Analyzing data with Jupyter notebook and Astropy.

Lindita Hamolli

We recommend to follow this tutorial by executing the code cells on your local machine .

What is Astropy?

"The Astropy Project is a community effort to develop a single core package for Astronomy in Python. It started in 2011 and now have a lot of people included".

Astropy is suported by numpy (for array representation and arithmetic), scipy (a wide variety of functions for scientific computing) and matplotlib (publication-quality plotting) libraries.

The Astropy package is structured into several submodules, the most important of them are:

1. **astropy.units and in particular astropy.units.Quantity to do the astronomical calculation with units.**
2. **astropy.coordinates and in particular the classes SkyCoord and Angle to handle the astronomical sky positions, coordinate systems and coordinate transformations.**
3. **astropy.table and the Table class to handle astronomical data tables.**
4. **astropy.io.fits to open and write data files in Fits Format.**
5. **Ploting the astronomical sky images with astropy.visualization.wcsaxes.**

In addition to the Astropy core package there is an infrastructure of Astropy affiliated packages dedicated to specific fields of Astronomy.

- **Astroplan: observing planning**
- **Reproject: reprojection of sky images**
- **Regions: handling of sky regions**
- **Photutils: sourse detection and photometry**
- **Gammapy: Gamma-ray Astronomy data analysis**
- **.... and more others.**

Other Ressources

There are other ressources with Astropy tutorials, we can recommend:

- **Learn.Astropy, webpage with a lot of tutorial material.**
- **Astropy documentation webpage, with a lor of small usage examples.**

See (https://www.astropy.org/help.html) for a list of references, how to get help in Astropy.

In [1]:

```
import numpy as np
import astropy
import matplotlib
from astropy import units as u
from astropy import constants as const
from matplotlib import pyplot as plt
from astropy.visualization import quantity_support
from astropy.coordinates import EarthLocation, AltAz
from astropy.time import Time
from astropy.coordinates import SkyCoord
from datetime import datetime
from IPython import display
from astropy.table import Table
from astropy.table import Column
from astropy.io import fits
```

```
from astropy.utils.data import download_file
from astropy.wcs import WCS
```

In [2]:

```
# This ia a code cell
x='Welcome to Astropy!'
print(x)
```

```
Welcome to Astropy!
```

The cell Markdown is basically plain text.

Header

Sub_header

bold text

italic text

For more formating ways of Markdown cells see [Markdown cheat sheet]

In [3]:

```
print('numpy:',np.__version__)
print('astropy:', astropy.__version__)
```

```
numpy: 1.18.5
astropy: 4.0.1.post1
```

Units

The astropy.units subpackage provide functions and classes to handle physical quantities with units.

The recomended way import the astropy.units submodule is:

In [4]:

```
from astropy import units as u      #this will conflict with any variables called u
```

In [5]:

```
u.m
```

Out[5]:

$$\mathrm{m}$$

In [6]:

```
u.m.__doc__
```

Out[6]:

```
'meter: base unit of length in SI'
```

In [7]:

```
u.m.physical_type
```

Out[7]:

```
'length'
```

Check tha available units with tab completion on the unit module u.TAB

In [8]:

```
u.pc
```

Out[8]:

pc

In [9]:

```
u.pc.physical_type
```

Out[9]:

```
'length'
```

In [10]:

```
print(u.pc.__doc__)
```

```
parsec: approximately 3.26 light-years.
```

A full list units is available here

We can create composite units:

In [11]:

```
u.m/u.kg/u.s**2
```

Out[11]:

$$\frac{\mathrm{m}}{\mathrm{kg\,s^{2}}}$$

In [12]:

```
repr(u.m/u.kg/u.s**2)       #represent a new unit
```

Out[12]:

```
'Unit("m / (kg s2)")'
```

Quantities

Quantities are created by multiplying any number with a unit object.

In [13]:

```
distance =1. *u.lightyear     #distance=c*1(year)
print(distance)
```

```
1.0 lyr
```

or by passing a string to the general Quantity object:

In [14]:

```
distance =u.Quantity('1.77777  lyr')
print(distance)
```

```
1.77777 lyr
```

In [15]:

```
distance.value        #take the last value
```

Out[15]:

```
1.77777
```

In [16]:

```
distance.unit
```

Out[16]:

lyr

Quantities can be also created using list or arrays

In [17]:

```
import numpy as np
distances =[1, 10, 12] *u.lightyear      #array
print(distances)
# or
distances =np.array([1, 10, 12])*u.lightyear
print(distances)
```

```
[ 1. 10. 12.] lyr
[ 1. 10. 12.] lyr
```

In [18]:

```
distances.value
```

Out[18]:

```
array([ 1., 10., 12.])
```

The quantity object has a value attribute, which is a plan numpy.ndarray

In [19]:

```
type(distance.value)      #this is a float variabel
```

Out[19]:

```
numpy.float64
```

In [20]:

```
type(distances.value)     #this is an array in numpy
```

Out[20]:

```
numpy.ndarray
```

A quantity behaves in many ways just a like numpy.ndarray with a attached unit.

In [21]:

```
distances*10
#distances_10=distances*10
#print(distances)
#distances_10
```

Out[21]:

[10, 100, 120] lyr

In [22]:

```
dir()     #show all variables
```

Out[22]:

```
['AltAz',
 'Column',
 'EarthLocation',
 'In',
 'Out',
```

```
'SkyCoord',
'Table',
'Time',
'WCS',
'_',
'_11',
'_12',
'_15',
'_16',
'_18',
'_19',
'_20',
'_21',
'_5',
'_6',
'_7',
'_8',
'_9',
'__',
'___',
'__builtin__',
'__builtins__',
'__doc__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'_dh',
'_i',
'_i1',
'_i10',
'_i11',
'_i12',
'_i13',
'_i14',
'_i15',
'_i16',
'_i17',
'_i18',
'_i19',
'_i2',
'_i20',
'_i21',
'_i22',
'_i3',
'_i4',
'_i5',
'_i6',
'_i7',
'_i8',
'_i9',
'_ih',
'_ii',
'_iii',
'_oh',
'astropy',
'const',
'datetime',
'display',
'distance',
'distances',
'download_file',
'exit',
'fits',
'get_ipython',
'matplotlib',
'np',
'plt',
'quantity_support',
'quit',
'u',
'x']
```

In [23]:

```
#del distances          #delete variables
```

In [24]:

```
distances.value
```

Out[24]:

```
array([ 1., 10., 12.])
```

Many numpy functions will work as expected and return again a Quantity object:

In [25]:

```
np.max(distances)
```

Out[25]:

12 lyr

In [26]:

```
np.mean(distances)
```

Out[26]:

7.6666667 lyr

But there are exceptions, where the unit handling is not well defined, e.g. in np.log arguments have to be dimensionless. Such as:

In [27]:

```
#np.log(30*u.Mev)           # Will raise an UnitConversionError
np.log(30*u.MeV/(10*u.MeV))                    # log is for natyral logarithm
```

Out[27]:

1.0986123

In [28]:

```
np.log10(200*u.MeV/(2*u.MeV))
```

Out[28]:

2

Probably the most useful method on the Quantity objects is the .to() method, which allows to convert a quantity to different units:

In [29]:

```
#print(distance)
```

In [30]:

```
distance.to(u.meter)
```

Out[30]:

$1.6819003 \times 10^{16} \text{ m}$

In [31]:

```
distance.to('meter')  #declare the unit as string
```

Out[31]:

1.6819003×10^{16} m

In [32]:

```
distance.to(u.parsec)
```

Out[32]:

0.54506676 pc

Quantities can be combined with any arithmetical expression to derive other quantities, astropy.units will propagate the units correctly:

In [33]:

```
distance =1. *u.lightyear
print(distance)
speed_of_light=distance/u.year
print(speed_of_light)
print(speed_of_light.to('km/s'))
```

```
1.0 lyr
1.0 lyr / yr
299792.458 km / s
```

In [34]:

```
print(speed_of_light.to('angstrom/day'))
```

```
2.5902068371199996e+23 Angstrom / d
```

Quantities can be combined:

In [35]:

```
q1=3.*u.m
```

In [36]:

```
q2=5.*u.cm/u.s/u.g**2
```

In [37]:

```
q1*q2
```

Out[37]:

$15 \frac{\mathrm{cm\,m}}{\mathrm{s\,g^2}}$

and converted to different units:

In [38]:

```
(q1*q2).to(u.m**2/u.kg**2/u.s)
```

Out[38]:

$150000 \frac{\mathrm{m^2}}{\mathrm{s\,kg^2}}$

The units of a quantity can be decomposed into a set of base units using the decompose() method.

For standardized unit systems such as "si" ore "cgs" there are convenience attributes on the quantity object:

In [39]:

```
(3. *u.cm* u.pc/u.g/u.year**2)
```

Out[39]:

Out[39]:

$3\ \frac{\mathrm{cm\,pc}}{\mathrm{g\,yr^2}}$

In [40]:

```
(3. *u.cm* u.pc/u.g/u.year**2).decompose()      # defauld is si system
```

Out[40]:

$929.53097\ \frac{\mathrm{m^2}}{\mathrm{kg\,s^2}}$

In [41]:

```
u.si.bases
```

Out[41]:

```
{Unit("A"),
 Unit("K"),
 Unit("cd"),
 Unit("kg"),
 Unit("m"),
 Unit("mol"),
 Unit("rad"),
 Unit("s")}
```

In [42]:

```
(3. *u.cm* u.pc/u.g/u.year**2).decompose(u.cgs.bases)     #converted in cgs system
```

Out[42]:

$9295.3097\ \frac{\mathrm{cm^2}}{\mathrm{g\,s^2}}$

In [43]:

```
u.cgs.bases
```

Out[43]:

```
{Unit("K"),
 Unit("cd"),
 Unit("cm"),
 Unit("g"),
 Unit("mol"),
 Unit("rad"),
 Unit("s")}
```

In [44]:

```
(3. *u.cm* u.pc/u.g/u.year**2).cgs                    # other way to converted in cgs system
```

Out[44]:

$9295.3097\ \frac{\mathrm{erg}}{\mathrm{g^2}}$

The most practical way to work with units is: define the input quatities with units, do the computation and forget about the units, convert the final result to the desired units. In most cases there is no need for intermediate unit conversions.

In [45]:

```
speed_of_light.si
```

Out[45]:

$2.9979246 \times 10^{8}\ \frac{\mathrm{m}}{\mathrm{s}}$

In [46]:

In [46]:

```
speed_of_light.cgs
```

Out[46]:

$2.9979246 \times 10^{10} \ \frac{\mathrm{cm}}{\mathrm{s}}$

Equivalencies

In astronomy, the physics quantities are often measured in more practical units, which are equivalent to the actual physical unit. In astropy.units this is handled with the concept of "equivalencies".

For example consider units to measure spectral quantities such as wavelength, frequency and energy :

In [47]:

```
frequency=3e20*u.hertz      #E=h*f
print(frequency)
frequency.to('MeV', equivalencies=u.spectral())
```

```
3e+20 Hz
```

Out[47]:

1.2407003 MeV

In [48]:

```
frequency.to('picometer', equivalencies=u.spectral())    #f=c/wavelength
```

Out[48]:

0.99930819 pm

Or for converting temperature

In [49]:

```
temperature=25*u.Celsius
print(temperature)
```

```
25.0 deg_C
```

In [50]:

```
temperature.to('Kelvin', equivalencies=u.temperature())   #T=273.15+t
```

Out[50]:

298.15 K

Constants

Astropy provides a lot of constants quantities in the astropy.constants submodule:

In [51]:

```
from astropy import constants as const
```

In [52]:

```
print (const.G)
```

```
  Name   = Gravitational constant
  Value  = 6.6743e-11
  Uncertainty  = 1.5e-15
  Unit   = m3 / (kg s2)
  Reference = CODATA 2018
```

In [53]:

```
print(const.c.to('km/s'))
```

```
299792.458 km / s
```

In [54]:

```
F=(const.G*1.*u.M_sun*100.*u.kg)/(3.2*u.au)**2    # force of interaction between the sun and a satellite
```

In [55]:

```
u.au.__doc__
```

Out[55]:

```
'astronomical unit: approximately the mean Earth--Sun distance.'
```

In [56]:

```
F
```

Out[56]:

$$6.5178711 \times 10^{-10}\ \frac{\mathrm{m^3\ M_\odot}}{\mathrm{AU^2\ s^2}}$$

In [57]:

```
F.to(u.N)
```

Out[57]:

$$0.057910972\ \mathrm{N}$$

In [58]:

```
print(u.M_sun.__doc__)
```

```
Solar mass
```

Here is a [list of available constances](#).

In [59]:

```
const.G
```

Out[59]:

$$6.6743 \times 10^{-11}\ \frac{\mathrm{m^3}}{\mathrm{kg\ s^2}}$$

In [60]:

```
const.k_B
```

Out[60]:

$$1.380649 \times 10^{-23}\ \frac{\mathrm{J}}{\mathrm{K}}$$

In [61]:

```
const.h
```

Out[61]:

$$6.6260701 \times 10^{-34}\ \mathrm{J\ s}$$

If you write a function you can make sure the input is given in the right units using the astropy.units.quantity_input decorator. A decorator for validating the units of arguments to functions.

In [62]:

```
@u.quantity_input(frequency=u.hertz, temperature=u.K)  # a decorator to validate the uni
ts of arguments to functions

def blackbody(frequency, temperature):     #declare a function
    pre_factor=2*(const.h*frequency**3)/const.c**2                          #E(f,T)=[2hf^3/
c^2]*[1/(exp(hf/kT)-1)]
    exponential_factor=1./(np.exp((const.h*frequency)/(const.k_B*temperature))-1)

    return pre_factor*exponential_factor
```

In [63]:

```
blackbody (300*u.hertz, 500*u.K)
#blackbody (300, 500*u.K)
```

Out[63]:

$1.3825636 \times 10^{-32} \ \frac{\mathrm{Hz^3\,s^3\,J}}{\mathrm{m^2}}$

Interfacing quantities with Matplotlib

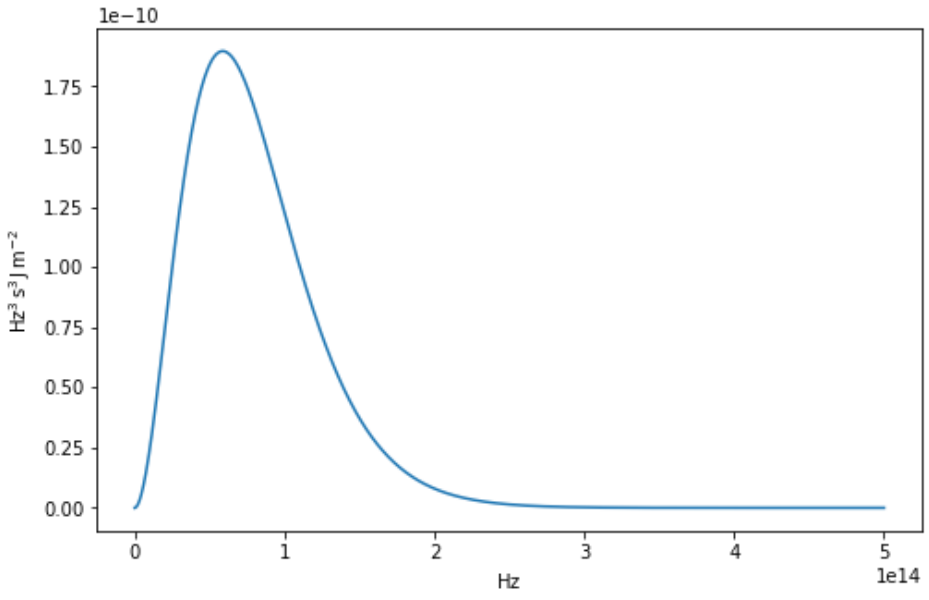
When writing code that uses quantities we are typicaly bound to use it everywhere in the code. But often we'd like to interface with functions (e.g. from Scipy or Matplotlib. For Matplotlib, Astropy has a builtin suport using the quantity_support() context manager.

In [64]:

```
from matplotlib import pyplot as plt
from astropy.visualization import quantity_support   #Enable support for plotting astropy
.units.Quantity instances in matplotlib.
quantity_support()
plt.figure(figsize=(8,5))
temperature=1000 * u.K
frequencies=np.linspace(1E-1, 0.5e15, 1000)* u.hertz             #vlera e pare, vlera e
fundit, 1000 vlera
radiance=blackbody(frequency=frequencies, temperature=temperature)
plt.plot(frequencies, radiance)
```

Out[64]:

```
[<matplotlib.lines.Line2D at 0xd4feb80>]
```



In [65]:

```
x1 = 1
x2 = 2
```

```
y=np.logspace(x1, x2, 10)     # nga 1 ne 2 ndahet ne 10 numra dhe pastaj 10^numri
print(y)

y3=np.geomspace(x1, x2, 10)   # kthen 10 numra nga 1 ne 2 sipas progresionit gjeometrik
print(y3)
```

```
[ 10.          12.91549665  16.68100537  21.5443469   27.82559402
  35.93813664  46.41588834  59.94842503  77.42636827 100.        ]
[1.         1.08005974 1.16652904 1.25992105 1.36079    1.46973449
 1.58740105 1.71448797 1.85174942 2.        ]
```

In [66]:

```
x3=np.sin(30*u.degree)
x3
```

Out[66]:

0.5

In [67]:

```
x4=np.sqrt(100*u.km*u.km)
x4
```

Out[67]:

$10 \mathrm{km}$

In [68]:

```
x5=np.exp(8.*u.m/(2.*u.km))
x5
```

Out[68]:

1.004008

In [69]:

```
x6=np.arcsin(1. *u.dimensionless_unscaled).to(u.degree)
print(x6)
x6=np.arcsin(1.)
print(x6)                                    # e kthen ne rad
```

```
90.0 deg
1.5707963267948966
```

Coordinates

With the submodule astropy.coordinates, Astropy provides a framework to handle sky positions in various coordinate systems and transformation between them.

The basic class to handle sky coordinates is SkyCoord:

In [70]:

```
from astropy.coordinates import SkyCoord
```

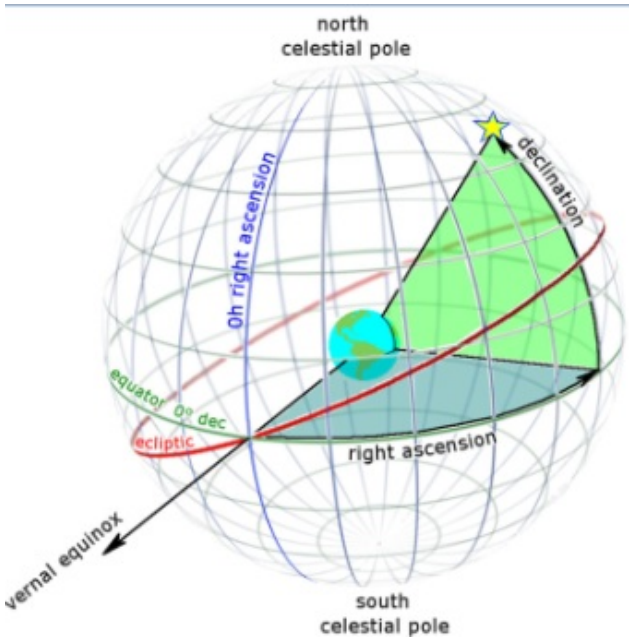
It can be created by passing a position angle for longitude and latitude and a keyword specifying a coordinate frame:

In [71]:

```
position_crab=SkyCoord(83.63*u.deg, 22.01*u.deg, frame='icrs')      #  International Cel
estial Reference System
print(position_crab)
```

```
<SkyCoord (ICRS): (ra, dec) in deg
    (83.63, 22.01)>
```

Insert Image in a Jupyter Notebook :Markdown, Edit, Insert Image, Choose file

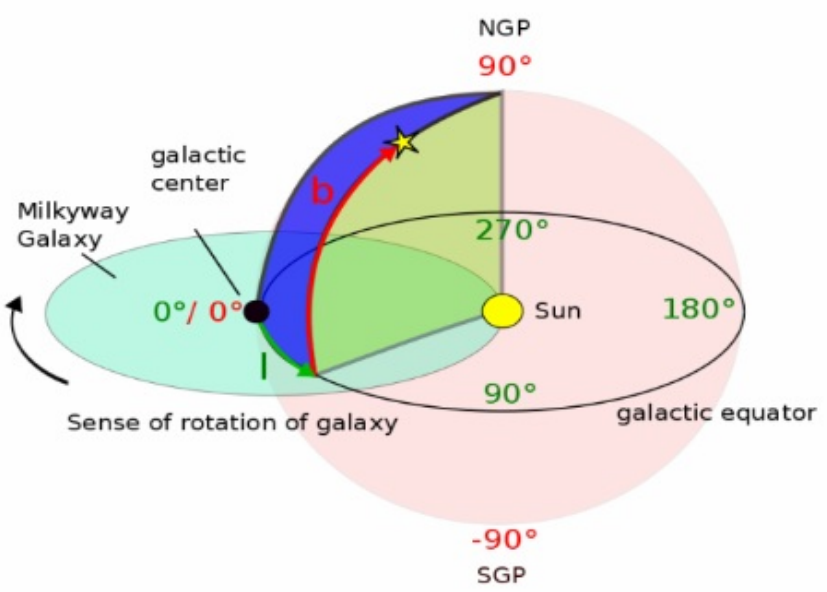


As for Quantities the instanciation with lists, arrays or even Quantities also work:

In [72]:

```
positions=SkyCoord([345.,234.3]*u.deg, [-0.1, 0.2]*u.deg, frame='galactic')
print(positions)
```

```
<SkyCoord (Galactic): (l, b) in deg
    [(345. , -0.1), (234.3,  0.2)]>
```



Alternatively the angles can be specified as string:

In [73]:

```
position_crab=SkyCoord('5h34m31.97s', '22d0m52.10s', frame='icrs')
print(position_crab)
```

```
<SkyCoord (ICRS): (ra, dec) in deg
    (83.63320833, 22.01447222)>
```

A very convenient way to get the coordinates of an individual object is querying the Sesame database with SkyCoord.from_name():

In [74]:

In [74]:

```
SkyCoord.from_name('Crab')         #Crab Nebula
```

Out[74]:

```
<SkyCoord (ICRS): (ra, dec) in deg
    (83.6333, 22.0133)>
```

In [75]:

```
SkyCoord.from_name('M33')            # Messier 33
```

Out[75]:

```
<SkyCoord (ICRS): (ra, dec) in deg
    (23.46206906, 30.66017511)>
```

In [76]:

```
SkyCoord.from_name('M31')         #Andromeda Galaxy
```

Out[76]:

```
<SkyCoord (ICRS): (ra, dec) in deg
    (10.6847083, 41.26875)>
```

In [77]:

```
pos_M31=SkyCoord.from_name('M31')
print(pos_M31)
```

```
<SkyCoord (ICRS): (ra, dec) in deg
    (10.6847083, 41.26875)>
```

In [78]:

```
pos_LMC=SkyCoord.from_name('LMC')    #Large Magellanic Cloud
print(pos_LMC)
```

```
<SkyCoord (ICRS): (ra, dec) in deg
    (80.89417, -69.75611)>
```

In [79]:

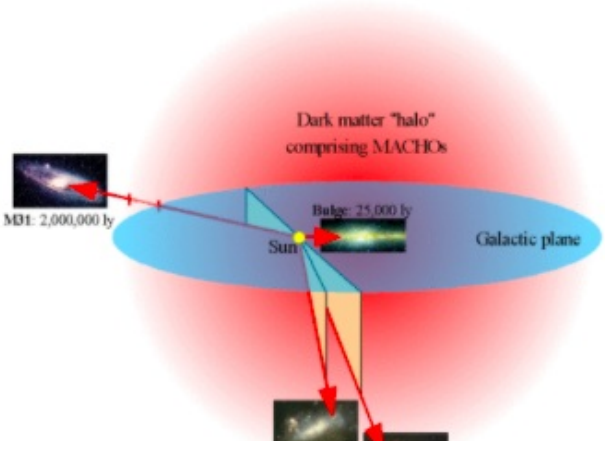
```
pos_SMC=SkyCoord.from_name('SMC')    #Large Magellanic Cloud
print(pos_SMC)
```

```
<SkyCoord (ICRS): (ra, dec) in deg
    (13.15833, -72.80028)>
```

In [80]:

```
pos_center=SkyCoord.from_name('Galactic Centre')    #qendra e galaksise
print(pos_center)
```

```
<SkyCoord (ICRS): (ra, dec) in deg
    (266.41500889, -29.00611111)>
```



To transform the coordinates to a different coordinate system we can use SkyCoord.transform_to():

In [81]:

```
pos_center=pos_center.transform_to('galactic')  #faktikisht duhet te jene 0,0
print(pos_center)
```

```
<SkyCoord (Galactic): (l, b) in deg
    (359.94487501, -0.04391769)>
```

In [82]:

```
pos_gal=position_crab.transform_to('galactic')
print(pos_gal)
```

```
<SkyCoord (Galactic): (l, b) in deg
    (184.55754381, -5.78427369)>
```

In [83]:

```
pos_LMC=pos_LMC.transform_to('galactic')
print(pos_LMC)
```

```
<SkyCoord (Galactic): (l, b) in deg
    (280.46521994, -32.88840086)>
```

For convenience we can also directly use the .galactic or .icrs attributes:

In [84]:

```
position_crab.galactic
```

Out[84]:

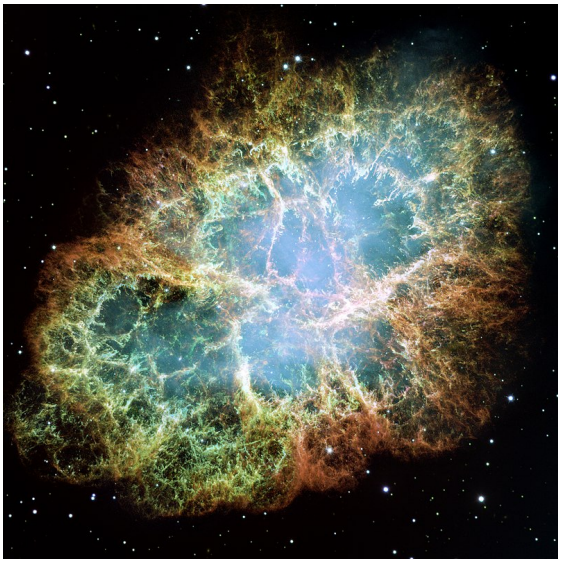
```
<SkyCoord (Galactic): (l, b) in deg
    (184.55754381, -5.78427369)>
```

Insert Image in a Jupyter Notebook with code

In [85]:

```
from IPython import display
display.Image("https://upload.wikimedia.org/wikipedia/commons/thumb/0/00/Crab_Nebula.jpg/800px-Crab_Nebula.jpg", width="300")
```

Out[85]:



In [86]:

```
position_crab.icrs
```

Out[86]:

```
<SkyCoord (ICRS): (ra, dec) in deg
    (83.63320833, 22.01447222)>
```

To access the longitude and latitude angles individually

In [87]:

```
position_crab.data.lon
```

Out[87]:

$5^{\mathrm{h}}34^{\mathrm{m}}31.97^{\mathrm{s}}$

In [88]:

```
position_crab.data.lat
```

Out[88]:

$22^\circ 00' 52.1''$

Measuring distances between positions in the sky

The angular distance between two SkyCoord objects, can be found using the SkyCoord.separation() method:

In [89]:

```
pos_LMC.separation(pos_SMC)
```

Out[89]:

$20^\circ 45' 57.3794''$

In [90]:

```
pos_LMC.separation(pos_SMC).to('rad')
```

Out[90]:

$0.362434\mathrm{rad}$

In [91]:

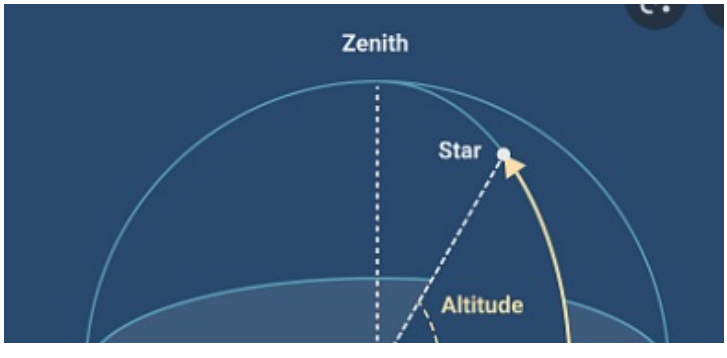
```
pos_center.separation(pos_SMC)
```

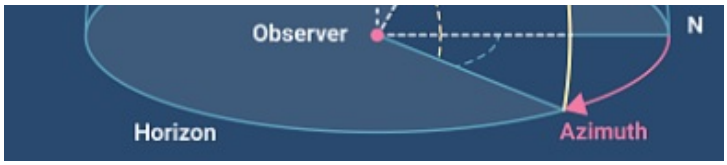
Out[91]:

$67^\circ 07' 31.4259''$

ALT - AZ coordinates (horizontal coordinate system)

In various circumstances, e.g. for planing observations, it can be usefull to transform a sky coordinate into a position in the horizontal coordinates system given a location on earth and a time.





In [92]:

```
from astropy.coordinates import EarthLocation, AltAz
from astropy.time import Time
```

We define a location using EarthLocation

In [93]:

```
paris=EarthLocation(lat=48.8567*u.deg, lon=2.3508*u.deg)
print(paris.geodetic)
```

```
GeodeticLocation(lon=<Longitude 2.3508 deg>, lat=<Latitude 48.8567 deg>, height=<Quantity
7.2605446e-10 m>)
```

In [94]:

```
from datetime import datetime
now = datetime.now()
current_time = now.strftime("%H:%M:%S")
print("Current Time =", current_time)
```

```
Current Time = 15:43:54
```

Now we can define a horizontal coordinate system using the AltAz class and use it to convert from the sky coordinate:

In [95]:

```
altaz=AltAz(obstime=now, location=paris)
crab_altaz=position_crab.transform_to(altaz)
print(crab_altaz)
```

```
<SkyCoord (AltAz: obstime=2022-06-17 15:43:54.261667, location=(4200910.64325784, 172456.
78503911, 4780088.65877593) m, pressure=0.0 hPa, temperature=0.0 deg_C, relative_humidity
=0.0, obswl=1.0 micron): (az, alt) in deg
    (262.93634699, 35.81603996)>
```

Tables

Astropy provides the Tables class in order to handle data tables.

Table objects can be created as shown in the following,

In [96]:

```
from astropy.table import Table
```

In [97]:

```
table=Table()
```

We add columns to the table like we would add entries to a dictionary

In [98]:

```
table['Source_Name']=['Crab', 'Sag A*', 'Cas A', 'Vela Junior']
table['GLON']=[184.5575438, 0, 111.74169477, 266.25914205]*u.deg
table['GLAT']=[-5.78427369, 0, -2.13544151, -1.21985818]*u.deg
table['Source_class']=['pwn', 'unc', 'snr', 'snr']
```

```
table['Flux']=[1.2, 3.4, 0.5, 3.2]
```

By executing the following cell, we get a nicely formatted version of the table printed in the notebook

In [99]:

```
table
```

Out[99]:

Table length=4

Source_Name	GLON	GLAT	Source_class	Flux
	deg	deg		
str11	float64	float64	str3	float64
Crab	184.5575438	-5.78427369	pwn	1.2
Sag A*	0.0	0.0	unc	3.4
Cas A	111.74169477	-2.13544151	snr	0.5
Vela Junior	266.25914205	-1.21985818	snr	3.2

In [100]:

```
np.array(table)          # if you want to work in plane numpy array
```

Out[100]:

```
array([('Crab', 184.5575438 , -5.78427369, 'pwn', 1.2),
       ('Sag A*',   0.        ,  0.        , 'unc', 3.4),
       ('Cas A', 111.74169477, -2.13544151, 'snr', 0.5),
       ('Vela Junior', 266.25914205, -1.21985818, 'snr', 3.2)],
      dtype=[('Source_Name', '<U11'), ('GLON', '<f8'), ('GLAT', '<f8'), ('Source_class',
'<U3'), ('Flux', '<f8')])
```

Accessing rows and columns

We have access to the defined columns. To check which ones are available you can use Table.colnames

In [101]:

```
table.colnames
```

Out[101]:

```
['Source_Name', 'GLON', 'GLAT', 'Source_class', 'Flux']
```

And access individual columns just by their name:

In [102]:

```
table['GLON']
```

Out[102]:

<Column name='GLON' dtype='float64' unit='deg' length=4>

184.5575438

0.0

111.74169477

266.25914205

In [103]:

```
table['Flux']
```

Out[103]:

<Column name='Flux' dtype='float64' length=4>
1.2
3.4
0.5
3.2

And also a subset of columns:

In [104]:

```
table[['Source_Name', 'GLON']]
```

Out[104]:

Table length=4

Source_Name	GLON
	deg
str11	float64
Crab	184.5575438
Sag A*	0.0
Cas A	111.74169477
Vela Junior	266.25914205

Often, it is handy to get the column data as astropy.units.Quantity using the .quantity property:

In [105]:

```
table['GLON'].quantity.to('arcmin')
```

Out[105]:

$[11073.453,\ 0,\ 6704.5017,\ 15975.549]\ ^{\prime}$

In [106]:

```
selection=table['Source_Name']=='Crab'     #may select only a row
table[selection]
```

Out[106]:

Table length=1

Source_Name	GLON	GLAT	Source_class	Flux
	deg	deg		
str11	float64	float64	str3	float64
Crab	184.5575438	-5.78427369	pwn	1.2

In [107]:

```
selection
```

Out[107]:

```
array([ True, False, False, False])
```

In [108]:

```
table.add_row(('LMC', 280.46521994, -32.88840086, 'snr', 1.5))  #add a row at the end of
table
```

In [109]:

```
table
```

Out[109]:

Table length=5

Source_Name	GLON	GLAT	Source_class	Flux
	deg	deg		
str11	float64	float64	str3	float64
Crab	184.5575438	-5.78427369	pwn	1.2
Sag A*	0.0	0.0	unc	3.4
Cas A	111.74169477	-2.13544151	snr	0.5
Vela Junior	266.25914205	-1.21985818	snr	3.2
LMC	280.46521994	-32.88840086	snr	1.5

In [110]:

```
from astropy.table import Column
```

In [111]:

```
col_c = Column(name='flux_1', data=[1.2, 2.3, 1.4, 2.8, 1.2])
table.add_column(col_c)
```

In [112]:

```
table
```

Out[112]:

Table length=5

Source_Name	GLON	GLAT	Source_class	Flux	flux_1
	deg	deg			
str11	float64	float64	str3	float64	float64
Crab	184.5575438	-5.78427369	pwn	1.2	1.2
Sag A*	0.0	0.0	unc	3.4	2.3
Cas A	111.74169477	-2.13544151	snr	0.5	1.4
Vela Junior	266.25914205	-1.21985818	snr	3.2	2.8
LMC	280.46521994	-32.88840086	snr	1.5	1.2

In [113]:

```
table['logflux']=np.log10(table['Flux'])
table
```

Out[113]:

Table length=5

Source_Name	GLON	GLAT	Source_class	Flux	flux_1	logflux
	deg	deg				
str11	float64	float64	str3	float64	float64	float64
Crab	184.5575438	-5.78427369	pwn	1.2	1.2	0.07918124604762482
Sag A*	0.0	0.0	unc	3.4	2.3	0.5314789170422551
Cas A	111.74169477	-2.13544151	snr	0.5	1.4	-0.3010299956639812
Vela Junior	266.25914205	-1.21985818	snr	3.2	2.8	0.505149978319906
LMC	280.46521994	-32.88840086	snr	1.5	1.2	0.17609125905568124

In [114]:

```
table['logflux'].format='%.2f'
table
```

Out[114]:

Table length=5

Source_Name	GLON	GLAT	Source_class	Flux	flux_1	logflux
	deg	deg				
str11	float64	float64	str3	float64	float64	float64
Crab	184.5575438	-5.78427369	pwn	1.2	1.2	0.08
Sag A*	0.0	0.0	unc	3.4	2.3	0.53
Cas A	111.74169477	-2.13544151	snr	0.5	1.4	-0.30
Vela Junior	266.25914205	-1.21985818	snr	3.2	2.8	0.51
LMC	280.46521994	-32.88840086	snr	1.5	1.2	0.18

Reading/Writing tables to disk

Astropy tables can be converted into many formats. To write the table in FITS format we can use:

In [115]:

```
table.write('example.fits', overwrite=True, format='fits')
```

In [116]:

```
table.write('example.ecsv', overwrite=True, format='ascii.ecsv')   # table with metadata
```

In [117]:

```
Table.read('example.fits')
```

Out[117]:

Table length=5

Source_Name	GLON	GLAT	Source_class	Flux	flux_1	logflux
	deg	deg				
bytes11	float64	float64	bytes3	float64	float64	float64
Crab	184.5575438	-5.78427369	pwn	1.2	1.2	0.08
Sag A*	0.0	0.0	unc	3.4	2.3	0.53
Cas A	111.74169477	-2.13544151	snr	0.5	1.4	-0.30
Vela Junior	266.25914205	-1.21985818	snr	3.2	2.8	0.51
LMC	280.46521994	-32.88840086	snr	1.5	1.2	0.18

In [118]:

```
#!cat example.ecsv
```

In [119]:

```
more example.ecsv
```

Indexing and Grouping

The Table object implements and .add_index() method, that allows to define an "index column" to access rows by the value contained in the index column:

In [120]:

```
table.add_index(colnames='Source_Name')   # put index for each row
```

Using the defined index column the row can now be accessed in a Pandas style, using the .loc[ ] syntax:

In [121]:

```
table.loc['Cas A']
```

Out[121]:

Row index=2

Source_Name	GLON	GLAT	Source_class	Flux	flux_1	logflux
	deg	deg				
str11	float64	float64	str3	float64	float64	float64
Cas A	111.74169477	-2.13544151	snr	0.5	1.4	-0.30

This works for multiple keys as well:

In [122]:

```
table.loc[['Cas A', 'Crab']]
```

Out[122]:

Table length=2

Source_Name	GLON	GLAT	Source_class	Flux	flux_1	logflux
	deg	deg				
str11	float64	float64	str3	float64	float64	float64
Cas A	111.74169477	-2.13544151	snr	0.5	1.4	-0.30
Crab	184.5575438	-5.78427369	pwn	1.2	1.2	0.08

In [123]:

```
table
```

Out[123]:

Table length=5

Source_Name	GLON	GLAT	Source_class	Flux	flux_1	logflux
	deg	deg				
str11	float64	float64	str3	float64	float64	float64
Crab	184.5575438	-5.78427369	pwn	1.2	1.2	0.08
Sag A*	0.0	0.0	unc	3.4	2.3	0.53
Cas A	111.74169477	-2.13544151	snr	0.5	1.4	-0.30
Vela Junior	266.25914205	-1.21985818	snr	3.2	2.8	0.51
LMC	280.46521994	-32.88840086	snr	1.5	1.2	0.18

Astropy's table object also supports the possibilityto group the rows by a given key column. The groups will be defined by the unique values contained in the column defined as key:

In [124]:

```
table_grouped=table.group_by('Source_class')  # ben renditjen sipas Source_class dhe i nd
an ne grupe, kemi 3 grupe,
                                              # dhe pastaj i printon njeri pas tjeteri
t
print(table_grouped)
```

```
Source_Name      GLON          GLAT      Source_class Flux flux_1 logflux
```

```
Source_Name      GLON          GLAT     Source_class Flux flux_1 logflux
                 deg           deg
----------- ------------ ------------ ------------ ---- ------ -------
       Crab  184.5575438  -5.78427369          pwn  1.2    1.2    0.08
      Cas A 111.74169477  -2.13544151          snr  0.5    1.4   -0.30
Vela Junior 266.25914205  -1.21985818          snr  3.2    2.8    0.51
        LMC 280.46521994 -32.88840086          snr  1.5    1.2    0.18
     Sag A*          0.0          0.0          unc  3.4    2.3    0.53
```

In [125]:

```
for group in table_grouped.groups:
    print(group,'\n')
```

```
Source_Name      GLON          GLAT     Source_class Flux flux_1 logflux
                 deg           deg
----------- ------------ ------------ ------------ ---- ------ -------
       Crab 184.5575438 -5.78427369          pwn  1.2    1.2    0.08

Source_Name      GLON          GLAT     Source_class Flux flux_1 logflux
                 deg           deg
----------- ------------ ------------ ------------ ---- ------ -------
      Cas A 111.74169477  -2.13544151          snr  0.5    1.4   -0.30
Vela Junior 266.25914205  -1.21985818          snr  3.2    2.8    0.51
        LMC 280.46521994 -32.88840086          snr  1.5    1.2    0.18

Source_Name GLON GLAT Source_class Flux flux_1 logflux
            deg  deg
----------- ---- ---- ------------ ---- ------ -------
     Sag A*  0.0  0.0          unc  3.4    2.3    0.53
```

In [126]:

```
print(table_grouped.groups[0])    # First group , indices=0
```

```
Source_Name      GLON          GLAT     Source_class Flux flux_1 logflux
                 deg           deg
----------- ------------ ------------ ------------ ---- ------ -------
       Crab 184.5575438 -5.78427369          pwn  1.2    1.2    0.08
```

In [127]:

```
print(table_grouped.groups[1])            # Second group , indices=1
```

```
Source_Name      GLON          GLAT     Source_class Flux flux_1 logflux
                 deg           deg
----------- ------------ ------------ ------------ ---- ------ -------
      Cas A 111.74169477  -2.13544151          snr  0.5    1.4   -0.30
Vela Junior 266.25914205  -1.21985818          snr  3.2    2.8    0.51
        LMC 280.46521994 -32.88840086          snr  1.5    1.2    0.18
```

In [128]:

```
print(table_grouped.groups.keys)        # Third group , indices=3
```

```
Source_class
------------
         pwn
         snr
         unc
```

There are a few other useful operations when working with Astropy tables.

Sort by key:

In [129]:

```
table_sorted=table.sort('Flux')
```

In [130]:

```
table
```

Out[130]:

Table length=5

Source_Name	GLON	GLAT	Source_class	Flux	flux_1	logflux
	deg	deg				
str11	float64	float64	str3	float64	float64	float64
Cas A	111.74169477	-2.13544151	snr	0.5	1.4	-0.30
Crab	184.5575438	-5.78427369	pwn	1.2	1.2	0.08
LMC	280.46521994	-32.88840086	snr	1.5	1.2	0.18
Vela Junior	266.25914205	-1.21985818	snr	3.2	2.8	0.51
Sag A*	0.0	0.0	unc	3.4	2.3	0.53

Note that .sort() is an in place operation on the table, i.e. changes the actual table.

To remove a specific row by index:

In [131]:

```
table.remove_row(0)
```

In [132]:

```
table
```

Out[132]:

Table length=4

Source_Name	GLON	GLAT	Source_class	Flux	flux_1	logflux
	deg	deg				
str11	float64	float64	str3	float64	float64	float64
Crab	184.5575438	-5.78427369	pwn	1.2	1.2	0.08
LMC	280.46521994	-32.88840086	snr	1.5	1.2	0.18
Vela Junior	266.25914205	-1.21985818	snr	3.2	2.8	0.51
Sag A*	0.0	0.0	unc	3.4	2.3	0.53

Astropy tables also suport row-wise interaction in python loop:

In [133]:

```
for row in table:
    print(row['Source_Name'])
```

```
Crab
LMC
Vela Junior
Sag A*
```

Another useful feature for quickly inspecting the data contained in the table is the .show_in_browser() method:

In [134]:

```
table.show_in_browser(jsviewer=True)
```

In [135]:

```
table.show_in_notebook()
```

Out[135]:

Table length=4

Show 50 entries Search:

idx	Source_Name	GLON	GLAT	Source_class	Flux	flux_1	logflux
		deg	deg				
0	Crab	184.5575438	-5.78427369	pwn	1.2	1.2	0.08
1	LMC	280.46521994	-32.88840086	snr	1.5	1.2	0.18
2	Vela Junior	266.25914205	-1.21985818	snr	3.2	2.8	0.51
3	Sag A*	0.0	0.0	unc	3.4	2.3	0.53

Showing 1 to 4 of 4 entries First Previous 1 Next Last

FITS Images and WCS

The flexible image transport system format (FITS) is widely used data format for astronomical images and tables. As example we will use image data of the supernova remnant Cassiopeia A taken by the Chandra X-ray observatory.

In [136]:

```
from astropy.io import fits
```

To open the fits file we use fits.open() and just specify the filename as an argument:

In [137]:

```
from astropy.utils.data import download_file
image_file = download_file('https://chandra.harvard.edu/photo/2009/casa/fits/casa_0.5-1.5keV.fits', cache=True )
```

In [138]:

```
hdu_list = fits.open(image_file)    #An HDU (Header Data Unit) is the highest level component of the FITS file structure,
hdu_list.info()                     # summarizes the content of the opened FITS file
```

```
Filename: C:\Users\User\.astropy\cache\download\py3\190e04919e6b7e3edd44a04cf28a5099
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU      26   (1024, 1024)   float32
```

In [139]:

```
image_hdu=hdu_list['PRIMARY']
#or
#image_hdu=hdu_list[0]
```

Generally, the image information is located in the PRIMARY block. The blocks are numbered and can be accessed by indexing hdu_list.

In [140]:

```
image_data = hdu_list[0].data
```

Our data is now stored as a 2D numpy array. But how do we know the dimensions of the image? We can look at the shape of the array.

In [141]:

```
print(type(image_data))
print(image_data.shape)
```

```
<class 'numpy.ndarray'>
(1024, 1024)
```

In [142]:

```
hdu=hdu_list[0]
```

In [143]:

```
hdu.header
```

Out[143]:

```
SIMPLE  =                    T / Fits standard
BITPIX  =                  -32 / Bits per pixel
NAXIS   =                    2 / Number of axes
NAXIS1  =                 1024 / Axis Length
NAXIS2  =                 1024 / Axis Length
OBJECT  = 'Cassiopeia A'
DATE-OBS= '2007-12-05T22:01:59'
DATE-END= '2007-12-08T20:07:24'
LTM1_1  =                    1
LTM1_2  =                    0
LTM2_1  =                    0
LTM2_2  =                    1
LTV1    =                -3651
LTV2    =                -3807
CTYPE1  = 'RA---TAN'
CTYPE2  = 'DEC--TAN'
CUNIT1  = 'deg'
CUNIT2  = 'deg'
CRPIX1  =                445.5
CRPIX2  =                289.5
CRVAL1  =          350.8841248
CRVAL2  =          58.78133011
CDELT1  =     -0.0001366666693
CDELT2  =      0.0001366666693
EQUINOX =                 2000
RADESYS = 'ICRS'
```

The meaning of all header can get here.

In [144]:

```
hdu.header ['DATE-OBS']
```

Out[144]:

```
'2007-12-05T22:01:59'
```

We can access the data with .data attribute.

In [145]:

```
image_hdu.data  #
```

Out[145]:

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 1., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

In [146]:

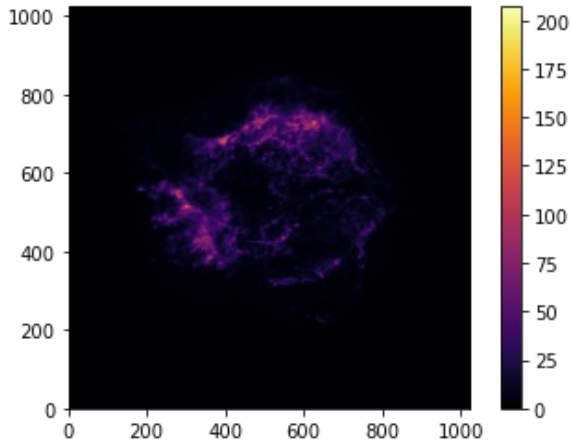
```
#%matplotlib inline                    # It is a magic function that renders the figure in
a notebook
import matplotlib.pyplot as plt
```

In [147]:

```
plt.imshow(image_hdu.data, origin='lower', cmap='inferno')      # colormap
plt.colorbar()
```

Out[147]:

```
<matplotlib.colorbar.Colorbar at 0x129de1c0>
```



Additional meta information is stored in the .header attribute:

In [148]:

```
#image_hdu.header      # are the same with hdu.header
```

We now use the header information to create a world coordinate to pixel coordinate transformation, using the astropy.wcs.WCS class:

astropy.wcs contains utilities for managing World Coordinate System (WCS) transformations in FITS files. These transformations map the pixel locations in an image to their real-world units, such as their position on the sky sphere.

In [149]:

```
from astropy.wcs import WCS
```

In [150]:

```
wcs = WCS(image_hdu.header)
print(wcs)
```

```
WCS Keywords

Number of WCS axes: 2
CTYPE : 'RA---TAN'  'DEC--TAN'
CRVAL : 350.8841248  58.78133011
CRPIX : 445.5  289.5
PC1_1 PC1_2  : 1.0  0.0
PC2_1 PC2_2  : 0.0  1.0
CDELT : -0.0001366666693  0.0001366666693
NAXIS : 1024  1024
```

Using the helper methods SkyCoord.to_pixel() and SkyCoord.from_pixel(), we can now convert every position in the image to the correspondinding sky coordinate:

In [151]:

```
SkyCoord.from_pixel(0, 0, wcs)    # for pixel 0, 0   find the Ra, dec cordinates
```

Out[151]:

```
<SkyCoord (ICRS): (ra, dec) in deg
    (351.00119731, 58.74184873)>
```

In [152]:

```
position_casa=SkyCoord.from_name('Cassiopeia A')
print(position_casa)
position_casa.to_pixel(wcs)
```

```
<SkyCoord (ICRS): (ra, dec) in deg
    (350.85, 58.815)>
```

Out[152]:

```
(array(573.79215824), array(534.89801134))
```

Ploting of sky images

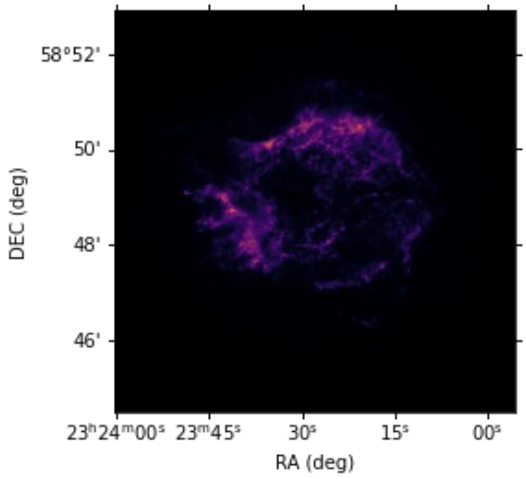
The Matplotlib package is a very popular plotting package for Python. Astropy provides a helper module astropy.vizualization.wcsaxes to simplify plotting of sky images with Matplotlib.

To use it we just pass projection=wcs to the plt.subplot() function:

In [153]:

```
ax = plt.subplot(projection=wcs)
ax.imshow(image_hdu.data, cmap='inferno', origin='lower')

ax.set_xlabel('RA (deg)')
ax.set_ylabel('DEC (deg)')
```

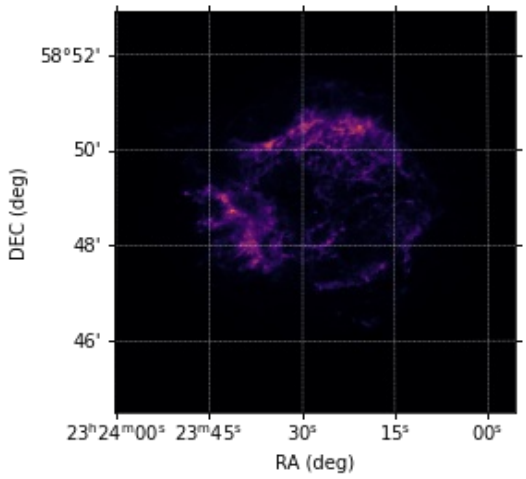


We can add a coordinate grid with ax.grid():

In [154]:

```
ax.grid(linewidth=0.3, linestyle='dashed', color='white')
ax.figure
```

Out[154]:



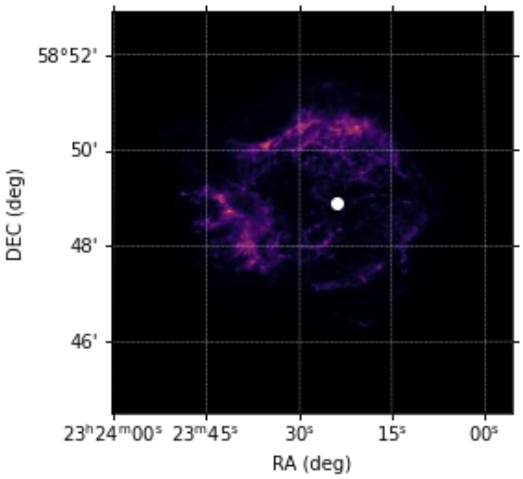
We add a white point to mark the center position of Cassiopeia A:

We add a white point to mark the center position of "Cassiopeia A"

In [155]:

```
ra=position_casa.icrs.ra.deg
dec=position_casa.icrs.dec.deg
ax.scatter(ra, dec, transform=ax.get_transform('icrs'), color='white')
ax.figure
```

Out[155]:



Thank you!