ASTR 5550: HW5

Jasmine Kobayashi

1. FFT Derivatives

Show that:

$$FT(df/dt) = i\omega\tilde{f}(\omega)$$

Assume that $f(t)$ is a *good* function.

(Fourier Transform)

$$FT(f(t)) = \tilde{f}(\omega) = \int_{-\infty}^{\infty} f(t)\, e^{-i\omega t}\, dt$$

$$FT\left(\frac{df}{dt}\right) = \int_{-\infty}^{\infty} \frac{df}{dt}\, e^{-i\omega t}\, dt$$

Integration by parts:

$$\int uv\, dx = u \int v\, dx - \int u' \left(\int v\, dx \right) dx$$

$v = \frac{d}{dt} f(t)$ $\qquad$ $u = e^{-i\omega t}$

$\Rightarrow \int v\, dx = \int_{-\infty}^{\infty} \frac{d}{dt} f(t)\, dt$ $\qquad$ $u' = \frac{d}{dt}\left(e^{-i\omega t}\right)$

$= [f(t)]_{-\infty}^{\infty}$ $\qquad$ $= (-i\omega) e^{-i\omega t}$

$$\Rightarrow FT\left(\frac{df}{dt}\right) = \int_{-\infty}^{\infty} \frac{df}{dt}\, e^{-i\omega t}\, dt$$

$$= \underbrace{[f(t)]_{-\infty}^{\infty}}\left(e^{-i\omega t}\right) - (-i\omega) \int_{-\infty}^{\infty} f(t)\, e^{-i\omega t}\, dt$$

$\rightarrow f(t) \equiv$ "good" function $\rightarrow \lim_{t \to \pm\infty} f(t) \to 0$

$$\Rightarrow FT\left(\frac{df}{dt}\right) = 0 - (-i\omega) \int_{-\infty}^{\infty} f(t) e^{-i\omega t}\, dt$$

$$= i\omega \underbrace{\int_{-\infty}^{\infty} f(t) e^{-i\omega t}\, dt}_{\tilde{f}(\omega)}$$

$$\boxed{= i\omega\, \tilde{f}(\omega)}$$

2. Convolution Theorem

Show that if:

$$f(t) = \int_{-\infty}^{\infty} g(t')h(t-t')dt'$$

Then:

$$\tilde{f}(\omega) = \tilde{g}(\omega)\tilde{h}(\omega)$$

$$FT(f(t)) = \int f(t)\, e^{-i\omega t}\, dt = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g(t')h(t-t')dt'\, e^{-i\omega t}\, dt$$

$$= \int_{-\infty}^{\infty} g(t') \int_{-\infty}^{\infty} h(t-t')\, e^{-i\omega t}\, dt\, dt'$$

$$u = t - t' \quad \rightarrow du = dt - 0 = dt$$

$$= \int_{-\infty}^{\infty} g(t') \int_{-\infty}^{\infty} h(u)\, e^{-i\omega(t'+u)}\, du\, dt'$$

$$= \underbrace{\int_{-\infty}^{\infty} g(t')\, e^{-i\omega t'}\, dt'}_{\tilde{g}(\omega)} \underbrace{\int_{-\infty}^{\infty} h(u)\, e^{-i\omega u}\, du}_{\tilde{h}(\omega)}$$

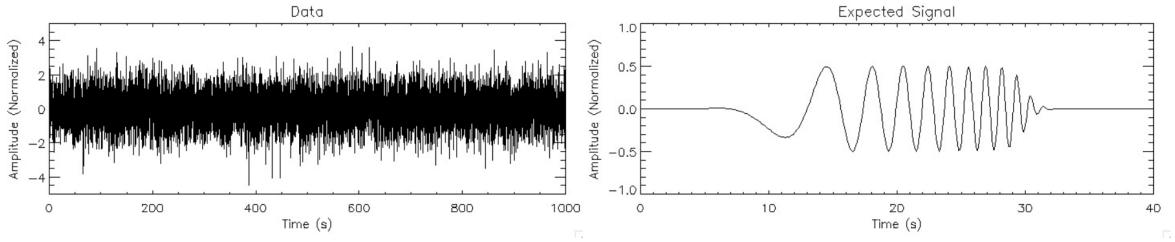
$$= \tilde{g}(\omega)\,\tilde{h}(\omega)$$

```
# Libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random
import math
import scipy
import os,sys
import seaborn as sns
sns.set_style('whitegrid')
```

```
# import helper script file
## change working directory
os.chdir("C:/Users/rokka/GH-repos/GitHubPages/Code-Reference-Notebook/CU-Boulder/AstroPhys/HW

## import my own code
import hw_helper_func2 as hf  # this is my own code I made (for probability/distribution fucr
```

3. Find the Signal

There have been many cases in which one must identify a signal buried in noise. It is easier than one thinks! LIGO, for example, offers an example in which distinct types of signals are expected. The data plotted below contains a signal embedded in random noise (fabricated). The data are ina file "`HW5_Data.txt`" (10000 values) and the signal is in the file "`HW5_Signal.txt`" (400 values).



Part (a)

Read in and verify signals. The cadence is 0.1 sec; generate time array for plotting. Let's start with the easiest method: cross-correlate the expected signal (400 points) with the data (1000 points). You will necessarily need to restrict your correlation to start at 20 sec and end at 980 sec to avoid edge effects. Plot the cross correlation as a function of time. Do you see a peak?

```
# read in data
data = np.loadtxt("hw5/HW5_Data.txt")
print("data shape:", data.shape)
data[:5]
```

```
data shape: (2000, 5)
```

```
array([[-1.4489671 , -0.05639172,  0.10236344,  0.03012188, -0.31627342],
       [-0.62731206,  1.5052398 ,  1.8104919 ,  0.41340446, -0.79464376],
       [ 0.88138539,  2.349612  , -0.44794255, -1.0353943 ,  0.92682785],
       [-0.07622421, -1.7109563 , -0.72154695,  1.3731886 ,  0.36365283],
       [-1.2074399 ,  0.32280344, -0.80500466,  1.4214896 ,  0.63947892]])
```

```
# Flatten 5d data to 1d
data_1d = data.flatten()
print(len(data_1d))
```

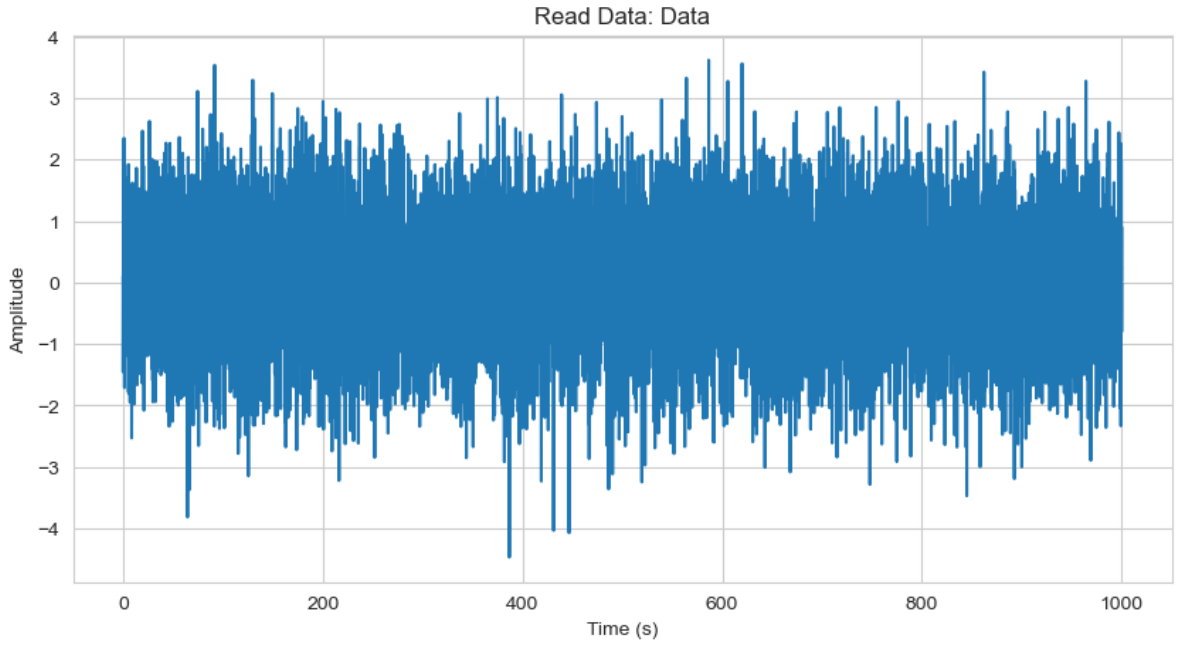
```
10000
```

```
t_data = np.linspace(0,1000,len(data_1d))
print(t_data)
print(len(t_data))
```

```
[0.00000000e+00 1.00010001e-01 2.00020002e-01 ... 9.99799980e+02
 9.99899990e+02 1.00000000e+03]
10000
```

```
plt.figure(figsize=(10,5))
plt.plot(t_data,data_1d)
plt.title("Read Data: Data")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
```

```
Text(0, 0.5, 'Amplitude')
```




```
# read in signal data
signal = np.loadtxt("hw5/HW5_Signal.txt")
print("signal data shape:", signal.shape)
signal[:5]
```

```
signal data shape: (80, 5)
```

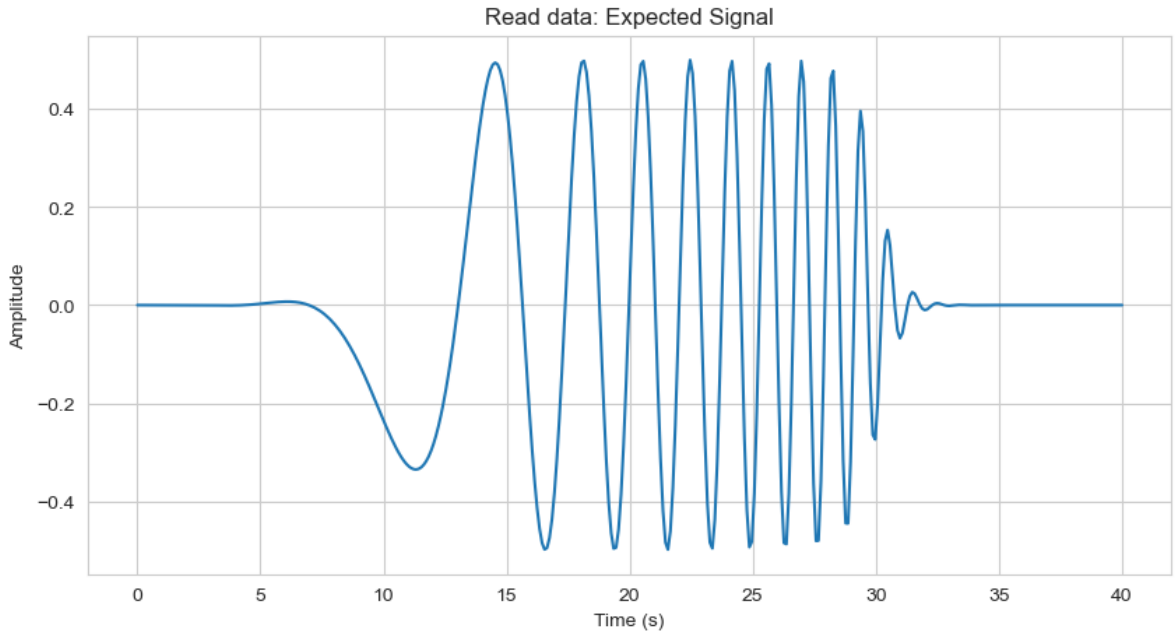
```
array([[-6.2465821e-06, -1.3047378e-05, -2.0128356e-05, -2.6956645e-05,
        -3.2930560e-05],
       [-3.7414051e-05, -3.9777609e-05, -3.9443527e-05, -3.5932957e-05,
        -2.8911940e-05],
       [-1.8233456e-05, -3.9726125e-06,  1.3547713e-05,  3.3743037e-05,
         5.5767006e-05],
       [ 7.8525801e-05,  1.0070760e-04,  1.2082675e-04,  1.3728147e-04,
         1.4842331e-04],
       [ 1.5263583e-04,  1.4841952e-04,  1.3447950e-04,  1.0981237e-04,
         7.3788494e-05]])
```

```
# Flatten to 1d
signal_1d = signal.flatten()
print(len(signal_1d))
```

400

```
t_signal = np.linspace(0,40,len(signal_1d))
plt.figure(figsize=(10,5))
plt.plot(t_signal,signal_1d)
plt.title("Read data: Expected Signal")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
```

```
Text(0, 0.5, 'Amplitude')
```



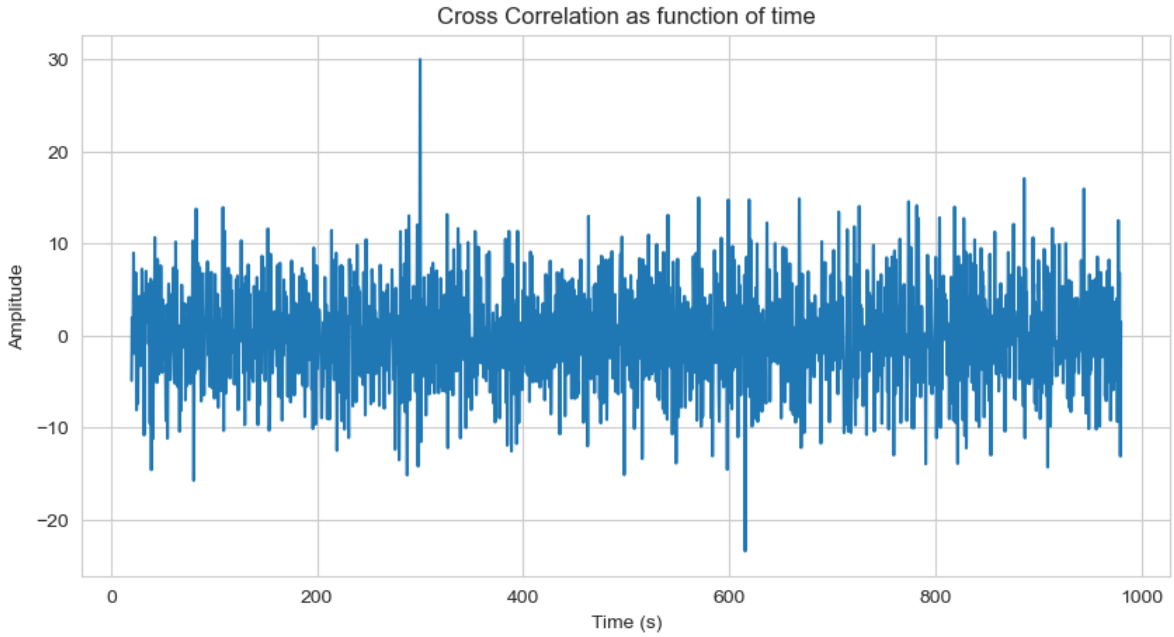

```
# cross correlate data + signal
cross_correlation = np.correlate(signal_1d, data_1d, mode='valid') # mode = 'valid' incorpora

print("length of cross-correlation:",len(cross_correlation))
```

```
length of cross-correlation: 9601
```

```
# time array for cross correlation
corr_t = np.linspace(20,980,len(cross_correlation))
```

```
plt.figure(figsize=(10,5))
plt.plot(corr_t,cross_correlation)
plt.title("Cross Correlation as function of time")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
```

```
Text(0, 0.5, 'Amplitude')
```



Part (b)

Isolate the maximum correlation value ($C_{\max}$) and then calculate the standard deviation (σ_C) of the 9600 correlation values. What is $C_{\max}/\sigma_C$? What is the probability of measuring such a correlation (or greater) given an Gaussian parent distribution?

Hint:

$$P = \frac{1}{2}\left[1 - \text{erf}\left(\frac{C_{\max}}{\sqrt{2}\sigma_C}\right)\right]; \text{ use double precision}$$

```
C_max = max(cross_correlation)          # maximum correlation value
print("Maximum correlation value (C_max):",C_max)

sigma_C = np.std(cross_correlation)     # standard deviation of correlation values
print("Standard Deviation of correlations (sigma_C):",sigma_C)
```

```
Maximum correlation value (C_max): 29.96898135397528
Standard Deviation of correlations (sigma_C): 4.792952440424132
```

```
print("C_max/sigma_C = ", C_max/sigma_C)
```

```
C_max/sigma_C =  6.2527182830387735
```

```
def corr_prob(c_max,std_c):
    inner = 1 - math.erf(c_max/(np.sqrt(2)*std_c))
    return inner/2

print("Probability of correlation (or greater) given Gaussian parent:", corr_prob(c_max=C_max
```

```
Probability of correlation (or greater) given Gaussian parent: 2.016846689656404e-10
```

Part (c)

At this point one may think that this event cannot possibly be a random fluctuation. However, how many 0.1 s periods are there in a year? There also are ~10^4 different variations in the expected signal. Given that information, how many random events would one expect in a year?

Jasmine's calculations

$$\text{period} = 0.1 \text{ s}$$
$$10 * \text{period} = 1 \text{ s}$$

There are 3.154×10^7 seconds in a year, meaning there are ~10^8 periods of 0.1 seconds in a year.

```
# TODO: Calculate number of random events in year
```

Part (d)

However, suppose that one made two such measurements (as did LIGO) and that the second data set had a correlation of 4.6 σ_C within the expected speed-of-light delay. Given the 2$^{\text{nd}}$ detection, what is the probability that these two measurements are from a random fluctuation?

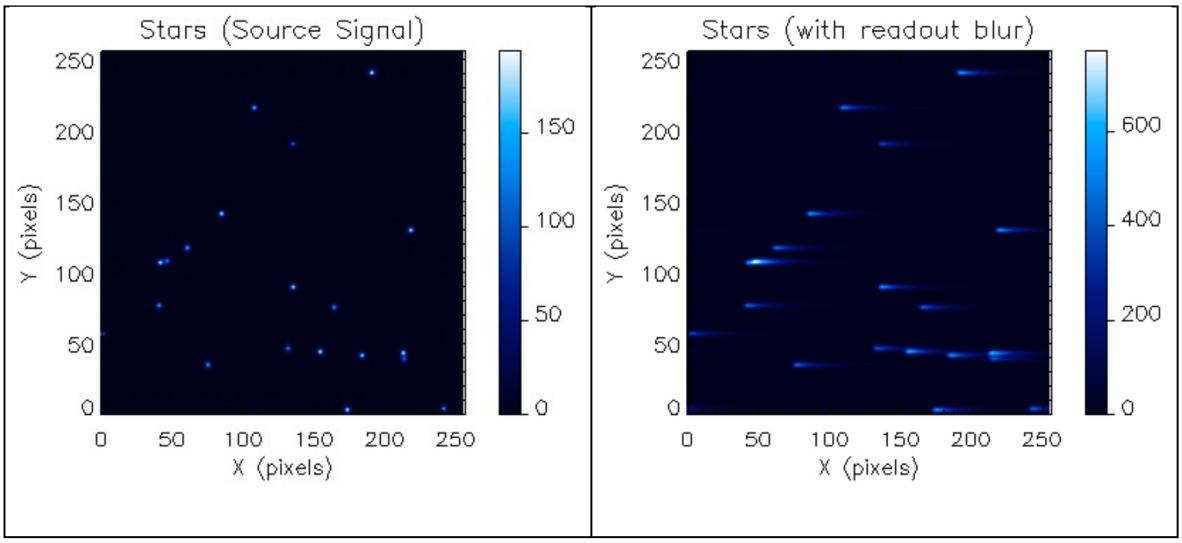
```
sigma2_C = 4.6
print("Probability of correlation (or greater) with second data:", corr_prob(c_max=C_max,std
```

```
Probability of correlation (or greater) with second data: 3.63458152463636e-11
```

4. Convolution Theorem

An older telescope has a horizontal CCD readout blur (see below). Unfortunately, the CCD readout has a memory of the previous pixel that exponentially fades causing horizontal streaks.

Examine the two images below.



Part (a)

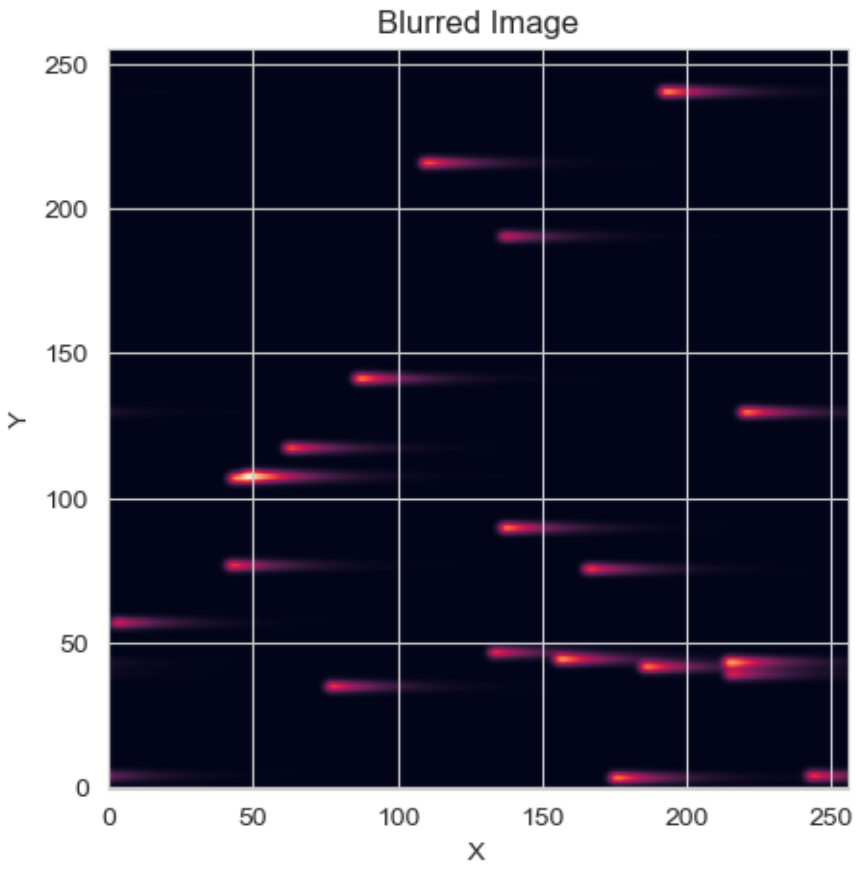
The blurred image is in a text file "`HW5_BlurredImage.txt`", which contains 256×256-point array (I_{Blur}). Read in this image and plot to verify

```
# Read BlurredImage text file (using numpy file version from Caroline)
I_blur = np.load("hw5/HW5_BlurredImage.npy")
I_blur.shape
```

```
(256, 256)
```

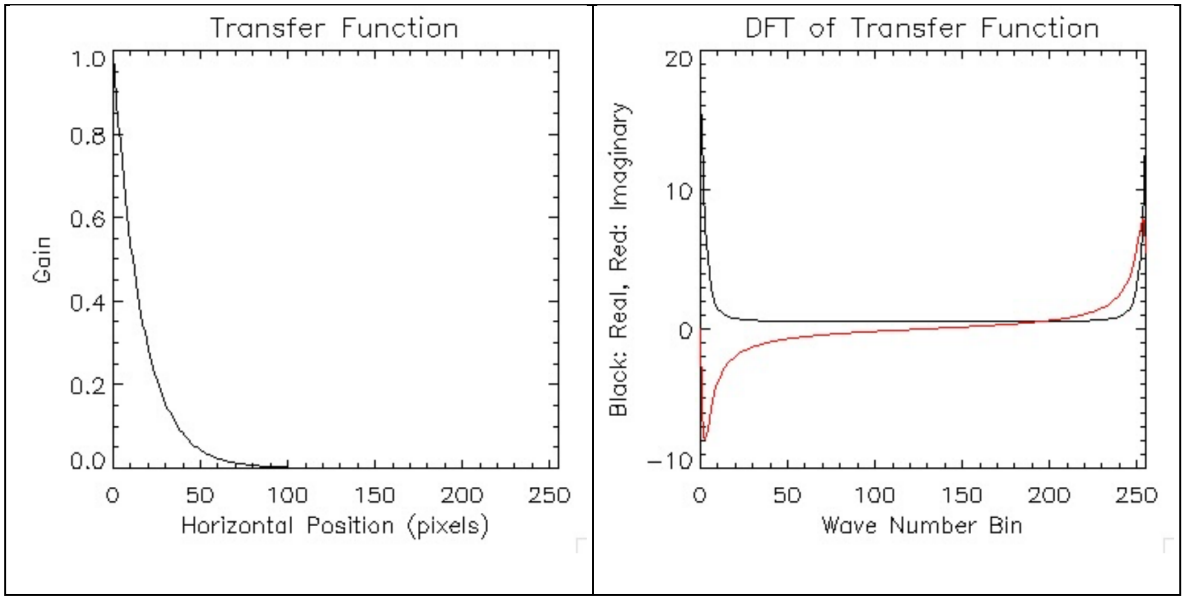
```
# plot BlurredImage data
plt.figure(figsize=(5,5))
plt.imshow(I_blur,origin='lower')
plt.title("Blurred Image")
plt.xlabel("X")
plt.ylabel("Y")
```

```
Text(0, 0.5, 'Y')
```



Part (b)

Fortunately, the transfer function is fairly well known. It is in the text file "`HW5_TransferFunc.txt`", which contains a 256-point array. The transfer function and it's DFT are shown below.



Part (c)

Read in and perform a DFT on the transfer function, $h(j)$. Plot your results and compare to above (and/or perform a reverse DFT to check). Pixel numbers are positive integers sot that one can calculate:

$$\tilde{h}(n) = \sum_{j=0}^{N-1} h(j) e^{-i2\pi nj/N}$$

IDL code:

```
jarr = [0,1,...255], npts=256
for n = 0, npts-1 do h_rl(n) = total(h*cos(jarr*n*2*!dpi/npts)) ; h is 256-point transfer fu
for n = 0, npts-1 do h_im(n) = -total(h*sin(jarr*n*2*!dpi/npts))
```

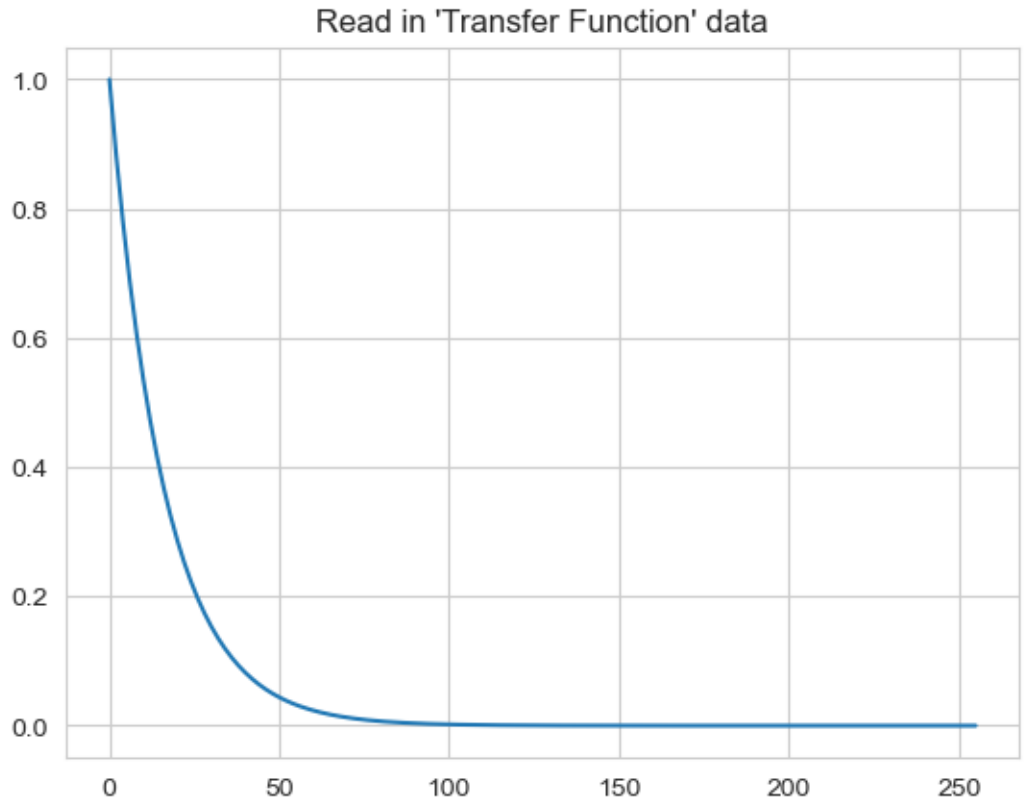
Note: One can use an FFT (with no window) if you prefer, but take care to understand how it works. I recommend doing a reverse FFT to check.

```
# read in transfer function (using numpy file version from Caroline)
transfer_function = np.load("hw5/HW5_TransferFunc.npy")
transfer_function.shape
```

```
(256,)
```

```
# Plot Tranfer Function data
j_arr = np.linspace(0,255,len(transfer_function))
plt.plot(j_arr,transfer_function)
plt.title("Read in 'Transfer Function' data")
```

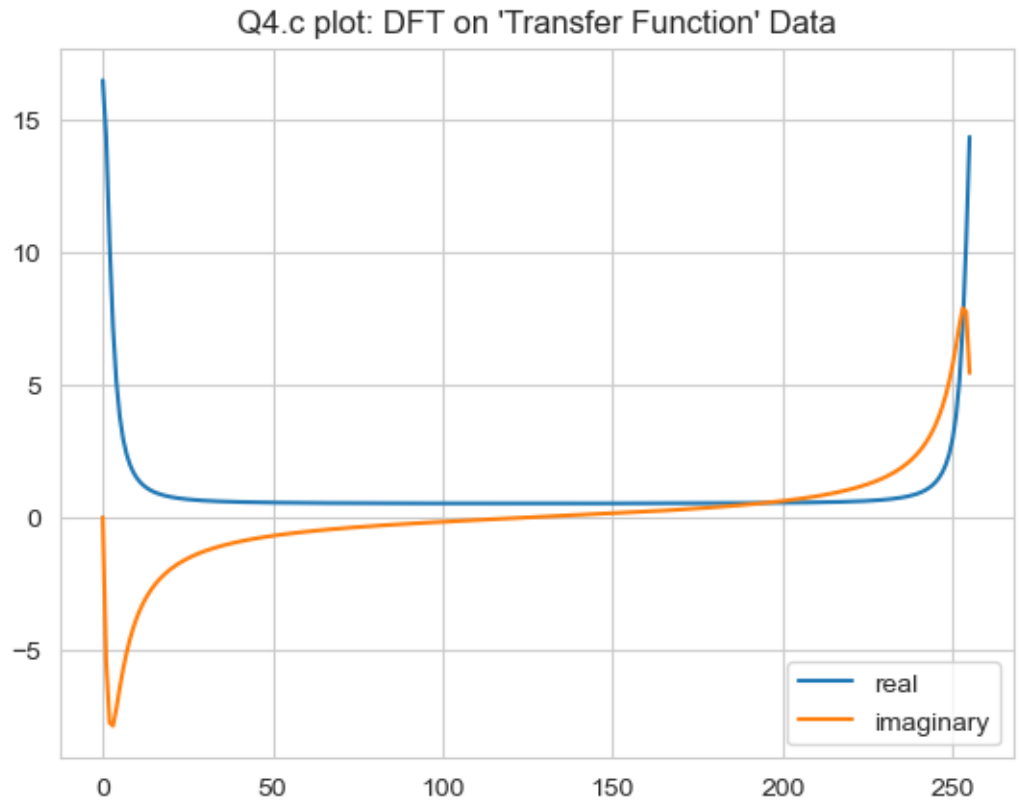
```
Text(0.5, 1.0, "Read in 'Transfer Function' data")
```




```
# calculate real and imaginary components of DFT
h_real = []
h_img = []
for n in j_arr:
    h_real.append(sum(transfer_function*np.cos(j_arr*n*np.pi*2/len(transfer_function))))
    h_img.append(-sum(transfer_function*np.sin(j_arr*n*np.pi*2/len(transfer_function))))
```

```
plt.plot(j_arr,h_real,label="real")
plt.plot(j_arr,h_img,label="imaginary")
plt.legend()
plt.title("Q4.c plot: DFT on 'Transfer Function' Data")
```

```
Text(0.5, 1.0, "Q4.c plot: DFT on 'Transfer Function' Data")
```




```python
import numpy as np

def dft(x,inverse=False):
    N = len(x)
    n = np.arange(N)
    k = n.reshape((N, 1))
    e_pow = -2j * np.pi * k * n / N
    if inverse:
        return np.dot(np.exp(-e_pow),x)/N
    return np.dot(np.exp(e_pow), x)

# Example usage
x = np.array([1, 2, 3, 4])
X = dft(x)
print(X)
X_inv = dft(X,inverse=True)
```

```
print(X_inv)
```

```
[10.+0.00000000e+00j -2.+2.00000000e+00j -2.-9.79717439e-16j
 -2.-2.00000000e+00j]
[1.-5.77996267e-16j 2.-2.60392190e-16j 3.-6.69535287e-17j
 4.+2.27765794e-16j]
```

```
I_blur
```

```
array([[4.3303071e+00, 4.0679471e+00, 3.8214826e+00, ..., 5.2233474e+00,
        4.9068808e+00, 4.6095879e+00],
       [2.4346232e+01, 2.2871169e+01, 2.1485475e+01, ..., 2.9367162e+01,
        2.7587896e+01, 2.5916430e+01],
       [8.3419708e+01, 7.8365564e+01, 7.3617634e+01, ..., 1.0062338e+02,
        9.4526914e+01, 8.8799817e+01],
       ...,
       [2.3808649e-16, 2.2366156e-16, 2.1011059e-16, ..., 2.8718712e-16,
        2.6978733e-16, 2.5344175e-16],
       [3.7566471e-19, 3.5290434e-19, 3.3152294e-19, ..., 4.5313814e-19,
        4.2568389e-19, 3.9989300e-19],
       [0.0000000e+00, 0.0000000e+00, 0.0000000e+00, ..., 0.0000000e+00,
        0.0000000e+00, 0.0000000e+00]])
```

Part (d)

Perform a DFT on the blurred image line by line in y. Be careful here. One must perform 256 DFTs; one for each position in y. I recommend finding a y value a star and testing your code.

$$\tilde{I}_{\text{Blur}}(n, y) = \sum_{j=0}^{N-1} I_{\text{Blur}}(j, y) e^{-i2\pi nj/N}$$

```
cnt = 0
for i in I_blur:
    print(len(i))
    cnt +=1
```

256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256

256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256

256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256

256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256

256
256
256

256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256
256

```
cnt
```

256

```
#TODO: Perform DFT on Blurred Image
# (writing own DFT should only take like two lines of code, supposedly)
```

```
# DFT using scipy
bi_fft = np.fft.rfft2(I_blur)
bi_fft
```

```
array([[ 6.86942762e+05+0.00000000e+00j, -3.18333759e+04+8.28437283e+04j,
        -9.08783496e+04+2.48929786e+04j, ...,
         8.31991046e+01-4.89622549e+00j,  8.26230421e+01-2.65030048e+00j,
         8.28029657e+01+0.00000000e+00j],
       [ 2.65009434e+04-2.58621884e+05j,  7.34812945e+04+1.42624552e+05j,
        -2.09239918e+04+9.66140669e+04j, ...,
         9.28511617e+00-8.26591150e+01j,  1.16847810e+01-8.15848490e+01j,
         1.45882393e+01-8.15777954e+01j],
       [ 1.02042383e+04-2.14384200e+04j,  1.68452575e+05-1.02903199e+05j,
         2.38241278e+04+7.50324914e+04j, ...,
        -7.91777867e+01-2.29869470e+01j, -7.84330896e+01-2.59113380e+01j,
        -7.77865886e+01-2.80957094e+01j],
       ...,
       [-1.59312172e+05+1.76230939e+04j,  9.34893036e+04-2.20118019e+04j,
        -6.46289744e+03-1.67760129e+05j, ...,
        -4.52632218e+01-6.90957709e+01j, -4.28212633e+01-7.02563246e+01j,
        -4.07922886e+01-7.11551093e+01j],
       [ 1.02042383e+04+2.14384200e+04j, -3.26547893e+04-2.08000166e+04j,
        -1.38723370e+05+1.55334067e+05j, ...,
        -7.59058811e+01+3.27326150e+01j, -7.64983144e+01+3.02890809e+01j,
        -7.77865886e+01+2.80957094e+01j],
       [ 2.65009434e+04+2.58621884e+05j, -1.75208878e+02+2.26770637e+05j,
         7.49787321e+04-7.74987310e+04j, ...,
         1.90865340e+01+8.04520931e+01j,  1.67813716e+01+8.11626117e+01j,
         1.45882393e+01+8.15777954e+01j]])
```

```
x = bi_fft.real
y = bi_fft.imag
```

```
x.shape
```

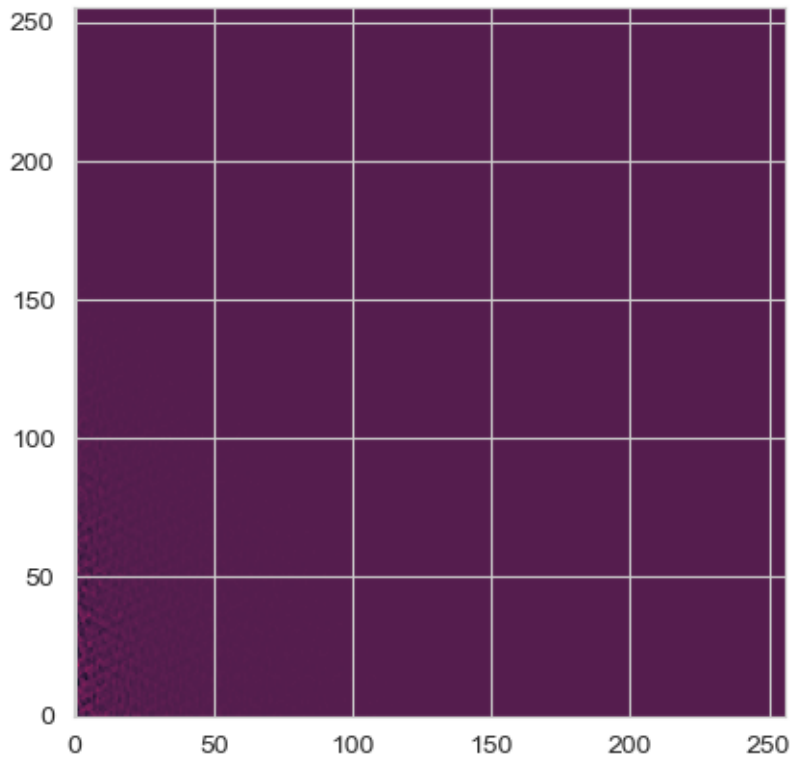
```
(256, 129)
```

```
y.shape
```

```
(256, 129)
```

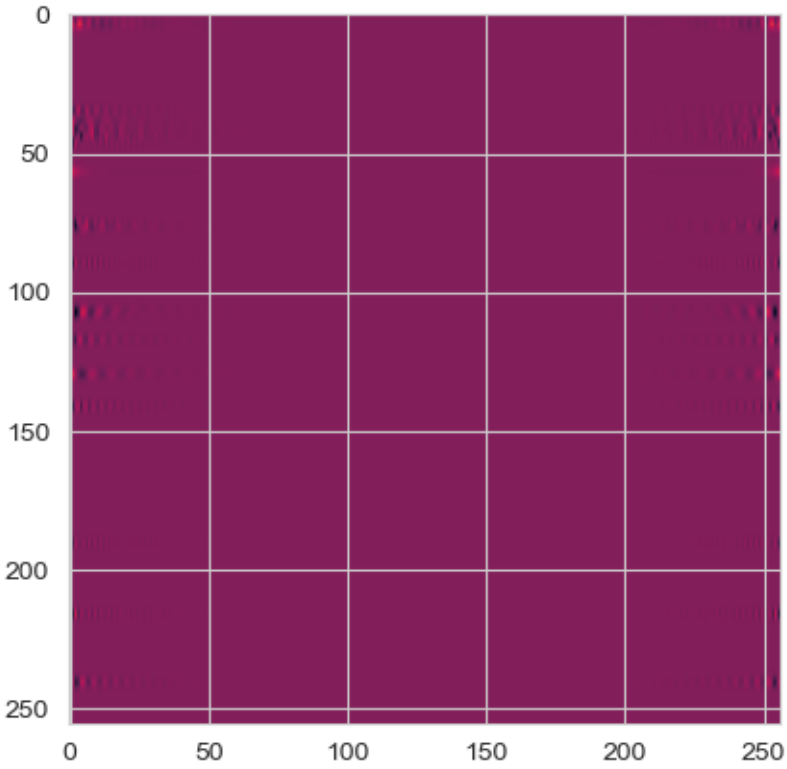
```
# DFT using scipy
bi_fft = scipy.fft.dctn(I_blur,norm='forward',type=2)
print(bi_fft.shape)
plt.imshow(bi_fft,origin='lower')
```

```
(256, 256)
```

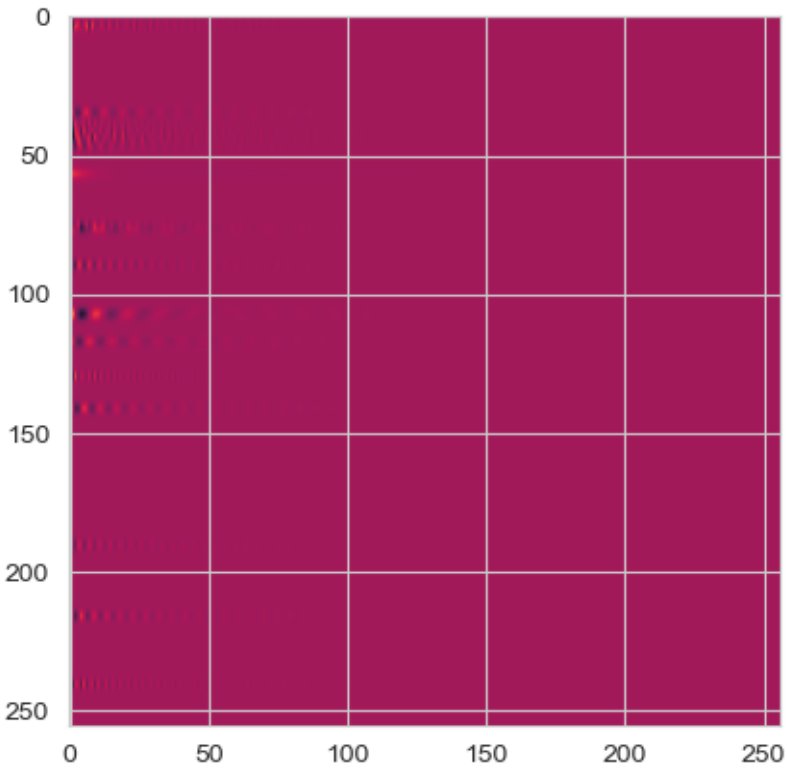


```
bi_fft = []
for i in I_blur:
    bi_fft.append(scipy.fft.fft(i))
```

```
plt.imshow(np.array(bi_fft).real)
```




```
plt.imshow(bi_fft)
```



Part (e)

Carefully calculate (watch out for signs!) line by line in y:

$$\tilde{I}_{\text{Source}}(n, y) = \frac{\tilde{I}_{\text{Blur}}(n, y)\tilde{h}^*(n)}{|\tilde{h}(n)|^2}$$

Part (f)

Perform a reverse DFT on $\tilde{I}_{Source}$. Plot your results. The image should be close but not exactly equal to the source image plotted above.

Hint: The most common error is in reconstructing $\tilde{I}_{\text{Source}}$. Check your reverse DFT by deriving h from $\tilde{h}$.

```
#TODO: Reverse DFT
```

```
#TODO: Plot results
```