hw08

November 16, 2021

1 Homework 08

Nathan LeRoy

[2]:
```
# better image quality
import matplotlib as mpl
%matplotlib inline
mpl.rcParams['figure.dpi'] = 300
```

1.1 Problem 1

1.1.1 a.)

We can start by writing out the first few terms of the definition of the discrete Fourrier Transform.

$$x[k] = \sum_{n=0}^{N-1} x[n]e^{-i\frac{2\pi}{N}kn}, k = 1, 2, 3, \ldots, N-1$$

Examining for $k = 0$

$$x[0] = x[0]e^{-i\frac{2\pi}{N}(0)(0)} + x[1]e^{-i\frac{2\pi}{N}(0)(0)} + \cdots + x[N-1]e^{-i\frac{2\pi}{N}(0)(0)}$$

$$X[0] = x[0] + x[1] + \cdots + x[N-1]$$

Examining for $k = 1$

$$X[1] = x[0]e^{-i\frac{2\pi}{N}(1)(0)} + x[1]e^{-i\frac{2\pi}{N}(1)(1)} + \cdots + x[N-1]e^{-i\frac{2\pi}{N}(1)(N-1)}$$

$$X[1] = x[0] + x[1]e^{-i\frac{2\pi}{N}(1)(1)} + \cdots + x[N-1]e^{-i\frac{2\pi}{N}(1)(N-1)}$$

Examining for $k = 2$

$$X[2] = x[0]e^{-i\frac{2\pi}{N}(2)(0)} + x[1]e^{-i\frac{2\pi}{N}(2)(1)} + \cdots + x[N-1]e^{-i\frac{2\pi}{N}(2)(N-1)}$$

$$X[2] = x[0] + x[1]e^{-i\frac{2\pi}{N}(2)(1)} + \cdots + x[N-1]e^{-i\frac{2\pi}{N}(2)(N-1)}$$

We can see a pattern emerging. If we write the kernal of the Fourrier transform as $\psi = e^{-i\frac{2\pi}{N}}$, then we can simplify the above calculations to matrix form:

$$X = Fx$$

Where,

$$F = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \phi^1 & \phi^2 & \dots & \phi^{(N-1)} \\ 1 & \phi^2 & \phi^4 & \dots & \phi^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \phi^{(N-1)} & \phi^{2(N-1)} & \dots & \phi^{(N-1)(N-1)} \end{bmatrix}$$

1.1.2 b.)

Employing a very similar strategy to the part a, we can start to build up a basis for the G matrix:

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]e^{i\frac{2\pi}{N}kn}, n = 1, 2, 3, \dots, N-1$$

Examining for $n = 0$

$$x[0] = \frac{1}{N}\left(X[0]e^{i\frac{2\pi}{N}(0)(0)} + X[1]e^{i\frac{2\pi}{N}(1)(0)} + X[2]e^{i\frac{2\pi}{N}(2)(0)} + \cdots + X[0]e^{i\frac{2\pi}{N}(N-1)(0)}\right)$$

$$x[0] = \frac{1}{N}\left(X[0] + X[1] + \cdots + X[N-1]\right)$$

Examining for $n = 1$

$$x[1] = \frac{1}{N}\left(X[0]e^{i\frac{2\pi}{N}(0)(1)} + X[1]e^{i\frac{2\pi}{N}(1)(1)} + X[2]e^{i\frac{2\pi}{N}(2)(1)} + \cdots + X[0]e^{i\frac{2\pi}{N}(N-1)(1)}\right)$$

$$x[1] = \frac{1}{N}\left(X[0] + X[1]e^{i\frac{2\pi}{N}(1)(1)} + \cdots + X[N-1]e^{i\frac{2\pi}{N}(N-1)(1)}\right)$$

Examining for $n = 2$

$$x[2] = \frac{1}{N}\left(X[0]e^{i\frac{2\pi}{N}(0)(2)} + X[1]e^{i\frac{2\pi}{N}(1)(2)} + X[2]e^{i\frac{2\pi}{N}(2)(2)} + \cdots + X[0]e^{i\frac{2\pi}{N}(N-1)(2)}\right)$$

$$x[2] = \frac{1}{N}\left(X[0] + X[1]e^{i\frac{2\pi}{N}(1)(2)} + \cdots + X[N-1]e^{i\frac{2\pi}{N}(N-1)(2)}\right)$$

Again we see a pattern emerging. If we write the kernal of the inverse Fourrier transform as $\omega = \frac{e^{i\frac{2\pi}{N}}}{N}$, we can simplify the matrix equation:

$$x = GX$$

where

$$G = \begin{bmatrix} \frac{1}{N} & \frac{1}{N} & \frac{1}{N} & \cdots & \frac{1}{N} \\ \frac{1}{N} & \omega^1 & \omega^2 & \cdots & \omega^{(N-1)} \\ \frac{1}{N} & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{N} & \omega^{(N-1)} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

1.1.3 c.)

We can demonstrate that $FG = I$ by multiplying the two simplified matrices together:

$$FG = \begin{bmatrix} \frac{1}{N} & \frac{1}{N} & \frac{1}{N} & \cdots & \frac{1}{N} \\ \frac{1}{N} & \phi^1 & \phi^2 & \cdots & \phi^{(N-1)} \\ \frac{1}{N} & \phi^2 & \phi^4 & \cdots & \phi^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{N} & \phi^{(N-1)} & \phi^{2(N-1)} & \cdots & \phi^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} \frac{1}{N} & \frac{1}{N} & \frac{1}{N} & \cdots & \frac{1}{N} \\ \frac{1}{N} & \omega^1 & \omega^2 & \cdots & \omega^{(N-1)} \\ \frac{1}{N} & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{N} & \omega^{(N-1)} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

Which can be simplified to the following expression(s):

$$\sum_{n=0}^{N-1} F_{l,n}G_{n,m}$$

Where l and m are lth row and mth column of the matrices.

$$\sum_{n=0}^{N-1} \phi^{ln}\omega^{nm}$$

$$\frac{1}{N}\sum_{n=0}^{N-1} e^{-i\frac{2\pi}{N}ln} e^{i\frac{2\pi}{N}nm}$$

Which is precisely the equation given.

1.2 Problem 2

Load the data:

[3]:
```
from scipy.io import loadmat
ftf = loadmat("data/hw8prob2data.mat")['ftf']
```

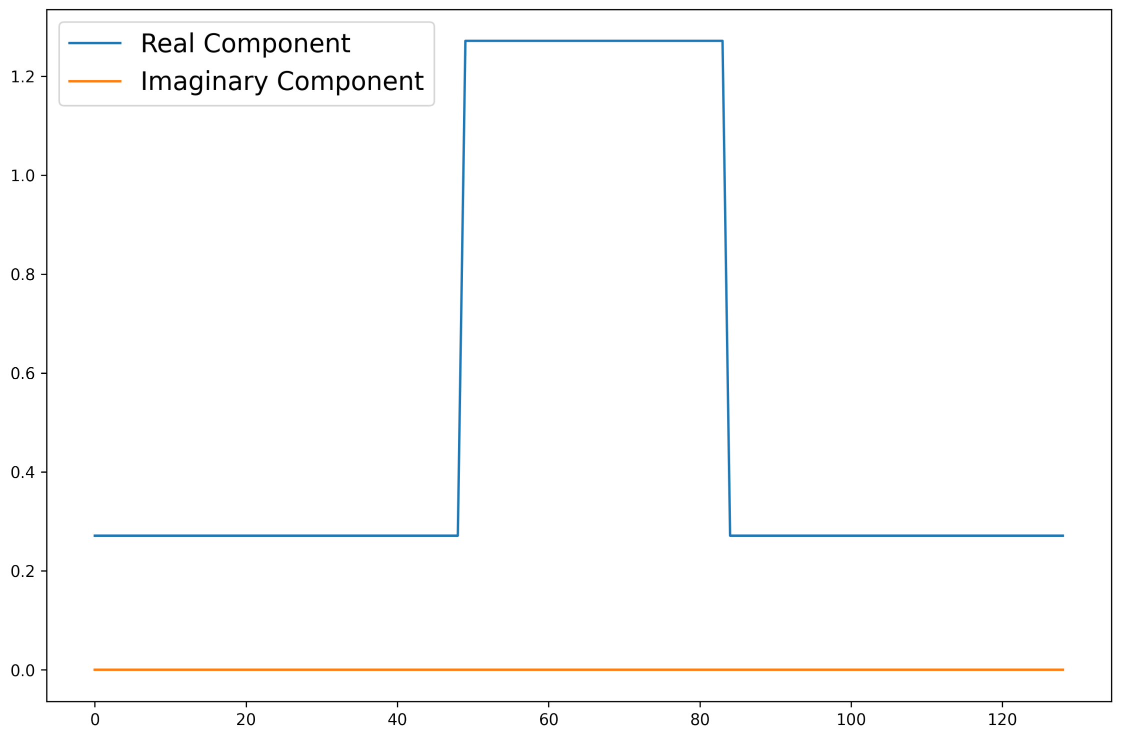
We can bring in the functions given for the homework to accomplish this problem. Since the output of a discrete Fourrier Transform is, by definition, mirrored if the input signal is **real-valued**, we know that at the midpoint/nyquist frequency, the output signal in the frequency domain will be mirrored along the complex components byway of their complex conjugates. That is, the second half will be the first half's real components with their corresponding complex conjugates.

Thus we can implement the following steps to reconstruct our function data:

1. Reverse/Flip the matrix/array.
2. Calculate the complex conjugate of these new points.
3. Substitute in the new calculated points for the second half of the old array from step 1.

This is implemented below:

[4]:
```
# import given helpers
from helpers import *

# utilize functions to
ftf_rev = np.flip(ftf) # flip matrix
ftf_rev_conj = np.conj(ftf_rev) # find complex conjugate
ftf_full = ftf + ftf_rev_conj # replace

data = ift(ftf_full)
pc(data.T)
```



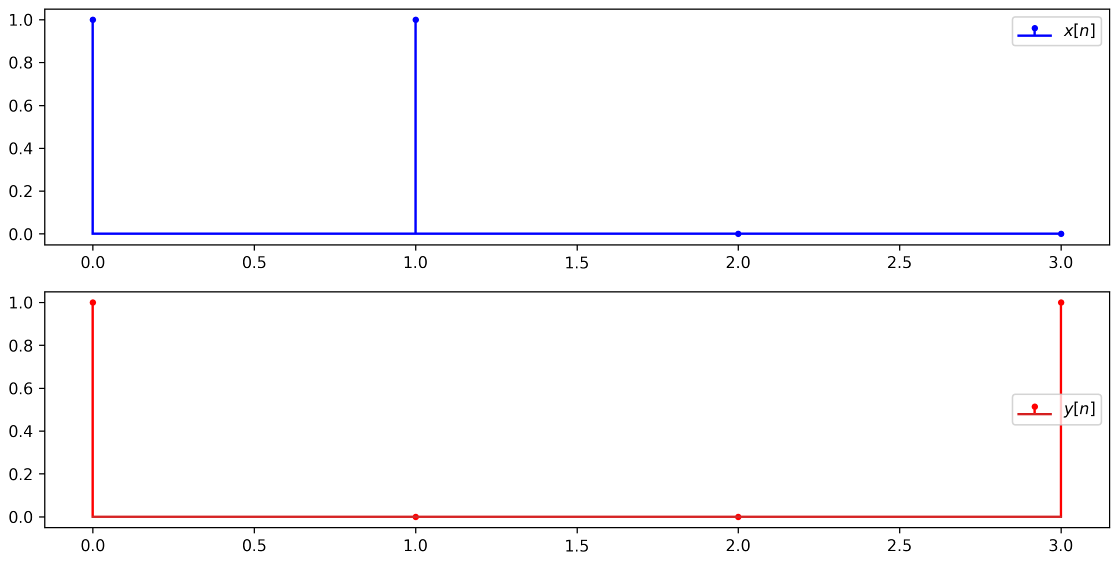
1.3 Problem 3

1.3.1 a.)

Plots of $x[n]$ and $y[n]$:

[5]:
```
import numpy as np
import matplotlib.pyplot as plt
x = np.array([1,1,0,0])
y = np.array([1,0,0,1])

plt.rcParams["axes.labelweight"] = "bold"
fig, ax = plt.subplots(2, figsize=(12,6))
ax[0].stem(
    x,
    linefmt="b-",
    markerfmt="b.",
    basefmt="b",
    label="$x[n]$"
)
ax[0].legend()
ax[1].stem(
    y,
    linefmt="r-",
    markerfmt="r.",
    label="$y[n]$"
)
ax[1].legend()
```

```
[5]: <matplotlib.legend.Legend at 0x17f7d3b50>
```



We can quickly see that the convolution will simply be:

$x[n] * y[n] = [1, 1, 0, 1, 1, 0, 0]$

1.3.2 b.)

When finding the DFT of $x[n]$ we can ignore the summation terms for $n = 2$ and $n = 3$ as we know that $x[2]$ and $x[3]$ are both equal to zero:

$$\begin{aligned}
X[0] &= x[0]e^{-i\frac{2\pi}{4}(0)(0)} + x[1]e^{-i\frac{2\pi}{4}(0)(1)} = 1 + 1 = 2 \\
X[1] &= x[0]e^{-i\frac{2\pi}{4}(1)(0)} + x[1]e^{-i\frac{2\pi}{4}(1)(1)} = 1 + e^{-i\frac{\pi}{2}} \\
X[2] &= x[0]e^{-i\frac{2\pi}{4}(2)(0)} + x[1]e^{-i\frac{2\pi}{4}(2)(1)} = 1 + e^{-i\pi} = 1 + (-1) = 0 \\
X[3] &= x[0]e^{-i\frac{2\pi}{4}(3)(0)} + x[1]e^{-i\frac{2\pi}{4}(3)(1)} = 1 + e^{-i\frac{3\pi}{2}}
\end{aligned}$$

Thus,

$X[k] = [2, 1 - i, 0, 1 + i]$.

We can do something similar for $Y[k]$, however this time we need not consider $y[1]$ and $y[2]$ as they are both equal to 0.

$$Y[0] = y[0]e^{-i\frac{2\pi}{4}(0)(0)} + y[3]e^{-i\frac{2\pi}{4}(0)(3)} = 1 + 1 = 2$$
$$Y[1] = y[0]e^{-i\frac{2\pi}{4}(1)(0)} + y[3]e^{-i\frac{2\pi}{4}(1)(3)} = 1 + e^{-i\frac{3\pi}{2}}$$
$$Y[2] = y[0]e^{-i\frac{2\pi}{4}(2)(0)} + y[3]e^{-i\frac{2\pi}{4}(2)(3)} = 1 + e^{-i3\pi} = 1 + (-1) = 0$$
$$Y[3] = Y[1] = y[0]e^{-i\frac{2\pi}{4}(3)(0)} + y[3]e^{-i\frac{2\pi}{4}(3)(3)} = 1 + e^{-i\frac{9\pi}{2}}$$

Thus,

$Y[k] = [2, 1 + i, 0, 1 - i]$

1.3.3 c.)

Using this information, we can find the circular convolution by finding $Z[k]$ by multiplying $X[k]Y[k]$ and taking the inverse DTF of $Z[k]$.

$Z[k] = [2 \cdot 2, (1 - i)(1 + i), 0 \cdot 0, (1 + i)(1 - i)]$

$Z[k] = [4, 2, 0, 2]$

We can then find $z[n]$, the circular convolution of $x[n]$ and $y[n]$ by taking the inverse convolution of $Z[k]$:

$$z[n] = \frac{1}{N}\sum_{k=0}^{N-1} Z[k]e^{i\frac{2\pi}{N}nk}$$

We need not consider the term for $k = 2$ as $Z[2]$ is equal to 0:

$$z[0] = \frac{1}{4}\left(Z[0]e^{i\frac{2\pi}{4}(0)(0)} + Z[1]e^{i\frac{2\pi}{4}(0)(1)} + Z[3]e^{i\frac{2\pi}{4}(0)(3)}\right) = \frac{1}{4}(4 + 2 + 2) = 2$$

$$z[1] = \frac{1}{4}\left(Z[0]e^{i\frac{2\pi}{4}(1)(0)} + Z[1]e^{i\frac{2\pi}{4}(1)(1)} + Z[3]e^{i\frac{2\pi}{4}(1)(3)}\right) = \frac{1}{4}(4 + 2i - 2i) = 1$$

$$z[2] = \frac{1}{4}\left(Z[0]e^{i\frac{2\pi}{4}(2)(0)} + Z[1]e^{i\frac{2\pi}{4}(2)(1)} + Z[3]e^{i\frac{2\pi}{4}(2)(3)}\right) = \frac{1}{4}(4 + 2(-1) + 2(-1)) = 0$$

$$z[3] = \frac{1}{4}\left(Z[0]e^{i\frac{2\pi}{4}(3)(0)} + Z[1]e^{i\frac{2\pi}{4}(3)(1)} + Z[3]e^{i\frac{2\pi}{4}(3)(3)}\right) = \frac{1}{4}(4 - 2i + 2i) = 1$$

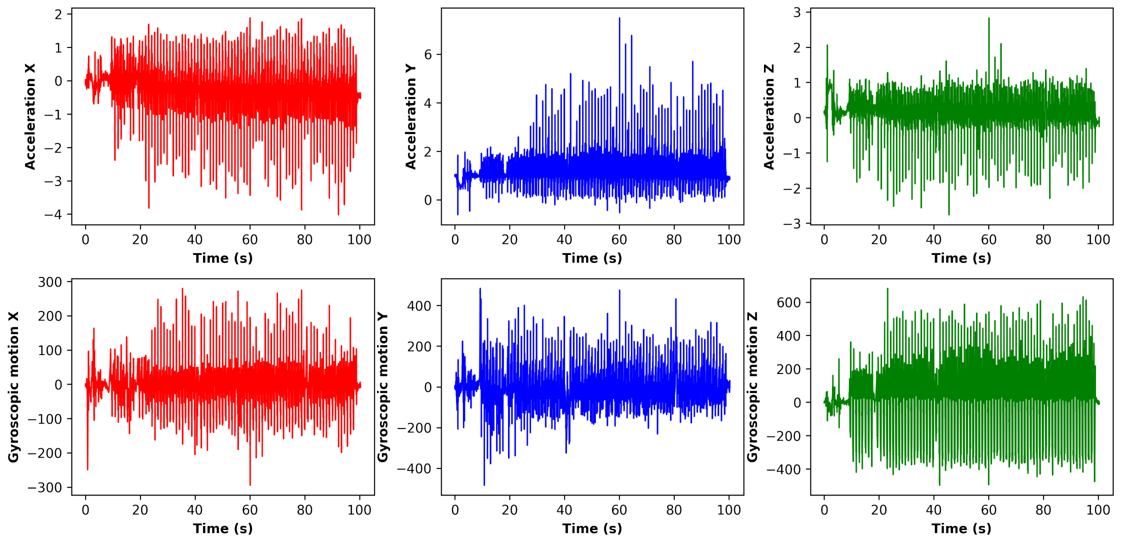
Thus,

$z[n] = [2, 1, 0, 1]$

1.4 Problem 4

Lets plot the data to get a look at it

[6]:
```
import matplotlib.pyplot as plt
import numpy as np
from scipy.io import loadmat
data = loadmat("data/hw8prob4data.mat")

plt.rcParams["axes.labelweight"] = "bold"
fig, ax = plt.subplots(2,3, figsize=(12,6))
fig.tight_layout(pad=2)

x = np.array(
    [i*4 for i in range(len(data["AccelX"]))]
)

# line format
line_settings = [
    "r-",
    "b-",
    "g-"
]

# accleromoter data
for (d,i) in zip("XYZ", range(len("XYZ"))):
    ax[0,i].plot(
        (x/1000), # time in sec
        data[f"Accel{d}"],
        line_settings[i],
        linewidth=1
    )
    ax[0,i].set_xlabel("Time (s)")
    ax[0,i].set_ylabel(f"Acceleration {d}")

# gyroscope data
for (d,i) in zip("XYZ", range(len("XYZ"))):
    ax[1,i].plot(
        x/1000, # time in sec
        data[f"Gyro{d}"],
        line_settings[i],
        linewidth=1
    )
    ax[1,i].set_xlabel("Time (s)")
    ax[1,i].set_ylabel(f"Gyroscopic motion {d}")
```



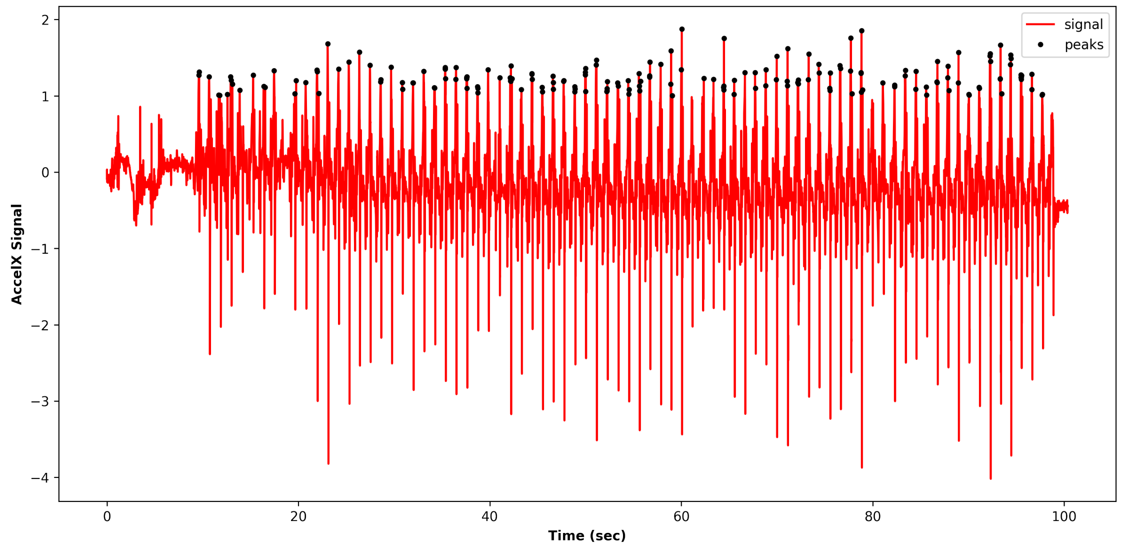
We can use `scipy`s built in peak finding function to find peaks in our walking signal. **For the remainder of this problem I am going to use only the accleration data in the X direction.**

[7]:
```
from scipy.signal import find_peaks

# extract out the accelX data and calculate peaks
accelX = data["AccelX"][:,0]
peaks, heights = find_peaks(accelX, height=1)
```

Lets plot the peaks onto our signal data to visually annotate the peaks:

[8]:
```
plt.rcParams["axes.labelweight"] = "bold"
fig, ax = plt.subplots(figsize=(12,6))
fig.tight_layout(pad=2)

time = np.array(range(len(accelX))) * 4 / 1000

ax.plot(
    time,
    accelX,
    'r-',
    label="signal"
)
ax.plot(
    peaks * 4 / 1000,
    heights['peak_heights'],
    '.k',
    label="peaks"
)
ax.legend()
ax.set_xlabel("Time (sec)")
ax.set_ylabel("AccelX Signal")
```

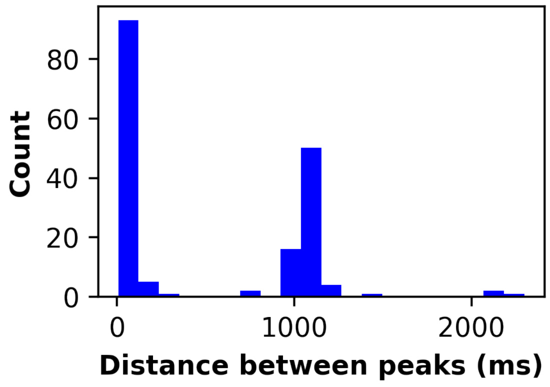
```
[8]: Text(337.79166666666663, 0.5, 'AccelX Signal')
```



We can see that our `find_peaks` function is adaquetly finding the peaks in our walking signal. There are still some points that are extremely close together, however, so lets look at a historgram of distances between two peaks to see if we can remove some data from the array to get a better picture:

```
[9]: # calculate peak distances
peak_dist = np.diff(peaks)

# convert to ms
peak_dist = peak_dist * 4

# plot
_ = plt.figure(figsize=(3,2))
plt.hist(
    peak_dist,
    bins=20,
    color="b"
)
plt.xlabel("Distance between peaks (ms)")
plt.ylabel("Count")
```

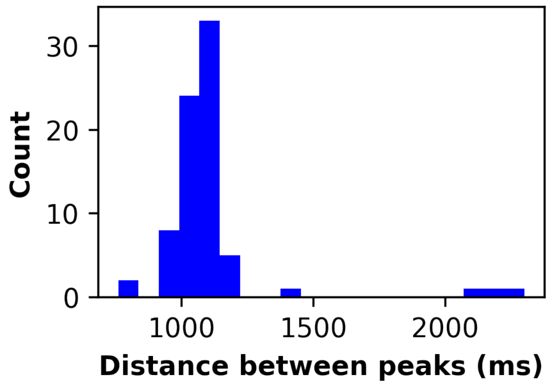
```
[9]: Text(0, 0.5, 'Count')
```



Based on this, lets remove all peaks less than 500 ms:

```
[10]: peak_dist = peak_dist[peak_dist > 500]

      # plot
      _ = plt.figure(figsize=(3,2))
      plt.hist(
          peak_dist,
          bins=20,
          color="b"
      )
      plt.xlabel("Distance between peaks (ms)")
      plt.ylabel("Count")
```

```
[10]: Text(0, 0.5, 'Count')
```



Using this final distribution, we can calculate the average step frequency by calculating the average time between steps:

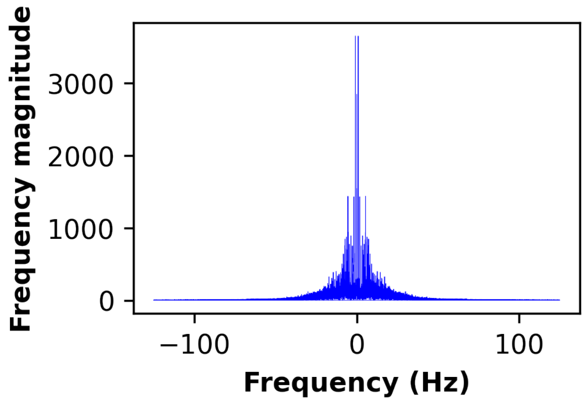
[11]:
```
avg_pace = np.mean(peak_dist)
print(f"Average steps per second: {round(1000/avg_pace, 3)} steps.")
```

```
Average steps per second: 0.905 steps.
```

Lets run the same analysis, but this time using the Fourier transform and extracting out the strongest frequencies in our signal. We can calculate the fast fourrier transform of our signal as well as get our frequencies in numpy:

[12]:
```
from numpy.fft import fft, fftfreq

T = 0.004 # ms
accelX_freq = fft(accelX)
freq = fftfreq(len(accelX), T)

# plot
_ = plt.figure(figsize=(3,2))
plt.plot(
    freq,
    np.abs(accelX_freq),
    "b-",
    linewidth=0.2
)
plt.xlabel("Frequency (Hz)")
plt.ylabel("Frequency magnitude")
```

[12]:
```
Text(0, 0.5, 'Frequency magnitude')
```



Using the above calculations, we can find our largest frequency and use that to determine the step frequency

[13]:
```
maxFreq = freq[np.argmax(np.abs(accelX_freq))]
print(f"Step frequency = {round(maxFreq, 3)} steps per second.")
```

```
Step frequency = 0.897 steps per second.
```

We can see that this value is almost identical to the value that we calculated with the prior method.

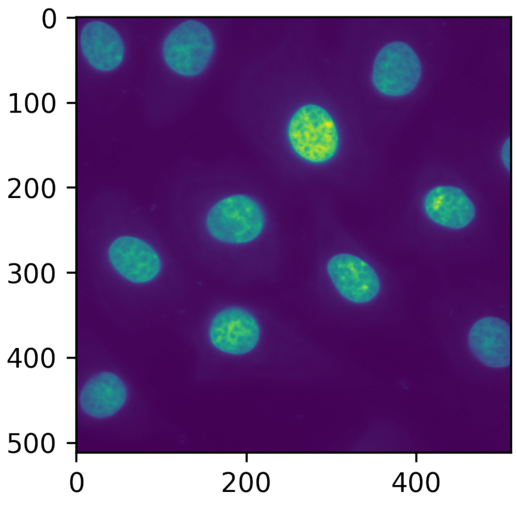
1.5 Problem 5

1.5.1 a.)

We can look at the 20th image by straight up plotting it like so:

[14]:
```
import matplotlib.pyplot as plt
data = loadmat("data/hw8prob5data.mat")
image = data['image']
psf = data['psf']
ref = data["reference"]

_ = plt.figure(figsize=(3,3))
plt.imshow(image[:,:,19])
```

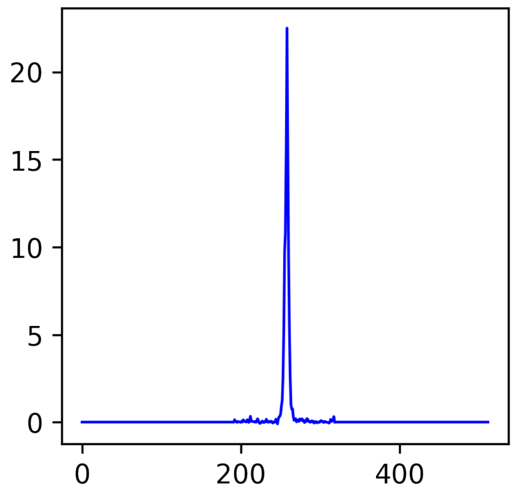
[14]:
```
<matplotlib.image.AxesImage at 0x17fd9b700>
```



As well, we can take a look at the point spread function (psf) given to us. Based on this, we can see that we may have measurement limitations in a sense that it may be hard to distinguish items in our images that are smaller than 50 nm.

[15]:
```
_ = plt.figure(figsize=(3,3))
plt.plot(
    psf[255,:,19],
    'b-',
    linewidth=1
)
```

[15]:
```
[<matplotlib.lines.Line2D at 0x17fd8bf70>]
```

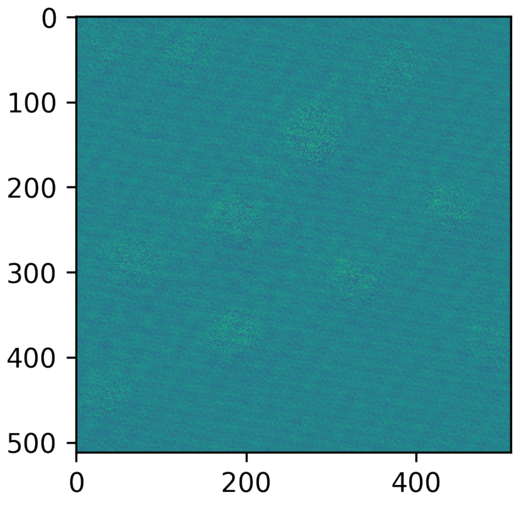


We can utilize the point spread function to attempt to deconvolute and sharpen our image by deconvoluting the image in the frequency domain:

[16]:
```
from numpy import fft

# compute the 3D fft of the image and PSF
image_fft = fft.fftn(image)
psf_fft = fft.fftn(psf)

# divide image imageFFT by psfFFT
image_new_fft = image_fft / np.abs(psf_fft)
image_new = fft.ifftn(image_new_fft)
```

[17]:
```
_ = plt.figure(figsize=(3,3))
plt.imshow(image_new.real[:,:,19])
```

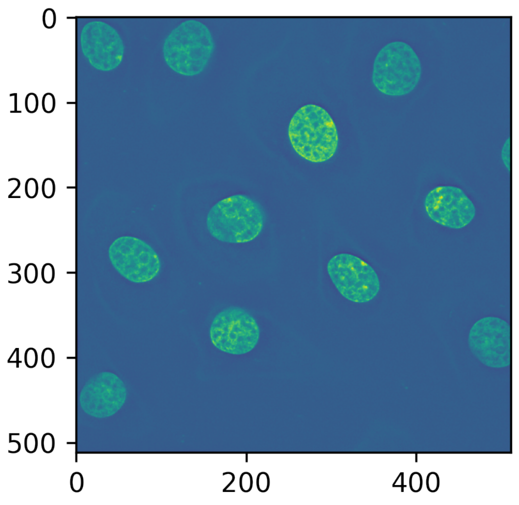
[17]:
```
<matplotlib.image.AxesImage at 0x17f879220>
```



We can change our deconvolution algorithm to get better results as such:

[18]:
```
A = 0.1 * np.max(np.max(np.max(np.abs(psf_fft))))

# divide image imageFFT by psfFFT
image_new_fft = image_fft / (np.abs(psf_fft) + A)
image_new = fft.ifftn(image_new_fft)

_ = plt.figure(figsize=(3,3))
plt.imshow(image_new.real[:,:,19])
```

[18]:
```
<matplotlib.image.AxesImage at 0x17f5cfd90>
```

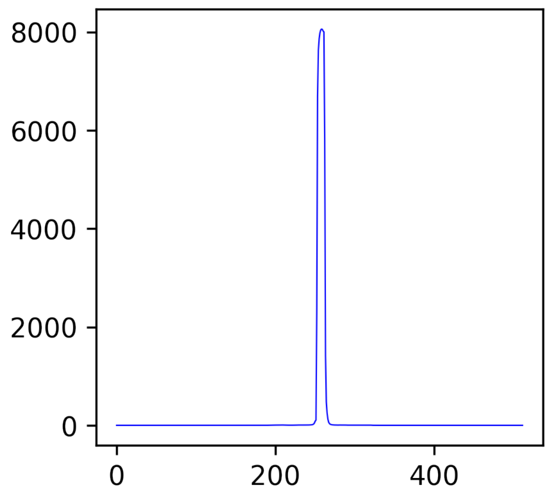


1.5.2 d.)

I am going to try and smooth the point spread function by using a two-dimensional moving-average convolution that is a window of ones.

```
[19]: from scipy.signal import convolve2d
      WINDOW = 10

      smoothedpsf = psf

      # convolute each psf slice as an square
      for i in range(psf.shape[-1]):
          smoothedpsf[:,:,i] = convolve2d(
              psf[:,:,i],
              np.ones((WINDOW, WINDOW)),
              "same"
          )

      # plot one of the psf's
      _ = plt.figure(figsize=(3,3))
      plt.plot(
          smoothedpsf[255,:,19],
          'b-',
          linewidth=0.5
      )
```

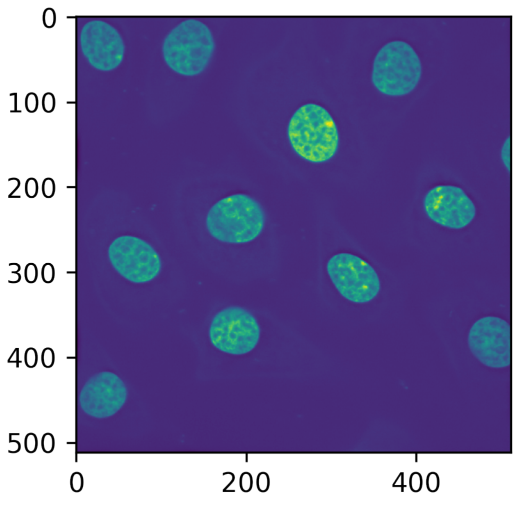
```
[19]: [<matplotlib.lines.Line2D at 0x17f950640>]
```



We can now run through the same algorithm from part c and attempt to sharpen the image:

[20]:
```
smoothedpsf_fft = fft.fftn(smoothedpsf)
A = 0.6 * np.max(np.max(np.max(np.abs(smoothedpsf_fft))))

# divide image imageFFT by psfFFT
image_new_fft = image_fft / (np.abs(smoothedpsf_fft)+A)
image_new = fft.ifftn(image_new_fft)

_ = plt.figure(figsize=(3,3))
plt.imshow(image_new.real[:,:,19])
```

[20]:
```
<matplotlib.image.AxesImage at 0x17fca73d0>
```



As we can see, the image still remains a bit blurry (if not worse than part c), but most of the original color remains in tact. Thus, smoothing our point spread function prior to deconvluting in the frequency domain seems to have some merit.