astro_py_scripts

Bill Chen

Jul 22, 2022

BASIC PYTHON

astro_py_scripts is a collection of useful scripts and tricks that can speed up or improve the quality of your Python code in various fields of astronomy.

CHAPTER

ONE

CONTENTS

1.1 Demo

This demo notebook prints “Hello World”. The copyright of this notebook belongs to Bill Chen under the MIT license.
Author: Bill Chen
Created: July 2022
Last modified: July 2022
License: MIT
Modules:

```
python = 3.8.3
```

[1]:
```
print("Hello World")
```

```
Hello World
```

1.2 Fast Fourier Transform using `numpy`

This notebook introduces how to perform Fast Fourier Transform (FFT) properly with the `numpy.fft` module.
Author: Bill Chen
Created: July 2022
Last modified: July 2022
License: MIT
Modules:

```
python = 3.8.3
numpy = 1.18.5
matplotlib = 3.4.3
```

1.2.1 The `numpy.fft` module

In `numpy.fft`, DFT from real space to phase space is defined as (see here):

$$\tilde{a}_f = \sum_{m=0}^{N-1} a_m e^{-2\pi i f x_m}$$

where $x_m = m\Delta x$, and Δx is the sampling interval in real space. Let's compare it with the continuous FT:

$$\tilde{a}(f) = \mathcal{F}\{a(x)\}(f) = \int a(x) e^{-2\pi i f x} dx$$

Normally, this is discretized as

$$\tilde{a}(f) = \mathcal{F}\{a(x)\}(f) \simeq \sum_{m=0}^{N-1} a_m e^{-2\pi i f x_m} \Delta x = \tilde{a}_f \Delta x$$

Therefore, there is a Δx difference in the normalization between `numpy.fft` and continuous FT.

Moreover, `numpy.fft` defines DFT from phase space to real space as

$$a_m = \frac{1}{N} \sum_{k=0}^{N-1} \tilde{a}_k e^{2\pi i f_k x_m}$$

where $f_k = k\Delta f$, and Δf is the sampling interval in phase space. However, the continuous inverse FT is

$$a(x) = \mathcal{F}^{-1}\{\tilde{a}(f)\}(x) = \int \tilde{a}(f) e^{2\pi i f x} df$$

Normally, this is discretized as

$$a(x) = \mathcal{F}^{-1}\{\tilde{a}(f)\}(x) \simeq \sum_{k=0}^{N-1} \tilde{a}_k e^{2\pi i f_k x_m} \Delta f = a_m N \Delta f$$

Therefore, there is an $N\Delta f$ difference in the normalization between `numpy.fft.ifft` and continuous inverse FT.

1.2.2 Basic usage: 1D case

Real space to phase space

Tophat function $\mathrm{rect(x)}$ as an example:

$$\mathcal{F}\{rect(x)\}(f) = \frac{\sin(\pi f)}{\pi f} = \mathrm{sinc}(f)$$

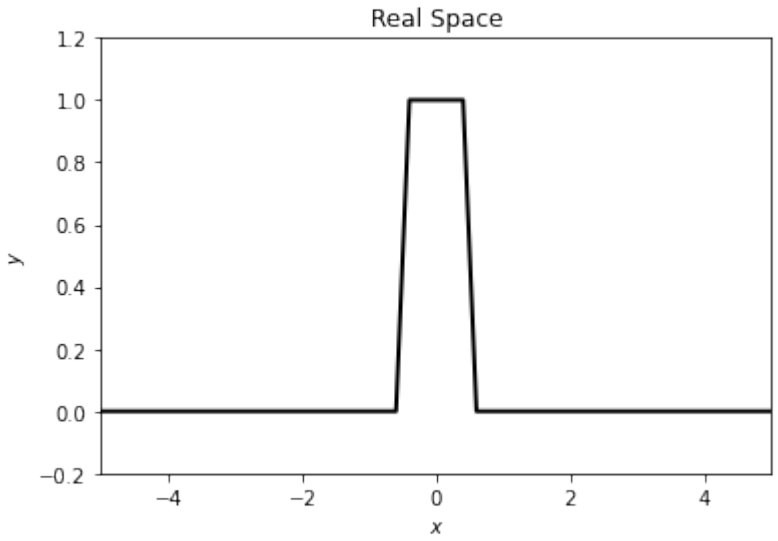
Here, sinc is the same definition as `numpy.sinc`. Note that the below FFT result is not exactly the same as the sinc function due to under-sampling.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

xmin = -5
xmax = 5
N = 100+1 # better be (xmax - xmin)^2, for comparable real space and k-space range

x = np.linspace(xmin, xmax, N)
```

(continues on next page)

(continued from previous page)

```
y = np.heaviside(0.5-np.abs(x),0.5) # tophat, corresponding to square pulse wave

fig, ax0 = plt.subplots(1, 1, figsize=(6,4))

ax0.plot(x, y, c='k', lw=2)

ax0.set_title(r'Real Space')
ax0.set_xlabel(r'$x$')
ax0.set_ylabel(r'$y$')
ax0.set_xlim(xmin, xmax)
ax0.set_ylim(-0.2, 1.2)

plt.show()
```

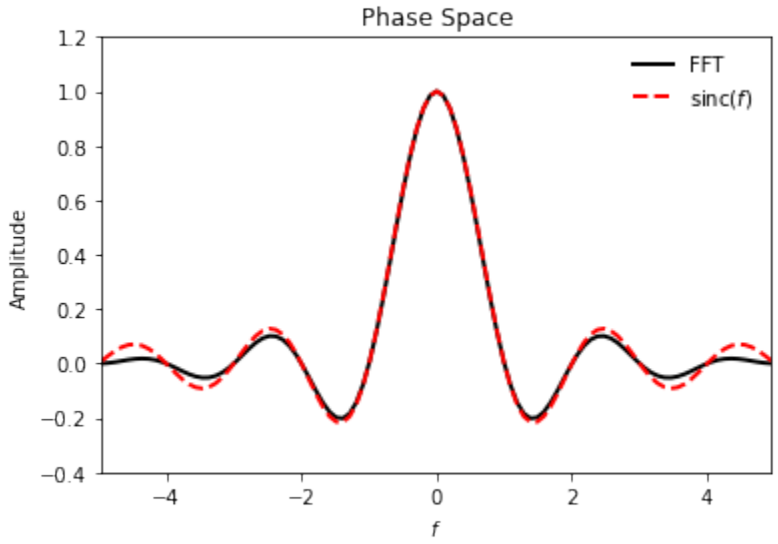


[2]:
```
y_shift = np.fft.ifftshift(y) # important! This centralizes the pattern

dx = x[1] - x[0] # sampling interval
sp = np.fft.fft(y_shift) * dx # need to multiply dx as FT is integral not summation
f = np.fft.fftfreq(N, dx)

sp = np.fft.fftshift(sp) # not necessarily, but for better plot
f = np.fft.fftshift(f)

fmax = np.max(f)
fmin = np.min(f)

# make lot

fig, ax0 = plt.subplots(1, 1, figsize=(6,4))

ax0.plot(f, sp.real, c='k', lw=2, label=r'FFT')
ax0.plot(f, np.sinc(f), c='r', lw=2, ls='--', label=r'${\rm sinc}(f)$')

ax0.set_title(r'Phase Space')
ax0.set_xlabel(r'$f$')
ax0.set_ylabel(r'Amplitude')
ax0.set_xlim(fmin, fmax)
```

(continues on next page)

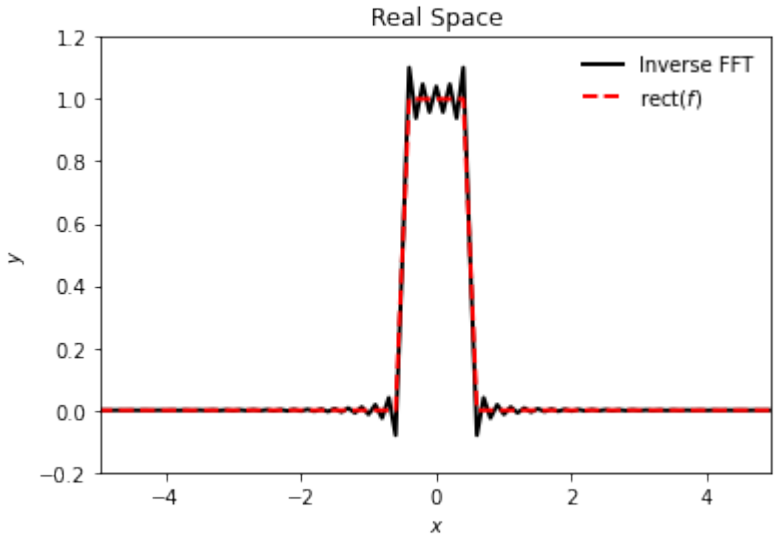
(continued from previous page)

```
ax0.set_ylim(-0.4, 1.2)
ax0.legend(frameon=False)

plt.show()
```



Phase space to real space

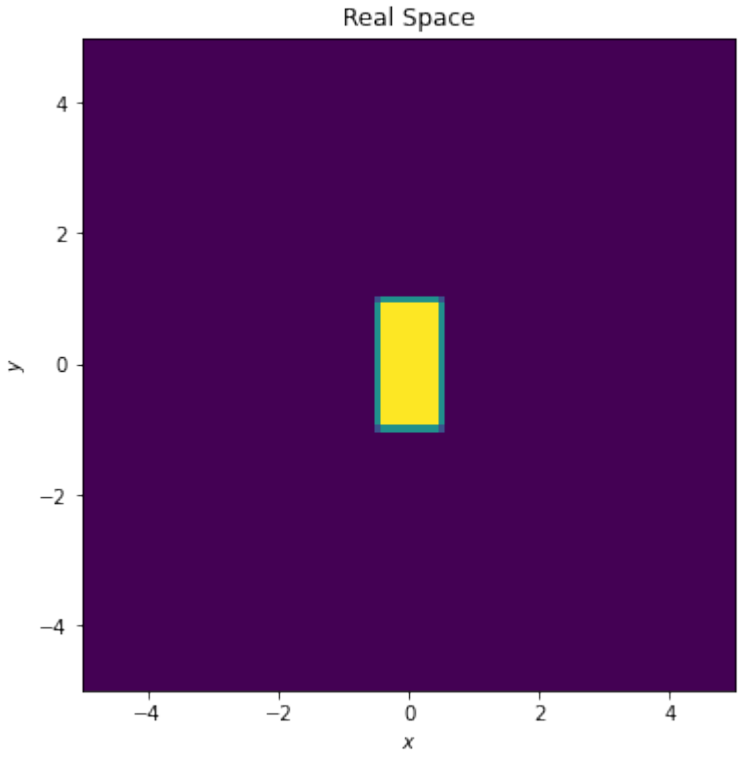
[3]:
```
amp = np.sinc(f)
amp_shift = np.fft.ifftshift(amp) # important! This centralizes the pattern

df = f[1] - f[0] # sampling interval
sp = np.fft.ifft(amp_shift) * N * df # need to multiply N and df according to numpy.
↪fft.ifft
x = np.fft.fftfreq(N, df)

sp = np.fft.fftshift(sp) # not necessarily, but for better plot
x = np.fft.fftshift(x)

xmax = np.max(x)
xmin = np.min(x)

# make lot

fig, ax0 = plt.subplots(1, 1, figsize=(6,4))

ax0.plot(x, sp.real, c='k', lw=2, label=r'Inverse FFT')
ax0.plot(x, y, c='r', lw=2, ls='--', label=r'${\rm rect}(f)$')

ax0.set_title(r'Real Space')
ax0.set_xlabel(r'$x$')
ax0.set_ylabel(r'$y$')
ax0.set_xlim(fmin, fmax)
ax0.set_ylim(-0.2, 1.2)
ax0.legend(frameon=False)

plt.show()
```



1.2.3 Basic usage: 2D case

Rectangular slit as an example.

[4]:
```
xmin = -5
xmax = 5
ymin = -5
ymax = 5
N = 100+1

slit_width = 1
slit_height = 2

x = np.linspace(xmin, xmax, N)
y = np.linspace(ymin, ymax, N)
xx, yy = np.meshgrid(x, y)

# tophat function again, but 2D
zz = np.heaviside(slit_width/2-np.abs(xx),0.5) * np.heaviside(slit_height/2-np.
↪abs(yy),0.5)

fig, ax0 = plt.subplots(1, 1, figsize=(6,6))

ax0.imshow(zz, origin='lower', extent=(xmin,xmax,ymin,ymax))

ax0.set_title(r'Real Space')
ax0.set_xlabel(r'$x$')
ax0.set_ylabel(r'$y$')
ax0.set_xlim(xmin, xmax)
ax0.set_ylim(ymin, ymax)

plt.show()
```

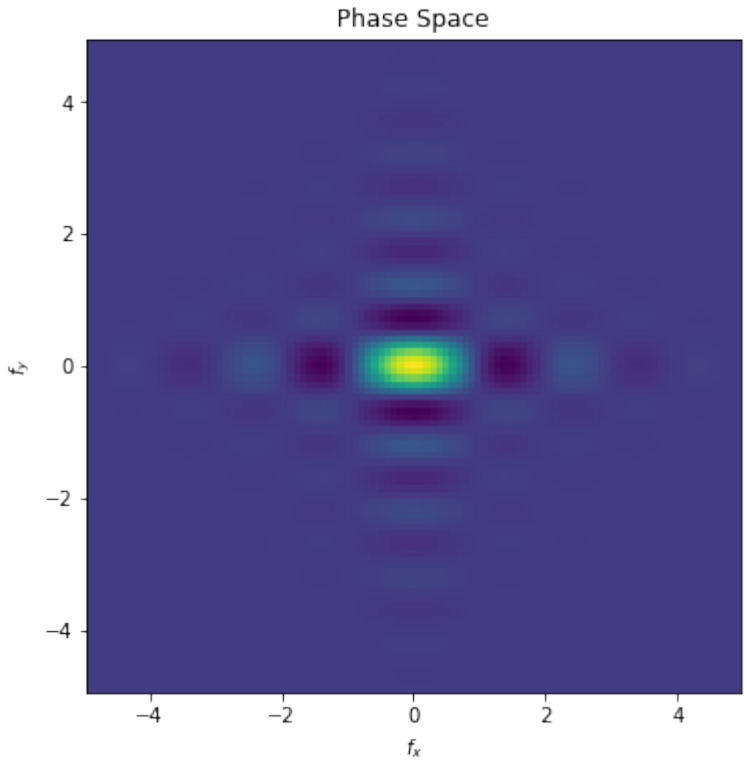


[5]:
```
zz_shift = np.fft.ifftshift(zz) # important! This centralizes the pattern

dx = x[1] - x[0] # sampling interval
dy = y[1] - y[0]

sp = np.fft.fftn(zz_shift) * dx * dy # need to multiply dx*dy as FT is integral not␣
↪summation
fx = np.fft.fftfreq(N, dx)
fy = np.fft.fftfreq(N, dy)

sp = np.fft.fftshift(sp) # necessary here if using imshow
fx = np.fft.fftshift(fx)
fy = np.fft.fftshift(fy)

fxmax = np.max(fx)
fxmin = np.min(fx)
fymax = np.max(fy)
fymin = np.min(fy)

# make lot

fig, ax0 = plt.subplots(1, 1, figsize=(6,6))

ax0.imshow(sp.real, origin='lower', extent=(fxmin,fxmax,fymin,fymax))

ax0.set_title(r'Phase Space')
ax0.set_xlabel(r'$f_x$')
ax0.set_ylabel(r'$f_y$')
ax0.set_xlim(fxmin, fxmax)
```

(continues on next page)

(continued from previous page)

```
ax0.set_ylim(fymin, fymax)

plt.show()
```



1.2.4 Application: Fraunhofer diffraction

Fraunhofer diffraction can be described as 2D FFT. Considering:

- Incoming plane wave λ
- Distance to image plane D
- Aperture shape $A(x', y')$

Then, the image pattern is

$$U(x, y) \propto \iint A(x', y') \exp\left[-\frac{2\pi i}{\lambda D}(xx' + yy')\right] dx'dy'$$

Therefore,

$$U(x, y) \propto \mathcal{F}\{A(x', y')\}\left(\frac{x}{\lambda D}, \frac{y}{\lambda D}\right)$$

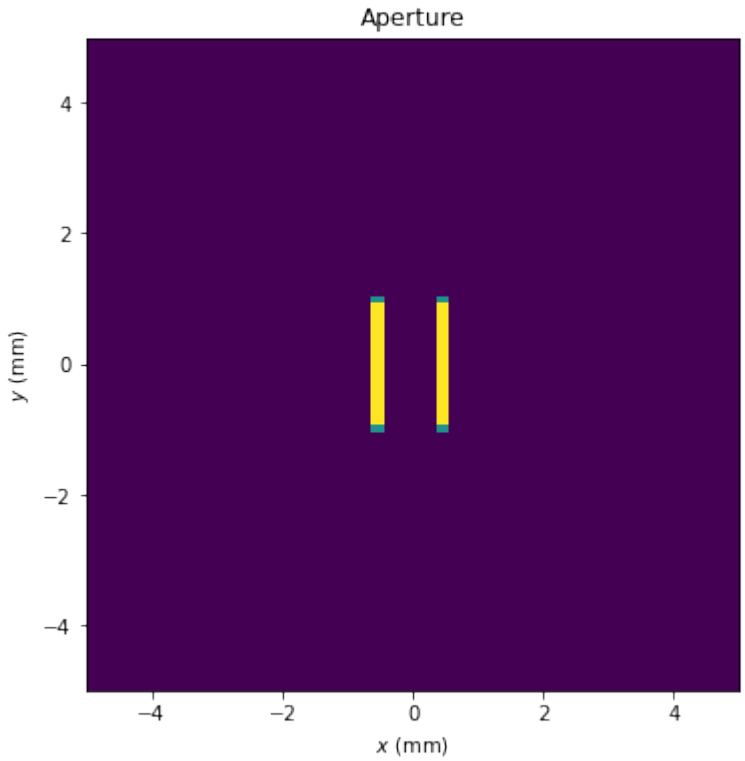
And the intensity is $I(x, y) \propto |U(x, y)|^2$.

Using the double-slit experiment as an example:

- $\lambda = 500$ nm (green-blue light)
- $D = 2$ m

- Slit size $= 2\ \text{mm} \times 0.2\ \text{mm}$
- Slit seperation $= 1\ \text{mm}$

[6]:
```
xmin = -5
xmax = 5
ymin = -5
ymax = 5
N = 100+1

# Aperture geometry, all units are mm
wavelength = 500e-6
D = 2e3
slit_width = 0.2
slit_height = 2
slit_seperation = 1

x = np.linspace(xmin, xmax, N)
y = np.linspace(ymin, ymax, N)
xx, yy = np.meshgrid(x, y)

# tophat function again, but 2D
zz = (
    np.heaviside(slit_width/2-np.abs(xx-slit_seperation/2),0.5) *
        np.heaviside(slit_height/2-np.abs(yy),0.5) +
    np.heaviside(slit_width/2-np.abs(xx+slit_seperation/2),0.5) *
        np.heaviside(slit_height/2-np.abs(yy),0.5))

fig, ax0 = plt.subplots(1, 1, figsize=(6,6))

ax0.imshow(zz, origin='lower', extent=(xmin,xmax,ymin,ymax))

ax0.set_title(r'Aperture')
ax0.set_xlabel(r'$x\ ({\rm mm})$')
ax0.set_ylabel(r'$y\ ({\rm mm})$')
ax0.set_xlim(xmin, xmax)
ax0.set_ylim(ymin, ymax)

plt.show()
```

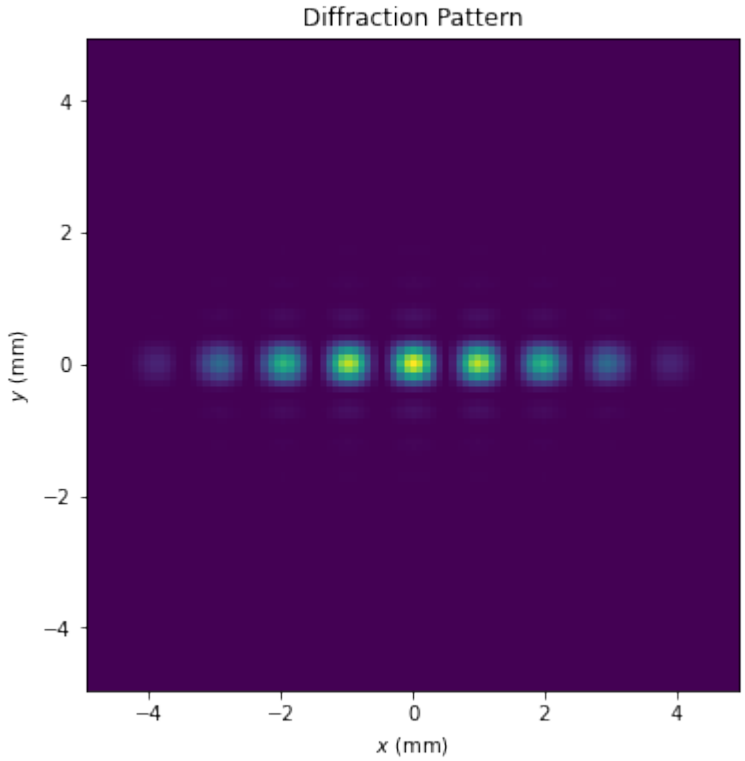


[7]:
```
zz_shift = np.fft.ifftshift(zz) # important! This centralizes the pattern

dx = x[1] - x[0] # sampling interval
dy = y[1] - y[0]

sp = np.fft.fftn(zz_shift) * dx * dy # need to multiply dx*dy as FT is integral not␣
↪summation
fx = np.fft.fftfreq(N, dx)
fy = np.fft.fftfreq(N, dy)

sp = np.fft.fftshift(sp) # necessary here if using imshow
fx = np.fft.fftshift(fx)
fy = np.fft.fftshift(fy)
x_image = fx * wavelength * D # in mm
y_image = fy * wavelength * D # in mm

x_image_max = np.max(x_image)
x_image_min = np.min(x_image)
y_image_max = np.max(y_image)
y_image_min = np.min(y_image)

# make lot

fig, ax0 = plt.subplots(1, 1, figsize=(6,6))

# plot the intensity
ax0.imshow(np.abs(sp)**2, origin='lower',
    extent=(x_image_min,x_image_max,y_image_min,y_image_max))
```

(continues on next page)

(continued from previous page)

```
ax0.set_title(r'Diffraction Pattern')
ax0.set_xlabel(r'$x\ ({\rm mm})$')
ax0.set_ylabel(r'$y\ ({\rm mm})$')
ax0.set_xlim(x_image_min, x_image_max)
ax0.set_ylim(y_image_min, y_image_max)

plt.show()
```



1.3 Linear fit using maximum likelihood

This notebook introduces how to perform linear fit with the maximum likelihood method.
Author: Bill Chen
Created: July 2022
Last modified: July 2022
License: MIT
Modules:

```
python = 3.8.3
numpy = 1.18.5
matplotlib = 3.4.3
scipy = 1.7.3
```

1.3.1 Linear fit with x error, y error, and instrinsic scatter

We write the model in the following form:

$$y = a + bx \pm \sigma.$$

We want to maximize the Gaussuan likelihood function:

$$\mathcal{L} = \prod_{i=1}^{N} \frac{1}{\sigma_i \sqrt{2\pi}} \exp\left(-\frac{\delta_i^2}{2\sigma_i^2}\right)$$

where $\delta_i = y - a - bx$ and $\sigma_i^2 = b\sigma_x^2 + \sigma_y^2 + \sigma^2$, σ_x and σ_y are the observational error of x and y. $\mathcal{L}$ is normally very small for large number of data. And, minimization can be more easily done with computers. So, we change the maximum likelihood problem to the manimum negative-log-likelihood problem:

$$-\log \mathcal{L} = \sum_{i=1}^{N} \left(-\frac{1}{2}\log(2\pi) + \frac{1}{2}\log\sigma_i + \frac{\delta_i^2}{2\sigma_i^2}\right) = \frac{1}{2}\sum_{i=1}^{N}\left(\log\sigma_i + \frac{\delta_i^2}{\sigma_i^2}\right) - \frac{N}{2}\log(2\pi)$$

That is, we only need to minimize

$$\sum_{i=1}^{N}\left(\log\sigma_i + \frac{\delta_i^2}{\sigma_i^2}\right)$$

[1]:
```
import numpy as np
from scipy.optimize import minimize

import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
import matplotlib.transforms as transforms

def neg_log_gaussian_likelihood(p, x, y, dx, dy):
    '''
    negative log of Gaussuan likelihood
    '''
    a = p[0]
    b = p[1]
    sig = p[2]
    sigi2 = sig**2 + dy**2 + (b*dx)**2
    return np.sum(np.log(sigi2) + (y-a-b*x)**2/sigi2)

def neg_log_ortho_likelihood(p, xb, yb, dxb, dyb):
    '''
    negative log of Orthogonal likelihood
    '''
    a = p[0]
    b = p[1]
    t = np.arctan(b)
    sig2 = p[2]**2*np.cos(t)**2
    D2 = ((yb - a)*np.cos(t) - xb*np.sin(t))**2
    S2 = dyb**2*np.cos(t)**2 + dxb**2*np.sin(t)**2
    return np.sum(np.log(S2+sig2)) + np.sum(D2/(S2+sig2))

def linear_fit_maxlike(x, y, dx, dy):
    bnds = ((None, None), (None, None), (0, None))
    o = minimize(neg_log_ortho_likelihood, [1.,0.,1.], args=(x,y,dx,dy), bounds=bnds)
```

(continues on next page)

(continued from previous page)

```
    return o.x

def linear_fit_maxlike_boot(x, y, dx, dy, size=100):
    data = np.column_stack((x, y, dx, dy))
    data_boot = data[np.random.randint(0, len(x), (size, len(x)))]

    a_list = np.zeros(size)
    b_list = np.zeros(size)
    sig_list = np.zeros(size)

    for i, d in enumerate(data_boot):
        bnds = ((None, None), (None, None), (0, None))
        o = minimize(neg_log_ortho_likelihood, [1.,0.,1.],
            args=(d[:,0],d[:,1],d[:,2],d[:,3]), bounds=bnds)
        a_list[i] = o.x[0]
        b_list[i] = o.x[1]
        sig_list[i] = o.x[2]

    return a_list, b_list, sig_list
```

[2]:
```
# generate test data

np.random.seed(0)

N_samp = 100

a_true = 0
b_true = 1
sig_true = 2
dx = 1*np.ones(N_samp) # observational error
dy = 1*np.ones(N_samp)

x = 5*np.random.normal(size=N_samp)
x = 30*np.random.rand(N_samp) - 15
y = a_true + b_true*x + sig_true*np.random.normal(size=N_samp)

# add observational error
x = x + dx*np.random.normal(size=N_samp)
y = y + dy*np.random.normal(size=N_samp)
```

[3]:
```
a, b, sig = linear_fit_maxlike_boot(x, y, dx, dy, 1000)
print('a_true = %.3f, a = %.3f +- %.3f'%(a_true, np.mean(a), np.std(a)))
print('b_true = %.3f, b = %.3f +- %.3f'%(b_true, np.mean(b), np.std(b)))
print('sig_true = %.3f, sig = %.3f +- %.3f'%(sig_true, np.mean(sig), np.std(sig)))
```

```
a_true = 0.000, a = -0.358 +- 0.260
b_true = 1.000, b = 1.033 +- 0.032
sig_true = 2.000, sig = 2.027 +- 0.224
```

[4]:
```
# visualization

def confidence_ellipse(ax, x, y, n_std=3.0, facecolor='none', **kwargs):

    if x.size != y.size:
        raise ValueError("x and y must be the same size")
```

(continues on next page)

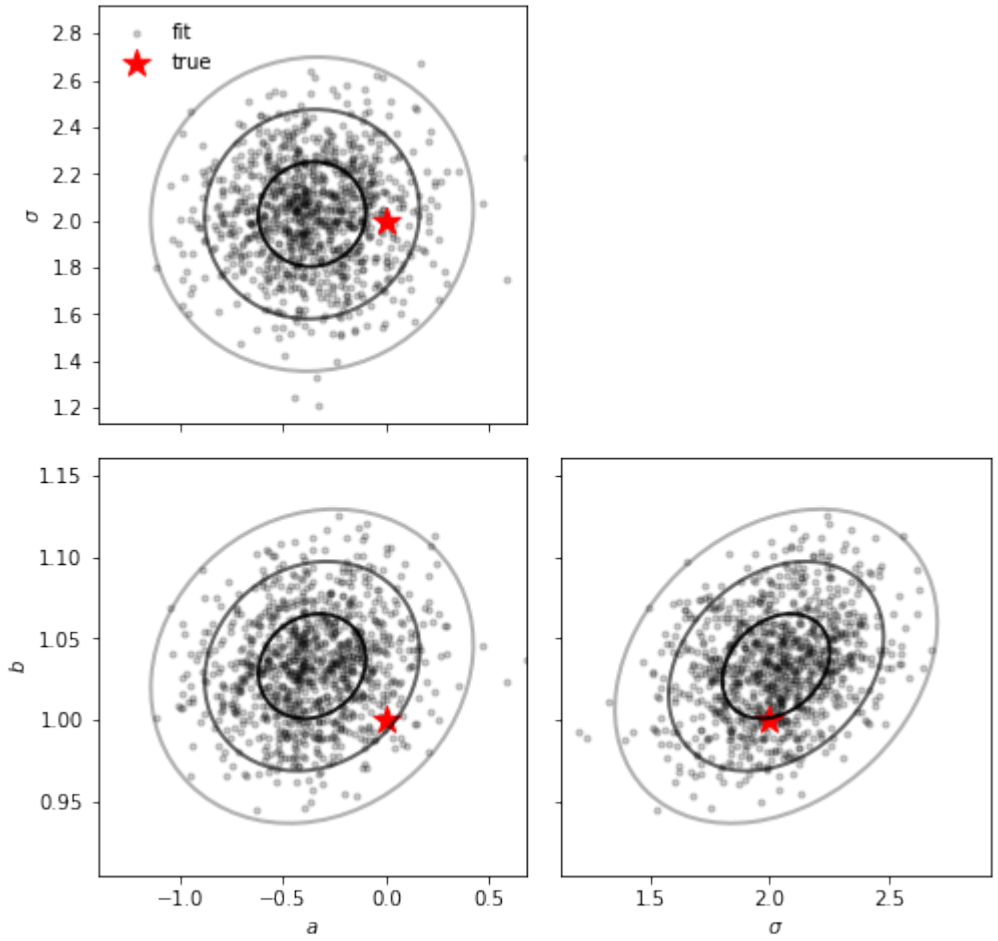
(continued from previous page)

```
    cov = np.cov(x, y)
    pearson = cov[0, 1]/np.sqrt(cov[0, 0] * cov[1, 1])
    # Using a special case to obtain the eigenvalues of this
    # two-dimensionl dataset.
    ell_radius_x = np.sqrt(1 + pearson)
    ell_radius_y = np.sqrt(1 - pearson)
    ellipse = Ellipse((0, 0), width=ell_radius_x * 2, height=ell_radius_y * 2,
                      facecolor=facecolor, **kwargs)

    # Calculating the stdandard deviation of x from
    # the squareroot of the variance and multiplying
    # with the given number of standard deviations.
    scale_x = np.sqrt(cov[0, 0]) * n_std
    mean_x = np.mean(x)

    # calculating the stdandard deviation of y ...
    scale_y = np.sqrt(cov[1, 1]) * n_std
    mean_y = np.mean(y)

    transf = transforms.Affine2D() \
        .rotate_deg(45) \
        .scale(scale_x, scale_y) \
        .translate(mean_x, mean_y)

    ellipse.set_transform(transf + ax.transData)
    return ax.add_patch(ellipse)

fig = plt.figure(figsize=(8,8))
ax0 = plt.subplot2grid((2, 2), (1, 0))
ax1 = plt.subplot2grid((2, 2), (0, 0))
ax2 = plt.subplot2grid((2, 2), (1, 1))
fig.subplots_adjust(hspace=0.08, wspace=0.08)

# bootstrap results
ax0.scatter(a, b, c='k', s=10, alpha=0.2)
confidence_ellipse(ax0, a, b, 1, ec='k', lw=2, alpha=0.9)
confidence_ellipse(ax0, a, b, 2, ec='k', lw=2, alpha=0.6)
confidence_ellipse(ax0, a, b, 3, ec='k', lw=2, alpha=0.3)
ax1.scatter(a, sig, c='k', s=10, alpha=0.2, label='fit')
confidence_ellipse(ax1, a, sig, 1, ec='k', lw=2, alpha=0.9)
confidence_ellipse(ax1, a, sig, 2, ec='k', lw=2, alpha=0.6)
confidence_ellipse(ax1, a, sig, 3, ec='k', lw=2, alpha=0.3)
ax2.scatter(sig, b, c='k', s=10, alpha=0.2)
confidence_ellipse(ax2, sig, b, 1, ec='k', lw=2, alpha=0.9)
confidence_ellipse(ax2, sig, b, 2, ec='k', lw=2, alpha=0.6)
confidence_ellipse(ax2, sig, b, 3, ec='k', lw=2, alpha=0.3)

# true values
ax0.scatter(a_true, b_true, c='r', marker='*', s=200, alpha=1)
ax1.scatter(a_true, sig_true, c='r', marker='*', s=200, alpha=1, label='true')
ax2.scatter(sig_true, b_true, c='r', marker='*', s=200, alpha=1)

# plot parameters
ax0.set_xlabel(r'$a$')
ax2.set_xlabel(r'$\sigma$')
ax0.set_ylabel(r'$b$')
ax1.set_ylabel(r'$\sigma$')
```

(continues on next page)

(continued from previous page)

```
ax0.set_xlim(np.mean(a)-4*np.std(a), np.mean(a)+4*np.std(a))
ax1.set_xlim(np.mean(a)-4*np.std(a), np.mean(a)+4*np.std(a))
ax1.set_xticklabels([])
ax0.set_ylim(np.mean(b)-4*np.std(b), np.mean(b)+4*np.std(b))
ax2.set_ylim(np.mean(b)-4*np.std(b), np.mean(b)+4*np.std(b))
ax2.set_yticklabels([])
ax2.set_xlim(np.mean(sig)-4*np.std(sig), np.mean(sig)+4*np.std(sig))
ax1.set_ylim(np.mean(sig)-4*np.std(sig), np.mean(sig)+4*np.std(sig))

ax1.legend(frameon=False, loc='upper left')
plt.show()
```



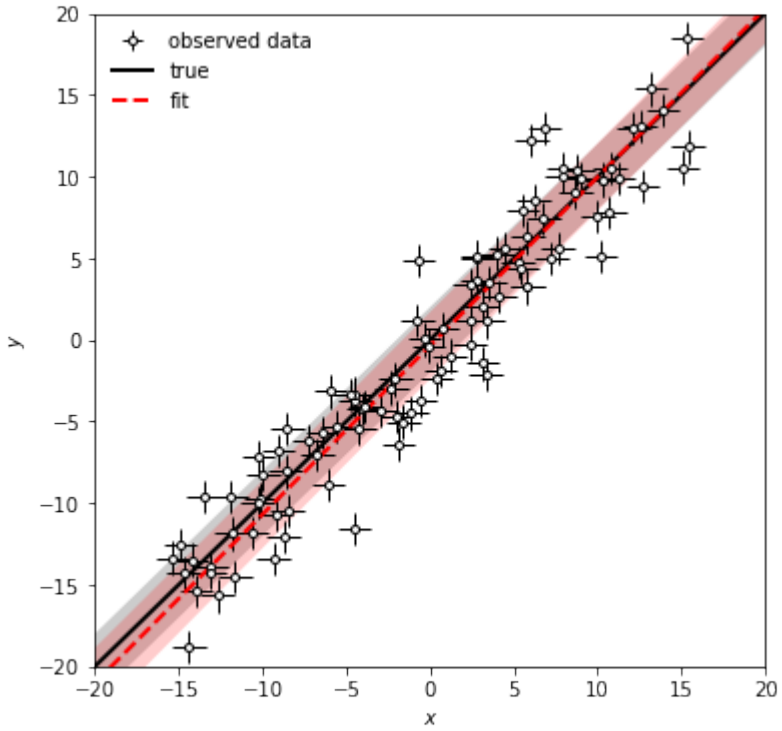
[5]:
```
def plot_linear_fit_with_error(ax, xmin, xmax, xmean, ymean, a, da, b, db, sig, dsig,␣
↪**kwargs):

    x = np.linspace(xmin, xmax, 100+1)
    y = a+b*(x-xmean)+ymean
    dy = np.sqrt(sig**2 + da**2 + (db*(x-xmean))**2)
    return ax.fill_between(x, y-dy, y+dy, **kwargs)

#     if (xmin < xmean) & (xmean < xmax):
#         ax.fill_between([xmin,xmean],
#             [(a+b*(xmin-xmean)-da-db*(xmean-xmin)-sig-dsig+ymean), (a-da-sig-
↪dsig+ymean)],
```

(continues on next page)

(continued from previous page)

```
#             [(a+b*(xmin-xmean)+da+db*(xmean-xmin)+sig+dsig+ymean),␣
↪(a+da+sig+dsig+ymean)],
#             **kwargs)
#         return ax.fill_between([xmean,xmax],
#             [(a-da-sig-dsig+ymean), (a+b*(xmax-xmean)-da-db*(xmax-xmean)-sig-
↪dsig+ymean)],
#             [(a+da+sig+dsig+ymean), (a+b*(xmax-xmean)+da+db*(xmax-
↪xmean)+sig+dsig+ymean)],
#             **kwargs)
#     else:
#         return ax.fill_between([xmin,xmax],
#             [(a+b*(xmin-xmean)-da-abs(db*(xmean-xmin))-sig-dsig+ymean),
#                 (a+b*(xmax-xmean)-da-abs(db*(xmax-xmean))-sig-dsig+ymean)],
#             [(a+b*(xmin-xmean)+da+abs(db*(xmean-xmin))+sig+dsig+ymean),
#                 (a+b*(xmax-xmean)+da+abs(db*(xmax-xmean))+sig+dsig+ymean)],
#             **kwargs)
```

[6]:
```
# visualization

fig, ax0 = plt.subplots(1, 1, figsize=(6,6), sharex=True, sharey=True)

low_limit= -20
high_limit= 20

lo = ax0.errorbar(x, y, dy, dx, c='k', ls='', lw=1, marker='o', ms=4, mfc='w', label=
↪'observed data')
lt, = ax0.plot([low_limit,high_limit], [a_true+b_true*low_limit,a_true+b_true*high_
↪limit],
    c='k', lw=2, label='true')
ax0.fill_between([low_limit,high_limit],
    [a_true+b_true*low_limit-sig_true,a_true+b_true*high_limit-sig_true],
    [a_true+b_true*low_limit+sig_true,a_true+b_true*high_limit+sig_true], fc='k', ec=
↪'none', alpha=0.2)
lf, = ax0.plot([low_limit,high_limit], [np.mean(a)+np.mean(b)*low_limit,np.mean(a)+np.
↪mean(b)*high_limit],
    c='r', lw=2, ls='--', label='fit')
plot_linear_fit_with_error(ax0, low_limit, high_limit, 0, 0,
    np.mean(a), np.std(a), np.mean(b), np.std(b), np.mean(sig), np.std(sig),
    fc='r', ec='none', alpha=0.2)

# plot parameters
ax0.set_xlabel(r'$x$')
ax0.set_ylabel(r'$y$')
ax0.set_xlim(low_limit, high_limit)
ax0.set_ylim(low_limit, high_limit)

ax0.legend(handles=[lo, lt, lf], frameon=False)
plt.show()
```


observed data
true
fit
20
15
10
5
0
−5
−10
−15
−20
y
−20
−15
−10
−5
0
5
10
15
20
x

CHAPTER

TWO

HOW TO CONTRIBUTE?

astro_py_scripts is deployed on GitHub. You are more than welcome to contribute your own script! Just email us your script, or raise an issue/pull request. We recommend contributors to follow the following format:

2.1 Format

We recommend you to put everythong in a single Jupyter Notebook if possible. However, if extra files are needed, you should put all files in one folder. Once being deployed, your notebook will be stored as:

```
docs/source/notebooks/<name_of_your_folder>/<name_of_your_notebook>.ipynb
```

Please also put it in a folder even if you only have one notebook. Some of our post-processing codes rely on such hierarchy.

It is super important to make sure everyone can run your notebook on their own devices! So, remember to restart your notebook before submitting it. Fix any bug if the notebook cannot run smoothly. Also, we recommend you to write down the versions of your Python and the modules you have imported at the begining of your notebook. If your script includes randomness, please also specify random seed!

2.2 License

Since **astro_py_scripts** is completely open source, you are recommended to have an open source license. Normally, the formal license file should be placed in the same folder as your notebook:

```
docs/source/notebooks/<name_of_your_folder>/LICENSE
```

And, you can also briefly mention your license information in the notebook. Write something like "The copyright of this notebook belongs to <your name> under the MIT license".

Tip: Please make sure your license is compatible with **astro_py_scripts**, which uses the MIT license. Clearly, a GPL license is not compatible. Click here for more information about licenses.

2.3 Contribute in a GitHub style

We maintain **astro_py_scripts** with GitHub Flow, which is a simple and effective workflow suitable for small projects like **astro_py_scripts**!

You can learn GitHub Flow with this video. It's a long video but don't be afraid! The standard GitHub Flow includes many subtle operations to avoid any potential conflict with other contributers. However, since our project is not that popular (at least for now), you are not likely to face a lot of annoying conflicts. You may alternatively want to watch this shoter video.