MapClass Documentation

Release 0.1.0

Christof Buchbender

June 01, 2012

CONTENTS

Contents:

CHAPTER

ONE

MAPCLASS/MAPTOOLS TUTORIAL

This tutorial explains how to work with the mapclass module to easily analyze and modify astronomical images.

1.1 Motivation

Before more details are given.

Say you have a number of maps of a galaxy at different wavelengths covering the dust sed and you want to smooth them all to the same resolution. They are all stored under one folder (with possible subfolders) and follow the "Naming Convention" I introduce below. Using MapClass and MapTools this task could be done as follows

```
>>> from astrolyze.mapclass import *
>>> from astrolyze.maptools import *
>>>          =          .
>>>                   = 40
>>>                       = '../anyfolder'
>>>          .
>>>
```

CHAPTER

TWO

MAPS

The Map class is the parent class for the handling of astronomical map files in the FITS, GILDAS and MIRIAD data formats. It is meant to make handling maps easier by wrapping functions of GILDAS and MIRIAD and providing new functions in python on the basis of pyfits. It is using a 'Name Convention' for ease of use. Meaning that the file name already includes basic information about the map it contains. A Name that follows this 'Convention' is eg:

M33_30m-HERA_CO21_Ta*_12_cube.fits

All items **MUST** be seperated by an undescore (_) and have to include at minimum the following properties:

1. source
2. telescope
3. wavelength OR frequency OR lineName
4. flux unit
5. resolution

Additionaly the map Class recognizes all following items as comments. In the name example above the comment be "cube". Comments are not transfered to internal variables of the map objects.

The last item is followed by the files extension:

- .fits -> FITS
- .gdf -> GILDAS
- nothing -> MIRIAD (Miriads file format uses directories to store the data)

Maps that are not following this name convention are not supported to assure that all parts of the program work, since they mostly depend on the items set as will be explained below.

Also it makes the life of your fellow astronomers easier when they have to work with your data since they directly know their basic properties.

2.1 Main

class `astrolyze.maps.main.`**`Map`**(*mapName*, *nameConvention=True*)

Parent Class with functions common to all three data Formats fits, Gildas and Miriad.

Methods

changeMapName(*source=None*, *telescope=None*, *species=None*, *fluxUnit=None*, *resolution=None*, *comments=None*, *dataFormat=None*, *prefix=None*)
This function can be used to change the names of the maps and make a copy of the file to the new name and/or location.

flux_conversion(*x=None*, *major=None*, *minor=None*, *nu_or_lambda='nu'*, *direction=None*)
Calulates conversion between K.km/s and Jy/beam and vise versa.

x: float [GHz] Wavelenght/frequency. Defaults to the frequency of the loaded map, i.e. self.frequency

major: float Major Axis Beam (arcsec). Default None, i.e. using self.resolution.

minor: float Minor Axis Beam(arcsec). Default None, i.e. using self.resolution.

nu_or_lambda: string Choose type of x: frequency = `'nu'` or wavelenght = `'lambda'`.

direction: string choose conversion direction `'kelvin_to_jansky'` means Kelvin to Jansky; `'jansky_to_kelvin'` Jansky to Kelvin.

Notes

Please note that if self.frequency and self.resolution are correctly set, this functions does not need any input.

returnName(*source=None*, *telescope=None*, *species=None*, *fluxUnit=None*, *resolution=None*, *comments=None*, *dataFormat=None*, *prefix=None*)
Returns the Name corresponding to the Name convention. Single keywords can be changed.

2.2 Fits

class astrolyze.maps.fits.**FitsMap**(*mapName*, *nameConvention=True*)
Fits Map manipulation making extensive use of the pyfits package.

Methods

change_unit(*final_unit*, *frequency=None*)
Changes the unit of a map in an automated way.

Parameters **final_unit: string** :

The unit to change the map to. Possible are:

1. Jy/beam: `"JyB"`, `"Jy/Beam"`
2. Jy/pixel: `"JyP"`, `"JyPix"`, `"JyPix"`
3. MJy/sterad: `"MJyPsr"`, `"MJy/sr"`
4. Temperature: `"Tmb"`, `"T"`, `"Kkms"`

frequency: float :

Can be used if self.frequency is NaN. The frequency (in GHz)is needed for conversions between temperature and Jansky/Erg scale. Other conversions don't need it.

Notes

Warning: This function is still in developement and not all conversions may work properly.

cut_map (*x1y1*, *x2y2*, *pix_or_coord='coord'*)

Cutting an rectangle out of a map. Giving the corners in coordinates or in pixels.

Parameters **x1y1: list** :

The upper right corner of the rectangle to cut out. Either in:

- pixel coordinated [x1, y1]

Or:

- equatorial coordinates ['RA','DEC']

x2y2: list :

The lower left corner in the same format as x1y1.

pix_or_coord: string :

Either `"pix"` or `"coord"` choosing what **x1y1** and **x2y2** represents.

Notes

This procedure cuts only rectangles paralell to the sides of the map.

Warning: Old function. Functionality not guaranteed. maybe not really useful. Test or remove.

gauss_factor (*beamConv*, *beamOrig=None*, *dx1=None*, *dy1=None*)

Caluclates the scaling factor to be applied after convolving a map in Jy/beam with a gaussian to get fluxes in Jy/beam again.

This function is a copy of the FORTRAN gaufac function from the Miriad package, which determine the Gaussian parameters resulting from convolving two gaussians. This function yields the same result as the MIRIAD gaufac function.

Parameters **beamConv: list** :

A list of the [major axis, minor axis, position_angle] of the gaussion used for convolution.

beamOrig: :

Same format as beamConv but giving the parameters of the original beam of the map. As Default the self.resolution list is used.

dx1, dy1: floats :

Being the pixel size in both dimensions of the map. By default the `CDELT1` and `CDELT2` keywords from the fits header are used.

Returns **fac:** :

Factor for the output Units.

amp: :

Amplitude of resultant gaussian.

bmaj, bmin: :

Major and minor axes of resultant gaussian.

bpa: :

Position angle of the resulting gaussian.

get_beam_size()
Calulates the Beamsize in m^2 if distance is given if it is not given the PixelSize is in sterradian.

get_pixel_size()
Calulates the Area of a pixel in m^2 if distance is given if it is not given the PixelSize is in sterradian.

pix2sky(*pixel*, *degrees_or_equatorial='degrees'*)
Calculates the Coordinates of a given Pixel.

Parameters **pixel: list** :

Pixel of the map; [x, y].

degrees_or_equatorial: string :

Either `"degrees"` or `"equatorial"`. Choosing the Format of the coordintates to be returnes. Defaults to `"degrees"`.

Returns **coordinate: list** :

The coordinates corresponding to pixel. Either in Degrees or in Equatorial coordinates, depending on the parameter *degrees_or_equatorial*.

read_aperture(*position*, *apertureSize=0*, *backgroundSize=0*, *output=False*, *annotation=False*, *newAnnotation=False*)
Extract the flux inside an aperture.

This function can be used to read the flux in the area of a circular/gaussian beam solid angle, as well as to correct for the background flux.

Parameters **position: list** :

The position in RA,DEC where the aperture is to be applied. The Format has to be either:

- ['RA','DEC'] with strings representing equatorial coordinates, e.g. ['01:34:32.8', '+30:47:00.6'].

or:

- [RA, DEC] where RA and DEC being the coordinates in Grad.

apertureSize: float [arcsec] :

The diameter of the aperture to be applied.

backgroundSize: float [arcsec] :

The Size of the Anulli in which the background is to be estimated. The number to be given here correspond to the diameter of the circle in [arcsec] descibing the outer border of the annuli, measured from the position given in position. Thus, the background is measurd in the ring described by apertureSize and backgroundSize. Default is 0 and thus **no background substaction** is applied.

output: True or False :

If True the function reports the calculated values during execution.

annotion: logical :

If True an kvis annotation file `"apertures.ann"` containing the aperture used to integrate the flux. Default is False, i.e. not to create the aperture.

newAnnotation: logical :

If True `"apertures.ann"` is overwritten. If False an old `"apertures.ann"` is used to append the new apertures. If it not exists a new one is created. The latter is the default.

Returns **List: [Sum, Mean, Number of pixels]** :

`read_flux`(*position*)

Returns the value of the pixel that corresponds to the given positions of RA, DEC (J2000) in units of equatorial coordinates or degrees.

Parameters **position: list** :

The position in RA,DEC where the aperture is to be applied. The Format has to be either:

- ['RA','DEC'] with strings representing equatorial coordinates, e.g. ['01:34:32.8', '+30:47:00.6'].

or:

- [RA, DEC] where RA and DEC being the coordinates in Grad.

Returns **flux: float** :

The flux at the given position.

See Also:

`sky2pix`, `astFunc.equatorial_to_degrees`, `wcs.wcs_sky2pix`

`sky2pix`(*coordinate*)

Calculates the Pixel corresponding to a given coordinate.

Parameters **coordinate: list** :

Either ['RA','DEC'] in equatorial coordinates or [RA, DEC] in GRAD.

Returns **pixel: List** :

[x, y]; the pixel coordinates of the map.

`strip`(*coords*, *radial=None*, *centerCoord=None*, *distance=None*)

Extracts a linear cut trough a map between two coordinates.

Distance in kpc.

Notes

`toGildas`(*prefix=None*)

Changes the current map to the Gildas Format.

The function takes changes to the map_name variables made outside of functions into account via `astrolyze.maps.main.Map.returnName()` into account.

Parameters **prefix: string or None** :

Path to location where the new gildas file will be stored. The default is None which defaults to the current self.prefix.

Examples

To continue working with the gildas map use:

```
>>> map = map.
```

To only store the current map in the gildas format and go on working with the fits file use:

```
>>> map.
```

Here map is an Instance of the FitsMap class.

toMiriad(*prefix=None*)
Changes the current map to the Miriad Format.

The function takes changes to the map_name variables made outside of functions into account via `maps.main.Map.returnName()` into account.

Parameters **prefix: string or None** :

Path to location where the new gildas file will be stored. The default is None which defaults to the current self.prefix.

Examples

This function works like `maps.mapClassFits.FitsMap.toGildas()` and the same Examples apply.

update_file(*backup=False*)
Writing changes to the self.data and/or self.header to the current file.

Parameters **backup: True or False** :

If True a copy of the original file is created having the extension `"_old"` after the file endind, i.e. some_name.fits -> some_name. fits_old.

Returns **FitsMap: Instance** :

Notes

If all variables that define the map name () are unchanged the current file is overwritten else

2.3 Gildas

class `astrolyze.maps.gildas.`**GildasMap**(*mapName*, *nameConvention=True*)
Wrapping GILDAS functions to use them inline with Python.

Methods

goRot (*angle*, *prefix=None*)
Wrapper to the GREG go rot command, which rotates maps around their central coordinate stored in the header.

Parameters **angle: float [deg]** :

Rotation angle.

prefix: string :

The path where the output is to be stored if different from the current prefix of the map.

Returns **GildasMap Object: Instance for the reprojected map.** :

Examples

```
>>> map.       45
```

To change the central coordinate first use `maps.gildas.GildasMap.reproject()` e.g.:

```
>>> map = map.                = 'new_RA_string' 'new_DEC_string'
>>> map.       45
```

lmv (*fileout=None*, *prefix=None*)
Wrapper to the lmv command of Class to extract spectra from a spectral cube.

Parameters **fileout: string** :

The name of the class file to write the spectra to. Defaults to the mapname with .30m ending.

prefix: string :

The path were the class file will be stores. Defaults to the current path.

Notes

Tested and working.

mask (*polygon*, *prefix=None*)
Wrapper to the GREG task mask:

Parameters **polygon: string** :

path to a GILDAS polygon file with ending `".pol"`

prefix: string :

The path where the output is to be stored if different from the current prefix of the map.

Returns **mapObject: The masked map object.** :

Examples

```
>>> map.     'poly/sourceA.pol'
```

moments (*velo_range=[0, 0], threshold=0, smooth='YES', prefix=None, comment=None*)
Wraps the GREG task moments creating the first three moments of the map.

Parameters **velo_range: list** :

Velocity range for the integration.

threshold: float :

Value under which pixels are blanked.

smooth: string :

One of Either `"NO"` or `"YES"`. Controls if the map is smoothed before applying the cut threshold. Getting rid of noise peaks over the threshold. Defaults to `'YES'`

prefix: string :

The path where the output is to be stored if different from the current prefix of the map.

comment: string :

Optional comments to be added to the new map name.

Returns **mean** : MapObject

The zeroth moment, i.e. the integrated intensity, is returned as a GildasMap object.

quick_preview (*save=False*, *filename=None*)
Plotting the map and optionally save the figure.

Parameters **save: True or False** :

Choose wether or nor to save the figure.

filename: string :

The filename to for the saved plot. If None defaults to `'quick_preview.eps'`.

regrid_to_arcsec (*value*)
Regrids the pixel size of the map to a multiple of arcseconds.

Parameters **value: float** :

The new pixel size in arcsecs.

Notes

> **Warning:** Old function no guarantee of functionality. Test or remove!

reproject (*template=None*, *coord=None*, *prefix=None*, *keep_pixsize=False*)
Wraps the GREG task reproject. Either use template *or* coord.

Parameters **template: string** :

Full path to a map in GDF Format whose central coordinate and pixel size will serve as a template.

coord: list :

List of coordinate strings in RA DEC (J2000) that will become the new centre of the map.

prefix: string :

The path where the output is to be stored if different from the current prefix of the map. If None the current self.prefix of the GildasMap instance is used.

keep_pixsize: bool :

If False reproject guesses the new pixel sizes after reprojection these are normally smaller than the original ones. If True the old pixel sizes are enforced.

Returns **GildasMap Object: Instance for the reprojected map.** :

Raises **SystemExit** :

If both **template** and **coord** are not `None`.

ValueError :

If keep_pixsize is not a boolean.

References

For more information on the Gildas task see: .. [R1] www.iram.fr/GILDAS/

[R1]

Examples

```
>>> map.                    =  '1:34:32.8'  '30:47:00.6'
>>> map.                       = 'M33_SPIRE_250_JyB_18.1.gdf'
```

smooth (*new_resolution*, *old_resolution=None*, *prefix=None*)

Wrapper to the GREG task gauss_smooth.

Parameters **new_resolution: float or list** :

The resulting resolution after the smoothing. It can be:

1. a float: i.e. the final major and minor beamsize. The position angle will default to 0.
2. a list with two floats: [major_axis, minor_axis]. The position angle defaults to 0.
3. a list with three floats: [major_axis, minor_axis, position_angle].

old_resolution: float or list :

Same format as new_resolution. Defaults to self.resolution of the map instance.

prefix: string :

The path where the output is to be stored if different from the current prefix of the map.

Notes

> **Warning:** The gauss_smooth Task from GILDAS only gives correct output units when the map is on a temperature or "per pixel" scale. **Maps in Jy/Beam won't be in Jy/Beam after smoothing.**

spectrum(*coordinates*, *fileout=None*, *prefix=None*)
Wrapper to the spectrum command from greg that extracts a spectrum from a cube at a given positions.

Parameters **coordinates: list** :

A list with the coordinates in floats in units of Degrees, or in string for equatorial coordinates.

fileout: string :

The name of the table where the spectrum will be stored. Default is the same name as the map with `".tab"` as ending.

prefix: :

The path to the folder where the newly created file will be stored. Defaults to the prefix currently stored in self.prefix.

Notes

Tested and working.

Examples

```
>>> from astrolyze.maps import *
>>> map =            'M33_PdBI_12co10_Tmb_22.0_2kms.gdf'
>>>            =  '1:34:7.00'  '+30:47:52.00'
>>> map.
Creates M33_PdBI_12co10_Tmb_22.0_2kms.tab in the present folder.
```

toFits()
Converts the actual map to a Fits map.

Examples

With:

```
>>> map =           'M33_MIPS_24mum_JyB_5.gdf'
>>> map = map.
```

it is possible to continue working with the fits map, using the `maps.fits.FitsMap` class.

toMiriad()
Converts the actual map to a Miriad map.

Examples

With:

```
>>> map =          'M33_MIPS_24mum_JyB_5.gdf'
>>> map = map.
```

it is possible to continue working with the Miriad map, using `maps.miriad.MiriadMap` class.

2.4 Miriad

class `astrolyze.maps.miriad.`**`MiriadMap`**(*mapName*, *nameConvention=True*)

Methods

`moment`(*iN=''*, *region=''*, *out=''*, *mom='0'*, *axis=''*, *clip=''*, *rngmsk=''*, *raw=''*)

Wrap around MIRIADs moment task. keywords are as in miriad. By default (-> if you give no arguments to the function) it creates the zeroth moment of the map

CHAPTER

THREE

SPECTRA

3.1 Class

class astrolyze.spectra.class_.**ClassSpectra**(*mapName*, *nameConvention=True*)
Provides some usefull automated functions to work on Class Spectra in an convenient way.

Methods

get_average_spectrum(*prefix=None*)
Averages all spectra in a cube.

Parameters **prefix: string** :

The new path where the averaged spectrum will be stored.

Notes

So far no selection is made so the files of the input file have to be consistent.

get_spectra_from_cube(*coordinate*, *angle=0*, *prefix=None*, *accuracy=2*)
Extracts one spectra at the position of coordinates from a spectral cube.

Parameters **coordinate: list** :

Equatorial coordinates, e.g. [‘1:34:7.00’, ‘+30:47:52.00’]

angle: float :

If the cube was rotated before the angle has to be specified to calculate the correct offset.

prefix: string :

The new path where the averaged spectrum will be stored.

accuracy: float :

The tolerance in arcsec to find a spectra corresponding to the given coordinate.

Returns **30m file** :

With the first spectrum in the list of spectra within the accuracy range with the given coordinate.

set_defaults()
Reset all selection criteria.

CHAPTER

FOUR

MAPTOOLS

astrolyze.maptools.maptools.**copy_structure**(*list*, *old_prefix*, *new_prefix*)

Copies a folder structure from old_prefix to new_prefix. To assure all folders exists before working with or copying data.

Parameters **list: list** :

A list containing the relative or absolute paths to files.

old_prefix: string :

The old path to the folder structure that has to be copied. Has to actually appear in all the strings in list.

new_prefix: string :

The path to where the folder structure is to be copied.

Notes

This is usefull if one is working on many files stored in several sub-folders retrieved using `get_list()`.

Examples

Say the folder structure is like this

```
>>>    ../        /
co10/
co21/
>>>    ../       /    /
map1/
map2/
>>>    ../        /    /
map1/
map2/
```

This can be copied to say ../even_more_modified by doing as follows:

```
>>> from astrolyze.maptools import *
>>> list =         .         ../
>>>         .               list            ='../modified'
>>>                                       ='../even_more_modified'
```

astrolyze.maptools.maptools.**get_list**(*folder*, *data_format=None*, *depth=False*)
Loading a list of files in all subfolders.

Parameters **folder: string** :

The path to the folder that has to be parsed.

data_format: string :

Search for specific files containing the string, e.g. '.fits'

depth: integer :

The steps into the subfolder system. Defaults to maximum depth.

Returns **final_list: array** :

Array with the string to the files in the folder and sub folders.

folder_list: :

Array with the strings to the folders. Only if depth is set.

astrolyze.maptools.maptools.**unifyMaps**(*list*, *tinMap*, *folder='reg'*)
changes the dimensions and pixel sizes off all maps to that of a template map.

astrolyze.maptools.maptools.**unifyResolution**(*liste*, *resolution=False*, *folder='smooth'*, *scaling=''*)
Approved.

astrolyze.maptools.maptools.**unifyUnits**(*list*, *folder='units'*)
NOT READY YET!

CHAPTER

FIVE

SEDS

class astrolyze.sed.sed.**Sed**(*source_name*, *coordinate*, *flux_array*)

This class handles a single sed. Fitting plotting and so on. It should contain the data, fitting,

Methods

grey_body_fit()

" Fitting a multi componenet grey body to the input data in flux_array.

See Also:

py:func:*astrolyze.functions.astro_functions.grey_body_fit*

plot_sed(*axes=<matplotlib.axes.AxesSubplot object at 0x4cb7ad0>*, *nu_or_lambda='nu'*, *color='black'*, *linewidth=0.5*, *xRange='normal'*)

Plot a multi component greybody model.

nu_or_lambda: plot against frequency 'nu' or wavelenght 'lambda'

kappa: The kappa to use. 'easy' or 'Kruegel'. Please refer to functions.astroFunctions.greyBody() for more information.

xRange: PLEASE ADD DESCRIPTION linewidth: float

The linewidth of the plotted lines. Default to 0.5.

color: matplotlib conform color the color of the plotted lines. Default to 'black'.

class astrolyze.sed.sed.**SedStack**(*folder*, *data_format='.fits'*, *filein=None*)

Reads in the SEDs from a stack of maps at given coordinates and creates a stack of Sed objects.

Methods

load_coordinates(*filein*)

Loads the coordinated where the SEDs should be evaluated from either a file or a list. Both are not possible.

Parameters **filein: string** :

Path to file that cotains the coordinates format has to be:

source_name RA DEC

RA, DEC has to be for epoch J2000 in Equatorial coordinates, see below for examples of the syntax.

Returns **self.coordinates: list** :

Format:: [[source_name_1, RA_1, DEC_1] , ... , [source_name_N, RA_N, DEC_N]]

Examples

The format of the coordinates given in the file must be in Equatorial: >>> equatorial_coordinates = ['02:23:21', '-02:23:21']

load_file_list (*folder*, *data_format*)

Intitializes the variable self.file_list containing a list of the maps that are to be used to create the SED.

Parameters **folder: string** :

May contain sub-folders that are all taken into account.

data_format: string :

Supported `'.fits'` (Default) and `'.gdf'`. Maybe extended later.

Returns **self.file_list: list** :

Paths to the file in potential subfolders in under the folder.

self.maps: list :

A list with FitsMap objects. If Gildas maps are loaded they are converted into fits maps.

See Also:

`maps.main`, `maps.fits`, `maps.gildas`

CHAPTER

SIX

LTE

Functions to calculate LTE column densities.

Add Documentation.

astrolyze.lte.lte.**calc_N**(*molecule*, *excitation_temperature*, *J*, *W*)
Calculates the column density for a molecule. !!! LOOK into the remaining Code and merge!!!

astrolyze.lte.lte.**calc_excitation_temperature**(*Tb*, *nu*)
Calculation of the excitation temperature of an optically thick 12CO line under the assumption of LTE.

Parameters **Tb:** :

astrolyze.lte.lte.**calc_jnu**(*nu*, *T*)
Calculates J_ν needed for lte_column_density. !!! CGS Units!!! Extend Documentations!!!

Parameters **nu: float** :

Frequency

T: float :

Temperature

astrolyze.lte.lte.**lte_column_density**(*nu*, *Tmb*, *excitation_temperature*, *J*, *Z*, *mu*)
This function calculates the Column densities of linear molecules

Units are all to be given in cgs Z is the array of partition function values for the corresponding temperatures in T these are the log values of Z

Notes

Extend documentation!!!!

This script generates a dictionary storing the information of molecular transitions. r

class astrolyze.lte.molecule_parameter.**Molecule**(*nu*, *Q*, *T*, *Eu*, *Aul*, *gu*, *mu*, *name=''*)
A class that holds attributes that define the characteristics of an individual tansition of a molecules.

The `astrolyze.functions.astro_functions.calc_N()` routine depends on this

Parameters **nu: float** :

Q: list :

The partition function of the molecule evaluated at discreet temperatures. Given in T.

T: list :

The temperatures at which the partition functions was evaluated. The value of Q used finally is interpolated to the excitation_temperature give.

Eu: float :

Energy of the upper state of the transition.

Aul: float :

The Einstein coefficient. For the transition of upper-to-lower state.

gu: float :

ADD DESCRIPTION.

mu: float :

ADD DESCRIPTION.

name = string :

The name of the molecule.

CHAPTER

SEVEN

FUNCTIONS

7.1 astroFunctions

astrolyze.functions.astro_functions.**LTIR**(*p2*, *kappa='Kruegel'*, *xmin=3.0*, *xmax=1100.0*, *beamConv=True*, *distance=847000.0*, *unit='JyB'*)

Integration of a multi-component greybody model.

Parameters **p2: list** :

The parameters defining the multi-component greybody model. Same format as p in `astrolyze.functions.astroFunctions.multi_component_grey_body()`

kappa: string :

The dust extinction coefficient used to describe the greybodies. See: py:func:*grey_body*

xmin, xmax: float :

The integration range in units of micron. Defaults to 3 – 110 micron. The definition of LTIR from [DA]

beamConv: True or False :

For units in Lsun the code is not well written. Hardcoded conversion between an 28" and 40" beam. !! CHANGE !!

unit: string :

If `'Lsun'` the returned integrated flux is in units of solar luminosities (erg s^-1). For this a distance is needed. If `'JyB'` the units are Jy/beam; distance is not used.

Notes

Needs some work to be generally usable. For units in Jy/beam the code seems to be safe.

References

[DA]

astrolyze.functions.astro_functions.**anti_line**(*p*, *y*)

Inverse of a line returning the x value corresponding to a y value, i.e. $x = y/m - b$.

Parameters **p: list** :

Contains the slope and the y-axis intersection of the line [m, b].

Returns **y: value of x corresponding to y.** :

`astrolyze.functions.astro_functions.`**`black_body`**(*x*, *T*, *nu_or_lambda='nu'*)

Calculation of the flux density of a black body at a temperature T and a wavelenght/frequency x.

Parameters **x: float or numpy array** :

wavelength [GHz] or frequency [micron]; specify type in nu_or_lambda

T: float [Kelvin] :

Temperature of the black_body

nu_or_lambda: string :

Specify whether x is a frequency ν `'nu'` or a wavelenght λ `'lambda'`; default is `'nu'`.

Returns **Flux density in Jansky: float [Jy]** :

Notes

This functions resembles the following formulas for input in frequency:

$$B_\nu = \frac{2h\nu^3}{c^2}(e^{\frac{h\nu}{kT}} - 1)^{-1}$$

and for input in wavelenght:

$$B_\lambda = \frac{2hc^2}{\lambda^5}(e^{\frac{h\lambda}{kT}} - 1)^{-1}$$

Both formulas are scaled by 1e26, thus returning the flux in Jansky.

Examples

The function works with linear numpy arrays. Thus the black_body can be evaluated at many points at the same time. Using matplotlib it can also be plotted:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import astrolyze.functions.astro_functions as astFunc
>>>                 =    .       1e4  1e7  1e4
>>>               = 6000
>>>               = 12000  # Kelvin
>>>             =         .
>>>             =         .
>>>         =     .
>>>       =         .            111
>>>     =      .                                          ='T = 6000 K'
>>>     =      .                                          ='T = 12000 K'
>>>     =      .
>>>     .        'black_body.eps'
```

`astrolyze.functions.astro_functions.`**`calc_offset`**(*central_coordinate*, *offset_coordinate*, *angle=0*, *output_unit='arcsec'*)

Calculates the offset between two coordinates.

Parameters **central_coordinate: list** :

The reference coordinate in degrees or equatorial.

offset_coordinate: list :

The second coordinate, the offset will be with rescpect to central_coordinate.

angle: float :

The angle in degrees, allowing rotated systems.

Returns **rotated_offset: list** :

The offsets, rotated only if angle given.

Notes

This functions includes a correction of the RA offset with declination:

astrolyze.functions.astro_functions.**degrees_to_equatorial**(*degrees*)
Converst RA, DEC coordinates in degrees to equatorial notation.

Parameters **degrees: list** :

The coordinates in degress in the format of: [23.4825, 30.717222]

Returns **equatorial: list** :

The coordinates in equatorial notation, e.g. corresponding [1:33:55.80, +30:43:2.00'].

astrolyze.functions.astro_functions.**equatorial_to_degrees**(*equatorial*)
Converst RA, DEC coordinates in equatorial notation to degrees.

Parameters **equatorial: list** :

The coordinates in degress in equatorial notation, e.g. [1:33:55.80, +30:43:2.00']

Returns **degrees: list** :

The coordinates in degreees, e.g. [23.4825, 30.717222].

astrolyze.functions.astro_functions.**gauss1D**(*x*, *fwhm*, *offset=0*, *amplitude=1*)
Calulcates 1D Gaussian.

Parameters **x: float or numpy.ndarray** :

the x-axis value/values where the gaussian is to be caluclated.

fwhm: float :

The width of the gaussian.

offset: :

The offset in x direction from 0. Default is 0.

amplitude: :

The height of the gaussian. Default is 1.

Returns **gauss: float or np.ndarray** :

The y value for the specified gaussian distribution evaluated at x.

Notes

The function used to describe the gaussian is:

$$f = \frac{1}{fwhm * sqrt(2 * \pi)} * e^{-1/2(\frac{x-x0}{fwhm})^2}$$

astrolyze.functions.astro_functions.**gauss2D**(*x*, *y*, *major*, *minor*, *pa=0*, *xOffset=0*, *yOffset=0*, *amplitude=1*)

Calulcates a 2D Gaussian at position x y.

Parameters **x: float or numpy.ndarray** :

the x-axis value/values where the gaussian is to be caluclated.

y: float or numpy.ndarray :

the y-axis value/values where the gaussian is to be caluclated.

major, minor: float :

The fwhm of the gaussian in x and y direction.

pa: float :

The position angle of the gaussian in degrees. Default is 0.

xOffset, yOffset: :

The offset in x and y direction from 0. Default is 0.

amplitude: :

The height of the gaussian. Default is 1.

Returns **gauss: float or np.ndarray** :

The y value for the specified gaussian distribution evaluated at x.

Notes

The function used to describe the gaussian is :

$$f = (amplitude * exp(-1(a * (x - xOffset)^2 + 2 * b * (x - xOffset) * (y - yOffset) + c * (y - yOffset)^2)))$$

where:

$$a = cos(pa) * *2/(2 * major * *2) + sin(pa) * *2/(2 * minor * *2)$$
$$b = (-1 * sin(2 * pa)/(4 * major * *2)) + (sin(2 * pa)/(4 * minor * *2))$$
$$c = sin(pa) * *2/(2 * major * *2) + cos(pa) * *2/(2 * minor * *2)$$

astrolyze.functions.astro_functions.**generate_monte_carlo_data_line**(*data*, *errors*)

This function makes a Monte Carlo Simulation of a data Set of measurements it uses the random.gauss() function to generate a data point from a gauss distribution, that has a mean equal to the measurement and its standard deviation corresonding to the error of the measurement.

Parameters **data: list** :

A list of original measurements.

errors: list :

A list of the corresponding errors.

Returns **newData: array in same format as data.** :

The monte carlo simulated measurement.

See Also:

`random.gauss`

`astrolyze.functions.astro_functions.`**`generate_monte_carlo_data_sed`**(*data*)

MonteCarlo Simulation of a set of flux measurements, assuming that the measurement data follows a gauss distribution.

This function makes use of the `random.gauss()` function to generate a data point from a gauss distribution, that has a mean equal to the Flux measurement and a standard deviation correponding to the error of the measurement.

Parameters **data: array** :

Same format as in grey_body_fit function: data= [[x1, x2, x3, ...][y1, y2, y3, ...][z1, z2, z3, ...]]

with x = wavelenght/frequency, y = flux, z = error on flux.

Returns **newData: array in same format as data.** :

The monte carlo simulated measurement.

See Also:

`random.gauss`

`astrolyze.functions.astro_functions.`**`grey_body`**(*p*, *x*, *nu_or_lambda='nu'*, *kappa='Kruegel'*, *distance=840000.0*)

Calculation of the flux density in Jansky of a grey_body under assumption of optically thin emission.

Please see Notes below for an detailed desciption assumptions and equations used.

Parameters **p: list** :

List of the parameters defining a grey_body, being Temperature [K], column density or mass (dependend on the kappa used) and the grey_body slope index beta, respectively (refer to notes for more information):

p = [T, N, beta]

x: float or numpy array :

Wavelength [GHz] or frequency [micron]; specify type in nu_or_lambda

kappa: string :

Chooses the dust extinction coeficient to use:

- `"easy"` -> kappa = nu^beta; tau = N * kappa
- `"Kruegel"` -> kappa = 0.04*(nu/250Ghz)^beta; tau = M/D^2 * kappa

Please refer to Notes below, for further explanation.

distance: float :

The distance to the source that is to be modeled if kappa `"Kruegel"` is used.

Other Parameters **nu_or_lambda: string** :

Specify whether x is a frequency ν `'nu'` or a wavelenght λ `'lambda'`; default is `'nu'`. if lambda the input converted to a frequency in [GHz].

Notes

The general equation for a grey_body is:

$$S(x,\tau) = (black_body(x,T)) * [1 - e^{\tau}]\Omega$$

describing the flux coming from an solid angle Ω and where τ is:

$$\tau_\nu = \frac{\kappa_d(\nu) * M_{dust}}{D^2\Omega}.$$

Here we assume optically thin emission and a source filling factor of unity. This simplyfies the equation of the grey_body to:

$$S(x,\tau) = \tau * (black_body(x,T))$$

This script supports two versions of the dust extinction coefficient.:: A simple version without a lot of physics put into, kappa = `'easy'` which defaults to the following grey_body equation:

$$S(x,\tau) = N * x^{\beta} * black_body(x,T),$$

with N being a column density scaling factor.

The second version, kappa = `'Kruegel'` uses the dust extinction coefficient reported in [KS] which renders the used equation to:

$$\kappa = 0.04 * (\frac{x\,[GHz]}{250\,GHz})^{\beta}$$
$$S_\nu = M[kg]/D^2[m^2] * \kappa * black_body(x,T).$$

References

[KS]

Examples

The same examples as for `black_body()` apply.

`astrolyze.functions.astro_functions.`**`grey_body_fit`**(*data*, *start_parameter*, *nu_or_lambda='nu'*, *fit_beta=False*, *fix_temperature=False*, *rawChiSq=None*, *kappa='Kruegel'*, *residuals=False*, *iterations=1000000000.0*)

This function fits a multi component grey body model to an observed SED for the optical thin case.

Parameters **data: array** :

The obseved data. Array of shape(3, x) first row has to be the X values (Frequency in [GHz]) of the measurements, second row the Y values (Flux [Jy]), and the third row the Z values the errors on the fluxes i.e.:

data = array([[X1, X2, X3, ...], [Y1, Y2, Y3,...], [Z1, Z2, Z3, ...]])

start_parameter: array :

Array of a first guess of the parameters of the grey_body components. The number of components is arbitrary

start_parameter = [[T1, T2, T3,...], [N1, N2, N3, ...], beta]

fit_beta: True or False :

If True Beta is allowed to vary. Default is False.

fix_temperature: True or False :

If True the Temperature is fixed allowed to vary.

rawChiSq: :

if None the function gives the reduced chisq Value. If True the function gives chisq without dividing it by the dof

Returns p2: list :

The final grey_body parameters that reduce the least squares for the given dataset.

chisq/rawChiSq: :

chisq is reduced chisq with degrees of freedom: dof= #dataPoints-#freeFitParameters-1

Other Parameters nu_or_lambda: string :

Specify whether x is a frequency ν `'nu'` or a wavelenght λ `'lambda'`; default is `'nu'`.:: **Don't** use `'lambda'` as this part of the `grey_body()` is not up-to-date.

See Also:

`scipy.optimize.leastsq` This function is used to perform the least squares

`fit.`, `multi_component_grey_body`, `grey_body`, `black_body`, `be`

Notes

A one component fit has four free parameters if beta is allowed to vary or three if beta is fixed (one more than parameters to fit). Each additional component adds two more free paramters to fit. Assure that:

number of data points > number of free parameters.

`astrolyze.functions.astro_functions.`**`grey_body_monte_carlo`**(*p*, *data*, *iterations*)

Function to evaluate the errors in the parameters fitted with the grey_body_fit function.

It uses Monte Carlo Simulated data (from `generate_monte_carlo_data_sed()`) and performs a fit to this new data giving back the results of the fit parameters.

Parameters p: list :

The parameters defining the multi component grey_body model to be fitted. Same format as p in `multi_component_grey_body()`

data: array :

The actual measured data of the SED, same format as for `grey_body_fitFunction()`

iterations: int :

Number of times new data is generated and fitted.

Returns **string:** :

Containing the mean, standard deviation of the fit parameters, ready to print out.

betaTlist: List of all fit results. Name misleading since it may not :

include the beta.

astrolyze.functions.astro_functions.**line**(*p*, *x*)

Line *y = m*x + b* equation. Returns y value at point x.

Parameters **p: list** :

Contains the slope and the y-axis intersection of the line [m, b].

Returns **y: value of y corresponding to x.** :

astrolyze.functions.astro_functions.**line_fit**(*p*, *x*, *y*, *y_error*, *x_error=False*, *iterations=10000*)

Linear Fit to data, taking either errors in y or both in x and y into account.

Parameters **p: list** :

Containg slope (m) and y-axis intersection (b) p=[m, b]. Same as in `line()` and `antiline()`.

x: float or list :

x measurements. Data.

y: float or list :

y measurements. Data.

y_error: float or list :

The y measurment errors.

x_error: float or list :

The x measurment errors. If unset only errors in y are taken into account.

astrolyze.functions.astro_functions.**line_monte_carlo**(*p*, *x*, *y*, *x_error*, *y_error*, *iterations*, *fitIterations=1000000000.0*)

Gererate an estimate of the errors of the fitted parameters determined by the `line_fit()` function.

Parameters **p: list** :

Containg slope (m) and y-axis intersection (b) p=[m, b]. Same as in `line()` and `antiline()`.

x: float or list :

x measurements. Data.

y: float or list :

y measurements. Data.

y_error: float or list :

The y measurment errors.

x_error: float or list :

The x measurment errors. If unset only errors in y are taken into account.

Returns **string: A string containing the results.** :

BList: A list containing the fittet y-Axis intersections. :

MList: A list containing the fitted slopes. :

chisqList: A list with the chisq values. :

resultArray: Array with the mean and the standard deviations of :

slopes and y-axis intersections, i.e. [mean(M), std(M), mean(B), std(B)]

See Also:

`grey_body_fit`, `generate_monte_carlo_data_line`

`astrolyze.functions.astro_functions.`**`linear_error_function`**(*p*, *x*, *y*, *y_error*, *x_error*)

Error function, i.e. residual from the measured value, which has to be minimised in the least square fit taking X and Y Error into account.

Parameters **p: list** :

Same as in `line()` and `anti_line()`.

x: float or list :

x measurements. Data.

y: float or list :

y measurements. Data.

x_error: float or list :

The x measurment errors.

y_error: float or list :

The y measurment errors.

`astrolyze.functions.astro_functions.`**`linear_fit`**(*x*, *y*, *x_error*, *y_error*)

This function resembled the analytical solution following chaper 8 from [TA]/

Parameters **x: float or list** :

x measurements. Data.

y: float or list :

y measurements. Data.

y_error: float or list :

The y measurment errors.

x_error: float or list :

The x measurment errors. If unset only errors in y are taken into account.

Notes

Without errors the following holds for

y=A+BX

A = sum(x^2)*sum(y)-sum(x)*sum(x*y)/Delta B = N*sum(x*y)-sum(x)*sum(y)/Delta Delta = N*sum(x^2)-(sum(x))**2

This has to be checked.

References

[TA]

astrolyze.functions.astro_functions.**multi_component_grey_body**(*pMulti*, *x*, *nu_or_lambda='nu'*, *kappa='Kruegel'*)

Combines multiple grey_body functions and returns the flux density in Jansky for the input frequency/wavelenght.

pMulti: nested lists Similar to p from `functions.astroFunctions.grey_body()` but the three entries are lists, i.e.:: pMulti = [[T1, T2, T3, ...Tn], [N1, N2, N3,...Nn], [beta]]

x: float or numpy array frequency [micron] (or wavelenght **Not maintained**, specify type in nu_or_lambda)

Returns **sum(snu): float** :

All dust components summed.

snu: :

A list with the fluxes of the individual components.

See Also:

`black_body`, `grey_body`

Notes

Only one common beta for all components can be used. May be expanded to mutliple betas if needed.

Examples

Same as for black_body, but all returned grey_bodies may be plotted.

astrolyze.functions.astro_functions.**rotation_2d**(*coordinate*, *angle*)

Implementation of the rotation matrix in two dimensions.

Parameters **coordinates: list of floats** :

Coordinates in the unrotated system [x, y].

angle: float :

The rotation angle

Returns **[x_rotated, y_rotated]: list of floats** :

Coordinates in the rotated system.

7.2 units

Constant unit conversions availabe in this module are:

```
#==============> Approved !!! <============================
              = 1e7  # 1W = 1e7 erg/s
              = 1e-7  # 1W = 1e-7 erg/s
              = 1e-26  # 1Jy = 1e-26 W/m2/Hz
              = 1e26  # 1W  = 1 Jy * m2 * Hz
                 = 1e23  # 1erg/s =  1e23 Jy * cm2 * Hz * s
                 = 1e-23  # 1 Jy = 1e-23 erg/s/cm2/Hz
               = 1e19  # 1 Jy = 1e-23 erg/s/cm2/Hz
```

astrolyze.functions.units.**Int2Lum**(*distance_in_pc*, *cm_or_m='cm'*)

Conversion factor to calculate luminosity from intensities by integrating over the sky 4 pi Distance^2.

Parameters **distance_in_pc: float** :

Distance to the source in parsecs.

cm_or_m: string :

Choose wether the out put is in cm^2 = `'cm'` or in m^2 = `'m'`.

Notes :

— :

Approved. :

astrolyze.functions.units.**JyBToErgsB**(*input_flux*, *distance*, *wavelenght*, *invert=False*)

Conversion between Jy/beam and ergs/beam.

Parameters **input_flux: float** :

Flux to be converted in Jy/beam

distance:float :

Distance to the source in parsec.

wavelenght: float :

Wavelenght λ in μm.

r :

astrolyze.functions.units.**JyBToWKpc2**(*input_flux*, *Distance*, *major*, *minor*, *wavelenght*, *invert=False*)

Conversion from JyB to W kpc^-2.

Parameters **input_flux: float** :

Flux to be converted.

distance: float :

Distance to source in parsec.

major: float :

Major Axis Beam (arcsec).

minor: float :

Minor Axis Beam(arcsec).

wavelenght: float :

Wavelenght λ in μm.

invert: True or False :

Changes the direction of conversion.

Returns float: the converted Flux. :

astrolyze.functions.units.**JyBToWM2Kpc2**(*input_flux*, *distance*, *major*, *minor*, *wavelenght*, *invert=False*)

Conversion between Jy/beam and W m^-2 kpc^-2

Parameters input_flux: float :

Flux to be converted.

distance: float :

Distance to source in parsec.

major: float :

Major Axis Beam (arcsec).

minor: float :

Minor Axis Beam(arcsec).

wavelenght: float :

Wavelenght λ in μm

invert: True or False :

Changes the direction of conversion.

Returns float: the converted Flux. :

astrolyze.functions.units.**WmToKkms**(*x*, *resolution=0*, *sterad=False*, *ToKKms=False*, *m2_or_cm2='m'*, *nu_or_lambda='nu'*)

Conversion between W/m2 and K km/s.

Parameters x: float :

wavelenght/frequency [GHZ].

resolution: float :

ToKKms: True or False :

Direction of the conversion.

sterad: True or False :

If False convert from per beam to per sterad.

m2_or_cm2: string :

Choose if conversion to/from W m-2 oder W cm-2. `'m2'` or `'cm2'`.

astrolyze.functions.units.**ergToKkms**(*x*, *toErg=False*, *nu_or_lambda='nu'*)

Conversion between ergs/cm2/s/sr and K km/s.

Parameters x: float :

wavelenght/frequency [GHZ],

toErg: True or False :

True converts the other direction, i.e. from K km/s to ergs/cm2/s/sr.

nu_or_lambda: string :

Choose type of x: frequency = 'nu' or wavelenght = 'lambda'.

Notes

Approved.

astrolyze.functions.units.**jansky_to_kelvin**(*x*, *major*, *minor*, *nu_or_lambda='nu'*)
Conversion from Jy/beam to K.km/s (Tmb).

Parameters **x: float** :

wavelenght/frequency [GHZ],

major: float :

Major Axis Beam (arcsec).

minor: float :

Minor Axis Beam(arcsec).

nu_or_lambda: string :

Choose type of x: frequency = 'nu' or wavelenght = 'lambda'.

Notes

Approved.

astrolyze.functions.units.**kelvin_to_jansky**(*x*, *major*, *minor*, *nu_or_lambda='nu'*)
Conversion from K.km/s (Tmb) and Jy/beam.

Parameters **x: float** :

wavelenght/frequency [GHZ],

major: float :

Major Axis Beam (arcsec),

minor: float :

Minor Axis Beam(arcsec),

nu_or_lambda: string :

Choose type of x: frequency = 'nu' or wavelenght = 'lambda'.

Notes

This function has been compared with the Time estimator from the [GILDAS] package ASTRO and yields the same conversion factors.

References

[GILDAS]

7.3 constants

```
  = 1.3806503e-23 # Boltzman constant [m^2 kg s^-1 K^-1]
    = 2.7 # Cosmic Microwave Background Temperature in [K]

# in CGS
     = 1.3806503e-16 # Boltzman constant [cm^2 g s^-1 K^-1]
     = 6.62606896e-27 # Plancks constant [Js]
    =2.99792458e10  #Speed of light [cm]

 =2.7182818284 # Eulers number

# Distances

             = 3.08568025e16 # parsec in m
                  = 3.08568025e18# parsec in cm

# redundant but maybe used in program parts.
      = 3.085e16# parsec in m
       = 3.08568025e18 # parsec in cm

           = 4.848e-6
            = 1./60./60

                    = 4.254517e10

#Masses

     = 1.9891e30 # [kg]
       = 1.672621637e-27 #[kg]


#Gauss constants
# GaussArea/(height*FWHM)
          = 1.064467

# Luminosities

      = 3.846e26 # [W]
        = 3.846e26*1e7 # erg/s

               = 1.e-18  # change from debye to esu/cm

# angle Conversions

# a: arseconds
# g: grad
# d: degrees
# r: radian

    = 4.848e-6
    = 1./60/60

    = 180./    .
    = 1./4.848e-6

    =     .  /180.
```

```
    = 60*60
```

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

[DA] Dale et al. 2001; ApJ; 549:215-227

[KS] Kruegel, E. & Siebenmorgen, R. 1994, A&A, 288, 929

[TA] “An introduction to the study of uncertainties in physicak measurement” by John R. Taylor.

[GILDAS] www.iram.fr/IRAMFR/GILDAS

PYTHON MODULE INDEX

a

PYTHON MODULE INDEX

a