

# Documentación de Tests aplicados a los módulos de Aurora: `convolutions.py` y `datacube.py`

C.R. Carvajal-Bohorquez  
Juan Carlos Basto Pineda

*Universidad Industrial de Santander*<sup>1</sup>

29 de julio de 2020

## Resumen

Las observaciones sintéticas son una clase de métodos recientes que se aplican a las simulaciones de galaxias para incluir los efectos observacionales y así hacer una interpretación más adecuada de los datos computacionales a la hora de compararlos con las observaciones reales obtenidas por los grandes telescopios. Un marco que incorpora estos métodos es el código de *Python* Aurora, que cuenta con diferentes módulos para articular y aplicar los efectos observacionales reales. Dado que para conseguir las observaciones sintéticas se deben manipular los datos de las simulaciones, se debe tener plena seguridad que los procesos numéricos no alteran los datos, por lo tanto se requieren aplicar test para asegurar la funcionalidad del código. En este documento se encuentra en detalle las pruebas que se aplicaron a los módulos: `convolutions.py`, `array_operations.py` y a la convolución para representar la resolución espectral presente en `spectrum_tools.py`. Se anexan los notebooks donde se ejecutaron las pruebas al final del documento.

## 1. Convolución analítica en el modulo `spectrum_tools.py`

La convolución presente en este modulo recrea la resolución espectral de los instrumentos que hacen las observaciones reales. Dicha convolución se hace entre las líneas de emisión de las partículas en la simulación, que se simulan con una gaussiana normalizada multiplicada por una constante que representa el flujo luminoso, y la **LSF** (line-spread function) por sus siglas en ingles, que también es representada como una gaussiana normalizada. Dado que la convolución es sobre dos gaussianas hay solución analítica y el resultado es otra gaussiana, pero donde el sigma es la suma cuadrática los sigmas de las gaussianas originales.

$$I \circledast LSF = A \frac{e^{-\frac{(x-\mu)^2}{2\sigma^2}}}{\sqrt{2\pi}\sigma} \circledast \frac{e^{-\frac{x^2}{2\sigma'^2}}}{\sqrt{2\pi}\sigma'} = A \frac{e^{-\frac{(x-\mu)^2}{2(\sigma^2 + \sigma'^2)}}}{\sqrt{2\pi}\sqrt{\sigma^2 + \sigma'^2}} \quad (1)$$

Antes de continuar con el test de dicha convolución, se deben abordar las características de la **LSF** para simular la resolución espectral del instrumento. Para dicho motivo debemos recurrir a las características reales de los instrumentos de observación, en este caso al poder de resolución ( $R$ ) de los espectrógrafos, que tienen un rango de 3000 a 5000 para instrumentos de normales a buenos. El poder de resolución se define como:

$$R = \frac{\lambda}{\Delta\lambda}, \quad (2)$$

con  $\lambda$  es la longitud de onda de observación y  $\Delta\lambda$  es la mínima distancia entre líneas que podría discernir el instrumento. En el caso numérico del código  $\Delta\lambda$  es el FWHM (*full width at half maximum*) por sus siglas en ingles de la **LSF**. Para determinarlo se asumirá que la longitud de onda observada corresponde con  $H_\alpha$  a una temperatura de 10000 K y con redshift  $z = 1$ .

Para calcular la longitud de onda observada recurrimos a la siguiente ecuación:

$$\lambda_{H_\alpha} = (1 + z)\lambda_{H_{\alpha 0}} = 2 \cdot 6,56278 \times 10^{-5} \text{ cm} = 13125,56 \text{ \AA} \quad (3)$$

Para determinar  $\Delta\lambda$ , tomaremos  $R = 5000$  en la ecuación 2, por lo tanto:

$$\Delta\lambda = FWHM = \frac{\lambda_{H_\alpha}}{5000} = 2,625 \text{ \AA} \quad (4)$$

Ahora con el valor anterior y el factor para pasar de FWHM a desviación estándar (2.3548) se calcula el sigma de la **LSF**:

$$\sigma_{LSF} = \frac{FWHM}{2,3548} = \frac{2,625112 \times 10^{-8} \text{ cm}}{2,3548} = 1,114 \text{ \AA} \quad (5)$$

Para asegurar que el sigma de la **LSF** calculado es adecuado se debe comparar con la desviación estándar de la distribución de velocidades de las partículas que hicieron la emisión y obtener una relación de 3:1. Para lo anterior se asumirá que las partículas se encontraban a una temperatura de 10000  $K$  y que siguen una distribución Maxwell-Boltzmann. Así, la desviación estándar de la distribución de velocidades de las partículas estará dada por:

$$\sigma(T) = \sqrt{\frac{k_B \cdot T}{\mu \cdot m_p}}, \quad (6)$$

donde  $k_B$  es la constante de Boltzmann,  $T$  es la temperatura en  $K$ ,  $m_p$  es la masa del protón en gramos y  $\mu$  es el peso molecular promedio del hidrógeno ionizado. Para esta ultima variable, se asume que en la mezcla hidrógeno-helio todo el hidrógeno esta ionizado, por lo tanto  $\mu = 0,63$ . Con lo anterior se obtiene:

$$\sigma_{H_\alpha} = 11,45 \frac{km}{s} \quad (7)$$

Dado que los cálculos se están haciendo en términos de la longitud de onda, se debe transformar la anterior cantidad a unidades de longitud de onda, para ello se acude al efecto Doppler:

$$\frac{\Delta\lambda}{\lambda_0} = \frac{v}{c}$$

Aplicado a la desviación de velocidades:

$$\sigma_{H_\alpha} [cm] = \lambda_{obs} \cdot \frac{\sigma_{H_\alpha} [cm/s]}{c} = 0,502 \text{ \AA} \quad (8)$$

Ahora, haciendo la la relación los sigmas:

$$\frac{\sigma_{LSF}}{\sigma_{H_\alpha}} = 2,22 \quad (9)$$

Por lo anterior la relación es adecuada para  $R = 5000$ , aplicando todos los cálculos anteriores para el rango mencionado, los valores que respetan la relación 3:1 son para  $R \geq 3500$ . Como ya se fijo el poder de resolución adecuado para la convolución, procedemos al código Aurora. Una vez insertados los parámetros de entrada adecuados con la configuración mencionada anteriormente, se aplica la simulación de la emisión de las partículas, como perfiles gaussianos y se proyectan en los canales de velocidad del cubo de datos, según fue la configuración inicial, realizando la integral del perfil gaussiano en los intervalos de los canales. Para poner a prueba que la convolución no esta alterando el flujo y que se aplica de forma adecuada, se hará la convolución con la **LSF** una vez se simulo la emisión de las partículas y posteriormente se proyectara en los canales, si bien el sigma del nuevo perfil cambia, no debe cambiar el flujo de las partículas. Lo anterior se aplico sobre una de las partículas en la simulación, los resultados son los siguientes:

$$Flujo_0 = 0,9998$$

$$\text{Flujo luego de la convolucion} = 0,9997$$

$$\text{Diferencia} = 5,8153 \times 10^{-5}$$

$$\text{Error} = 0,0058 \%$$

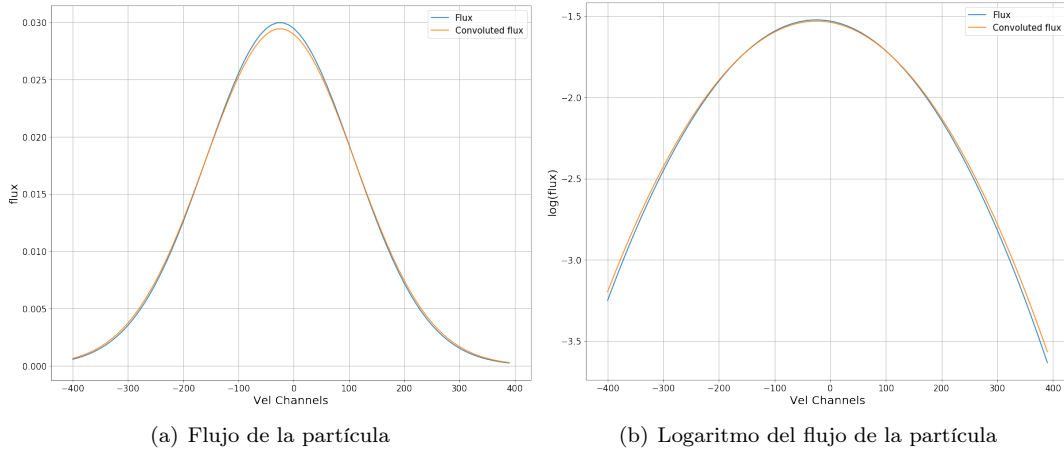


Figura 1: Perfil gaussiano de la emisión de una partícula proyectado en los canales de velocidad del cubo

A partir de los resultados se puede concluir que el flujo es levemente afectado por la convolución, además en las gráficas se puede observar que los perfiles son muy similares, y que hay un leve ensanchamiento por parte del perfil que sufrió la convolución.

## 2. Modulo convolutions.py

Los métodos presentes en este modulo son numerosos y los podemos dividir en 3 grupos: los que se encargan de simular la resolución espacial y los efectos por la atmósfera, los que simulan la resolución espectral, y por último los métodos auxiliares que realizan algunas operaciones para facilitar el desarrollo de los anteriores. Es importante aclarar que la forma de simular la resolución, en los dos casos, se hace por medio de una convolución numérica con un kernel gaussiano generado con la librería **Astropy**. Los test se realizaron sobre los métodos de convolución, aplicándolos sobre la convolución de dos gaussianas para comparar el resultado con su solución analítica como se presento en la ecuación 1.

### 2.1. Métodos para la convolución espectral

En este apartado intervienen 4 funciones, una que se encarga de crear el kernel de la **LSF** (`create_lsf`) y 3 que realizan la convolución entre la señal de entrada y el kernel dado (`spectral_astropy_convolution`, `fft_spectral_astropy_convolution` y `fft_spectral_aurora_convolution`). Para realizar los test primero hay que tomar algunas condiciones numéricas partiendo de las características que se quieren simular. La convolución busca recrear la respuesta de los espectrógrafos cuando interactúan con una línea de emisión, como instrumentos no son perfectos y no identifican exactamente la posición, se deben “ensanchar” los perfiles de emisión simulados. Recordando lo discutido al principio de este documento, cuando se convolucionan dos gaussianas el resultado es una nueva, donde el sigma resultante es la suma en cuadratura de los originales, por lo tanto resulta una gaussiana más ancha, justo lo que se realiza al aplicar esta operación entre las líneas de emisión con una **LSF**, sin embargo por simplicidad la

convolución se hará directamente sobre dos gaussianas, en el caso estricto se sugiere abordar la sección anterior para conocer los procedimientos de traducir el poder de resolución de los instrumentos a los parámetros de una gaussiana.

Por lo anterior el sigma de la señal de entrada, que representa la línea de emisión, debe ser mayor al del kernel, por trabajos anteriores y pequeñas pruebas, se sabe que debe ser al menos 3 veces mayor, sin embargo para asegurar los resultado en estas pruebas controladas se hará con una relación de 6:1. Por otro lado el tamaño para la señal gaussiana fue definido en 20 sigmas, esta característica también la comparte el kernel “analítico”, sin embargo este además esta normalizado numéricamente, es decir esta dividido por la suma de todos sus los valores, con ello se garantiza que el la sumatorio de los valores del kernel resultante da la unidad. Una vez fueron creados los arreglos correspondientes a la señal de entrada y al kernel, se crearon dos kernel más, uno directamente con la librería **Atropay** por medio de la función `create_lsf` y para el segundo, se tomo el resultado anterior y se aplico un **Padding**, una redimensionización de tal manera que la dimensión del nuevo kernel sea igual a la de la señal de entrada rellenando con ceros los nuevos valores añadidos al extenderlo, los resultados de los anteriores arreglos se presentan en la siguiente figura:

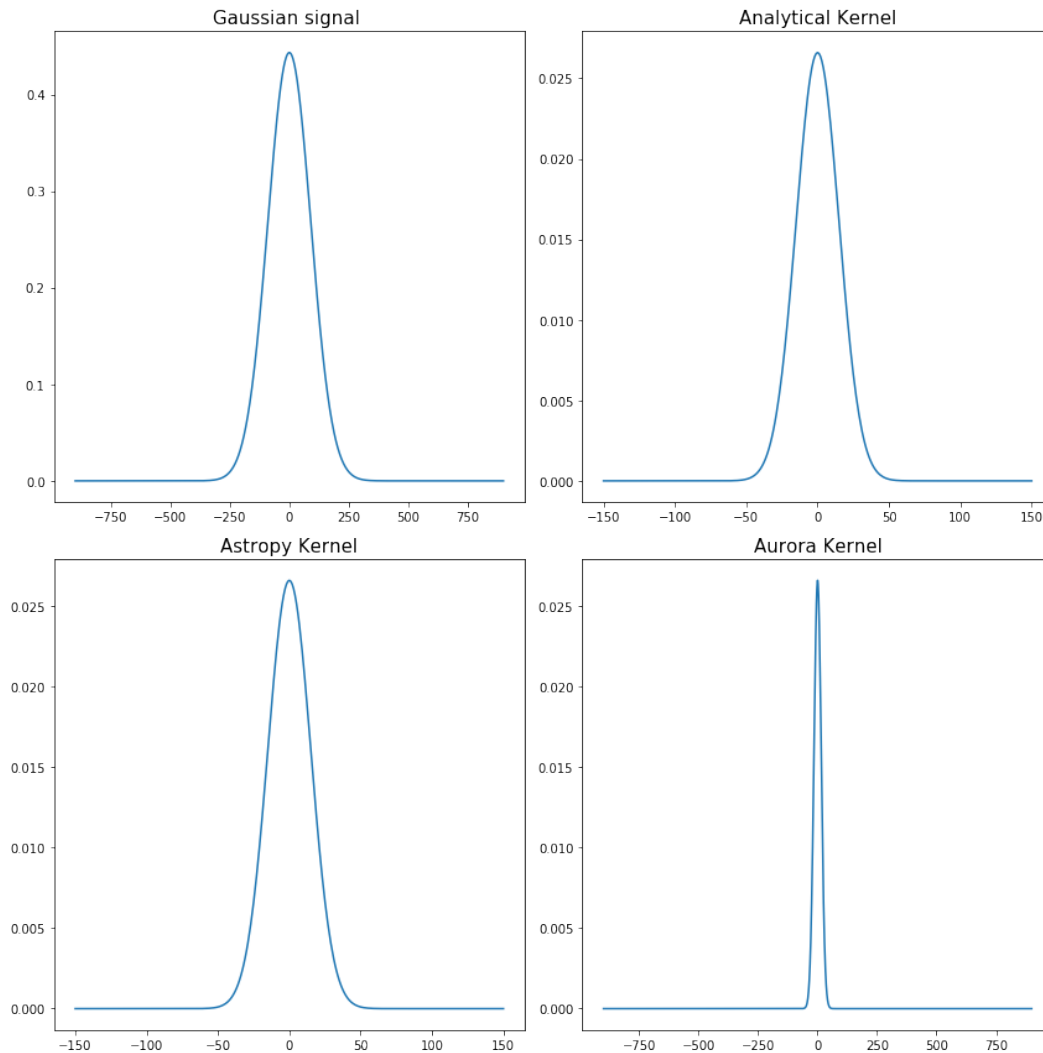


Figura 2: Señal gaussiana de entrada, y los tres kernel.

Es importante notar que el kernel creado con la función `create_lsf` recibe el nombre de *Astropy Kernel*,

y el redimensionalizado se nombro *Aurora Kernel*. Por otro lado, los dos anteriores son de base la misma señal, solo que el segundo se extendió con ceros. Los numerosos kernel son debido a las convoluciones que se realizaron para los test. El primero es solo representativo, ya que el resultado de la convolución analítica, es una nueva gaussiana según se ha discutido. El segundo se utiliza con los métodos que llevan en el nombre **astropy** y el tercero se emplea con la función [fft\\_spectral\\_aurora\\_convolution](#).

Como la convolución se realizo entre la señal gaussiana y con kernel normalizados, el “flujo” de la señal debe mantenerse constante. Los resultados de la sumatoria de la señal resultante luego de la convolución son los siguientes:

$$\begin{aligned}
 \text{Signal Gaussiana}_{sum} &= 100,0 \\
 \text{Convolucin Analtica}_{sum} &= 100,0 \\
 \text{Convolucin Astropy}_{sum} &= 100,0 \\
 \text{Convolucin fft Astropy}_{sum} &= 100,0 \\
 \text{Convolucin fft Aurora}_{sum} &= 100,99
 \end{aligned}$$

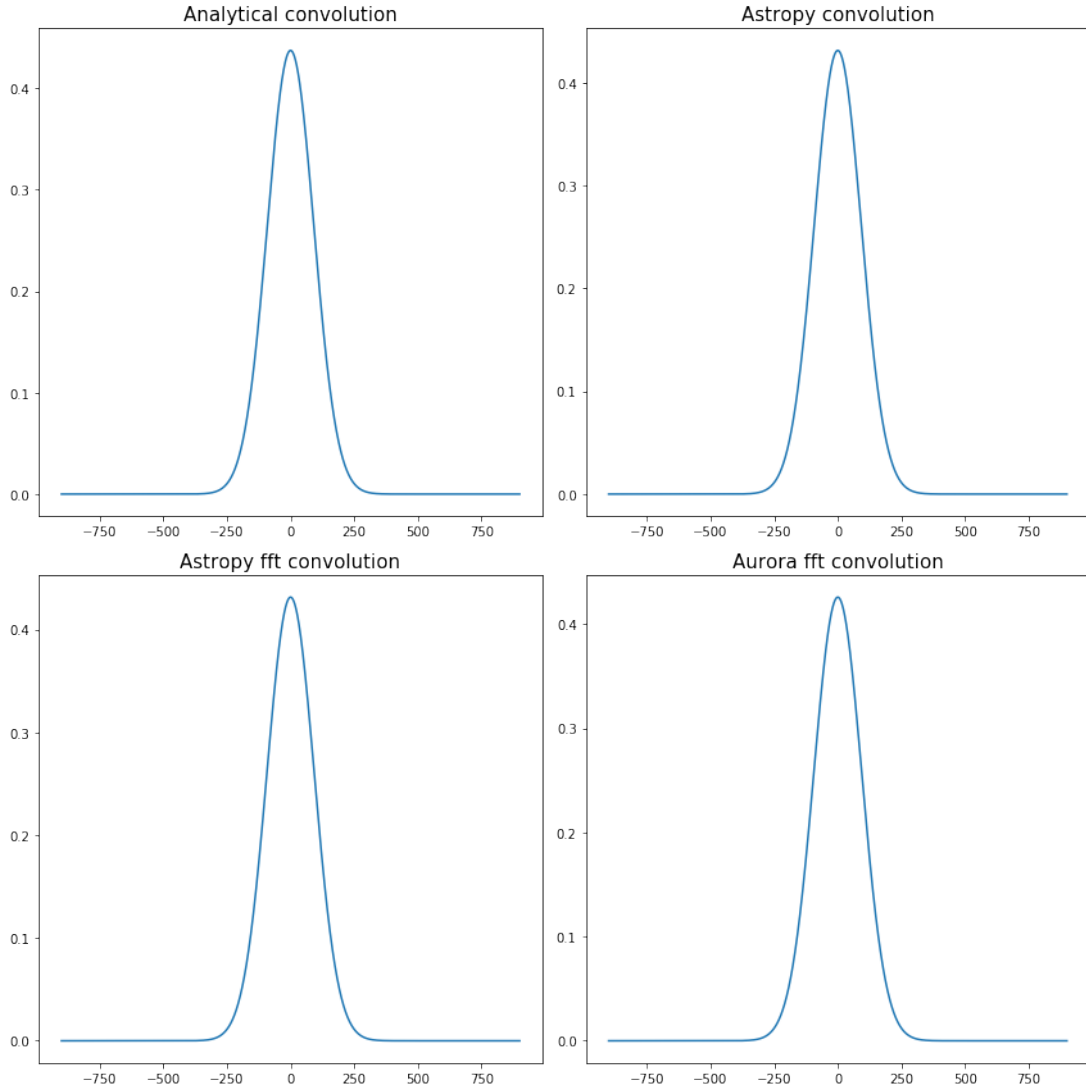


Figura 3: Convolución analítica y resultados de las convoluciones numéricas.

Para comprar los resultados, se resto la señal analítica con los resultados de las convoluciones y se presentan en la siguiente figura:

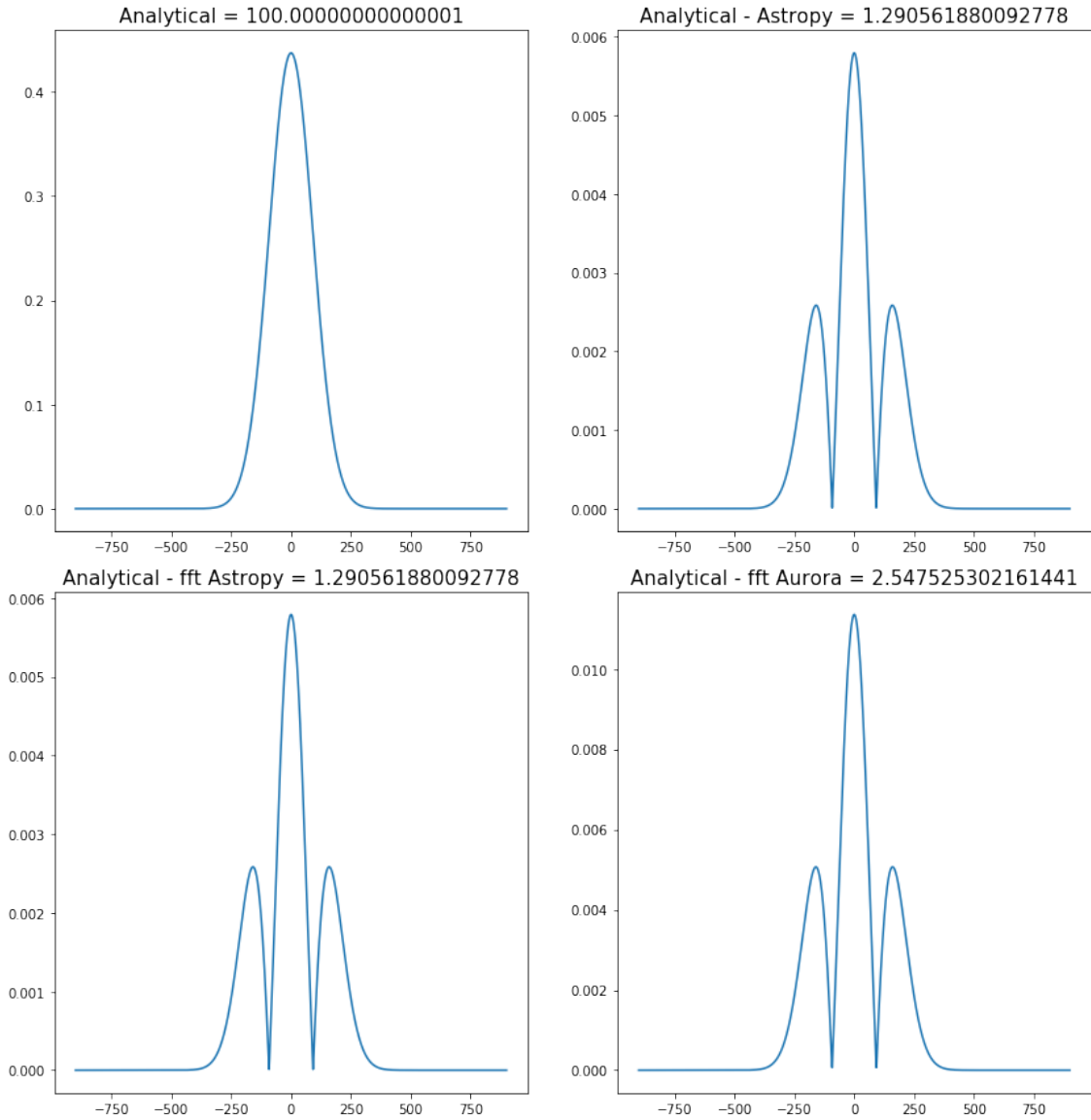


Figura 4: Convolución analítica y resultados de la diferencia entre la convolución analítica y las convoluciones numéricas. En la parte superior de cada figura se presenta la sumatoria de los valores de cada señal resultante.

La suma del “flujo” o de los valores de la señal resultante luego de aplicar las convoluciones es muy similar y prácticamente el mismo con respecto al resultado analítico. Por otro lado el resultado de la diferencia de las señales, presenta un comportamiento no muy adecuado, sin embargo el valor más grande de las diferencias es alrededor de tres ordenes de magnitud menor que el de la convolución analítica (para apreciar mejor este resultado ver la figura 5) por lo mismo el rango de valores de las diferencias es considerablemente menor y despreciable comparado con el resultado analítico.

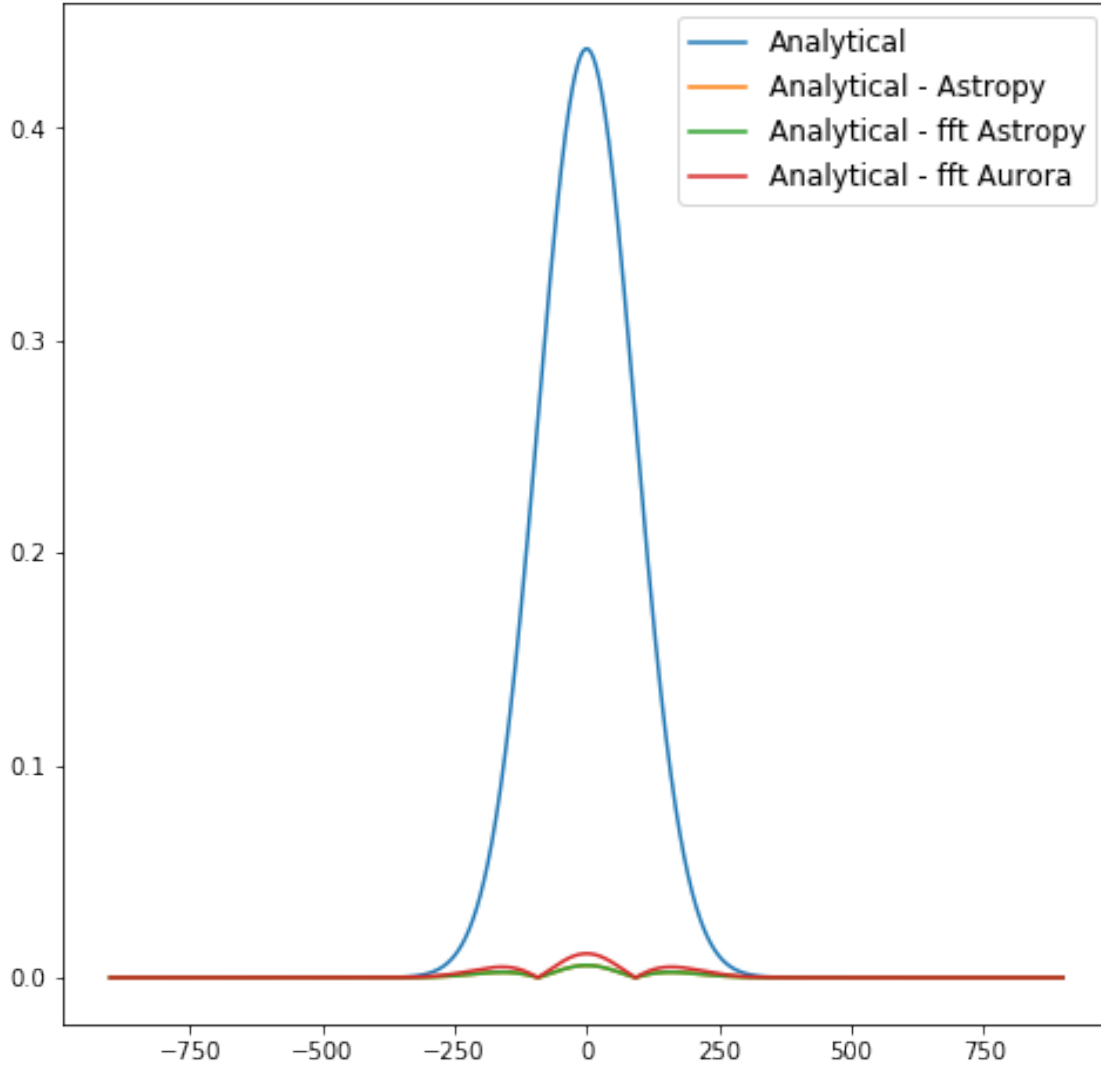


Figura 5: Convolución analítica y resultados de la diferencia entre la convolución analítica y las convoluciones numéricas.

## 2.2. Métodos para la convolución espacial

Igual que en la sección anterior una que crea el kernel de la **PSF** ([create\\_psf](#)) y 3 que realizan la convolución numérica ([spacial\\_astropy\\_convolution](#), [fft\\_spacial\\_astropy\\_convolution](#) y [fft\\_spacial\\_aurora\\_convolution](#)). Las condiciones para realizar las pruebas de esta sección son iguales que la sección anterior pero extendido a dos dimensiones (x, y), es decir la señal de entrada es una gaussiana simétrica ( $\sigma_x = \sigma_y$ ) de dos dimensiones, igual que los kernel que se van a emplear, y además la relación de los sigamas también es de 6 : 1 ( $\sigma_{signal} = 6\sigma_{kernel}$ ), también se mantiene que el tamaño de los arreglos de la señal de entrada, el kernel analítico y el kernel creado con la librería **Astropy** serán de  $20\sigma$ , en cada dimensión.

Igual que en el anterior apartado, se pondrán a prueba 3 métodos de convolución numérica, pero ahora aplicado sobre dimensiones. Por lo que los kernel que se aplicaron siguen con la misma mecánica, el analítico solo es representativo, ya que la convolución es directa (tal como se presento en la ecuación 1 pero extendido a dos dimensiones), el segundo fue creado con la función [create\\_psf](#) (Astropy Kernel) y el tercero (Aurora Kernel) es de base el mismo anterior, pero extendido con ceros a la dimensión de la señal de entrada. Los resultados de estos arreglos se presentan a continuación:

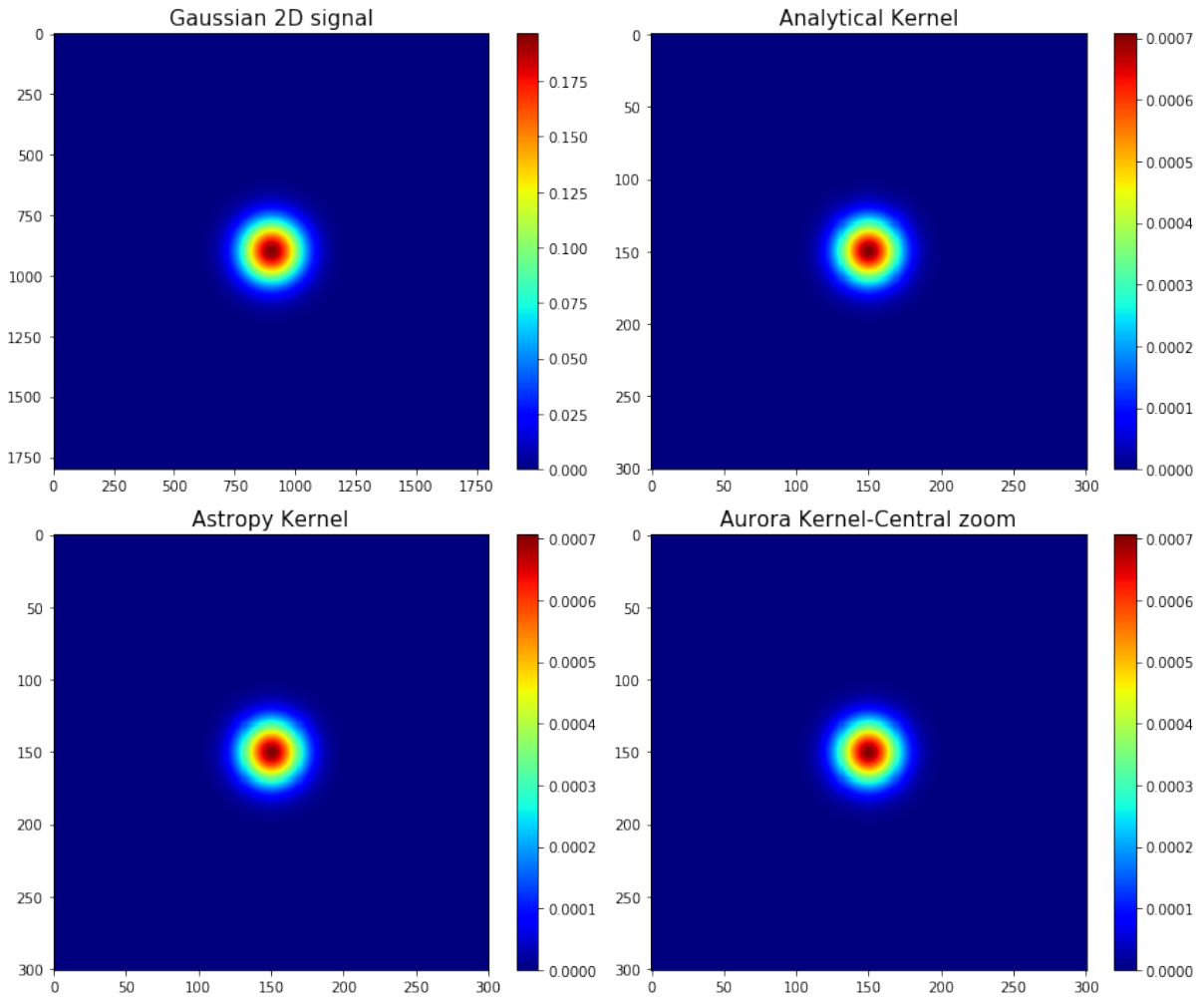


Figura 6: Señal de entrada y los tres kernel

Las convoluciones se realizaron de la siguiente manera: los métodos `spacial_astropy_convolution` y `fft_spacial_astropy_convolution` empearon el Astropy Kernel, mientras que la función `fft_spacial_aurora_convolution` se aplicó con Aurora Kernel, donde todos los kernel estaban normalizados para no alterar el “flujo” total de la señal de entrada. Los resultados fueron los siguientes:

$$Signal\ Gaussian_{sum} = 9999,99$$

$$Convolucin\ Analtica_{sum} = 9999,00$$

$$Convolucin\ Astropy_{sum} = 9999,99$$

$$Convolucin\ fft\ Astropy_{sum} = 9999,99$$

$$Convolucin\ fft\ Aurora_{sum} = 10000,99$$



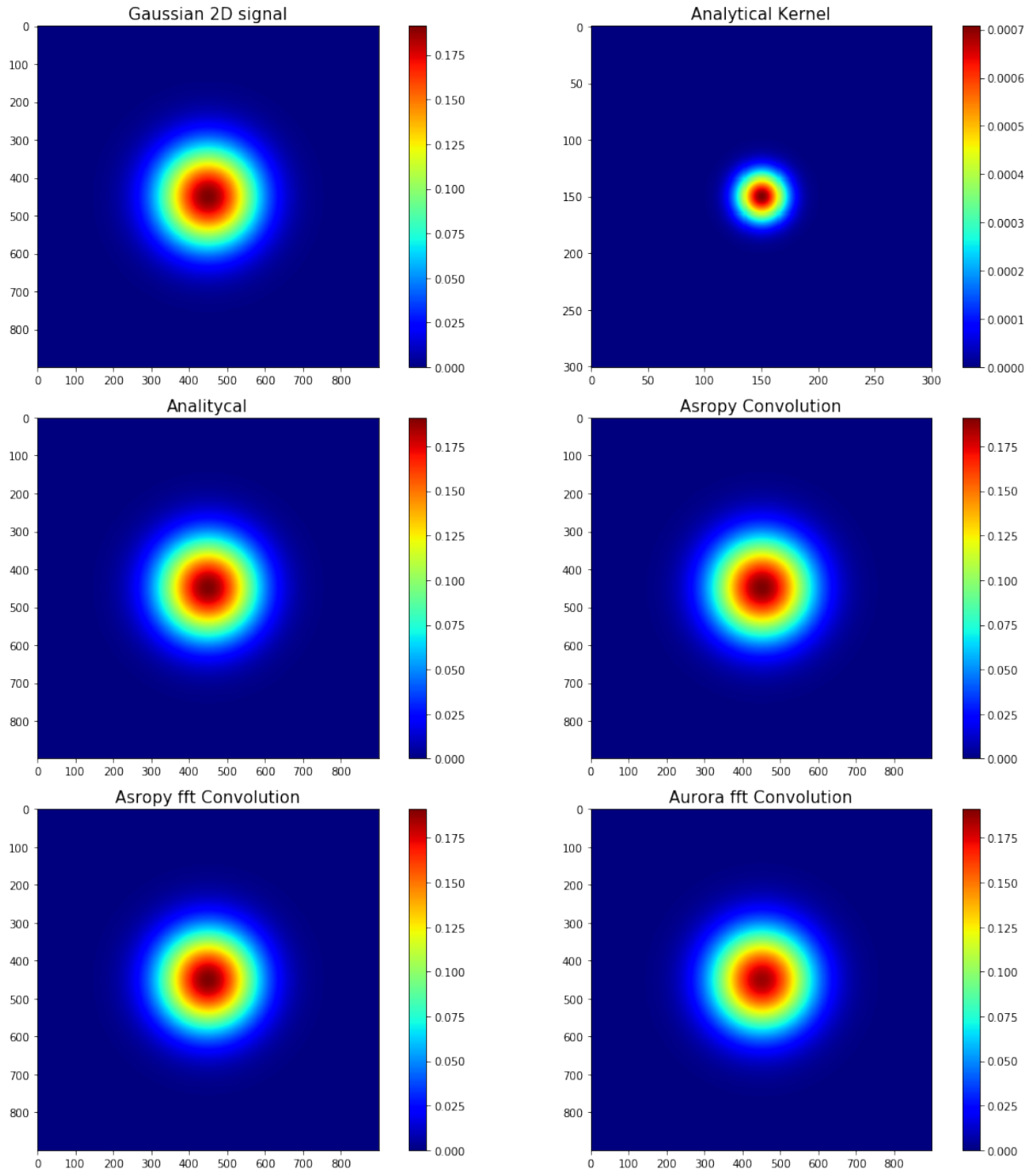


Figura 7: Señal gaussiana de entrada, kernel analítico y resultados de las convoluciones.

Los resultados de la suma de los valores de las convoluciones o el “flujo” total de la señal mantiene el valor original de la señal de entrada, tal como se esperaba, ya que los kernel empleados estaban normalizados. Por otro lado las gráficas de las convoluciones presentadas en la figura 7 son muy similares a simple observación. Para comparar con mayor rigurosidad se restaron los resultados numéricos con la solución analítica:

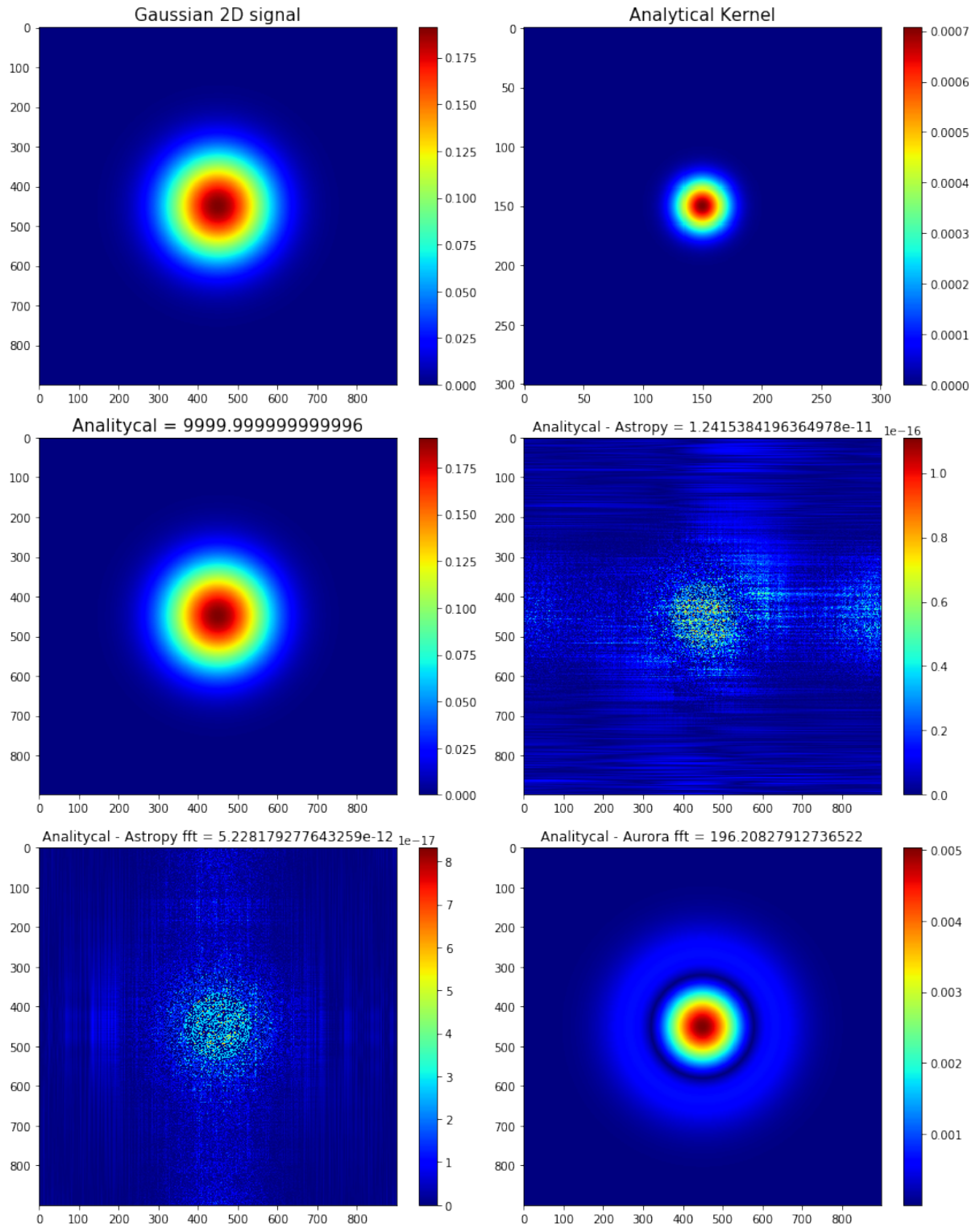


Figura 8: Resta de los resultados de las convoluciones con la convolución analítica. En la zona superior se muestra el “flujo” resultante luego de hacer la diferencia entre las señales.

Los resultados de las diferencias presentó estructuras gaussianas donde el pico central es el mayor valor donde divergen las señales, para el caso de la convolución Aurora fft. Dicho valor es dos ordenes de

magnitud menor que el valor máximo presentado en la señal analítica. Además, la suma del “flujo” total de la señal resultante presenta un error  $\approx 2$ . En el caso de los resultados con la librería **Astropy** el residuo son valores insignificantes por lo que en un resultado practico es basicamente la misma señal analítica. Para comparar mejor gráficamente el resultado se volvió a graficar la diferencia entre las señales pero manteniendo la escala en la barra de colores de la convolución analítica.

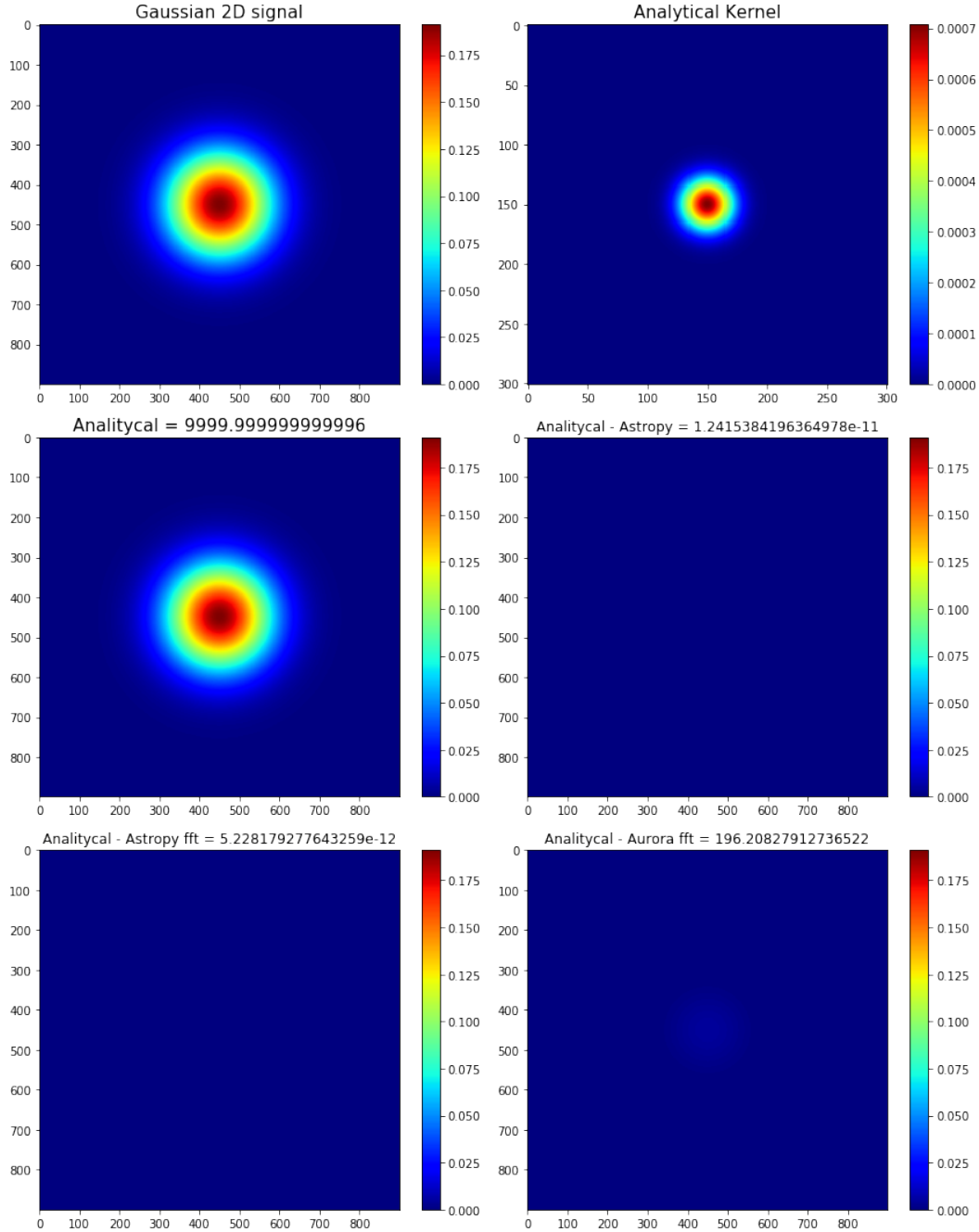


Figura 9: Resta de los resultados de las convoluciones con la convolución analítica. En la zona superior se muestra el “flujo” resultante luego de hacer la diferencia entre las señales. En esta ocasión se mantiene el rango de la barra de colores de la convolución analítica.

En la gráfica anterior se puede apreciar la estructura gaussianas discutidas para el caso de la convolución

Aurora fft, sin embargo es muy tenue y próxima a cero, en los otros dos casos no se aprecia nada, por lo tanto el resultado es muy positivo. Finalmente se seleccionaron los resultados de los píxeles centrales en el eje y, y se graficaron en un perfil de una dimensión.

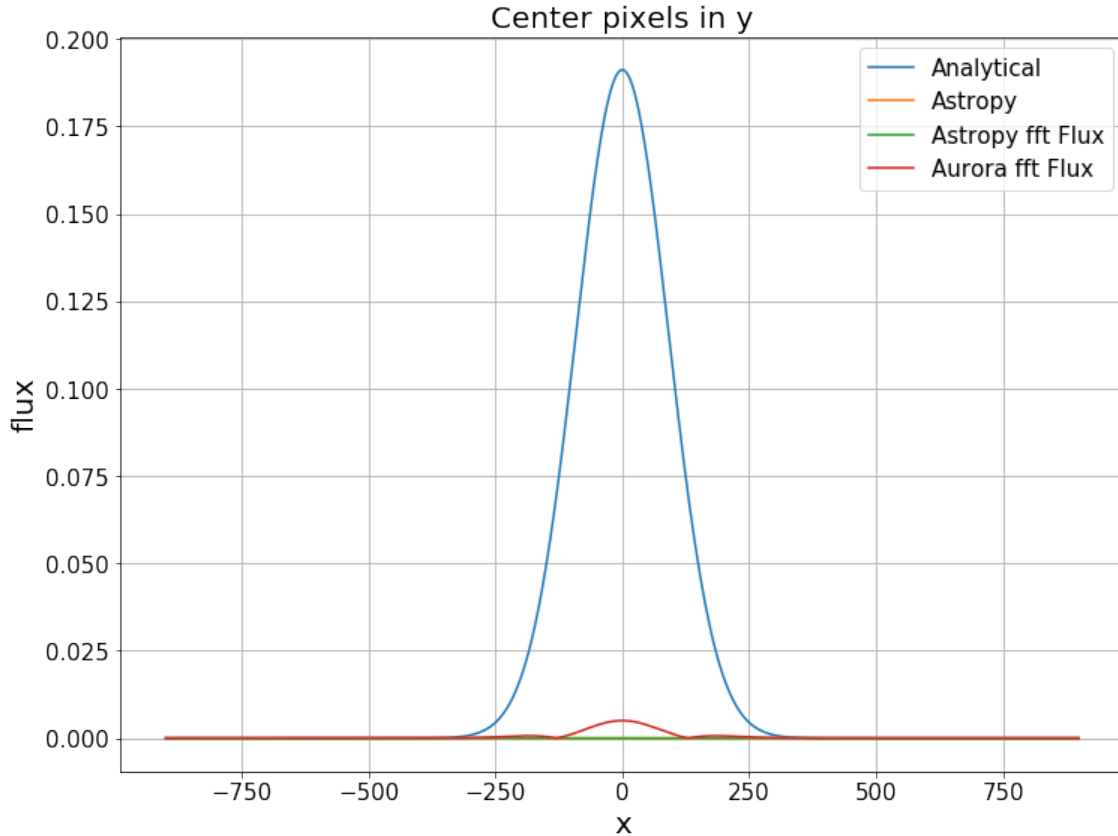


Figura 10: Resta de los resultados de las convoluciones con la convolución analítica para los píxeles centrales en y.

### 3. Modulo `array_operations.py`

En este modulo se encuentran dos funciones que permiten crear nuevos cubos de datos a partir de uno ya procesado y darles diferentes configuraciones dependiendo del producto final deseado. La función `bin_array` en principio puede trabajar con arreglos de dos y tres dimensiones, no necesariamente cubos de datos o mapas cinemáticos, y permite crear nuevos arreglos donde se agrupan, dependiendo los parámetros de entrada, los elementos en una dirección. Por otro lado la función `cube_resampling` interpola un nuevo cubo de datos sobre las tres dimensiones a partir de un cubo procesado, dando como parámetros de entrada las nuevas configuraciones para hacer una transformación a las nuevas posiciones e interpolar los valores de flujo.

#### 3.1. Función `bin_array`

Para asegurar el correcto funcionamiento de este método se busca la conservación de la suma total de los valores una vez se aplique a un cubo de datos y a un mapa de intensidad. Para lo anterior se cargo un cubo de datos previamente procesado por **Aurora** y se aplico la función con el fin que hiciera una agrupación de a dos elementos a lo largo de las dimensiones espaciales (x, y). Al comprar los resultados se obtuvo:

$$Flujo\ original = 7,024 \times 10^{-13} [ERG.S^{-1}.cm^{-2}.KM^{-1}.S]$$

$$Flujo\ nuevo = 7,024 \times 10^{-13} [ERG.S^{-1}.cm^{-2}.KM^{-1}.S]$$

$$Error = 1,438 \times 10^{-14} \%$$

Los resultados son muy similares, y el porcentaje de error es completamente despreciable. Posteriormente, se procedió a realizar el mismo proceso pero sobre un arreglo de dos dimensiones, por lo tanto se creo el mapa de intensidad a partir del cubo original y se aplico la función, los resultados fueron los siguientes:

$$Flujo\ del\ mapa\ original = 1,405 \times 10^{-11} [ERG.S^{-1}.cm^{-2}]$$

$$Flujo\ del\ nuevo\ mapa = 1,405 \times 10^{-11} [ERG.S^{-1}.cm^{-2}]$$

$$Error = 1,150 \times 10^{-14} \%$$

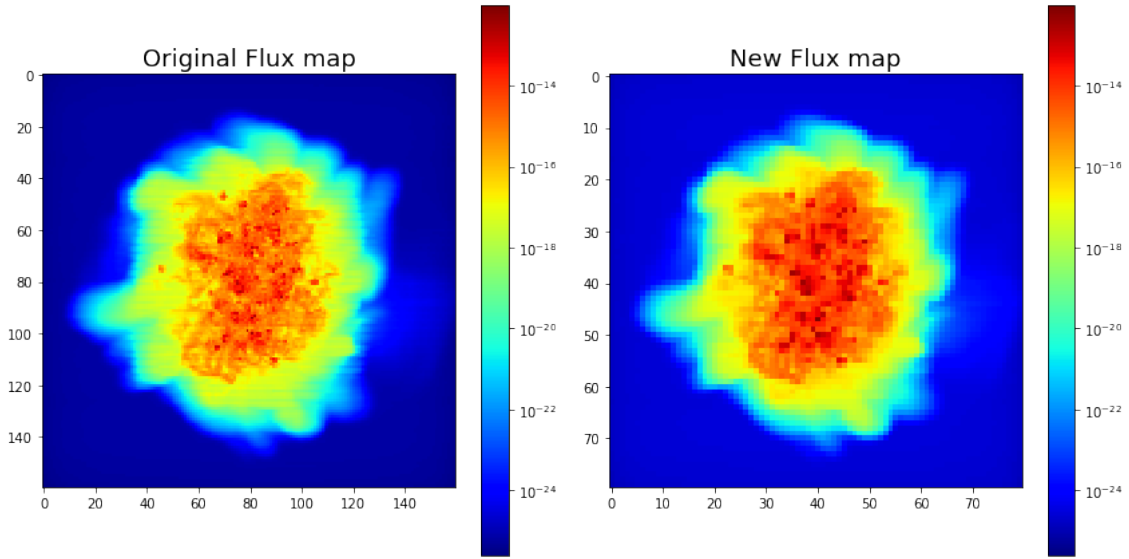


Figura 11: Derecha: Mapa de flujo original. Izquierda: Mapa de flujo luego de agrupar de 2 elementos en las dimensiones espaciales (x, y).

Igual que los resultados de la primera parte de la prueba, son idóneos y el porcentaje de error es despreciable, por lo tanto se asegura el correcto funcionamiento de la función.

### 3.2. Función `cube_resampling`

El funcionamiento del presente método depende de la interpolación y la interpretación de la configuración del nuevo cubo de datos, ya que se debe asegurar que el flujo total de del cubo procesado se mantenga en el resultado final, no el de los valores almacenados, ya que estos realmente corresponden al flujo por el tamaño de los canales de cada configuración. Por tal razón al momento de comprar los resultados, se debe multiplicar el cubo por el tamaño de los canales. Dado que en la prueba anterior ya se cargo el cubo procesado, se procedió a crear una instancia de la misma clase que el cubo cargado y se le atribuyeron las nuevas dimensiones manteniendo las relaciones originales para asegurar que

se abarca el mismo tamaño espacial y espectral para poder asegurar la conservación del flujo total. Para ello las nuevas dimensiones se almacenaron como múltiplos semi-enteros de las dimensiones del cubo cargado, específicamente, se definieron dos factores,  $n$  para las los ejes espaciales y  $m$  para el eje espectral ( $n = 1,5$  y  $m = 1,5$ ). En caso espacial se multiplico el tamaño del píxel por el factor  $n$  y se dividió en número de píxeles por el mismo, recordando que la observación se hizo sobre un área cuadrada por lo que afecta de la misma manera en ambas direcciones ( $x, y$ ), de tal manera se asegura la relación. Para el caso espectral se hizo lo mismo, se multiplico el tamaño de los canales por el factor  $m$  y se dividió el numero total de canales por el mismo. Llegados a este punto es importante hacer una pequeña nota:

**Nota:** Si las nuevas configuraciones son múltiplos enteros de las dimensiones originales se recomienda emplear la función `bin_array` anteriormente abordada, ya que la interpolación numérica puede almacenar algunos errores, no es un resultado perfectamente ceñido al cubo original.

Una vez los atributos de la nueva instancia fueron almacenados se procede a aplicar el método en cuestión, aclarando que el resultado es aplicado como un atributo al objeto del nuevo cubo. Como ya se menciono anteriormente, se va a comprar el flujo total por lo tanto se multiplica el cubo por el ancho de los canales en cada configuración, los resultados fueron los siguientes:

$$Flujo\ original = 1,405 \times 10^{-11} [ERG.S^{-1}.cm^{-2}]$$

$$Flujo\ nuevo = 1,423 \times 10^{-11} [ERG.S^{-1}.cm^{-2}]$$

$$Error = 1,431 \%$$

El resultado de la interpolación modifiko levemente el flujo total original con un error de menos del 2 %. Si se compara con los resultados anteriores, claramente hay un error más alto, pero se debe tener en cuenta que este proceso es una interpolación y por lo tanto se pueden albergar pequeños errores numéricos. Para ver gráficamente el resultado, se procedió a obtener el mapa del flujo de las dos configuraciones y el resultado es el siguiente:

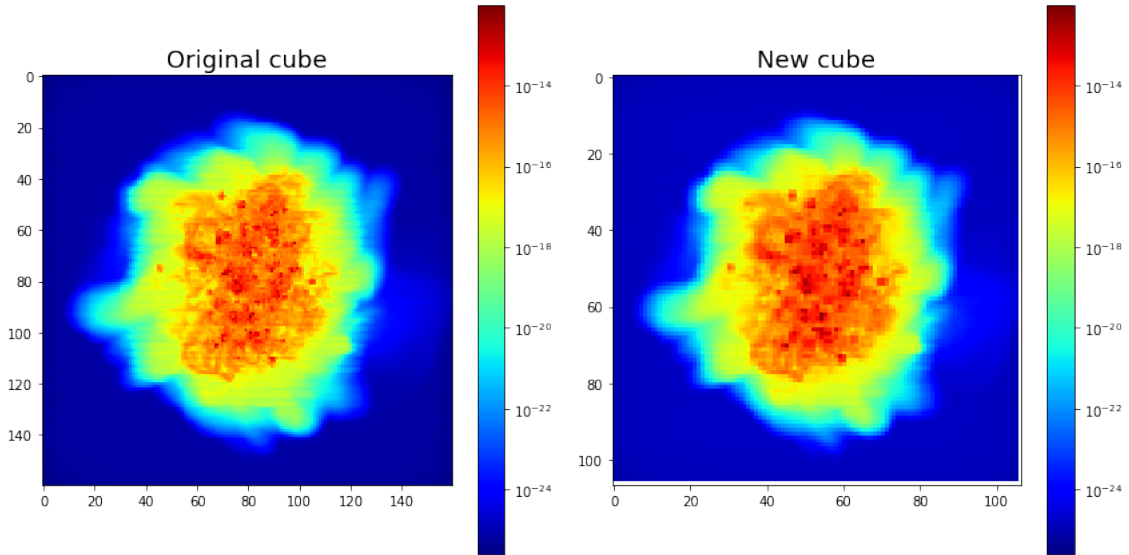
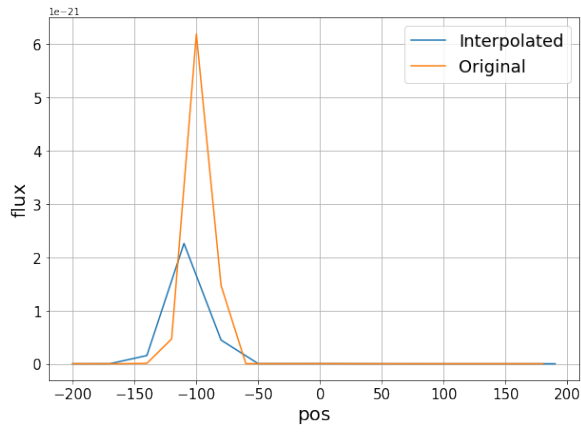
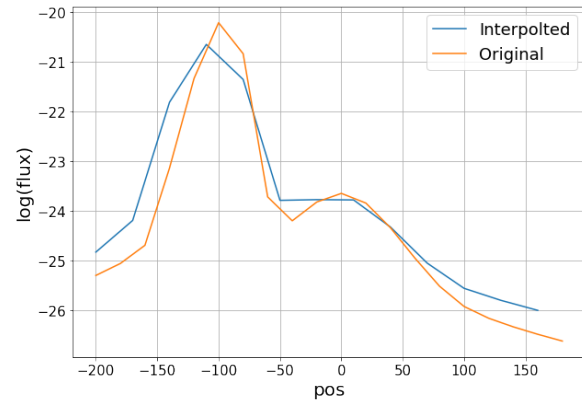


Figura 12: Derecha: Mapa de flujo original. Izquierda: Mapa de flujo del cubo interpolado.

Finalmente para observar el resultado de la interpolación sobre la dirección espectral, se ubico el píxel central de cada una de las configuraciones, ya que las dimensiones son diferentes, y se obtuvo el espectro en las siguientes gráficas:



(a) Espectro del píxel central



(b) Logaritmo del espectro del píxel central

Figura 13: Espectro del píxel central de las dos configuraciones

# Test\_analytical\_spectral\_convolutions

July 29, 2020

## 0.1 Analytical spectral convolution test

```
[1]: # Necessary libraries are imported
import sys
import logging
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
from astropy import units as unit
from matplotlib.colors import LogNorm
from numpy.fft import fftshift, rfft2, irfft2
sys.path.append("/home/rolando9807/Documentos/tesis/Aurora/")

import astropy.convolution
from bisect import bisect_left

from aurora import aurora as au
from aurora import set_output as sto
from aurora import constants as ct
from aurora import convolutions as cv
from aurora import emitters as emit
```

## 0.2 Functions to calculate spectral resolution with the resolving power ( $R$ )

### 0.3

$$R = \frac{\lambda_{H\alpha}}{\Delta\lambda} = \frac{c}{\Delta v}$$

```
[2]: # Halpha wavelength redshifted function
def Halpha_shift(z):
    return (1 + z) * ct.Halpha0

# Velocity dispersion function for a temperature
def sigma_T(T):
    """Calculate the velocity dispersion for a temperature (in K), assuming
    that in the hydrogen helium mixture all the hydrogen is ionized
```



```

"""
mu = 0.63 # Average molecular weight of the hydrogen ionized
return np.sqrt(ct.k_B * T * unit.K/(mu * ct.m_p)).to("cm s**-1")

# Spectral resolution function
def spectral_res(lsf_fwhm):
    return ct.c.to("km s-1") / lsf_fwhm.to("km s-1")

# Velocity dispersion associated with spectral resolution
def lsf_sigma(lsf_fwhm):
    return lsf_fwhm / ct.fwhm_sigma

def vel_to_wavelength(lambda_obs, vel):
    wavelength = lambda_obs * vel.to("km s-1").value / (
        ct.c.to("km s-1").value)
    return wavelength

def wavelength_to_vel(lambda_obs, wavelength):
    vel = ct.c.to("km s-1") * wavelength.to("angstrom").value / (
        lambda_obs.to("angstrom").value)
    return vel

```

```

[3]: # Wavelength of ionized hydrogen for z = 1
lambda_Halpha = Halpha_shift(1)

# Sigma for interstellar gas with t = 10,000 K
sigma_10000 = vel_to_wavelength(lambda_Halpha, sigma_T(10000))

# Range to resolving power
R = np.arange(3000, 5001, 500)

# lsf parameters
lsf_fwhm = lambda_Halpha / R
sigma_lsf = lsf_sigma(lsf_fwhm)

# Sigma ratio
sigma_lsf / sigma_10000

```

```

[3]: [3.7073989, 3.1777705, 2.7805492, 2.4715993, 2.2244394]

```

0.4 A sigma ratio of 3: 1 is sought. From the previous result, for this experiment,  $R > \sim 3500$  can be taken:

```
spectrom.spectral_res = 5000
```

# 1 Flux conservation test after analytical convolution

```
[82]: # The parameter file is loaded to process the cube with Aurora
```

```
ConfigFile = 'params_20_0a.config'
ConfigFile
au.__setup_logging()
au.__aurora_version()
```

```

  ---
 / _ |__ -----
/ _ / // / _ / _ \ / _ / _ \
/_ / | \ _ _ / _ \ _ _ / _ \ _ _ /
///// Version 2.1
```

```
[83]: geom, run, spectrom = au.config.get_allinput(ConfigFile)
```

```
[84]: spectrom.vel_channels
```

```
[84]: [-500, -490, -480, -470, -460, -450, -440, -430, -420, -410, -400, -
390, -380, -370, -360, -350, -340, -330, -320, -310, -300, -290, -280, -
270, -260, -250, -240, -230, -220, -210, -200, -190, -180, -170, -160, -
150, -140, -130, -120, -110, -100, -90, -80, -70, -60, -50, -40, -30, -20, -
10, 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210]
```

```
[85]: data = au.snap.read_snap(run.input_file)
```

```
[86]: data_gas = au.snap.set_snapshots_ready(geom, run, data)[0]
```

```
[87]: lim = spectrom.fieldofview.to("kpc").value/2.
data_gas = au.snap.filter_array(data_gas, ["x", "y"], 2*[-lim], 2*[lim], 2*["kpc"])
au.snap.set_hsml_limits(run, data_gas)
spectrom.oversample()
```

```
[88]: # The flux of the first 10,000 particles is processed directly with the
      ↪ emitters module
em = emit.Emitters(data_gas[0:10000], spectrom.redshift_ref)
em.get_state()
em.get_luminosity(spectrom.lum_dens_rel)
em.density_cut(spectrom.density_threshold, spectrom.equivalent_luminosity)
em.get_vel_dispersion()
cube_side, n_ch = spectrom.cube_dims()
x, y, index = spectrom.position_in_pixels(em.x, em.y)

scale = np.digitize(em.smooth.to("kpc"), 1.1 * run.fft_hsml_limits.to("kpc"))

line_center, line_sigma, line_flux = em.get_vect_lines(n_ch)
```

```

channel_center, channel_width = em.get_vect_channels(spectrom.vel_channels,
↳spectrom.velocity_sampl, n_ch)

# Flux in the channels without analytical convolution
flux_in_channels = em.int_gaussian_with_units(channel_center, channel_width,
                                             line_center, line_sigma) ## line_flux
#flux_in_channels = flux_in_channels.to("erg s^-1").value / spectrom.
↳velocity_sampl.to("km s^-1").value / geom.dl.to("cm").value**2

```

Nothing to cut

```

[89]: # Analytical spectral convolution
spectrom.spectral_res = 5000
if(spectrom.spectral_res > 0):
    lsf_fwhm = ct.c/spectrom.spectral_res
    lsf_sigma = lsf_fwhm / ct.fwhm_sigma
    line_sigma = np.sqrt(line_sigma**2 + lsf_sigma**2)
flux_in_channels_Aconvolution = em.int_gaussian_with_units(channel_center,
↳channel_width,
                                                         line_center, line_sigma) ## line_flux
#flux_in_channels_Aconvolution = flux_in_channels_Aconvolution.to("erg s^-1").
↳value / spectrom.velocity_sampl.to("km s^-1").value / geom.dl.to("cm").
↳value**2

```

## 1.1 Spectral numerical convolution

```
[90]: lsf = cv.create_lsf(spectrom)
```

```
[91]: index_p = 0
```

```
[92]: flux_in_channels.shape
```

```
[92]: (10000, 100)
```

```

[93]: flux_channel_cube = np.reshape(flux_in_channels[index_p].copy(), (spectrom.
↳spectral_dim, 1, 1))
flux_channel_astropy = cv.spectral_convolution_astropy(flux_channel_cube.
↳copy(), lsf)
flux_channel_astropy_fft = cv.
↳spectral_convolution_astropy_fft(flux_channel_cube.copy(), lsf)
flux_channel_aurora_fft = cv.spectral_convolution_astropy_fft(flux_channel_cube.
↳copy(), lsf)

```

```

[94]: # Particle index
print('Total flux[1500] = {}'.format(flux_in_channels[index_p].sum()))

```

```

print('Total convoluted flux[1500] = {}'.format(flux_in_channels_Aconvolution[index_p].sum()))
print('Total convoluted astropy flux[1500] = {}'.format(flux_channel_astropy.sum()))
print('Total convoluted astropy fft flux[1500] = {}'.format(flux_channel_astropy_fft.sum()))
print('Total convoluted aurora fft flux[1500] = {}'.format(flux_channel_aurora_fft.sum()))

dif = flux_in_channels[index_p].sum() - flux_in_channels_Aconvolution[index_p].sum()
print('Difference = {}'.format(dif))

error = np.abs(flux_in_channels[index_p].sum() - flux_in_channels_Aconvolution[index_p].sum()) / flux_in_channels[index_p].sum() * 100
print('Error = {}'.format(error))

```

```

Total flux[1500] = 0.9998003708262844
Total convoluted flux[1500] = 0.9997422173713648
Total convoluted astropy flux[1500] = 0.9997025203955773
Total convoluted astropy fft flux[1500] = 0.9997025203955768
Total convoluted aurora fft flux[1500] = 0.9997025203955768
Difference = 5.8153454919551706e-05
Error = 0.0058165066363689%

```

```

[98]: fig = plt.figure(figsize=(15, 12))
plt.xlabel('Vel Channels', fontsize = 20)
plt.ylabel('flux', fontsize = 20)

plt.plot(spectrom.vel_channels.value,
         flux_in_channels[index_p], '-', label='Flux')

plt.plot(spectrom.vel_channels.value,
         flux_channel_astropy[:,0,0], '-', label='Astropy Flux')

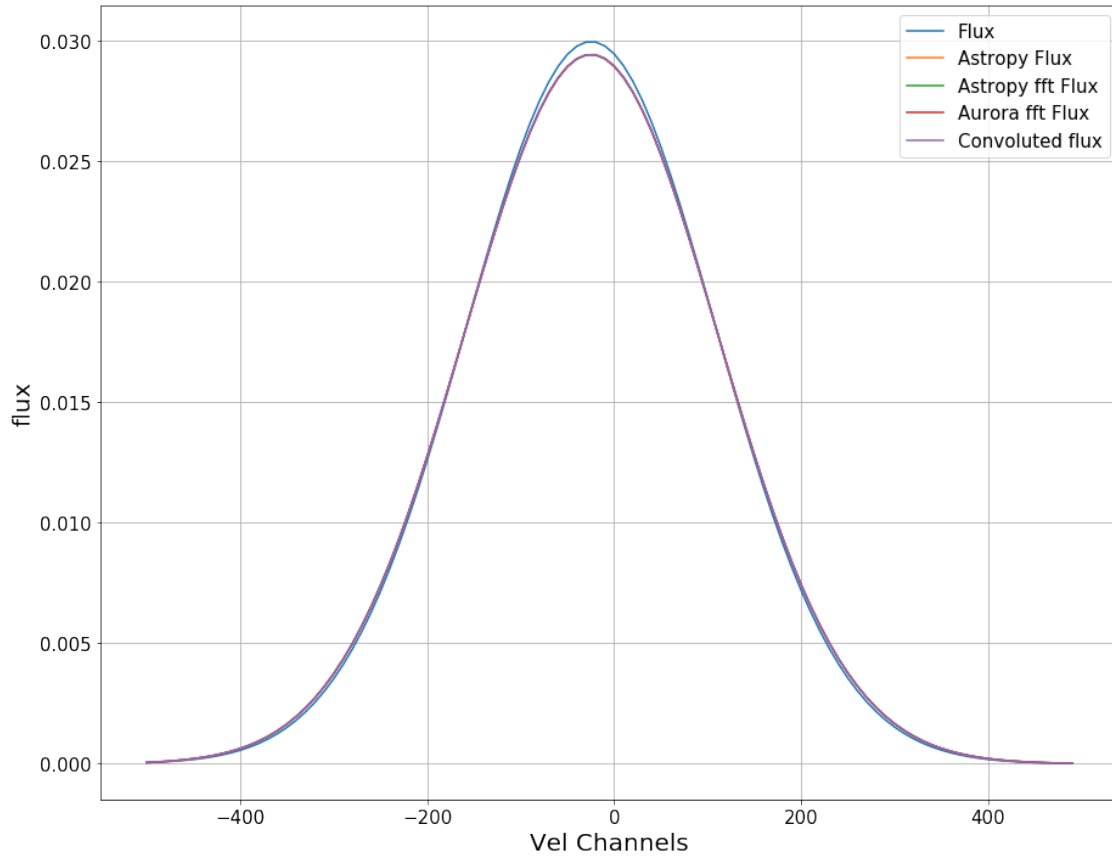
plt.plot(spectrom.vel_channels.value,
         flux_channel_astropy_fft[:,0,0], '-', label='Astropy fft Flux')

plt.plot(spectrom.vel_channels.value,
         flux_channel_aurora_fft[:,0,0], '-', label='Aurora fft Flux')

plt.plot(spectrom.vel_channels.value,
         flux_in_channels_Aconvolution[index_p], '-',
         label='Convoluted flux')

```

```
plt.legend(fontsize = 15, loc=0)
plt.xticks(size = 15)
plt.yticks(size = 15)
plt.grid()
plt.show()
```



```
[100]: fig = plt.figure(figsize=(15, 12))
plt.xlabel('Vel Channels', fontsize = 20)
plt.ylabel('log(flux)', fontsize = 20)

plt.plot(spectrom.vel_channels.value,
         np.log10(flux_in_channels[index_p]), '-', label='Flux')

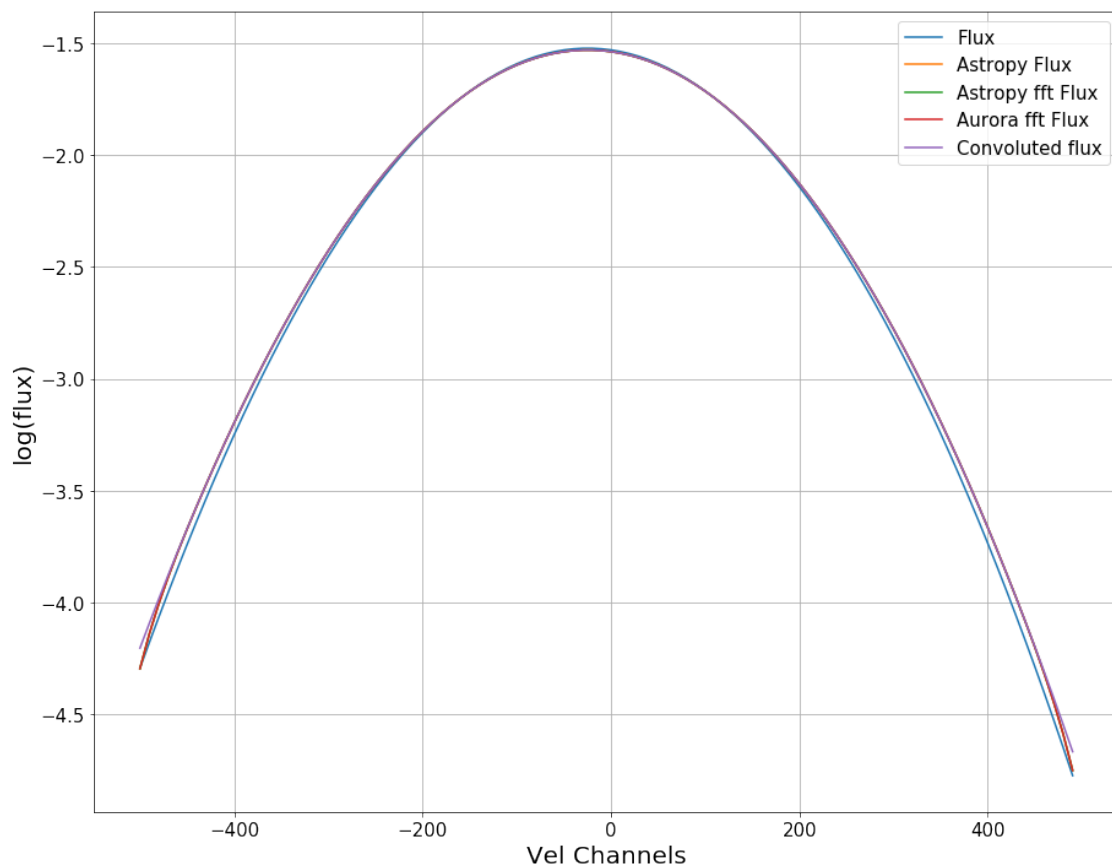
plt.plot(spectrom.vel_channels.value,
         np.log10(flux_channel_astropy[:,0,0]), '-', label='Astropy Flux')

plt.plot(spectrom.vel_channels.value,
         np.log10(flux_channel_astropy_fft[:,0,0]), '-', label='Astropy fft_
         ↪Flux')
```

```
plt.plot(spectrom.vel_channels.value,
         np.log10(flux_channel_aurora_fft[:,0,0]), '-', label='Aurora fft Flux')

plt.plot(spectrom.vel_channels.value,
         np.log10(flux_in_channels_Aconvolution[index_p]), '-',
         label='Convolved flux')

plt.legend(fontsize = 15, loc=0)
plt.xticks(size = 15)
plt.yticks(size = 15)
plt.grid()
plt.show()
```



[ ]:

[ ]:

[ ]:

# Test\_convolutions\_spectral

July 29, 2020

## 0.1 Spectral convolutions test

```
[1]: # Necessary libraries are imported
import sys
import scipy
import logging
import numpy as np
from scipy import fftpack
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
from numpy.fft import fftshift, rfft2, irfft2
sys.path.append("/home/rolando9807/Documentos/tesis/Aurora/")

import astropy.convolution
from bisect import bisect_left

from aurora import aurora as au
from aurora import set_output as sto
from aurora import constants as ct
from aurora import convolutions as cv
```

## 1 Spectral functions

```
[2]: # Gaussian 1D function
def gaus(x=0, mx=0, sx=0.5):
    gaus = (1 / (np.sqrt(2 * np.pi) * sx)) * np.exp(-( (x - mx)**2 / (2 *
    ↪sx**2))))
    return gaus

# lsf kernel creator function
def create_lsf(psf_fwhm, size = 20):
    psf_sigma = psf_fwhm
    psf = astropy.convolution.Gaussian1DKernel(psf_sigma, x_size = cv.
    ↪next_odd(size * psf_sigma))
    psf = np.array(psf)
```

```

    psf = psf / psf.sum()
    return psf

# Aurora fft spectral convolution function
def lsf_convolution(cube, lsf):
    fshape = fftpack.next_fast_len(cube.shape[0]+lsf.shape[0])
    center = fshape - (fshape+1) // 2

    lead_zeros = np.zeros(center - lsf.shape[0] // 2)
    trail_zeros = np.zeros(fshape - lsf.shape[0] - lead_zeros.shape[0])
    lsf = np.concatenate((lead_zeros, lsf, trail_zeros), axis=0)
    lsf = np.fft.fftshift(lsf)
    lsf = lsf.reshape(lsf.size, 1, 1)
    lsf = np.fft.rfft(lsf, axis=0)

    index = slice(center - cube.shape[0] //
                  2, center + (cube.shape[0] + 1) // 2)
    lead_zeros = np.zeros([center - cube.shape[0] // 2,
                          cube.shape[1], cube.shape[2]])
    trail_zeros = np.zeros(
        [fshape - cube.shape[0] - lead_zeros.shape[0], cube.shape[1], cube.
        ↪shape[2]])
    cube = np.concatenate((lead_zeros, cube, trail_zeros), axis=0)
    cube = np.fft.rfft(cube, axis=0)

    cube = cube * lsf
    cube = np.fft.irfft(cube, fshape, axis=0)
    cube = cube[index, :, :]

    return cube

# Astropy spectral convolution function
def lsf_astropy_convolution(cube, lsf):
    x, y, z = cube.shape

    for j in range(y):
        for i in range(z):
            cube[:, j, i] = astropy.convolution.convolve(cube[:,j,i],lsf)
    return cube

# Astropy fft spectral convolution function
def fft_lsf_astropy_convolution(cube, lsf):
    x, y, z = cube.shape

    for j in range(y):
        for i in range(z):

```



```

        cube[:, j, i] = astropy.convolution.convolve_fft(cube[:, j, i],
↳lsf, psf_pad = True,
                                                                    fft_pad = True,
↳allow_huge=True)
        return cube

```

## 2 Gaussian signal and lsf kernels

```

[3]: # The Gaussian signal is defined based on the kernel sigma,
# with sufficient resolution not to store numerical errors.

s_lsf = 15 # Sigma lsf
s_gaus = 6 * s_lsf # Sigma gaussian signal
n_s = 20 # Sigma numbers to solve Gaussian signal

size_z = cv.next_odd(n_s * s_gaus) # Boundaries of the gaussians
#size_z_p = cv.next_odd(int(size_z) * 4) # Points numbers to solve Gaussian
↳signal
x_1 = np.arange(-int(size_z/2), int(size_z/2)+1) # Points to solve Gaussian
↳signal

A = 1e2
z_1 = A*gaus(x_1, sx = s_gaus) # Gaussian signal

# Analytical kernel

size_lsf = cv.next_odd(n_s * s_lsf)
x_lsf = np.arange(-int(size_lsf/2), int(size_lsf/2)+1)
lsf = gaus(x_lsf, sx = s_lsf)
lsf /= lsf.sum()

```

```

[4]: # Astropy Kernel
lsf_1d = create_lsf(s_lsf, size = n_s)

# Aurota Kernel (padding of the kernel as a function of the Gaussian signal to
↳Aurora fft)
fshape = fftpack.next_fast_len(z_1.shape[0]+lsf_1d.shape[0])
fshape2 = fftpack.next_fast_len(x_1.shape[0]+x_lsf.shape[0])

center = fshape - (fshape+1) // 2

lead_zeros = np.zeros(center - lsf_1d.shape[0] // 2)

trail_zeros = np.zeros(fshape - lsf_1d.shape[0] - lead_zeros.shape[0])

```

```

x_1_n = np.linspace(x_1.min(), x_lsf.min(), len(lead_zeros))
x_1_p = np.linspace(x_lsf.max(), x_1.max(), len(trail_zeros))

lsf_1d_Au = np.concatenate((lead_zeros, lsf_1d, trail_zeros), axis=0)
x_1_Au = np.concatenate((x_1_n, x_lsf, x_1_p), axis=0)

```

### 3 Gaussian and kernels

```

[5]: fig, axs = plt.subplots(nrows = 2, ncols = 2, figsize=(12, 12))

fig.subplots_adjust(hspace=0.3)

axs[0, 0].set_title("Gaussian signal ", fontsize = 15)
axs[0, 0].plot(x_1, z_1)

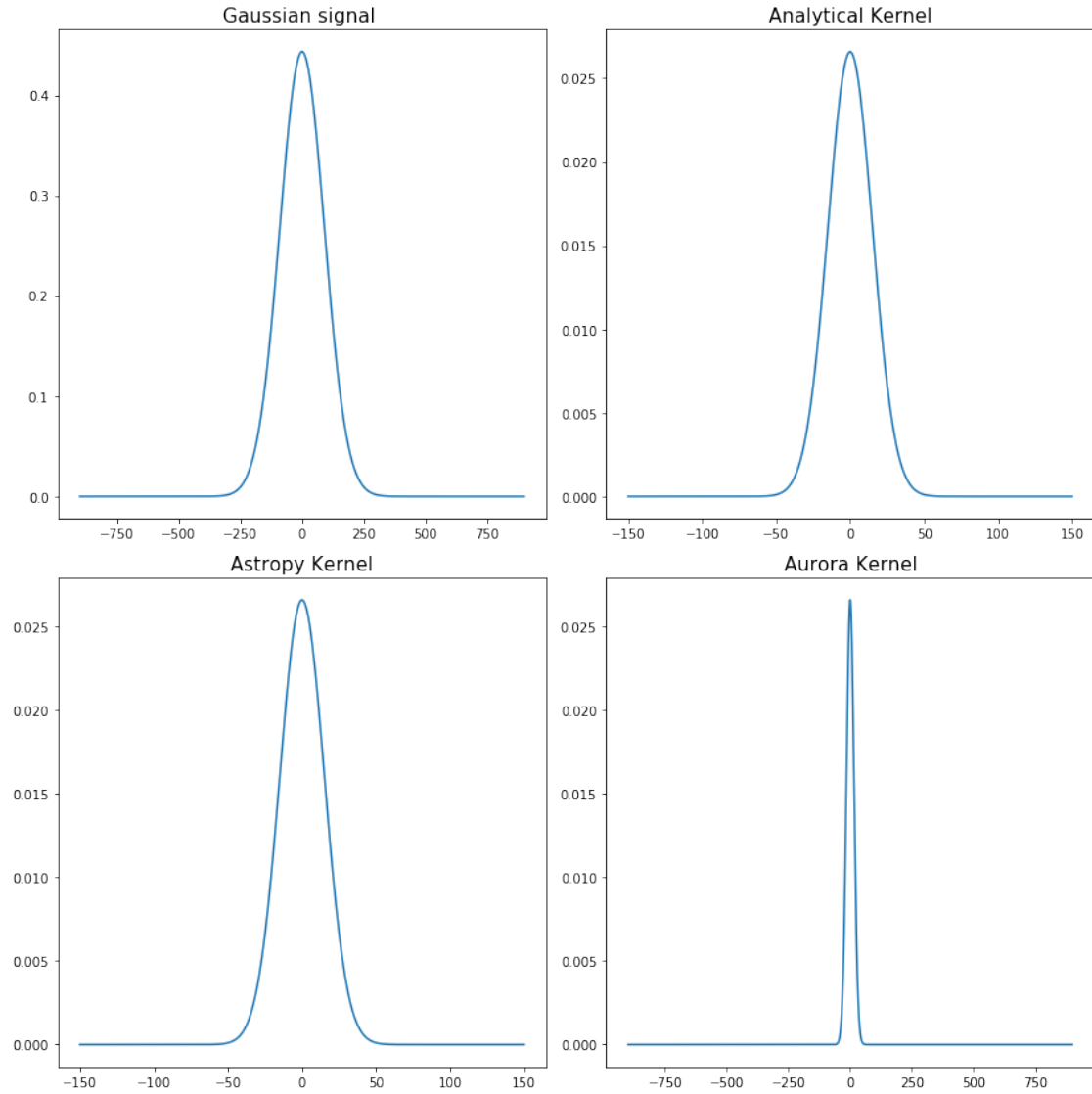
axs[0, 1].set_title("Analytical Kernel", fontsize = 15)
axs[0, 1].plot(x_lsf, lsf)

axs[1, 0].set_title("Astropy Kernel", fontsize = 15)
axs[1, 0].plot(x_lsf, lsf_1d)

axs[1, 1].set_title("Aurora Kernel", fontsize = 15)
axs[1, 1].plot(x_1_Au, lsf_1d_Au)

fig.tight_layout()
plt.show()

```



## 4 Convolutions

```
[6]: # Analytical convolution
s_c = np.sqrt(s_lsf**2 + s_gaus**2) # New sigma
z_f = A*gaus(x_1, sx = s_c) # Analytical convolution signal

# Astropy convolution
c_astropy = lsf_astropy_convolution(np.reshape(z_1, (z_1.shape[0],1,1)), lsf_1d)

# Astropy fft convolution
```

```

c_astropy_fft = fft_lsf_astropy_convolution(np.reshape(z_1,(z_1.shape[0],1,1)),  

↳lsf_1d)

# Aurora fft convolution
c_fft = lsf_convolution(np.reshape(z_1,(z_1.shape[0],1,1)), lsf_1d)

```

```

[7]: # Sum of signal flux after convolutions
print(r'$z_{sum}$=', z_1.sum())
print(r'Analitical convolution $z_f_{sum}$=', z_f.sum())
print(r'convolution $astropy_{sum}$=', c_astropy.sum())
print(r'convolution astropy $fft_{sum}$=', c_astropy_fft.sum())
print(r'convolution aurora $fft_{sum}$=', c_fft.sum())

```

```

$z_{sum}$= 100.00000000000004
Analitical convolution $z_f_{sum}$= 100.00000000000001
convolution $astropy_{sum}$= 100.00000000000004
convolution astropy $fft_{sum}$= 100.00000000000004
convolution aurora $fft_{sum}$= 100.00000000000004

```

## 5 Convolutions results

```

[8]: fig, axs = plt.subplots(nrows = 2,  ncols = 2, figsize=(12, 12))

fig.subplots_adjust(hspace=0.3)

axs[0, 0].set_title("Analytical convolution", fontsize = 15)
axs[0, 0].plot(x_1, z_f)

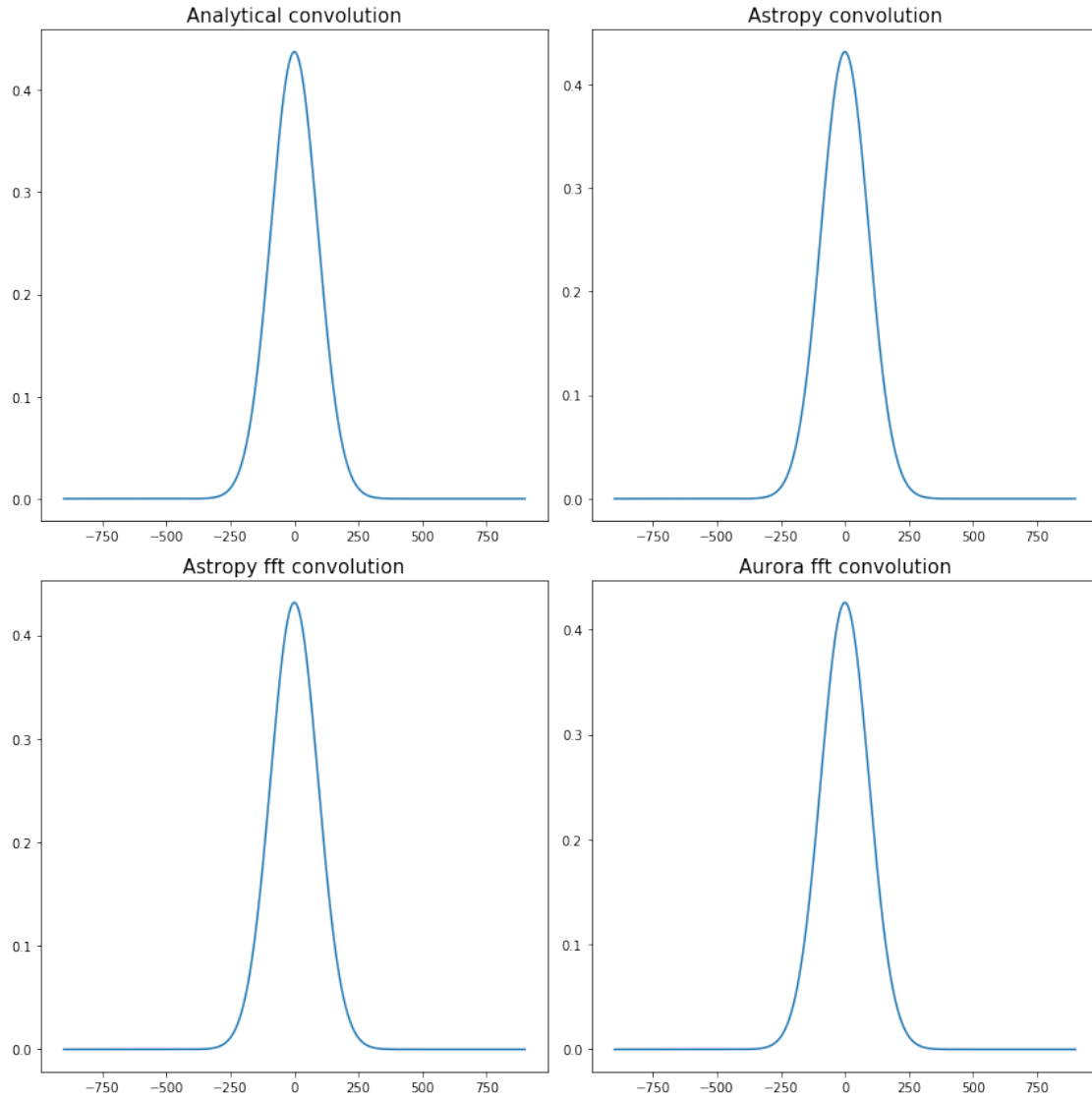
axs[0, 1].set_title("Astropy convolution", fontsize = 15)
axs[0, 1].plot(x_1,  c_astropy.reshape(len(c_astropy)))

axs[1, 0].set_title("Astropy fft convolution", fontsize = 15)
axs[1, 0].plot(x_1, c_astropy_fft.reshape(len(c_astropy)))

axs[1, 1].set_title("Aurora fft convolution", fontsize = 15)
axs[1, 1].plot(x_1, c_fft.reshape(len(c_astropy)))

fig.tight_layout()
plt.show()

```



## 6 Comparison of convolutions with respect to *Analytical* :

### 6.1 $|Analytical - convolution|$

```
[9]: fig, axs = plt.subplots(nrows = 2, ncols = 2, figsize=(12, 12))

fig.subplots_adjust(hspace=0.3)

axs[0, 0].set_title("Analytical = {}".format(z_f.sum()), fontsize = 15)
axs[0, 0].plot(x_1, z_f)
```

```

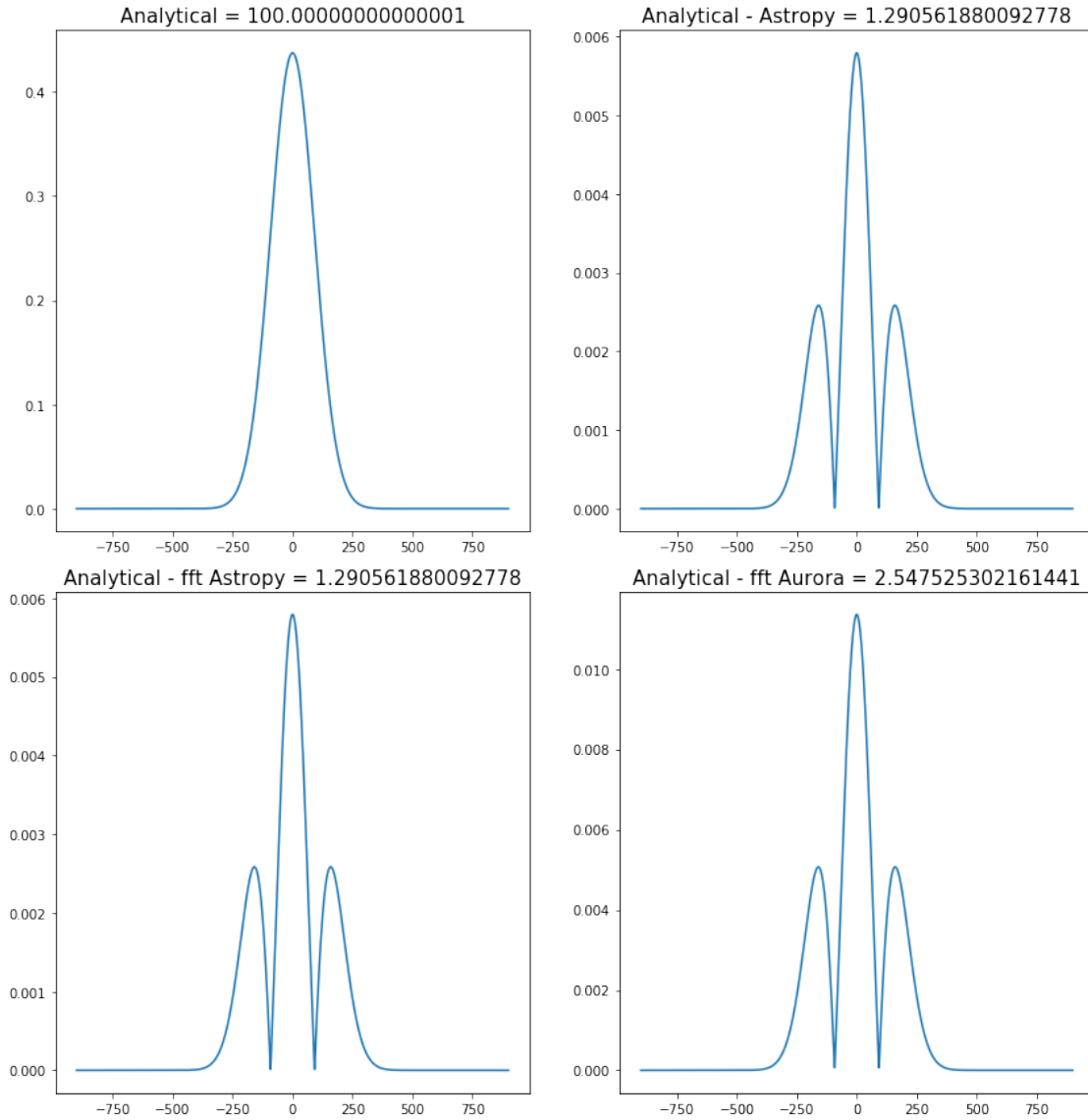
rest = np.abs(z_f - c_astropy.reshape(len(c_astropy)))
axs[0, 1].set_title("Analytical - Astropy = {}".format(rest.sum()), fontsize = 15)
axs[0, 1].plot(x_1, rest)

rest = np.abs(z_f - c_astropy_fft.reshape(len(c_astropy_fft)))
axs[1, 0].set_title("Analytical - fft Astropy = {}".format(rest.sum()),
    fontsize = 15)
axs[1, 0].plot(x_1, rest)

rest = np.abs(z_f - c_fft.reshape(len(c_fft)))
axs[1, 1].set_title("Analytical - fft Aurora = {}".format(rest.sum()), fontsize = 15)
axs[1, 1].plot(x_1, rest)

fig.tight_layout()
plt.show()

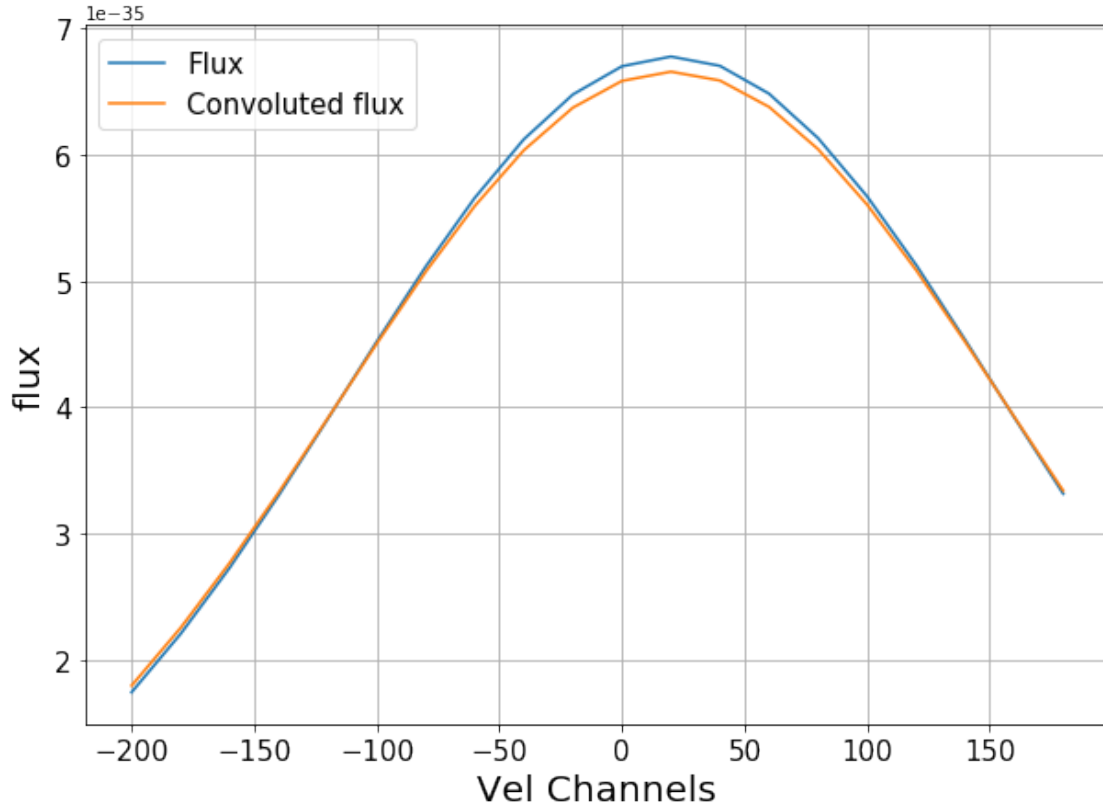
```



```
[10]: rest1 = np.abs(z_f - c_astropy_fft.reshape(len(c_astropy_fft)))

rest2 = np.abs(z_f - c_astropy_fft.reshape(len(c_astropy_fft)))
rest3 = np.abs(z_f - c_fft.reshape(len(c_fft)))

plt.figure(figsize=(8, 8))
plt.plot(x_1, z_f, label = "Analytical")
plt.plot(x_1, rest1, label = "Analytical - Astropy")
plt.plot(x_1, rest2, label = "Analytical - fft Astropy")
plt.plot(x_1, rest3, label = "Analytical - fft Aurora")
plt.legend(fontsize = 12)
plt.show()
```



## 7 Comparison of convolutions with respect to *Analytical* :

7.1  $\frac{|Analytical - convolution|}{Analytical} \cdot 100\%$

```
[11]: fig, axs = plt.subplots(nrows = 2, ncols = 2, figsize=(12, 12))

fig.subplots_adjust(hspace=0.3)

axs[0, 0].set_title("Analytical = {}".format(z_f.mean()), fontsize = 15)
axs[0, 0].plot(x_1, z_f)

rest = np.abs(z_f - c_astropy.reshape(len(c_astropy)))/z_f *100
axs[0, 1].set_title("Analytical - lsf_astro = {}".format(round(rest.mean()))),
    ↳ fontsize = 15)
axs[0, 1].plot(x_1, rest)
axs[0, 1].axis([-70,70, 0,5])

rest = np.abs(z_f - c_astropy_fft.reshape(len(c_astropy_fft)))/z_f *100
```



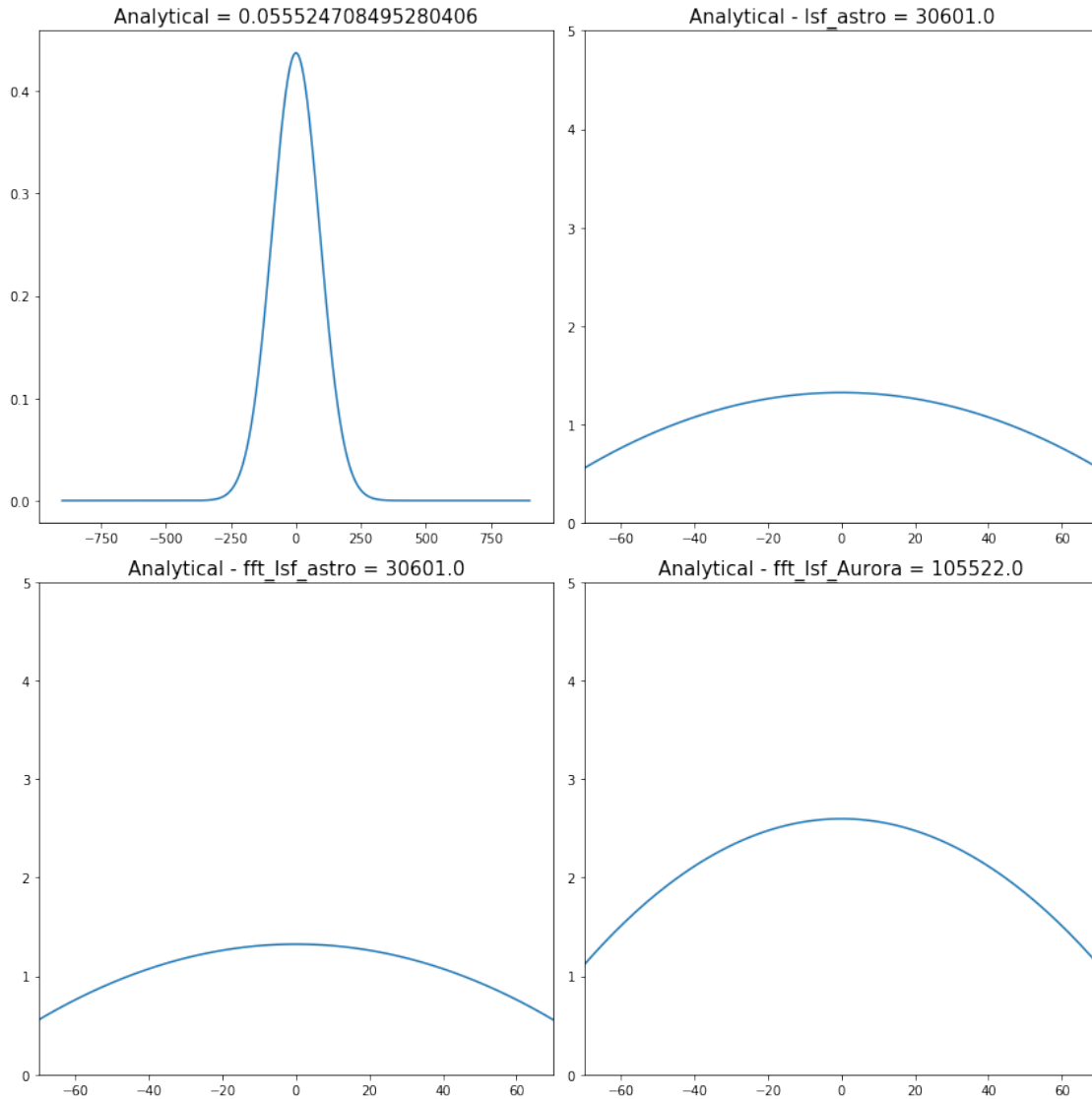
```

axs[1, 0].set_title("Analytical - fft_lsf_astro = {}".format(round(rest.
    ↪mean()))), fontsize = 15)
axs[1, 0].plot(x_1, rest)
axs[1, 0].axis([-70,70, 0,5])

rest = np.abs(z_f - c_fft.reshape(len(c_fft)))/z_f *100
axs[1, 1].set_title("Analytical - fft_lsf_Aurora = {}".format(round(rest.
    ↪mean()))), fontsize = 15)
axs[1, 1].plot(x_1, rest)
axs[1, 1].axis([-70,70, 0,5])

fig.tight_layout()
#plt.axis([-60,60, 0,20])
plt.show()

```





# Test\_convolutions\_spatial

July 29, 2020

## 0.1 Spatial convolutions test

```
[1]: # Necessary libraries are imported
import sys
import scipy
import logging
import numpy as np
from scipy import fftpack
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
from numpy.fft import fftshift, rfft2, irfft2
sys.path.append("/home/rolando9807/Documentos/tesis/Aurora/")

import astropy.convolution
from bisect import bisect_left

from aurora import aurora as au
from aurora import set_output as sto
from aurora import constants as ct
from aurora import convolutions as cv
```

## 1 Spatial functions

```
[2]: # Gaussian 2D function
def gaus2d(x=0, y=0, mx=0, my=0, sx=0.6, sy=0.6):
    gaus = (1 / (2 * np.pi * sx * sy)) * np.exp(-( ((x - mx)**2 / (2 * sx**2)) +
    ↪ ((y - my)**2 / (2 * sy**2)) ))
    return gaus

# Kernel creator function
def create_psf(scale_sigma, size = 20):
    psf = astropy.convolution.Gaussian2DKernel(scale_sigma,
    ↪ x_size = cv.next_odd(size * scale_sigma), y_size = cv.next_odd(size *
    ↪ scale_sigma))
    psf = np.array(psf)
```

```
psf = psf / psf.sum()
return (psf, scale_sigma)
```

## 2 Gaussian 2D signal and pdf kernels

```
[3]: # The Gaussian signal is defined based on the kernel sigma,
# with sufficient resolution not to store numerical errors.

s_psf = 15 # Sigma psf
s_gaus = 6 * s_psf # Sigma gaussian signal
n_s = 20 # Sigma numbers to solve Gaussian signal

size_z = cv.next_odd(n_s * s_gaus) # Boundaries of the gaussians
x_1 = np.arange(-int(size_z/2), int(size_z/2)+1) # X points to solve Gaussian
↳signal
y_1 = np.arange(-int(size_z/2), int(size_z/2)+1) # y points to solve Gaussian
↳signal
x_1, y_1 = np.meshgrid(x_1, y_1) # 2D points to solve Gaussian signal

A = 1e4
z = A*gaus2d(x_1, y_1, sx = s_gaus, sy = s_gaus) # Gaussian 2D signal

# Analytical kernel
size_psf = cv.next_odd(n_s * s_psf)
x_psf = np.arange(-int(size_psf/2), int(size_psf/2) + 1)
y_psf = np.arange(-int(size_psf/2), int(size_psf/2) + 1)
x_psf, y_psf = np.meshgrid(x_psf, x_psf)

psf = gaus2d(x_psf, y_psf, sx= s_psf, sy= s_psf)

no_psf = psf.sum()
psf /= psf.sum()

[4]: # Astropy kernel
psf_1 = create_psf(s_psf, size = n_s)[0]

# Aurota Kernel (padding of the kernel as a function of the Gaussian signal to
↳Aurora fft)
x, y = z.shape
fshape = fftpack.next_fast_len(x + psf_1.shape[0])
center = fshape - (fshape+1) // 2
new_psf = np.zeros([fshape, fshape])
index = slice(center - psf_1.shape[0] // 2, center + (psf_1.shape[0] + 1) // 2)
new_psf[index, index] = psf_1
```

### 3 Gaussian and kernels

```
[5]: zoomA = int(new_psf.shape[0] / psf.shape[0])
zoomA2 = int(new_psf.shape[0] / zoomA)

[6]: cmap = 'jet'
vmin = psf.min()
vmax = psf.max()

fig, axs = plt.subplots(nrows = 2, ncols = 2, figsize=(12, 10))

fig.subplots_adjust(hspace=0.3)

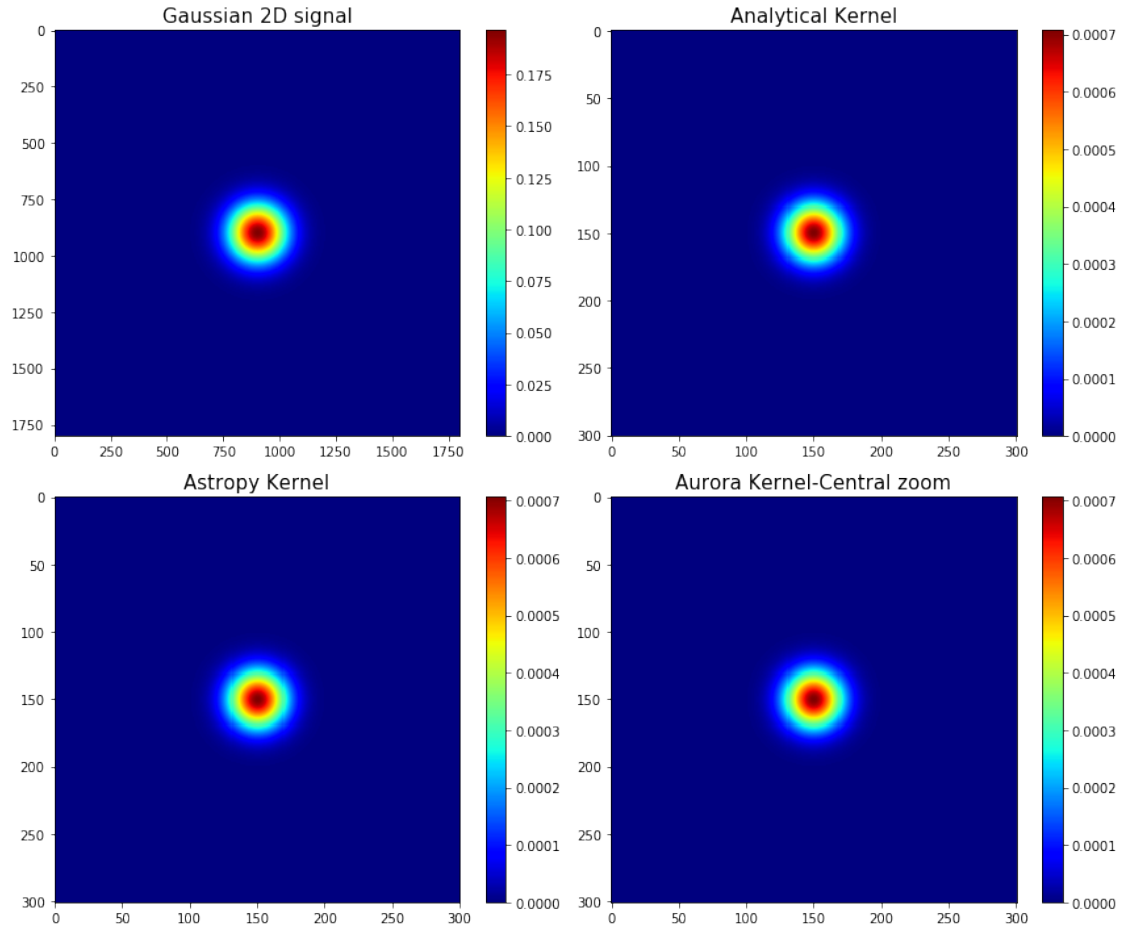
axs[0, 0].set_title("Gaussian 2D signal", fontsize = 15)
f1 = axs[0, 0].imshow(z, cmap = cmap)
fig.colorbar(f1, ax=axs[0, 0])

axs[0, 1].set_title("Analytical Kernel", fontsize = 15)
f2 = axs[0, 1].imshow(psf, cmap = cmap, vmin = vmin, vmax = vmax)
fig.colorbar(f2, ax=axs[0, 1])

axs[1, 0].set_title("Astropy Kernel", fontsize = 15)
f3 = axs[1, 0].imshow(psf_1, cmap = cmap, vmin = vmin, vmax = vmax)
fig.colorbar(f3, ax=axs[1, 0])

axs[1, 1].set_title("Aurora Kernel-Central zoom", fontsize = 15)
f4 = axs[1, 1].imshow(new_psf[index,index], cmap = cmap, vmin = vmin, vmax =
    ↪vmax)
fig.colorbar(f4, ax=axs[1, 1])

fig.tight_layout()
plt.show()
```



## 4 Convolutions

```
[7]: # Analytical convoluion
sx = np.sqrt(s_gaus**2 + s_psf**2) # New sigma
sy = sx
z_f = A*gaus2d(x_1, y_1, sx = sx, sy = sy) # Analytical convolution

# Astropy convolution
c_astro = cv.spatial_convolution_aurora_fft(np.reshape(z, (1, z.shape[0], z.
    ↪ shape[1])), psf_1)

# Astropy fft convolution
c_astro_fft = cv.spatial_convolution_astropy_fft(np.reshape(z, (1, z.shape[0], z.
    ↪ shape[1])), psf_1)

# Aurora fft convolution
```

```
c_fft = cv.spatial_convolution_aurora_fft(np.reshape(z,(1,z.shape[0],z.
↪shape[1])),psf_1)
```

```
[8]: # Sum of signal flux after convolutions
print(r'$z_{sum}$=',z.sum())
print(r'Analitical convolution $z_f_{sum}$=', z_f.sum())
print(r'convolution $astro_{sum}$=',c_astro[0].sum())
print(r'convolution astropy $fft_{sum}$=',c_astro_fft[0].sum())
print(r'convolution aurora $fft_{sum}$=',c_fft[0].sum())
```

```
$z_{sum}$= 9999.999999999996
Analitical convolution $z_f_{sum}$= 9999.999999999996
convolution $astro_{sum}$= 9999.999999999999
convolution astropy $fft_{sum}$= 9999.999999999996
convolution aurora $fft_{sum}$= 10000.000000000002
```

```
[9]: cmap = 'jet'
vmin = 0
vmax = c_astro[0].max()

#zoom
zoom = 4
x_map = int(z.shape[1] / zoom)
y_map = int(z.shape[0] / zoom)

fig, axs = plt.subplots(nrows = 3,  ncols = 2, figsize=(15, 15))

fig.subplots_adjust(hspace=0.3)

axs[0, 0].set_title("Gaussian 2D signal", fontsize = 15)
f1 = axs[0, 0].imshow(z[y_map:3*y_map, x_map:3*x_map], cmap = cmap)
fig.colorbar(f1, ax=axs[0, 0])

axs[0, 1].set_title("Analytical Kernel", fontsize = 15)
f2 = axs[0, 1].imshow(psf, cmap = cmap)
fig.colorbar(f2, ax=axs[0, 1])

axs[1, 0].set_title("Analitical", fontsize = 15)
f1 = axs[1, 0].imshow(z_f[y_map:3*y_map, x_map:3*x_map], cmap = cmap, vmin = ↵
↪vmin, vmax = vmax)
fig.colorbar(f1, ax=axs[1, 0])

axs[1, 1].set_title("Asropy Convolution", fontsize = 15)
f2 = axs[1, 1].imshow(c_astro[0][y_map:3*y_map, x_map:3*x_map], cmap = cmap, ↵
↪vmin = vmin, vmax = vmax)
```

```

fig.colorbar(f2, ax=axes[1, 1])

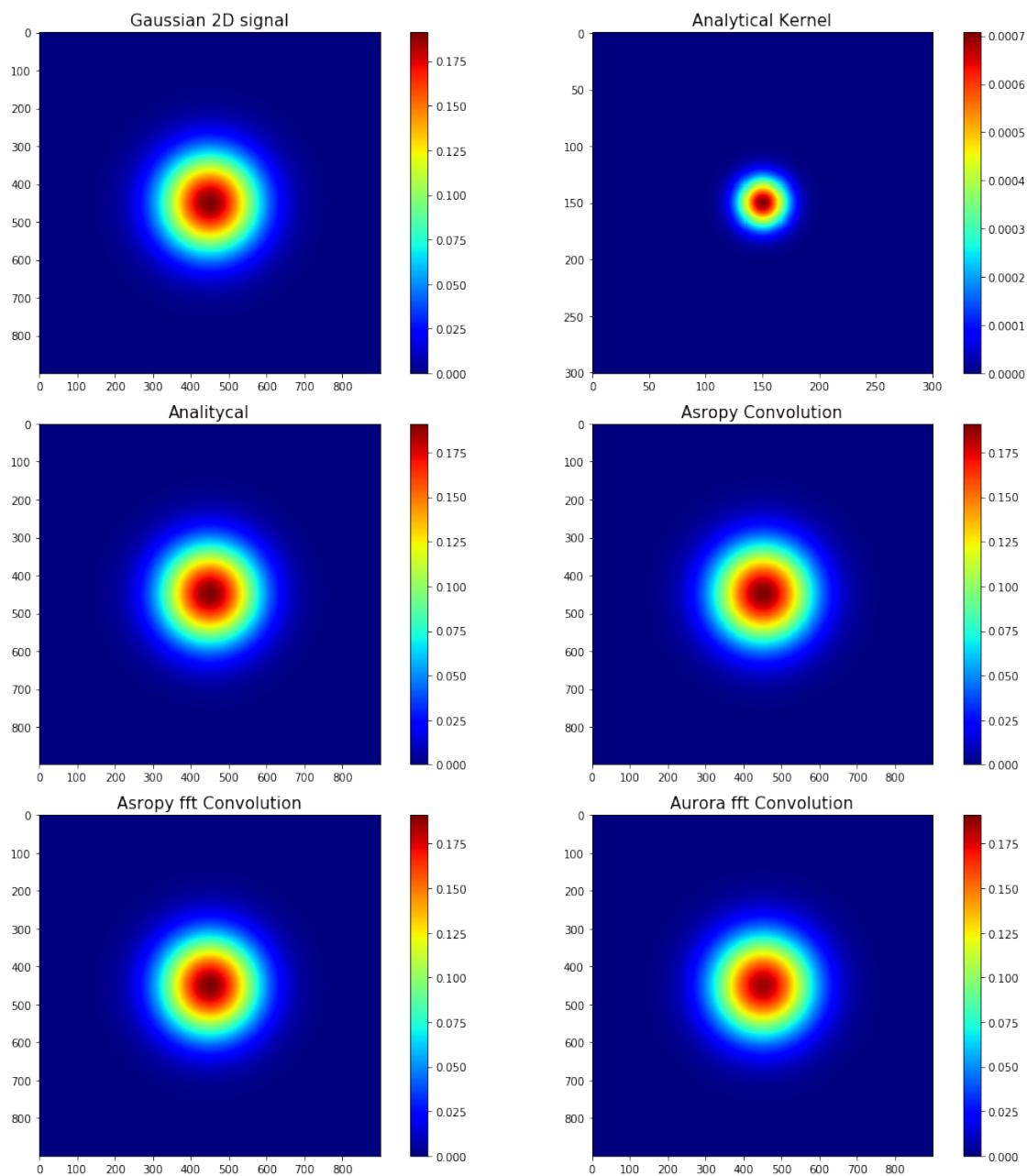
axes[2, 0].set_title("Asropy fft Convolution", fontsize = 15)
f3 = axes[2, 0].imshow(c_astro_fft[0][y_map:3*y_map, x_map:3*x_map], cmap = _
↪ cmap, vmin = vmin, vmax = vmax)
fig.colorbar(f3, ax=axes[2, 0])

axes[2, 1].set_title("Aurora fft Convolution", fontsize = 15)
f4 = axes[2, 1].imshow(c_fft[0][y_map:3*y_map, x_map:3*x_map], cmap = cmap, vmin=_
↪ vmin, vmax = vmax)
fig.colorbar(f4, ax=axes[2, 1])

fig.tight_layout()
plt.show()

```





## 5 comparison of convolutions with respect to *Analytical* :

### 5.1 |*Analytical – convolution*|

```
[10]: #zoom
zoom = 4
x_map = int(z.shape[1] / zoom)
y_map = int(z.shape[0] / zoom)

fig, axs = plt.subplots(nrows = 3, ncols = 2, figsize=(12, 15))

fig.subplots_adjust(hspace=0.3)

axs[0, 0].set_title("Gaussian 2D signal", fontsize = 15)
f1 = axs[0, 0].imshow(z[y_map:3*y_map, x_map:3*x_map], cmap = cmap)
fig.colorbar(f1, ax=axs[0, 0])

axs[0, 1].set_title("Analytical Kernel", fontsize = 15)
f2 = axs[0, 1].imshow(psf, cmap = cmap)
fig.colorbar(f2, ax=axs[0, 1])

axs[1, 0].set_title("Analytical = {}".format(z_f.sum()), fontsize = 15)
f1 = axs[1, 0].imshow(z_f[y_map:3*y_map, x_map:3*x_map], cmap = cmap)
fig.colorbar(f1, ax=axs[1, 0])

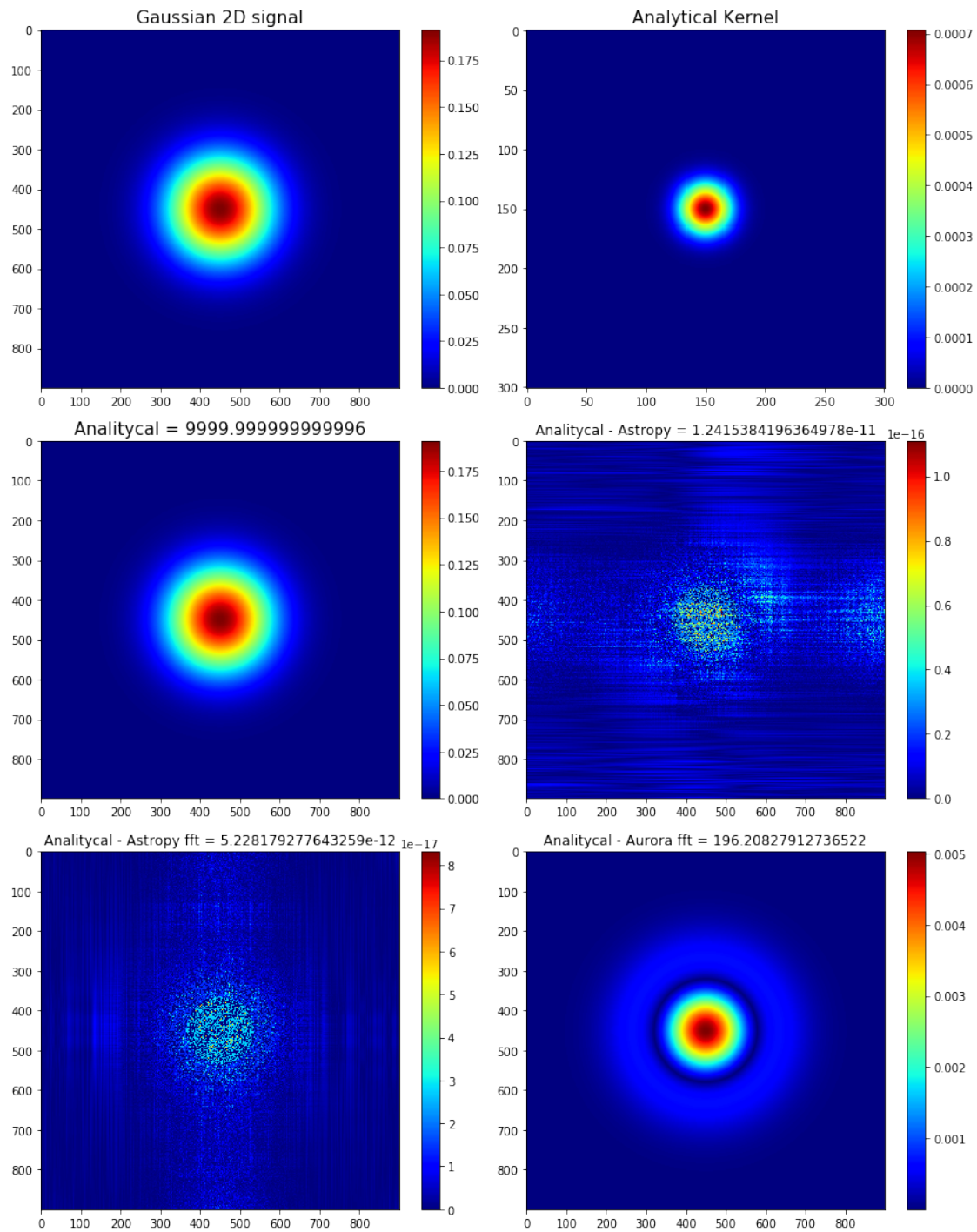
img = np.abs(z_f-c_astro[0])
axs[1, 1].set_title("Analytical - Astropy = {}".format(img.sum()), fontsize = 12)
f1 = axs[1, 1].imshow(img[y_map:3*y_map, x_map:3*x_map], cmap = cmap)
fig.colorbar(f1, ax=axs[1, 1])

img = np.abs(z_f-c_astro_fft[0])
axs[2, 0].set_title("Analytical - Astropy fft = {}".format(img.sum()), fontsize = 12)
f1 = axs[2, 0].imshow(img[y_map:3*y_map, x_map:3*x_map], cmap = cmap)
fig.colorbar(f1, ax=axs[2, 0])

img = np.abs(z_f - c_fft[0])
axs[2, 1].set_title("Analytical - Aurora fft = {}".format(img.sum()), fontsize = 12)
f2 = axs[2, 1].imshow(img[y_map:3*y_map, x_map:3*x_map], cmap = cmap)
fig.colorbar(f2, ax=axs[2, 1])

fig.tight_layout()
```

```
plt.show()
```



```
[11]: #zoom
zoom = 4
x_map = int(z.shape[1] / zoom)
```

```

y_map = int(z.shape[0] / zoom)

vmin = 0
vmax = z.max()

fig, axs = plt.subplots(nrows = 3, ncols = 2, figsize=(12, 15))

fig.subplots_adjust(hspace=0.3)

axs[0, 0].set_title("Gaussian 2D signal", fontsize = 15)
f1 = axs[0, 0].imshow(z[y_map:3*y_map, x_map:3*x_map], cmap = cmap)
fig.colorbar(f1, ax=axs[0, 0])


axs[0, 1].set_title("Analytical Kernel", fontsize = 15)
f2 = axs[0, 1].imshow(psf, cmap = cmap)
fig.colorbar(f2, ax=axs[0, 1])

axs[1, 0].set_title("Analytical = {}".format(z_f.sum()), fontsize = 15)
f1 = axs[1, 0].imshow(z_f[y_map:3*y_map, x_map:3*x_map], cmap = cmap, vmin =
    ↪vmin, vmax = vmax)
fig.colorbar(f1, ax=axs[1, 0])

img = np.abs(z_f-c_astro[0])
axs[1, 1].set_title("Analytical - Astropy = {}".format(img.sum()), fontsize =
    ↪12)
f1 = axs[1, 1].imshow(img[y_map:3*y_map, x_map:3*x_map], cmap = cmap, vmin =
    ↪vmin, vmax = vmax)
fig.colorbar(f1, ax=axs[1, 1])

img = np.abs(z_f-c_astro_fft[0])
axs[2, 0].set_title("Analytical - Astropy fft = {}".format(img.sum()), fontsize
    ↪= 12)
f1 = axs[2, 0].imshow(img[y_map:3*y_map, x_map:3*x_map], cmap = cmap, vmin =
    ↪vmin, vmax = vmax)
fig.colorbar(f1, ax=axs[2, 0])

img = np.abs(z_f - c_fft[0])
axs[2, 1].set_title("Analytical - Aurora fft = {}".format(img.sum()), fontsize
    ↪= 12)
f2 = axs[2, 1].imshow(img[y_map:3*y_map, x_map:3*x_map], cmap = cmap, vmin =
    ↪vmin, vmax = vmax)
fig.colorbar(f2, ax=axs[2, 1])



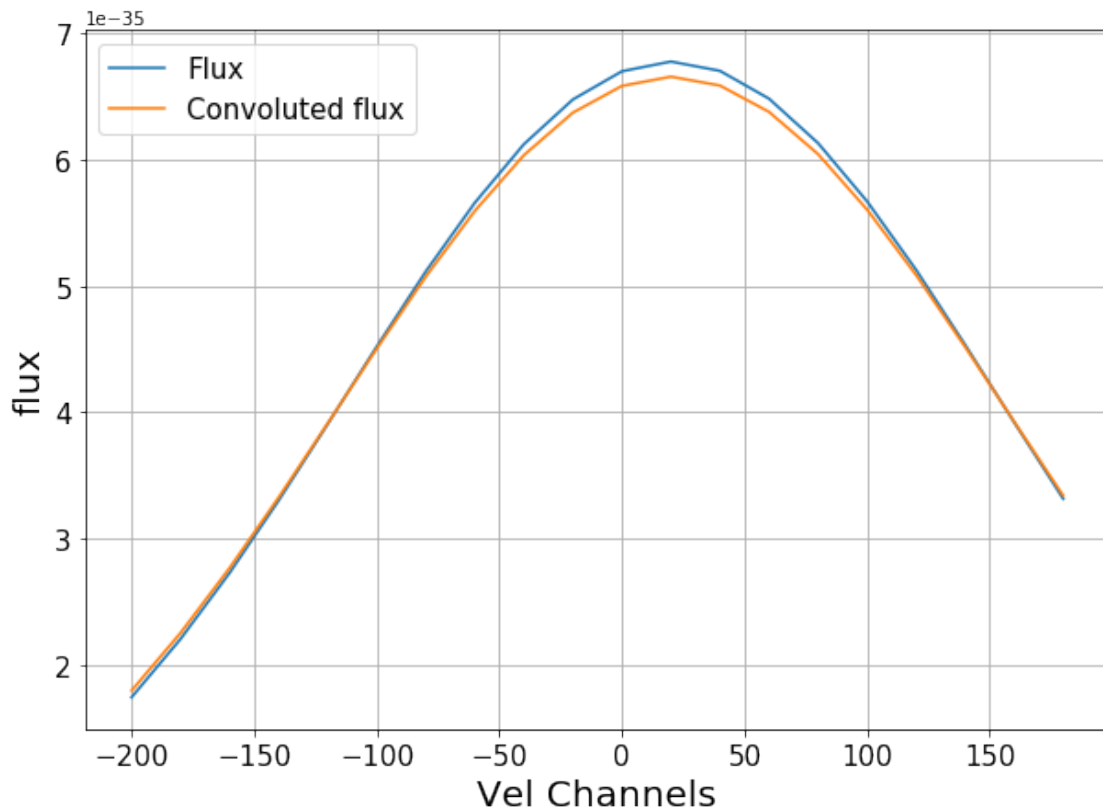
```

```

#axs[3, 0].set_title("Analytical - Benoit = {}".format(img.sum()), fontsize = 12)
#f3 = axs[3, 0].imshow(img[y_map:3*y_map, x_map:3*x_map], cmap = cmap, vmin = vmin, vmax = vmax)
#fig.colorbar(f3, ax=axs[3, 0])

#axs[3, 1].remove()
fig.tight_layout()
plt.show()

```



```

[23]: line = int(z_f.shape[0]/2)
img1 = np.abs(z_f-c_astro[0])
img2 = np.abs(z_f-c_astro_fft[0])
img3 = np.abs(z_f - c_fft[0])

fig = plt.figure(figsize=(12, 9))
plt.title('Center pixels in y', fontsize = 20)
plt.xlabel('x', fontsize = 20)
plt.ylabel('flux', fontsize = 20)

```

```

plt.plot(x_1[0,:],
         (z_f[line,:]), '-', label='Analytical')

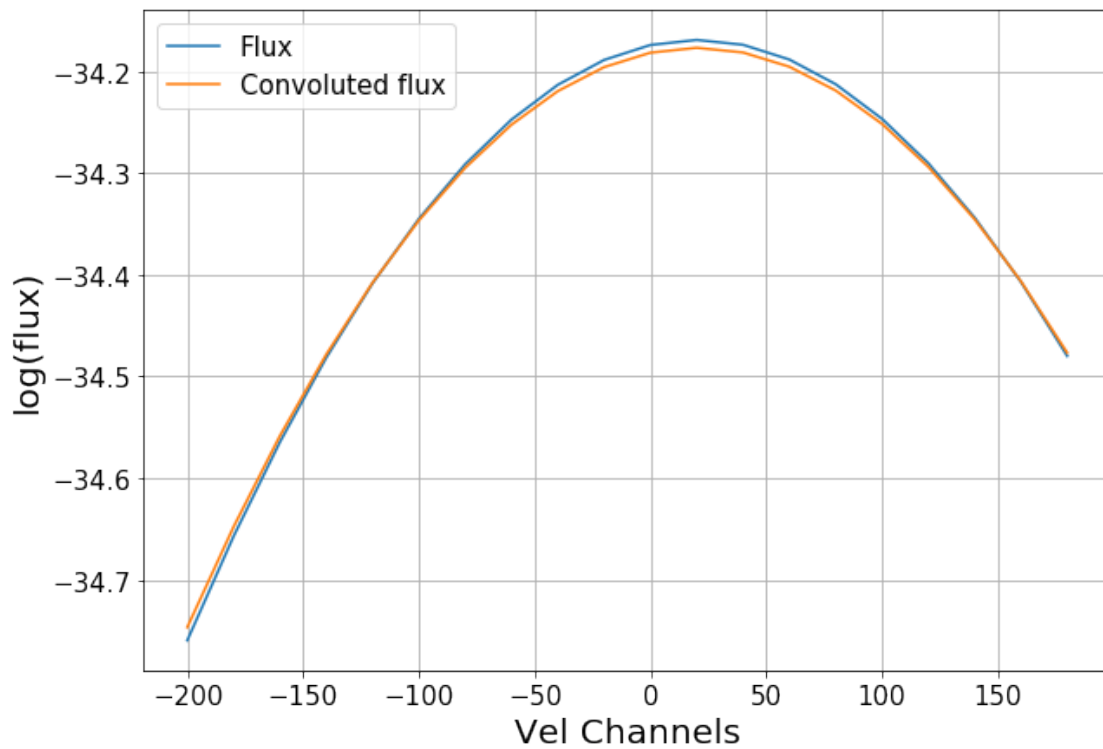
plt.plot(x_1[0,:],
         (img1[line,:]), '-', label='Astropy')

plt.plot(x_1[0,:],
         (img2[line,:]), '-', label='Astropy fft Flux')

plt.plot(x_1[0,:],
         (img3[line,:]), '-', label='Aurora fft Flux')

plt.legend(fontsize = 15, loc=0)
plt.xticks(size = 15)
plt.yticks(size = 15)
plt.grid()
plt.show()

```



[ ]:

# Test array\_operations.py

July 19, 2020

```
[1]: # Necessary libraries are imported
import sys
import numpy as np
from scipy import ndimage
from scipy import special
from scipy import interpolate
import matplotlib.pyplot as plt
from nose.tools import assert_equal
from matplotlib.colors import LogNorm

import logging

sys.path.append("/home/rolando9807/Documentos/tesis/Aurora/")
from aurora import aurora as au
from aurora import array_operations as ao
from aurora import datacube as dc

[2]: # Functions to test
def bin_array(x, n, axis=0):
    """
    Bin the elements in one direction of a 2D or 3D array.

    Parameters
    -----
    input_file : str
        Snapshot file name. It can be a simulation file RAMSES or GADGET.
    x : ndarray (3D or 2D)
        Array to be binned.
    n : int
        Binning factor, i.e, number of elements to be added together
        must be a divisor of the number of elements on the axis.
    axis : int
        Axis along which the binning operation takes place.

    Returns
    -----
    array : ndarray (3D or 2D)
```

```

    Binned Array.
    """

    # Code flow:
    # =====
    # > Switch axes to work along axis 0 as default
    # > Perform the binning operation
    x = np.swapaxes(x, 0, axis)
    if x.shape[0]%n != 0:
        logging.error(f"// n is not a divisor of the number of elements of the_
↪axis")
        sys.exit()
    dim = int(x.shape[0] / n)
    if len(x.shape) == 2:
        x = x[:dim*n, :]
        array = np.zeros([dim, x.shape[1]])
        for i in range(dim):
            array[i, :] = x[i*n:(i+1)*n, :].sum(axis = 0)
    elif len(x.shape) == 3:
        x = x[:dim*n, :, :]
        array = np.zeros([dim, x.shape[1], x.shape[2]])
        for i in range(n):
            array += x[i::n, :, :]
    array = np.swapaxes(array, 0, axis)
    return array

def cube_resampling(cube, new_cube):
    """
    Interpolate a 3D master datacube to new spatial/spectral coordinates
    at once.

    Parameters
    -----
    cube : aurora.datacube.DatacubeObj
        Old datacube. Instance of class DatacubeObj whose attributes make
        code computational performance properties available. See definitions
        in datacube.py

    new_cube : aurora.datacube.DatacubeObj
        New datacube. Must store information about spatial/spectral coordinates.
        Instance of class DatacubeObj whose attributes make code computational
        performance properties available. See definitions in datacube.py
    """

    # Code flow:
    # =====
    # > Calculate the transformation factor to the new spatial/spectral

```



```

# coordinates
# > Determines the spatial coordinates of the new system with respect to
# the original
# > Interpolates into the new data_cube
pixratio = cube.pixsize.value / new_cube.pixsize.value
channelratio = cube.velocity_sampl.value / new_cube.velocity_sampl.value
channelratio_dim = cube.spectral_dim / new_cube.spectral_dim
origin_spatial = (cube.spatial_dim - new_cube.spatial_dim/pixratio
                  - 1. + 1/pixratio) / 2
new_positions = origin_spatial + np.arange(new_cube.spatial_dim)/pixratio
origin_spectral = (cube.spectral_dim - new_cube.spectral_dim/channelratio
                  - 1. + 1/channelratio) / 2
new_channels = origin_spectral + np.arange(new_cube.spectral_dim)/
↳channelratio
X, Y, Z = np.meshgrid(new_positions, new_positions, new_channels)

new_cube.cube = ndimage.map_coordinates(cube.cube * cube.velocity_sampl.
↳value, [Z, X, Y], order=1).T
new_cube.cube = (new_cube.cube * channelratio_dim) / (pixratio**2 *
↳new_cube.velocity_sampl.value)

```

```

[3]: # The cube to probes is loaded
cube_data = dc.DatacubeObj()
cube_data.read_data('test.fits')
cube_data.get_attr()
cube_data.all_maps()

```

## 0.1 Test of the bin\_array function

```

[4]: # A new cube is created where it is halved in spatial dimensions
New_cube = bin_array(cube_data.cube, 2, axis = 1)
New_cube = bin_array(New_cube, 2, axis = 2)

[5]: # The total flux of the new cube is compared to the original cube
print('Original flux = {} ERG.S^-1.cm^-2'.format(cube_data.cube.sum()))
print('New flux = {} ERG.S^-1.cm^-2'.format(New_cube.sum()))
print('Percentage difference = {}%'.format(np.abs(cube_data.cube.sum()
- New_cube.sum())/cube_data.cube.sum() * 100 ))

```

```

Original flux = 7.024129608299879e-13 ERG.S^-1.cm^-2
New flux = 7.02412960829988e-13 ERG.S^-1.cm^-2
Percentage difference = 1.437533210505911e-14%

```

```

[6]: # A new flux map is created by replicating the previous step
New_flux_map = bin_array(cube_data.fluxmap, 2, axis = 0)
New_flux_map = bin_array(New_flux_map, 2, axis = 1)

```

```
[7]: # The total flux of the new map is compared to the original map
print('Original Flux map = {} ERG.S-1.cm-2'.format(cube_data.fluxmap.sum()))
print('New Flux map = {} ERG.S-1.cm-2'.format(New_flux_map.sum()))
print('Percentage difference = {}%'.format(np.abs(cube_data.fluxmap.sum()
- New_flux_map.sum())/cube_data.fluxmap.sum() * 100 ))
```

Original Flux map = 1.4048259216599758e-11 ERG.S<sup>-1</sup>.cm<sup>-2</sup>

New Flux map = 1.404825921659976e-11 ERG.S<sup>-1</sup>.cm<sup>-2</sup>

Percentage difference = 1.1500265684047288e-14%

```
[8]: vmin = cube_data.fluxmap.min()
vmax = cube_data.fluxmap.max()

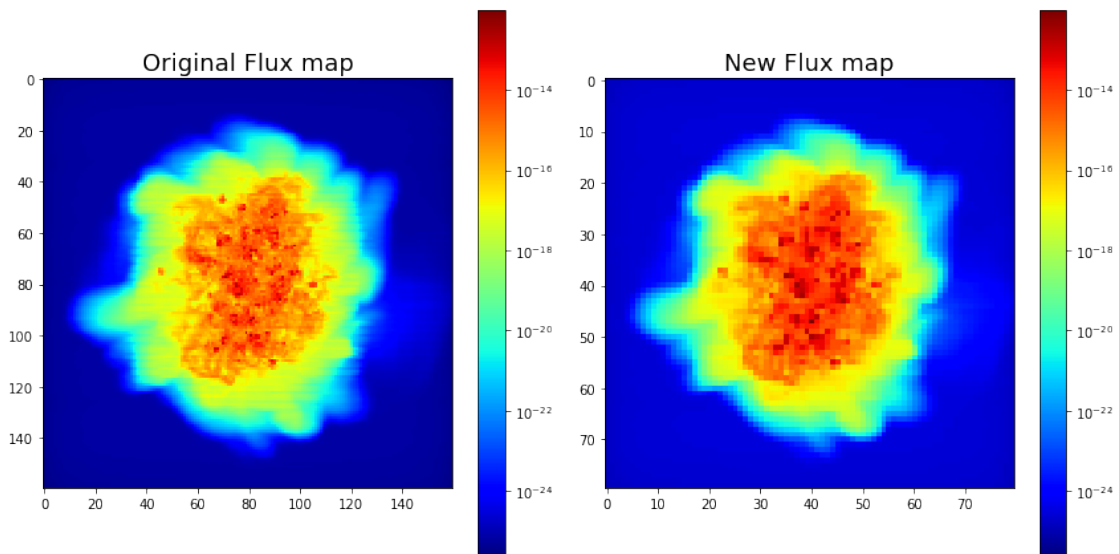
fig, axs = plt.subplots(nrows = 1,  ncols = 2,  figsize=(12, 6))

fig.subplots_adjust(hspace=0.3)

axs[0].set_title("Original Flux map",  fontsize = 18)
f1 = axs[0].imshow(cube_data.fluxmap,  cmap='jet',  vmin = vmin,  vmax = vmax,
↳norm= LogNorm())
fig.colorbar(f1,  ax=axs[0])

axs[1].set_title("New Flux map",  fontsize = 18)
f2 = axs[1].imshow(New_flux_map,  cmap='jet',  vmin = vmin,  vmax = vmax,  norm=
↳LogNorm())
fig.colorbar(f2,  ax=axs[1])

fig.tight_layout()
plt.show()
```



## 0.2 Interpolation test of the cube\_resampling function

```
[9]: # Instance of the DatacubeObj class is created to store the new interpolated cube
N = dc.DatacubeObj()
```

The dimensions of the new cube are assigned as semi-integer multiples of the original dimensions.

```
[10]: n = 1.5 # Spatial dimension multiplier
m = 1.5 # Spectral dimension multiplier

N.assign_attr(cube_data.pixsize.value*n, cube_data.velocity_sampl.value*m,
              cube_data.spatial_dim/n, cube_data.spectral_dim/m)
```

**Note:** If the new dimensions are integer multiples it is recommended to use the `bin_array` function of the `array_operations.py` module.

```
[11]: # The new cube is interpolated with the cube_resampling function
cube_resampling(cube_data, N)
```

The results are compared with the flux of the original cube. > **Note:** The flux stored in the channels of the cube is per unit of speed, for this reason the values must be multiplied by the width of the channels.

```
[12]: flux_original = cube_data.cube * cube_data.velocity_sampl.value
flux_new = N.cube * N.velocity_sampl.value
```

```
[13]: print('Original flux = {} ERG.S^-1.cm^-2'.format(flux_original.sum()))
print('New flux = {} ERG.S^-1.cm^-2'.format(flux_new.sum()))
print('Percentage difference = {}%'.format(np.abs(flux_original.sum()
- flux_new.sum())/flux_original.sum() * 100 ))
```

Original flux = 1.4048259216599761e-11 ERG.S<sup>-1</sup>.cm<sup>-2</sup>

New flux = 1.4249223691587615e-11 ERG.S<sup>-1</sup>.cm<sup>-2</sup>

Percentage difference = 1.4305293765535712%

### 0.2.1 Intensity maps and the spectrum of a pixel are displayed

```
[14]: N.all_maps()
```

```
[17]: vmin = cube_data.fluxmap.min()
vmax = cube_data.fluxmap.max()
```

```

fig, axs = plt.subplots(nrows = 1,  ncols = 2, figsize=(12, 6))

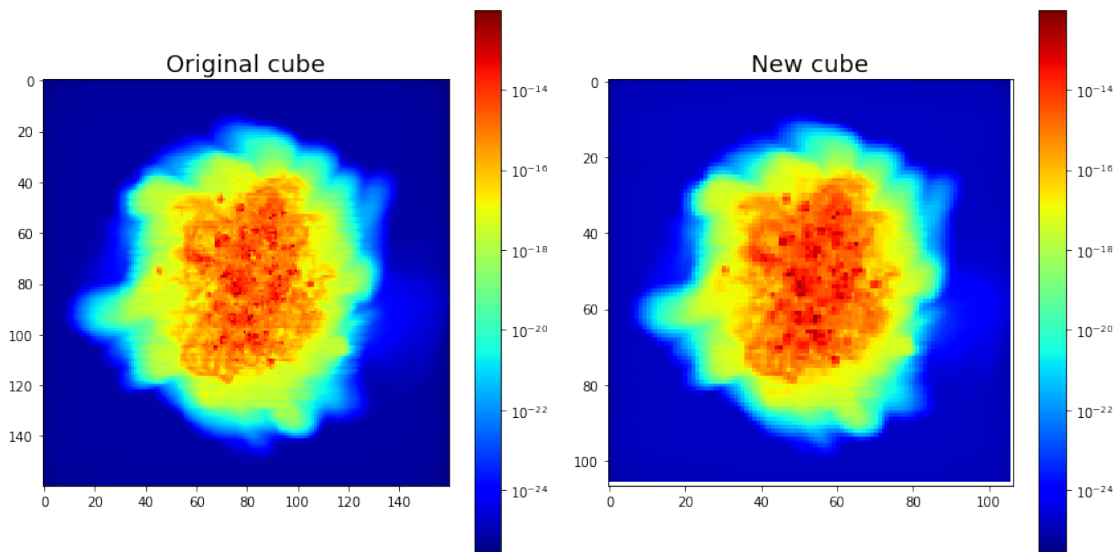
fig.subplots_adjust(hspace=0.3)

axs[0].set_title("Original cube", fontsize = 18)
f1 = axs[0].imshow(cube_data.fluxmap, cmap='jet',  vmin = vmin, vmax = vmax,
    ↪norm= LogNorm())
fig.colorbar(f1, ax=axs[0])

axs[1].set_title("New cube", fontsize = 18)
f2 = axs[1].imshow(N.fluxmap, cmap='jet',  vmin = vmin, vmax = vmax, norm=
    ↪LogNorm())
fig.colorbar(f2, ax=axs[1])

fig.tight_layout()
plt.show()

```



```

[18]: # Original cube
p_x = int(flux_original.shape[2]/4) # Pixel at x
p_y = int(flux_original.shape[1]/4) # Pixel at y

# New cube
p_x_n = int(flux_new.shape[2]/4) # Pixel at x
p_y_n = int(flux_new.shape[1]/4) # Pixel at y

```

```

[21]: fig = plt.figure(figsize=(10, 7))
plt.xlabel('pos', fontsize = 20)

```

```

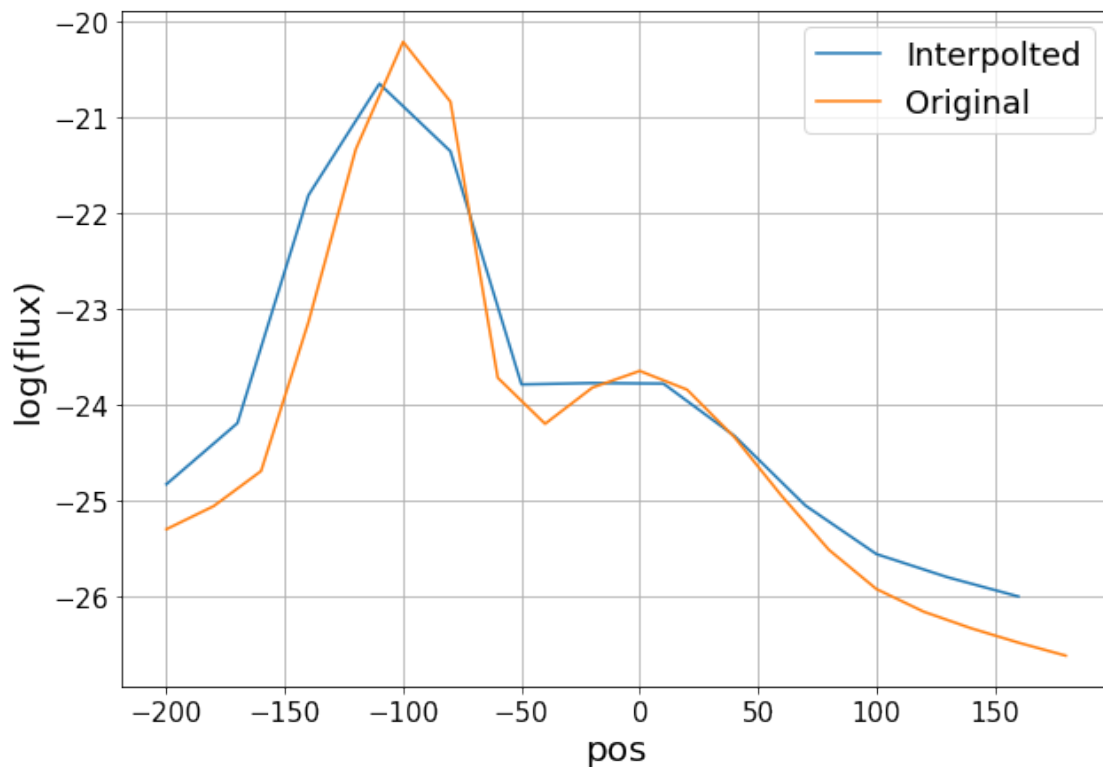
plt.ylabel('log(flux)', fontsize = 20)

plt.plot(N.channels,
         np.log10(flux_new[:, p_y_n, p_x_n]), '-', label='Interpolated')

plt.plot(cube_data.channels,
         np.log10(flux_original[:, p_y, p_x]), '-', label='Original')

plt.legend(fontsize = 18, loc=0)
plt.xticks(size = 15)
plt.yticks(size = 15)
plt.grid()
plt.show()

```



```

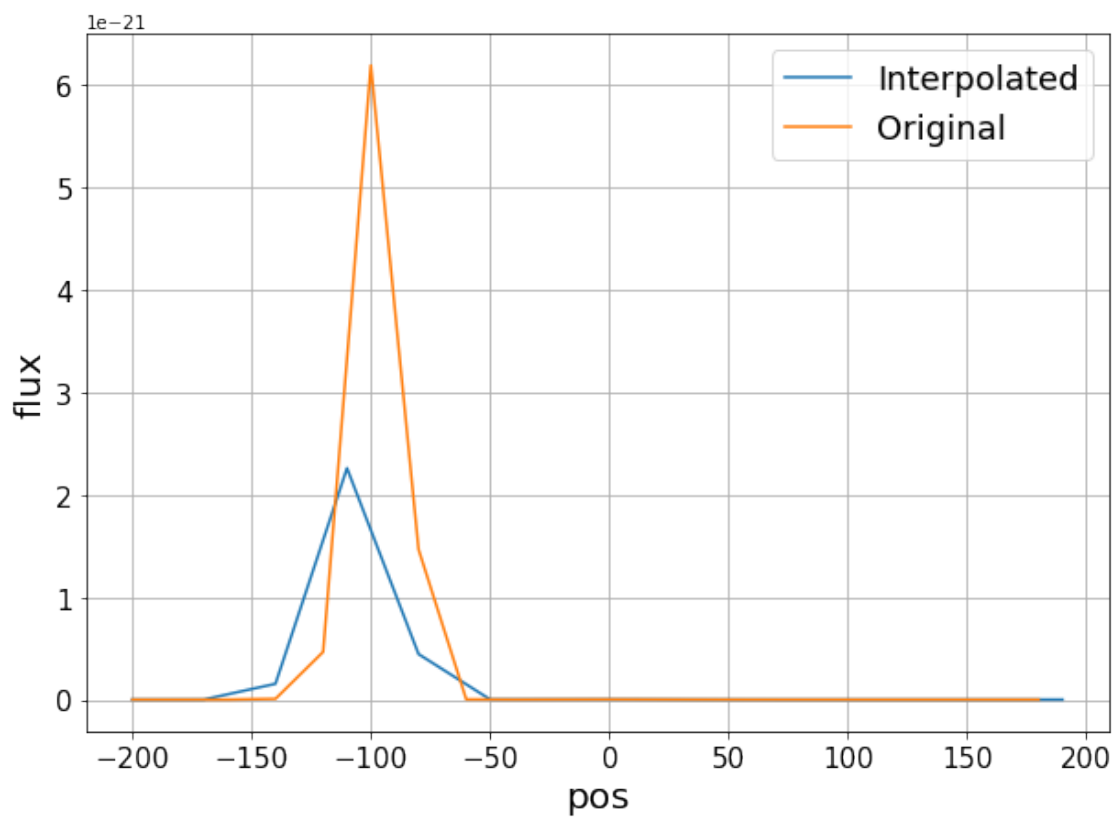
[22]: fig = plt.figure(figsize=(10, 7))
plt.xlabel('pos', fontsize = 20)
plt.ylabel('flux', fontsize = 20)

plt.plot(N.channels,
         (flux_new[:, p_y_n, p_x_n]), '-', label='Interpolated')

```

```
plt.plot(cube_data.channels,  
         (flux_original[:, p_y, p_x]), '-', label='Original')
```

```
plt.legend(fontsize = 18, loc=0)  
plt.xticks(size = 15)  
plt.yticks(size = 15)  
plt.grid()  
plt.show()
```



```
[ ]:
```

```
[ ]:
```