Astrolyze Documentation

Release 0.1.0

Christof Buchbender

November 02, 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.

CHAPTER

ONE

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 it 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 sent 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] Incli
    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.

CHAPTER

TWO

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_spectra_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.

CHAPTER

THREE

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.

CHAPTER FOUR

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.GilsdasMap
- miriad.MiriadMap
- stack.Stack
- tools

CHAPTER

FIVE

MAIN

This module contains the class: `Main`, which is the parent class for all actions not depending on the actual datatype. 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*.

CHAPTER

SIX

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 caluclated:

```
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 fro 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')
```

CHAPTER

SEVEN

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 Atrolyze:

```
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 tun 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

CHAPTER

EIGHT

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.

CHAPTER

NINE

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.

-

CHAPTER

TEN

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