AMATH 482 Homework 1

Vineet Palepu

January 27, 2021

1. Abstract

In this paper, we are trying to find a submarine that is located in the Puget Sound. We are given acoustic data that is very noisy, and thus requires processing in order to get accurate location estimates. Using the Fourier transform, we are able to average the frequency data to find the central frequency emitted by the submarine. We then denoise the data using a Gaussian filter centered around that frequency. After doing this, we get a much smoother output on the location of the submarine and can better track it in the future.

2. Introduction and Overview

In this paper, we were tasked with locating a submarine in the Puget Sound. We were given acoustic data sampled in 3-dimensional Cartesian space at periods of 30 minutes for a total of 24 hours. However, the data was noisy and had to be processed to be made useful.

We know that the submarine emits a specific frequency, and therefore assumed all other frequencies to be noise. Thus, to denoise the signal, we took the n-dimensional fast Fourier transform in order to transform the data into frequency space. Because the noise was assumed to be white noise, it had a mean of 0, and thus when averaged over many trials allowed us to isolate only the submarine's central frequency.

Once the frequency was isolated, we then returned to the original Fourier transformed data and applied a 3-dimensional Gaussian filter to the isolate only the submarine's frequency and remove all other frequencies. We then applied the inverse Fourier transform to return the signal back into the space and time domain, at which point we find the maximal values.

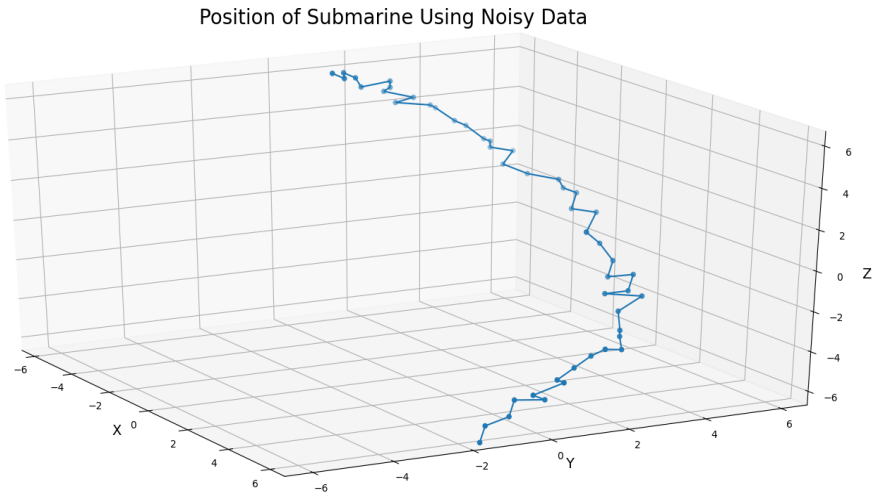

Figure 1. This plot shows the trajectory of the submarine without doing any denoising or processing of the data. The path is very jagged and likely would look smoother if we removed noise from the data

3. Theoretical Background

3.1. Fourier Transform

One of the integral parts to denoising the data is the Fourier transform, or in our case the fast Fourier transform. If we have a function, $f(x)$, its Fourier transform $\hat{f}(k)$ can be calculated as shown in Equation (1). In this case, we transform the function from the spatial or time domain into the frequency space. We can transform back from frequency space into our original domain using the inverse Fourier transform, shown in Equation (2).

$$\hat{f}(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) e^{-ikx} dx \tag{1}$$

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \hat{f}(k) e^{ikx} dk \tag{2}$$

However, one problem with this is that it requires an infinite domain, which outside of pure math, usually doesn't happen. Thus, we turn to the discrete Fourier transform. Rather than requiring a function, we only need be given a series of values. For a series of data with N values, the discrete Fourier transform (DFT) is calculated as shown in Equation (3).

$$\hat{x}_k = \frac{1}{N} \sum_{n=0}^{N-1} x_n e^{\frac{2\pi i k n}{N}} \tag{3}$$

The DFT only needs a vector of values sampled at equally spaced points. However, because of this, it does have the limitation that it can only output a limited number of frequencies, $0, 1, 2, \ldots N-1$. This is because certain frequencies, for example k and $k+N$ will appear the same due to the discrete sampling.

Finally, we arrive at the fast Fourier transform (FFT), which is an optimized version of the DFT. Unlike the DFT, which has a computational complexity of $\mathcal{O}(n^2)$, the FFT has time complexity of $\mathcal{O}(n \cdot \log n)$. This means it can be significantly faster, especially as the number of points used increases. This is important due to the moderately large size of the dataset we are using.

There also exists a corresponding inverse FFT (IFFT) that converts from the frequency domain back to the original function domain.

One thing to note is that in this paper, we used the n-Dimensional FFT and IFFT, which transform data of multiple dimensions to and from the frequency domain in each dimension. This allows us to apply filters and other processing effects to the data.

3.2. Gaussian Filtering

The other key concept used in this paper is the Gaussian filter. A filter is a function that can be applied to a signal to somehow limit or change the frequencies that appear in the signal. In our case, we do this by taking the FFT of the signal, which then easily allows us to remove certain frequencies. We do this by applying the Gaussian filter.

A Gaussian filter is a function of the form shown in Equation (4).

$$g(k) = e^{-\tau(k-k_0)^2} \tag{4}$$

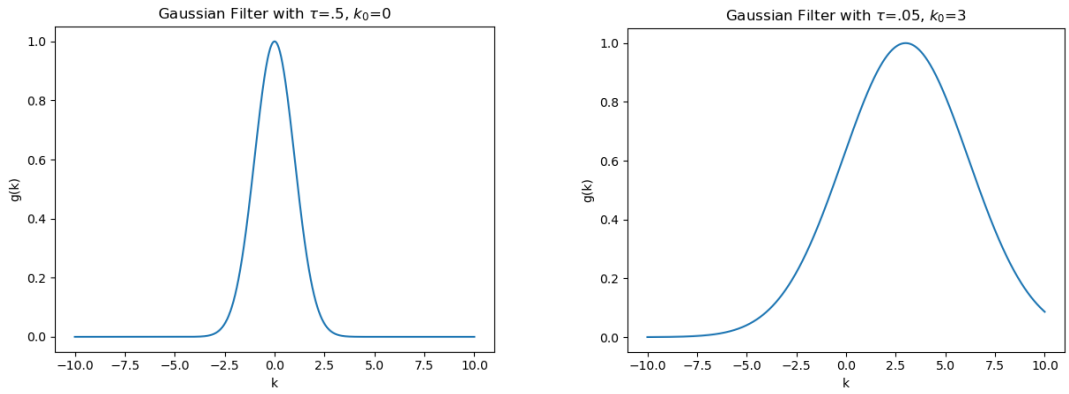

Figure 2. This is the general 2-dimensional Gaussian filter graphed with various values for the parameters τ and k_0. As τ decreases, the curve becomes wider. k_0 affects the center of the peak of the filter

There are several parameters to note. τ affects the width of the filter, with larger values making the filter narrower. k_0 represents the central frequency of the filter. That is, the filter will have its maximal value at k_0. By multiplying our signal with a Gaussian filter, we can effectively filter frequencies far outside the central frequency. If we then use the IFFT, we get back our original signal without the unwanted noise. However, one thing to note is that this requires knowledge of what frequency you are searching for. If improperly applied, the Gaussian filter could remove the signal itself, and leave only noise.

4. Algorithm Implementation and Development

To start, the first thing we did was to visualize the path of the submarine without denoising to get a baseline. From this path, shown in Figure 1, we saw that while the overall path was clear, individual locations were very sporadic and jumpy. Thus, denoising was necessary to get a more accurate view of the submarine's location. This was done in Section 0 of Appendix B.

To start, as mentioned above, in order to use a Gaussian filter, you need a central frequency to filter around. As we are trying to track a new type of submarine, we did not know what frequency it emitted. Thus, our first task was to figure out what frequency was emitted.

To do this, we started by taking the n-dimensional FFT of the data over the three spatial axes. This gives us the frequency distribution at each point in time. Since the FFT is shift invariant, meaning shifting the signal in the time domain doesn't affect the frequencies given, we can then average all 49 instances. (Note that while the FFT is shift invariant if you look at the magnitude, the imaginary part does change based on the shift; this is how the original signal is able to be recovered, even though it may appear as if information is being lost). After averaging the transformed data, we are left with a $64 \times 64 \times 64$ matrix, with each point representing how present a certain frequency is. We then normalized the data by dividing by its maximal value. Thus, to find the central frequency, all we did was find the maximal value of the matrix. However, the maximal value itself doesn't tell us anything; we instead looked for where the maximal value occurs. This gives us an index into the array. We then translated the index into the 3-dimensional index, which gives us the index for each dimension. Again however, this is just an index, not the actual location. To get the actual coordinates, we used the fact that the FFT gives us frequencies in a range of $-\pi$ to π. So, all we need to do is rescale the frequencies by $2\pi/20$. Doing this, we get the dominant frequency in each axis. This was done in Section 1a of Appendix B

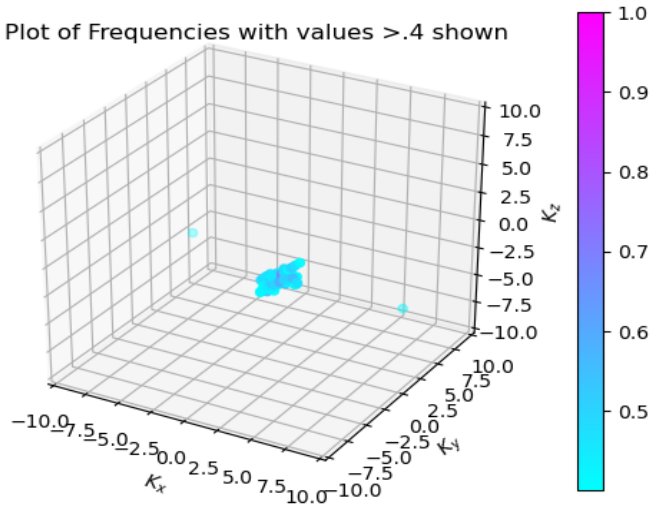

Figure 3. Here we have the superlevel set of the frequencies, using a value of .4 as the threshold. The maximal value lies somewhere in the blue spot, as seen by the purple and pink points that are slightly obscured.

Now, we have our central location to filter around. As a quick check, we visualized the graph of the averaged and normalized FFT of the data, opting to only show those frequencies with a value of .4 or greater (to make it easier to see in three dimensions). In Figure 3, we can easily see that the points with highest magnitude are very near to the central frequency, confirming that we properly translated the index into a location. This was done in Section 1b of Appendix B.

Next, we had to generate the Gaussian filter to apply to each time point. We used our newly found central frequency to generate a 3-dimensional Gaussian filter. Again, we visualize the Gaussian filter by graphing only those points that have a value of .5 or greater, to show the frequencies that are left most unchanged by the filter. Points outside this region will be multiplied by a value of .5 or smaller, meaning that those frequencies will be greatly attenuated. Visualizing the filter allowed us to make sure that we implemented the filter correctly and that it is positioned correctly. This was done in Section 2a of Appendix B.

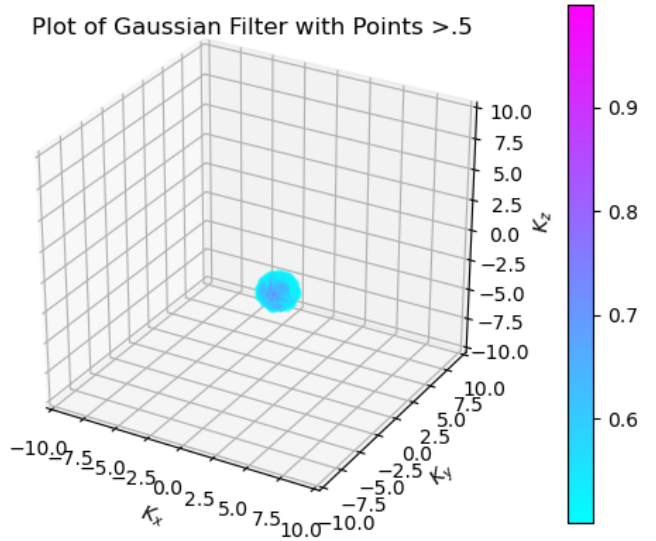

Figure 4. This plot shows the superlevel set of the 3D Gaussian filter with parameter $\tau = .5$ and $(k_x, k_y, k_z) = (5.3125, -6.875, 2.1875)$. The value of .5 was chosen for the superlevel set, as outside this sphere, frequencies would be more than halved.

Now that we had verified the filter was correct, we looped over each of the 49 individual realizations, and took the FFT. After we have the FFT of the data, we perform element-wise multiplication of the filter with the transformed data. This removed most of the noise frequencies. Next, we transformed the data back into its original domain, where we found the maximum value. Again, we translate the maximum value into its actual location and save this to an array. This was done in Section 2b of Appendix B.

At this point, we had to tune the parameter τ on the Gaussian filter. This process was done by trial and error, testing out larger and smaller values of τ. When $\tau > 1$, meaning that only a very narrow range of frequencies were not attenuated greatly, the resulting data was still very noisy, likely due to the fact that our central frequency is not exactly correct and we cannot perfectly remove all noise. With large τ, it is likely that we also removed some of the actual data itself and not just the noise. When $\tau < .1$, we found that again the data still appeared noisy, likely because many of the noise frequencies were not attenuated enough. We found that the best value was $\tau = .5$, as at this width the filter removed much of the noise, but wasn't too exact on the frequencies left alone.

5. Computational Results

Now, we will show the results achieved by following the methods described in the previous section. To start, we computed the central frequency and found it to be $(k_x, k_y, k_z) = (5.3125, -6.875, 2.1875)$.

Next, after using the Gaussian filter centered around our central frequency, we found the coordinates of the submarine at each point in time and graphed them on a 3-dimensional plot shown in Figure 5.

We compared this to the location found with noisy data, shown in Figure 6, and saw a significant improvement. This was done in Section 2c of Appendix B.

Finally, from these 3-dimensional locations, we computed where to send our aircraft to follow the submarine. Since the aircraft is in the air and cannot follow the submarine underwater, we ignore the z coordinate of the location. That way, the aircraft will be directly overhead the submarine at each point in time measured if it follows this path. This was done in Section 3 of Appendix B.

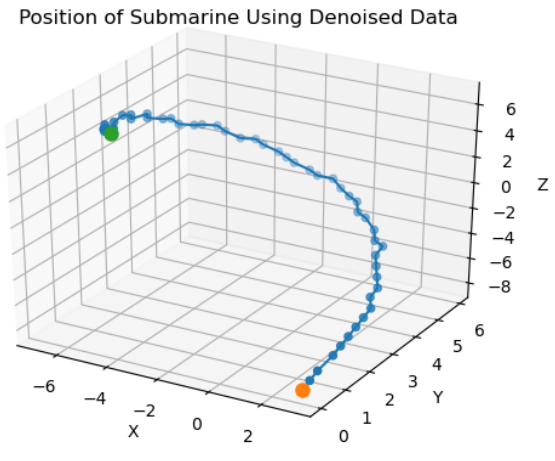

Figure 5. This plot shows the location of the submarine after denoising the data. The green dot is the starting location and the red dot is the final location.

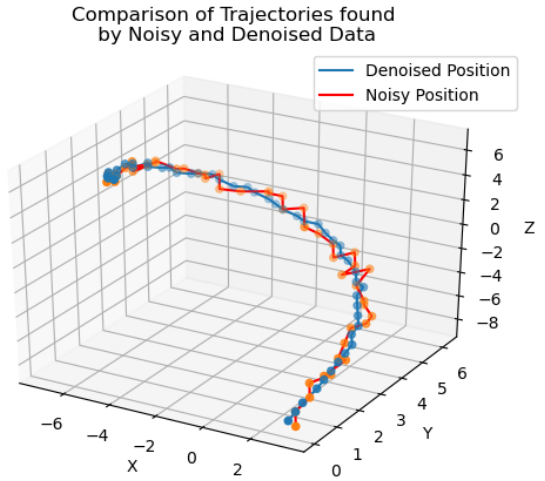

Figure 6. This graph compares the two trajectories. The denoised position (blue) is much smoother than the position gathered from noisy data (red)

Time:	1	2	3	4	5	6	7	8	9	10	11	12
X:	3.125	3.125	3.125	3.125	3.125	3.125	3.125	3.125	3.125	2.8125	2.8125	2.5
Y:	0	0.3125	0.625	1.25	1.5625	1.875	2.1875	2.5	2.8125	3.125	3.4375	3.75

Time:	13	14	15	16	17	18	19	20	21	22	23	24
X:	2.1875	1.875	1.875	1.5625	1.25	0.625	0.3125	0	-0.3125	-0.9375	-1.25	-1.875
Y:	4.0625	4.375	4.6875	4.6875	5	5.3125	5.3125	5.625	5.625	5.9375	5.9375	5.9375

Time:	25	26	27	28	29	30	31	32	33	34	35	36
X:	-2.1875	-2.8125	-3.125	-3.4375	-4.0625	-4.375	-4.6875	-5.3125	-5.625	-5.9375	-5.9375	-6.25
Y:	5.9375	5.9375	5.9375	5.9375	5.9375	5.9375	5.625	5.625	5.625	5.3125	5	4.6875

Time:	37	38	39	40	41	42	43	44	45	46	47	48	49
X:	-6.5625	-6.5625	-6.875	-6.875	-6.875	-6.875	-6.875	-6.5625	-6.25	-6.25	-5.9375	-5.625	-5
Y:	4.6875	4.375	4.0625	4.0625	3.4375	3.4375	3.125	2.5	2.1875	1.875	1.5625	1.25	0.9375

Table 1. Here, we show the location of the submarine in x, y coordinates at each time period. This is the location that the plane should fly to in order to follow the submarine.

We generated a table of locations for the plane to go to, which can be seen in Table 1.

6. Summary and Conclusions

By using the FFT and averaging over multiple time periods, we were able to take the noisy location data and find the central frequency emitted by the submarine. Using that frequency, we were able to apply a Gaussian filter to denoise the data and find more accurate estimates of the location of the submarine. One thing to note is that real time tracking of the submarine would have been difficult because after the first measurement, we would have had no data to average in order to find the central frequency. Denoising would have been significantly more difficult. However, now that we know the frequency, we can apply this to any future data, without needing multiple time points, and will be able to denoise. Thus, if we were given only a single frame of data and tasked to find the submarine, this would now be possible, allowing our plane to follow the submarine in the future in real time.

Appendix A: Python Functions Used

NumPy

`np.abs` : Returns the absolute value. Used because we have complex data, but only care about the magnitude of each data point.

`np.amax` : Returns the largest value of a matrix. Used to normalize certain matrices.

`np.append` : Adds to matrices or vectors together.

`np.arange` : Generates a vector with consecutive integer values. We used this to generate the rescaled frequencies.

`np.argmax` : Gets the index of the maximal element, although for the flattened array. We still need to use `np.unravel_index` to get the indexes to each axis.

`np.exp` : Calculates the exponential. Used to generate the 3D Gaussian filter.

`np.fft.fftn` : Calculates the n-dimensional fast Fourier transform. Used to translate our data into frequency space.

`np.fft.fftshift` : Shifts the zero-frequency to the center of the vector. Used because of how the fast Fourier transform and inverse fast Fourier transform structure their return values.

`np.fft.ifftn` : Calculates the n-dimensional inverse fast Fourier transform. Used to convert data from the frequency domain back to its original domain.

`np.fft.ifftshift` : Shifts the zero-frequency back when doing the inverse transform. Used again because of how the return values are structured after the inverse fast Fourier transform.

`np.genfromtxt` : Allows us to generate a NumPy matrix from a csv file.

`np.linspace` : Creates a vector of evenly spaced numbers. Used to generate the vector of submarine locations for each axis.

`np.load` : Load in a matrix stored as a *.npy file. Saves time over loading csv file.

`np.mean` : Used to calculate the mean of the matrix over the 49 time samples.

`np.meshgrid` : Used for 3-dimensional graphing and converting indexes into the values they represent, like position or frequency.

`np.reshape` : Allows us to change the dimensions of a matrix. We used this to change the matrix from a 262144×49 matrix into the properly dimensioned $64 \times 64 \times 64 \times 49$ matrix.

`np.save` : Save a matrix in NumPy format as a *.npy file. Reading this is much faster than reading a csv.

`np.savetxt` : Save a matrix in *.csv format so that it can easily be transformed into a table.

`np.unravel_index` : Converts a flat index into the n-dimensional index into an array.

MatPlotLib

`pyplot.figure` : Generates a figure to display graphs on.

`pyplot.plot` : Creates a line graph of the data. We used this to generate the trajectories

`pyplot.scatter` : Creates a scatterplot of the data. We used this to indicate the discrete points along the trajectory.

7. Appendix B

```
import numpy as np
from numpy import genfromtxt
import os
import matplotlib.pyplot as plt
from numpy.fft import fftn, fftshift, ifftn, ifftshift

if not os.path.isfile("subdata.npy"):
    subdata = np.genfromtxt("subdata modified.csv", dtype=np.complex64, delimiter=',')
    np.save("subdata", subdata)
else:
    subdata = np.load("subdata.npy")

subdata = np.reshape(subdata, (64,64,64,49), order='F')

L = 10
n = 64
x = np.linspace(-L, L, n+1)[0:n]
y = np.linspace(-L, L, n+1)[0:n]
z = np.linspace(-L, L, n+1)[0:n]
k = (2*np.pi / (2 * L))*(np.append(np.arange(0, n/2), np.arange(-n/2, 0)))
ks = fftshift(k)

tau = .05
k0=3
gauss_filter = np.exp(-tau * (np.linspace(-10, 10, 1000)-k0)**2)
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_xlabel('k')
ax.set_ylabel('g(k)')
ax.plot(np.linspace(-10, 10, 1000), gauss_filter)
plt.title('Gaussian Filter with $\\tau$=.05, $k_0$=3')
plt.show()

X, Y, Z = np.meshgrid(x, y, z)
```

```
Kx, Ky, Kz = np.meshgrid(ks, ks, ks)

# Section 0: Plot trajectory from noisy data
noisy_pos = np.zeros((49, 3))
for i in range(49):
    x_ind, y_ind, z_ind = np.unravel_index(np.argmax(np.abs(subdata[:,:,:,i])), subdata[:,:,:,
i].shape)
    noisy_pos[i, :] = [X[x_ind, y_ind, z_ind], Y[x_ind, y_ind, z_ind], Z[x_ind, y_ind, z_ind]]

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_xlabel('X', fontsize=14)
ax.set_ylabel('Y', fontsize=14)
ax.set_zlabel('Z', fontsize=14)
plt.title('Position of Submarine Using Noisy Data', fontsize=20)
ax.set_xlim([-10, 10])
ax.set_ylim([-10, 10])
ax.set_zlim([-10, 10])
ax.plot(noisy_pos[:, 0], noisy_pos[:, 1], noisy_pos[:, 2])
ax.scatter(noisy_pos[:, 0], noisy_pos[:, 1], noisy_pos[:, 2])
plt.show()

# Section 1: Determine frequency signatures

# Section 1a: Take the Fourier Transform over the 3 spatial axes
subdata_transformed = fftshift(fftn(subdata, axes=(0,1,2)))
subdata_transformed_mean = np.mean(subdata_transformed, axis=3)
subdata_transformed_normalized = subdata_transformed_mean / np.amax(subdata_transformed_mean)
x_ind, y_ind, z_ind = np.unravel_index(np.argmax(subdata_transformed_normalized), subdata_tran
sformed_normalized.shape)
freq = np.array([X[x_ind, y_ind, z_ind], Y[x_ind, y_ind, z_ind], Z[x_ind, y_ind, z_ind]])
print(freq)

# Generate points to graph, skipping those that are below the threshold
def getXYZC(array, X, Y, Z, threshold):
    x = []
    y = []
    z = []
    c = []

    for xi in range(array.shape[0]):
        for yi in range(array.shape[1]):
            for zi in range(array.shape[2]):
                val = np.abs(array[xi, yi, zi])
                if val < threshold:
```

```
                    continue
                x.append(X[xi, yi, zi])
                y.append(Y[xi, yi, zi])
                z.append(Z[xi, yi, zi])
                c.append(val)

    return x,y,z,c

# Section 1b: Show frequencies that are strongest
x,y,z,c = getXYZC(subdata_transformed_normalized, Kx, Ky, Kz, .4)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
img = ax.scatter(x,y,z, c=np.abs(c), cmap=plt.cool())
ax.set_xlabel('$K_x$')
ax.set_ylabel('$K_y$')
ax.set_zlabel('$K_z$')
plt.title('Plot of Frequencies with values >.4 shown')
ax.set_xlim([-10, 10])
ax.set_ylim([-10, 10])
ax.set_zlim([-10, 10])
fig.colorbar(img)
plt.show()

# Section 2: Determine path of submarine

# Section 2a: Generate and visualize Gaussian filter
width = .5
gauss = np.exp(-width * ((Kx - freq[0])**2 + (Ky - freq[1])**2 + (Kz - freq[2])**2))
x,y,z,c = getXYZC(gauss, Kx, Ky, Kz, .1)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
img = ax.scatter(x,y,z, c=np.abs(c), cmap=plt.cool())
ax.set_xlabel('$K_x$')
ax.set_ylabel('$K_y$')
ax.set_zlabel('$K_z$')
plt.title('Plot of Gaussian Filter with Points >.5')
ax.set_xlim([-10, 10])
ax.set_ylim([-10, 10])
ax.set_zlim([-10, 10])
fig.colorbar(img)
plt.show()

# Section 2b: Apply Gaussian filter to data and take inverse transform
pos = np.zeros((49, 3))
for i in range(49):
```

```
    subdata_i_transformed = fftshift(fftn(subdata[:,:,:,i]))
    subdata_i_transformed_filtered = subdata_i_transformed * gauss
    subdata_filtered = np.abs(ifftn(ifftshift(subdata_i_transformed_filtered)))
    x_ind, y_ind, z_ind = np.unravel_index(np.argmax(subdata_filtered), subdata_filtered.shape
)
    pos[i, :] = [X[x_ind, y_ind, z_ind], Y[x_ind, y_ind, z_ind], Z[x_ind, y_ind, z_ind]]

# Section 2c: Graph results
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.title('Position of Submarine Using Denoised Data')
ax.plot(pos[:, 0], pos[:, 1], pos[:, 2])
ax.scatter(pos[1:-1, 0], pos[1:-1, 1], pos[1:-1, 2])
ax.scatter(pos[0, 0], pos[0, 1], pos[0, 2], 'g*', s=64)
ax.scatter(pos[-1, 0], pos[-1, 1], pos[-1, 2], 'r*', s=64)
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111,projection='3d')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.title('Comparison of Trajectories found\n by Noisy and Denoised Data', y=1.08)
ax.plot(pos[:, 0], pos[:, 1], pos[:, 2], label="Denoised Position")
ax.scatter(pos[:, 0], pos[:, 1], pos[:, 2])
ax.plot(noisy_pos[:, 0], noisy_pos[:, 1], noisy_pos[:, 2], 'r', label="Noisy Position")
ax.scatter(noisy_pos[:, 0], noisy_pos[:, 1], noisy_pos[:, 2], 'r')
ax.legend(loc="upper right")
plt.show()

# Section 3: Determine coordinates to send subtracking aircraft
print(pos[:, 0:2])
np.savetxt("Sub Pos.csv", pos[:, 0:2])
```