AMATH 582 Homework 1: A submarine problem

Brady Griffith

January 27, 2021

Abstract

In this project, 49 wide spectrum sonar recordings are used to identify that a submarine in the Pudget Sound with a characteristic frequency around $k_x = -0.078$, $k_y = -0.125$, and $k_z = 0.047$. When filtering out other noise at other frequencies, the location of the submarine in each recording becomes observable. The location is automatically detected to give a the submarine's path.

1 Introduction and Overview

The data set presented for this problem contains 49 spectrum acoustic measurements over a 24 hour period. These should contain the signature of a submarine, but the noise is too high. Since the submarine is moving, it isn't possible to simply average all of the samples.

Instead, it would be preferable to remove as much of the noise as possible without changing the submarine information. If the submarine produces a characteristic frequency, a band pass filter around it would remove most of the noise while leaving the signal intact.

The objective of this project is to first determine the frequency signature of the submarine. Then, with filtering, determine the submarine's location in each recording. Finally, the x-y coordinates of the submarine need to be reported so that a P-8 Orion may be deployed to follow the submarine.

This process was modeled in a single dimension in both the textbook for this course [1] and the lectures. This project applies that methodology, but now in 3D. It also asks for a more quantitative statement about the location of the submarine, so an automated process to identify the peak in space is developed.

2 Theoretical Background

The key principle allowing this analysis has to do with how a shift in space affects the wavenumber, and vice versa. As an illustration of this point, consider the following signal in space

$$f(x) = Ae^{2\pi i k_0 x} \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-x_0)^2}{2\sigma^2}} \tag{1}$$

and its fourier transform

$$\hat{f}(k) = Ae^{-2\pi i(k-k_0)x}e^{\frac{-(k-k_0)^2\sigma^2}{2}} \tag{2}$$

The important observation is that when moving the submarine in time (by shifting k_0), it only affects the complex phase of the fourier transform, not the magnitude. This means that, as long as the center frequency remains the same, the movement of the submarine will not affect the power spectral density.

When examining the signal in frequency space, I make use of the power spectral density (PSD). It is defined as follows

$$PSD(k_x, k_y, k_z) = \frac{1}{2}\frac{|\hat{f}(k_x, k_y, k_z)|^2}{(\Delta k)^3} \tag{3}$$

where $\hat{f}(k_x, k_y, k_z)$ is the fourier transform of the signal. It has the advantage of being real and positive. It also has reasonable physical intuition, as integrating the PSD over some volume of frequency space gives the energy there. The unspecified units of this problem however mean that this is much less important.

3 Algorithm Implementation and Development

The data is loaded from the subdata.mat file using the scipy scioy.io.loadmat function. The axes that will be used with the signal in both time and frequency space are calculated. Note that when calculating the wavenumber using np.fft.fftfreq, you must multiply by (2*np.pi/(2*L)) so that it is scaled correctly to angular frequency.

The discrete fourier transform of the provided recordings is calculated using the numpy n dimentional fft, numpy.fft.fftn. This is then averaged over all of the recordings and then the PSD of this is calculated. It is projected onto the x-y, x-z, and y-z axes so that the shape of the characteristic frequency can be noted and used when deciding filter parameters.

The frequency response along each axis, integrating the perpendicular PSD, yields the total energy per slice width of that axis. The advantage of looking at it this way is that finding the peak now becomes a 1D problem. The result is a smooth bell curve shaped peak around the center frequency. I would like to identify two properties of this, the location of the center and it's width, so the filer can sufficiently wide.

I choose to find this by fitting the k_{0i}, bg, σ, and A parameters in equation 4 using the scipy scipy.optimize.curve_fit function. This projection along with the best fit is illustrated in figure 3. This has the advantage of being simple to implement and tolerant of the remaining noise. This sort of optimization is well behaved on this data, so I don't need to worry about common problems like false fits. It does have the disadvantage of being slow, but since this is only used a few times on this problem, I am willing to accept the added computation time for the simplicity. This was used instead of manually determining the parameters because this code could be reused when identifying the submarine position.

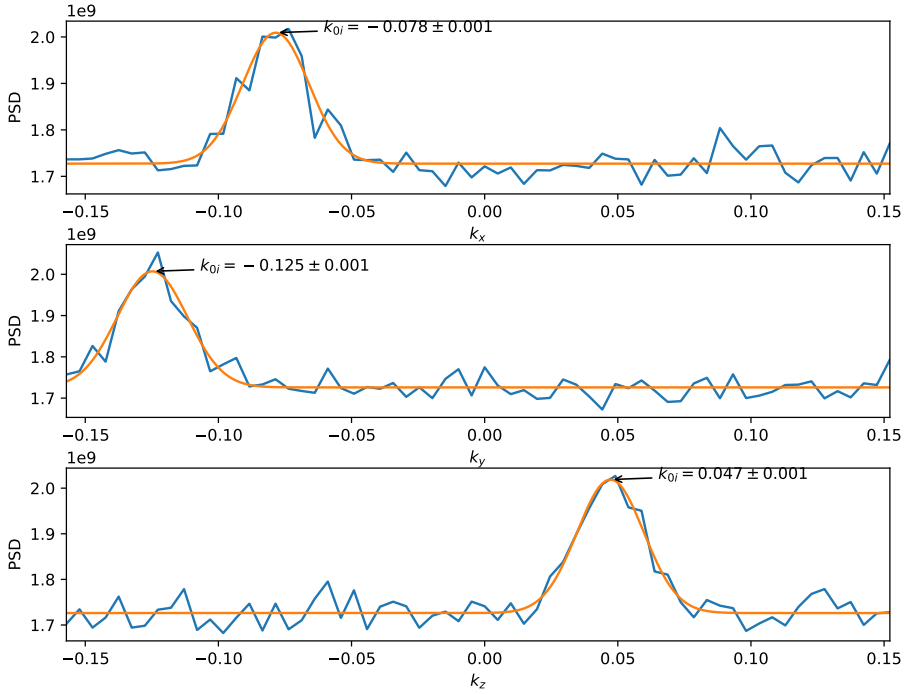

Figure 1: The PSD of the frequency averaged over all recordings, projected onto the 3 axes, along with the best fit curve in orange.

$$f(k_i) = \frac{A}{\sqrt{2\pi}\sigma} \exp - \frac{(k_i - k_{i0})^2}{2\sigma} + bg \tag{4}$$

In order to get the best results some care must be taken that the initial value of the optimization are somewhat close to solution. As such, I chose the initial background, bg, to be the minimum of the data. I chose A to be the integral over the axis of the data with bg subtracted. I chose $k_{0i} = 0$, the center of the range and $w = 8\Delta k$, 1/8 of the axis.

With the central frequency characterized, it is time to multiply a filter to remove the noise away from it. A gaussian filter is used because a sharp cutoff is not needed, but the introduction of any ringing into the space domain would be unideal. The filter used was a symmetric 3D gaussian about some center (k_{0x}, k_{0y}, k_{0z})

$$\mathcal{F}(k_x, k_y, k_z) = \frac{1}{(2\pi\sigma^2)^{3/2}} \exp\left(-\frac{(k_x - k_{0x})^2 + (k_y - k_{0y})^2 + (k_z - k_{0z})^2}{2\sigma^2}\right) \tag{5}$$

This is a simple design for the filter. A true 3D gaussian with 3 different

covariance axes might have been able to reduce the noise further while preserving the signal, but this design is good enough to leave the a prominent maximum, and much simpler to determine the ideal parameters.

The width is chosen to be the center of the fits to equation 4. The width, σ, is chosen to be twice the maximum of the axes' fit for standard deviations. This is multiplied and then the inverse discrete fourier transform ia taken, again using the function from numpy. This then results in 49 different 3D recordings with a prominent peak.

The same fitting functions are then used again. Here it is used 49 times, far too many to manually identify each peak, but still few enough to not seek a computationally efficient peak identifier. The center from each best fit reported and then put in a list to plot the path.

4 Computational Results

The submarine produces a characteristic frequency centered at $k_x = -0.078$, $k_y = -0.125$, and $k_z = 0.047$. It's shape is shown projected in figure 4. It appears to flattened along the $\langle 1, -1, 0 \rangle$.

The submarine travels from $(3.017, -7.984, -0.012)$ to $(-5.093, 6.423, 0.840)$. The full path is drawn in figure 4.

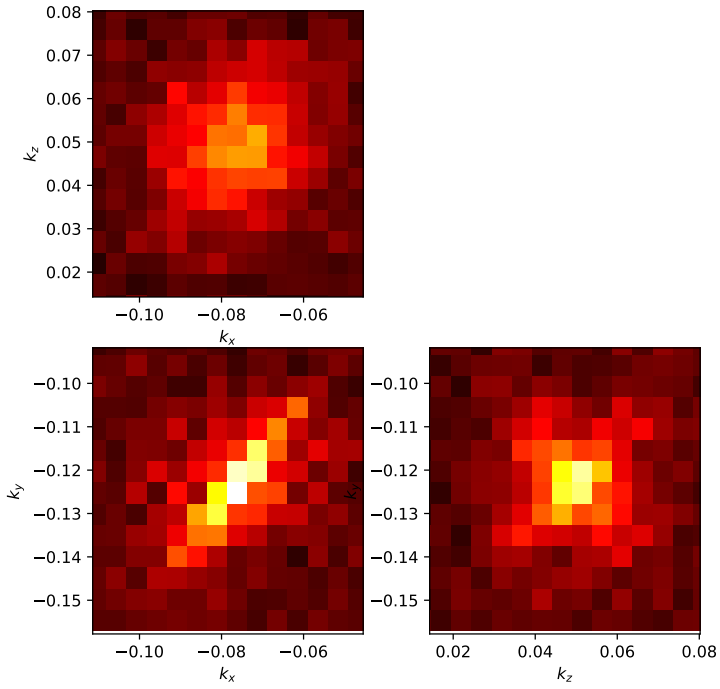

Figure 2: Total PSD of the frequency averaged over all recordings, integrated along the perpendicular axis for the x-y, y-z, and x-z planes. This shows the shape of submarine's characteristic frequency.

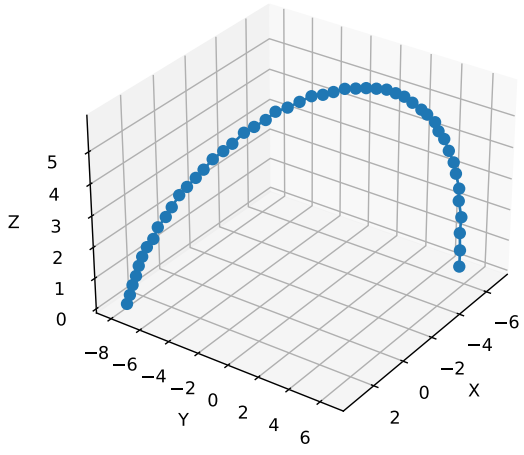

Figure 3: Path of the submarine. Each dot is the location during one of recordings. You can see the movement of the sub in half hour steps.

5 Summary and Conclusions

When filtered to just the characteristic frequency of the submarine, $k_x = -0.078$, $k_y = -0.125$, and $k_z = 0.047$, the location can clearly be picked out in each recording. The submarine ends at the XY position $(-5.093, 6.423)$. I would send the P-8 Orion there so it can follow the submarine.

References

[1] J. N. Kutz, *Data-driven modeling & scientific computation: methods for complex systems & big data*. Oxford: Oxford University Press, first edition ed., 2013. OCLC: ocn858608087.

A Python Functions

I'm going to leave out this section in the future, since it seems redundant. For the information that would be here, I direct the reader to the docstrings at the beginning of each function.

B Python Code

B.1 main.py

```
import numpy as np
import scipy.io

import fourier
import peak_finding
import visualize


def run_all():
    # Shape of the submarine data
    n_w = 64  # Space points
    n_t = 49  # Time points
    L = 10    # Width of the cube of space

    # Establish axes in space
    x = np.linspace(-L, L, n_w + 1)[:-1]
    y = x.copy()
    z = x.copy()

    # And in wavenumber
    k = (2*np.pi/(2*L))*np.fft.fftfreq(n_w)
    ks = np.fft.fftshift(k)
    dk = k[1] - k[0]

    # Load the data file
    sub_data_file = 'data/subdata.mat'
    sub_data_raw = scipy.io.loadmat(sub_data_file)['subdata']

    # Reshape the data to be shaped (x, y, z, t)
    sub_data = np.reshape(sub_data_raw, (n_w, n_w, n_w, n_t))

    # Move into wavenumber space
    sub_spectrum = fourier.fourier_transform(sub_data)
    psd = fourier.psd_averaged(sub_spectrum, dk)
```

```
    # Identify the characteristic frequency
    comp_opt, comp_cov = peak_finding.fit_center(psd, ks)
    visualize.plot_xyz_psd(psd, ks, comp_opt, comp_cov)
    center_f = peak_finding.get_center(comp_opt)
    signal_width = peak_finding.get_signal_width(comp_opt)
    visualize.plot_projections(psd, ks, center_f, 2.5*max(signal_width))

    # Filter the signal and return to position space
    filer_width = 2*max(signal_width)
    sub_spec_filtered = \
        fourier.filter_around(sub_spectrum,
                              center_f,
                              filer_width,
                              [ks]*3)

    sub_data_filtered = fourier.inv_fourier_transform(sub_spec_filtered)
    sub_data_psd_f = np.abs(sub_data_filtered)**2

    # Find that sub
    sub_loc = peak_finding.find_peaks_over_t(sub_data_psd_f, x)
    visualize.record_path(sub_loc)


if __name__ == '__main__':
    run_all()
```

B.2 fourier.py

```
import numpy as np


def fourier_transform(data_space):
    """Uses np.nfft to calculate the 3D Fourier
    transform of each spacial recording.

    Args:
        data_space (np.ndarray): 4D array arranged with
        axes X,Y,Z,T

    Returns:
        np.ndarray: The fourier transformed data. Arranged
            with axes Kx,Ky,Kz,T
    """
    return np.fft.fftn(data_space, axes=[0, 1, 2])
```

```
def inv_fourier_transform(data_spec):
    """Uses np.infft to calculate the 3D Inverse
    Fourier transform of each frequency recording.

    Args:
        data_spec (np.ndarray): 4D array arranged with
        axes Kx,Ky,Kz,T

    Returns:
        np.ndarray: The inverse fourier transformed data.
            Arranged with axes X,Y,Z,T
    """
    return np.fft.ifftn(data_spec, axes=[0, 1, 2])


def psd_averaged(data_spec, dk):
    """Averages the provided spectral data over all recordings
    and then returns its PSD.

    Args:
        data_spec (np.ndarray): 4D array arranged with
        axes Kx,Ky,Kz,T
        dk: The frequency step size

    Returns:
        np.ndarray: PSD from the averaged frequency. Reduced 1
        dimension from data_spec
    """
    data_avg_spec = np.apply_along_axis(np.mean, 3, data_spec)
    psd = .5 * np.abs(data_avg_spec)**2 / (dk**3)
    return psd


def filter_around(data, center, width, axes):
    """Applies a symmetric 3D gaussian filter to each recording
    in a provided frequency array.

    Args:
        data (np.ndarray): 4D array arranged with axes Kx,Ky,Kz,T
        center (tuple):

    Returns:
        np.ndarray: Data with the filter applied
    """
    u, v, w = axes
    U, V, W = np.meshgrid(u, v, w)
```

```
    R2 = sum([(U_i - c_i)**2 for U_i, c_i in zip([U, V, W], center)])
    filter_weights = np.exp(-R2/(2*width**2)) / (2 * np.pi * width**2)**(3/2)
    return data * np.tile(filter_weights[:, :, :, np.newaxis],
                          np.shape(data)[-1])
```

B.3 peakfinding.py

```
import numpy as np
import scipy.optimize


def gaussian_model(u, a, c, width, bg):
    """A 1D gaussian added to a background to be used
    for fitting in the peak identification.

    Args:
        u: The independent variable
        a (float): The scaling factor of the gaussian
        c (tuple): 3 long tuple of the center x, y,
            and z of the gaussian
        width (float): The standard deviation of the gaussian
        bg (float): The constant value added to the gaussian

    Returns:
        The model with provided parameters at u
    """
    return a * np.exp(-(u-c)**2/(2*(width**2))) \
                / (np.sqrt(2*np.pi)*width) + bg


def fit_axis(u, data):
    """Finds the optimal parameters of to fit gaussian_model
    to the data over the axis u.

    Args:
        u (np.ndarray): The axis to fit over
        data (np.ndarray): The 1D data det evaluated over u to fit

    Returns:
        popt (tuple): The optimal parameters
        pcov (np.ndarray): The covariance matrix of
            the parameter fit
    """
    du = u[1] - u[0]

    # Choose the initial guess. A sensible choice
```

```
    # is important to reliable convergence
    bg0 = np.min(data)
    a0 = np.sum(data - bg0)*du
    c0 = u[np.argmax(data)]
    popt, pcov = scipy.optimize.curve_fit(gaussian_model,
                                          u,
                                          data,
                                          p0=(a0,
                                              c0,
                                              8*du,
                                              bg0))
    return popt, pcov


def fit_center(data_psd, u):
    """Averages the psd along each axis and then returns the
    best fit parameters and covariance matrix for each
    """
    over = [(0, 2), (1, 2), (0, 1)]

    comp_opt = []
    comp_cov = []

    for axis in over:
        comp_avg = np.mean(data_psd, axis=axis)
        popt, pcov = fit_axis(u, comp_avg)
        comp_opt.append(popt)
        comp_cov.append(pcov)

    return comp_opt, comp_cov


def get_center(comp_opt):
    return [popt[1] for popt in comp_opt]


def get_center_unc(comp_cov):
    return [np.sqrt(np.diag(pcov)[1]) for pcov in comp_cov]


def get_signal_width(comp_opt):
    return [popt[2] for popt in comp_opt]


def find_peaks_over_t(data, u):
    centers = np.zeros((3, np.shape(data)[-1]))
```

```
    for i in range(np.shape(data)[-1]):
        comp_opt, comp_cov = fit_center(data[:, :, :, i], u)
        centers[:, i] = get_center(comp_opt)

    return centers
```

B.4 visualize.py

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

import peak_finding


def plot_xyz_psd(data, u, comp_opt, comp_cov):
    """Plots the total of slices perpendicular to x, y and z, and
    then overlays the gaussian model with the provided paramters.

    Args:
        data (np.ndarray): The signal power density. Should be
            3D of shape (nx, ny, xz).

        u (np.ndarray): An array of values for the x, y and z axes.

        comp_opt (list): 3 long list of lists of parameters for
            gaussian_model to be plotted.

        comp_cov (list): 3 long list of covariance matrices for
            parameters for gaussian_model to be plotted.
    """
    # High resolution span over u
    u_hr = np.linspace(u[0], u[-1], 256)

    over = [(0, 2), (1, 2), (0, 1)]
    labels = ['$k_x$', '$k_y$', '$k_z$']

    center = peak_finding.get_center(comp_opt)
    center_unc = peak_finding.get_center_unc(comp_cov)

    fig, axs = plt.subplots(3, figsize=(8, 6))
    fig.tight_layout()

    for n, ax in enumerate(axs):
        ax.set_xlabel(labels[n])
```

```
        ax.set_ylabel('PSD')
        ax.set_xlim(u[0], u[-1])
        ax.plot(u, np.mean(data, axis=over[n]))
        ax.plot(u_hr, peak_finding.gaussian_model(u_hr, *comp_opt[n]))

        center_label = r'$k_{0i} = ' + \
            r'{:.3f} \pm {:.3f} $'.format(center[n],
                                          center_unc[n])

        ax.annotate(center_label,
                    (center[n],
                     peak_finding.gaussian_model(center[n],
                                                 *comp_opt[n])),
                    (30, 0),
                    arrowprops={'arrowstyle': '->'},
                    textcoords='offset pixels')

    fig.savefig('figures/axes_fit.pdf', bbox_inches="tight")


def plot_projections(data, u, center, width):
    """Plot the projection onto the XY, XZ, and YZ planes by summing the
    perpendicular PSD.

    Args:
        data (np.ndarray): The signal power density. Should be
            3D of shape (nx, ny, xz).

        u (np.ndarray): An array of values for the x, y and z axes.

        center (tuple): Tuple of the center frequency (x, y, z)

        width (float): Width to show around that center frequency.
    """
    fig, ax_all = plt.subplots(2, 2, figsize=(6, 6))

    fig.tight_layout()

    fig.delaxes(ax_all[0][1])  # Get rid of the unused 3rd axis

    axs = [ax_all[0][0], ax_all[1][1], ax_all[1][0]]
    n_x_list = [0, 2, 0]
    n_y_list = [2, 1, 1]
    labels = ['$k_x$', '$k_y$', '$k_z$']

    proj_list = [np.sum(data, axis=n) for n in range(3)]
```

```
    high = np.max(proj_list)
    low = np.min(proj_list)

    for n, (n_x, n_y, ax) in enumerate(zip(n_x_list, n_y_list, axs)):
        # ax.set_aspect()
        ax.set_xlim(center[n_x]-width, center[n_x]+width)
        ax.set_ylim(center[n_y]-width, center[n_y]+width)

        ax.set_xlabel(labels[n_x])
        ax.set_ylabel(labels[n_y])

        if n == 0:
            proj_list[n] = proj_list[n].transpose()  # Preserve handedness
        ax.pcolormesh(u, u, proj_list[n], cmap='hot', vmin=low, vmax=high)

    plt.savefig('figures/projected_freq.pdf', bbox_inches="tight")


def record_path(loc_array):
    """Produces a csv and 3D rendering of a path from an
    array of positions.

    Args:
        loc_array (np,ndarray): 3xN array of points in the path
    """
    # Write a csv for the table
    np.savetxt('figures/position.csv',
               loc_array.transpose(),
               fmt='%.3f',
               delimiter=',',
               comments='',
               header="X, Y, Z")

    # Generate 3D figure
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.view_init(30, 35)

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    ax.plot(*(loc_array[i, :] for i in range(3)), marker='o')
    fig.savefig('figures/path.pdf', bbox_inches="tight")
```