Sonar Ranging with Raspberry Pi and MATRIX Voice

Introduction:

Sonar can be used to determine the range of a target object by transmitting a known signal and recording properties of its return. A frequency shift might indicate its speed from a known angle, multiple microphones might allow a time-difference-of-arrival (TDOA) process to be done so the direction of the target from the transmitter can be known, or more simply, the time it takes to receive a reflection from the target can be used to determine its distance from the transmitter. The latter of these was built and demonstrated to give the approximate range of targets, which seems to be accurate to a few meters.

Hardware Setup

Transmitter

A large amount of gain in the transmission will allow for a stronger return to be seen. This is crucial for some methods of detection to distinguish an echo above the ambient noise of a test environment. A JBL Micro Wireless speaker was used as a transmitter, since it has a 3W output – considerably more than any available laptop or tablet speakers on hand. Using a Lysol wipes container and some foam sheets, a crude method of directing the output was taped to the Bluetooth speaker (Figure 1) so any multipath effects might be minimized, although its effectiveness at doing so was not tested thoroughly.

Figure 1: 3W Bluetooth speaker with directional gain

Data Capture and Processing

The Raspberry Pi and MATRIX Voice hardware combo allowed for up to 8 microphones to capture echoes, but collecting all 8 channels proved challenging in the limited time available to complete this work, and the ability of the Raspberry Pi seemed to be strained by processing only one channel. As a result, only one microphone was used to collect reflected echoes from targets.

Software Setup

Python Growing Pains

Learning Python was a challenge to overcome for this work. As a result, the architecture of ProcessWav.py is likely crude or otherwise in need of cleanup.

Initially, the intention was to import a pre-recorded .wav file with PyAudio and process from there, so the main program name is ProcessWav.py, although this approach was never actually used. A combination of pythonaudiocollect.py and the RadarSim (in Jupyter Notebook) was used as a basic framework for envelope detection, then cross-correlation was added after initial ranging was verified to work. Later, the enveloping and cross-correlation sections were moved to their own functions in an attempt to organize and speed up troubleshooting without having to scroll through so much.

The full text of ProcessWav.py, the enveloping function, and the cross-correlation function are appended to the end of this report for reference. This code is heavily commented with its intended function, and some major sections of troubleshooting code are commented out to expedite processing time, since several plots of intermediate steps tend to slow that down. A simplified description of our process can be discussed here.

Audio Collection

Very little of pythonaudiocollect.py was changed at all. Since the method of its operation was well outside the scope of this work, the only significant change was the removal of the live-streamed audio to a plot.

Enveloping and Filtering

RadarSim worked so well in Jupyter Notebook that it seemed the most obvious first step in enveloping a real-world signal. A 7th-order bandpass filter centered on the carrier frequency was added in RadarSim to drop the noise floor by about 7 dB. In sonar, our carrier frequency was much lower than the 40 MHz RadarSim carrier, and this 7th-order filter was not cooperating with a 1 kHz tone, so it was reduced to a 1st-order filter, which still had the effect of dropping the noise floor by a significant margin. A plot was added (but later commented out) to show this effect.

In RadarSim it was encouraged to avoid using the peaks() function available from the scipy Python module. A secondary filter was used to smooth the noisy pulses for the bandwidth frequency, ideally leaving one sample that can be identified per peak as the top. Due to this second filter, the tops of the peaks were attenuated severely, but the only pertinent data was the time sample of the peak, not its amplitude. By dividing the sample number by the sample rate, it's simple to find the time at which the reflection was recorded. By multiplying the speed of the wave through the transmission medium (in this case, the speed of sound in air) and dividing by 2, the distance to the target can be approximated with a resolution determined by the width of the pulse transmitted.

$$\frac{sample\ number}{samples/_{sec}} = \frac{n_t}{f_s} = time\ of\ sample\ (t_s)$$

$$\frac{t_s * v_p}{2} = \frac{t_s * 344\,{}^{m}/_{sec}}{2} = distance\ to\ target\ (m)$$

Cross-correlation

A challenge of using the enveloping and filtering method was that the amplitude of the echoed returns was so small that any measurable peaks were almost indistinguishable from the noise, even post-filtering. Cross-correlation allows for that small amplitude to be normalized and easily detectable since the amplitude of the echo is not relevant to how a cross-correlation works. It has the added benefit of improving resolution, despite an increase in pulse width, which was done to produce a chirp that sounded (to human ears) more correct than a shorter pulse could produce.

Using the "Burdell Generate Pulse Train" Jupyter Notebook code, a single line was commented out and replaced with an FM chirp using scipy.signal's chirp() function. The chirp was also used as the kernel for performing the cross-correlation in the function.

```
#signal = A*np.sin(2. * np.pi * freq * t)
signal = A * signal.chirp(t_vector, f0 = fc/4, f1 = fc*4, t1 = PRI, method='linear')
```

This allowed testing of both envelope detection (with tones) and cross-correlation detection (with chirps) by commenting one line out and the other in, so the correct transmit signal could be produced for a given detection method.

A simple menu selection was added to ProcessWav.py so a user can decide which detection method to use, although cross-correlation produced a response that was the easiest to read, which made enveloping almost obsolete.

Testing and Results

The long hallway outside the ECE office (Figure 2) was expected to have acceptable acoustic properties, with the added bonus of available electrical plugs to power the equipment needed to test. Although it was a narrow passage, it has a large window which was expected to produce a strong echo and the hallway was nearly empty aside from a large trashcan about halfway between the transmitter and the window. The length of transmission (one-way) was estimated to be about 30m. Multipath was expected to be somewhat of an issue from the hard sides of the walls, but the ceiling and floors are materials that were assumed to dampen acoustic noise and somewhat reduce multipathing as compared to testing in an entirely concrete environment. Also, it was cold outside and the testers are wimps.

Figure 2: Testing Hallway

Three clear peaks in the cross-correlation can be seen (Figure 3). The return at almost 20m is expected to be the trashcan, and the one at almost 30m should have been the window. The 40m return is a mystery, and we're not sure if it can be attributed to some multipath event. The sonar appears to work, although with some further mysteries that our time-budget could not be spent towards solving.

It's also possible that a multipath even did not occur and that the 20m pulse is the initial transmission (perhaps the roll() function was misbehaving?) and the 30m and 40m are the trrash can and the window, respectively. A long enough tape measure was not available to confirm this, but the distance to the window from the test site seemed to be further than 20m so this possibility seems less likely.

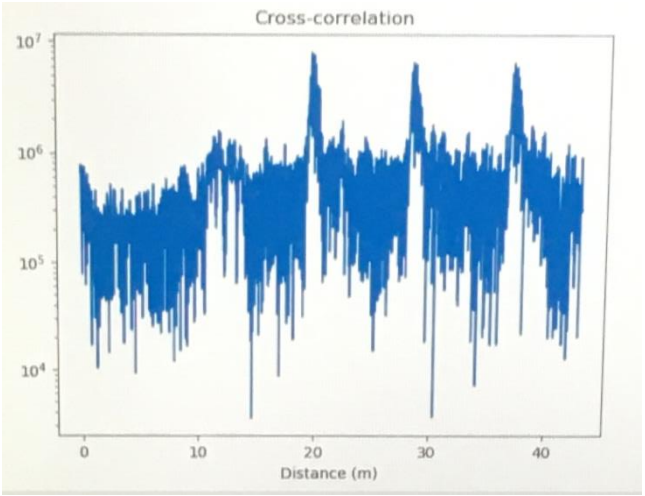

Figure 3: Cross-correlated Range Results

Conclusion

Given more available time to work on this, both testers are confident that this system could be optimized for a specific location. For a general-purpose sonar device, there may be less opportunity for optimization, but this design is acceptable for low-resolution ranging, accurate within a meter or so.

Cross-correlation proved far superior to enveloping, and unexpectedly easier to implement as well. TDOA and beamforming for directionality measurements would be have been exciting to add, but with such little time left before the deadline, the designers were uncomfortable adding these large features to an already-rushed project. Any sort of Doppler testing was rejected for similar reasons, as well as a reliable environment that could accommodate this kind of test safely.

Ultimately the work produced was functional and resolution was greatly improved by cross-correlating an FM chirp. The work was engaging and became very exciting when results started making sense.

Appendix 1: ProcessWav.py code

```
import detection as det
print('Imported Detection.py')
import butterworth as butt
print('Imported Butterworth.py')
import pyaudio
print('Imported PyAudio')
import os
print('Imported OS')
import numpy as np
print('Imported NumPy')
from matplotlib import pyplot as plt
print('Imported PyPlot from MatPlotLib')
import scipy
print('Imported SciPy')
from scipy import signal as sig
print('Imported Signal from SciPy')
from scipy.signal import butter, lfilter
print('Imported Butter and LFilter from SciPy.Signal')
plt.switch_backend('Qt4Agg')
import enveloping as env
print('Imported enveloping.py')
import cross_correlate_function as ccf
print('Imported xcorr.py')
#---------------------------------------------------------------------
#---------------------------------------------------------------------- MAIN SETUP
#---------------------------------------------------------------------
fc = 2000          # Carrier Frequency, Center Frequency
vp = 344           # Phase Velocity of the wave
T  = 1/fc          # period of one Carrier Frequency
#derived values
Lambda = vp/fc

# Setup Time portion
PRF = 4        # Pulses per second (hertz)
PRI = .25        # Pulse Repetition Interval (seconds)
R_unamb = PRI *vp/2 # Unambiguous Range

#Num cycles per pulse packet
k = 100           # k cycles of fc in the pulse packet
PW = k*T          # k cycles * Period of fc
BW = 1/PW         # Bandwidth of the RADAR Pulse
K_pulses = 20
dt_k = 40
#---------------------------------------------------------------------
#---------------------------------------------------------------------- PYAUDIO
#---------------------------------------------------------------------
CHUNK = 2048
FORMAT = pyaudio.paInt16
CHANNELS = 1
RATE = 44100 #samples per second
fs = RATE
RECORD_SECONDS = K_pulses*PRI

#WAVE_OUTPUT_FILENAME = "recording.wav"
```

```
p = pyaudio.PyAudio()
stream = p.open(format=FORMAT, channels=CHANNELS,rate=RATE,input=True,frames_per_buffer=CHUNK)

#clear display of ALSA errors
clear = lambda: os.system('clear')
clear()

print("Recording for {} seconds...".format(round(RECORD_SECONDS,1)))

frames = []
for i in range(0, int(RATE/CHUNK * RECORD_SECONDS)):
    audio_in = stream.read(CHUNK)
    frames.append(audio_in)

stream.stop_stream()
stream.close()
p.terminate()
print("Complete!")
print('')
#---------------------------------------------------------------------
#---------------------------------------------------------------------- RAW DATA MANIPULATION
#---------------------------------------------------------------------
t_vector = np.arange(0,PRI,1/fs)
len_PRI = len(t_vector)

frames = np.array(frames)
audio_in = np.frombuffer(frames, dtype=np.int16)
len_audio = len(audio_in)
len_env = len_PRI*(K_pulses)

#the length of the audio doesn't match the reshaping of the envelope detection
#this fixes that, but it's a janky solution
if len_audio < len_env: #if the audio length is too short, pad it with the median value of the audio
    audio_in = np.pad(audio_in, int(((len_PRI*(K_pulses)-len_audio)/2)), mode = 'median')
else: #if the audio length is too long, remove samples from the beginning where there's a weird artifact anyway
    audio_in = audio_in[len_audio-len_env::]

nsamps = len_env
'''
plt.semilogy(audio_in)
plt.title('Raw Audio Input')
plt.xlabel('sample number')
plt.ylabel('Power (dB)')
plt.show()
'''
print('**Tx Parameters Input OK')

#---------------------------------------------------------------------
#---------------------------------------------------------------------- FILTER RAW SIGNAL FOR fc
#---------------------------------------------------------------------

#Tx pulse detection to shift Tx to left side of plots (happens at ~600th sample)
roll = 0
for idx in range(1000,nsamps):
    if audio_in[idx] > 1.5*10e4:
```

```
        roll = idx
        break

print('What type of analysis?')
print('1. Cross Correlation (takes a long time)')
print('2. Envelope Detection (low resolution)')
cc_or_env = int(input('Enter Selection: '))

#make a blank list and add in each peak as these conditions are met (note: Tx peak will be first pulse detected)
#maxima = list()

if cc_or_env == 1:
    fc_BP_filt_start = (0.25-0.1)*fc
    fc_BP_filt_stop = (4+1)*fc
    filt_trace = butt.butter_bandpass_filter(audio_in, fc_BP_filt_start, fc_BP_filt_stop, fs, order=1)
    print('**Filtered audio for {} Hz to {} kHz'.format(fc_BP_filt_start, fc_BP_filt_stop/1000))
    print('')

    filt_trace_roll = np.roll(filt_trace,-roll) #roll the filt_trace using the above roll NEW
    filt_trace_reshape = filt_trace_roll.reshape(K_pulses, len_PRI) #reshapE into an array of K_pulses by len_PRI #NEW
    filt_trace_sum = filt_trace_reshape.sum(axis=0) #filt_trace summed up together #NEW

    x = np.linspace(0,R_unamb, len(filt_trace_sum))
#     plt.semilogy(x,filt_trace_sum) #NEW
#     plt.xlabel('Distance (m)')
#     plt.title('Filtered/Gated/Summed Signal')
#     plt.ylim(10e2, 10e5)
#     plt.show() #NEW

    print('')
    print('**Cross-correlating, please wait...')
    c_c = ccf.cross_correlation(filt_trace_sum, PRI) #changed to cross_correlation of filt_trace_sum
    print('Cross-correlation complete!')

    c_c_env = det.envelope(c_c)

    print('Cross-correlated Ranging')
    plt.semilogy(x,c_c_env)
    plt.title('Cross-correlation')
    plt.xlabel('Distance (m)')
    #plt.ylim(10e2, 10e5)
    plt.show()

    #triggerdB = int(input('Input Threshold (dB): '))
#     #trigger = 10**((triggerdB)/10)
#
#     for idx in range(1, len(c_c)-1):
#         if c_c[idx] > c_c[idx-1] and\
#         c_c[idx] > c_c[idx+1] and\
#         c_c[idx] > trigger:
#             maxima.append(idx)


elif cc_or_env == 2:
    # Filter signal for fc
    fc_BP_filt_start = (0.75)*fc
```

```
    fc_BP_filt_stop = (1.25)*fc
    filt_trace = butt.butter_bandpass_filter(audio_in, fc_BP_filt_start, fc_BP_filt_stop, fs, order=1)
    print('**Filtered audio for {} Hz to {} kHz'.format(fc_BP_filt_start, fc_BP_filt_stop/1000))
    print('')
    # Envelope
    filt_env, n_obs_main_trace_env = env.enveloping(audio_in, K_pulses, len_PRI, filt_trace, R_unamb, BW, fs)
    #triggerdB = int(input('Input Threshold (dBm): '))
    #trigger = 10**((triggerdB-30)/10)

#     for idx in range(1, len(filt_env)-1):
#         if filt_env[idx] > filt_env[idx-1] and\
#         filt_env[idx] > filt_env[idx+1] and\
#         filt_env[idx] > trigger:
#             maxima.append(idx)

#---------------------------------------------------------------------
#--------------------------------------------------------------------- PEAK DETECTION
#---------------------------------------------------------------------

#this is an array of all the sample numbers where a peak occurs above the threshold.
#maxima = np.array(maxima)


#---------------------------------------------------------------------
#--------------------------------------------------------------------- DISPLAY OUTPUT
#---------------------------------------------------------------------
#establish the noise floor for later calculations
noisefloor = np.average(filt_trace)
noisefloordBm = np.average(10*np.log10(np.abs(filt_trace)/1e-3))

#clear display for neato readout
clear = lambda: os.system('clear')
clear()

# #headers for neato readout
# print('Noise Floor at {} dB'.format(round(noisefloordBm,1)))
# print('{} Targets found above {} dB threshold:'.format(maxima.size-1,triggerdB))
# print('')
# print('     Range     Amplitude    Detection Positivity')
#
# #average this number of samples before & after each detected peak to reduce impact of unexpected minima in the envelope
waveform
#
# for idx in range(1, len(maxima)): #note: start this index at 1 (not 0) to skip over the Tx pulse
#     T_Range = vp*(maxima[idx]/fs)/(2) #range in m
#     T_Power = np.average(n_obs_main_trace_env[(maxima[idx]-dt_k):(maxima[idx]+dt_k)])
#     T_PowerdBm = 10*np.log10(np.abs(T_Power))
#     SNRpeak = T_Power/noisefloor
#     SNRpeakdB = 10*np.log10(SNRpeak)
#     T_Conf = np.round(100*(1-np.exp(-SNRpeak*np.sqrt(K_pulses))),1)
#
#     print('{: >2}: {: >7} m |{: >7} dB |{: >7}% confidence'.format(idx, np.round(T_Range,2), np.round(T_PowerdBm,2), T_Conf))
```

Appendix 2: Enveloping Function

```
import detection as det
print('Imported Detection.py')
```

```
import butterworth as butt
print('Imported Butterworth.py')
import pyaudio
print('Imported PyAudio')
import os
print('Imported OS')
import numpy as np
print('Imported NumPy')
from matplotlib import pyplot as plt
print('Imported PyPlot from MatPlotLib')
import scipy
print('Imported SciPy')
from scipy import signal as sig
print('Imported Signal from SciPy')
from scipy.signal import butter, lfilter
print('Imported Butter and LFilter from SciPy.Signal')
plt.switch_backend('Qt4Agg')
#-----------------------------------------------------------------
#----------------------------------------------------------------- ENVELOPE, GATE, & SUM (RAW SIGNAL)
#-----------------------------------------------------------------
def enveloping(audio_in, K_pulses, len_PRI, filt_trace, R_unamb, BW, fs):
    # Envelope detect the signals
    main_trace_env = det.envelope(audio_in)
    print('**Envelope created for raw audio input')

    nsamps = len(audio_in)
    x = np.linspace(0,R_unamb, nsamps)

    #Tx pulse detection to shift Tx to left side of plots (happens at ~600th sample)
    roll = 0
    for idx in range(1000,nsamps):
        if main_trace_env[idx] > 5e3:
            roll = idx
            break

    main_trace_env = np.roll(main_trace_env,-roll)


    # Gate the signal & sum them up for n observation effects
    n_obs_main_trace_env = main_trace_env.reshape(K_pulses, len_PRI)

    #Display all pulses together before adding (they should line up)
    for idx in range(0, K_pulses):
        plt.subplot(K_pulses+1,1,idx+1)
        plt.semilogy(n_obs_main_trace_env[idx,::])
        #plt.title('Gated Pulses to be Summed')
        plt.ylabel('{}'.format(idx))
    plt.show()

    # add them all together
    n_obs_main_trace_env = n_obs_main_trace_env.sum(axis=0)
    print('**Gated and summed raw audio input')


    #-----------------------------------------------------------------
    #----------------------------------------------------------------- ENVELOPE, GATE, & SUM (FILTERED SIGNAL)
```

```
#-----------------------------------------------------------------

# Redo envelope detection on filtered signal
filt_trace_env = det.envelope(filt_trace)
print('**Envelope created for filtered audio input')

'''
plt.semilogy(x,filt_trace_env)
plt.ylim(10e1, 10e5)
plt.title('Filtered Trace Envelope')
plt.xlabel('samples')
plt.ylabel('Power (dB)')
plt.show()
'''

# Redo gating and summing
filt_trace_env = np.roll(filt_trace_env,-roll)
n_obs_filt_trace_env = filt_trace_env.reshape(K_pulses, len_PRI)
n_obs_filt_trace_env = n_obs_filt_trace_env.sum(axis=0)
print('**Gated and summed filtered audio input')
#build x-axis
nsamps = len(n_obs_main_trace_env)
x = np.linspace(0,R_unamb, nsamps)

#plot the filtered signal over the unfiltered one to show noise reduction
#plt.rcParams['figure.figsize'] = [20, 10]
plt.plot(x,10*np.log10(n_obs_main_trace_env/1e-3), label='Unfiltered')
plt.plot(x,10*np.log10(n_obs_filt_trace_env/1e-3), label='f$_c$-Filtered')
plt.grid(True)
plt.axis('tight')
plt.legend(loc='best')
plt.title('Bandpass Filter Signal for f$_c$ to drop noise floor')
plt.xlabel('Distance (m)')
plt.ylabel('Power (dBm)')

plt.show()

#-----------------------------------------------------------------
#----------------------------------------------------------------- FILTERING GATED/SUMMED ENVELOPES
#-----------------------------------------------------------------
#filter the envelope to smooth out envelope. This will degrade the amplitude but all I need is the sample number for each peak
BW_BP_filt_start = 1e-20 #lazy lowpass
BW_BP_filt_stop = BW*0.75
filt_env = butt.butter_bandpass_filter(n_obs_filt_trace_env, BW_BP_filt_start, BW_BP_filt_stop, fs, order=1 )
return filt_env, n_obs_main_trace_env;
print('**Filtered Detected Envelope for {} to {} Hz'.format(BW_BP_filt_start, BW_BP_filt_stop))

#make a plot for the double-filtered signal used to detect peaks and show the trigger threshold
#plt.rcParams['figure.figsize'] = [20, 10]
plt.plot(x, 10*np.log10(np.abs(filt_env)/1e-3),label='BW-Filtered')
plt.axhline(y=10*np.log10(trigger/1e-3), xmin=0, xmax=len(filt_env), linewidth=1, color = 'r', linestyle='dotted', label='Trigger
Threshold')
plt.grid(True)
plt.axis('tight')
plt.legend(loc='best')
plt.title('Detected Peaks (post-filtering)')
```

```
    plt.xlabel('Distance (m)')
    plt.ylabel('Power (dB)')
    plt.show()


    ##*** STOP FUNCTION
```

Appendix 3: Cross-correlation Function

```
import numpy as np
import scipy
from scipy import signal as sig
from scipy.signal import chirp

def cross_correlation(function, PRI):
    t = np.linspace(0, PRI, 10000)
    kernel = chirp(t,f0=500,f1=8000,t1=PRI, method='linear')
    print('Length of function is {}'.format(len(function)))
    print('Length of kernel is {}'.format(len(kernel)))
    c_c = sig.correlate(function, kernel, mode='same',method='direct')
    return c_c
```