pYSOVAR Documentation

Release 0.7

H. M. Günther and K. Poppenhaeger for the YSOVAR team

August 17, 2015

Contents

This the the collection of python moduls that we (Katja and Moritz) use for our analysis of the YSOVAR data of those two clusters we are currently working on. While we attempt to write things general and applicable to more datasets, in some cases it's still very specific to our two clusters. So, don't be surprised if it does not work out-of-the-box for you, but feel free to try. You find the entire code on https://github.com/YSOVAR/YSOVAR .

First, we show how we process one of our clusters:

CHAPTER 1

How we process a cluster

First, we need to import the separate modules:

```
import astropy.io.ascii as ascii
import YSOVAR
import YSOVAR.atlas as atlas
import YSOVAR.plot as plot_atlas
import YSOVAR.lightcurves as lc
```

We did not write our papers yet, so we don't know everything we'll eventually do in the analysis. The list of processing steps below shows what we do, but we don't discuss every parameter value. For that, please check the documentation of the individual routines.

1.1 Reading the data

We start with the csv file that we download from the YSOVAR2 database. `match_dist = 0.` means that no source merging will be performed at this stage, i.e. we assume that this was done to the data in the database already. The routine reads a csv files (with data for both IRAC bands). Also, it calls *YSOVAR.atlas.dict_cleanup()*, which performs three important tasks:

1. It throws out sources with few entries
2. It constructs a merged lightcurve with IRAC 1 and IRAC 2 values that are close in time.
3. It adds Scott's error floor value to all errors in the data.

All this is done as part of the read in:

```
stars = atlas.dict_from_csv('myinputfile.csv', match_dist = 0.)
```

After reading, it's a good idea to do a few sanity checks on the data. Errors can happen, either in the database or in the sqlquery or elsewhere:

```
atlas.check_dataset(stars)
```

This routine performs a few simple tests, e.g. it looks for sources that are so close together, that they probably should be merged. Obviously, this is not going to find all possible problems, but every time we run into something we add a test here to make sure we find is next time.

Then, we make an *YSOVAR.atlas.YSOVAR_atlas* object. These object can auto-generate some column based on the lightcurves we put in, so check out the documentation for that:

```
mycloud = atlas.YSOVAR_atlas(lclist = stars)
```

The *YSOVAR.atlas.YSOVAR_atlas* is build on top of a astropy.table.Table (documentation here) object. See that documentation for the syntax on how to access the data or add a column.

The object `mycloud` combines two separate things: If represents a table of properties for each lightcurve, but it also keeps a copy of the original list of dictionaries. It is still possible to add lightcurves for new bands to that dictionary, which can be found in `mycloud.lclist`. For each band there have to be three lists of equal lengths in that dictionary:

- `'t'+band` for the observation times
- `'m'+band` for the magnitudes
- `'m'+band+'_error'` for the uncertainties

`band` can be any string **not** containing `_`. As an example, here we loop through all lightcurves and construct a combined IRAC1 and IRAC2 lightcurve. We then calculate the IRAC1-IRAC2 color and its error and add it to the dictionary, that the `'3645'` can be treated as if it were a separate band in the analysis later:

```
for d in mycloud.lclist:
    merged = atlas.merge_lc(d, ['36', '45'])
    if len(merged) > 5:
        d['t3645'] = np.array(merged['t'])
        d['m3645'] = np.array(merged['m36'] - merged ['m45'])
        d['m3645_error'] = np.array(np.sqrt(merged['m36_error']**2 + merged['m45_error']**2))
```

1.2 Merging with auxiliary data

There are all sorts of things that Astropy can read automatically (fits, votable, ascii), I just show two examples here.

The first is a machine-readable table that I downloaded from the ApJ website:

```
import astropy.io.ascii
# in this example from Barsony et al. (2012)
bar12 = astropy.io.ascii.read('apj426650t3_mrt.txt')
```

The second is a votable. I opened an image of my region in ds9, then did Analysis -> Catalogs -> SIMBAD and exported the SIMBAD catalog in the catalog window as a votable:

```
import asctropy.io.votable
simbad = astropy.io.votable.parse_single_table('SIMBAD.xml').to_table()
```

Unfortunately, in both cases the (RA, DEC) in the tables are not in degrees. pYSOVAR provides two methods of fixing that (or you could always compute the values by hand and add them to the table:

```
# by hand
mycloud.add_column(astropy.table.Column(name = 'RAdeg', data = RA_by_Hand))

# if RA, DEC are in CDS format, i.e. there are several columns
# `RAh`, `RAm`, `RAs`, `DE-`, `DEd`, `DEm` and `DEs`
atlas.coord_CDS2RADEC(bar12)

# if RA and DEC are string in the form
# hh:mm:ss.ss and dd:mm:ss.sss
atlas.coord_str2RADEC(mytable, ra='myRAcol', dec = 'myDEcol')
```

Then, we want to merge the auxiliary data with our YSOVAR data. To make sure that there is no name clash, I first add an `bar12_` to the name of each column:

```
for col in bar12.colnames:
    bar12.rename_column(col, 'bar12_'+col)
```

Then, we decide which column should be copied to `mycloud`. By default the matching is done by position and matched within 1 arcsec are accepted. `mycloud` objects with no counterpart in `bar12` get an empty value in the column, `bar12` objects with no counterpart in `mycloud` are ignored:

```
bar12_names = ['bar12_AV', 'bar12_Teff', 'bar12_Jmag']
mycloud.add_catalog_data(bar12, names = bar12_names, ra1='ra', dec1='dec', ra2='bar12_RAdeg', dec2='
```

1.3 Calculating properties

This is where it gets interesting. There are three basic ways to calculate properties form the lightcurves and add them to the `mycloud` table.

1.3.1 Auto-generate columns

This *YSOVAR.atlas.YSOVAR_atlas* auto-generates some content in the background, so we really encourage you to read the documentation (I promise it's only a few lines because I am too lazy to type much more), e.g.:

```
print mycloud['median_45']
```

will calculate the median for all lightcurves, add a column called `median_45` to the `mycloud` table and print the numbers to the screen.

The following properties can be calculated:

```
YSOVAR.registry.list_lcfuncs()
```

1.3.2 Call `calc()`

While the above method works fine for simple things like the median or the mean, more complicated functions need extra parameters. All functions listed with *YSOVAR.registry.list_lcfuncs()* can also be called using the *YSOVAR.atlas.YSOVAR_atlas.calc()* method. This allows far more flexibility. You can pass arguments to the functions you call, limit the calculation to use only a subset of the lightcurve or perform some filtering or smoothing of the lightcurve.

As an example, we search for periods. The parameters in the call are the maximum period in days, the oversampling factor, and a multiplier for the maximum frequency (see *YSOVAR.lombscargle.lombscargle()* for details):

```
mycloud.calc('lombscargle', '36', maxper = 100)
mycloud.calc('lombscargle', '45', maxper = 100)
# If we added the band '3645' as in the example above, we can do:
mycloud.calc('lombscargle', '3645', maxper = 100)

mycloud.is_there_a_good_period(20, 1,100)
```

Try fitting x vs. y (this is NOT like bisector, because we always use both x and y errors). This will show differences if the slope is almost vertical in one of the coordinate systems:

```
mycloud.calc('cmdslopeodr',['36','45'])

mycloud.calc('fitpoly', ['36'])
```

1.3.3 Add columns manually

`mycloud` is derived from an astropy.table.Table and you can calculate your own table columns and add them to `mycloud` as to any other astropy.table.Table.

The slope of the SED can be calculated (according to the definitions used by Luisa in the big data paper) like this:

```
mycloud.classify_SED_slope()
```

The command adds the `IRclass` column to `mycloud`, which takes the values `I`, `F`, `II` and `III`, for class I, flat-spectrum, class II and class III sources, respectively.

1.4 Make all the pretty plots

We write all the stuff in `outroot` and determine that a pdf will be good format. All matplotlib output formats are supported. Then, we do all the cool plots:

```
outroot = '/my/directory/'

# set output file type to pdf (for pdflatex)
plot_atlas.filetype = ['.pdf']

plot_atlas.get_stamps(mycloud, outroot)
plot_atlas.plot_polys(mycloud, outroot)
plot_atlas.make_lc_plots(mycloud, outroot)
plot_atlas.make_cmd_plots(mycloud, outroot)
plot_atlas.make_ls_plots(mycloud, outroot_stars, 300, 4, 1)
plot_atlas.make_phased_lc_cmd_plots(mycloud, outroot)
```

Write latex files for atlas. In this case we select for YSOs (numerical class < 4) only:

```
ind_ysos = np.where(mycloud['ysoclass'] < 4)[0]
plot_atlas.make_latexfile(mycloud, outroot, 'atlas_ysos', ind_ysos)
```

1.5 Write (parts of) a catalog to file

astropy.table.Table offers several methods to quickly look at the data, e.g. `mycloud.more()` to receive a more-style output in your terminal. See the documentation for details.

Here, is one example of output to LaTeX:

```
import astropy.io.ascii as ascii

ascii.write(mycloud, sys.stdout, Writer = ascii.Latex,
      names = mycloud.colnames,
      include_names=['ra', 'dec','YSOVAR2_id', 'median_45'],
      formats = {'ra':'%10.5f', 'dec':'%10.5f', 'median_45': '%4.2f'})
```

Here is a detailed documentation and API for all modules (possibly incomplete, since we document methods only after we know they work):

CHAPTER 2

Generate atlas

Generate an atlas of YSOVAR lightcurves

This module collects all procedures that are required to make the atlas. This starts with reading in the csv file from the YSOVAR2 database and includes the calculation of the some fits and quantities. More specific tasks for the analysis of the lightcurve can be found in *YSOVAR.lightcurves*, more stuff for plotting in *YSOVAR.plot*.

The basic structure for the YSOVAR analysis is the *YSOVAR.atlas.YSOVAR_atlas*. To initialize an atlas object pass in a numpy array with all the lightcurves:

```
from YSOVAR import atlas
data = atlas.dict_from_csv('/path/to/my/irac.csv', match_dist = 0.)
MyRegion = atlas.YSOVAR_atlas(lclist = data)
```

The *YSOVAR.atlas.YSOVAR_atlas* is build on top of a astropy.table.Table (documentation here) object. See that documentation for the syntax on how to acess the data or add a column.

This *YSOVAR.atlas.YSOVAR_atlas* auto-generates some content in the background, so I really encourage you to read the documentation (We promise it's only a few lines because we are too lazy to type much more).

YSOVAR.atlas.**IAU2radec**(*isoy*)

convert IAU name do decimal degrees.

Parameters **isoy** : string

IAU Name

Returns **ra, dec** : float

ra and dec in decimal degrees

class YSOVAR.atlas.**YSOVAR_atlas**(**args*, ***kwargs*)

The basic structure for the YSOVAR analysis is the *YSOVAR_atlas*. To initialize an atlas object pass in a numpy array with all the lightcurves:

```
from YSOVAR import atlas
data = atlas.dict_from_csv('/path/tp/my/irac.csv', match_dist = 0.)
MyRegion = atlas.YSOVAR_atlas(lclist = data)
```

The *YSOVAR_atlas* is build on top of a astropy.table.Table (documentation here) object. See that documentation for the syntax on how to acess the data or add a column.

Some columns are auto-generated, when they are first used. Some examples are

- median
- mean

- stddev
- min
- max
- mad (median absolute deviation)
- delta (90% quantile - 10% quantile)
- redchi2tomean
- wmean (uncertainty weighted average).

When you ask for `MyRegion['min_36']` it first checks if that column is already present. If not, if adds the new column called `min_36` and calculates the minimum of the lightcurve in band `36` for each object in the atlas, that has `m36` and `t36` entries (for magnitude and time in band `36` respectively. Data read with *`dict_from_csv()`* atomatically has the required format.

More functions may be added to this magic list later. Check:

```
import YSOVAR.registry
YSOVAR.registry.list_lcfuncs()
```

to see which functions are implemented. More function can be added.

Also, table columns can be added to an `YSOVAR_atlas` object manually, giving you all the freedom to do arbitrary calculations to arrive at those vales.

add_catalog_data(*catalog*, *radius=0.0002777777777777778*, *names=None*, ***kwargs*)
add information from a different Table

The tables are automatically cross matched and values are copied only for objects that have a counterpart in the current table.

Parameters **catalog** : astropy.table.Table

This is the table where the new information is provided.

radius : np.float

matching radius in degrees

names : list of strings

List column names that should be copied. If this is *None* (the default) copy all columns. Column names have to be unique. Thus, make sure that no column of the same name aleady exisits (this will raise an exception).

All other keywords are passed to :func:'YSOVAR.atlas.makecrossids' (see there :

for the syntax). :

add_mags(*data*, *cross_ids*, *band*, *channel*)
Add lightcurves to some list of dictionaries

Parameters **data** : astropy.table.Table or np.rec.array

data table with new mags

cross_ids : list of lists

for each elements in self, `cross_ids` says which row in *data* should be included for this object

band : list of strings

[name of mag, name or error, name of time]

channel : string

name of this channel in the lightcurve. Should be short and unique.

autocalc_newcol(*name*)
automatically calcualte some columns on the fly

calc(*name*, *bands*, *timefilter=None*, *data_preprocessor=None*, *colnames=[]*, *colunits=[]*, *coldescriptions=[]*, *coltypes=[]*, *overwrite=True*, *t_simul=None*, ***kwargs*)
calculate some quantity for all sources

This is a very general interface to the catalog that allows the user to initiate calculations with any function defined in the `registry`. A new column is added to the datatable that contains the result. (If the column exists before, it is overwritten).

Parameters **name** : string

name of the function in the function registry for YSOVAR (see `registry`)

bands : list of strings

Band identifiers In some cases, it can be useful to calculate a quantity for the error (e.g. the mean error). In this case, just give the band as e.g. `36_error`. (This only works for simple functions.)

timefilter : function or None

If not `None`, this function accepts a np.ndarray of observation times and it should return an index array selecteing those time to be included in the calculation. The default function selects all times. An example how to use this keyword include certain times only is shown below:

```
cat.calc('mean', '36', timefilter = lambda x : x < 55340)
```

data_preprocessor : function or None

If not None, for each source, a row from the present table is extracted as `source = self[id]`. This yields a *YSOVAR_atlas* object with one row. This row is passed to `data_preprocessor`, which can modify the data (e.g. smooth a lightcurve), but should keep the structure of the object intact. Here is an example for a possible `data_preprocessor` function:

```
def smooth(source):
    lc = source.lclist[0]
    w = np.ones(5)
    if 'm36' in lc and len(lc['m36'] > 10):
        lc['m36'] = np.convolve(w/w.sum(),lc['m36'],mode='valid')
        # need to keep m36 and t36 same length
        lc['t36'] = lc['t36'][0:len(lc['m36'])]
    return source
```

colnames : list of strings

Basenames of columns to be hold the output of the calculation. If not present already, the bands are added automatically with `dtype = np.float`:

```
cat.calc('mean', '36', colnames = ['stuff'])
```

would add the column `stuff_36`. If this is left empty, its default is set based on the function. If `colnames` has less elements than the function returns, only the first few are kept.

colunits : list of strings

Units for the output columns. If this is left empty, its default is set based on the function. If `colunits` has fewer elements than there are new columns, the unit of the remaining columns will be `None`.

coldescriptions : list of strings

Descriptions for the output columns. If this is left empty, its default is set based on the function. If `coldescriptions` has fewer elements than there are new columns, the description of the remaining columns will be `''`.

coltypes: list :

list of dtypes for autogenerated columns. If the list is empty, its set to the default of the function called.

overwrite : bool

If True, values in existing columns are silently overwritten.

t_simul : float

max distance in days to accept datapoints in band 1 and 2 as simultaneous In L1688 and IRAS 20050+2720 the distance between band 1 and 2 coverage is within a few minutes, so a small number is sufficent to catch everything and to avoid false matches. If `None` is given, this defaults to `self.t_simul`.

All remaining keywords are passed to the function identified by ‘‘name‘‘. :

`calc_allstats`(*band*)

calcualte all simple statistical descriptors for a single band

This function calculates all simple statistical quantities that can be autogenerated for a certain band. The required columns are added to the data table.

This does no include periodicity, which requires certain user selected parameters.

Parameters band : string

name of band for which the calcualtion should be performed

`classify_SED_slope`(*bands=['mean_36', 'mean_45', 'Kmag', '3.6mag', '4.5mag', '5.8mag', '8.0mag'], colname='IRclass'*)

Classify the SED slope of an object

This function calculates the SED slope for each object according to the prescription outlined by Luisa in the big data paper.

It uses all available datapoints in the IR from the bands given If no measurement is present (e.g. missing or upper limit only) this band is ignored. The procedure performs a least-squares fit with equal weight for each band and then classifies the resulting slope into class I, flat-spectrum, II and III sources.

Parameters bands : list of strings

List of names for all bands to be used. Bands must be defined in `YSOVAR.atlas.sed_bands`.

colname : string

The classification will be placed in this column. If it exists it is overwritten.

`is_there_a_good_period`(*power, minper, maxper, bands=['36', '45'], FAP=False*)

check if a strong periodogram peak is found

This method checks if a period exisits with the required power and period in any of of the bands given in `bands`. If the peaks in several bands fullfill the criteria, then the band with the peak of highest power is selected. Output is placed in the columns `good_peak`, `good_FAP` and `good_period`.

New columns are added to the datatable that contains the result. (If a column existed before, it is overwritten).

Parameters **power** : float

minimum power or maximal FAP for "good" period

minper : float

lowest period which is considered

maxper : float

maximum period which is considered

bands : list of strings

Band identifiers, e.g. `['36', '45']`, can also be a list with one entry, e.g. `['36']`

FAP : boolean

If `True`, then `power` is interpreted as maximal FAP for a good period; if `False` then `power` means the minimum power a peak in the periodogram must have.

sort (*keys*)

Sort the table according to one or more keys. This operates on the existing table and does not return a new table.

Parameters **keys** : str or list of str

The key(s) to order the table by

YSOVAR.atlas.**check_dataset** (*data*, *min_number_of_times=5*, *match_dist=0.0002777777777777778*)

check dataset for anomalies, cross-match problems etc.

Of course, not every problem can be detected here, but every time I find something I add a check so that next time this routine will warn me of the same problem.

Parameters **data** : list of dicts

as read in with e.e. *dict_from_csv*

YSOVAR.atlas.**coord_CDS2RADEC** (*dat*)

transform RA and DEC from CDS table to degrees

CDS tables have a certain format of string columns to store coordinates (`RAh`, `RAm`, `RAs`, `DE-`, `DEd`, `DEm`, `DEs`). This procedure parses that and calculates new values for RA and DEC in degrees. These are added to the Table as `RAdeg` and `DEdeg`.

Parameters **dat** : *YSOVAR.atlas.YSOVAR_atlas* or `astropy.table.Table`

with columns in the CDS format (e.g. from reading a CDS table with `astropy.io.ascii`)

YSOVAR.atlas.**coord_add_RADEfromhmsdms** (*dat*, *rah*, *ram*, *ras*, *design*, *ded*, *dem*, *des*)

transform RA and DEC in table from hms, dms to degrees

Parameters **dat** : *YSOVAR.atlas.YSOVAR_atlas* or `astropy.table.Table`

with columns in the CDS format (e.g. from reading a CDS table with `astropy.io.ascii`)

rah, ram, ras, ded, dem, des: np.ndarray :

RA and DEC hms, dms values

design: +1 or -1 :

Sign of the DE coordinate (integer or float, not string)

YSOVAR.atlas.**coord_hmsdms2RADEC**(*dat, ra=['RAh', 'RAm', 'RAs'], dec=['DEd', 'DEm', 'DEs']*)
transform RA and DEC from table to degrees

Tables where RA and DEC are encoded as three numeric columns each like `hh:mm:ss` and `dd:mm:ss` can be converted into decimal deg. This procedure parses that and calculates new values for RA and DEC in degrees. These are added to the Table as `RAdeg` and `DEdeg`.

Warning: This format is ambiguous for sources with dec=+/-00:xx:xx, because python does not differentiate between `+0` and `-0`.

Parameters **dat** : *YSOVAR.atlas.YSOVAR_atlas* or astropy.table.Table

with columns in the format given above

ra : list of three strings

names of RA column names for hour, min, sec

dec : list of three strings

names of DEC column names for deg, min, sec

YSOVAR.atlas.**coord_strhmsdms2RADEC**(*dat, ra='RA', dec='DEC', delimiter=':'*)
transform RA and DEC from table to degrees

Tables where RA and DEC are encoded as string columns each like *hh:mm:ss dd:mm:ss* can be converted into decimal deg. This procedure parses that and calculates new values for RA and DEC in degrees. These are added to the Table as *RAdeg* and *DEdeg*.

Parameters **dat** : *YSOVAR.atlas.YSOVAR_atlas* or astropy.table.Table

with columns in the format given above

ra : string

name of RA column names for hour, min, sec

dec : string

name of DEC column names for deg, min, sec

delimiter : string

delimiter between elements, e.g. `:` in `01:23:34.3`.

YSOVAR.atlas.**dict_cleanup**(*data, channels, min_number_of_times=0, floor_error={}*)
Clean up dictionaries after add_ysovar_mags

Each object in the *data* list can be constructed from multiple sources per band in multiple bands. This function averages the coordinates over all contributing sources, converts and sorts lists of times and magnitudes and makes multiband lightcurves.

Parameters **data** : list of dictionaries

as obtained from `YSOVAR.atlas.add_ysovar_mags()`

channels : dictionary

This dictionary traslantes the names of channels in the csv file to the names in the output structure, e.g. that for *'IRAC1'* will be *'m36'* (magnitudes) and *'t36'* (times).

min_number_of_times : integer

Remove all lightcurves with less than *min_number_of_times* datapoints from the list

floor_error : dict

Floor errors will be added in quadrature to all error values. The keys in the dictionary should be the same as in the channels dictionary.

Returns **data** : list of dictionaries

individual dictionaries are cleaned up as decribed above

YSOVAR.atlas.**dict_from_csv**(*csvfile*, *match_dist=0.0002777777777777778*, *min_number_of_times=5*, *channels={'IRAC2': '45', 'IRAC1': '36'}*, *data=[]*, *floor_error={'IRAC2': 0.007, 'IRAC1': 0.01}*, *mag='mag1'*, *emag='emag1'*, *time='hmjd'*, *bg=None*, *source_name='sname'*, *verbose=True*, *readra='ra'*, *readdec='de'*, *sourceid='ysovarid'*, *channelcolumn='fname'*)

Build YSOVAR lightcurves from database csv file

Parameters **cvsfile** : sting or file object

input csv file

match_dist : float

maximum distance to match two positions as one sorce

min_number_of_times : integer

Remove all sources with less than min_number_of_times datapoints from the list

channels : dictionary

This dictionary translates the names of channels in the csv file to the names in the output structure, e.g. that for `IRAC1` will be `m36` (magnitudes) and `t36` (times).

data : list of dicts

New entries will be added to data. It can be empty (the default).

mag : string

name of magnitude column

emag : string

name of column holding the error on the mag

time : string

name of column holding the time of observation

bg : string or None

name of column holding the bg for each observations (None indicates that the bg column is not present).

floor_error : dict

Floor errors will be added in quadrature to all error values. The keys in the dictionary should be the same as in the channels dictionary.

verbose : bool

If True, print progress status.

Returns **data** : empty list or list of dictionaries

structure to hold all the information

TBD: Still need to deal with double entries in lightcurve (and check manually...) :

YSOVAR.atlas.**get_sed**(*data, sed_bands={'Bmag': ['e_Bmag', 0.43, 4000.87], '5.8mag': ['e_5.8mag', 5.8, 115.0], 'mean_36': ['e_3.6mag', 3.6, 280.9], 'simbad_B': [None, 0.43, 4000.87], '4.5mag': ['e_4.5mag', 4.5, 179.7], 'Vmag': ['e_Vmag', 0.623, 3597.28], 'Imag': ['e_Imag', 0.798, 2587], 'Kmag': ['e_Kmag', 2.159, 666.7], 'simbad_V': [None, 0.623, 3597.28], 'Rmag': ['e_Rmag', 0.759, 3182], '3.6mag': ['e_3.6mag', 3.6, 280.9], 'mean_45': ['e_4.5mag', 4.5, 179.7], 'imag': ['e_imag', 0.763, 2515.7], 'Jmag': ['e_Jmag', 1.235, 1594], 'rmag': ['e_rmag', 0.622, 3173.3], 'Hmag': ['e_Hmag', 1.662, 1024], '8.0mag': ['e_8.0mag', 8.0, 64.13], '24mag': ['e_24mag', 24.0, 7.14], 'Umag': ['e_Umag', 0.355, 1500], 'nomad_Rmag': [None, 0.759, 3182], 'nomad_Bmag': [None, 0.43, 4000.87], 'Hamag': ['e_Hamag', 0.656, 2974.4], 'nomad_Vmag': [None, 0.623, 3597.28]}, valid=False*)

make SED by collecting info from the input data

Parameters **data** : *YSOVAR.atlas.YSOVAR_atlas* or astropy.table.Table

input data that has arrays of magnitudes for different bands

sed_bands : dict

keys must be the name of the field that contains the magnitudes in each band, entries are lists of [name of error field, wavelength in micron, zero_magnitude_flux_freq in Jy]

valid : bool

If true, return only bands with finite flux, otherwise return all bands that exist in both `data` and `sed_bands`.

Returns **wavelen** : np.ndarray

central wavelength of bands in micron

mags : np.ndarray

magnitude in band

mags_error : np.ndarray

error on magnitude

sed : np.ndarray

flux in Jy

YSOVAR.atlas.**makecrossids**(*data1, data2, radius, ra1='RAdeg', dec1='DEdeg', ra2='ra', dec2='dec', double_match=False*)

Cross-match two lists of coordinates, return closest match

This routine is not very clever and not very fast. It should be fine up to a hundred thousand entries per list.

Parameters **data1** : astropy.table.Table or np.recarray

This is the master data, i.e. for each element in data1, the results wil have one (or zero) index numbers in data2, that provide the best match to this entry in data1.

data2 : astropt.table.Table or np.recarray

This data is matched to data1.

radius : np.float or array

maximum radius to accept a match (in degrees); either a scalar or same length as data2

ra1, dec1, ra2, dec2 : string

key for access RA and DEG (in degrees) the the data, i.e. the routine uses *data1[ra1]* for the RA values of data1.

double_match : bool

If true, one source in data2 could be matched to several sources in data1. This can happen, if a source in data2 lies between two sources of data1, which are both within `radius`. If this switch is set to `False`, then a strict one-on-one matching is enforced, selecting the closest pair in the situation above.

Returns **cross_ids** : np.ndarray

Will have len(data1). For each elelment it contains the index of data2 that provides the best match. If no match within *radius* is found, then entry will be -99999.

YSOVAR.atlas.**makecrossids_all**(*data1*, *data2*, *radius*, *ra1='RAdeg'*, *dec1='DEdeg'*, *ra2='ra'*, *dec2='dec'*, *return_distances=False*)

Cross-match two lists of coordinates, return all matches within radius

This routine is not very clever and not very fast. If should be fine up to a hundred thousand entries per list.

Parameters **data1** : `astropy.table.Table` or np.recarray

This is the master data, i.e. for each element in data1, the results wil have the index numbers in data2, that provide the best match to this entry in data1.

data2 : `astropy.table.Table` or np.recarray

This data is matched to data1.

radius : np.float or array

maximum radius to accept a match (in degrees)

ra1, dec1, ra2, dec2 : string

key for access RA and DEG (in degrees) the the data, i.e. the routine uses *data1[ra1]* for the RA values of data1.

return_distances : bool

decide if distances should be returned

Returns **cross_ids** : list of lists

Will have len(data1). For each elelment it contains the indices of data2 that are within *radius*. If no match within *radius* is found, then the entry will be *[]*.

distances : list of lists

If `return_distances==True` this has the same format a `cross_ids` and contains the distance to the match in degrees.

YSOVAR.atlas.**merge_lc**(*d*, *bands*, *t_simul=0.01*)

merge lightcurves from several bands

This returns a lightcurve that contains only entries for those times, where *all* required bands have an entry.

Parameters **d** : dictionary

as obtained from `YSOVAR.atlas.add_ysovar_mags()`

bands : list of strings

labels of the spectral bands to be merged, e.g. ['36','45']

t_simul : float

max distance in days to accept datapoints in band 1 and 2 as simultaneous In L1688 and IRAS 20050+2720 the distance between band 1 and 2 coverage is within a few minutes, so a small number is sufficent to catch everything and to avoid false matches.

Returns **tab** : astropy.table.Table

This table contains the merged lightcurve and contains times, fluxes and errors.

YSOVAR.atlas.**phase_fold**(*time*, *period*)

Phase fold a set of time on a period

Parameters **time** : np.ndarray

array of times

period : np.float

YSOVAR.atlas.**radec_from_dict**(*data*, *RA='ra'*, *DEC='dec'*)

return ra dec numpy array for list of dicts

Parameters **data** : list of several *dict*

RA, DEC : strings

keys for RA and DEC in the dictionary

Returns **radec** : np record array with RA, DEC columns

YSOVAR.atlas.**sed_slope**(*data, sed_bands={'Bmag': ['e_Bmag', 0.43, 4000.87], '5.8mag': ['e_5.8mag', 5.8, 115.0], 'mean_36': ['e_3.6mag', 3.6, 280.9], 'simbad_B': [None, 0.43, 4000.87], '4.5mag': ['e_4.5mag', 4.5, 179.7], 'Vmag': ['e_Vmag', 0.623, 3597.28], 'Imag': ['e_Imag', 0.798, 2587], 'Kmag': ['e_Kmag', 2.159, 666.7], 'simbad_V': [None, 0.623, 3597.28], 'Rmag': ['e_Rmag', 0.759, 3182], '3.6mag': ['e_3.6mag', 3.6, 280.9], 'mean_45': ['e_4.5mag', 4.5, 179.7], 'imag': ['e_imag', 0.763, 2515.7], 'Jmag': ['e_Jmag', 1.235, 1594], 'rmag': ['e_rmag', 0.622, 3173.3], 'Hmag': ['e_Hmag', 1.662, 1024], '8.0mag': ['e_8.0mag', 8.0, 64.13], '24mag': ['e_24mag', 24.0, 7.14], 'Umag': ['e_Umag', 0.355, 1500], 'nomad_Rmag': [None, 0.759, 3182], 'nomad_Bmag': [None, 0.43, 4000.87], 'Hamag': ['e_Hamag', 0.656, 2974.4], 'nomad_Vmag': [None, 0.623, 3597.28]}*)

fit the SED slope to data for all bands in `data` and `sed_bands`

Parameters **data** : *YSOVAR.atlas.YSOVAR_atlas* or `astropy.table.Table`

input data that has arrays of magnitudes for different bands

sed_bands : dict

keys must be the name of the field that contains the magnitudes in each band, entries are lists of [name of error field, wavelength in micron, zero_magnitude_flux_freq in Jy]

Returns **slope** : float

slope of the SED determined with a least squares fit. Return `np.nan` if there is too little data.

YSOVAR.atlas.**val_from_dict**(*data*, *name*)

return ra dec numpy array for list of dicts

Parameters **data** : list of dict

name : strings

keys for entry in the dictionary

Returns **col** : list of values

CHAPTER 3

Register functions to automatically generate columns in the atlas

Provide a registry of functions to analyse lightcurves.

YSOVAR.atlas.YSOVAR_atlas objects can autogenerate a certain number of columns in the data table. This module provides the mechanism for defining the functions that are required for this autogeneration to work.

For each function a certain number of metadata information is required, so that *YSOVAR.atlas.YSOVAR_atlas* can find them and call them with the right parameters.

3.1 Example

This is probably best explained by an example:

```
import numpy as np
from YSOVAR import registry
from YSOVAR import atlas

registry.register(np.mean, n_bands=1, error = False,
                  time = False, name = "mean", default_colnames = ["mean"],
                  description = "mean of lightcurve")
```

The modules uses sensible defaults, e.g. the name of the function is used as default for the autogenerated column name, so, the following would also work:

```
registry.register(np.mean, n_bands=1, error = False, time = False)
```

After this function is registered, you can now automatically generate columns for the mean values:

```
<read your datafile>
my_cloud  = atlas.YSOVAR_atlas(lclist = your_data_here)
temp = my_cloud['mean_36']
temp = my_cloud['mean_45']
```

Note: In practice, you do not need to register `np.mean` because a selection of common functions is automatically registered when the module is imported. The following command will list the available functions:

```
registry.list_lcfuncs()
```

3.2 Call signatures for functions

The string in the colname is split in the name that will be used to look up the function in the function registry (`'mean'`) and the name of the band (`'36'`). For this reason, registered function names cannot contain underscores.

Registered functions have to follow a convention on which inputs they accept:

```
def func([time],band1, band2, ..., [band1_err, band2_err ...])
```

All inputs will be numpy arrays. `time = True / False` controls if the time array is present, `error = True / False` if the uncertainties for each band are passed in. `n_bands` says how many bands are expected on input.

class `YSOVAR.registry.`**`LightcurveFunc`**(*func, n_bands, error, time, name=''', default_colnames=[], default_colunits=[None], default_coldescriptions=None, other_cols={}, description=''', kwargs={}*)

Wrap function for the lightcurve analysis function registry

This wrapper is to be used for fucntions that operate on individual lightcurves.

Functions in the registry of analysis function for lightcurves need some metadata for make sure that they can be called correctly, when `atlas.YSOVAR_atlas` autogenerated columns in the table.

This class warps a function and provides some metadata. This metadata includes:

- The number of bands the function requires as input.
- Does the function require the uncertainty of the mags?
- Does the function require the time of the observations?
- What is the name of the function, so that it can be found?
- Some short description of the function.
- Default names for the columns that are autogenerated.

Use the following command to read the full docstring of the function that is wrapped in this object:

```
help(my_object.func)
```

Note: This class is usually not called directly. Use *`YSOVAR.registry.register()`*.

`kwargs = {}`

`YSOVAR.registry.`**`list_lcfuncs`**`()`

List all function currently registered with a one-line summary

`YSOVAR.registry.`**`register`**(*func*, *n_bands=1*, *error=False*, *time=False*, *force=False*, ***kwargs*)

Register a new function for lightcurve analysis

Parameters **func** : function or callable object

This function will be called.

n_bands : int

Number of spectral bands required by this function.

error : bool

If `True` the uncertainties in each lightcurve band will be passed to the function.

time : bool

If `True` the observations times will be passed as first argument.

force : bool

If `True` a function of same name that was previously registered will be over written.

All remaining keywords will be passed to :func:‘LightcurveFunc.__init__‘. :

See that function for description and default values of other :

accepted keywords. :

CHAPTER 4

Plot the atlas

This module holds specialized plotting function for YSOVAR data.

This module hold some plotting functions for YSOVAR data, i.e. lightcurves and color-color or color-magnitude diagrams. All the plotting is done with matplotlib.

4.1 Module level variables

There are several module level variables, that define defaults for plots.

Default offset of x-axis for lightcurve plots:

```
YSOVAR.plot.mjdoffset = 55000
```

List of all formats for output for those routines that make e.g. the lightcurve plot for each source. This has to be a list *even if it contains only one element*. List multiple formats to obtain each image in each format:

```
YSOVAR.plot.filetype  = ['.eps']
```

This routine can connect to the internet and download thumbnail images from the YSOVAR database. To do so, you need to set the username and password for the YSOVAR database:

```
YSOVAR.plots.YSOVAR_USERNAME = 'username'
YSOVAR.plots.YSOVAR_PASSWORD = 'password'
```

4.2 Plotting functions

The most important purpose of this module is to provide functions that can generate a big atlas, which holds some key diagnostic plots (e.g. the lightcurve, color-mag diagram) for every star in a *YSOVAR.atlas.YSOVAR_atlas* object. However, some of the functions are also useful for stand-alone plots. Functions that generate multiple plots often start with `make` and have a plural name (e.g. *YSOVAR.plot.make_lc_plots()*). These function then call a function that makes the individual plot (and sometimes that is broken down again into one function that sets up the figure and the axis and a second one that executes the actual plot command), see *YSOVAR.plot.plot_lc()* and *YSOVAR.plot.lc_plot()*.

Use those functions to plot individual lightcurves e.g. for a paper.

YSOVAR.plot.**cmd_plot**(*atlas*, *mergedlc*, *redvec=None*, *verbose=True*)

Parameters **atlas** : ysovar_atlas.YSOVAR_atlas with one row only

mergedlc : np.ndarray

contains 't' as time for merged lightcurves and 'm36' and 'm45' as magnitues for lightcurves

redvec : float

slope of reddening vector in the CMD. If `None` use default.

YSOVAR.plot.**extraplots_1**()

YSOVAR.plot.**extraplots_2**(*data*, *infos*, *outroot_overview*)

YSOVAR.plot.**fancyplot**(*x*, *y*, *outroot*, *filename*, *colors*, *ind*, *xlabel*, *ylabel*, *marker*, *xlog=False*, *ylog=False*, *legendtext=''*, *legendpos='upper right'*)

YSOVAR.plot.**get_stamps**(*data*, *outroot*, *verbose=True*)

Retrieve stamp images from the YSOVAR2 database

The database requires a login. To get past that set the following variables before calling this routine: - YSOVAR.plot.YSOVAR_USERNAME - YSOVAR.plot.YSOVAR_PASSWORD

Parameters **data** : ysovar_atlas.YSOVAR_atlas

needs to contain a column called *YSOVAR2_id*

outroot : string

directory where the downloaded files will end up

YSOVAR.plot.**lc_plot**(*catalog*, *xlim=None*, *twinx=True*)

plot one or two lcs for a single object

Parameters **catalog** : single row from YSOVAR.atlas.YSOVAR_atlas

contains 't1' and / or 't2' as time for lightcurves and 'm1' and / or 'm2' as magnitues for lightcurves

xlim : None or list

None auto scales the x-axis list of [x0, x1] scales the xaxis form x1 to x2 list of lists [[x00,x01], [x10, x11], ...] splits in multiple panels and each [x0, x1] pair gives the limits for one panel.

twinx : boolean

if true make seperate y axes for IRAC1 and IRAC2 if both are present

YSOVAR.plot.**make_cmd_plots**(*atlas*, *outroot*, *verbose=True*)

plot cmds into files for all objects in *atlas*

Parameters **atlas** : ysovar_atlas.YSOVAR_atlas

contains dict with 't36' and / or 't45' as time for lightcurves and 'm36' and / or 'm45' as magnitues for lightcurves

YSOVAR.plot.**make_info_plots**(*infos, outroot, bands=['36', '45'], bandlabels=['[3.6]', '[4.5]']*)

YSOVAR.plot.**make_latexfile**(*atlas, outroot, name, ind=None, plotwidth='0.45\\textwidth', output_figs=[['_lc', '_color'], ['_ls', '_sed'], ['_lc_phased', '_color_phased'], ['_stamp', '_lcpoly']], output_cols={'simbad_MAIN_ID': 'ID Simbad', 'YSOVAR2_id': 'ID in YSOVAR 2 database', 'median_36': 'median [3.6]', 'mad_45': 'medium abs dev [4.5]', 'mad_36': 'medium abs dev [3.6]', 'stddev_36': 'stddev [3.6]', 'IRclass': 'Rob class', 'stetson_36_45': 'Stetson [3.6] vs. [4.5]', 'stddev_45': 'stddev [4.5]', 'median_45': 'median [4.5]', 'simbad_SP_TYPE': 'Simbad Sp type'}, pdflatex=True*)

make output LeTeX file that produces an atlas

This procedure actually checks the directory *outroot* and only includes Figures in the LaTeX document that are present there. For some stars (e.g. if they only have one lightcurve) certain plots may never be produced, so this strategy will ensure that the LaTeX document compiles in any case. It also means that files, that are not present because you forgot to produce them, will not be present.

Parameters **atlas** : ysovar_atlas.YSOVAR_atlas

This is the atlas with the data to be plotted

outroot : string

path to directory where all figures are found. The LeTeX file will be written in the same directory.

name : string

filename of the atlas file (without the *.tex*)

ind : index array

Only objects in this index array will be included in the LaTeX file. Use *None* to output the entire atlas.

plotwidth : string

width of the plots in LeTeX notation. It is the users responsibility to ensure that the plots chosen with *output_figs* fit on the page.

output_figs : list of lists

List of file extensions of plots to be included. Filenames will be of format *i* + *fileextension*. This is a list of lists in the form:

```
`[[fig1_row1, fig2_row1, fig3_row1], [fig1_row2, ...]]`
```

Each row in the figure grid can have a different number of figures, but it is the users responsibility to choose *plotwidth* so that they all fit on a page.

output_cols : dictionary

Select columns in the table to print out below the figures. Format is *{'colname': 'label'}*, where label is what will appear in the LaTeX document.

pdflatex : bool

if *True* check for files that pdflatex uses (jpg, png, pdf), otherwise for fiels LaTeX uses (ps, eps).

YSOVAR.plot.**make_lc_cmd_plots**(*atlas*, *outroot*, *lc_xlim=None*, *lc_twinx=False*)

plot cmds and lc

See *meth:make_lc_plots* and *meth:make_cmd_plots* for documentation.

Parameters **atlas** : ysovar_atlas.YSOVAR_atlas

contains dict with 't36' and / or 't45' as time for lightcurves and 'm36' and / or 'm45' as magnitues for lightcurves

YSOVAR.plot.**make_lc_plots**(*atlas*, *outroot*, *verbose=True*, *xlim=None*, *twinx=False*, *ind=None*, *filedescription='_lc'*)

plot lightcurves into files for all objects in *atlas*

Parameters **atlas** : ysovar_atlas.YSOVAR_atlas

contains dict with 't36' and / or 't45' as time for lightcurves and 'm36' and / or 'm45' as magnitues for lightcurves

verbose : boolean

if true print progress in processing

xlim : None or list

None auto scales the x-axis list of [x0, x1] scales the xaxis form x1 to x2 list of lists [[x00,x01], [x10, x11], ...] splits in multiple panels and each [x0, x1] pais gives teh limits for one panel.

twinx : boolean

if true make separate y axes for IRAC1 and IRAC2 if both are present

ind : list of integers

index numbers of elements, only for those elements a lightcurve is created. If None, make lightcurve for all sources.

filedescription : string

Output files are named `YSOVAR2_id + filedescription + extension`. The extension(s) is specified in `YSOVAR.plots.filetype`. Use the `filedescription` parameters if this method is called more than once per star.

`YSOVAR.plot.`**`make_ls_plots`**(*atlas*, *outroot*, *maxper*, *oversamp*, *maxfreq*, *verbose=True*)

calculates & plots Lomb-Scargle periodogram for each source

Parameters **atlas** : ysovar_atlas.YSOVAR_atlas

input atlas, which includes lightcurves

outroot : string

data path for saving resulting files

maxper : float

maximum period to be used for periodogram

oversamp : integer

oversampling factor

maxfreq : float

maximum frequency to be used for periodogram

verbose : bool

Show progress as output?

`YSOVAR.plot.`**`make_phased_lc_cmd_plots`**(*atlas, outroot, bands=['36', '45'], marker=['o', '+'], lw=[0, 1], colorphase=True, lc_name='_lc_phased'*)

plots phased lightcurves and CMDs for all sources

Parameters **atlas** : ysovar_atlas.YSOVAR_atlas

input atlas, which includes lightcurves

outroot : string

data path for saving resulting files

bands : list of strings

band identifiers

marker : list of valid matplotlib markers (e.g. string)

marker for each band

lw : list of floats

linewidth for each band

lc_name : string

filenames for phased lightcurve plots

colorphase : bool

If true, entries in the lightcurves will be color coded by phase, if not, by time (to see if there are e.g. phase shifts over time).

YSOVAR.plot.**make_plot_skyview**(*outroot*, *infos*)

YSOVAR.plot.**make_reddeningvector_for_plot**(*x1*, *x2*, *y1*, *y2*)

YSOVAR.plot.**make_sed_plots**(*infos*, *outroot*, *title='SED (data from Guenther+ 2012)'*, *sed_bands={}*)

YSOVAR.plot.**make_slope_plot**(*infos*, *outroot*)

YSOVAR.plot.**multisave**(*fig*, *filename*)

YSOVAR.plot.**plot_lc**(*ax*, *data*, *mergedlc*)
plot lc in a given axes container

Parameters **data** : dictionary

contains 't36' and / or 't45' as time for lightcurves and 'm36' and / or 'm45' as magnitues for lightcurves

YSOVAR.plot.**plot_polys**(*atlas*, *outroot*, *verbose=True*)
plot lightcurves into files for all objects in data

Parameters **atlas** : YSOVAR.atlas.YSOVAR_atlas

each ls in the atlas contains 't36' and / or 't45' as time for lightcurves and 'm36' and / or 'm45' as magnitues for lightcurves

outroot : string

data path for saving resulting files

verbose : boolean

if true print progress in processing

YSOVAR.plot.**setup_lcplot_axes**(*data*, *xlim*, *twinx=True*, *fig=None*)
set up axis containers for one or two lcs for a single object.

This function checks the xlim and divides the space in the figure so that the xaxis has the same scale in each subplot.

Parameters **data** : dict

This lightcurve is inspected for the number of bands present

xlim : None or list

None auto scales the x-axis list of [x0, x1] scales the xaxis form x1 to x2 list of lists [[x00,x01], [x10, x11], ...] splits in multiple panels and each [x0, x1] pair gives the limits for one panel.

twinx : boolean

if true make seperate y axes for IRAC1 and IRAC2 if both are present

fig: matplotlib figure instance or ‘‘None‘‘ :

If `None`, it creates a figure with the matplotlib defaults. Pass in a figure instance to customize e.g. the figure size.

Returns **fig** : matplotlib figure instance

axes : list of matplotlib axes instances

This list holds the default axes (axis labels at the left and bottom).

taxes : list of matplotlib axes instances

This list holds the twin axes (labels on bottom and right).

CHAPTER 5

Lomb-Scargle periodograms

Fast algorithm for spectral analysis of unevenly sampled data

The Lomb-Scargle method performs spectral analysis on unevenly sampled data and is known to be a powerful way to find, and test the significance of, weak periodic signals. The method has previously been thought to be 'slow', requiring of order 10(2)N(2) operations to analyze N data points. We show that Fast Fourier Transforms (FFTs) can be used in a novel way to make the computation of order 10(2)N log N. Despite its use of the FFT, the algorithm is in no way equivalent to conventional FFT periodogram analysis.

Keywords: DATA SAMPLING, FAST FOURIER TRANSFORMATIONS, SPECTRUM ANALYSIS, SIGNAL PROCESSING

Example:

```
>>> import numpy
>>> import lombscargle
>>> x = numpy.arange(10)
>>> y = numpy.sin(x)
>>> fx,fy, nout, jmax, prob = lombscargle.fasper(x,y, 6., 6.)
```

Reference: Press, W. H. & Rybicki, G. B. 1989 ApJ vol. 338, p. 277-280. Fast algorithm for spectral analysis of unevenly sampled data bib code: 1989ApJ...338..277P

`YSOVAR.lombscargle.`**`fasper`**(*x*, *y*, *ofac*, *hifac*, *MACC=4*)

function fasper

Given abscissas x (which need not be equally spaced) and ordinates y, and given a desired oversampling factor `ofac` (a typical value being 4 or larger). This routine creates an array `wk1` with a sequence of `nout` increasing frequencies (not angular frequencies) up to `hifac` times the "average" Nyquist frequency, and creates an array `wk2` with the values of the Lomb normalized periodogram at those frequencies. The arrays `x` and `y` are not altered. This routine also returns `jmax` such that `wk2[jmax]` is the maximum element in `wk2`, and `prob`, an estimate of the significance of that maximum against the hypothesis of random noise. A small value of `prob` indicates that a significant periodic signal is present.

Reference: Press, W. H. & Rybicki, G. B. 1989 ApJ vol. 338, p. 277-280. Fast algorithm for spectral analysis of unevenly sampled data (1989ApJ...338..277P)

History:

02/23/2009, v1.0, MF Translation of IDL code (orig. Numerical recipies)

Parameters **X** : array

Abscissas (e.g. an array of times).

Y : array

Ordinates (e.g. corresponding counts).

Ofac : integer

Oversampling factor.

Hifac : float

Hifac * "average" Nyquist frequency = highest frequency for which values of the Lomb normalized periodogram will be calculated.

Returns **Wk1** : array

Lomb periodogram frequencies.

Wk2 : array

corresponding values of the Lomb periodogram.

Nout : tuple

Wk1 & Wk2 dimensions (number of calculated frequencies)

Jmax : integer

The array index corresponding to the MAX(Wk2).

Prob : float

False Alarm Probability of the largest Periodogram value

MACC : integer

Number of interpolation points per 1/4 cycle of highest frequency

YSOVAR.lombscargle.**getSignificance**(*wk1*, *wk2*, *nout*, *ofac*)
returns the peak false alarm probabilities

Hence the lower is the probability and the more significant is the peak

YSOVAR.lombscargle.**lombscargle**(*time*, *mag*, *maxper=15.0*, *oversamp=4*, *maxfreq=1.0*)
calculate Lomb-Scargle periodograms for all sources

A new column is added to the datatable that contains the result. (If the column exists before, it is overwritten).

Parameters **time** : np.ndarray

times of observation

mag : np.ndarray

Observed magnitudes

maxper : float

periods above this value will be ignored

oversamp : integer

oversampling factor

maxfreq : float

max freq of LS periodogram is maxfeq * "average" Nyquist frequency For very inhomogenously sampled data, values > 1 can be useful

CHAPTER 6

Analyse the timing in the lightcurves

The routines here are for non-periodic timing, see *YSOVAR.lombscargle* for periodograms.

YSOVAR.lightcurves.**ARmodel**(*t*, *val*, *degree=2*, *scale=0.5*)

Fit an auto-regressive (AR) model to data and retrn some parameters

The inout data can be irregularly binned, it will be resampled on a regular grid with bin-width `scale`.

Parameters **t** : np.ndarray

input times

val : np.ndarray

input values

degree : int

degree of AR model

scale : float

binning ofthe resampled lightcurve

Returns **params** : list of `(degree + 1)` floats

parameters of the model

sigma2 : float

sigma of the Gaussian component of the model

aic : float

value of the Akaike information criterion

YSOVAR.lightcurves.**calc_poly_chi**(*data, bands=['36', '45']*)

Fits polynoms of degree 1..6 to all lightcurves in data

One way to adress if a lightcurve is "smooth" is to fit a low-order polynomial. This routine fits polynomial of degree 1 to 6 to each IRAC1 and IRAC 2 lightcurve and calculates the chi^2 value for each fit.

Parameters **data** : astropy.table.Table

structure with the defined object properties.

bands : list of strings

Band identifiers, e.g. ['36', '45'], can also be a list with one entry, e.g. ['36']

YSOVAR.lightcurves.**combinations_with_replacement**(*iterable*, *r*)

defined here for backwards compatibility From python 2.7 on its included in itertools

YSOVAR.lightcurves.**corr_points**(*x*, *data1*, *data2*)

Make all combinations of two variables at times x

Parameters **x** : np.ndarray

independend variable (x-axis), e.g. time of a lightcurve

data1, data2 : np.ndarray

dependent variables (y-axis), e.g. flux for a lightcurve

Returns **diff_x** : np.ndarray

all possible intervals of the independent variable

d_2 : 2-d np.ndarray

corresponding values of dependent variables. Array as shape (N, 2), where N is the number of combinations.

YSOVAR.lightcurves.**delta_corr_points**(*x*, *data1*, *data2*)

correlate two variables sampled at the same (possible irregular) time points

Parameters **x** : np.ndarray

independend variable (x-axis), e.g. time of a lightcurve

data1, data2 : np.ndarray

dependent variables (y-axis), e.g. flux for a lightcurve

Returns **diff_x** : np.ndarray

all possible intervals of the independent variable

d_2 : np.ndarray

corresponding correlation in the dependent variables

..note:: :

Essentially, this is a correltation function for irregularly sampled data

YSOVAR.lightcurves.**delta_delta_points**(*data1*, *data2*)

make a list of scatter delta_data1 vs delta_data2 for all combinations of

E.g. this can be used to calculate delta_T vs. delta mag

Parameters **data1** : np.ndarray

independend variable (x-axis), e.g. time of a lightcurve

data2 : np.ndarray

dependent variable (y-axis), e.g. flux for a lightcurve

Returns **diff_1** : np.ndarray

all possible intervals of the independent variable

diff_2 : np.ndarray

corresponding differences in the depended variable

..note:: :

Essentially, this is an autocorrelation for irregularly sampled data

YSOVAR.lightcurves.**describe_autocorr**(*t*, *val*, *scale=0.1*, *autocorr_scale=0.5*, *autosum_limit=1.75*)

describe the timescales of time series using an autocorrelation function

#This procedure takes an unevenly sampled time series and computes #the autocorrelation function from that. The result is binned in time bins #of width *scale* and three numbers are derived from the shape of the #autocorrelation function.

This is based on the definitions used by Maria for the Orion paper. A visual definition is given on the YSOVAR wiki (restriced access).

Parameters **t** : np.ndarray

times of time series

val : np.ndarray

values of time series

scale : float

In order to accept irregular time series, the calculated autocorrelation needs to be binned in time. `scale` sets the width of those bins.

autocorr_scale : float

`coherence_time` is the time when the autocorrelation falls below `autocorr_scale`. `0.5` is a common value, but for sparse sampling `0.2` might give better results.

autosum_limit : float

The autocorrelation function is also calculated with a time binning of `scale`. To get a robust measure of this, the function calculate the timescale for the cumularitve sum of the autocorrelation function to exceed `autosum_limit`.

Returns **cumsumtime** : float

time when the cumulative sum of a finely binned autocorrelation function exceeds `autosum_limit` for the first time; `np.inf` is returned if the autocorrelation function never reaches this value.

coherence_time : float

time when the autocorrelation function falls below `autocorr_scale`

autocorr_time : float

position of first positive peak

autocorr_val : float

value of first positive peak

YSOVAR.lightcurves.**discrete_struc_func**(*t*, *val*, *order=2*, *scale=0.1*)

discrete structure function

Parameters **t** : np.ndarray

times of time series

val : np.ndarray

values of time series

order : float

the exponent of the structure function

scale : float

In order to accept irregular time series, the calculated autocorrelation needs to be binned in time. `scale` sets the width of those bins.

Returns **timebins** : np.ndarray

time bins corresponding to the values in `dsf`

dsf : np.ndarray

binned and averaged discrete structure function

`YSOVAR.lightcurves.`**`fit_poly`**(*x*, *y*, *yerr*, *degree=2*)

Fit a polynom to a dataset

..note:: For numerical stability the `x` values will be shifted, such that x[0] = 0!

Thus, the parameters describe a fit to this shifted dataset!

Parameters **x** : np.ndarray

array of independend variable

y : np.ndarray

array of dependend variable

yerr: np.ndarray :

uncertainty of y values

degree : integer

degree of polynomial

Returns **res_var** : float

residual of the fit

shift : float

shift applied to x value for numerical stability.

beta : list

fit parameters

`YSOVAR.lightcurves.`**`gauss_kernel`**(*scale=1*)

return a Gauss kernel

Parameters **scale** : float

width (sigma) of the Gauss function

Returns **kernel** : function

kernel(x, loc), where *loc* is the center of the Gauss and *x* are the bin boundaries.

`YSOVAR.lightcurves.`**`normalize`**(*data*)

normalize data to mean = 1 and stddev = 1

Parameters **data** : np.array

input data

Returns **data** : np.array

normalized set of data

YSOVAR.lightcurves.**plot_all_polys**(*x*, *y*, *yerr*, *title=''*)

plot polynomial fit of degree 1-6 for a dataset

Parameters **x** : np.ndarray

array of independend variable

y : np.ndarray

array of dependend variable

yerr : np.ndarray

uncertainty of y values

title : string

title of plot

Returns **fig** : matplotlib.figure instance

YSOVAR.lightcurves.**slotting**(*xbins*, *x*, *y*, *kernel=None*, *normalize=True*)

Add up all the y values in each x bin

xbins defines a (possible non-uniform) bin grid. For each bin, find all (x,y) pairs that belong in the x bin and add up all the y values in that bin. Optionally, the x values can be convolved with a kernel before, so that each y can contribute to more than one bin.

Parameters **xbins** : np.ndarray

edges of the x bins. There are *len(xbins)-1* bins.

x, y : np.ndarry

x and y value to be binned

kernel : function

Kernel input is binedges, kernel output bin values: Thus, len(kernelout) must be len(kernelin)-1! The kernal output should be normalized to 1.

normalize : bool

If false, get the usual correlation function. For a regularly sampled time series, this is the same as zero-padding on the edges. For *normalize = true* divide by the number of entries in a time bin. This avoids zero-padding, but leads to an irregular "noise" distribution over the bins.

Returns **out** : np.ndarray

resulting array of added y values

n : np.ndarray

number of entries in wach bin. If *kernel* is used, this can be non-integer.

CHAPTER 7

Distances on the sky

Calculate distances on the celestial sphere

Methods in here are required as helper functions for cross-matching sources by position. Different algorithms with different numerical stability are implemented, but usually you should not need to call this directly. See the source code of *YSOVAR.atlas.dict_from_csv()* for an example how how this works.

YSOVAR.great_circle_dist.**Haversine**(*phi_0*, *lam_0*, *phi_1*, *lam_1*)

Calculates the angular distance between point 0 and point 1

uses the Haversine function, is numerically stable, except for antipodal points http://en.wikipedia.org/wiki/Great-circle_distance

Parameters **phi_0** : float or numpy array

lattitude phi of point 0

lam_0 : float or numpy array

longitude lambda of point 0

phi_1 : float or numpy array

lattitude phi of point 1

lam_1 : float or numpy array

longitude lambda of point 1

Returns **dist** : float or numpy array

angular distance on great circle

YSOVAR.great_circle_dist.**Vincenty**(*phi_0*, *lam_0*, *phi_1*, *lam_1*)

Calculates the angular distance between point 0 and point 1

uses a special case of the Vincenty formula (which is for ellipsoides) numerically accurate, but computationally intensive see http://en.wikipedia.org/wiki/Great-circle_distance

Parameters **phi_0** : float or numpy array

lattitude phi of point 0

lam_0 : float or numpy array

longitude lambda of point 0

phi_1 : float or numpy array

lattitude phi of point 1

lam_1 : float or numpy array

longitude lambda of point 1

Returns **dist** : float or numpy array

angular distance on great circle

YSOVAR.great_circle_dist.**dist**(*phi_0*, *lam_0*, *phi_1*, *lam_1*, *unit=None*)

Calculates the angular distance between point 0 and point 1

see http://en.wikipedia.org/wiki/Great-circle_distance

Parameters **phi_0** : float or numpy array

lattitude phi of point 0

lam_0 : float or numpy array

longitude lambda of point 0

phi_1 : float or numpy array

lattitude phi of point 1

lam_1 : float or numpy array

longitude lambda of point 1

Returns **dist** : float or numpy array

angular distance on great circle

YSOVAR.great_circle_dist.**dist_radec**(*phi_0*, *lam_0*, *phi_1*, *lam_1*, *unit=None*)

Calculates the angular distance between point 0 and point 1

see http://en.wikipedia.org/wiki/Great-circle_distance

Parameters **ra0** : float or numpy array

RA of point 0

dec0 : float or numpy array

DEC of point 0

ra1 : float or numpy array

RA of point 1

dec1 : float or numpy array

DEC of point 1

Returns **dist** : float or numpy array

angluar distance on great circle

YSOVAR.great_circle_dist.**dist_radec_fast**(*ra0*, *dec0*, *ra*, *dec*, *scale=inf*, **arg*, ***kwargs*)

Calculates the angular distance between point 0 and point 1

Only if delta_dec is < scale, the full trigonometric calculation is done, otherwise return np.inf

see http://en.wikipedia.org/wiki/Great-circle_distance

Parameters **ra0** : float or numpy array

RA of point 0

dec0 : float or numpy array

DEC of point 0

ra1 : float or numpy array

RA of point 1

dec1 : float or numpy array

DEC of point 1

Returns **dist** : float or numpy array

angluar distance on great circle

..note:: :

To be done:

- cut on RA as well, but that requires knowledge of scale and the decorator transforms ra, dec only
- merge this with ra_dec_dist and do fast verion if scale != None

YSOVAR.great_circle_dist.**simple**(*phi_0*, *lam_0*, *phi_1*, *lam_1*)

Calculates the angular distance between point 0 and point 1

uses a very simple formula, prone to numeric inaccuracies see http://en.wikipedia.org/wiki/Great-circle_distance

Parameters **phi_0** : float or numpy array

lattitude phi of point 0

lam_0 : float or numpy array

longitude lambda of point 0

phi_1 : float or numpy array

lattitude phi of point 1

lam_1 : float or numpy array

longitude lambda of point 1

Returns **dist** : float or numpy array

angluar distance on great circle

YSOVAR.great_circle_dist.**simple_decorator**(*decorator*)

This decorator can be used to turn simple functions into well-behaved decorators, so long as the decorators are fairly simple. If a decorator expects a function and returns a function (no descriptors), and if it doesn't modify function attributes or docstring, then it is eligible to use this. Simply apply @simple_decorator to your decorator and it will automatically preserve the docstring and function attributes of functions to which it is applied.

YSOVAR.great_circle_dist.**unitchecked**(*f*)

Decorator to transfrom units of angles

This decorator transforms units of angle, before they are fed into any a function to calculate the angular distance. It expects the unit as a keyword and transforms two sets of angular coordinates (phi_0, lam_0, phi_1, lam_1) to radian, calls the function and converts the output (in radian) into the unit of choice.

CHAPTER 8

Generate atlas

This module holds helper routines that are not part of the main package.

Examples for this are:

```
- Routines that were written for a specific cluster and are not
  generalized yet.
- I/O routines that are not directly related to the YSOVAR database
  (e.g. reading the IDL .sav files that Rob Gutermuth uses).
```

YSOVAR.misc.**format_or_string**(*format_str*)

YSOVAR.misc.**makeclassinteger**(*guenther_data_yso*)

YSOVAR.misc.**read_cluster_grinder**(*filepath*)

Import Robs Spitzer data

read Rob's IDL format and make it into a a catalog, deleting multiple columns and adding identifiers

Parameters **filepath** : string

Path to a directory that holds the output of the ClusterGrinder pipeline. All files need to have standard names. Specifically, this routine reads:

- `cg_merged_srclist_mips.sav`
- `cg_classified.sav`

Returns **cat** : astropy.table.Table

Table with 2MASS ans Spitzer magnitudes and the clustergrinder classification.

YSOVAR.misc.**spectra_check**(*infos*, *ra*, *dec*, *files*, *night_id*, *radius*)

CHAPTER 9

License

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

y

y

N

P

R

S

U

V

Y