
Astrolyze Documentation

Release 0.1.0

Christof Buchbender

November 05, 2012

CONTENTS

13. MapClass documentation master file, created by sphinx-quickstart on Tue May 15 17:38:24 2012. You can adapt this file completely to your liking, but it should at least contain the root *toctree* directive.

Astrolyze is a python-package with several functions for reduction and analysis of (mainly) radioastronomical data analysis. It is developed over the course of my Diploma and PhD thesis. I decided to package the functions because I think that they may be useful to other astronomers. The package is open for collaboration.

INSTALLATION OF ASTROLYZE

astrolyze is only tested on Linux/Ubuntu so far.

1.1 Dependencies

1.1.1 Python

astrolyze depends on the following python packages

```
numpy  
pyfits  
matplotlib  
scipy  
pywcs  
pysqlite2
```

In Ubuntu (and thus most probable also Debian) these dependencies can be installed via:

```
sudo apt-get install python-matplotlib python-pywcs python-scipy python-numpy  
python-pysqlite2 python-pyfits
```

1.1.2 Gildas

To be able to use GILDAS from within astrolyze it is enough to have a working GILDAS installation compiled with the python support. The actual GILDAS version and installation instructions can be found here:

<http://www.iram.fr/IRAMFR/GILDAS/>

1.1.3 Miriad

Also Miriad just has to be installed and working. At the moment only the smooth function of miriad is used in astrolyze but it is worth installing it since it is a common task. Miriad and instructions for installation can be found here:

<ftp://ftp.atnf.csiro.au/pub/software/miriad/INSTALL.html>

1.2 Installation

If you are interested in the package please send me a short E-Mail (buchbend@iram.es) and I'll send it to you. Later it will be available on-line.

So far it astrolyze is only available as a source distribution. To install astrolyze unpack the tar-ball via:

```
> tar -xvzf astrolyze-0.1.0.tar.gz
```

which will unpack into a directory “astrolyze-0.1.0” in the current path. After entering this directory the installation of astrolyze is done via:

```
sudo python setup.py install
```

In the current state the setup does not allow a custom installation path. The files will be installed depending on your system configuration in either of the following locations:

```
/usr/lib/pythonX.Y/site-packages
/usr/local/lib/pythonX.Y/site-packages
```

Here X and Y are the major and minor number your python installations.

1.3 Configuration of the (optional) Database

The setup.py script will generate `parameter.db` sqlite database containing information about the maps and files opened in astrolyzed. From the keyword of the “Naming Convention”.

In the current state astrolyze reads in

- Additional informations of the source
- Frequencies and wavelengths
- Calibration error for specific telescopes

The database has to be populated by the user. This is done via the three text files in the `cfg` folder located in `astrolyze-0.1.0` that contains:

`galaxy_parameter.txt`:

#	Name	MorphType	Distance[pc]	VLSR[km/s]	RA	DEC	PA[degrees]	Inclination
	M33	SA(s)cd	840e3	-179	01:33:51.02	+30:39:36.7	-22.5	56
	NGC3627	SAB(s)b	9.1e6	727	11:20:15.027	+12:59:29.58	173	64

`line_parameter.txt`:

```
# line_name frequency[GHz]
HCOP10      89.188523
HCN10       88.6304156
12CO10      115.271204
13CO10      110.2013543
12CO21      230.542408
HI           1.427583133
HALPHA      457121.40
```

`calibration_error.txt`:

#	telescope	species	cal_error	Reference
	WISE	2.4MUM	0.15	guess
	IRAC	3.6MUM	0.1	guess
	IRAC	4.5MUM	0.1	guess
	WISE	4.6MUM	0.15	guess
	IRAC	5.8MUM	0.1	guess
	IRAC	8MUM	0.1	guess
	WISE	12MUM	0.15	guess
	WISE	22MUM	0.15	guess
	MIPS	24MUM	0.07	Spitzers Observers Manual v.8.0
	MIPS	70MUM	0.07	Spitzers Observers Manual v.8.0
	PACS	100MUM	0.2	Kramer et al. 2010
	PACS	160MUM	0.2	Kramer et al. 2010
	SPIRE	250MUM	0.15	Kramer et al. 2010
	SPIRE	350MUM	0.15	Kramer et al. 2010

SPIRE	500MUM	0.15	Kramer et al. 2010
GISMO	2MM	0.15	guess

The Names of the source telescope and lines have to be exactly how they are used in the map names. However the writing can be an arbitrary mix of upper an lower case characters. Internally Astrolyze converts them to upper case before comparing.

MANUAL

This manual explains how astrolyze can be used to ease reduction, analysis and modification of (radio-)astronomical data and images.

2.1 Motivation

First, before we delve into the details, let me give some examples of what astrolyze is capable of, so that you can decide if it could be of use for you. This is a snapshot of the most powerful functions of astrolyze; a thorough introduction with more features and possibilities follows below.

2.1.1 Inter-operating Fits, Gildas and Miriad

Using the astrolyze package it is possible to use different file formats and different programs from within python seamlessly. Say, for the sake of demonstrations, that you have a Fits image called `M33_30m_12CO10_Tmb_12.fits` and that you want to smooth it in miriad to 40 arcsec resolution, re-project it with Gildas to a new central coordinate and finally convert it back to fits-format:

This is how you would do it with astrolyze:

```
from astrolyze import *
map_ = FitsMap('M33_30m_12CO10_Tmb_12.fits')
map_ = map_.toMiriad()
map_ = map_.smooth(40)
map_ = map_.toGildas()
map_ = map_.reproject(coordinate=['01:34:50.890', '+31:08:28.03'])
map_ = map_.toFits()
```

Please note the special format of the map name. The format follows a certain naming convention that has to be used with astrolyze. The reason for the naming conventions and it's internal logic is explained below.

As a side note I use an underscore for the `map_` variable, because otherwise the python function `map` is overwritten which may lead to problems.

2.1.2 Changing map units

With astrolyze the units of a map can be quickly transformed between common units used in (Radio-) Astronomy (as far as the conversion was implemented already). Take for example again the `M33_30m_12CO10_Tmb_12.fits` map. Following the naming convention this map is in main beam temperature (Tmb). Changing its units to JyB is as easy as:

```
from astrolyze import *
map_ = FitsMap('M33_30m_12CO10_Tmb_12.fits')
map_ = map_.change_unit('JyB')
```

Another side note: It's only that easy when astrolyze is set-up correctly as described in [TODO](#). [Link to Installation](#). Also please check the results for plausibility in case there are faults in the internal conversion algorithms.

2.1.3 Working with stacks of images

The previous examples show the principles how a single map can be treated in astrolyze. However the package also includes a way to work on a stack of images and perform tasks on all of them.

To create a stack, all files that are going to be in the stack have to be located in one folder (with possible sub-folders NOTE: The functionality with sub-folders are is however not thoroughly tested however.) A stack is initialized as follows:

```
from astrolyze import *
example_stack = Stack('path_to_folder')
```

The maps can be a mix of GILDAS, Fits and MIRIAD maps. The Instance of the Stack object (here: example_stack) contains a variable called stack which is a list with Instances of the corresponding maps Objects (GildasMap, FitsMap and MiriadMap).

The stack module provides several tools to unify the stack for further analysis. The maps can be all re-gridded and re-projected to the same central coordinates, pixel-sizes and dimensions as a given template image via:

```
example_stack.unify_dimensions(template='path_to_template_file',
                              folder='path_to_output_folder') .
```

Also the maps can be all smoothed to the same resolution, by default this is the largest resolution found in the stack but can also be given manually:

```
example_stack.unify_resolutions(folder='path_to_output_folder') .
```

Astrolyze includes also some unit conversions that can be used to change all maps to the same resolutions as long as the input and output units are programmed. See (TODO) for more details. For the stack:

```
example_stack.unify_units(folder='path_to_output_folder') ,
```

can be used.

2.1.4 Producing SEDs

Build on-top of the stack module astrolyze also contains an sed module which allows to analyze and plot dust-seds. The SEDs can be read out for an arbitrary number of positions or for a full maps. In the latter case temperature and mass maps will be created.

When all maps have the same resolution and dimension (i.e. pixel size and number) producing temperature maps can be done as follows:

```
from astrolyze import *
sed = SedStack(folder='path_to_input_folder', full_map=True,
               output_folder='path_to_output_folder') .
```

To generate SEDs at given coordinates it is easiest to provide a separate file (e.g. coordinates.txt) with the names and coordinates of the positions to be extracted as follows:

```
source_1    1:34:7.00    +30:47:52.00
source_2    1:33:55.80   +30:43:2.00
source_3    1:33:52.40   +30:39:18.00
.           .           .
.           .           .
.           .           .
```

Then a stack of seds can be created:

```
from astrolyze import *
seds = SedStack(folder='path_to_input_folder', flux_acquisition='pixel')
```

By default the SED is also directly fitted. One can produce a quick preview plot of the SEDs via:

```
for i in seds.sed_stack:
    i.create_figure()
```

2.1.5 Not only images ...

Last but not least astrolyze is also able to work with 30m class spectra from within python based on the same principles used to work with images/maps. The implementation makes extensive use of **pyGildas**. For example if you have a file with the spectra of a cube, e.g. `M33_30m_12CO10_Tmb_21_cube.30m` you can extract the spectra that corresponds closest to a given variable as follows:

```
from astrolyze import *
spectra = ClassSpectra('M33_30m_12CO10_Tmb_21_cube.30m')
coordinate = ['1:34:7.00', '+30:47:52.00']
spectrum = spectra.get_spectrum_from_cube(coordinate)
# To show the spectrum in the Class window
spectrum.quick_view()
```

2.2 astrolyze

I started to develop astrolyze to be able to inter-operate the Programs PyFits, MIRIAD and GILDAS. One reason was that the Gildas task are very cumbersome to script and once scripted, the scripts are not very flexible. Also there are some tasks in GILDAS that are (in my opinion) easier to use than in miriad and vice-versa, due to different sets of features.

However, the real power of astrolyze comes from it's internal tracking of changes to the most important parameters of the maps (or spectral-files), which are stored directly in the file-names, following a naming-convention as well as additional databases, from that more informations can be loaded depending on the file-name-keywords.

2.3 Naming Convention

A Name that follows the 'Naming Convention' is for example: `M33_30m-HERA_CO21-Ta*_12_cube_regrid.fits`

All items **MUST** be separated by an underscore (`_`) and **HAVE** to include at the following properties in the same order:

1. source
2. telescope
3. species (wavelength OR frequency OR line-name)
4. flux unit
5. resolution

When opening a map with astrolyze these items are transferred to python variables of the *Map class* (see below). All additional items separated by underscores are treated as comments. Comments are not transferred to individual internal variables of the map objects but are passed on as a list to the single variable comments.

The last item is followed by the files extension:

- .fits -> FITS
- .gdf, .mean, .velo, .width, .lmv -> GILDAS

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

Maps that are not following this name convention are **not** supported. This is to assure that all parts of the program work, since they strongly depend on the parameters passed on by the file-name, as is explained in this tutorial or the documentation of the functions.

Using the example file-name above, opening this file with astrolyze as follows:

```
from astrolyze import *
map_ = FitsMap('M33_30m_12CO10_Tmb_12_cube_regrid.fits')
```

would generate the following python variables:

```
map_.source = 'M33'
map_.telescope = '30m'
map_.species = '12CO10'
map_.fluxUnit = 'Tmb'
map_.resolution = '12'
map_.comments = ['cube', 'regrid']
map_.dataFormat = 'fits'
```

Using one parameter database with line-names, objects and telescope parameters which can be edited by the user (see *Installation of astrolyze*), astrolyze is able to pull more information about the object, telescope and line emission of the map. When the information is present it generates automatically the following variables:

```
map_.frequency [Hz]
map_.wavelength [m]
map_.calibrationError
map_.type [Hubble Type]
map_.distance [pc]
map_.vlsr [km/s]
map_.centralPosition [RA, DEC] J2000
map_.pa [degrees]
map_.inclination [degrees]
map_.R25 [kpc]
```

If not present in the database these variables are set to 'NaN' (Not a Number).

Although, all of this information is somewhat redundant to the header information in the files, it has been decided to go that way since unfortunately not all headers are kept up to date and manipulating the file name is easier to do.

The maps module tries to keep track if a variable that should go into the header of a fits file is changed and up-dates the header subsequently (Maybe not true in all cases, though.).

Last but not least using this name convention has another benefit since it makes the life of your fellow astronomers easier when they have to work with your data since they readily know their most important basic properties.

MODULES

astrolyze is divided in different modules and classes which can inter-operate with each other and which are:

- maps
- spectra
- sed
- functions
- lte

In the following I will introduce the individual modules of astrolyze and their functionality.

Not all functions of the individual astrolyze modules will be covered in this manual-page. All functions are documented and can be found [HERE](#) `TODO`. The next sections attempts to show the basis principles, structure and work-flow of astrolyze.

MAPS

The `maps` module is the heart of the `astrolyze` package. It provides the framework to work with astronomical images (and spectra). It is able to modify and to track the most important properties of the maps such that they are always fast at hand if needed. Further it contains genuine functions written in python (and `pyGildas`), alongside wrapper functions to functions and tasks of `GILDAS` and `MIRIAD`.

The `maps` module contains the following modules/Classes itself:

- `main.Main`
 - `fits.FitsMap`
 - `gildas.GildasMap`
 - `miriad.MiriadMap`
 - `stack.Stack`
 - `tools`
-

MAIN

This module contains the class: `Main`, which is the parent class for all actions not depending on the actual data-type. In principle it is the implementation of the *Naming Convention* and is not directly used to open a map. All other classes in maps inherit from the `Main` class, such that it is used any time the other modules of maps are used. Also the *Using the sed module* uses `Main`.

5.1 Variables created by Main

When opening a map with `Main` variables are created that contain information about the maps and can be used to work with the data. The variables are class attributes. They can be accessed by attaching them to the object with a dot between object and attribute:

```
object.attribute
```

In astrolyze the maps that are opened are objects. The information for the variables come from different places:

5.1.1 From naming convention

```
map_name  
prefix  
source  
telescope  
species  
fluxUnit  
resolution  
comments  
dataFormat
```

Calculated Values

Assuming a Gaussian beam size the following variables are calculates from the `resolution` keyword.

```
beamSizeM2      Beam-Size in m^2  
beamSizeSterad  Beam-Size in sterad
```

Known Format endings of the different file types

These variables contain a list that contain all strings that are recognized by the system as a valid file name ending (TODO: find the correct name for file ending).

```
fits_formats
miriad_formats
gildas_formats
class_formats
```

Warning: These are hard-coded, should be initialized via the `cfg/` folder of the package when installing.

5.1.2 Variables loaded from Database

As explained in TODO. Astrolyze uses a database to store additional informations about maps. The database is set-up during installation as explained in *Installation of astrolyze*.

From Line Database

Using the `species` keyword:

```
frequency : [GHz]
wavelength : [TODO]
```

From Galaxy Database

Using the `source` keyword.:

```
centralPosition : [RA, DEC]
distance : [kpc]
R25 : [kpc TODO]
inclination : [degrees]
vlsr : [km/s]
pa : [degrees]
type : The Hubble type of the galaxy.
```

From Calibration Database

Using the `species` and `telescope` keywords.:

```
calibrationError
```

The in-depth documentation of the internal functions can be found here *Main*.

FITS

The fits module contains the class `FitsMap`. This class inherits all functions from `Main` but adds functionality and methods needed to work with fits data. This class makes use of the python-pyfits package. Please check the pyfits web-page (follow the previous link) for the pyfits documentation.

To open a fits-map do, e.g.

```
from astrolyze import *
map = FitsMap('M33_30m_12CO10_Tmb_12.fits')
```

6.1 Variables created by FitsMap

Opening a Fits map creates, besides the variables discussed in *Main* the following additional variables specific to the `FitsMap`:

6.1.1 Variables loaded with pyfits

```
header : a dictionary with the fits-header cards
data : numpy.ndarray containing the data
hdulist : Some fits-files have more than one plane. HDUlist contains all
          planes.
```

6.1.2 Calculated values

Because the fits-header is read via python-pyfits the pixel dimensions are known and thus the pixel-sizes can be calculated:

```
pixelSizeM2 : [m^2]
pixelSizeSterad [sterad] .
```

6.1.3 Known header keywords

```
headerKeywords : A dictionary containing different fits-header keywords that
                  are used for the same quantity.
```

6.1.4 Known names of units

The `FitsMap` class contains the method `change_unit` which is able to check the conversion that is needed to change to the desired unit. However for this another Unit Naming Convention is needed. The following variables contain lists of valid names for certain variables:

```
tmb_names
jansky_beam_names
MJy_per_sterad_names
erg_sec_pixel_names
erg_sec_beam_names
jansky_pixel_names
erg_sec_sterad_names
known_units .
```

Warning: This should also be setup during installation of astrolyze to be able to customize unit-names easily.

6.2 Working with Fits Maps

6.2.1 Changing Formats

Note: Changing formats works the same way in all `maps` modules.

The loaded fits map can be transformed to the Gildas and Miriad Formats via the methods:

```
toGildas()
toMiriad()
```

and from either of the other two Formats to Fits via:

```
toFits()
```

This creates a copy of the image in the Gildas, Miriad (or Fits) Format on the disk in the same folder as the actual map. To keep working with the map in the new format one can use:

```
map = map.toGildas()
```

to change the out-put folder there are two ways:

1. Set the `map.prefix` variable to the new path prior to changing the map format as shown above.
2. Give the new path to the function, e.g. `map.toGildas(folder='new_path')`

Thus e.g.:

```
map.prefix = 'new_path'
map = map.toGildas()
```

or:

```
map = map.toGildas(folder='new_path')
```

6.2.2 Making use of Pyfits

As listed in *Variables created by FitsMap* a data and a header variable is created. These are a `numpy.ndarray` of the image data and a dictionary containing the header keywords with their arguments. Thus changing the properties of the fits map and the header variables is straight-forward.

For example you may want to change the units of a map `M33_30m_12CO10_Tmb_12.fits` that is in main beam temperature to Jansky/beam by multiplying all pixels with the corresponding conversion factor. This can be done via:

```
from astrolyze import *
map = FitsMap('M33_30m_12CO10_Tmb_12.fits')
conversion_factor = map.flux_conversion()
map.data = map.data * conversion_factor
```

The conversion factor between the two units is calculated here via `flux_conversion` please refer to the function for more details.

Having changed the unit it may be a good idea to change the header variables that stores the actual unit as well as the unit-keyword from the naming convention., this can be done as follows:

```
from astrolyze import *
map = FitsMap('M33_30m_12CO10_Tmb_12.fits')
print map.header['BUNIT']
map.unit = 'JyB'
map.header['BUNIT'] = 'Jy/beam'
```

Note: The individual functions in of the `FitsMap`-class change the header of the Fits file automatically if map properties are changed. At the same time they have access to all fits-header variables if needed.

Finally you may want to save the modified image as a new fits file which can be done with:

```
map.updateFile()
```

This creates a file called `M33_30m_12CO10_JyB_12.fits`

6.2.3 Change Map Units

Since it is easy to modify all pixel via python-pyfits the function `change_unit` that converts units into another is implemented in the `FitsMap`-class. See the documentation of `change_unit` for the conversions that are implemented.

Using the information from the “Naming Convention” and the additional information from the database, the code has all information to calculate the conversion factors. And also to decide which conversion is to be applied. Changes are possible between any of the units that are implemented.

Warning: This is not thoroughly tested yet!!

For example the same change of units as in the previous section can be done as follows:

```
from astrolyze import *
map = FitsMap('M33_30m_12CO10_Tmb_12.fits')
map = map.change_unit('JyB')
```

USING GILDAS

When Opening a Gildas map with:

```
from astrolyze import *  
map = GildasMap('M33_30m_12CO10_Tmb_12.gdf') ,
```

astrolyze uses pyGildas to read some of the header variables and stores them as python variables. The following values are read.

```
dimensions  
naxis_1 : number of pixels in x-axis.  
naxis_2 : number of pixels in y-axis.  
  
crpix_1 : The reference pixel (x-axis).  
crval_1 : The coordinate value at crpix_1 [radians].  
cdelt_1 : The pixel increment.  
  
crpix_2 : Same as crpix_1 for y-axis.  
crval_2 : Same as crval_1 for y-axis.  
cdelt_2 : Same as crdelt_1 for y-axis.  
  
ra_coordinate : RA coordinate.  
dec_coordinate : DEC coordinate  
  
ra_coordinate : TODO  
dec_coordinate : TODO  
  
central_coordinate_degrees : TODO  
central_coordinate_equatorial : TODO
```

Note: If needed more variables can be loaded when opening a Gildas map without problems , so far only those that were needed are implemented.

7.1 How is Gildas used by astrolyze

Gildas is used in two different ways in Astrolyze:

1. A Gildas script, with a final exit at the end is created and executed via the os python package, sending commands to the bash.
2. The pyGildas package is used directly by astrolyze to run command in SIC.

The first method is mainly used to wrap GREG (and maybe later CLASS) tasks since these are cumbersome to script so far in Gildas. To do so the GildasMap method create an `init` file and a temporary Greg script that is (normally) deleted after execution. The standard script looks as follows:

`exit`

7.1.1 Wrapping Gildas Tasks

7.1.2 Direc use of pyGildas

USING MIRIAD

A miriad map is opened with:

```
from astrolyze import *  
map = GildasMap('M33_30m_12CO10_Tmb_12.gdf') ,
```

In this mode only the variables created by `main` are created.

Only the "smooth" function from miriad is implemented in astrolyze see `smooth()`. When all header variables are set correctly this function is aware of the map units and beam sizes and can calculate the correct scaling that has to be applied after convolution. The way astrolyze is set up the smooth function expects the map to be in units of Jy/beam.

Warning: Miriad could be excluded from astrolyze in the future if the GILDAS smooth wrapper is programmed such that it is aware of different input variables.

USING THE STACK MODULE

The `Stack` class builds on top of the classes:

```
Main  
FitsMap  
GildasMap  
MiriadMap
```

It provides the possibility to load a “stack” of maps inside a folder at once and perform changes on all maps. It’s intention is mainly to provide means to unify maps for further processing. Tasks such as for example pixel-to-pixel comparisons or the creation of temperature-maps from SED-fits needs individual maps to be on the same resolution and/or the same pixel grid.

-

USING THE SED MODULE

The module `SED` makes extensive use of the stack functionality of the `maps` module...

10.1 Using Functions

The module `functions` contains all functions that are not directly related to manipulation of maps or spectra and counter intuitive also all constants used in `astrolyze` as long as they are not available by the standard python installation (this may not be true however.).

10.2 Using LTE

10.3 References

MAPS

11.1 Main

class `astrolyze.maps.main.Map` (*map_name*, *name_convention=True*)

Map is the parent Class for the maps-package. It contains all functions that are common to all supported map-formats, i.e. Fits, GILDAS and Miriad. This class is only supposed to be called through the FitsMap, GildasMap, and MiriadMap classes.

Parameters `map_name` : string

The name and path of the file that is to be initialized to the maps package.

`name_convention` : True or False

Only files following the `name_convention` are supported.

Methods

resolutionToString (*resolution=None*)

Converts the resolution list to a string to be printed and included in the file names.

get_beam_size ()

Calculates the beam-size in steradians and in m². For the latter the distance to the source has to be given.

Returns Initialization if the variables :

`self.beamSizeSterad` and `self.beamSizeM2` :

Notes

The formula used is:

change_map_name (*source=None, telescope=None, species=None, fluxUnit=None, resolution=None, comments=None, dataFormat=False, 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.

returnName (*source=None, telescope=None, species=None, fluxUnit=None, resolution=None, comments=None, dataFormat=False, prefix=None*)

Returns the Name corresponding to the Name convention. Single keywords can be changed.

This function is useful to generate a writeout name for a changed file without overwriting the current `self.map_name`.

Parameters All possible parameters from the “Naming Convention” plus the new :

`prefix` :

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

Parameters **x** : float [GHz]

Wavelength/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 wavelength = 'lambda'.

direction : string

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

Notes

If self.frequency and self.resolution are correctly set, this functions does not need any input. Otherwise this has to be given explicitly.

11.2 FitsMap

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

Methods

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

get_pixel_size ()

Calculates the Area of a pixel in m² and steradians if distance is given. If not only steradians are calculated.

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`

read_aperture (*position*, *apertureSize=0*, *backgroundSize=0*, *output=False*, *annotation=False*, *newAnnotation=False*)

Extract the sum and mean flux inside an aperture of a given size and at a given position.

This function can be used to read the flux in the area of a circular aperture, 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 aperture-Size and backgroundSize. Default is 0 and thus **no background substaaction** is applied.

output : True or False

If True the function reports the calculated values during execution.

annotation : logical

If True a kvis annotation file "`apertures.ann`" containing the aperture used to integrate the flux is created. 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]

Notes

The pixel sizes have to be quadratic for the algorithm to work. It measures a circle by counting the pixels from the central pixel corresponding to the given coordinate.

sky2pix (*coordinate*, *origin=0*)

Calculates the Pixel corresponding to a given coordinate.

Parameters **coordinate** : list

Either ['RA','DEC'], e.g. ['1:34:7.00', '+30:47:52.00'] in equatorial coordinates or [RA, DEC] in GRAD.

origin : int

0 or 1; this steers how the first pixel is counted 0 is for usage with python as it starts to count from zero. 1 is the fits standart.

Returns **pixel** : List

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

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 coordinates to be returns. 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*.

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 gaussian 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.

change_unit (*final_unit, frequency=None, debug=False*)

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" "JyBeam"
2. Jy/pixel: "JyP", "JyPix", "JyPixel"
3. MJy/sterad: "MJyPsr", "MJPSR", "MJy/sr"
4. Main Beam Temperature: "Tmb", "T", "Kkms"
5. erg/s/pixel: "ergs" "ERGPSECPPIX", "ERGPSECPPIXEL",
"ERG-S-1-Pixel", "ERG-S-1"
6. erg/s/beam: "ERGPSECPBEAM"
7. erg/s/sterad "ERGPSTER"

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 development and not all conversions may work properly.

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.toGildas()
```

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

```
>>> map.toGildas()
```

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.

11.3 GildasMap

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

Methods

set_defaults ()
Reset all selection criteria.

spectrum (*coordinate, fileout=None, prefix=None, create_spectrum=True*)
Wrapper to the spectrum command from greg that extracts a spectrum from a cube at a given positions. By default it also creates a 30m file readable by class from the table.

Parameters `coordinate` : list

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

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.

create_spectrum : True or False

Turn the creation of a 30m with the spectrum of False or on True.

Examples

```
>>> from astrolyze.maps import *
>>> map = GildasMap('M33_PdBI_12co10_Tmb_22.0_2kms.gdf')
>>> coordinate = ['1:34:7.00', '+30:47:52.00']
>>> map.spectrum(coordinate)
```

Creates M33_PdBI_12co10_Tmb_22.0_2kms.tab in the present folder.

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 map_name with .30m ending.

prefix [string] The path where the class file will be stored. Defaults to the current path.

.. **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.mask('poly/sourceA.pol')
```

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.reproject(coord = ['1:34:32.8', '30:47:00.6'])
>>> map.reproject(template = 'M33_SPIRE_250_JyB_18.1.gdf')
```

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 in velocity 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.

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.goRot(45)
```

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

```
>>> map = map.reproject(coord=['new_RA_string', 'new_DEC_string'])
>>> map.goRot(45)
```

slice (*coordinate1*, *coordinate2*, *prefix*=None, *comment*=None)

Wrapper for the GREG task slice. Producing Position velocity cuts trough a map between coordinate1 and coordinate2.

Parameters **coordinate1** : string

The coordinate where the cut trough the map starts.

coordinate2 : string

The coordinate where the cut trough the map ends.

Returns A GildasMap object containing the slice. :

Notes

This only works with cubes.

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

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.**

custom_go_spectrum (*coordinate=False*, *size=False*, *angle=0*)

This function uses the go_spectrum command from GREG to plot the spectra in a region given by size around the coordinate given by coordinate.

Parameters **coordinate** : list

A list with the coordinates in floats in units of Degrees, or in string for equatorial coordinates. Default to False which means that the center of the map, determined from the map header, is used.

size : list

The region around the coordinate from which spectra are plotted in arcsec, e.g. size = [50, 50] means a region of 50x50 arcsec around the given coordinate. Defaults to None, which translates to size = [0, 0] which in turn is interpreted as the full map size by GREG.

angle : float [degrees]

Needed if the map is rotated to get the correct offsets. Defaults to 0.

save_figure (*name=None*)

Helper function that saves the current plot.

quick_preview (*save=False*, *filename=None*, *window=True*)

Plotting the map and optionally save the figure.

Parameters `save` : True or False

Choose whether or not to save the figure.

filename : string

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

window : True or False

Choose whether the image display is opened or not. Default True.

toFits()

Converts the actual map to a Fits map.

Returns **FitsMap Object.** :

Examples

With:

```
>>> map = gildasMap('M33_MIPS_24mum_JyB_5.gdf')
>>> map = map.toFits()
```

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.

Returns **MiriadMap Object.** :

Examples

With:

```
>>> map = gildasMap('M33_MIPS_24mum_JyB_5.gdf')
>>> map = map.toMiriad()
```

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

11.4 MiriadMap

```
class astrolyze.maps.miriad.MiriadMap (map_name, name_convention=True)
```

Methods

toFits()

Converts the actual map to a Fits map.

Returns **FitsMap Object.** :

Examples

With:

```
>>> map = miriadMap('M33_MIPS_24mum_JyB_5')
>>> map = map.toFits()
```

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

toGildas()

Converts the actual map to a Gildas map.

Returns **GildasMap Object.** :

Examples

With:

```
>>> map = miriadMap('M33_MIPS_24um_JyB_5')
>>> map = map.toGildas()
```

it is possible to continue working with the Fits map, using `maps.gildas.GildasMap` class.

toMiriad()

Copies the actual map changing the name such that it takes changes in keywords into account.

Returns **MiriadMap Object.** :

Examples

With:

```
>>> map = miriadMap('M33_MIPS_24um_JyB_5')
>>> map = map.toMiriad()
```

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

smooth(new_resolution, old_resolution=None, scale='')

Smooths a miriad map to the new resolution.

Parameters **new_resolution** : float or list

The resolution in of the smoothed image. Can be a:

- **float:** Output beam has same major and minor axis [arcsec] and the position angle (PA) [degrees] is 0.
- **A list with two entries:** The major and minor axis. PA is 0. E.g. [major_axis, minor_axis]
- **A list with three entries:** [major_axis, minor_axis, PA]

old_resolution : float

If None the self.resolution information is taken into account. Otherwise, it is assumed that old_resolution is the actual resolution of the map.

scale : string

If unset (scale=''), the miriad function will attempt to make the units of the smoothed image be Jy/beam for Gaussian convolution. If 0.0, then the convolution integral is scaled (multiplied) by the inverse of the volume of the convolving function. Otherwise, the integral is scaled by "scale"

Returns **MiriadMap** : object

The smoothed image.

Notes

The function used to calculate the fwhm (Omega) of the convolving Gaussian for both major and minor axis is:

$$\Omega_{\text{convolve}} = \sqrt{\Omega_{\text{new}}^2 - \Omega_{\text{old}}^2}$$

11.5 Stack

class `astrolyze.maps.stack.Stack` (*folder, data_format=None*)

Allows to treat a folder of images as a whole and to perform the same transformations on all of the maps. It is also the basis for the SED package. The images in the input folder can have arbitrary formats, units, resolutions and other specifications. The Stack class can help to unify the different parameters of the maps to help comparing them.

Methods

get_map_format (*map_name*)

This function creates returns the correct `GildasMap`, `FitsMap` or `MiriadMap` object without that the data format of the map has to be given.

Parameters `map_name` : string

Path and name of the input map.

Returns Either a “`GildasMap`“, “`FitsMap`“ or “`MiriadMap`“ Either a :

“`Map`“ class object depending on the data format of the input map. :

get_list (*data_format=None, depth=False*)

Loading a list of files in all sub-folders.

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 sub-folder 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.

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

Examples

Say the folder structure is like this

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

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

```
>>> from astrolyze.maptools import *
>>> list = maptools.get_list('../modified')
>>> maptools.copy_structure(list, old_prefix='../modified',
>>>                          new_prefix='../even_more_modified')
```

unify_resolutions (*folder=None, resolution=False*)

Smoothing all maps to the same resolution; either the maximum resolution found in the stack or a given resolution.

Parameters **resolution** : float or list

This may be either:

1. A list with three entries, i.e. `[[minor_fwhm], [major_fwhm], [position_angle]]`
2. A list with two entries, `position_angle` defaults to 0, i.e. `[[minor_fwhm], [major_fwhm]]`
3. A float. Same minor, major fwhm `pa=0`

folder : string

The path tot the folder in which the files are to be stored.

Notes

The position angle of the output images is fixed to `Zero` and can currently not be modified.

Depending on the map units different scaling normalizations have to be used after smoothing such that the output units are correct. This function tries to deduce the scaling by itself based on the unit that is given in the map name. Be sure that this is correct otherwise the flux values may be wrong in the output images.

unify_units (*unit='JyB', folder=None, debug=True*)

Change all maps in the stack to the same unit.

Parameters **unit** : string

See `astrolyze.maps.fits.FitsMap.change_units()` for details.

folder : string

The target folder. By default the maps are put into their current folder.

unify_dimensions (*template=None, folder=None*)

Reproject all maps to the same central coordinates and map dimensions. All properties of one template map are copied to all the other maps using the `"reproject"` task of GILDAS.

Parameters **template** : string

Path to the map that will serve as a template, it may be one of the input maps of the stack.

folder : string

Path to the folder where the output maps are stored.

unify_projections (*coordinate=None, angle=None, folder=None*)

Changing the central coordinate and the rotation angle.

unify_formats (*target_format='fits', folder=None*)

Converts all maps to the `target_format` which has to be one that is known to astrolyze. Please see TODO for a list of known formats. Default is fits.

Parameters **target_format** : string

The format to which all maps are converted.

folder : string

Path to the folder where the new maps are stored. If `None` the current folder will be used.

pixel_pixel_compare (*folder=None, plot=False, tol=1000000.0*)

Producing a pixel-to-pixel comparison for all combinations possible between the maps of the stack.

Parameters **folder** : string

Path to the folder where the text files with the pixel to pixel comparisons are stored

plot : [True | False]

Decides whether to produce pixel-to-pixel plots directly.

tol : float

The tolerance for the maximum difference between the values of the pixel of two compared maps. Default to 1e6.

Returns **Created text files in the specified folders that contain two columns :**

with the pixel-to-pixel comparisons. These can be used

11.6 Tools

`astrolyze.maps.tools.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.maps.tools.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

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

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

```
>>> from astrolyze.maptools import *
>>> list = maptools.get_list('../modified')
>>> maptools.copy_structure(list, old_prefix='../modified',
>>>                          new_prefix='../even_more_modified')
```

`astrolyze.maps.tools.unifyUnits` (*list, folder='units'*)

NOT READY YET!

`astrolyze.maps.tools.unifyMaps` (*list, tinMap, folder='reg'*)

changes the dimensions and pixel sizes off all maps to that of a template map.

`astrolyze.maps.tools.unifyResolution` (*liste, resolution=False, folder='smooth', scaling=''*)

Approved.

SPECTRA

12.1 Class

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

Examples

Extracting a spectra at a given position from a spectral cube can be done as follows

```
>>> from astrolyze.spectra import *
>>>
>>> cube = ClassSpectra(filename)
>>> coordinate = ['1:34:7.00', '+30:47:52.00']
>>> cube.get_spectra_from_cube(coordinate)
Generates a 30m file with comment extract in the actual cube.prefix path.
```

Methods

set_defaults()
Reset all selection criteria.

get_spectra_from_cube (*coordinate, angle=0, prefix=None, accuracy=2, region=False*)
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.

region: True or False :

Returns either all spectra found `True` or only the first `False`.

Returns 30m file :

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

get_region_from_cube (*coordinate*, *angle=0*, *prefix=None*, *accuracy=10*)

The same as :py:func:~get_spectra_from_cube but returns all spectra found inside a circular region around coordinate and in a radius of accuracy arcsec. ("set match ""accuracy")

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.

save_figure (*name=None*)

Helper function that saves the current plot.

quick_view (*number=1*)

Helper Functions that displays the first spectrum of the loaded file.

SEDS

class `astrolyze.sed.sed.Sed`(*source_name*, *coordinate*, *flux_array*, *number_components*=2,
 init_fit=True)

This class handles a single SED. Basically it is able to fit, and plot the sed.

Parameters *source_name* : string

The name of the source to which the SED corresponds to.

coordinate : list

The coordinate of the source. [RA, DEC]

flux_array : list

The array that is created by SedStack with the entries of wavelength, flux, and error.

init_fit : logic

Steers whether the SED is fitted already during creation.

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 0x4834a50>, *nu_or_lambda*='nu',
 color='black', *linewidth*=0.5, *x_range*='normal')

Plot a multi component greybody model.

Parameters *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'.

```
create_figure (save=True, plotLegend=False, color=['black'], marker=['x'], title=None,
               x_label=None, y_label=None, nu_or_lambda='nu', fontdict=None,
               textStringLoc=[1, 1], lineWidth=0.5, kappa='easy', x_range='normal')
```

Creates a quick preview of the loaded SED. TODO: extend documentation.

```
class astrolyze.sed.sed.SedStack (folder, data_format='.fits', coordinates=False,
                                   flux_acquisition='aperture', aperture=120, annotation=False, full_map=False, output_folder=None, number_components=2)
```

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

Methods

```
load_coordinates (input_file)
```

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

self.source_names : 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']

```
get_seds ()
```

Creates a stack of SEDs from the stack of maps loaded from the input folder if particular coordinates are given.

```
get_map_seds (folder)
```

This functions fits the SED at every pixel of the input maps. Note that the maps have to be exactly the same size for this function to work. This can be achieved with e.g.:

```
from astrolyze import *
stack = Stack('some_input_folder')
folder = 'some_output_folder'
template = 'Path_to_a_template_map'
stack = stack.unify_dimension(template, folder)
```

Parameters **folder** : string

The path to the folder where the temperature, mass, beta and chisq maps are created. If the folder does not exist it will be created.

Notes

Depending on the number of pixels in the input image this function may take a good while to finish.

LTE

Functions to calculate LTE column densities.

TODO: Add Documentation.

`astrolyze.lte.lte.calc_jnu` (*nu*, *T*)
Calculates J_ν needed for `lte_column_density`.
Parameters *nu* : float
Frequency
T : float
Temperature

Notes

The formula (in cgs units) implemented here is:

$$\mathcal{J}_\nu(T) = \frac{h\nu}{k} \frac{1}{e^{h\nu/kT_{ex}} - 1}$$

where:

- k*: the Boltzman constant in CGS
- h*: the Planck constant in CGS
- ν*: the frequency
- T*: excitation temperature

References

Doktorarbeit

`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

The implemented formula, taken from Doktorarbeit is:

$$N = \frac{3h}{8\pi^3\mu^2} \frac{Z}{J} \frac{\exp(\frac{h\nu}{kT_{ex}})}{[1 - \exp(-\frac{h\nu}{kT_{ex}})]} (\mathcal{J}_\nu(T_{ex}) - \mathcal{J}_\nu(T_{BG}))^{-1} \int T_{mb} d\nu,$$

where:

- k: the Boltzman constant in CGS
- h: the Planck constant in CGS
- W: integrated Intensity in Kelvin cm/s
- Aul: the Einstein coefficient of the transition
- gu: the statistical Weight of the upper level
- Eu: the Energy of the upper level
- excitation_temperature
- Z: the partition Function

References

Warning: Extend documentation!!!!

[Doktorarbeit]

`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: :

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

class `astrolyze.lte.molecule_parameter.Molecule(nu, Q, T, Eu, Aul, gu, mu, name='')`

A class holding attributes that define the characteristics of an individual transition of a molecule.

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

Parameters nu: float

Q: list

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

T: list

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

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.

FUNCTIONS

15.1 astro_functions

`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 wavelength/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 wavelength 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 wavelength:

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

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:

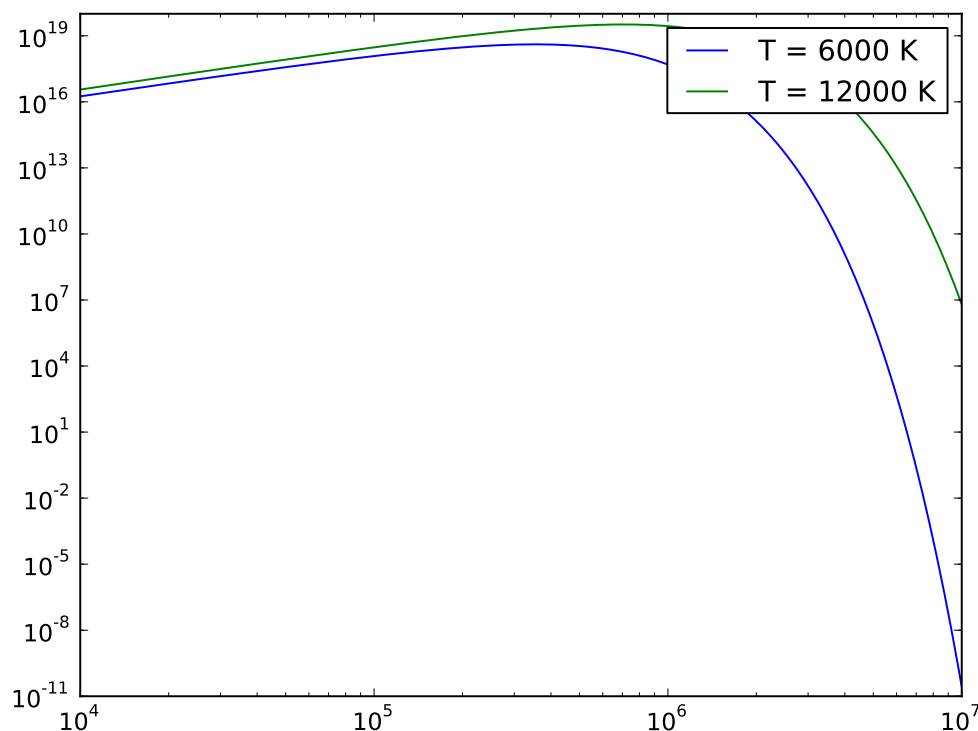
```
import numpy as np
import matplotlib.pyplot as plt
import astrolyze.functions.astro_functions as astFunc

frequency_range = np.arange(1e4, 1e7, 1e4)
temperature_1 = 6000
temperature_2 = 12000 # Kelvin
blackbody_1 = astFunc.black_body(frequency_range, temperature_1)
```

```

blackbody_2 = astFunc.black_body(frequency_range, temperature_2)
figure = plt.figure()
axis = figure.add_subplot(111)
pl = axis.loglog(frequency_range, blackbody_1, label='T = 6000 K')
pl = axis.loglog(frequency_range, blackbody_2, label='T = 12000 K')
pl = axis.legend()
plt.savefig('black_body.eps')

```



```

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 a detailed description assumptions and equations used.

Parameters **p** : list

List of the parameters defining a grey_body, being Temperature [K], column density or mass (dependent on the kappa used) and the grey_body slope index beta, respectively (refer to notes for more information): $p = [T, N, \text{beta}]$

x : float or numpy array

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

kappa : string

Chooses the dust extinction coefficient to use:

- "easy" -> $\kappa = \nu^\beta$; $\tau = N * \kappa$
- "Kruegel" -> $\kappa = 0.04 * (\nu / 250 \text{ GHz})^\beta$; $\tau = M / D^2 * \kappa$

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 wavelength 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 while 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 simplifies 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:

$$\kappa = 0.04 * \left(\frac{x [GHz]}{250 GHz} \right)^\beta$$

$$S_\nu = M[kg] / D^2[m^2] * \kappa * black_body(x, T).$$

References

[KS]

Examples

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

```
astrolyze.functions.astro_functions.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/wavelength.

Parameters `pMulti` : nested lists

Similar to `p` from `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 wavelength **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.grey_body_fit (data, start_parameter,
                                                    nu_or_lambda='nu',
                                                    fit_beta=False,
                                                    fix_temperature=False,
                                                    rawChiSq=None,
                                                    kappa='Kruegel', resid-
                                                    uals=False, itera-
                                                    tions=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 wavelength 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 parameters to fit. Assure that: number of data points > number of free parameters.

```
astrolyze.functions.astro_functions.LTIR(p2, kappa='Kruegel', xmin=3.0,
                                         xmax=1100.0, beamConv=True, dis-
                                         tance=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 L_{sun} 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.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 corresponding 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` = wavelength/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_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.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.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.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

Containing 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.analytic_linear_fit(x, y, x_error, y_error)`

This function resembles the analytical solution following chapter 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:

$$y = A + Bx$$

$$A = \frac{\Sigma(x^2) \cdot \Sigma(y) - \Sigma(x) \cdot \Sigma(x \cdot y)}{\Delta}$$

$$B = N \frac{\Sigma(x \cdot y) - \Sigma(x) \cdot \Sigma(y)}{\Delta}$$

$$\Delta = N \cdot \Sigma(x^2) - (\Sigma(x))^2$$

Warning: This has to be checked.

References

[TA]

`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 corresponding 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.line_monte_carlo(p, x, y, x_error, y_error, iterations, filterations=1000000000.0)`

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

Parameters `p` : list

Containing 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 fitted 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.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 calucated.

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:

$$f = \frac{1}{fwhm * \sqrt{2 * \pi}} * e^{-1/2(\frac{x-x_0}{fwhm})^2}$$

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

Calculates a 2D Gaussian at position x y.

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

the x-axis value/values where the Gaussian is to be calculated.

y : float or numpy.ndarray

the y-axis value/values where the Gaussian is to be calculated.

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:

$$\begin{aligned} a &= \cos(pa) * 2 / (2 * major * 2) + \sin(pa) * 2 / (2 * minor * 2) \\ b &= (-1 * \sin(2 * pa) / (4 * major * 2)) + (\sin(2 * pa) / (4 * minor * 2)) \\ c &= \sin(pa) * 2 / (2 * major * 2) + \cos(pa) * 2 / (2 * minor * 2) \end{aligned}$$

`astrolyze.functions.astro_functions.degrees_to_equatorial(degrees)`

Converts RA, DEC coordinates in degrees to equatorial notation.

Parameters `degrees` : list

The coordinates in degrees 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*)

Converts RA, DEC coordinates in equatorial notation to degrees.

Parameters `equatorial` : list

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

Returns `degrees` : list

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

Raises `SystemExit` :

If `equatorial` is not a list of strings in the above format.

`astrolyze.functions.astro_functions.calc_offset` (*central_coordinate*, *off-*
set_coordinate, *angle=0*, *out-*
put_unit='farcsec')

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 respect 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.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.

`astrolyze.functions.astro_functions.vel_to_freq_resolution` (*center_frequency*,
veloc-
ity_resolution)

Converts a velocity resolution to frequency resolution for a given center frequency.

Parameters `center_frequency` : float

Center frequency in GHz.

velocity_resolution : :

Velocity resolution in km/s.

Returns `frequency_resolution` : float

The corresponding frequency resolution in Mhz

Notes

Approved!

```
astrolyze.functions.astro_functions.freq_to_vel_resolution(center_frequency,  
                                                         fre-  
                                                         quency_resolution)
```

Function to convert a frequency resolution to a velocity resolution for a given center frequency.

Parameters `center_frequency` : float

Center frequency in GHz.

frequency_resolution : float

The frequency resolution in MHz.

Returns `velocity_resolution in km/s.` :

Notes

Uses the formula TODO $v_{\text{LSR}} = c((\nu_0 - \nu_{\text{Obs}})/\nu_0)$

Approved!

```
astrolyze.functions.astro_functions.v_lsr(center_frequency, observation_frequency)
```

Calculates the velocity that corresponds to a certain frequency shift between two frequencies.

Parameters `center_frequency` : float

center_frequency in GHz

observation_frequency : float

The observation frequency in GHz.

Returns `v_lsr` : float

The velocity corresponding to the frequency shift in km/s

Notes

Approved!

```
astrolyze.functions.astro_functions.redshifted_frequency(rest_frequency,  
                                                         v_lsr)
```

Calculates the sky frequency corresponding to a rest frequency for a source with a velocity `v_lsr`.

Parameters `rest_frequency` : float

The frequency of the line at rest in Ghz (More often state the obvious :)).

v_lsr : float

The velocity of the source in km/s.

Returns `redshifted_frequency` : float

The sky frequency in GHz.

Notes

The formula used is:

$$\nu_{sky} = \nu_{rest} * \frac{-1v_{lsr}}{c + 1}$$

Approved!

```
astrolyze.functions.astro_functions.frequency_to_wavelength(frequency)
```

Converting frequency to wavelength.

Parameters **frequency** : float [GHZ]

Returns **wavelength** : float [micron]

15.2 units

Constant unit conversions available in this module are:

```
# Constant conversion factors.
#=====> Approved !!! <=====
WattToErgs      = 1e7 # 1W = 1e7 erg/s
ErgsToWatt      = 1e-7 # 1W = 1e-7 erg/s
JanskyToWatt    = 1e-26 # 1Jy = 1e-26 W/m2/Hz
WattToJansky    = 1e26 # 1W = 1 Jy * m2 * Hz
ErgsToJansky_cm = 1e23 # 1 erg/s = 1e23 Jy * cm2 * Hz * s
JanskyToErgs_cm = 1e-23 # 1 Jy = 1e-23 erg/s/cm2/Hz
ErgsToJansky_m  = 1e19 # 1 erg/s = 1e-19 Jy * m2 * Hz * s
```

```
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

wavelength/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'.

Returns **factor** : float

The conversion factor.

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

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

Parameters **x** : float

wavelength/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 wavelength = 'lambda'.

Returns **factor** : float

The conversion factor.

Notes

Approved.

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

Conversion factor to calculate luminosity from intensities by integrating over the sky $4\pi \text{Distance}^2$.

Parameters **distance_in_pc** : float

Distance to the source in parsecs.

cm_or_m : string

Choose whether the output is in cm^2 = 'cm' or in m^2 = 'm'.

Notes

Approved.

`astrolyze.functions.units.JyBToErgsB(input_flux, distance, wavelength, invert=False, map_use=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.

wavelength : float

Wavelength λ in μm .

map_use :

Returns The conversion factor (**map_use = true**) or the already converted flux :

(**map_use = False**). :

r :

`astrolyze.functions.units.JyBTOWM2Kpc2(input_flux, distance, major, minor, wavelength, invert=False, map_use=False)`

Conversion between Jy/beam and $\text{W m}^{-2} \text{kpc}^{-2}$

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).

wavelength : float

Wavelength λ in μm

invert : True or False

Changes the direction of conversion.

Returns **float** : the converted Flux.

`astrolyze.functions.units.JyBTOWKpc2` (*input_flux, distance, major, minor, wavelength, invert=False, map_use=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).

wavelength : float

Wavelength λ in μm .

invert : True or False

Changes the direction of conversion.

Returns **float** : the converted Flux.

`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

wavelength/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 wavelength = 'lambda'.

Notes

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

References

[GILDAS]

`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

wavelength/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 wavelength = 'lambda'.

Notes

Same as `kelvin_to_jansky()`

15.3 constants

```
# Natural Constants
c = 299792458. # Speed of light [m]
c_in_cm = c * 1e2 # Speed of light [cm]
h = 6.62606896e-34 # Plancks constant [Js]
k = 1.3806503e-23 # Boltzman constant [m^2 kg s^-1 K^-1]
tBG = 2.7 # Cosmic Microwave Background Temperature in [K]
e = 2.7182818284 # Eulers number

# Natural Constants in cgs units.
k_CGS = 1.3806503e-16 # Boltzman constant [cm^2 g s^-1 K^-1]
h_CGS = 6.62606896e-27 # Plancks constant [Js]
c_CGS = 2.99792458e10 # Speed of light [cm]

# Conversion of distances
parsec_in_m_1 = 3.08568025e16
parsec_in_m = 3.085e16 # parsec in m
parsec_in_cm = 3.08568025e18 # parsec in cm
km_in_cm = 1e5

# Masses
m_sun = 1.9891e30 # [kg]
m_proton = 1.672621637e-27 # [kg]

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

# Luminosities
LsunW = 3.846e26 # Watts
Lsunergs = 3.846e26*1e7 # erg/s
debye_to_EsuCm = 1.e-18 # Change from debye to esu/cm

# Angle Conversions
a2r = 4.848e-6 # arcsec to radian
a2d = 1./60/60 # arcsec to degree
r2d = 180./math.pi # radian to degree
```


INDICES AND TABLES

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

BIBLIOGRAPHY

[GILDAS] <http://www.iram.fr/IRAMFR/GILDAS>

[MIRIAD] <http://www.atnf.csiro.au/computing/software/miriad/taskindex.html>

[python-pyfits] http://www.stsci.edu/institute/software_hardware/pyfits

[Doktorarbeit] add reference

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

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

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

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

PYTHON MODULE INDEX

a

`astrolyze.functions.astro_functions,`
 `??`
`astrolyze.functions.units, ??`
`astrolyze.lte.lte, ??`
`astrolyze.lte.molecule_parameter, ??`
`astrolyze.maps, ??`
`astrolyze.maps.tools, ??`

PYTHON MODULE INDEX

a

`astrolyze.functions.astro_functions,`
 `??`
`astrolyze.functions.units, ??`
`astrolyze.lte.lte, ??`
`astrolyze.lte.molecule_parameter, ??`
`astrolyze.maps, ??`
`astrolyze.maps.tools, ??`