Mount Control Software

version 1.0

Alain Klotz and Antoine Cailleau

janvier 22, 2021

Contents

Welcome in the AstroMeCCA software documentation

107, Bis route de Servies En Raynaud 81570 Cuq les Vielmur FRANCE	Director: Antoine Cailleau antoine.cailleau@astromecca.fr +33 6 64 45 73 25 http://astromecca.fr/

A German TM 350 AtroMeCCA mount equiped by a reflector 300mm telescope and a refractor 60mm telescope.

1. Using the Mount Control Software

Describes the basic use of the Mount Control Software and how to use its pad to slew the mount towards sky objects.

First steps with the Mount Control Software

The Mount Control Software (MCS) is the program that makes the link between astronomers and the hardware of the mount. MCS is installed aboard the computer linked to the mount hardware controlers.

MCS is running as a service as soon as the raspberry is booted. It allows a connection with any astronomy software using ASCOM protocol via serial port RS232 (driver astromecca, 115200 bauds). We named *astronomy software* any software that communicates with MCS (Cartes du Ciel, Prism, MaxIm DL, AudeLA, etc.).

If the ASCOM connection failed you must check if the MCS is running as service.

Check if MCS is running as service

In case of ASCOM no connection you must check the MCS state. To verify if MCS is running, in a Linux console:

```
ps -edf | grep mount
```

The answer may seems to be:

```
root       569     1  1 16:47 ?        00:00:16 /usr/bin/python3 /home/pi/astromecca/mount_c
pi        1195  1161  0 17:05 pts/1    00:00:00 grep --color=auto mount
```

What is important is the first line indicating the mount_control_software.py script is run by python3. Note the PID (process identifier) is 569. It should be different for your session.

If the mount_control_software.py appears in *ps -edf* it means the possibility to drive the mount using a astronomy software using the ASCOM protocol. The check is OK.

If the mount_control_software.py does not appear in the *ps -edf* command, the check is not OK and you should check the error log file. In a Linux console:

```
more /tmp/mcs_err.txt
```

According the contents of the file mcs_err.txt, contact the manufacturer.

Moreover, the log file can help to identify some problems. To check the log of MCS, in a Linux console:

```
tail -n30 /tmp/mcs_log.txt
```

If you want to change the running service configuration, read the section to change the configuration for running as service.

Use the pad of the Mount Control Software

If the astronomer wants to drive manually the mount, it is possible to launch MCS in another way than a service. By this way it is possible to use the virtual pad of the MCS.

First you have to kill the current instance of mount_control_system.py (i.e. that is running as service). To kill a running MCS, in a Linux console:

```
sudo kill -9 569
```

569 is the PID previously noticed in *ps -edf* list (replace 569 by the PID of your session). Once kill done, verify the MCS has disapeared of the *ps -edf* list.

Then you can launch manually an instance of mount_control_system.py

```
cd ~/astromecca
python3 mount_control_software.py -config_mount config_mcs_default.py -mount_real -pad &
```

In the command line, the file name config_mcs_default.py provides a simulator only. For a real mount you have to replace the file name config_mcs_default.py by the file name delivered for your specific mount (contact the vendor if not present in the folder).

After MCS launch the virtual pad must appears on the desktop. The pad is a window divided in tabs.

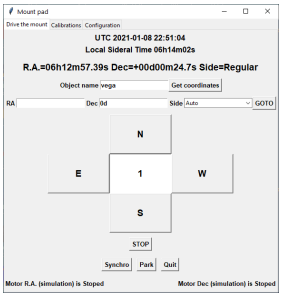
- **Drive the mount** tab is used to drive the mount for astronomical observations.
- **Calibrations** tab is to help the polar alignment.
- **Configuration** tab displays the configuration parameters.

Drive the mount

The first tab is used to drive the mount. It is possible to enter equatorial coordinates (J2000) and the side of the tube. Then push the button GOTO to start pointing. Note the side of the tube is *Regular* or *Fliped* for fork mounts and is *Tube West* or *Tube East* for German mounts. The telescope will track in sideral motion after pointing.

It is possible to move the pointed position using the N, E, S, W buttons. The moving velocity is set changing the number (1 deg/sec by default) in the center of the buttons. When buttons are released the motion return to the previous motion state (tracking or stoped).

The button STOP stops any motion of the mount (slew or track). The button Synchro records the RA, Dec, Side of pointing as the pointed coordinates. Use the button Synchro after having pointed and centered a bright star for instance. The button Park is used to park the mount. The button Quit exits the pad.

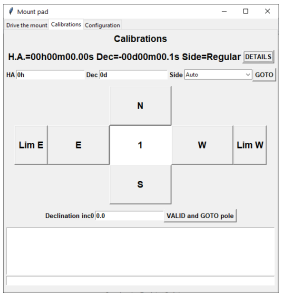


Pad tab to drive the mount.

Help for polar aligment

The second tab is for advanced users. It is possible to enter local equatorial coordinates (equinox at the date) and the side of the tube. H.A. means *hour angle*. Then push the button GOTO to start pointing. The telescope will not drift after pointing.

It is possible to move the pointed position using the N, E, S, W buttons. The moving velocity is set changing the number (1 deg/sec by default) in the center of the buttons. After release the direction buttons the telescope is stoped (i.e. no tracking).



Pad tab to help the polar aligment.

The buttons *Lim E*, *Lim W* and the *Declination inc0* value are used to fit the mechanical pole in the center of images. The buttons *Lim E*, *Lom W* allow to slew the mount around the hour angle axis in order to obtain images of stars as circles.

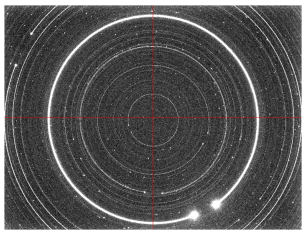


Image taken during the rotation from Lim E to Lim W.

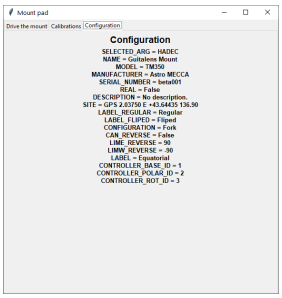
One can deduce the center of rotation. The goal is to fit the center of star circles exactly in the center of the image. One axis of offcentring depends on the value of inc0 of the declination axis:

- First: Push the *Lim E* button to put the mount in the eastern position.
- Second: Change the inc0 value and push the button *VALID and GOTO pole*.
- Third: Start an image exposure.
- Fourth: Push the *Lim W* button to put the mount in the western position.
- Fifth: Display the image and return to the first step and iterate until reaching star circles centered on the image center (see Fig. 4).

This setting must be done before any procedure of polar alignment (see the dedicated documentation align the pole).

Mount configuration

The third tab display the configuration paramters.



Pad tab to display the mount configuration.

Mount Control Software for experimented users

All the fonctionalities described now imply to kill all the MCS instances (using kill -9 described above) before launching a new one.

It is possible to launch manually mount_control_software.py via Spyder IDE instead of the command line. We let users chosing what they prefer.

There are many options to adapt MCS for your needs. Especially it is possible to choose the protocols for astronomy programs.

Change the MCS configuration of the command line

There are many options to fit the needs. The help of the software is displayed as the following procedure:

```
pi@raspberrypi:~/astromecca $ python3 mount_control_software.py -h

usage: mount_control_software.py [-h] [-config_path CONFIG_PATH]
                                                                [-config_mount CONFIG_MOUNT
                                                                [-mount_port MOUNT_PORT] [-
                                                                [-langage LANGAGE] [-transp
                                                                [-port PORT] [-client]
                                                                [-client_port CLIENT_PORT]


Launch MCS

optional arguments:
  -h, --help            show this help message and exit
  -config_path CONFIG_PATH
                                                Path where configuration files will be read.
  -config_mount CONFIG_MOUNT
                                                Configuration files for the mount.
  -mount_real           Connect to a real mount controller. You must specify
                                                the -mount_port.
  -mount_port MOUNT_PORT
                                                Port identifier for the connection with a re
                                                controller -mount_real.
  -server               To activate a server. You must specify the -langage,
                                                -transport, -port.
  -langage LANGAGE      Server langage protocol (LX200, ASCOM, ASTROMECCA).
  -transport TRANSPORT  Server transport protocol (SERIAL, TCP).
  -port PORT            Port identifier for the server to listen a client.
  -client               To test a client. You must specify the -client_port
  -client_port CLIENT_PORT
                                                Port identifier for the client to dialog wit
```

```
                                                server.
  -pad                  Display the virtual pad.
```

MCS codes are installed in the folder /home/pi/astromecca. Codes are written in Python and consist of two Python modules, files and folders:

- Module **mounastro**: Contains Python classes to drive the mount.
- Module **celme**: Contains Python classes to perform calculations of celestial mechanics.
- File **mount_control_software.py**: The Python code of MCS. It uses modules mountastro and celme.
- Folder **catalogues**: Contains catalogs of astronomical sources used by the mountastro module.
- Folder **doc_html**: Contains this documentation in HTML format (index.html).
- Folder **doc_pdf**: Contains this documentation in PDF format (index.html).
- Folder **doc_rst**: Contains this documentation in RST format (source of the documentation).

For example, if you want to launch MCS with the server MCS protocol over TCP port 1111:

```
cd /home/pi/astromecca/
python3 mount_control_software.py -pad -server -langage MCS -transport TCP -port 1111
```

If you want to launch MCS with the server ASCOM protocol over serial port /dev/ttySC1:

```
cd /home/pi/astromecca/
python3 mount_control_software.py -pad -server -langage ASCOM -transport SERIAL -port /dev/t
```

Change the MCS configuration for running as service

For AstroMECCA mounts, MCS is installed in a Raspberry computer. When the computer is booting, MCS is automatically launched according the configuration of the /etc/rc.local file. To read or change the configuration:

```
sudo geany /etc/rc.local
```

The following line is the code must be found in the rc.local file to launch MCS during the boot of the Raspberry:

```
nohup /usr/bin/python3 /home/pi/astromecca/mount_control_software.py -config_path /home/pi/a
```

If you want to start the service with the MCS protocol over TCP port 1111:

```
nohup /usr/bin/python3 /home/pi/astromecca/mount_control_software.py -config_path /home/pi/a
```

If you want to start the service another configuration file (e.g. myconf.py) placed in the folder /home/pi/astromecca:

```
nohup /usr/bin/python3 /home/pi/astromecca/mount_control_software.py -config_path /home/pi/a
```

Do not forget to reboot the Raspberry to start the service:

```
sudo reboot
```

2. Communication protocols for AstroMecCA

Describes the communication protocols to commuunicate between a client software (ASCOM, INDI or any external software) and an astromecca server.

Communication protocols for astromecca mounts

This section describes how to use a client software to communicate with a running astromecca server.

Available protocol transports are:

- Serial RS232.
- Ethernet TCP.

Available protocol langages are:

- LX200. The classical Meade LX200 syntax.
- ASCOM. ASCOM driver for Astromecca mounts
- MCS. Json based messages to allow any operation with the mount.

The default configuration of the MCS script is to run using ASCOM protocol via serial port RS232 (driver astromecca, 115200 bauds).

It is possible to mix any protocol transport with any protocol langage. However, do not forget that LX200 langage protocol is usually used with a RS232 serial transport at 9600 bauds.

1. Serial port and LX200 protocol

The client must connect a serial wire between its computer and the astromecca computer. For example, you can use an ASCOM or INDI client to communicate with the astromecca server. You have configure ASCOM or INDI to the protocol langage LX200 and the serial transport protocol as 9600 bauds.

If you want to write a Python code to create client, you can use the class MountremoteClient which wraps the serial protocol. When the MountremoteClient is instanciated the method putread_chan send the command and receive the response. The following example gets the declination of the mount via the LX200 command :GD:

```
from mountremote import MountremoteClient
remote_client = MountremoteClient("SERIAL", port="COM1", protocol="LX200", baud=9600)
command = ':GD#'
result = remote_client.putread_chan(command)
print("Command = {}\nResult = {}".format(command,result))
del(remote_client)
```

The implemented LX200 commands described in https://www.meade.com/support/LX200CommandSet.pdf are: :CM, :Gc, :GC, :GD, :Gg, :GL, :Gm, :GR, :Gt, :H, :Me, :Mn, :Ms, :Mw, :MS, :P, :Qe, :Qn, :Qs, :Qw, :Q, :RC, :RG, :RM, :RS, :Sr, :Sd, :Sg, :Sts, :U.

2. Ethernet TCP port and MCS protocol

The client must connect an Ethernet wire or establish a wifi connection between its computer and the astromecca computer. Configure the TCP transport protocol to send messages on the port 1111.

The following example gets the right ascension, the declination and the peer side of the mount via the astromecca command get radec.

```
from mountremote import MountremoteClient
remote_client = MountremoteClient("TCP", port=1111, protocol="MCS")
command = "{'req': {'get': 'radec'}}"
result = remote_client.putread_chan(command)
print("Command = {}\nResult = {}".format(command,result))
del(remote_client)
```

The most useful command is ‘exec’ which allow to execute any python command of the mount thread. By this way it is possible to drive the mount with only this command. The following example show how to call directly the method which returns the radec coordinates:

```
from mountremote import MountremoteClient
remote_client = MountremoteClient("TCP", port=1111, protocol="MCS")
command = "{'req': {'do': {'exec' , 'self.radec_coord()'}}}"
result = remote_client.putread_chan(command)
print("Command = {}\nResult = {}".format(command,result))
del(remote_client)
```

You can execute a GOTO action:

```
from mountremote import MountremoteClient
remote_client = MountremoteClient("TCP", port=1111, protocol="MCS")
command = "{'req': {'do': {'exec' , 'self.radec_goto("4h45m,"+05d18m")'}}}"
result = remote_client.putread_chan(command)
print("Command = {}\nResult = {}".format(command,result))
del(remote_client)
```

3. The MCS langage protocol

The messages are Json formated. Basically the hierarchy is defined as:

```
command = "{typemsg: {action: {cmd: val}}}"
```

The following combinations can be used:

typemsg	action	cmd	val
req	do	exec	*Python command*
req	get	radec	
req	get	hadec	

The reponse is also a Json structure for which the keys reflect those of the request (but ‘req’ is replaced by ‘res’):

```
Command = {'req': {'get': 'radec'}}
Result = {'res': {'get': {'radec': [{'ra': 93.77897552123532, 'dec': 0.007976942637754572, '
```

Note the val is a list or any other kind of type in case of (req,de,exec).

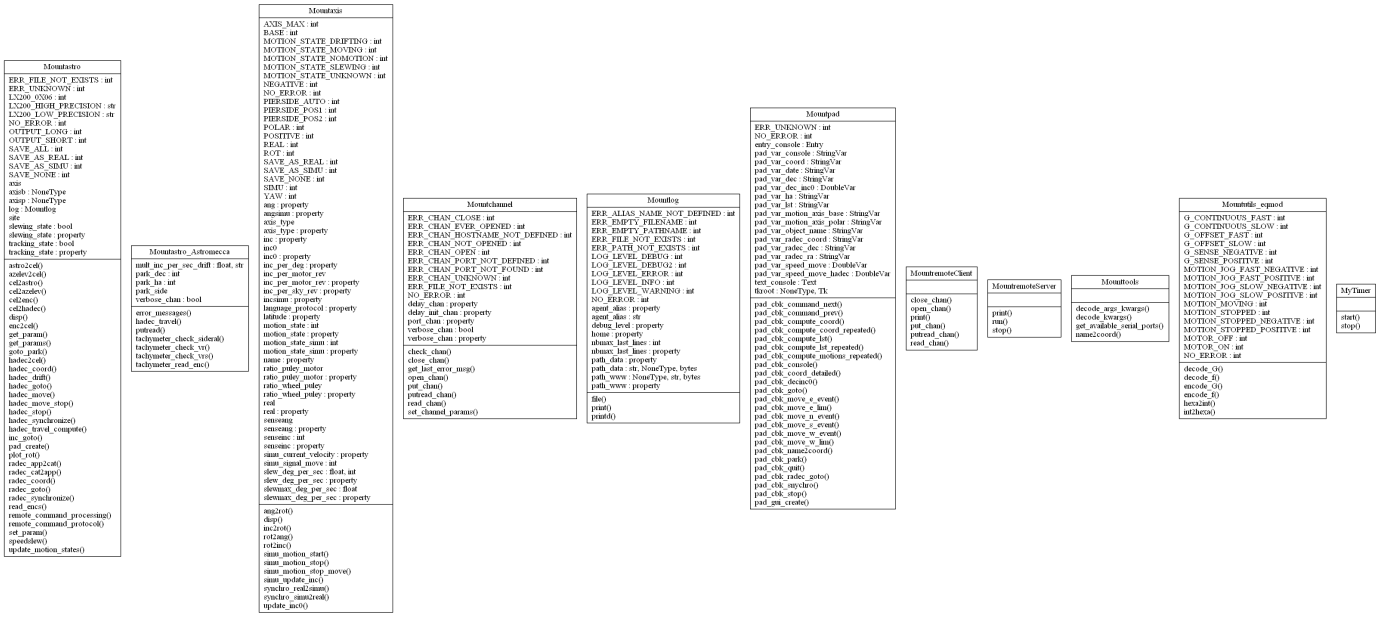
3. User guide to program with mount Python classes

Describe how to instanciate mount classes to use them in a personal Python code.

User guide to program with mount Python classes

This section show how to use Mount classes in a Python code.

1. Classes and attributes



The mountastro module provides the class Mountastro which contains almost all the methods to drive mounts. Mountastro provides methods only for simulations but all these methods call also an abstract method for real mounts. The abstracts methods are concreted in a dedicated class. For example, for AstroMECCA mounts the class Mountastro_Astromecca inherits the Mountastro methods and provides concrete methods.

2. Examples of Python codes to instanciate the mount

To include the mount classes in a python program, one must import the desired classes:

```
import celme
import mountastro_astromecca
```

Define the site where the telescope is installed:

```
home = celme.Home("GPS 2.0375 E 43.6443484725 136.9")
site = celme.Site(home)
```

The home geographic coordinates must be adjusted to the location of the mount. Instanciate the class Mountastro_Astromecca:

```
mount1 = mountastro.Mountastro_Astromecca("HADEC", name="My Mount", manufacturer="AstroMECCA
```

HADEC stands for the fact the axes of the mount are oriented according an equatorial mount. The protocol to communicate with the controller used for AstroMECCA mounts must be configured. The following lines show how to get available list of serial ports and configre it:

```
tools = mountastro.Mounttools()
available_serial_ports = tools.get_available_serial_ports()
print("Avaible serial ports {}".format(available_serial_ports))
port = "/dev/ttySC0"
mount1.set_channel_params("SERIAL", port=port, baud_rate=115200, delay_init_chan=0.1, end_of
```

The port key must be available in the list of serial ports.

3. Examples of Python codes to drive the mount

Prerequisite is to have instanciated an object (mount1) as explained in the previous section.

To get current coordinates of the mount:

```
ra, dec, side = mount1.radec_coord()
```

The side is where the tube lies. Side can be 1 or -1. If side = 1 the tube is in regular position. If side = -1 the tube is in fliped position.

To slew the mount to a target defined by J2000 astronomical coordinates:

```
mount1.radec_goto("3h5m56.45s","+3d51m34.1s")
```

The methods radec_coord and radec_goto, it is possible to add options as equinox or output formats.

Example of options to get coordinates at the equinox of the date and outputs in decimal degrees:

```
ra, dec, side = mount1.radec_coord("deg", "deg", equinox="NOW")
```

To start a move on declination axis at the rate of 2 deg/sec:

```
mount1.hadec_move(0, 2)
```

To stop a move motion:

```
mount1.hadec_move_stop()
```

To stop a any motion:

```
mount1.hadec_stop()
```

4. User guide to program with celme Python classes

Describe how to instanciate celme classes to use them in a personal Python code. Celme is a tool used by mount classes.

User guide to program with celme Python classes

This section show how to use celme in a Python code.

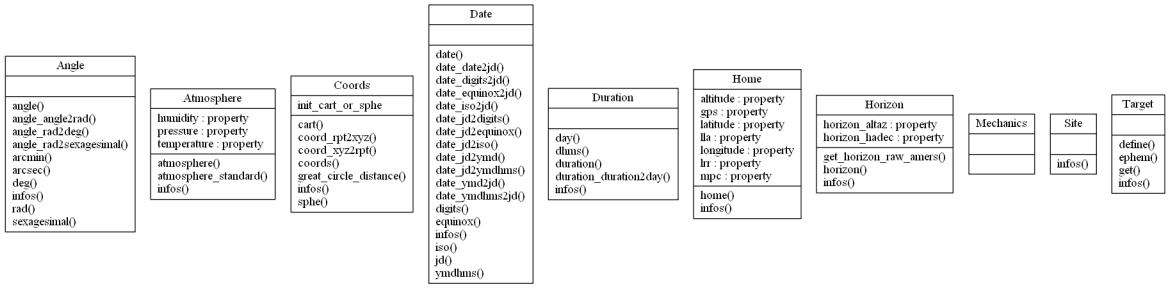
1. Context

Celestial mechanics allow to compute celestial bodies positions and manage dates, durations, angles, coordinates.

2. Python classes provided by celme

Brief descrition of the celme classes:

- Angle: Convert angles in complex formats.
- Coords: Convert angles in complex formats.
- Home: Define a mount location on the Earth surface.
- Horizon: Define horizon line to set the limits of telescope pointings.
- Atmosphere: Define atmospheric conditions to set the conditions to compute refraction effects.
- Site: Define an observation site by merging Home, Horizon, Atmosphere.
- Date: Convert dates in complex formats.
- Duration: Convert durations in complex formats.
- Mechanics: Low level methods for celestial mechanics
- Targets: High level methods for celestial mechanics



3. Using the Python module celme for angles

3.1. Simple example to convert angles

```
import celme
a = celme.Angle("2h3m27s")
print("angle deg =", a.deg())
```

angle deg = 30.862499999999994

3.2. Operation with angles

We want to compute sum of the angles 2h3m27s and -0d28m12.4s. Display the results in sexagesimal format expressed in degrees:

```
import celme
a = celme.Angle("2h3m27s")
b = celme.Angle("-0d28m12.4s")
c = a + b
print("{}".format(c.sexagesimal("d")))
```

‘30d23m32.60s’

4. Using the Python module celme for dates

Celme module is located in the folder pyros/src/utils Examples: pyros/src/utils/report/dates.py help(Date)

4.1. Simple example to convert dates

```
import celme
a = celme.Date("now")
print("Date ISO =", a.iso())
Date ISO = 2018-10-11T22:45:44.958
print("Date Julian Day =", a.jd())
```

Date Julian Day = 2458403.448437011

4.2. Operation with dates

We want to compute date corresponding to add 2 days and 28 minutes to the 2018-10-11T22:45:44.958. Display the results in ISO format:

```
import celme
a = celme.Date("2018-10-11T22:45:44.958")
b = celme.Duration("2d28m")
c = a + b
print("Date ISO =", c.iso())
```

Date ISO = 2018-10-13T23:13:44.958

5. Using the Python module celme for ephemeris

Celme module is located in the folder pyros/src/utils
Examples: pyros/src/utils/report/targets.py

```
help(Target)
```

5.1. Simple example to compute Sun coordinates

First we compute RA, DEC coordinates of the Sun:

```
import celme
date = celme.Date("now")
site = celme.Site("GPS 0 E 49 200")
skyobj= {'planet':'Sun'}
target = celme.Target()
target.define(skyobj)
output0s = ['ra','dec']
results = target.ephem(date,site,output0s)
ra = results['RA']
dec = results['DEC']
```

Second, we inject RA, DEC to compute its local coordinates:

```
skyobj= {'RA':ra , 'DEC':dec }
outputs = ['elev']
```

```
target.define(skyobj)
results = target.ephem(date,site,outputs)
elev = results['ELEV']
```

6. Angle formats

Input accepted formats:

-12.345 : Decimal degrees
-0.3421r : Decimal radians
6h3m2s : Sexagesimal HMS
3mh2s : Sexagesimal MS
2sh : Sexagesimal S
6d3m2s : Sexagesimal dms
3m2s : Sexagesimal ms
2s : Sexagesimal s (i.e. arcsec)
“6 3 2” : Sexagesimal dms
“6:3:2” : Sexagesimal dms
“6h3:2” : Sexagesimal HMS

Sexagesimal formating using uspzad convention rad is an angle in radian Sexagesimal format:

- u (unit) = h,H,d,D (default=D). Capital mean module [0:360[, lower case means module [-180:180[
- s (separator) = ” “,:,”” (default=”” means letters hms or dms)
- p (plus/minus) = +,”” (default=””)
- z (zeros) = 0,”” (default=””)
- a (angle_limits) = “”,90, (+/-limit if unit D,H, default=”” means 360)
- d (sec_digits) = “”,”.1”,”.2”,... (default=””)

Style 1:

- To Display a R.A.: “H0.2” => 23h07m42.49s
- To Display a Decl.: “d+090.1” => +00d34m22.6s
- To Display a H.A.: “h0.2” => -08h43m16.05s

Style 2:

- To Display a R.A.: “H 0.2” => 23 07 42.49
- To Display a Decl.: “d +090.1” => -00 34 22.6
- To Display a H.A.: “h 0.2” => -08 43 16.05

Style 3:

- To Display a R.A.: “H:0.2” => 23:07:42.49
- To Display a Decl.: “d:+090.1” => -00:34:22.6
- To Display a H.A.: “h:0.2” => -08:43:16.05

7. Date formats

Input accepted formats:

- now = Now. e.g. “now”
- jd = Julian day. e.g. 24504527.45678
- iso = ISO 8601. e.g. 2018-02-28T12:34:55.23
- sql = ISO 8601. e.g. 2018-02-28 12:34:55.23

- ymdhms = Calendar. e.g. 2018 2 28 12 34 55.23
- equinox = Equinox. e.g. J2000,0
- digits = Pure digits e.g. 20180228123455.23

5. Class and method documentation

For developers of Python code in mount classes.

Generate this documentation

This documentation is generated by the use of Sphinx and Pyreverse. Sphinx uses the Restructured Text format. The following links gives some informations about the syntax:

- Tutorial Sphinx 1.
- Tutorial Sphinx 2.
- Tutorial documentation.
- Tutorial reST.
- Tutorial Napoleon extension.

Installation of Sphinx and Pyreverse

Procedure for Linux (root privileges):

```
sudo pip3 install sphinx
sudo pip3 install rst2pdf
sudo pip3 install pylint
sudo pip3 install sphinx_pyreverse
sudo pip3 install graphviz
sudo apt-get install graphviz graphviz-dev
sudo pip3 install pygraphviz
```

Procedure for Windows (user xxx). First you must download and install Graphviz. Then:

```
cd c:\Users\xxx\Anaconda3\Scripts
.\pip install -U Sphinx
.\pip install rst2pdf
.\conda install pyreverse
.\conda install python-graphviz
```

Generate the documentation

To generate the documentation you must start to execute pyreverse followed by sphinx.

Procedure pyreverse for Linux:

```
cd /home/pi/astromecca/doc_rst/doc_pyreverse
pyreverse3 -p mount -o png --ignore=celme ../../mountastro
pyreverse3 -p celme -o png --ignore=mountastro ../../celme
cp classes_*.png ../doc_images
```

Procedure sphinx for Linux:

```
cd /home/pi/astromecca/doc_rst
sphinx-build -b html . ./../doc_html
sphinx-build -b pdf . ./../doc_pdf
```

Procedure pyreverse for Windows (user xxx):

```
cd astromecca\doc_rst\doc_pyreverse
c:\Users\xxx\Anaconda3\Scripts\pyreverse -p mount -o png --ignore=celme ../../mountastro
```

```
c:\Users\xxx\Anaconda3\Scripts\pyreverse -p celme -o png --ignore=mountastro ../../celme
copy classes_*.png ..\doc_images
```

Procedure sphinx for Windows (user xxx):

```
cd astromecca\doc_rst
C:\Users\xxx\Anaconda3\Scripts\sphinx-build -b html . ..\..\doc_html
C:\Users\xxx\Anaconda3\Scripts\sphinx-build -b pdf . ..\..\doc_pdf
```

The classes of Mounts for astronomy.

Classes for driving mounts

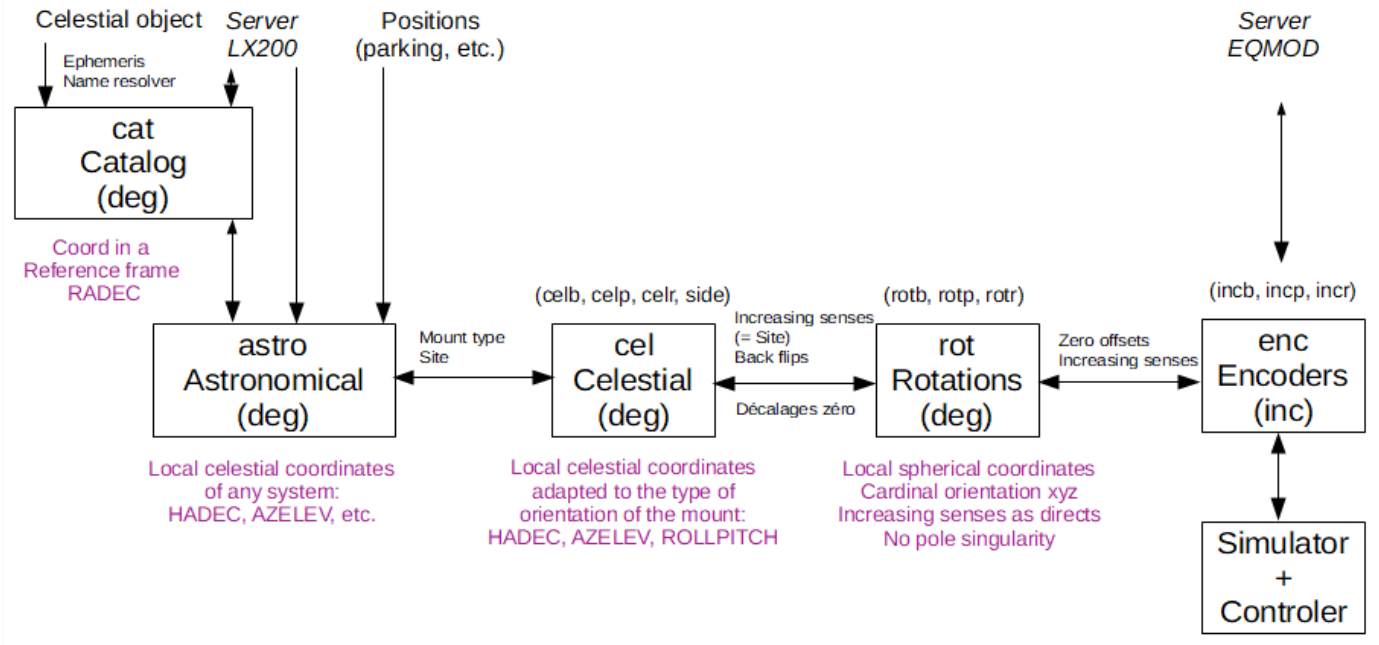
This section is the documentation for developpers of classes related to the astronomical mounts.

This page is generated according the docstrings of the python code. It displays only public methods of classes.

Details of angle definitions are given in the following links:

Definition of the rotation system of a mount

Relation between celestial coordinates and the encoders of the mount



The **enc** (encoder) system is defined by motor encoders of a telescope mount. The usual unit is inc (increment).

The **cel** (celestial) system is defined by local apparent celestial coordinates. The usual unit is deg (degrees).

The choice of the local apparent celestial coordinates type (ha, dec) or (az, elev) depends on the type of the mount (equatorial, altaz).

Note that astronomical coordinates **astro** (ra, dec, equinoxe) of celestial object catalogs are not locals but the conversion into **cel** system is univoque using celestial mechanic relations.

Many difficulties exist to convert systems **enc** to **cel**:

- The conversion **enc** to **cel** is not bijective because a pointing direction in the sky **cel** can be realized by two different **enc** obtained by a back flip of the mount. For example, (ha,dec) = (45, 50) corresponds to the same direction than (ha,dec) = (225,130).
- The zero points of **enc** may not coorespond to zero points of **cel**.
- The increasing angles of **enc** may not correspond to the increasing sense of **cel**.

We must simplfy the conversion between **enc** and **cel** introducting an intermediate universal system which allows to decouple the problems of senses and zero points of encoders with the back flips of the mount.

The conversion of **enc** to **cel** will be realized using the **rot** (rotation) system.

Definition of the angles rotb, rotp

The system of **rotb,rotp** angles is linked to the machanics of the mount axes. The goal is to have a local spherical coordinate system which have no ambiguity and easily recognizable on any type of mounts.

rotb,rotp are spherical coordinates according the trihedron rx,ry,rz:

- The rz axis is directed toward the mechanical roation pole which is above horizon.
- The (rx,0,ry) plan turns around the rz axis according the **rotb** (basis angle).
- The rx axis is perpendicular to the rz axis and directed toward the hihest elevation (meridian).
- The ry axis is defined by the direct trihedron with rx and rz.
- The (rx,0,rz) turns around the ry axis according the angle **rotp** (polar angle).

The origin of **rotb,rotp**:

- The angle rotp=0 is at the visible pole from the observation site (rx,ry,rz) = (0,0,1).
- The angle rotb=0 is at the meridian with (rx,ry,rz) = (1,0,0).

The sense of **rotb,rotp**:

- The increasing sense of rotp is direct in the (ry,0,rz) plane observed from the end of the ry axis.
- The increasing sense of rotb is direct in the (rx,0,ry) plane observed from the end of the rz axis.

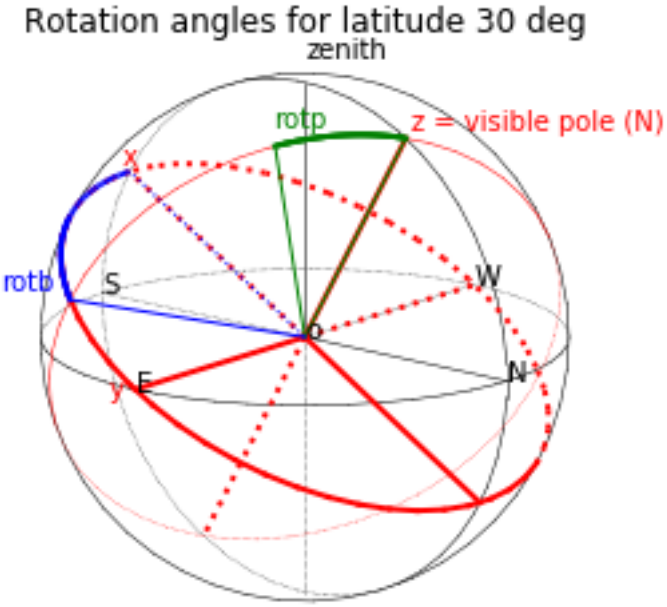
Attention, the pointed direction (rotb, rotp) may not correspond to (ha, dec) because the optical tube could be offseted by an angle according the zero points of rotb, rotp.

The back flip is univokely identified by the sign of rotp. It is the main advantage of the rot system:

- side = 1 if rotp>=0
- side = -1 if rotp<0

Relations between (rotb, rotp) and (celb, celp)

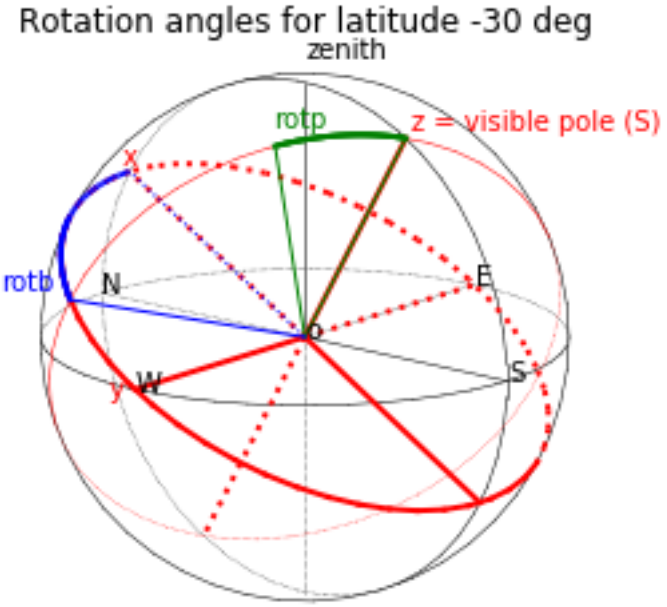
Northern hemisphere



Relations:

- Hour angle = celb = -rotb
- Declination = celp = 90-abs(rotp)
- Side = sign of rotp

Southern hemisphere



Relations:

- Hour angle = celb = rotb

- Declination = celp = abs(rotp)-90
- Side = sign of rotp

Module mountastro

mountastro.mountastro

class mountastro.mountastro.**Mountastro** (*args, **kwargs)

Class to define a mount.
The first element of args is a string to define the mount type amongst:

- HADEC: Eqquatorial mount.
- HADECROT: Eqquatorial mount + field rotator.
- AZELEV: Altazimutal mount.
- AZELEVROT: Altazimutal mount + field rotator.

Dictionnary of motion parameters are:

- NAME: A string to identify the mount in memory.
- LABEL: A string to identify the mount for prints.

Example:

```
>>> mymount = Mountastro("HADEC", name = "My telescope")
```

A mount is constituted of many axes instanciated automatically using the Mountaxis class.
The class Mountastro provides methods to convert astronomical coordinates into low level orders for controlers (i.e. increments). For example:

- radec_coord(): To read the current position of the mount axes.
- radec_synchronize(): To synchronize the current position of the mount axes.
- radec_goto(): To slew the mount axes to a target position.

These methods are implemented only in simulation mode in the Mountastro class. All these mods call a second one prefixed by _my_ to communicate physically with the controller. For example, radec_coord() calls _my_radec_coord(). In the class Mountastro the methos _my_* are only abstracts. The concrete methods will be defined in another class specific to the langage of the controller. For exemple the AstroMECCA commands are written in the class Mount_Astromecca which inherites metohds of the Mountastro class.

Example:

```
>>> mymount = Mountastro_Astromecca("HADEC", name = "My telescope")
>>> mymount.set_channel_params("SERIAL", port="COM1", baud_rate=115200)
>>> ra, dec, pierside = mymount.radec_coords()
```

_get_slewing_state () → bool

True if telescope is currently moving in response to one of the Slew methods or the MoveAxis(TelescopeAxes, Double) method, False at all other times.
Reading the property will raise an error if the value is unavailable. If the telescope is not capable of asynchronous slewing, this property will always be False. The definition of “slewing” excludes motion caused by sidereal tracking, PulseGuide, RightAscensionRate, and DeclinationRate. It reflects only motion caused by one of the Slew commands, flipping caused by changing the SideOfPier property, or MoveAxis(TelescopeAxes, Double).

_get_tracking_state () → bool

The state of the telescope’s sidereal tracking drive.

Tracking Read must be implemented and must not throw a PropertyNotImplementedException. Tracking Write can throw a PropertyNotImplementedException. Changing the value of this property will turn the sidereal drive on and off. However, some telescopes may not support changing the value of this property and thus may not support turning tracking on and off. See the CanSetTracking property.

_langage_mcs_req_decode (command: str**) →** tuple
Format of the request messages: message = "{typemsg: {action: {cmd: val}}}"
typemsg, action, cmd, val = self._langage_mcs_req_decode(command)

_my_read_encs (incsimus: list**) →** list
Read the real raw increment values of the axes

Parameters: **incsimus** (*list*) – List of simulated increments.
Returns: List of real increments.
Return type: list

Inputs are simulated increments. Outputs are real increments if a real mount exists. This is an abstract method. Please overload it according your language protocol to read real increments. This conversion method is at level 1/4.

_set_slewing_state (state: bool**)**
True if telescope is moving in response to one of the Slew methods or the MoveAxis(TelescopeAxes, Double) method, False at all other times.
Internal for the Python code

_set_tracking_state (state: bool**)**
The state of the telescope's sidereal tracking drive.
See details in the documentation of _get_tracking_state()

astro2cel (astro_type: str**,** base: celme.angles.Angle**,** polar: celme.angles.Angle**,** base_deg_per_sec: str = ''**,** polar_deg_per_sec: str = ''**) →** tuple
Convert any astronomical coordinate system (astro_type) into celestial coordinates and the corresponding velocities.

astro_type can be "HADEC" or "AZELEV":

- base = ha or az (according astro_type)
- polar = dec or elev (according astro_type)
- base_deg_per_sec = dha or daz (according astro_type)
- polar_deg_per_sec = ddec or delev (according astro_type)

Parameters:
- **astro_type** (*str*) – Specification of the astronomical coordinate system.
- **base** (*celme.Angle*) – Hour angle or azimuth (unit defined in celme.Angle).
- **polar** (*celme.Angle*) – Declination or elevation (unit defined in celme.Angle).
- **base_deg_per_sec** (*celme.Angle*) – Declination or elevation (unit defined in celme.Angle).
- **base_deg_per_sec** – Declination or elevation (unit defined in celme.Angle).

Returns: tuple of celestial coordinates (basis, polar, rotation) (degrees) and their derivatives (deg/sec)
Return type: tuple of 6 elements

This conversion method is at level 3/4.

azelev2cel (az: celme.angles.Angle**,** elev: celme.angles.Angle**) →** tuple
Convert (azimuth, elevation) coordinates into celestial coordinates.

Parameters:
- **ha** (*celme.Angle*) – Azimuth (unit defined in celme.Angle).
- **dec** (*celme.Angle*) – Elevation (unit defined in celme.Angle).

Returns: tuple of celestial coordinates (basis, polar, rotation) (degrees)

Return type: tuple of 3 elements

This conversion method is at level 3/4.

cel2astro (celb: float**,** celp: float**,** unit_ha: str = ''**,** unit_dec: str = ''**,** unit_az: str = ''**,** unit_elev: str = ''**) →** tuple

Convert celestial coordinates into (H.A., Dec) (azimuth, elevation, rotation) astronomical coordinates
It is possible to choice the unit of the outputs for each astronomical coordinate.

Parameters:
- **celb** (*float*) – Celestial base coordinate (degrees).
- **celp** (*float*) – Celestial polar coordinate (degrees).
- **unit_ha** (*str*) – Unit of the Hour angle (unit defined in celme.Angle).
- **unit_dec** (*str*) – Unit of the déclination (unit defined in celme.Angle).
- **unit_az** (*str*) – Unit of the azimuth (unit defined in celme.Angle).
- **unit_elelv** – Unit of the elevation (unit defined in celme.Angle).

Returns: tuple of celestial coordinates (ha, dec, az, elev, rotation) (degrees)

Return type: tuple of 5 elements

This conversion method is at level 3/4.

cel2azelev (celb: float**,** celp: float**,** unit_az: str = ''**,** unit_elev: str = ''**) →** tuple

Convert celestial coordinates into (azimuth, elevation, rotation) astronomical coordinates
It is possible to choice the unit of the outputs for each astronomical coordinate.

Parameters:
- **celb** (*float*) – Celestial base coordinate (degrees).
- **celp** (*float*) – Celestial polar coordinate (degrees).
- **unit_az** (*str*) – Unit of the azimuth (unit defined in celme.Angle).
- **unit_elelv** – Unit of the elevation (unit defined in celme.Angle).

Returns: tuple of celestial coordinates (az, elev, rotation) (degrees)

Return type: tuple of 3 elements

This conversion method is at level 3/4.

cel2enc (celb: float**,** celp: float**,** pierside: int**,** output_format=0**,** save=0**) →** tuple

Convert celestial apparent coordinates into encoder values.

Parameters:
- **celb** (*float*) – Celestial coordinate of the base axis (degrees).
- **celp** (*float*) – Celestial coordinate of the polar axis (degrees).
- **pierside** (*int*) – Pier side: Mountaxis().PIERSIDE_AUTO or Mountaxis().PIERSIDE_POS1 or Mountaxis().PIERSIDE_POS2.
- **output_format** (*int*) – OUTPUT_SHORT or OUTPUT_LONG.
- **save** (*int*) – SAVE_NONE or SAVE_AS_SIMU or SAVE_AS_REAL or SAVE_ALL.

Returns: Increment coordinates

Return type: tuple of 2 or 14 elements

Input = celb, celp, pier side (deg units) Output = Raw encoder values (inc)
This conversion method is at level 2/4.

cel2hadec (celb: float**,** celp: float**,** unit_ha: str = ''**,** unit_dec: str = ''**) →** tuple

Convert celestial coordinates into (H.A., Dec) astronomical coordinates
It is possible to choice the unit of the outputs for each astronomical coordinate.

Parameters:
- **celb** (*float*) – Celestial base coordinate (degrees).
- **celp** (*float*) – Celestial polar coordinate (degrees).
- **unit_ha** (*str*) – Unit of the Hour angle (unit defined in celme.Angle).
- **unit_dec** (*str*) – Unit of the déclination (unit defined in celme.Angle).

Returns: tuple of celestial coordinates (ha, dec) (degrees)

Return type: tuple of 2 elements

This conversion method is at level 3/4.

disp ()

Display a summary of the Mount parameters, the current values of encoders and celestial angles. This method is used to check and debug the mount.

enc2cel (simulation_incs: list**,** output_format=0**,** save=0**) →** tuple

Read encoder values and convert them into celestial apparent coordinates

Parameters:
- **simulation_incs** (*list*) – List of simulated increments.
- **output_format** (*int*) – OUTPUT_SHORT or OUTPUT_LONG.
- **save** (*int*) – SAVE_NONE or SAVE_AS_SIMU or SAVE_AS_REAL or SAVE_ALL.

Returns: Celestial coordinates (degrees)

Return type: tuple of 3 or 14 elements

Input = Raw encoder values (inc) Output = HA, Dec, pier side (any celme Angle units)
This conversion method is at level 2/4.

hadec2cel (ha: celme.angles.Angle**,** dec: celme.angles.Angle**) →** tuple

Convert (H.A,Dec) coordinates into celestial coordinates.

Parameters:
- **ha** (*celme.Angle*) – Hour angle (unit defined in celme.Angle).
- **dec** (*celme.Angle*) – Declination (unit defined in celme.Angle).

Returns: tuple of celestial coordinates (basis, polar, rotation) (degrees)

Return type: tuple of 3 elements

This conversion method is at level 3/4.

hadec_coord (**kwargs: dict**) →** tuple

Read the current astronomical coordinates.

Parameters: **kwargs** (*dict*) – Parameters for output units.

Returns: tuple of (ha, dec, pierside)

Return type: tuple

Dictionnary of unit parameters are:

- UNIT_HA: A string (uspzad format).
- UNIT_DEC: A string (uspzad format).

Example:

```
>>> mymount = Mountastro("HADEC", name = "My telescope")
>>> mymount.hadec_coords("H0.2", "d+090.1")
```

hadec_drift (hadec_speeddrift_ha_deg_per_sec: float**,** hadec_speeddrift_dec_deg_per_sec: float**)**

Drift the mount

Parameters:
- **hadec_speeddrift_ha_deg_per_sec** (*float*) – Hour angle drift (deg/sec)
- **hadec_speeddrift_dec_deg_per_sec** (*float*) – Declination drift (deg/sec)

Example:

```
>>> mymount = Mountastro("HADEC", name = "My telescope")
>>> mymount.hadec_drift(0.0,-0.05)
```

hadec_goto (ha: celme.angles.Angle, dec: celme.angles.Angle, **kwargs)

Slew the mount to a target defined by astronomical coordinates.

Parameters:
- **ha** (*celme.Angle*) – Hour angle target (unit defined in celme.Angle).
- **dec** (*celme.Angle*) – Declination target (unit defined in celme.Angle).
- **kwargs** (*dict*) – Parameters for pointing.

Dictionnary of pointing parameters are:

- BLOCKING: A boolean to block the programm until the mount is arrived (False by default).
- SIDE: Integer to indicate the back flip action:
  - PIERSIDE_AUTO (=0) back flip is performed if needed
  - PIERSIDE_POS1 (=1) pointing in normal position
  - PIERSIDE_POS2 (=-1) pointing in back flip position

Example:

```
>>> mymount = Mountastro("HADEC", name = "My telescope")
>>> mymount.hadec_goto("12h28m47s", "+5d45m28s", side = Mountaxis().PIERSIDE_AUTO)
```

hadec_move (ha_move_deg_per_sec, dec_move_deg_per_sec) → tuple

Slew or modify the drift of the mount.

Parameters:
- **ha_drift_deg_per_sec** (*float*) – Hour angle velocity (deg/s).
- **dec_drift_deg_per_sec** (*float*) – Declination velocity (deg/s).

Returns: tuple of (error code, result)

Return type: tuple

Example:

```
>>> mymount = Mountastro("HADEC", name = "My telescope")
>>> mymount.hadec_move(0.1,-0.2)
```

hadec_move_stop ()

Stop the mount motion.

Example:

```
>>> mymount = Mountastro("HADEC", name = "My telescope")
>>> mymount.hadec_move_stop()
```

hadec_stop ()

Stop any motion of the mount.

Returns: tuple of (error code, result)

Return type: tuple

Example:

```
>>> mymount = Mountastro("HADEC", name = "My telescope")
>>> mymount.hadec_stop()
```

hadec_synchronize (ha: celme.angles.Angle, dec: celme.angles.Angle, pierside: int = '') → tuple

Synchronize the encoders of the current position with the given astronomical coordinates.

Parameters:
- **ha** (*celme.Angle*) – Hour angle (unit defined in celme.Angle).
- **dec** (*celme.Angle*) – Declination (unit defined in celme.Angle).
- **pierside** (*int*) – Mountaxis().PIERSIDE_AUTO Mountaxis().PIERSIDE_POS1 Mountaxis().PIERSIDE_POS2.

Returns: tuple of astronomical coordinates (ha, dec, pierside)

Return type: tuple of 3 elements

Example:

```
>>> mymount = Mountastro("HADEC", name = "My telescope")
>>> mymount.hadec_synchronize("12h28m47s", "+5d45m28s", Mountaxis().PIERSIDE_POS1)
```

hadec_travel_compute (ha_target: celme.angles.Angle, dec_target: celme.angles.Angle, pierside_target: int = 0) → tuple

Compute the duration of a slewing from the current position to a target.

Parameters:
- **ha_target** (*celme.Angle*) – Hour angle (unit defined in celme.Angle).
- **dec_target** (*celme.Angle*) – Declination (unit defined in celme.Angle).
- **pierside_target** (*int*) – Mountaxis().PIERSIDE_AUTO or Mountaxis().PIERSIDE_POS1 or Mountaxis().PIERSIDE_POS2.

Returns: tuple of celestial coordinates (elevmin, ts, elevs)

Return type: tuple of 3 elements

Returned values are:
- elevmin: Minimum elevation (degrees).
- ts: List of time from the start of slewing (sec).
- elevs: List of computed elevations for each element of ts (degrees).

This conversion method is at level 4/4.

plot_rot (lati, azim, elev, rotb, rotp, outfile='')

Vizualize the rotation angles of the mount according the local coordinates # — Site latitude lati (deg) : Site latitude # — Observer view elev = 15 # turn around the X axis azim = 140 # turn around the Z axis (azim=0 means W foreground, azim=90 means N foreground) # — rob, rotp

radec_coord (**kwargs)

Read the current astronomical coordinates.

Parameters: **kwargs** (*dict*) – Parameters for output units.

Returns: tuple of (ra, dec, pierside)

Return type: tuple

Dictionnary of unit parameters are:
- EQUINOX: Equinox of the ra,dec coordinates (J2000 by default).
- UNIT_RA: A string (uspzad format).
- UNIT_DEC: A string (uspzad format).

Example:

```
>>> mymount = Mountastro("HADEC", name = "My telescope")
>>> mymount.radec_coords("H0.2", "d+090.1")
```

radec_goto (ra_angle: celme.angles.Angle, dec_angle: celme.angles.Angle, **kwargs)

Slew the mount to a target defined by astronomical coordinates.

Parameters:
- **ha_angle** (*celme.Angle*) – Right ascension target (unit defined in celme.Angle).
- **dec_angle** (*celme.Angle*) – Declination target (unit defined in celme.Angle).
- **kwargs** (*dict*) – Parameters for pointing.

Returns: tuple of (error code, result)

Return type: tuple

Dictionnary of pointing parameters are:

- EQUINOX: Equinox of the ra,dec coordinates (J2000 by default).
- BLOCKING: A boolean to block the programm until the mount is arrived (False by default).
- SIDE: Integer to indicate the back flip action:
  - PIERSIDE_AUTO (=0) back flip is performed if needed
  - PIERSIDE_POS1 (=1) pointing in normal position
  - PIERSIDE_POS2 (=-1) pointing in back flip position

Example:

```
>>> mymount = Mountastro("HADEC", name = "My telescope")
>>> mymount.radec_goto("12h28m47s", "+5d45m28s", side = Mountaxis().PIERSIDE_AUTO)
```

radec_synchronize (ra_angle: celme.angles.Angle, dec_angle: celme.angles.Angle, **kwargs) → tuple

Synchronize the encoders of the current position with the given astronomical coordinates.

Parameters:
- **ra_angle** (*celme.Angle*) – Right ascension (unit defined in celme.Angle).
- **dec_angle** (*celme.Angle*) – Declination (unit defined in celme.Angle).
- **pierside** (*int*) – Mountaxis().PIERSIDE_AUTO Mountaxis().PIERSIDE_POS1 Mountaxis().PIERSIDE_POS2.
- **kwargs** (*dict*) – Pointing parameters

Returns: tuple of astronomical coordinates (ra, dec, pierside)

Return type: tuple of 3 elements

Dictionnary of pointing parameters are:

- EQUINOX: Equinox of the ra,dec coordinates (J2000 by default).
- SIDE: Integer to indicate the back flip action:
  - PIERSIDE_AUTO (=0) back flip is performed if needed
  - PIERSIDE_POS1 (=1) pointing in normal position
  - PIERSIDE_POS2 (=-1) pointing in back flip position

Example:

```
>>> mymount = Mountastro("HADEC", name = "My telescope")
>>> mymount.radec_synchronize("12h28m47s", "+5d45m28s", Mountaxis().PIERSIDE_POS1)
```

read_encs (simulation_incs: list**) → list**
Read the simulated and real raw increment values of the axes

Parameters: **incsimus** (*list*) – List of simulated increments.
Returns: List of real increments.
Return type: list

For the simulation:

- if simulation_incs==”” the value is calculated from simu_update_inc
- if simulation_incs==34.5 the value is taken equal to 34.5

For real:

- if the axis is not real then real=simulated value
- else the encoder is read

Output = Raw calues of encoders (inc)
This conversion method is at level 1/4.

remote_command_processing (command**)**
Execute a command when this code is used as server.

Parameters: **command** (*str*) – Command to execute
Returns: error code
Return type: int
Example:

```
>>> mymount = Mountastro("HADEC", name = "My telescope")
>>> mymount.remote_command_protocol("LX200")
>>> mymount.remote_command_processing(":GD")
```

remote_command_protocol (remote_command_protocol='LX200'**)**
Set the langage protocol when this code is used as server.

Parameters: **remote_command_protocol** (*str*) – Command to execute
Example:

```
>>> mymount = Mountastro("HADEC", name = "My telescope")
>>> mymount.remote_command_protocol("LX200")
```

***property* slewing_state**
True if telescope is currently moving in response to one of the Slew methods or the MoveAxis(TelescopeAxes, Double) method, False at all other times.
Reading the property will raise an error if the value is unavailable. If the telescope is not capable of asynchronous slewing, this property will always be False. The definition of “slewing” excludes motion caused by sidereal tracking, PulseGuide, RightAscensionRate, and DeclinationRate. It reflects only motion caused by one of the Slew commands, flipping caused by changing the SideOfPier property, or MoveAxis(TelescopeAxes, Double).

speedslew (*args**) → tuple**
Update the speed slewing velocities (deg/sec).
The order of velocities must be respected.

Returns: tuple of velocities
Return type: tuple of 1 to 3 elements

This conversion method is at level 2/4.

***property* tracking_state**
The state of the telescope’s sidereal tracking drive.

Tracking Read must be implemented and must not throw a PropertyNotImplementedException. Tracking Write can throw a PropertyNotImplementedException. Changing the value of this property will turn the sidereal drive on and off. However, some telescopes may not support changing the value of this property and thus may not support turning tracking on and off. See the CanSetTracking property.

update_motion_states ()

Get the current motions states of the axes

Returns: slewing_state, tracking_state, list of axes_motion_states

Return type: tuple

slewing_state and tracking_state are booleans. These states are computer according the combination of all active axes (real and simulated ones). axes_motion_states is a list of states for each axis combining real and simulated.

mountastro.mountaxis

class mountastro.mountaxis.**Mountaxis** (*args, **kwargs)

Class to define an axis of a mount.
The first element of args is a string to define the axis type amongst:

- HA: Hour angle axis of an eqquatorial mount.
- DEC: Declination axis of an eqquatorial mount.
- AZ: Azimuth axis of an altaz mount.
- ELEV: Elevation axis of an altaz mount.
- ROT: Paralactic roation axis of an altaz mount.
- ROLL: Roll axis of an altalt mount.
- PITCH: Pitch axis of an altalt mount.
- YAW: Yaw axis of an altalt mount.

Dictionnary of motion parameters are:

- NAME: A string to identify the axis in memory.
- LABEL: A string to identify the axis for prints.

Example:

```
>>> axisb = Mountaxis("HA", name = "Hour angle", label= "H.A.")
>>> axisp = Mountaxis("DEC", name = "Declination", label= "Dec.")
```

A mount axis is defined by a frame called ‘rot’ constituted by two perpendicular axes:

- Axisp: The polar axis, a great circle passing by the poles
- Axisb: The base axis. Its axis is parallel to the pole direction

The natural unit of ‘rot’ is degree.
The definition of the ‘rot’ frame is compatible with equatorial and altaz or altalt mounts:

- hadec : Equatorial, axisp = declination (dec), axisb = hour angle (ha)
- altaz : Altaz , axisp = elevation (alt) , axisb = azimuth (az)
- altalt : Altalt , axisp = pitch (pit) , axisb = roll (rol)

The definition of ‘rot=0’ depends on the mount types:

- Axisp: ‘rot=0’ at the pole direction upper horizon
- Axisb: ‘rot=0’ depdns on the mount type (meridian, south, etc)

The encoders provide another frame called ‘enc’ which shares the same rotational axis than ‘rot’.
The natural unit of ‘enc’ is increments.

The celestial coordinate system provide another frame called ‘cel’ which shares the same rotational axis than ‘rot’.
The natural unit of ‘cel’ is degree.
The ‘cel’ frame if fixed to the ‘rot’ frame. The zeroes are fixed by definition (see above).
The ‘enc’ frame is considered as absolute encoders. As it is impossible to place the inc=0 of the encoder exactly on the rot=0, we define a inc0 = inc when rot=0. As a consequence, for a linear response encoder:

rot = (inc-inc0) * deg_per_inc

However, a rotational sense (senseinc) is indroduced to take accound the increasing increments are in the increasing angles of ‘rot’ or not:

rot = (inc-inc0) * deg_per_inc * senseinc

deg_per_inc is always positive.

_get_ang () → int
Get the arrival angle of a calculated movement for a target.
The orientation of the coordinate system are orthonormal. (Right hand rules, tom pouce for visible polar axis !)

Returns: Error if value is not a real.
Return type: int

_get_angsimu () → int
In simulation mode, get the arrival angle of a calculated movement for a target.
The orientation of the coordinate system are orthonormal. (Right hand rules, tom pouce for visible polar axis !)

Returns: Error if value is not a real.
Return type: int

_get_axis_type () → int
Get type and mechanical position of an axis on the mount.

- BASE : Azimut or hour angle axis,
- POLAR : Elevation or declination axix,
- ROT : Derotator system for non equatorial mount (if equiped),
- YAW : Equivalent to secondary azymtuh base (for Alt-Alt mount).

Returns: BASE = 0, POLAR = 1, ROT = 2, YAW = 3
Return type: int

_get_inc () → float
Get the value for actual increments position of an axis, direct interogation of the controller.

Returns: Number of increments (for example : 37265)
Return type: float

_get_inc0 () → float
Get the value of increments for “rot=0”.

Returns: Number of increments for “rot=0” (for example : 1800)
Return type: float

_get_inc_per_deg () → float
Get the number of increments for a single degrees on the sky.

Returns: Number of increments (for example : env 970000)
Return type: float

_get_inc_per_motor_rev () → float
Get the number of increments for a single turn of the motor.

Returns: Number of increments for a single turn of the motor (for example : 1000).
Return type: float

_get_inc_per_sky_rev () → float
Get the number of increments for a single complete turn on the sky.

Returns: Number of increments.
Return type: float

_get_incsimu () → float
Get the value for actual increments position of an axis, direct interogation of the controller. Value are real if axle is real.

Returns: Number of increments in simulation mode (for example : 37265)
Return type: float

_get_language_protocol () → str
Get the type of controller language protocol for an axis (for example : SCX 11 type, or another).

Returns: Type of controller language.
Return type: str

_get_latitude () → str
Get the latitude of the observational site. Positive for north.

Returns: Latitude of site (for example : 47,2 Degrees)
Return type: str

_get_motion_state () → int
Get the current motion state

Returns: Moton state code (0=no motion, 1=slewing, 2=drifting, 3=moving).
Return type: int

Slewiwng state is an absolute motion followed by a drift. Moving state is an infinite motion. If a Moving is stopped we retreive the Drift state.

_get_motion_state_simu () → int
Get the current motion state for simulation

Returns: Moton state code (0=no motion, 1=slewing, 2=drifting, 3=moving).
Return type: int

Slewiwng state is an absolute motion followed by a drift. Moving state is an infinite motion. If a Moving is stopped we retreive the Drift state.

_get_name () → str
Get the nickname of the axis.

Returns: Nickname of the axis (for example : Declination, ...)
Return type: str

_get_ratio_puley_motor () → float
Get the ratio between pulley and motor.

Returns: Ratio between pulley and motor (for example : 100).
Return type: float

_get_ratio_wheel_puley () → float
Get the ratio between wheel and motor puley, in diameter.

Returns: Ratio between wheel and motor puley (for example : 5.25)

Return type: float

_get_real () → bool

Get the axis mode, real or simulation.

Returns: True or False

Return type: bool

_get_senseang () → int

If progression of mechanical angles referentiel are positive and progression of rot0 are positive, ‘set_senseang’ are positive. However, ‘set_senseang’ are negative when progression are inverse.
The sense depend of the orientation of celestial coordinates systems and mechanical coordinates systems.
The orientation of the coordinate system are orthonormal. (Right hand rules, tom pouce for visible polar axis !)

Returns: Error if value is not a real.

Return type: int

_get_senseinc () → int

If progression of increments are positive and progression of ‘rot0’ are positive, ‘senseinc’ are positive. However, ‘senseinc’ are negative when progression are inverse.
The sense depend of the physical rolling sense of motor cable system.

Returns: Value sense are “-1” or “1”.

Return type: int

_get_simu_current_velocity () → int

Get the final cruising speed during the motion. Motion are celestial slewing speed or any other, like goto for example.

Returns: Terminal velocity speed for a movement in degrees / sec.

Return type: int

_get_slew_deg_per_sec () → int

Get the setting speed of a goto motion.

Returns: Speed for a goto movement in degrees / sec.

Return type: int

_get_slewmax_deg_per_sec () → float

Get the maximum speed for slew motion.
The value have a maximum, setting by a limit (_slewmax_deg_per_sec).

Returns: Maximum speed for a goto movement in degrees / sec.

Return type: float

_incr_variables () → float

Update and calculus of two parameters :

- number of increments for a complete turn of an axis
- number of increments for single decimal degrees.

Returns: Number of increments for a complete turn of an axis and number of increments for single decimal degrees.

Return type: float

_set_ang (ang: float**) → int**

Set the arrival angle of a calculated movement for a target.
The orientation of the coordinate system are orthonormal. (Right hand rules, tom pouce for visible polar axis !)

Parameters: **ang** – Celestial angle of an axis (degrees)
Returns: Error if value is not a real.
Return type: int

_set_angsimu (ang: float**) →** int
In simulation mode, set the arrival angle of a calculated movement for a target.
The orientation of the coordinate system are orthonormal. (Right hand rules, tom pouce for visible polar axis !)

Parameters: **ang** – Celestial angle of an axis (degrees)
Returns: Error if value is not a real.
Return type: int

_set_axis_type (axis_type: str**) →** int
Set type and mechanical position of an axis on the mount.

- BASE : Azimut or hour angle axis,
- POLAR : Elevation or declination axix,
- ROT : Derotator system for non equatorial mount (if equiped),
- YAW : Equivalent to secondary azymtuh base (for Alt-Alt mount).

:param axis_type : BASE = 0, POLAR = 1, ROT = 2, YAW = 3 :returns: Error if value is not a real. :rtype: int

_set_inc (inc: float**) →** int
Set the value for actual increments position of an axis, direct interogation of the controller.

Parameters: **inc** – Value of the increments for the actual position.
Returns: Error if value is not a real.
Return type: int

_set_inc0 (inc0: float**) →** int
Set the value of increments for “rot=0”. When mount was initialized, the “inc0” are set by the fonction “update_inc0”.

Parameters: **inc0** – Value of the increments for “rot=0” (for example : 1800)
Returns: Error if increment is not positive.
Return type: int

_set_inc_per_deg (inc_per_deg: float**)**

Attention!

no setting for this attribute

Parameters: **inc_per_deg** (*float*) – Incrment per degree
Returns: Error if ratio is not positive.
Return type: int

_set_inc_per_motor_rev (nbr_inc: float**) →** int
Set the number of increments for a single turn of the motor.

Parameters: **nbr_inc** (*float*) – Number of increments for a single turn of the motor (for example : 1000).
Returns: Error if ratio is not positive.
Return type: int

_set_inc_per_sky_rev (inc_per_sky_rev: float**)**

> ***Attention!***
> no setting for this attribute

Parameters: **inc_per_sky_rev** (*float*) – Incrment per sky turn
Returns: Error if ratio is not positive.
Return type: int

_set_incsimu (inc: float**) →** int
Set the value for actual increments position of an axis in simulation mode.

Parameters: **inc** – Value of the increments for the simulated position.
Returns: Error if value is not a real.
Return type: int

_set_language_protocol (language_protocol: str**) →** int
Set the type of controller language protocol for an axis (for example : SCX 11 type, or another). :param language_protocol : Specified the protocol language type (for example : SCX11) :returns: Error if value is not a real. :rtype: int

_set_latitude (latitude_deg: float**) →** int
Set the latitude of the observational site. Positive for north.

Parameters: **latitude_deg** (*float*) – Latitude of site (for example : 47.2 Degrees)
Returns: Error if value is not a real.
Return type: int

_set_motion_state (motion_state: int**)**
Set the current motion state

Returns: Error code (0=no error).
Return type: int

_set_motion_state_simu (motion_state: int**)**
Set the current motion state for simulation

Returns: Error code (0=no error).
Return type: int

_set_name (name: str**)**

> ***Attention!***
> no setting for this attribute

The name are setted at the instanciation of the mount axis. The name of an axis can have several value :

- Declination,

- Azimuth
- Hour angle,
- Elevation,
- Rotator,
- Roll,
- Pitch,
- Yaw,

You cannot set the value cause it is an protected attribute.

Parameters: **name** (*str*) – Name of the axis
Returns: Error if ratio is not positive.
Return type: int

_set_ratio_puley_motor (ratio: float**) →** int
Set the ratio between pulley and motor, take care about the ratio of motor reducer type.

Parameters: **ratio** (*float*) – Ratio between pulley and motor (for example : 100).
Returns: Error if ratio are not strictly positive.
Return type: int

_set_ratio_wheel_puley (ratio: float**) →** int
Set the ratio between wheel and motor puley, in diameter.

Parameters: **ratio** – Ratio between wheel and puley (for exampe : 5.25)
Returns: Error if ratio are not strictly positive.
Return type: int

_set_real (real: bool**) →** int
Set the axis in real mode or simulation mode. With simulation mode, the value of the axis are given by Mountaxis simulation value.

Parameters: **real** – True or False.
Returns: Error if value is not a real.
Return type: int

_set_senseang (sense: int**) →** int
If progression of mechanical angles referentiel are positive and progression of rot0 are positive, ‘set_senseang’ are positive. However, ‘set_senseang’ are negative when progression are inverse.
The sense depend of the orientation of celestial coordinates systems and mechanical coordinates systems.
The orientation of the coordinate system are orthonormal. (Right hand rules, tom pouce for visible polar axis !)

Parameters: **sense** – Value sense are “-1” or “1”.
Returns: Error if value is not a real.
Return type: int

_set_senseinc (sense: int**) →** int
If progression of increments are positive and progression of rot0 are positive, senseinc are positive. However, senseinc are negative when progression are inverse.
The sense depend of the physical rolling sense of motor cable system.

Parameters: **sense** – Value sense are “-1” or “1”.
Returns: Error if value is not a real.
Return type: int

_set_simu_current_velocity (simu_current_velocity_deg_per_sec: float**)**

> ***Attention!***
>
> no setting for this attribute

Returns: Error if ratio is not positive.

Return type: int

_set_slew_deg_per_sec (deg_per_sec: float**) → int**

Set the setting speed for a goto motion.
The value are limited by the maximun limited speed (_slewmax_deg_per_sec)

Parameters: **deg_per_sec** (*float*) – Speed for a goto movement in degrees / sec (for example : 30).

Returns: Error if value is not a real.

Return type: int

_set_slewmax_deg_per_sec (deg_per_sec: float**) → int**

Set the maximum speed for slew motion. Set carrefully this parameter due to issue response of the mount.
The value have a maximum (for example : 30)

Parameters: **deg_per_sec** (*float*) – Speed for a slewing movement in degrees / sec (for example : 30)

Returns: Error if value is not a real.

Return type: int

***property* ang**

Get the arrival angle of a calculated movement for a target.
The orientation of the coordinate system are orthonormal. (Right hand rules, tom pouce for visible polar axis !)

Returns: Error if value is not a real.

Return type: int

ang2rot (ang: float**,** pierside: int = 1**,** save: int = 0**) → float**

Calculation rot from ang and pierside.

Parameters:

- **ang** (*float*) – Celestial angle (degrees)
- **pierside** (*int*) – Location of the optical tube against the mount pier: PIERSIDE_POS1 (=1) normal position PIERSIDE_POS2 (=-1) back flip position
- **save** (*int*) – Define how the results are stored: SAVE_NONE (=0) SAVE_AS_SIMU (=1) SAVE_AS_REAL (=2)

Returns: rot

Return type: float

The rot value is computed according rot, _latitude, _senseang and _axis_type. The save parameter allows to update the real or simu internal values of and, pierside and rot:

- SAVE_AS_SIMU: Only _angsimu, _rotsimu and _piersidesimu for simulated values are updated.
- SAVE_AS_REAL: Only _ang, _rot and _pierside for real values are updated.
- SAVE_NONE: No internal variables are updates.

Instanciation of the axis is indispensable.

Instanciation Usage:

```
>>> axisb = Mountaxis("HA", name = "Unknown")
```

Usage:

```
>>> axisb.ang2rot(-10, axisb.PIERSIDE_POS1, axisb.SAVE_NONE)
```

***property* angsimu**

In simulation mode, get the arrival angle of a calculated movement for a target.
The orientation of the coordinate system are orthonormal. (Right hand rules, tom pouce for visible polar axis !)

Returns: Error if value is not a real.

Return type: int

***property* axis_type**

Get type and mechanical position of an axis on the mount.

- BASE : Azimut or hour angle axis,
- POLAR : Elevation or declination axix,
- ROT : Derotator system for non equatorial mount (if equiped),
- YAW : Equivalent to secondary azymtuh base (for Alt-Alt mount).

Returns: BASE = 0, POLAR = 1, ROT = 2, YAW = 3

Return type: int

disp ()

Get information about an axis and print it on the console. Usefull for debug. Instanciation of the axis are indispensable. However, the mountaxis module when running, have by default axisb et axisp instancied.

Instanciation Usage:

```
>>> axisb = Mountaxis("HA", name = "Unknown")
>>> axisp = Mountaxis("DEC", name = "Unknown")
```

Usage:

```
>>> axisb.disp()
>>> axisp.disp()
```

Return table of an axis:

```
--------------------
AXIS name         = SCX11
axis_type         = HA
latitude          = 43
real hardware     = False
--------------------
ratio_wheel_puley = 5.25
ratio_puley_motor = 100.0
inc_per_motor_rev = 1000.0
--------------------
inc_per_sky_rev   = 525000.0
inc_per_deg       = 1458.3333333333333
```

```
--------------------
senseinc          = 1            : 1=positive
inc0              =          0.0 : Place mount rot at meridian and set inc0 = inc
senseang          = 1            : 1=positive
--------------------
slew_deg_per_sec  = 5.0
-------------------- SIMU INC -> ANG = HA
inc               =          0.0 : inc is read from encoder
rot               =    0.0000000 : rot = (inc - inc0) * senseinc / inc_per_deg
pierside          = 1            : pierside must be given by polar axis
ang               =    0.0000000 : ang = senseang * rot
-------------------- SIMU ANG = HA -> INC
ang               =    0.0000000 : Next target celestial angle HA
pierside          = 1            : Next target pier side (+1 or -1)
rot               =    0.0000000 : rot = -ang / senseang
inc               =          0.0 : inc = inc0 + rot * inc_per_deg / senseinc
-------------------- REAL INC -> ANG = HA
inc               =          0.0 : inc is read from encoder
rot               =    0.0000000 : rot = (inc - inc0) * senseinc / inc_per_deg
pierside          = 1            : pierside must be given by polar axis
ang               =    0.0000000 : ang = senseang * rot
-------------------- REAL ANG = HA -> INC
ang               =    0.0000000 : Next target celestial angle HA
pierside          = 1            : Next target pier side (+1 or -1)
rot               =    0.0000000 : rot = -ang / senseang
inc               =          0.0 : inc = inc0 + rot * inc_per_deg / senseinc
```

***property* inc**

Get the value for actual increments position of an axis, direct interogation of the controller.

Returns: Number of increments (for example : 37265)

Return type: float

***property* inc0**

Get the value of increments for “rot=0”.

Returns: Number of increments for “rot=0” (for example : 1800)

Return type: float

inc2rot (inc: float**,** save=0**)** → tuple

Calculation of rot and pierside from inc.

Parameters:

- **inc** (*float*) – Encoder increments (inc)
- **save** (*int*) – Define how the results are stored: SAVE_NONE (=0) SAVE_AS_SIMU (=1) SAVE_AS_REAL (=2)

Returns: Tuple of (rot, pierside)

Return type: tuple

The rot value is computed according inc, _inc0, _senseinc, _inc_per_deg and _axis_type. The save parameter allows to update the real or simu internal values of inc and rot:

- SAVE_AS_SIMU: Only _incsimu, _rotsimu and _piersidesimu for simulated values are updated.
- SAVE_AS_REAL: Only _inc, _rot and _pierside for real values are updated.
- SAVE_NONE: No internal variables are updates.

Instanciation of the axis is indispensable.

Instanciation Usage:

```
>>> axisb = Mountaxis("HA", name = "Unknown")
```

Usage:

```
>>> axisb.inc2rot(2000,axisb.SAVE_NONE)
```

***property* inc_per_deg**
Get the number of increments for a single degrees on the sky.

Returns: Number of increments (for example : env 970000)
Return type: float

***property* inc_per_motor_rev**
Get the number of increments for a single turn of the motor.

Returns: Number of increments for a single turn of the motor (for example : 1000).
Return type: float

***property* inc_per_sky_rev**
Get the number of increments for a single complete turn on the sky.

Returns: Number of increments.
Return type: float

***property* incsimu**
Get the value for actual increments position of an axis, direct interogation of the controller. Value are real if axle is real.

Returns: Number of increments in simulation mode (for example : 37265)
Return type: float

***property* language_protocol**
SCX 11 type, or another).

Returns: Type of controller language.
Return type: str
Type: Get the type of controller language protocol for an axis (for example

***property* latitude**
Get the latitude of the observational site. Positive for north.

Returns: Latitude of site (for example : 47,2 Degrees)
Return type: str

***property* motion_state**
Get the current motion state

Returns: Moton state code (0=no motion, 1=slewing, 2=drifting, 3=moving).
Return type: int

Slewiwng state is an absolute motion followed by a drift. Moving state is an infinite motion. If a Moving is stopped we retreive the Drift state.

***property* motion_state_simu**
Get the current motion state for simulation

Returns: Moton state code (0=no motion, 1=slewing, 2=drifting, 3=moving).
Return type: int

Slewiwng state is an absolute motion followed by a drift. Moving state is an infinite motion. If a Moving is stopped we retreive the Drift state.

***property* name**
Get the nickname of the axis.

Returns: Nickname of the axis (for example : Declination, ...)
Return type: str

***property* ratio_puley_motor**
Get the ratio between pulley and motor.

Returns: Ratio between pulley and motor (for example : 100).
Return type: float

***property* ratio_wheel_puley**
Get the ratio between wheel and motor puley, in diameter.

Returns: Ratio between wheel and motor puley (for example : 5.25)
Return type: float

***property* real**
Get the axis mode, real or simulation.

Returns: True or False
Return type: bool

rot2ang (rot: float**,** pierside: int**,** save: int = 0**) →** float
Calculation of ang from rot and pierside.

Parameters:
- **rot** (*float*) – Rotation angle (degrees)
- **pierside** (*int*) – Location of the optical tube against the mount pier: PIERSIDE_POS1 (=1) normal position PIERSIDE_POS2 (=-1) back flip position
- **save** (*int*) – Define how the results are stored: SAVE_NONE (=0) SAVE_AS_SIMU (=1) SAVE_AS_REAL (=2)

Returns: ang
Return type: float

The ang value is computed according rot, _latitude, _senseang and _axis_type. The save parameter allows to update the real or simu internal values of and, pierside and rot:

- SAVE_AS_SIMU: Only _angsimu, _rotsimu and _piersidesimu for simulated values are updated.
- SAVE_AS_REAL: Only _ang, _rot and _pierside for real values are updated.
- SAVE_NONE: No internal variables are updates.

Instanciation of the axis is indispensable.

Instanciation Usage:

```
>>> axisb = Mountaxis("HA", name = "Unknown")
```

Usage:

```
>>> axisb.rot2ang(10, axisb.PIERSIDE_POS1, axisb.SAVE_NONE)
```

rot2inc (rot: float**,** save: int = 0**) →** float
Calculation of inc from rot.

Parameters:
- **rot** (*float*) – Rotation angle (degrees)
- **save** (*int*) – Define how the results are stored: SAVE_NONE (=0) SAVE_AS_SIMU (=1) SAVE_AS_REAL (=2)

Returns: inc

Return type: float

The inc value is computed according rot, _inc0, _senseinc, _inc_per_deg. The inc values are calculated in the interval from -inc_per_sky_rev/2 to +inc_per_sky_rev/2. The save parameter allows to update the real or simu internal values of inc and rot:

- SAVE_AS_SIMU: Only _incsimu, _rotsimu for simulated values are updated.
- SAVE_AS_REAL: Only _inc, _rot for real values are updated.
- SAVE_NONE: No internal variables are updates.

Instanciation of the axis is indispensable.

Instanciation Usage:

```
>>> axisb = Mountaxis("HA", name = "Unknown")
```

Usage:

```
>>> axisb.rot2inc(10,axisb.SAVE_NONE)
```

***property* senseang**

If progression of mechanical angles referentiel are positive and progression of rot0 are positive, ‘set_senseang’ are positive. However, ‘set_senseang’ are negative when progression are inverse.
The sense depend of the orientation of celestial coordinates systems and mechanical coordinates systems.
The orientation of the coordinate system are orthonormal. (Right hand rules, tom pouce for visible polar axis !)

Returns: Error if value is not a real.

Return type: int

***property* senseinc**

If progression of increments are positive and progression of ‘rot0’ are positive, ‘senseinc’ are positive. However, ‘senseinc’ are negative when progression are inverse.
The sense depend of the physical rolling sense of motor cable system.

Returns: Value sense are “-1” or “1”.

Return type: int

***property* simu_current_velocity**

Get the final cruising speed during the motion. Motion are celestial slewing speed or any other, like goto for example.

Returns: Terminal velocity speed for a movement in degrees / sec.

Return type: int

simu_motion_start (*args, **kwargs**)**

Start a simulation motion.

Parameters:
- **args** (*args*) – First args is a string to define the type of motion to do.
- **kwargs** (*kwargs*) – Dictionnary of motion parameters:

Returns: _incsimu

Return type: float

Types of motion can be:

- SLEW or ABSOLUTE: Absolute position of the target position.
- MOVE or CONTINUOUS: Infinite motion.

Dictionnary of motion parameters are:

- Case motion type = SLEW or ABSOLUTE:
  - POSITION (inc or ang according the FRAME).
  - VELOCITY (deg/sec). Can be negative.
  - DRIFT (deg/sec). Can be negative.
- Case motion type = MOVE or CONTINUOUS:
  - VELOCITY (deg/sec). Can be negative.
  - DRIFT (deg/sec). Can be negative.
- For all cases of motions:
  - FRAME (str). “inc” (by default) or “ang”

Instanciation of the axis is mandatory:

Instanciation Usage:

```
>>> axisb = Mountaxis("HA", name = "Unknown")
```

Usage:

```
>>> axisb.simu_motion_start("SLEW", position=1000, velocity=100, frame='inc', drift=0)
```

simu_motion_stop ()
Stop a simulation motion.

simu_motion_stop_move ()
Stop a moving motion.

simu_update_inc ()
Calculate the current position of a simulation motion.
A simple rectangular profile is applied to velocity.

***property* slew_deg_per_sec**
Get the setting speed of a goto motion.

Returns: Speed for a goto movement in degrees / sec.
Return type: int

***property* slewmax_deg_per_sec**
Get the maximum speed for slew motion.
The value have a maximum, setting by a limit (_slewmax_deg_per_sec).

Returns: Maximum speed for a goto movement in degrees / sec.
Return type: float

synchro_real2simu ()

Synchronisation between simulation value of axis to real values of the axis. Parameters are setted :

- _incsimu,
- _rotsimu,

- _angsimu,
- _piersidesimu,

Useful for ending slewing movement to prevent difference offset due to calculation time of the simulation mode.
Instanciation of the axis are indispensable. However, the mountaxis module when running, have by default axisb et axisp instancied.

Instanciation Usage:

```
>>> axisb = Mountaxis("HA", name = "Unknown")
>>> axisp = Mountaxis("DEC", name = "Unknown")
```

Usage:

```
>>> axisb.synchro_real2simu()
>>> axisp.synchro_real2simu()
```

Returns: No message returned by the fonction

synchro_simu2real ()

Synchronisation between real value of axis to simulation values of the axis. Parameters are setted :

- _inc,
- _rot,
- _ang,
- _pierside,

Useful for ending slewing movement to prevent difference offset due to calculation time of the simulation mode.
Instanciation of the axis are indispensable. However, the mountaxis module when running, have by default axisb et axisp instancied.

Instanciation Usage:

```
>>> axisb = Mountaxis("HA", name = "Unknown")
>>> axisp = Mountaxis("DEC", name = "Unknown")
```

Usage:

```
>>> axisb.sydispnchro_simu2real()
>>> axisp.synchro_simu2real()
```

Returns: No message returned by the fonction

update_inc0 (inc**,** ang**,** pierside=1**)**
Update the value of the inc0.
Instanciation of the axis are indispensable. However, the mountaxis module when running, have by default axisb et axisp instancied.

Instanciation Usage:

```
>>> axisb = Mountaxis("HA", name = "Unknown")
>>> axisp = Mountaxis("DEC", name = "Unknown")
```

Usage:

```
>>> axisb.update_inc0()
>>> axisp.update_inc0()
```

Parameters:
- **inc** –
- **ang** –
- **pierside** –

Returns: No message returned by the fonction.

mountastro.mountchannel

mountastro.mounttools

mountastro.mountpad

class mountastro.mountpad.**Mountpad** (main, pad_type)
Pad to drive the mount.
To access to methods of the main thread of the mount one can use the line under the pad console and type:

```
dir(self._main)
```

mountastro.mountlog

class mountastro.mountlog.**Mountlog** (agent_alias: str, home: str = 'GPS 0 E 43 150', path_data: str = '', path_www: str = '')
Manage logs in display, file and database.
First, create an instance: log = Mountlog("test",None)
Second, use the print methods to log: log.print("Something to log")

file (*args, **kwargs)
This is the method to print in a log file. In the log file the message is presented as: Date-ISO message The last file is also apended with the message

print (*args, **kwargs)
This is the method to print in the console display and in a log file. In the console display, the message is presented as: (Agent_name) message In the log file the message is presented as: Date-ISO message

printd (*args, **kwargs)
Same as print method but only if debug level is > threshold

mountastro.mountastro_astromecca

class mountastro.mountastro_astromecca.**Mountastro_Astromecca** (*args, **kwargs)

hadec_travel ()
Exemple of using RUN lutpos1

Classes for celestial mechanics

This section is the documentation for developpers of classes related to celestial mechanics.

This page is generated according the docstrings of the python code. It displays only public methods of classes.

Module celme

celme.dates

class celme.dates.**Date** (date='')
Class to convert dates for astronomy
Date formats are:

now = Now. e.g. “now”
jd = Julian day. e.g. 24504527.45678
iso = ISO 8601. e.g. 2018-02-28T12:34:55.23
sql = ISO 8601. e.g. 2018-02-28 12:34:55.23
ymdhms = Calendar. e.g. 2018 2 28 12 34 55.23
equinox = Equinox. e.g. J2000,0
digits = Pure digits e.g. 20180228123455.23
Usage:
First, instanciate an object from the class:

::

date = Date()

Second, assign a date in any date format:

::

date.date(“2018-02-28T12:34:55”)

Third, get the converted date:

::

jd = date.jd() date = date.date() iso = date.iso() ymdhms = date.ymdhms() equinox = date.equinox()

Informations:
All dates are in UTC. Some other useful methods:

::

help(Date) Date().infos(“doctest”) Date().infos(“doc_methods”)

__add__ **(**duration**)**
Add a duration to a date

Parameters: **duration** (*string*) – Duration in dhms format (e.g 3d20h5m3s).

Example:

```
>>> objdate = Date()
>>> objdate.date("2018-02-28T12:34:55.234") ; date = objdate + "12m45s" ; date.iso()
'2018-02-28T12:34:55.234'
'2018-02-28T12:47:40.234'
```

__eq__ **(**date**)**
Comparaison of dates. Return True if dates are defined and equals.

Parameters: **date** (*Date*) – An object instancied on Date

Example:

```
>>> objdate = Date()
>>> objdate.date("2018 02 28"); objdate2 = Date("2018 02 28"); objdate == objdate2
'2018 02 28'
True
```

__ge__ **(**date**)**
Comparaison of dates: date1 >= date 2. Return True if dates are defined and date1 >= date 2.

Parameters: **date** (*Date*) – An object instancied on Date

Example:

```
>>> objdate = Date()
>>> objdate.date("2018 02 28"); objdate2 = Date("2018 02 27"); objdate >= objdate2
'2018 02 28'
True
```

__gt__ (date)
Comparaison of dates: date1 > date 2. Return True if dates are defined and date1 > date 2.

Parameters: **date** (*Date*) – An object instancied on Date

Example:

```
>>> objdate = Date()
>>> objdate.date("2018 02 28"); objdate2 = Date("2018 02 27"); objdate > objdate2
'2018 02 28'
True
```

__iadd__ (duration)
Add a duration to a date

__isub__ (duration)
Subtract a duration to a date

Parameters: **duration** (*str*) – Duration in dhms format (e.g 3d20h5m3s)

Example:

```
>>> objdate = Date()
>>> objdate.date("2018-02-28T12:34:55.234") ; objdate -= "12m45s" ; objdate.iso()
'2018-02-28T12:34:55.234'
'2018-02-28T12:22:10.234'
```

__le__ (date)
Comparaison of dates: date1 <= date 2. Return True if dates are defined and date1 <= date 2.

Parameters: **date** (*Date*) – An object instancied on Date

Example:

```
>>> objdate = Date()
>>> objdate.date("2018 02 26"); objdate2 = Date("2018 02 27"); objdate <= objdate2
'2018 02 26'
True
```

__lt__ (date)
Comparaison of dates: date1 < date 2. Return True if dates are defined and date1 < date 2.

Parameters: **date** (*Date*) – An object instancied on Date

Example:

```
>>> objdate = Date()
>>> objdate.date("2018 02 26"); objdate2 = Date("2018 02 27"); objdate < objdate2
'2018 02 26'
True
```

__ne__ (date)
Comparaison of dates. Return True if dates are defined and not equals.

Parameters: **date** (*Date*) – An object instancied on Date

Example:

```
>>> objdate = Date()
>>> objdate.date("2018 02 28"); objdate2 = Date("2018 02 28"); objdate != objdate2
'2018 02 28'
False
```

__radd__ **(**duration**)**
Right addition a duration to a date

__rsub__ **(**date**)**
Right subtraction only for a date to another date

__sub__ **(**object_or_duration**)**
Subtract a duration to a date (returns an object date) or Compute the duration between two date objects.

Parameters: **object_or_duration** (*Date object*) – Duration in dhms format (e.g 3d20h5m3s)

Example:

```
>>> objdate = Date()
>>> objdate.date("2018-02-28T12:34:55.234") ; date = objdate - "12m45s" ; date.iso()
'2018-02-28T12:34:55.234'
'2018-02-28T12:22:10.234'
>>> objdate.date("2018-02-28T12:34:55.234") ; objdate2 = Date("2018-02-25T12:34:55.234")
'2018-02-28T12:34:55.234'
3.0
```

date **(**date=''**)**
Set the input date in any format

Parameters: **date** (*any*) – date is a date in any supported format (cf. help(Date))

Returns: The input date.

Return type: string

Example:

```
>>> objdate = Date()
>>> objdate.date("2018-02-28T12:34:55.234")
'2018-02-28T12:34:55.234'
```

Note

After using objdate.date() get conversions with methods as objdate.jd() or objdate.iso().

date_date2jd **(**date**)** → Tuple[int, float]
Compute a julian day from any date format

Parameters: **date** (*string*) – A string formated as “2d7h23m12.5s”

Returns: A tuple of init_dateformat, julian day. init_dateformat: The identified format of the input: * 0 = Error, format not known * 1 = Now * 2 = Equinox * 3 = Sql * 4 = Calendar * 5 = ISO 8601 * 6 = Julian day * 7 = Modified Julian day * 8 = Digits

Return type: tuple(int, float)

Example:

```
>>> objdate = Date()
>>> objdate.date_date2jd("2018-02-28T12:34:55.234")
(5, 2458178.0242503937)
```

> ***Note***
>
> Prefer using objdate.date() followed by objdate.jd().

date_digits2jd (string**)**
Compute a julian day from a date with only digits

Parameters: **string** (*string*) – A string formated in digits
Returns: A tuple of error, julian day. error = 0 means no error
Return type: tuple(int, float)
Example:

```
>>> objdate = Date()
>>> objdate.date_iso2jd("2018-02-28T12:34:55.234")
(0, 2458178.0242503937)
```

First integer is an error code. 0 = no problem.

> ***Note***
>
> Prefer using objdate.date() followed by objdate.digits().

date_equinox2jd (string**)**
Compute a julian day from a equinoxal date

Parameters: **string** (*string*) – date string is a string in Equinox format (cf. help(Date))
Returns: A tuple of error, julian day. error = 0 means no error
Return type: tuple(int, float)
Example:

```
>>> objdate = Date()
>>> objdate.date_equinox2jd("J2025,0")
(0, 2460676.25)
```

> ***Note***
>
> Prefer using objdate.date() followed by objdate.jd()

date_iso2jd (string**) →** tuple
Compute a julian day from a ISO8601 date

Parameters: **string** (*string*) – A string formated in ISO 8601
Returns: A tuple of error, julian day. error = 0 means no error
Return type: tuple(int, float)
Example:

```
>>> objdate = Date()
>>> objdate.date_iso2jd("2018-02-28T12:34:55.234")
(0, 2458178.0242503937)
```

First integer is an error code. 0 = no problem.

> ***Note***
>
> Prefer using objdate.date() followed by objdate.jd().

date_jd2digits (jd**,** nb_subdigit=3**)**
Compute a date only with digits from a julian day

Parameters:
- **jd** (*float*) – A julian day
- **nb_subdigit** (*int*) – The number of digits returned after the seconds.

Returns: A tuple of error, string of a date formatted into ISO8601. error = 0 means no error.

Return type: tuple(int, float)

Example:

```
>>> objdate = Date()
>>> objdate.date_jd2iso(2458178.0242503937)
(0, '2018-02-28T12:34:55.234')
```

> ***Note***
>
> Prefer using objdate.date() followed by objdate.digits()

date_jd2equinox (jd**,** year_type=''**,** nb_subdigit=1**)**
Compute an equinoxal date from a julian day

Parameters:
- **jd** (*float*) – A julian day
- **year_type** (*string*) – “B” (Bessel) or “J” (Julian) or “” for automatic choice
- **nb_subdigit** (*int*) – The number of digits returned after the year.

Returns: A tuple of error, string of a date formatted into ISO8601. error = 0 means no error.

Return type: tuple(int, float)

Example:

```
>>> objdate = Date()
>>> objdate.date_jd2equinox(2458178.0242503937)
(0, 'J2018.2')
```

> ***Note***
>
> Prefer using objdate.date() followed by objdate.equinox()

date_jd2iso (jd**,** nb_subdigit=3**,** letter='T'**) →** tuple
Compute a ISO8601 date from a julian day

Parameters:
- **jd** (*float*) – A julian day
- **nb_subdigit** (*int*) – The number of digits returned after the seconds.
- **letter** (*int*) – The letter to separate date end time. If letter is “” then the output format sticks all digits without any characters :-T. So the format is no longer ISO but useful for a pure digit code.

Returns: A tuple of error, string of a date formatted into ISO8601. error = 0 means no error.

Return type: tuple(int, float)

Example:

```
>>> objdate = Date()
>>> objdate.date_jd2iso(2458178.0242503937)
(0, '2018-02-28T12:34:55.234')
```

> ***Note***
>
> Prefer using objdate.date() followed by objdate.iso()

date_jd2ymd (jd**)**

Compute a calendar date from a julian day

Parameters: **jd** (*float*) – Julian day

Example:

```
>>> objdate = Date()
>>> objdate.date_jd2ymd(2457825.14232)
(0, 2017, 3, 12.64232000010088)
```

First integer is an error code. 0 = no problem. Following items are:

- year = year
- month = month
- day = day and fraction of day

Related topics:

Prefer using objdate.date() followed by objdate.ymdhms()

date_jd2ymdhms (jd**)**

Compute a calendar date from a julian day

Parameters: **jd** (*float*) – Julian day

Example:

```
>>> objdate = Date()
>>> objdate.date_jd2ymdhms(2457824.516115046)
(0, 2017, 3, 12, 0, 23, 12.339983582496643)
```

First integer is an error code. 0 = no problem. Following items are:

- y = year
- m = month
- d = day
- hh = hour
- mm = minutes
- ss = seconds

Related topics:

Prefer using objdate.date() followed by objdate.ymdhms()

date_ymd2jd (year**,** month**,** day**)**

Compute a julian day from a calendar date

Parameters:
- **year** (*int*) – Year
- **month** (*int*) – Month
- **day** (*float*) – Day and fraction of the day

Example:

```
>>> objdate = Date()
>>> objdate.date_ymd2jd(2017,3,12.64232)
(0, 2457825.14232)
```

First integer is an error code. 0 = no problem.

Related topics:

Prefer using objdate.date() followed by objdate.jd()

date_ymdhms2jd (y: int, m: int, d: int, hh: int = 0, mm: int = 0, ss: float = 0**)** → tuple

Compute a julian day from a calendar date

Parameters:
- **y** (*int*) – Year
- **m** (*int*) – Month
- **d** (*int*) – Day
- **hh** (*int*) – Hour
- **mm** (*int*) – Minutes
- **ss** (*float*) – Seconds

Returns: A tuple of error, float corresponding to the julian day. error = 0 means no error.

Return type: tuple(int, float)

Example:

```
>>> objdate = Date()
>>> objdate.date_ymdhms2jd(2017,3,12,0,23,12.34)
(0, 2457824.516115046)
```

First integer is an error code. 0 = no problem.

Related topics:

Prefer using objdate.date() followed by objdate.jd()

digits (nb_subdigit=3**)**

Get the date in digits format

Parameters: **nb_subdigit** (*int*) – The number of digits returned after the seconds.

Example:

```
>>> objdate = Date()
>>> objdate.date("2018-02-28T12:34:55.234")
'2018-02-28T12:34:55.234'
>>> objdate.iso(2)
'2018-02-28T12:34:55.23'
```

Related topics:

Before use objdate.date() to set the input date.

equinox (year_type='J', nb_subdigit=1**)**

Get the date in equinox format

Parameters:
- **year_type** (*string*) – “B” (Bessel) or “J” (Julian) or “” for automatic choice
- **nb_subdigit** (*int*) – The number of digits returned after the year.

Example:

```
>>> objdate = Date()
>>> objdate.date("2018-02-28T12:34:55.234")
'2018-02-28T12:34:55.234'
>>> objdate.equinox()
'J2018.2'
```

Related topics:

Before use objdate.date() to set the input date.

infos (action**)**

To get informations about this class

Parameters: **action** (*string*) – A command to run a debug action (see examples).

Example:

```
Date().infos("doctest")
Date().infos("doc_methods")
Date().infos("internal_attributes")
Date().infos("public_methods")
```

iso (nb_subdigit=3**,** letter='T'**)**

Get the date in ISO 8601 format

Parameters:
- **nb_subdigit** (*int*) – The number of digits returned after the seconds.
- **letter** (*int*) – The letter to separe date end time. If letter is “” then the output format sticks all digits without any characters :-T. So the format is no longer ISO but useful for a pure digit code.

Example:

```
>>> objdate = Date()
>>> objdate.date("2018-02-28T12:34:55.234")
'2018-02-28T12:34:55.234'
>>> objdate.iso(2)
'2018-02-28T12:34:55.23'
```

Related topics:

Before use objdate.date() to set the input date.

jd ()

Get the date in julian day format

Returns: The julian day.

Return type: float

Example:

```
>>> objdate = Date()
>>> objdate.date("2018-02-28T12:34:55.234")
'2018-02-28T12:34:55.234'
>>> objdate.jd()
2458178.0242503937
```

> ***Note***
>
> Before use objdate.date() to set the input date.

ymdhms ()
Get the date in ymdhms format

Example:

```
>>> objdate = Date()
>>> objdate.date("2018-02-28T12:34:55.234")
'2018-02-28T12:34:55.234'
>>> objdate.ymdhms()
[2018, 2, 28, 12, 34, 55.23401856422424]
```

Related topics:

Before use objdate.date() to set the input date.

celme.durations

class celme.durations.**Duration** (duration='')
Class to convert dates for astronomy
Date formats are:

- day = days. e.g. 12.3457
- dhms = day, hour, minutes seconds. e.g. 2d12h34m55.23s

__add__ (duration**)**
Add a duration to a duration.

Parameters: **duration** (*Duration()*) – A duration in any supported format (cf. help(Duration))
Returns: The result of the addition
Return type: Duration()
Example:

```
>>> objduration1 = Duration()
>>> objduration2 = Duration()
>>> objduration1.duration(12.25) ; objduration2.duration("56d28m") ; objduration = objdu
12.25
'56d28m'
'+68d06h28m00.000s'
```

__eq__ (duration**)**
Comparaison of durations. Return True if durations are defined and equals.

Parameters: **duration** (*Duration()*) – A duration in any supported format (cf. help(Duration))
Returns: The logic result of the comparison.
Return type: bool
Example:

```
>>> objduration1 = Duration()
>>> objduration2 = Duration()
>>> objduration1.duration(12.345) ; objduration2.duration("56d28m") ; objduration1 == ob
12.345
'56d28m'
False
```

__ge__ (duration)
Comparaison of durations. Return True if self >= duration

Parameters: **duration** (*Duration()*) – A duration in any supported format (cf. help(Duration))
Returns: The logic result of the comparison.
Return type: bool
Example:

```
>>> objduration1 = Duration()
>>> objduration2 = Duration()
>>> objduration1.duration(12.345) ; objduration2.duration("56d28m") ; objduration1 >= ob
12.345
'56d28m'
False
```

__gt__ (duration)
Comparaison of durations. Return True if self > duration

Parameters: **duration** (*Duration()*) – A duration in any supported format (cf. help(Duration))
Returns: The logic result of the comparison.
Return type: bool
Example:

```
>>> objduration1 = Duration()
>>> objduration2 = Duration()
>>> objduration1.duration(12.345) ; objduration2.duration("56d28m") ; objduration1 > obj
12.345
'56d28m'
False
```

__iadd__ (duration)
Add a duration to a duration.

__isub__ (duration)
Subtract a duration to a duration.

Parameters: **duration** (*Duration()*) – A duration in any supported format (cf. help(Duration))
Returns: The result of the addition
Return type: Duration()
Example:

```
>>> objduration1 = Duration()
>>> objduration2 = Duration()
>>> objduration1.duration(12.25) ; objduration2.duration("56d28m") ; objduration1 -= obj
12.25
'56d28m'
'-43d18h28m00.000s'
```

__le__ (duration)
Comparaison of durations. Return True if self <= duration

Parameters: **duration** (*Duration()*) – A duration in any supported format (cf. help(Duration))
Returns: The logic result of the comparison.
Return type: bool
Example:

```
>>> objduration1 = Duration()
>>> objduration2 = Duration()
>>> objduration1.duration(12.345) ; objduration2.duration("56d28m") ; objduration1 <= ob
12.345
'56d28m'
True
```

__lt__ **(**duration**)**
Comparaison of durations. Return True if self < duration

Parameters: **duration** (*Duration()*) – A duration in any supported format (cf. help(Duration))
Returns: The logic result of the comparison.
Return type: bool
Example:

```
>>> objduration1 = Duration()
>>> objduration2 = Duration()
>>> objduration1.duration(12.345) ; objduration2.duration("56d28m") ; objduration1 < obj
12.345
'56d28m'
True
```

__mul__ **(**multiplier**)**
multiplication of duration by a float or int. Return a duration

Parameters: **multiplier** (*float*) – A real number
Returns: A duration
Return type: Duration()
Example:

```
>>> objduration = Duration()
>>> objduration.duration(30.43) ; (objduration*2).day()
30.43
60.86
```

__ne__ **(**duration**)**
Comparaison of durations. Return True if durations are defined and not equals.

Parameters: **duration** (*Duration()*) – A duration in any supported format (cf. help(Duration))
Returns: The logic result of the comparison.
Return type: bool
Example:

```
>>> objduration1 = Duration()
>>> objduration2 = Duration()
>>> objduration1.duration(12.345) ; objduration2.duration("56d28m") ; objduration1 != ob
12.345
'56d28m'
True
```

__radd__ **(**duration**)**
Right addition a duration to a duration.

__rmul__ **(**multiplier**)**
Right multiplication of a duration by a float or int.

__rsub__ **(**duration**)**

Right subtraction a duration to a duration.

__sub__ (duration)
Subtract a duration to a duration.

Parameters: **duration** (*Duration()*) – A duration in any supported format (cf. help(Duration))
Returns: The result of the addition
Return type: Duration()
Example:

```
>>> objduration1 = Duration()
>>> objduration2 = Duration()
>>> objduration1.duration(12.25) ; objduration2.duration("56d28m") ; objduration = objdu
12.25
'56d28m'
'-43d18h28m00.000s'
```

__truediv__ (divisor)
division of an angle by a float or int. Return a duration

Parameters: **divisor** (*float*) – A real number
Returns: A duration
Return type: Duration()
Example:

```
>>> objduration = Duration()
>>> objduration.duration(30.43) ; (objduration/2).day()
30.43
15.215
```

day ()
Get the date in julian day format

Returns: The julian day.
Return type: float
Example:

```
>>> objduration = Duration()
>>> objduration.duration("2d7h23m12.5s")
'2d7h23m12.5s'
>>> objduration.day()
2.3077835648148146
```

> ***Note***
> After using objdate.duration() get conversion with objdate.day().

dhms (dhms_format)
Get the date in dhms format

Parameters: **dhms_format** (*string*) – dhms format (cf. help(Duration))
Returns: The formated string.
Return type: string
Example:

```
>>> objduration = Duration()
>>> objduration.duration("2d7h3m12.5s")
```

```
'2d7h3m12.5s'
>>> objduration.dhms("0")
'2d07h03m12.50s'
```

Note

After using objdate.duration() get conversion with objdate.dhms().

duration (duration=''**)**
Set the input duration in any format

Parameters: **duration** (*any*) – duration is a duration in any supported format (cf. help(Duration))
Returns: The input duration.
Return type: string
Example:

```
>>> objduration = Duration()
>>> objduration.duration("2d7h23m12.5s")
'2d7h23m12.5s'
```

Note

After using objdate.duration() get conversions with methods as objdate.day() or objdate.dhms().

duration_duration2day (duration**) →** Tuple[int, float]
Compute day number from any duration format

Parameters: **date** (*string*) – A string formated as “2d7h23m12.5s” or “2 7 23 12.5”
Returns: A tuple of decode, julian day. decode : (sign, da, ho, mi, se)
Return type: tuple(tuple, float)
Example:

```
>>> objduration = Duration()
>>> objduration.duration_duration2day("2d7h23m12.5s")
((1, 2.0, 7.0, 23.0, 12.5), 2.3077835648148146)
```

Note

Prefer using objduration.duration() followed by objduration.day().

infos (action**) →** None
To get informations about this class

Parameters: **action** (*string*) – A command to run a debug action (see examples).
Example:

Duration().infos(“doctest”) Duration().infos(“doc_methods”) Duration().infos(“internal_attributes”) Duration().infos(“public_methods”)

celme.angles

class celme.angles.**Angle** (angle='')
Class to convert angles for astronomy
Angle formats are:

- deg = degrees. e.g.
- rad = radian. e.g.

Usage:

First, instanciate an object from the class:

```
angle = Angle()
```

Second, assign a date in any angle format:

```
angle.angle("-0d3m28.56s")
```

Third, get the converted angle:

```
rad = angle.rad()
deg = angle.deg()
arcmin = angle.arcmin()
arcsec = angle.arcsec()
dms = angle.dms()
uspzad = angle.sexagesimal(sexagesimal_format)
```

Informations :

```
help(Angle)
Angle().infos("doctest")
Angle().infos("doc_methods")
```

__add__ (angle)
Add an angle to an angle.
:param angle : An angle in any supported format (cf. help(Angle)) :type angle : Angle() :returns: The result of the addition :rtype: Angle()

Example:

```
>>> objangle1 = Angle()
>>> objangle2 = Angle()
>>> objangle1.angle(12.345) ; objangle2.angle("56d28m") ; objangle = objangle1 + objangl
12.345
'56d28m'
'68d48m42.00s'
```

__eq__ (angle)
Comparaison of angles. Return True if angles are defined and equals.
:param angle : An angle in any supported format (cf. help(Angle)) :type angle : Angle() :returns: The logic result of the comparison. :rtype: bool

Example:

```
>>> objangle1 = Angle()
>>> objangle2 = Angle()
>>> objangle1.angle(12.345) ; objangle2.angle("56d28m") ; objangle1 == objangle2
12.345
'56d28m'
False
```

> ***Note***
>
> Does not account for the modulo.

__ge__ **(**angle**)**
Comparaison of angles. Return True if self >= angle
:param angle : An angle in any supported format (cf. help(Angle)) :type angle : Angle() :returns: The logic result of the comparison. :rtype: bool

Example:

```
>>> objangle1 = Angle()
>>> objangle2 = Angle()
>>> objangle1.angle(12.345) ; objangle2.angle("56d28m") ; objangle1 >= objangle2
12.345
'56d28m'
False
```

> ***Note***
>
> Does not account for the modulo.

__gt__ **(**angle**)**
Comparaison of angles. Return True if self > angle
:param angle : An angle in any supported format (cf. help(Angle)) :type angle : Angle() :returns: The logic result of the comparison. :rtype: bool

Example:

```
>>> objangle1 = Angle()
>>> objangle2 = Angle()
>>> objangle1.angle(12.345) ; objangle2.angle("56d28m") ; objangle1 > objangle2
12.345
'56d28m'
False
```

> ***Note***
>
> Does not account for the modulo.

__iadd__ **(**angle**)**
Add an angle to an angle.

__isub__ **(**angle**)**
Subtract an angle to an angle.
:param angle : An angle in any supported format (cf. help(Angle)) :type angle : Angle() :returns: The result of the subtraction :rtype: Angle()

Example:

```
>>> objangle1 = Angle()
>>> objangle2 = Angle()
>>> objangle1.angle(12.345) ; objangle2.angle("56d28m") ; objangle1 -= objangle2 ; objan
12.345
'56d28m'
'315d52m42.00s'
```

__le__ **(**angle**)**
Comparaison of angles. Return True if self <= angle
:param angle : An angle in any supported format (cf. help(Angle)) :type angle : Angle() :returns: The logic result of the comparison. :rtype: bool

Example:

```
>>> objangle1 = Angle()
>>> objangle2 = Angle()
>>> objangle1.angle(12.345) ; objangle2.angle("56d28m") ; objangle1 <= objangle2
12.345
'56d28m'
True
```

> ***Note***
>
> Does not account for the modulo.

__lt__ **(**angle**)**
Comparaison of angles. Return True if self < angle
:param angle : An angle in any supported format (cf. help(Angle)) :type angle : Angle() :returns: The logic result of the comparison. :rtype: bool

Example:

```
>>> objangle1 = Angle()
>>> objangle2 = Angle()
>>> objangle1.angle(12.345) ; objangle2.angle("56d28m") ; objangle1 < objangle2
12.345
'56d28m'
True
```

> ***Note***
>
> Does not account for the modulo.

__mod__ **(**angle**)**
Modulo of an angle by another angle. Return an angle
:param angle : An angle in any supported format (cf. help(Angle)) :type angle : Angle() :returns: An angle :rtype: Angle()

Example:

```
>>> objangle1 = Angle()
>>> objangle2 = Angle()
>>> objangle1.angle(30.56) ; objangle2.angle(2.34) ; (objangle1 % objangle2).deg()
30.56
2.34
0.14000000000000412
```

__mul__ **(**multiplier**)**
multiplication of an angle by a float or int. Return an angle

Parameters:	**multiplier** (*float*) – A real number
Returns:	An angle
Return type:	Angle()

Example:

```
>>> objangle = Angle()
>>> objangle.angle(30.43) ; (objangle*2).deg()
30.43
60.86000000000001
```

__ne__ (angle)
Comparaison of angles. Return True if angles are defined and not equals.
:param angle : An angle in any supported format (cf. help(Angle)) :type angle : Angle() :returns: The logic result of the comparison. :rtype: bool

Example:

```
>>> objangle1 = Angle()
>>> objangle2 = Angle()
>>> objangle1.angle(12.345) ; objangle2.angle("56d28m") ; objangle1 != objangle2
12.345
'56d28m'
True
```

> ***Note***
>
> Does not account for the modulo.

__radd__ (angle)
Right addition an angle to an angle.

__rmul__ (multiplier)
Right multiplication of an angle by a float or int.

__rsub__ (angle)
Right subtraction only an angle to an angle.

__sub__ (angle)
Subtract an angle to an angle.
:param angle : An angle in any supported format (cf. help(Angle)) :type angle : Angle() :returns: The result of the subtraction :rtype: Angle()

Example:

```
>>> objangle1 = Angle()
>>> objangle2 = Angle()
>>> objangle1.angle(12.345) ; objangle2.angle("56d28m") ; objangle = objangle1 - objangl
12.345
'56d28m'
'315d52m42.00s'
```

__truediv__ (divisor)
division of an angle by a float or int. Return an angle

Parameters: **divisor** (*float*) – A real number
Returns: An angle
Return type: Angle()

Example:

```
>>> objangle = Angle()
>>> objangle.angle(30.43) ; (objangle/2).deg()
```

```
30.43
15.215000000000002
```

angle (angle=''**)**
Set the input angle in any format

Parameters: **angle** (*str*) – angle is an angle in any supported format (cf. help(Angle))

Example:

```
>>> objangle = Angle()
>>> objangle.angle("23d 27m")
'23d 27m'
```

Related topics:

After using objangle.angle() get conversions with methods as objangle.deg() or objdate.rad().

angle_angle2rad (angle**)**
Compute radian from any angle format

Parameters: **angle** (*str*) – angle is an angle in any supported format (cf. help(Angle))

Usage:

```
>>> objangle = Angle()
>>> objangle.angle_angle2rad("-57d45m34s")
([('-57', 'D'), ('45', 'M'), ('34', 'S')], -1.0080924796783026)
```

First integer is used to check the recognized input date format: 0 = Error, format not known

Related topics:

Prefer using objangle.angle() followed by objangle.rad()

angle_rad2deg (rad**)**
Compute a angle in degrees from a angle in radian

Parameters: **rad** (*float*) – rad is an angle in radian.

Usage:

```
>>> objangle = Angle()
>>> objangle.angle_rad2deg(1)
(0, 57.29577951308232)
```

First integer is an error code. 0 = no problem.

Related topics:

Prefer using objangle.angle() followed by objangle.deg()

angle_rad2sexagesimal (rad**,** sexagesimal_format**)**
Compute a sexagesimal format string from radian

Parameters:

- **rad** (*float*) – rad is an angle in radian
- **sexagesimal_format** (*str*) – The uspzad format (see details below)

Sexagesimal format:

- u (unit) = h,H,d,D (default=D). Capital mean module [0:360[, lower case means module [-180:180[
- s (separator) = ” “,:,”” (default=”” means letters hms or dms)

- p (plus/minus) = +,"" (default="")
- z (zeros) = 0,"" (default="")
- a (angle_limits) = "",90, (+/-limit if unit D,H, default="" means 360)
- d (sec_digits) = "",".1",".2",... (default="")

Style 1:

- To Display a R.A.: "H0.2" => 23h07m42.49s
- To Display a Decl.: "d+090.1" => +00d34m22.6s
- To Display a H.A.: "h0.2" => -08h43m16.05s

Style 2:

- To Display a R.A.: "H 0.2" => 23 07 42.49
- To Display a Decl.: "d +090.1" => -00 34 22.6
- To Display a H.A.: "h 0.2" => -08 43 16.05

Style 3:

- To Display a R.A.: "H:0.2" => 23:07:42.49
- To Display a Decl.: "d:+090.1" => -00:34:22.6
- To Display a H.A.: "h:0.2" => -08:43:16.05

Example:

```
>>> objangle = Angle()
>>> objangle.angle_rad2sexagesimal(-0.01,"d:-090.1")
(0, '-00:34:22.6')
```

First integer is used to check the recognized input date format: 0 = Error, format not known

Related topics:

Prefer using objangle.angle() followed by objangle.sexagesimal()

arcmin ()

Get the angle in arcmin

Example:

```
>>> objangle = Angle()
>>> objangle.angle("-23 d 56'")
"-23 d 56'"
>>> objangle.arcmin()
-1436.0
```

Related topics:

Before use objdate.angle() to set the input angle.

arcsec ()

Get the angle in arcsec

Example:

```
>>> objangle = Angle()
>>> objangle.angle("-23 d 56'")
"-23 d 56'"
>>> objangle.arcsec()
-86160.0
```

Related topics:

Before use objdate.angle() to set the input angle.

deg ()

Get the angle in degrees

Example:

```
>>> objangle = Angle()
>>> objangle.angle("-23 d 56'")
"-23 d 56'"
>>> objangle.deg()
-23.933333333333334
```

Related topics:

Before use objdate.angle() to set the input angle.

infos (action**) →** None

To get informations about this class

Parameters: **action** (*string*) – A command to run a debug action (see examples).

Example:

Angle().infos("doctest") Angle().infos("doc_methods") Angle().infos("internal_attributes") Angle().infos("public_methods")

rad ()

Get the angle in radian

Example:

```
>>> objangle = Angle()
>>> objangle.angle("-23 d 56'")
"-23 d 56'"
>>> objangle.rad()
-0.4177154676439762
```

Related topics:

Before use objdate.angle() to set the input angle.

sexagesimal (sexagesimal_format**)**

Get the angle in sexagesimal

Parameters:

- **rad** (*float*) – rad is an angle in radian
- **sexagesimal_format** (*str*) – The uspzad format (see details below)

Sexagesimal format:

- u (unit) = h,H,d,D (default=D). Capital mean module [0:360[, lower case means module [-180:180[
- s (separator) = " ",:,"" (default="" means letters hms or dms)
- p (plus/minus) = +,"" (default="")
- z (zeros) = 0,"" (default="")
- a (angle_limits) = "",90, (+/-limit if unit D,H, default="" means 360)
- d (sec_digits) = "","1","2",... (default="")

Style 1:

- To Display a R.A.: “H0.2” => 23h07m42.49s
- To Display a Decl.: “d+090.1” => +00d34m22.6s
- To Display a H.A.: “h0.2” => -08h43m16.05s

Style 2:

- To Display a R.A.: “H 0.2” => 23 07 42.49
- To Display a Decl.: “d +090.1” => -00 34 22.6
- To Display a H.A.: “h 0.2” => -08 43 16.05

Style 3:

- To Display a R.A.: “H:0.2” => 23:07:42.49
- To Display a Decl.: “d:+090.1” => -00:34:22.6
- To Display a H.A.: “h:0.2” => -08:43:16.05

Example:

```
>>> objangle = Angle()
>>> objangle.angle(-0.57)
-0.57
>>> objangle.sexagesimal("d:-090.1")
'-00:34:12.0'
```

Related topics:

Before use objdate.angle() to set the input angle.

celme.coords

class celme.coords.**Coords** (cart_or_sphe=(0, 0, 0))

Class to convert coordinates for astronomy
Coordinate formats are:

- cart = tuple of x, xy, xyz
- sphe = tuple of r, rp, rpt

Usage:

First, instanciate an object from the class:

```
coord = Coords()
```

Second, assign a coordinate in any format:

```
xyz = (x, y, z)
coord.coords(xyz)
```

or

```
phi = Angle(45)
theta = Angle(120)
rpt = (r, phi, theta)
coord.coords(rpt)
```

Third, get the converted angles:

```
r, phi, theta = coord.sphe()
x,y,y = coord.cart()
```

Fourth, some tools

```
coord.separation(coord2)
coord.pole(coord2)
coord.scalar_product(coord2)
coord.vectorial_product(coord2)
```

Informations :

```
help(Coords)
Coords().infos("doctest")
Coords().infos("doc_methods")
```

cart ()
Get the cartesian coordinates

Returns: The tuple (x,y,z)

Return type: tuple(float, float, float)

Example:

```
>>> objcoord = Coords()
>>> objcoord.coords((1,3,6))
(1, 3, 6)
>>> objcoord.cart()
(1, 3, 6)
```

> ***Note***
>
> Before use objcoord.cart() to set the input coords with objcoord.coords().

coords (cart_or_sphe**)**
Set the input cart_or_sphe in any format

Parameters: **cart_or_sphe** (*tuple*) – Cartesian or spherical coordinates. If the tupple contains two elements it is spherical coordinates. If the tupple contains three elements it is cartesian coordinates.

Angles are in any supported format (cf. help(Coords))

Example:

```
>>> objcoord = Coords()
>>> objcoord.coords((1,3,6))
(1, 3, 6)
```

Related topics:

After using objcoord.coords() get conversions with methods as objcoord.cart() or objcoord.sphe().

infos (action**) →** None
To get informations about this class

Parameters: **action** (*string*) – A command to run a debug action (see examples).

Example:

Coords().infos(“doctest”) Coords().infos(“doc_methods”) Coords().infos(“internal_attributes”) Coords().infos(“public_methods”)

sphe (format_phi='deg', format_theta='deg'**)**
Get the spherical coordinates

Parameters:
- **format_phi** (*str*) – Angle format unit.
- **format_theta** (*str*) – Angle format unit.

Returns: The tuple (x,y,z)

Return type: tuple(float, float, float)

Example:

```
>>> objcoord = Coords()
>>> objcoord.coords((1,3,6))
(1, 3, 6)
>>> objcoord.sphe("H","d")
(6.782329983125268, '4h46m15.61s', '62d12m31.30s')
```

Note

Before use objcoord.sphe() to set the input coords with objcoord.coords().

celme.horizons

class celme.horizon.**Horizon** (home, *args)
Class to describe an Horizon

horizon (home**,** *args**)**
Object initialization

infos (action**) →** None
To get informations about this class

Parameters: **action** (*string*) – A command to run a debug action (see examples).

Example:

Horizon().infos(“doctest”) Horizon().infos(“doc_methods”) Horizon().infos(“internal_attributes”) Horizon().infos(“public_methods”)

celme.atmosphere

class celme.atmosphere.**Atmosphere**
Class to describe an atmosphere

infos (action**) →** None
To get informations about this class

Parameters: **action** (*string*) – A command to run a debug action (see examples).

Example:

Atmosphere().infos(“doctest”) Atmosphere().infos(“doc_methods”) Atmosphere().infos(“internal_attributes”) Atmosphere().infos(“public_methods”)

celme.home

class celme.home.**Home** (*args)
Class to describe a home

home (args**)**
Object initialization
{GPS long_deg sense lati_deg altitude_m} {MPC IAU_long_deg rhocosphi rhosinphi }

infos (action) → None
To get informations about this class

Parameters: **action** (*string*) – A command to run a debug action (see examples).

Example:

Home().infos("doctest") Home().infos("doc_methods") Home().infos("internal_attributes") Home().infos("public_methods")

celme.site

class celme.site.**Site** (home)
Class to describe an Site

infos (action) → None
To get informations about this class

Parameters: **action** (*string*) – A command to run a debug action (see examples).

Example:

Site('GPS 0 E 0 0').infos("doctest") Site('GPS 0 E 0 0').infos("doc_methods") Site('GPS 0 E 0 0').infos("internal_attributes") Site('GPS 0 E 0 0').infos("public_methods")

celme.mechanics

class celme.mechanics.**Mechanics**
Class to compute celestial mechanics

celme.targets

class celme.targets.**Target** (*args, **kwargs)
Class to compute planets for astronomy

infos (action) → None
To get informations about this class

Parameters: **action** (*string*) – A command to run a debug action (see examples).

Example:

Target().infos("doctest") Target().infos("doc_methods") Target().infos("internal_attributes") Target().infos("public_methods")

Index

A

C

D

E

F

H

I

J

L

M

N

P

R

S

T

U

Y

Python Module Index

c

m