Radar and Sonar
Experiment Report

ECE 435/535: Radar and Sonar
PSU Winter Quarter 2020
Professor: Brian McGarvey

Jens Evans and Sarah Mehler
3/17/2020

Introduction

The purpose of this project was to create a functional SONAR system using the basic principles of RADAR and SONAR. This project was developed in two main steps: simulation and implementation. Through the simulation, we were able to create a complete model of the system from pulse generation to signal detection. Using this model, we were then able to implement the same principles using a raspberry pi based device, which acted as a rudimentary SONAR. The following contains code from both simulation and implementation, as well as a record of the system's performance.

Materials

For the SONAR project, the main materials used were a laptop for the development of code and to play custom audio files on the laptop speaker. We also used a raspberry pi with a Matrix Voice kit as the primary SONAR. Finally, we also bought a parabolic dish to increase the gain and directionality of our SONAR.

Hardware Methods

We set the Raspberry pi at the focus of a parabolic reflector. While the gain of the reflector was not directly measured, using a 5kHz center frequency and assuming that the microphone was within ½ inch of the focus, we calculated approximately 20dB of gain. This calculation was made using:

$$Gain\ dB = 20log_{10}(\frac{3.25 * Diameter * Efficiency\ Factor}{Wavelength})$$

For the speaker, we used a laptop speaker which was approximately an isotropic radiator.

Software Methods

The development of the SONAR was done completely in the Python programming language. Specifically, we used Python 3. The custom audio files were generated in an iPython notebook and the main development was done using Spyder. The full code is included in an appendix, but here we will go over the general flow of the code for data processing.

1. Import Libraries
    a. Pyaudio for audio data processing
    b. Numpy for fast array processing
    c. Matplotlib for graphic display
    d. Scipy for signal processing tools
2. Define Functions
    a. Envelope for smoothing

    b. Bandpass functions for noise reduction
3. Import audio file of chirp train for cross-correlation purposes later.
4. Set variables for data processing
    a. Number of observations helps increase SNR but also takes longer to run and makes data larger so it is harder to process on the pi.
    b. PRI and CHUNKSIZE are determined by the pulse train parameters.
5. Setup the device to listen and record audio data
    a. Initialize portaudio
    b. Listen for as long as it takes to reach the required number of observations
    c. Push that data into a numpy array
    d. Close portaudio
6. Process audio data
    a. Bandpass the return data within the range of the chirp to increase SNR.
    b. Use an envelope function to smooth the data
    c. Slice and dice the data into PRI length chunks.
    d. Store the envelope data for later comparison with the cross-correlated version (optional).
    e. Use the max index to roll all data so that it starts at the peak. The max signal is always at transmission so we want that to be time 0.
    f. Plot the kernel, the envelope of the kernel and the fifth PRI of the microphone data so that we can see all data is looking as it should as we process. We expect to see a chirp and its envelope correctly aligned with each other as well as a big audio spike with a possible small return visible.
    g. Now we cross correlate the kernel envelope with our audio return envelope (implementation of a matched filter). This will increase our SNR again. We chose to cross correlate the envelope instead of the bare signal because the SNR was better.
    h. Sum all of the observations to implement gating
7. Use the processed data to make a human-readable sonar.
    a. Find the max value of the gated data
    b. Roll back the max value to index 0 so the time values are correct.
    c. Find and index the peaks using a threshold function
    d. Set up a distance per index measurement using the speed of sound and the sampling rate.
    e. Print the number of targets and the distance to targets.
    f. Use the distance per index measurement to set up a vector for displaying returns in meters.
    g. Plot the return data with peaks on top of the return data and graphically differentiated in a way that makes sense.

Results

We performed our system tests in a long hallway. The MatrixOne Voice, as well as the laptop playing the chirp, were placed at one end of the hallway. About 6 meters down the hallway, we placed a small table to be used as a second return. Because of the nature of indoor testing, we are actually recording three returns in this test setup: the end of the hallway at 30 meters, the table at 6.2 meters, and the ceiling at 2.5 meters. Figures 1 and 2 contain a visual for the testing environment setup. Figure 3 displays the SONAR return data from the Raspberry Pi, which shows three relatively accurate returns and one additional return. This additional return has appeared because our threshold was set slightly too low in this case.

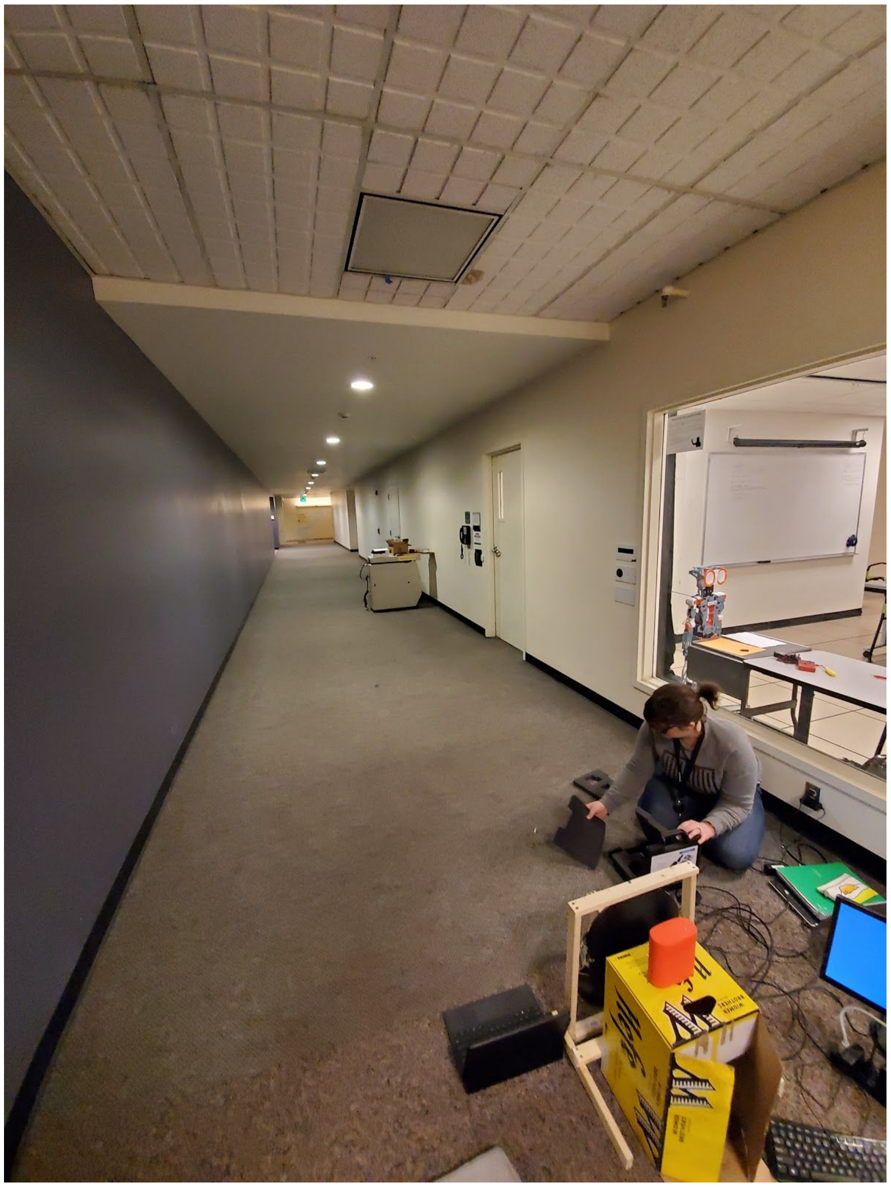

Figure 1: Test Environment Setup 1

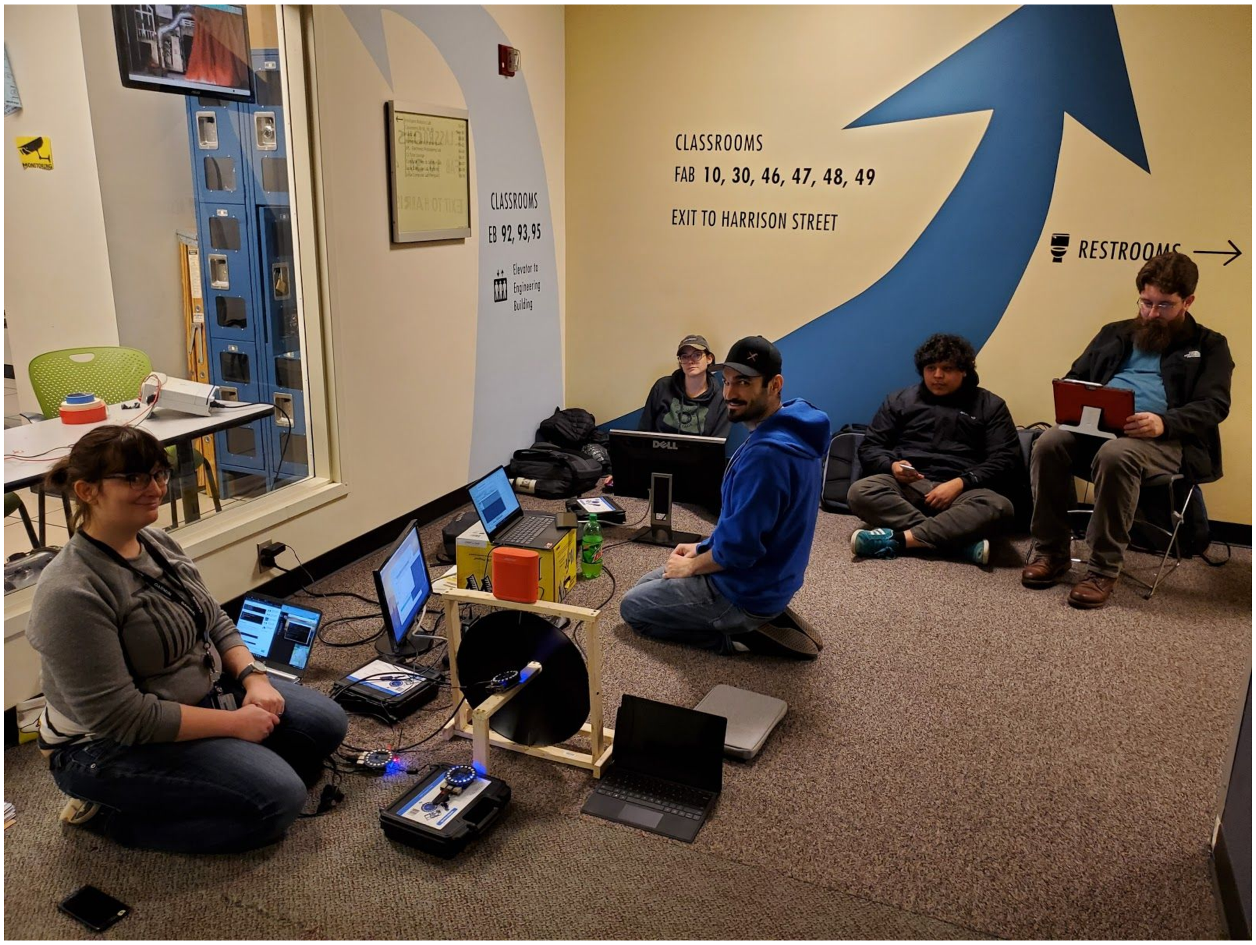

Figure 2: Test Environment Setup 2

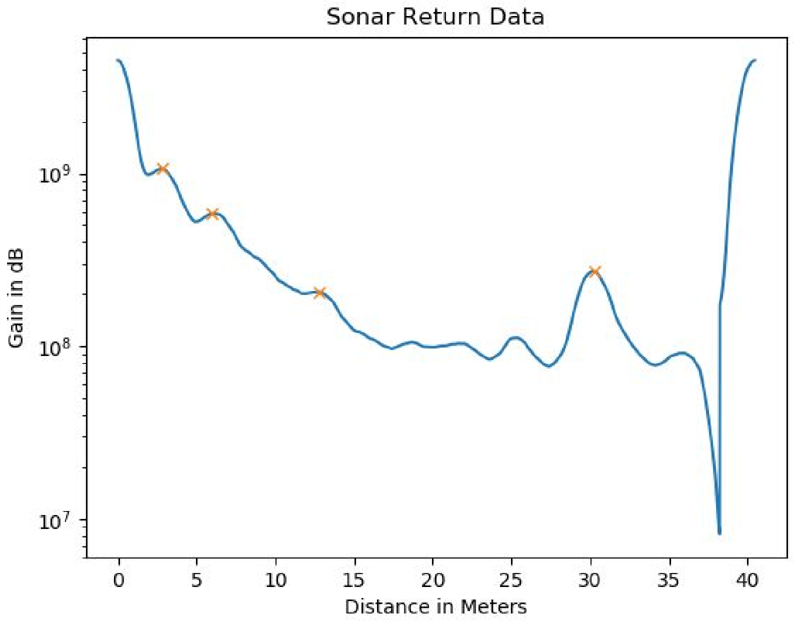

Figure 3: SONAR Test Returns

Discussion

Overall the goal to build a Raspberry Pi SONAR was a success. For the final test, we were able to successfully pick up 3 targets with high accuracy in software. The result was able to be displayed in a way that makes sense to a human, so that it is obvious where a return is in comparison to noise (even without a peak finding algorithm). We were also able to use the concepts of RADAR and SONAR signal processing from the class to greatly improve the signal to noise ratio and help differentiate returns. The RADAR/SONAR concepts that made the biggest difference in our returns were gating, envelope detection, and cross correlation. For future improvements to the project, the first thing we would probably do is add some sort of circulator in hardware or software. The reason for this is that the return graph is dominated by our transmission, so if we shut off the receiver during transmission it would help sharpen the remaining return peaks drastically. The next step would be to use the microphone array to calculate directionality as that is an important feature in modern RADAR systems.

Appendix: Code

RADAR Simulation

```
import matplotlib
matplotlib.use('AGG')
import matplotlib.pyplot as plt
import numpy as np
import scipy.signal as sig

# Custom libraries
import pulsetrain as pt
import pulsenoise as pn
import detection as det

# Student Libary
import lastname as GPD # Intials (George P. Burdell)
```

Student Custom Functions

In [ ]:

```
# Probability of False Detection & Threshold Calculation
def PFalseDetect(DataArray,NumberObservations=1,Certainty=90):
   SignalPower=max(DataArray)
   NoisePower=np.median(DataArray)
   FalseDetect =
100*np.exp(-np.sqrt(NumberObservations)*(SignalPower/NoisePower))
   # Signal Power Required for certainty level
   Threshold =
-np.log(1-Certainty/100)*NoisePower/np.sqrt(NumberObservations)
   print("Probability of False Detect is", format(round(FalseDetect,1)),"%")
   return(Threshold)

# Noise Floor Calculation
def NoiseCalc(DataArray):
   NoiseFloor=np.median(DataArray)
   print("The noise floor is %e" %(NoiseFloor))

# Smoothing Function
def Smoothing(DataArray, Window=10001, PolyOrder=2):
   DataArray = sig.savgol_filter(DataArray, Window, PolyOrder)
   return DataArray
```

```
# Custom Find Peaks
#Done for homework but not used because it runs very slowly compared to
find_peaks especially on the pi.
def CustomPeaks(DataArray, Window=1000, Thresh=5):
    temp=DataArray.copy()
    temp[temp < Thresh*min(temp)] = 0
    temp=sig.argrelmax(temp,order=Window)
    temp=np.array(temp)

    return temp
```

Setup RADAR Parameters

In [ ]:

```
# Basic RADAR Parameters
Pavg = 100e3          # Basic Power level output of the radar
Gt = 15               # Scalar Gain of TX antenna
Gr = Gt               # Scalar Gain of RX antenna  if Gr == Gt same antenna
fc = 40e6             # Carrier Frequency, Center Frequency
vp = 3e8              # Phase Velocity of the EM wave
NF = 1                # Receiver Noise Figure
T  = 1/fc             # period of one Carrier Frequency
Lambda = vp/fc        # Wavelength

# Time Parameters
PRF = 500             # Pulses per second (hertz)
PRI = 1/PRF           # 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

# Error Check
if PW >= PRI:
    print('Error: Pulse width much too long -- PRI: {}, PW = {}'.format(PRI,
PW))
```

Find R_max & R_umabigous

In [ ]:

```
# Calculate maximum range with SNR = 1, Number of Observations = 1
```

```
SNRmin = 1
RCS = 1
Rmax = pt.calcRmax(Pavg,Gt,Gr,Lambda, BW, SNRmin = SNRmin, RCS = RCS) #, RCS,
T, NF = 1,L=1, SNRmin=1)
print('Rmax(SNR:{}, RCS:{}) \t= {:.02f} km'.format(SNRmin, RCS, Rmax/1e3))
print('R unambigouse \t\t= {:.02f}km'.format(R_unamb/1e3))
```

Setup Testing Enviroment

In [ ]:

```
# Number of Targets & Range
num_targets = 10
target_ranges = np.random.randint(Rmax//4,Rmax,num_targets)
target_rcs = np.random.randint(1,1000,num_targets)

# Time Series Constraints
K_pulses = 20       # how many PRI's get simulated
dt_k = 20           # how many samples per fc period (Tc)

# Build Sample Pulse Train & Examine

# Make a signal smaller in amplitude to simulate the blanking / attenuation
in normal RADAR systems
attenuate = True
if attenuate == True:
    dBm = -100        #dBm
    scalar = 1e-3 * np.power(10,(dBm/10))
else:
    scalar = Pavg

main_train, PW, dt, len_PRI = pt.createPulseTrain(A=scalar,fc = fc, k=k,
PRI=PRI, dt_k=dt_k, K_pulses = K_pulses)
```

Create Target Reflections

In [ ]:

```
# Now we create the returns...
main_trace = np.zeros_like(main_train) # return without TX

for idx, target_range in enumerate(target_ranges):

    pwr, dbm = pt.RadarEquationAdv(Pavg, Gt, target_range, RCS, Gr, Lambda,
dB=False)
```

```
    print(':: idx: {} Power at RX {} dBm @ range: {} rmax
{}'.format(idx,(10*np.log10(Pavg/1e-3)),

target_range, R_unamb ))
    p_train, PW, dt, len_PRI = pt.createPulseTrain(A=pwr,fc = fc, k=k,
PRI=PRI,
                                                    dt_k=dt_k, K_pulses =
np.int(K_pulses))
    # time shift to correct spot
    p_train = pt.timeShift(p_train, target_range,vp, dt, len_PRI)
    main_trace = main_trace + p_train
```

Merge Trasmission & Reflection, Add AWGN

In [ ]:

```
# -------------------------------
# now we add the two systems together.
# Add noise to the pulse traing
main_trace = main_trace + main_train
NoiseTest = main_trace
main_trace = pn.addNoiseToPulseTrain(main_trace,1/PW)
RealNoiseFloor=np.mean(main_trace-NoiseTest)
print("Noise Floor is %e" %(RealNoiseFloor))
```

Detection

In [ ]:

```
# Envelope detect the signals
main_trace_env = det.envelope(main_trace)

# Gate the signal & sum them up to provide n observation effects
n_obs_main_trace_env = main_trace_env.reshape(K_pulses+1, len_PRI)

# add them all together
n_obs_main_trace_env = n_obs_main_trace_env.sum(axis=0)

# Optional smoothing function
#n_obs_main_trace_env = Smoothing(n_obs_main_trace_env)
```

Threshold Data

In [ ]:

```
from scipy.signal import find_peaks

# Set Threshold Manually (Not used)
dBm = -100 #dBm
scalar = 1e-3 * np.power(10,(dBm/10))
```

```
#height = scalar

# Find Probability of False Detection & Threshold
Certainty = 90 # Set probability of detection desired to 90%
threshold = PFalseDetect(n_obs_main_trace_env,K_pulses,Certainty)

#Best Peaks Function
peaks, _ = find_peaks(n_obs_main_trace_env, height=scalar, distance=10000)

# Custom Peaks Function
#peaks = CustomPeaks(n_obs_main_trace_env,2000)
#Makes list a countable
#peaks = np.hstack(peaks)
print("Peaks at sample locations: \n", peaks)
```

```
In [ ]:
```

SONAR Using Matrix One

```
import pyaudio
import wave
import numpy as np
from matplotlib import pyplot as plt
#plt.switch_backend('QT4Agg')
from scipy import signal as sig
from scipy.signal import find_peaks
from scipy.signal import butter, lfilter
from scipy.signal import correlate
from scipy.io import wavfile as wfile

def envelope(signal):
    '''
    Fast envelop detection of real signal using thescipy hilbert transform
    Essentially adds the original signal to the 90d signal phase shifted
version
    similar to I&Q
    '''
    signal = sig.hilbert(signal)
    envelope = abs(signal)# np.sqrt(signal.imag**2 + signal.real**2)
    return envelope


def butter_bandpass(lowcut, highcut, fs, order=5):
   nyq = 0.5 * fs
   low = lowcut / nyq
   high = highcut / nyq
   b, a = butter(order, [low, high], btype='band')
   return b, a

def butter_bandpass_filter(data, lowcut, highcut, fs, order=5):
   b, a = butter_bandpass(lowcut, highcut, fs, order=order)
   y = lfilter(b, a, data)
   return y

# Import audio for cross correlation
samplerate, ChirpKernel = wfile.read('C:\\Users\\jense\\ChirpTrain.wav',
mmap=False)


#Number of observations
N = 10
# PRI length in Seconds
PRI=0.236
CHUNKSIZE = int(44100*PRI) # fixed chunk size
rate=44100
# initialize portaudio
p = pyaudio.PyAudio()
stream = p.open(format=pyaudio.paInt16, channels=1, rate=44100, input=True,
frames_per_buffer=CHUNKSIZE)

data = []

# listen
print('Listening...')
x = 0
while x < N:
   values = stream.read(CHUNKSIZE)
   data.append(values)
   x = x+1

# record
data = np.array(data)
MicData = np.frombuffer(data, dtype=np.int16)
stream.stop_stream()
stream.close()
p.terminate()
```

```
print('Done.')

#Process Data
# Envelope Hilbert Transform
MicData = butter_bandpass_filter(MicData, 3.5e3, 6.5e3, fs=44100, order=1)
MicData = envelope(MicData)
#Reshape
MicData = MicData.reshape(N,int(x*CHUNKSIZE/N))

# Store Envelope without Cross Correlation for later Comparison
GatedEnvelope=np.sum(MicData,axis=0)
max_value = max(GatedEnvelope)
max_index = np.argmax(GatedEnvelope)
GatedEnvelope = np.roll(GatedEnvelope,-1*max_index)

# Cross Correlation
# Setup Kernel to same length as PRI
ChirpKernel = ChirpKernel[0:len(MicData[0,:])]
ChirpKernel = np.roll(ChirpKernel,int(len(ChirpKernel)/2))
#Normalize Chirpkernel for a better correlation
ChirpKernel = ChirpKernel*(np.amax(MicData)/np.amax(ChirpKernel))
#Test Point
plt.figure(1)
plt.plot(ChirpKernel)
# Find Envelope of ChirpKernel
ChirpKernel = envelope(ChirpKernel)

# Compares ChirpKernel preenvolope vs envelope vs MicData

plt.plot(ChirpKernel)
plt.plot(MicData[5,:])
plt.show()

# Cross Correlate Signals row by row
x=0
while x < N:
   MicData[x,:] = sig.correlate(MicData[x,:],ChirpKernel,mode='same')/N
   x=x+1



#Gate Data
```

```
GatedData=np.sum(MicData,axis=0)

# Find Max
max_value = max(GatedData)
max_index = np.argmax(GatedData)

GatedData = np.roll(GatedData,-1*max_index)
MicData = np.roll(MicData,-1*max_index)

#Index Peaks
peaks, _ = find_peaks(GatedData, height=2*np.median(GatedData), distance=200)



dx=343/(rate*2)
peakdistance=dx*peaks

print('The Number of target is', (len(peakdistance)))
print("The distance to targets in meters is", peakdistance)

# Create a distance vector so things are in meters /2 is to compensate
# for the return trip
t=np.arange(0,len(GatedData))*dx

# plot data
plt.figure(2)
plt.title('Sonar Return Data')
plt.xlabel('Distance in Meters')
plt.ylabel('Gain in dB')
plt.semilogy(t,GatedData)
plt.semilogy(peaks*dx,GatedData[peaks], 'x')
plt.show()
# close stream
```

In [ ]:

Generate and Write Out a Chirp Train

```
## Name: Jens Evans and Sarah Mehler
## Date: March, 17th 2020
## Create Wave File of Chirp Trains
```

```
In [ ]:
# Setup Notebook
%matplotlib inline
from matplotlib import pyplot as plt
import numpy as np
from scipy import signal
from scipy.signal import chirp
```

```
In [ ]:
# Setup Parameters
DEBUG = False
#___|Center Freq = fc |_______________________________|Center Freq = fc
|_________________
#   [<-------- Pulse Repetition Interval = PRI   ---->]
#   [<--Pulse Width-->]
amplitude = np.iinfo(np.int16).max
A = amplitude

# This is the setup for the chirp. It will do a chirp from fmin to fmax for
however long PW is. If fmax is
# too high then you might have to reduce your sampling rate. Number of
periods is how many periods there
# would be for fmax so you need a lot for your fmin to have good resolution.

#Pick a center frequency
center_freq = 5e3

# Set the sweep to be 1/4 as wide as fc according to random research paper on
sonar.
# http://www.htisonar.com/publications/FMSlideChirp.pdf
fmin = center_freq - center_freq/8
fmax = center_freq + center_freq/8

# fc is now set to fmax so the old logic on how to set sampling for a single
tone
# sinusoid will be applied to our highest frequency
fc = fmax # 4kHz kilohertz


# What resolution do you want in meters
res_meters = 1
num_periods = 1
```

```
Tc= res_meters/343
PW = Tc*num_periods

# Whats your desired Unambiguos Range in meters?
UR=40 # meters
PRI=round((2*UR+343*PW)/343,3)


print('='*20)
print("Time and Frequency")
print('PRI:{} s, PW {}s, Fc {} Hz, Tc {} s'.format(PRI, PW, fc, Tc))
print('\n')

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

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))
```

In [ ]:

```
# Making the data pretty
import pandas as pd
data1 = [[PRI, PW, fc, Tc],[PRI, PW*1e3, fc/1e3, Tc*1e6]]
pd.DataFrame(data1[1:], columns=["PRI (s)", "PW(ms)", "fc (kHz)", "Tc
($\mu$s)"])
```

In [ ]:

```
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)"])
```

Making the unit waveform

In [ ]:

```
# samples per period
#samps_per_period = 10
FS_MAX = 100000 # maximum sample rate for audio transmitter

# create the time step for sampling
fmax = fc
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')
```

In [ ]:

```
# 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
```

In [ ]:

```
print(PW)
```

In [ ]:

```
## 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


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))
```

In [ ]:

```
## 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))
```

In [ ]:

```
# Create the sine wave
amplitude = np.iinfo(np.int16).max
A =amplitude
print(fc)
#data = amplitude * np.sin(2. * np.pi * freq * t)
t = np.arange(0,PW,dt)
signal = A*chirp(t_vector,fmin,PW,fmax,'linear',phi=-90)
plt.figure()
plt.plot(signal[0:len(t)])

plt.figure()
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(fc/1e3))
plt.xlabel('time (s)')
plt.ylabel('Amplitude')
signal[:20]
```

In [ ]:

```
# now combine everything
signal = signal * mask
plt.plot(t_vector,signal)
plt.title('Pulse Train Unit')
plt.xlabel('time (s)')
plt.ylabel('Amplitude')
```

In [ ]:

```
# 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)
```

In [ ]:

```
plt.plot(signal[:72000])
```

In [ ]:

```

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])
'''
```

```
In [ ]:
```