Experiment Report: ECE 435/535

Jacob Pratt, Samantha Fink

Winter 2020

Table of Contents

Table of Contents **2**
- **Abstract** **3**
- **Purpose** **3**
- **Materials** **3**
- **Experiment** 3
  - **Acquiring sounds and gating:** **4**
    - Enveloping 4
    - Chirp **4**
  - **Angle of Arrival** **4**
- **Results** **5**
- **Conclusion** **6**
- **Appendix** 7
  - Python Code: 7

Abstract

A SONAR system was created using a Raspberry Pi 3B+ and Matrix Voice hat. Two methods of detection (envelope and FMChirp) were performed on a target 12 m away. Angle of Arrival (AoA) experiments were performed in a long hallway using ODAS open source software. Reflections from the adjacent hallway walls and from the end of the hallway were picked up by the Matrix.

Purpose

To use common off the shelf components to demonstrate a working knowledge of radar and sonar systems. To be able to construct a basic sonar system to show that we can translate the theory of radar and sonar systems to a working small scale model.

Materials

Materials used in this experiment include:

- Raspberry Pi 3B+
- Matrix Voice development board
- Laptop

Software used for this experiment:

- Jupyter Notebook
- Raspbian OS

Experiment

The experiment for the enveloping and chirp were performed in the EB basement lobby. The Matrix Voice station was set up along the western wall behind the stairs. Figure 1 shows the location of the experiment setup.

The target in this experiment was the set of elevators shown in Figure 1, which was measured to be 12 m away. Two adjacent walls to the left and right of the setup were measured to be 9 meters away.

Figure 1: Experiment location

Acquiring sounds and gating:

Enveloping

Pulses with a frequency of 2024 Hz and pulse width of 14.5 ms were played from a laptop speaker and directed at the target.

Chirp

Next, a signal that swept from 2024 Hz to 4048 Hz with a pulse width of 29.6 ms was transmitted at the target.

Angle of Arrival

To determine the angle of arrival, we used ODAS (Open embeddeD Audition System). ODAS is an open source library used to determine sound source localization and tracking. The experiment was performed in the hallway near the EB basement lobby to maximize echo.

A demo specifically for the Matrix Voice was used to estimate AoA. The original code filters out lower volume noises to find the loudest nearby source of sound, which then lights up an LED on the Matrix Voice in the direction of the sound. Figure 2 illustrates the experiment setup.

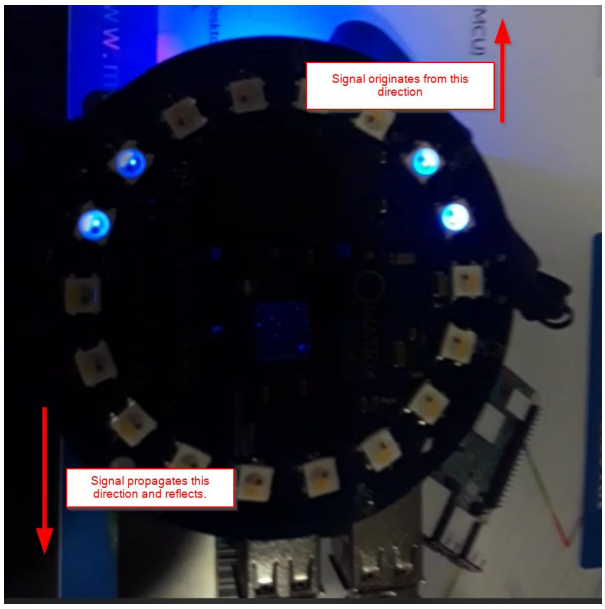

Figure 2: AoA explanation

For the purposes of this project, the code was modified to filter out any sounds over a specific volume so as to not catch the transmitted signal. The gain on the microphones was increased to catch more low volume sounds.

Results

The results of the envelope detection experiment are shown in Figure 2. With 10 samples, a return was detected at 9.7 m away (about a 19% error). It's likely that the signal was returned from the adjacent walls instead of the elevator doors.

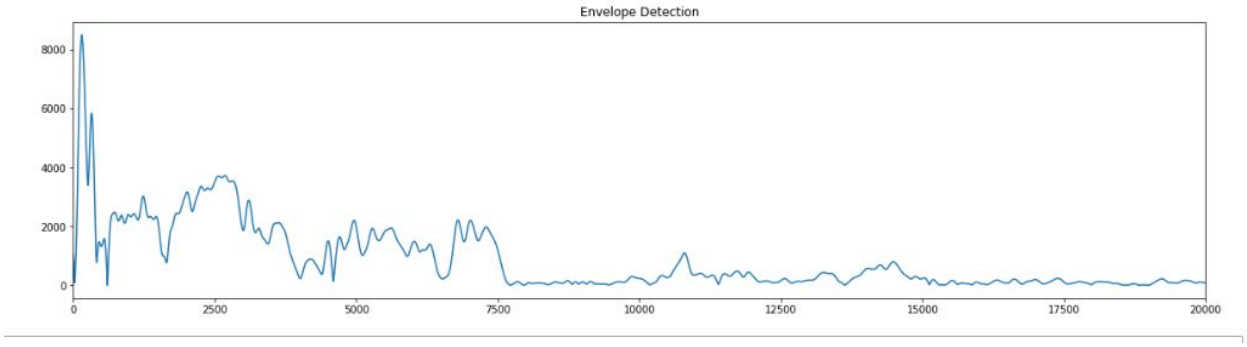

Figure 2: Envelope detection results

Figure 3 shows the FMChirp results. Also with 10 samples, the return was detected at 13.2 m away (a 10% error).

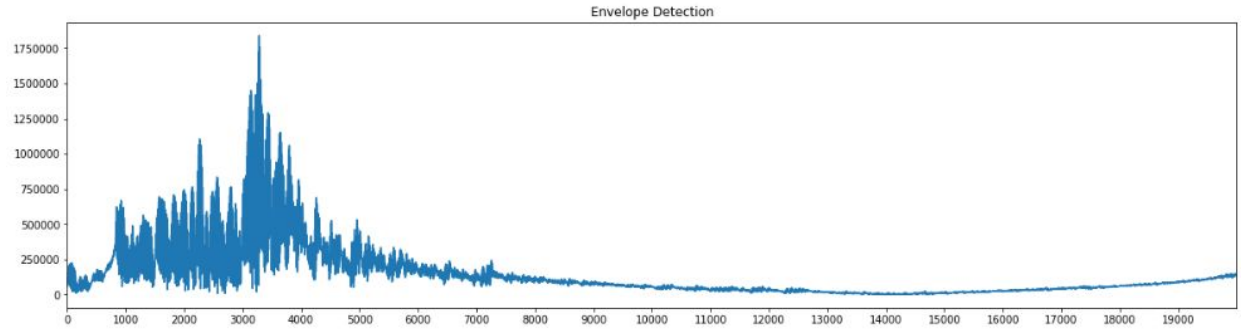

Figure 3: FMChirp detection results

For the AoA, we noticed a large amount of reflection from the adjacent walls. The LEDs that were 90° from the signal source would light up, then light up sequentially as the signal propagated down the hallway and reflected off the adjacent walls until the LED 180° from the transmitted sound would light up.

We did notice that the Matrix picked up environmental noise, specifically from the building's HVAC system. Should this experiment be reproduced in the future, it might be beneficial to either decrease the microphone gain or move to a quieter environment.

Conclusion

The small scale SONAR system was successful. Major errors present during testing were likely the result of the testing environment (i.e. indoors). Additional improvements can be made to the AoA code in order to improve results.

Appendix

Python Code:

```
# Processing an audio file to find sonar returns.
# Written by:
# Jake Pratt
# Samantha Fink

import numpy as np
import matplotlib.pyplot as plt
import wave
import sys
import struct
import scipy
import scipy.signal
import scipy.io
import scipy.io.wavfile
import math
import sklearn.preprocessing as sk

%matplotlib inline

rawAudio = wave.open('output.wav', 'r')
unitSignal = scipy.io.wavfile.read('UnitChirp_2k-4k.wav')
#kernel = wave.open('UnitChirp_2k-4k.wav', 'r')

unitSignal = unitSignal[1]

print(type(unitSignal))
print(f"UnitSignal = {unitSignal}")
kernel = np.trim_zeros(unitSignal, 'fb')


# Extract Raw Audio from Wav File

signal = rawAudio.readframes(-1)
#print(signal)
signal = np.frombuffer(signal, dtype = "int16")
#signal = struct.unpack('h', 0)

# Use the below to get rid of the weird negative peak at the begining.
```

```
#signal = signal[340:]

# If Stereo
if rawAudio.getnchannels() == 2:
    print("Just mono files")
    sys.exit(0)

signal = signal[500:]
plt.figure(1)
plt.title("Raw Signal Return")
plt.plot(signal)
plt.show()
# Find Peaks to determine the PRI and how large of a Gate to use.
audioPeaks = scipy.signal.find_peaks(signal, height = 10000, distance
=2500)[0]

plt.figure(1)
plt.title("Signal Return with Peaks Shown")
plt.plot(signal)
plt.plot(audioPeaks, signal[audioPeaks], "X")
#plt.scatter(audioPeaks)
plt.show()

print(audioPeaks)
# Shift Signal so that the first peak occures at t = 0.
rolledSignal = signal[audioPeaks[0]:]

# Take the Absolute value of the signal
rolledSignal = abs(rolledSignal)
plt.figure(1)
plt.plot(rolledSignal)
plt.show()
# Figure out the average PRI from the average delta between peaks.
added = 0
print(audioPeaks)

for i in range((len(audioPeaks))):
    if i == 0: continue
    diff = (audioPeaks[i] - audioPeaks[i-1])
    added = added + diff
    #print(i)

averagePRI = added / (len(audioPeaks) - 1)
```

```
#print(f"Average Time Samples Between Peaks: {averagePRI}")
#print(averagePRI)
averagePRI = int(round(averagePRI, 0))
print(round(averagePRI))
# Starting to Gate:

gatedSample = []
#averagePRI = 25059
gateLengths = np.arange(0,(averagePRI*10), averagePRI)
print(gateLengths)


for i in range((len(gateLengths)-1)):
    print(i)
    gatedSample.append(rolledSignal[gateLengths[i]:gateLengths[i+1]])

gatedSample = np.array(gatedSample)

plt.figure(figsize=(20, 5))
plt.title("1 Gated Sample")
#plt.semilogy(gatedSample[0])
plt.plot(gatedSample[1])
plt.xlim(0, 20000)
plt.show()
#print(f"gatedSamples = {gatedSample}")
# Normalizing
normalizedSample = sk.normalize(gatedSample, norm = 'l2')

plt.figure(figsize=(20, 5))
plt.title("Normalized Samples")
plt.semilogy(normalizedSample)
#plt.plot(normalizedSample)
plt.xlim(500, 20000)
plt.show()
# Adding the samples.

multiSample = np.sum(gatedSample, axis = 0)
plt.figure(figsize=(20, 5))
plt.title("Summed")
#plt.semilogy(gatedSample[0])
plt.plot(multiSample)
plt.xlim(0, 20000)
plt.show()
```

```
#print(type(multiSample))
print(f"Kernel = {kernel}")
print(np.shape(multiSample))
print(np.shape(kernel))

# Filtering
#b, a = scipy.signal.butter(5, [1500, 2500], btype = 'band')
fs = 44100
lowcut = 1800
highcut = 4400
order = 5

nyq = 0.5 * fs
low = lowcut / nyq
high = highcut / nyq
b, a = scipy.signal.butter(order, [low, high], btype='band')

filteredSample = scipy.signal.lfilter(b, a, multiSample)

correlatedSignal = abs(scipy.signal.correlate(filteredSample, kernel,
'full'))


plt.figure(figsize=(20, 5))
plt.title("XCorr")
plt.plot(correlatedSignal)
plt.xlim(0, 20000)
plt.xticks(np.arange(0, 20000, 1000))
plt.show()
import scipy.signal


# Envelope Detection

signalAnalysis = scipy.signal.hilbert(correlatedSignal)
envelopeDetection = abs(signalAnalysis)

plt.figure(figsize=(20, 5))
plt.title("Envelope Detection on Cross Correlation")
plt.plot(envelopeDetection)
plt.xlim(0, 20000)
plt.xticks(np.arange(0, 20000, 1000))
plt.show()
```

```
# Distance Math
# Sample rate of 44100 Samples per Second
# 43 feet to target.
peak = int(input("Where is a peak observed? (sample number) "))
v = 343 #m/s
feet = 43
meters = feet * 0.3048
meters

secPsample = 1/44100
mPsample = secPsample * v
distanceM = (mPsample*peak)/2 # "There and Back Again" -Bilbo Baggins
distanceM



# Generating an FM Chirp.
# This is based on the Jupyter Notebook provided by Prof. McGarvey, but
modified to produce a chirp instead of a simple pulse.

# Setup Notebook
%matplotlib inline
from matplotlib import pyplot as plt
import numpy as np
import scipy
import scipy.signal
# Setup Parameters
DEBUG = False
#___|Center Freq = fc |_____________________________|Center Freq = fc
|_________________
#   [<------- Pulse Repetition Interval = PRI   ---->]
#   [<--Pulse Width-->]
amplitude = np.info(np.int16.max)
A = amplitude
fc1 = 2024 # 4kHz kilohertz
fc2 = 4048
PRI = 0.5 # seconds

# Class decided on number of Periods or wavelengths in a pulse
num_periods = 60
Tc = 1/(fc1)
PW = Tc*num_periods  # 25ms Pulse Width
```

```
print('='*20)
print("Time and Frequency")
print('PRI:{} s, PW {}s, Fc1 {}, Fc2, Hz, Tc {} s'.format(PRI, PW, fc1, fc2,
Tc))
print('\n')

# Spatial information
vp = 343 # m/s speed of sound
PRI_x = PRI * vp
PW_x = PW * vp
kc = fc1/vp
Lc = vp/fc1

print('='*20)
print("Space and Time")
print('PRI:{:.02f} m, PW {:.02f} m, kc {:.02f} cycles/meter, lambda {:.02f}
m'
      .format(PRI_x, PW_x, kc, Lc))
# Making the data pretty
import pandas as pd
data1 = [[PRI, PW, fc1, Tc],[PRI, PW*1e3, fc1/1e3, Tc*1e6]]
pd.DataFrame(data1[1:], columns=["PRI (s)", "PW(ms)", "fc (kHz)", "Tc
($\mu$s)"])
data2 = [[PRI_x, PW_x, kc, Lc],[PRI_x, PW_x, kc, Lc]]
pd.DataFrame(data2[1:], columns=["PRI (m)", "PW(m)", "kc (cycles/meter)",
"$\lambda$ (m)"])
# samples per period
#samps_per_period = 10
FS_MAX = 80000 # maximum sample rate for audio transmitter

# create the time step for sampling
fmax = fc2
Tmax = 1/fmax
dt = Tmax/20

# Optional: create a waveform with dt/[2,3,4,5,6,7,8,9,10,15,20] and plot the
FFTs of each
# You should see what happens to the energy when the signals are reproduced
cleanly.

# calculate required sample rate
fs = 1/dt
derp = fs < FS_MAX
```

```
print('Desired Sample Rate : {} samples/sec'.format(fs))
print('Sample Rate Check: is {} less than {} == {}'.format(fs, FS_MAX, derp))

if derp == False:
    print('\n ***** Desired Sample Rate too high! ***** ')
else:
    print(' Desired Sample Rate usable')
# create the time series array
t_unit = PRI
t_vector = np.arange(0,t_unit,dt)
print('Samples in unit vector {}'.format(len(t_vector)))
plt.plot(t_vector)
plt.xlabel('index')
plt.ylabel('seconds')
plt.title('Time Series Vector')

# checking to see if we are creating it correctly
print(PW)
## Making PW the iteration way
# now we are going to create a mask for bits that will be on or off
# create a single pulse train
# 111111000000000

#PW = 0.06

mask = np.zeros_like(t_vector)
sample = t_vector[0]
idx = 1
while sample < PW:
    mask[idx] = 1
    idx = idx+1
    sample = t_vector[idx]

plt.plot(t_vector[:len(mask)],mask)
plt.xlabel('seconds')
plt.ylabel('True/False')
plt.title('Mask for Pulse Train PW = {}s'.format(PW))
## Create mask vector method
# figure out how many samples are in the pulse
PWidx = np.int(PW/dt)

mask_vector = np.zeros_like(t_vector)
mask_vector[:PWidx] = 1
```

```
mask_vector[0]=0 # makes the pulse nice.

plt.plot(t_vector,mask_vector)
plt.xlabel('seconds')
plt.ylabel('True/False')
plt.title('Mask for Pulse Train PW = {}s'.format(PW))
# Create the sine wave
import scipy

amplitude = np.iinfo(np.int16).max
A = amplitude
print(fc1)
#data = amplitude * np.sin(2. * np.pi * freq * t)
#signal = A*np.sin(2. * np.pi*np.int(fc)*t_vector)

chirpSetup = np.linspace(0 , PW, 4048)

signal = scipy.signal.chirp(chirpSetup, fc1, PW, fc2)

plt.plot(t_vector[0:PWidx//200], signal[0:PWidx//200],'o-')
#plt.plot(t_vector[:PWidx], signal[:PWidx]) # zeros are optional
plt.title('Fc {} kHz    For the duration of the pulse'.format(fc1/1e3))
plt.xlabel('time (s)')
plt.ylabel('Amplitude')
signal[:20]
while len(signal) < len(mask):
    signal =  np.append(signal, signal)

signal = signal[0:40480]
print(np.shape(signal))
print(np.shape(mask))
# now combine everything
signal = signal * mask
plt.plot(t_vector,signal)
plt.title('Pulse Train Unit')
plt.xlabel('time (s)')
plt.ylabel('Amplitude')
# Now we make a long time series of these
t_max = 60+.01 # 5 min the +3 is to show rounding

periods_to_copy = t_max / PRI
print('Periods to copy {}'.format(periods_to_copy))
```

```
periods_to_copy = np.int(np.ceil(periods_to_copy))
print('Periods to copy {}'.format(periods_to_copy))

unit_signal = signal
signal = None
signal = unit_signal
for idx in range(0,periods_to_copy):
    signal=np.concatenate((signal, unit_signal), axis=None)

t_max_idx = len(signal)
t_vector = np.arange(0,len(signal)*dt,dt)
if DEBUG == True:
    plt.plot(t_vector, signal, 'xkcd:mango')
    plt.title('Pulse Train')
    plt.xlabel('Seconds')
    plt.ylabel('Amplitude')

print(len(signal)*dt)
plt.plot(signal[:72000])
from scipy.io.wavfile import write
samplerate = 44100;
'''
#freq = 100
t = np.linspace(0., 1., samplerate)
amplitude = np.iinfo(np.int16).max
print(amplitude)
data = amplitude * np.sin(2. * np.pi * freq * t)
write("example.wav", samplerate, data)
print(data[:20])
'''

# Write out the waveform
print(signal[0:20])
samplerate = np.int(fs)
print(fs)
write("FMChirp2024-4048.wav", samplerate, signal)
write("UnitChirp_2k-4k.wav", samplerate, unit_signal)
print(signal[0:20])
```