.man      a.nargs    a.opt      a.optkey
>>> print scriptcall.nargs
5
>>> print scriptcall.optkey
['w', 'logfile']
>>> print scriptcall.attrs
['1', 'TP', '234']
>>> print scriptcall['w']
True
>>> print scriptcall['logfile']
~/log.txt
>>> print scriptcall['e']
None
```

1.7 img Module

libs.img.**add_poisson_noise**(*data*)

Adds poisson noise to a N-dim array. Replaces each value of the array with a random new value following poisson distribution whose mean is the initial value of the given array.

Parameters **data** (*real*) – N-dim array

Returns the same N-dim array where each value has been replaced with a random poisson distribution value with same mean (integer values)

```
>>> import misc.funcs as funcs
>>> funcs.add_poisson_noise([[4.9,5.3],[5.1,5.2]])
array([[ 3.,  4.],
       [ 6.,  7.]])
```

libs.img.**azimuthal**(*image*, *center=None*, *zone_radius=None*, *bin_size=None*)

Calculate the azimuthally averaged radial profile.

image - The 2D image center - The [x,y] pixel coordinates used as the center. The default is

None, which then uses the center of the image (including fracitonal pixels).

TODO : implémenter bin_size

libs.img.**blur_image**(*input*, *sigma*, *order=0*, *output=None*, *mode='reflect'*, *cval=0.0*)

Multi-dimensional Gaussian filter.

Parameters

- **input** – input array to filter
- **sigma** (*scalar or sequence of scalars*) – standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.
- **order** (*0, 1, 2, 3 or sequence from same set*) – The order of the filter along each axis is given as a sequence of integers, or as a single number. An order of 0 corresponds to convolution with a Gaussian kernel. An order of 1, 2, or 3 corresponds to convolution with the first, second or third derivatives of a Gaussian. Higher order derivatives are not implemented

- **output** (*array*) – The output parameter passes an array in which to store the filter output
- **mode** (*'reflect','constant','nearest','mirror', 'wrap'*) – The mode parameter determines how the array borders are handled, where `cval` is the value when `mode` is equal to 'constant'. Default is 'reflect'
- **cval** (*scalar*) – Value to fill past edges of input if `mode` is 'constant'

Note: The multi-dimensional filter is implemented as a sequence of one-dimensional convolution filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a limited precision, the results may be imprecise because intermediate results may be stored with insufficient precision.

This function is an alias of scipy.ndimage.gaussian_filter

`libs.img.`**`disk_add2array`**(*arr*, *x0*, *y0*, *radius*, *max_amplitude=1.0*)

Draws and adds a disk patterns to an input array.

Parameters

- **arr** – The 2D array to which the disk will be added. If dim(`arr`)>2, only the 2 last dimensions will be considered
- **x0-y0** (*real, in pixels*) – The pixel coordinates of the disk center
- **radius** (*real, in pixels*) – The radius of the disk
- **max_amplitude** (*real*) – The maximum amplitude of the disk

Returns 2D numpy array – the input array to which the disk was added. Values are real between 0 and `max_amplitude`.

Note: The addition of the disk pattern to the input array is genuinely an addition of the disk pattern values, not a replacement. Overlapping patterns will give values greater than `max_amplitude`

If the radius is lower than 0.5 px, the function will just add max_amplitude at `x0` and `y0` coordinates

---

```
>>> import misc.funcs as funcs
>>> import matplotlib.pyplot as plt
>>> myarray = funcs.disk_add2array(np.zeros([512,512]), 80, 250, 30)
>>> plt.matshow(myarray)
```

`libs.img.`**`find_boundaries`**(*center*, *subframesize*, *framesize=None*, *mode='fit'*, *integer=True*, *fuckingvariablewhichnobodyknowswhatisit='hui'*)

Returns the coordinates of a subframe in a frame, given its center and shape.

Parameters

- **center** (*real or array of real*) – the N-dim array of the subframe center's coordinates in the frame
- **subframesize** (*int or array of int*) – the N-dim array of the subframe shape
- **framesize** (*int or array of int*) – the N-dim array of the frame shape, None means "infinite" which is equivalent to `mode``=``simple`
- **mode** (*keyword*) –
  - *simple*: no checks on the frame borders are performed

- *fit*: in the case of a subframe centered too close to a frame border, the subframe center is shifted so the subframe fits inside the frame with the requested sub-frame size
- *cut*: in the case of a subframe centered too close to a frame border, the subframe is truncated so it doesn't go outside the frame. The sub-frame size is not conserved

- **integer** (*bool*) – forces the output to be integers

Returns a N-tuple of 2-element arrays ((xmin, xmax), (ymin, ymax), ...) of the subframe coordinates in the frame

Raises Exception, if the subframe is smaller than the frame in at least 1 dimension

```
>>> import misc.funcs as funcs
>>> funcs.find_boundaries([5.32, 16.76], subframesize=14, framesize=20,
    mode="fit")
(array([ 0, 14]), array([ 6, 20]))
```

libs.img.**get_blob_centers**(*im*, *n=2*, *blob_radius=10*, *sigma_blurring=None*, *return_sigma=False*, *stop_if_error=False*, *silent=False*)

Returns the coordinates of the n blobs of the image im after having performed a blur if specified

TODO: - blob mode sub ou flat rond ou carré, - auto find size of blob from 3 px then sigma (or 9 px etc) - silent option - opt return cleaned frame - auto blur sub pixel

libs.img.**subframe**(*data*, *center*, *subframesize*, *mode='fit'*, *fill_value=0.0*, *by_ref=False*)

Selects a N-dim subframe out of a N-dim frame, given its center and shape.

Parameters

- **data** – the N-dim array from which the subframe will be taken
- **center** (*real or array of real*) – the N-dim array of the subframe center's coordinates in the frame
- **subframesize** (*real or array of real*) – the N-dim array of the subframe shape
- **mode** (*keyword*) –
  - *fill*: in case the subframe goes over the edge of the frame, conserves the center location in the frame and the subfram size by filling the output with value `fill_value`
  - *fit*: in the case of a subframe centered too close to a frame border, the subframe is shifted so the subframe fits inside the frame with the requested sub-frame shape
  - *cut*: in the case of a subframe centered too close to a frame border, the subframe is truncated so it doesn't go outside the frame. The sub-frame shape is not conserved
- **fill_value** (*real or int*) – the fill value that is used if `mode` is set to fill
- **by_ref** (*bool*) – if True, the function will return a subframe of the frame that points to the corresponding slice of the frame. If False, the function returns a copy of it

Returns a N-dim array of shape `subframesize`

Raises

- Exception, if the subframe selection is done by reference with `mode` set to "fill"
- Exception, if the subframe selection is done by reference but data is not a nparray

```
>>> import misc.funcs as funcs
>>> funcs.subframe(np.array(np.matrix(np.arange(10)).T*np.arange(10)),
    center=[3.4, 1.7], subframesize=3, mode="fit")
[[ 2  4  6]
 [ 3  6  9]
 [ 4  8 12]]
```

Note: This function calls `find_boundaries()`.

1.8 `progbar` Module

class `libs.progbar.`**`progbar`**(*valmax=100*, *barsize=None*, *title='Be patient...'*)

Progress bar *valmax* is the last value of the iteration. Default is 100. *barsize* is the size of the bar in the opened window. If empty, the bar will automatically fit the window *title* is the title.

`reset`(*valmax=None*, *barsize=None*, *title=None*)

Progress bar *valmax* is the last value of the iteration. Default is 100. *barsize* is the size of the bar in the opened window. If empty, the bar will automatically fit the window *title* is the title.

`update`(*val='add'*)

Calling .update() adds 1 to the progress of the bar. Calling .update(i) puts the progress of the bar to i value.

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

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

l

p

w

INDEX