Reducción de datos CCD con IRAF y Python

Alexis Andrés

2 de septiembre de 2024

Índice general

Unidad 1

Introducción a Python

Unidad 2

Introducción a la astronomía observacional

Unidad 3

Python para datos CCD

Clase 9 | Reducción de datos con Python

Para realizar las tareas básicas de reducción de datos astronómicos usando Python se utiliza el módulo llamado «`ccdproc`». Este módulo es una herramienta poderosa diseñada para la reducción y el procesamiento de datos de imágenes CCD. Está desarrollado como parte del ecosistema de «`astropy`» y proporciona una serie de funciones y clases para manejar tareas comunes en el preprocesamiento de datos astronómicos, como la calibración de imágenes, la corrección de bias, la corrección de flat-field, y la combinación de imágenes. Primero revisaremos el módulo astropy.

9 El paquete Astropy

Astropy es un paquete diseñado específicamente para la astronomía y la astrofísica. Cuenta con muchas funciones y módulos, pero nosotros revisaremos algunos de los más utilizados y que nos serán de ayuda para comprender el funcionamiento de `ccdproc`. Para instalar `astropy`, si es que no lo tienes aún, es suficiente con escribir **`pip install astropy`** en una terminal.

Algunos de los módulos más populares dentro del ecosistema de astropy son `units`, `Coordinates`, y `fits`. A continuación realizaremos algunas tareas básicas con estos módulos.

9.1 El módulo astropy.units

El módulo «`astropy.units`» permite trabajar con unidades físicas tales como los kilogramos, metros, segundos y cualquiera de sus múltiplos, submúltiplos y además hace posible transformar entre diferentes unidades equivalentes de otros sistemas de medición.

Primero debemos importarlo:

[1]:
```
import astropy.units as u
```

Ahora podemos definir una variable que tenga unidades de distancia. Por ejemplo, podemos asignarle unidades de kilopársec:

[2]:
```
#- Distancia en kilopársec
distance = 1 * u.kpc
distance
```

[2]:
```
<Quantity 1. kpc>
```

Como se ve en la celda de código anterior, se crea una «cantidad» al multiplicar cualquier variable numérica por una unidad definida en `astropy`. En este caso, el kilopársec se obtiene con «`u.kpc`». Si ahora queremos saber a cuántos metros equivale un kilopársec, entonces podemos usar la función «`u.to()`» de la siguiente manera:

[3]:
```
#- Convertir a metros
distance.to(u.m)
```

[3]:
```
<Quantity 3.08567758e+19 m>
```

Existe otra forma de utilizar la función `u.to()`, que es un poco más flexible. Para ilustrarlo, intentemos convertir la variable `distance` a varias unidades de distancia, tales como los centímetros, metros, kilómetros, años luz, párSec, nanómetros y unidades astronómicas. Una forma de hacerlo es la siguiente:

[4]:
```
distance_units = ['cm', 'm', 'km', 'lyr', 'pc', 'nm', 'au']

#- Distancia equivalente en varias unidades
print(f'{distance} es equivalente a:\n=========================')

for unit in distance_units:
    print( distance.to(u.Unit(unit)) )
```

```
1.0 kpc es equivalente a:
=========================
3.0856775814913673e+21 cm
3.085677581491367e+19 m
3.085677581491367e+16 km
3261.5637771674333 lyr
1000.0 pc
3.085677581491367e+28 nm
206264806.24709633 AU
```

Por supuesto que se pueden utilizar unidades derivadas, tales como las de la velocidad (m/s, por ejemplo) o las de la fuerza (N $\equiv$ kg m s^{-2}). Para esto usamos los operadores de multiplicación y división según sea el caso. Específicamente, para definir cantidades con unidades de velocidad:

[5]:
```
velocity = 60 * u.km / u.h
velocity
```

[5]: <Quantity 60. km / h>

Obteniendo su equivalente en $\mathbb{mm}/\mathbb{s}$:

[6]:
```
velocity.to(u.m/u.s)
```

[6]: <Quantity 16.66666667 m / s>

9.2 Trabajando con tablas

Con astropy se pueden leer tablas casi en cualquier formato. Un formato muy popular en astronomía son los archivos con extensión fits (**F**lexible **I**mage **T**ransport **S**ystem). Una de las formas de leer este tipo de archivos es usando la función `Table()` del módulo `astropy.table`. Las imágenes CCD se almacenan en archivos `.fits`. Sin embargo, no es posible leerlas con `Table()`. Afortunadamente, existen más opciones.

Para poder trabajar con imágenes `.fits` provenientes de archivos CCD, se puede utilizar el módulo `fits` que está dentro de `astropy.io`. Toma como ejemplo las imágenes disponibles en este enlace de Google drive: https://drive.google.com/drive/folders/1TxopD_fIa1ZHplm_SH0Gvhe663psuJn-?usp=sharing.

El enlace contiene una carpeta organizada de la siguiente manera: hay un directorio llamado `bias`, otro llamado `flats`, otro llamado `object` y uno llamado `stds`. Los directorios `bias` y `flat` contienen las imágenes bias y flat, respectivamente. El directorio `object` contiene las imágenes del objeto de interés, también llamadas «*science images*» o imágenes de ciencia. El directorio `stds` contiene imágenes de una estrella estándar, que no usaremos por el momento.

Recomiendo que extraigas el contenido de la carpeta en tu mismo directorio de trabajo donde utilizas los Notebooks de Jupyter para que puedas acceder a sus datos de manera más simple y con las rutas especificadas en el Notebook llamado «Astropy-package».

Primero establecemos las rutas a las imágenes:

[7]:
```
#- Ruta a la ubicación de los datos
bias_image = 'OB0001/bias/0002611406-20200712-OSIRIS-OsirisBias1.fits'
```

[8]:
```
flat_image =
'OB0001/flat/0002615401-20200712-OSIRIS-OsirisSkyFlat1.fits'
object_image =
'OB0001/object/0002611437-20200712-OSIRIS-OsirisBroadBandImage1.fits'
```

Para leer estas imágenes con el módulo `fits`, se hace de la siguiente manera:

[9]:
```
#- Leyendo la imagen con astropy
bias_data   = fits.open(bias_image)[0]
flat_data   = fits.open(flat_image)[0]
object_data = fits.open(object_image)[0]
```

Debes tener en cuenta que las instrucciones anteriores generarán errores si no guardaste los datos en tu directorio actual de trabajo. Si en cualquier momento ocurre un error de compilación, siempre revisa el mensaje de error. Como hemos visto en ocasiones anteriores, Python brinda mucha información en sus mensajes de errores para que puedas corregirlos.

Puedes verificar la información contenida en cada imagen aplicando el atributo `header`. Por ejemplo para la imagen bias:

[10]:
```
bias_data.header
```

```
SIMPLE  =                    T / Fits standard
BITPIX  =                   16 / Bits per pixel
NAXIS   =                    2 / Number of axes
NAXIS1  =                 1049 / Axis length
NAXIS2  =                 2051 / Axis length
EXTEND  =                    F / File may contain extensions
BSCALE  =           1.000000E0 / REAL = TAPE*BSCALE + BZERO
BZERO   =           3.276800E4 /
ORIGIN  = 'NOAO-IRAF FITS Image Kernel July 2003' / FITS file
originator
DATE    = '2021-10-07T01:52:03' / Date FITS file was generated
IRAF-TLM= '2021-10-07T01:52:03' / Time of last modification
OBJECT  = 'BIAS    '           / Name of the object observed
ORIGFILE= 'Jul12_204354.fits'  / Filename
INSTRUME= 'OSIRIS  '           / Instrument Name
DETECTOR= 'E2V CCD44_82_BI'    / Detectors Model
DETSIZE = '[1:4096,1:4102]'    / Maximum Imaging Pixel Area
DATE-OBS= '2020-07-12T20:43:39.592' / Time when starts the first
exposure (in fr
ELAPSED = '22.969  '           / Total elapsed time from start to end
(s)
...
```

Para acceder a la imagen se necesita del atributo `data`:

[11]:
```
bias_data.data
```

```
array([[1039, 1137, 1156, ..., 1203,   29,  881],
       [1037, 1133, 1171, ..., 1192,   28,  892],
       [1044, 1135, 1170, ..., 1204,   29,  896],
       ...,
       [1069, 1160, 1184, ..., 1224,   28,  897],
       [1062, 1157, 1180, ..., 1231,   29,  892],
       [1068, 1160, 1188, ..., 1221, 1224,  102]], dtype=uint16)
```

Como puedes darte cuenta, la imagen CCD es un arreglo de numpy bidimensional. Para poder visualizarlo, necesitamos usar la función `imshow()`. Sin embargo, los datos tienen una cantidad bastante grande de pixeles:

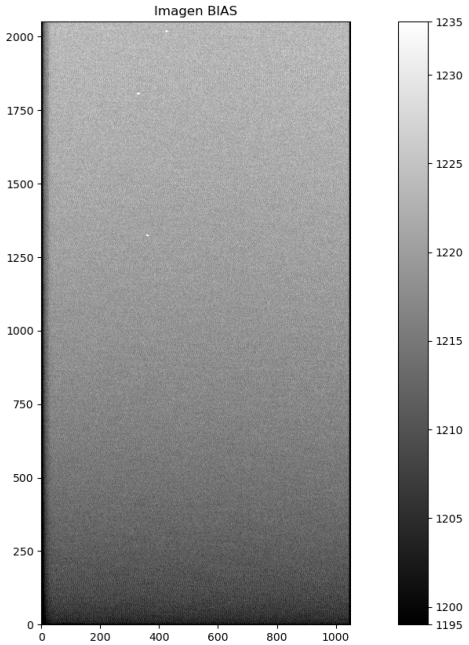
[12]:
```
bias_data.data.size
```

[12]:
```
2151499
```

Dada la cantidad de pixeles, es necesario aplicar una función para escalar la imagen. Para eso utilizamos la función `show_image` que está definida dentro del archivo llamado `show_image.py` en el directorio de Notebooks. Importamos esa función y la utilizamos:

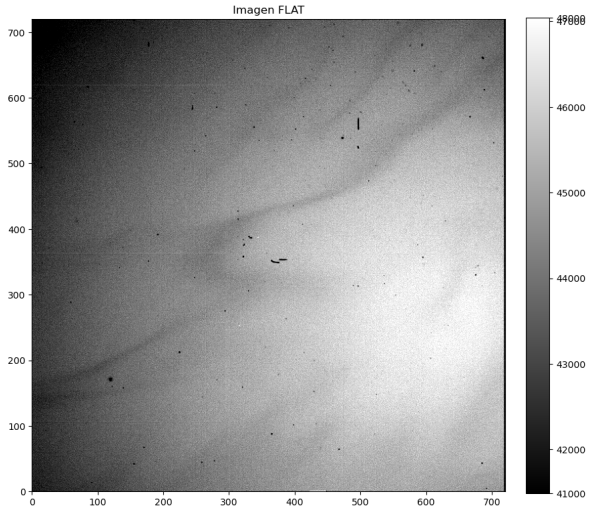
[13]:
```
from show_image import show_image

show_image(bias_data.data, cmap='gray')
plt.title('Imagen BIAS')
plt.show()
```



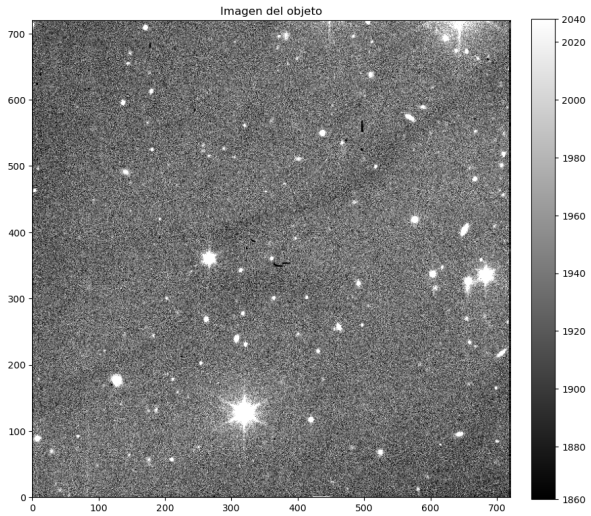
De igual manera podemos visualizar la imagen de campo plano:

[14]:
```
show_image(flat_data.data, cmap='gray')
plt.title('Imagen FLAT')
plt.show()
```



Y también la imagen del objeto:

[15]:
```
show_image(object_data.data, cmap='gray')
plt.title('Imagen del objeto')
plt.show()
```



9.3 Un ejemplo un poco realista

Ahora revisemos un ejemplo bastante sencillo sobre el proceso de reducir una imagen CCD. Para eso vamos a generar una imagen sintética, o en otras palabras, a simular una imagen

astronómica. Utilizaremos las funciones definidas dentro del módulo `image_sim` para generar una imagen de bias, una flat y una de dark current. Primero importamos el módulo:

[16]:
```
import image_sim as imsim
```

Comenzamos definiendo los parámetros de cada imagen:

[17]:
```
stars_exposure = 30.0
dark_exposure = 60.0
dark = 0.1
sky_counts = 20
bias_level = 1100
read_noise = 700
max_stars_counts = 2000
```

Generamos las imágenes de bias, dark current y flat:

[18]:
```
bias_with_noise = (imsim.bias(image, bias_level, realistic=True) +
                   imsim.read_noise(image, read_noise))

dark_frame_with_noise = (imsim.bias(image,bias_level,realistic=True) +
                         imsim.dark_current(image, dark, dark_exposure,
                         hot_pixels=True) +
                         imsim.read_noise(image, read_noise))

flat = imsim.sensitivity_variations(image)
```

Creamos la imagen con estrellas y ruido:

[19]:
```
realistic_stars = (imsim.stars(image,50,max_counts=max_stars_counts) +
                   imsim.dark_current(image, dark, stars_exposure,
                   hot_pixels=True) +
                   imsim.bias(image, bias_level, realistic=True) +
                   imsim.read_noise(image, read_noise)
                  )
```

Ya que las imágenes simuladas para bias y dark frames no tienen el mismo tiempo de exposición, se necesita hacer una escalación:

[20]:
```
scaled_dark_current = stars_exposure * (dark_frame_with_noise -
bias_with_noise) / dark_exposure
```

Y este es nuestro primer intento de reducir una imagen astronómica:

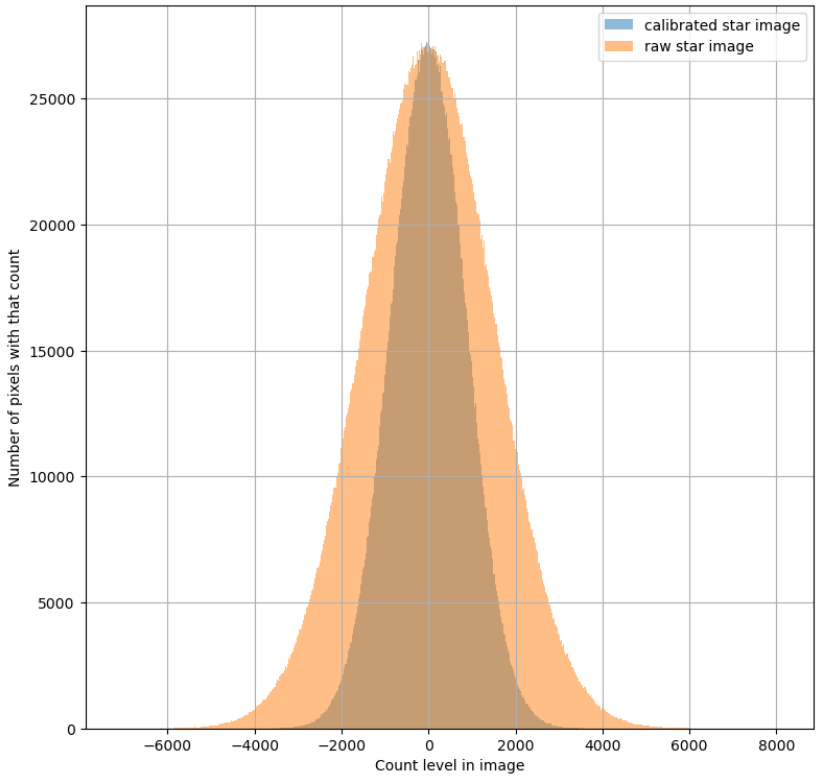
[21]:
```
calibrated_stars = (realistic_stars - bias_with_noise -
scaled_dark_current) / flat
```

Visualizar la imagen tal vez no nos brinde mucha información sobre si la imagen mejoró o no. Pero podemos observar el histograma de cuentas. Para lograrlo, importamos el siguiente módulo:

[22]:
```
from astropy.visualization import hist
```

Ahora graficamos el histograma para la imagen simulada y su correspondiente calibración:

[23]:
```
plt.figure(figsize=(9, 9))
hist(calibrated_stars.flatten(), bins='freedman', label='Imagen
calibrada', alpha=0.5)
hist(stars_with_noise.flatten(), bins='freedman', label='Imagen cruda',
alpha=0.5)
plt.legend()
plt.grid()
plt.xlabel('Nivel de cuentas en la imagen')
plt.ylabel('Número de pixeles con esas cuentas')
```



En la gráfica anterior, el ruido de la imagen CCD está representado por el espesor de la distribución del histograma. Vemos que el espesor de la imagen calibrada es menor al de la imagen sin calibrar. En otras palabras, hemos logrado reducir el ruido de la imagen CCD. En las siguientes clases veremos cómo realizar una calibración de manera correcta.