Final Project: Radar & Sonar Processing

Written By: Richard Romano and Juan Rivera-Mena

Date of Report submission: 03/17/2020

ECE 435/535 – Radar & Sonar Processing
Department of Electrical and Computer Engineering
Portland State University

Table of Contents

A. Abstract

This project was to build a 4 KHz simple radar system using a Matrix Audio Microphone, a Raspberry Pi, and Python code. The project implements fundamental radar theories such as Cross Correlation using an FM Chirp, Gating and Windowing returns, Noise and Thresholding, Target detection, and Range determination. The radar was successfully tested on targets from 0 - 5m with a range resolution of approximately a half of a meter.

B. Introduction & Background

Overview

For this project we were tasked to build a simple radar system using an audio source as a transmitter and a purchased microphone setup called “Matrix Voice.” A Raspberry PI with a python script was used for control and processing. The overall minimum goal was to detect targets a few meters away from our setup and process these returns to determine target range.

Background

The Radar class used the Chain-Home System design to teach us the fundamentals of radar and our project provides a practical example that uses the techniques and theories that made the Chain-Home System possible.

The Chain-Home System was developed during World War II by English intelligence as a method of detecting incoming bombers. This method of detection allowed them more time to prepare a counter attack and thus saved many lives. The theory of Radar and Sonar were developed at the MIT radiation laboratory during this time for much the same purpose.

Hypothesis

Using a 4.0 KHz FM-Chirp audio source, a microphone can be used to measure target returns as a basic Radar system. Using basic DSP theory these returns can be processed and used to detect targets and determine their range from the system.

C. Experiment Setup

Hardware

The hardware is based on the Matrix Voice Kit.

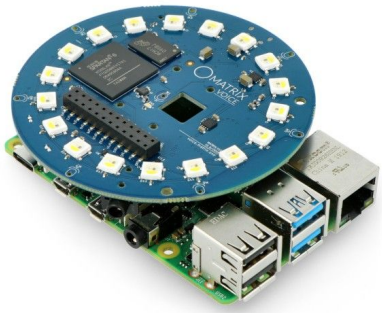


Figure 1 - Matrix Voice

1. Matrix Voice Kit
    a. Matrix Voice (ESP32)
    b. Raspberry Pi 3B+
    c. MicroSD Card
    d. 5V/2.5A USB Power Supply
2. Audio Source for a Transmitter (We used our cellphone)
3. Target (Preferably metal with at least a 1m cross-section)
4. Internet Connection (For updating/adding Python Libraries)

Software

The Radar processing was completely done in Python. Below is what was used for this experiment.

1. Anaconda - Python Setup and Library Manager
2. Jupyter Notebooks - A lightweight browser based modular Python IDE
3. Python Libraries:
    a. Pyaudio
    b. Wave
    c. Sys
    d. Matplotlib
    e. Numpy
    f. Time
    g. Pylab
    h. Scipy.signal
    i. pandas
4. Raspbian Stretch Desktop
    a. Matrix Core installed
    b. Libraries above installed

D. Experimental Data & Results

Indoor Test with 2m & 3m Target

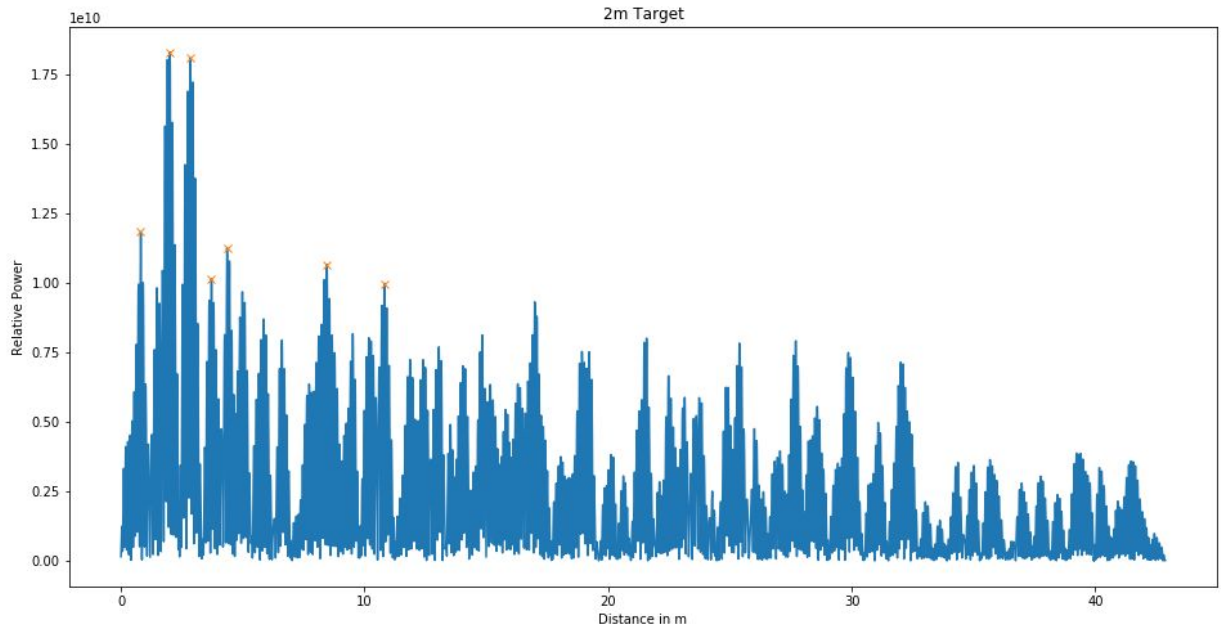


Figure 2 - 2m Distance Plot

	Distance	Return Strength
0	0.807472	1.18169e+10
1	2.0099	1.82807e+10
2	2.84371	1.80694e+10
3	3.71262	1.01325e+10
4	4.35333	1.12515e+10
5	8.45212	1.06477e+10
6	10.8131	9.96702e+09

```
Total Targets: 7
```

Figure 3 - Pandas Target List Output

Most of the testing was done in small offices/classrooms/etc. This definitely made it more challenging in the beginning as it was difficult to separate returns and figure out what we were actually looking at. Above is a typical return inside a relatively empty 4 meter x 4 meter office. There is a significant amount of multipathing and reflections.

Indoor Test with 3m & 4m Targets

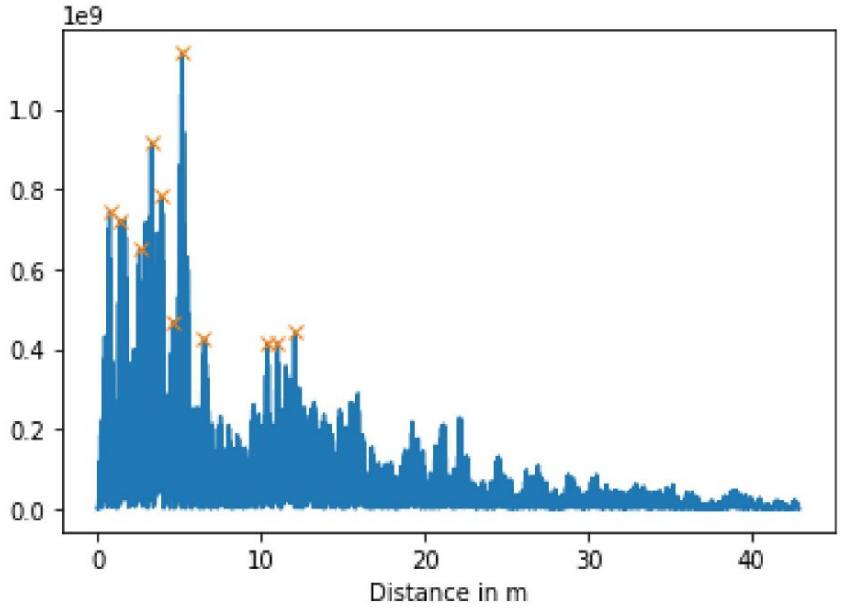


Figure 4 - 3m Distance Plot

This test was done prior to cleaning up the code and plots so it's not as nice looking but it's our best plot in terms of returns as it was done in a large hallway at PSU. We also had several other teams testing their setup as well so it was useful to compare and contrast our data.

The largest peak was at 4.61m which is approximately where our largest metal target was at. Before that the 2nd largest return is a chair that we placed at roughly 3m away. It showed up as a 3.32m return. With the FM chirp and correlation working and a narrow pulse width we are getting roughly a range resolution of half a meter.

E. Conclusion

This project was significantly more challenging than expected. I think the two problems we ran into the most were issues with the Raspberry PI and not doing a good enough job at the project planning and psuedo-code stage. Overall I'd say our code was kind of all over the place as we remembered missing parts and struggled to implement others. We did end up getting mostly everything working in the end and it was definitely exciting to actually see targets at their correct ranges.

F. Python Code Functional Description

Radar Parameters

The first block is the basic Radar Parameters setup. This was pulled from the provided Radar sim and is updated based on what transmitted signal we would like to test with.

In [2]:
```
fc = 4e3            # Carrier Frequency, Center Frequency
vp = 343            # Phase Velocity of sound
T  = 1/fc           # period of one Carrier Frequency
Lambda = vp/fc

# Setup Time portion
PRI = 0.25         # Pulse Repetition Interval (seconds)
PRF = 1/PRI         # Pulses per second (hertz)

#Num cycles per pulse packet
k = 5              # 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))

print('='*20)
print("Time and Frequency")
print('PRI:{} s, PW {}s, Fc {} Hz, Tc {} s'.format(PRI, PW, fc, T))
print('\n')

# Spatial information
R_unamb = (PRI*vp)/2 # Unambiguous Range
PRI_x = PRI * vp
PW_x = PW * vp
kc = fc/vp

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, Lambda))
```

```
====================
Time and Frequency
PRI:0.25 s, PW 0.00125s, Fc 4000.0 Hz, Tc 0.00025 s


====================
Space and Time
PRI:85.75 m, PW 0.43 m, kc 11.66 cycles/meter, lambda 0.09 m
```

Audio Receiving

The second block is our main audio recording block. A useful feature we added was a trigger to start recording. Typical radars have the transmitter send trigger signals to the receiver; since we used a cell phone for our transmitter we needed some way to tell the program when to start recording. The trigger sits in a constant loop waiting for the threshold to be met which will only happen during a loud noise, in our case the transmitted signal.

Most of our design is based on the RECORD_SECONDS variable. This determines our number of observations and our gating/windowing parameters.

In [3]:
```
import pyaudio
import wave
import sys
import matplotlib.pyplot as plt
import numpy as np
from time import sleep
import pylab

# recording configs
CHUNK = 2048
FORMAT = pyaudio.paInt16
CHANNELS = 2
RATE = 96000
RECORD_SECONDS = 4
NFRAMES = int((RATE / CHUNK) * RECORD_SECONDS)
NSAMPS = NFRAMES * CHUNK
print(NSAMPS)
WAVE_OUTPUT_FILENAME = "output.wav"

# create & configure microphone
mic = pyaudio.PyAudio()
stream = mic.open(format=FORMAT,
                  channels=CHANNELS,
                  rate=RATE,
                  input=True,
                  frames_per_buffer=CHUNK)

# read & store microphone data per frame read
frames = []
trigger = 0
sleep(0.01)
print("* Awaiting Trigger *")
while trigger < 20000:
    dataTrig = stream.read(CHUNK)
    decoded = np.frombuffer(dataTrig,dtype=np.int16);
    trigger = np.amax(decoded)
    #display(trigger)
    sleep(0.001)

print("* recording *")

for i in range(0, int(RATE / CHUNK * RECORD_SECONDS)):
    data = stream.read(CHUNK)
    numpydata = np.frombuffer(data,dtype=np.int16)
    frames.append(data)

print("* done recording *")

# kill the mic and recording
stream.stop_stream()
stream.close()
mic.terminate()
```

```
382976
* Awaiting Trigger *
* recording *
* done recording *
```

Audio Import & Filtering

Although the data was technically there from pyaudio it gets stored in wav format which isn't really useful. To give us a useful numpy array we actually output the file as a .wav and bring it back in using scipy. Not the most efficient method but it was quick to implement.

We tested this code with all 8 channels recording but for ease of debugging we often just used the first channel. In our code the channels would be stored interweaved and just need to broke out using:

```
channelX = audData[:X] # Channel # = X
```

An important feature that we added way too late into our design was filtering out the noise. For this we used a bandpass filter function from 2500 to 5500 Hertz. This dramatically reduced our noise and improved target detection.

In [4]:
```
from scipy.signal import butter, lfilter
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
```

In [5]:
```
#combine & store all microphone data to output.wav file
outputFile = wave.open(WAVE_OUTPUT_FILENAME, 'wb')
outputFile.setnchannels(CHANNELS)
outputFile.setsampwidth(mic.get_sample_size(FORMAT))
outputFile.setframerate(RATE)
outputFile.writeframes(b''.join(frames))
outputFile.close()

# Plot
time = np.arange(0, float(numpydata.shape[0]), 1) / RATE
#plt.plot(time, numpydata)

# SciPy Import
import scipy.io.wavfile
rate,audData=scipy.io.wavfile.read('output.wav')

#wav bit type the amount of information recorded in each bit often 8, 16 or 32 bit
audData.dtype

print("Data Shape:")
print(np.shape(audData))

#wav length
len_PRI = audData.shape[0] / rate

#wav number of channels mono/stereo
audData.shape[1]

#if stereo grab both channels
channel1=audData[:,0] #left
# channel2=audData[:,1] #right

# Quick bandpass filter to clean up noise.
channel1 = butter_bandpass_filter(channel1, 2.5e3, 5.5e3, fs=96000, order=1)
```

```
96000
[[ 602  626]
 [ 965  976]
 [1242 1247]
 ...
 [ 499  343]
 [ 849  537]
 [1192  718]]
Data Shape:
(382976, 2)
```

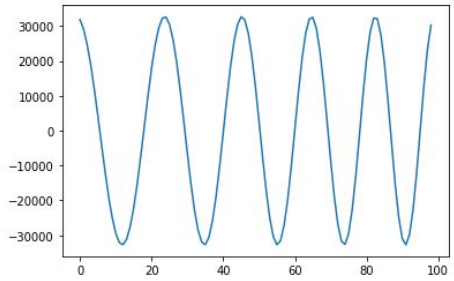
FM Chirp Kernel Import

Next was to bring in our Cross Correlation Kernel. For this I used our waveform generator file to create a second kernel file that only had the unit signal in it. We then used the trim zeros function to remove everything but our FM chirp.

In [6]:
```
# Cross Correlation Kernel
# Plot
time = np.arange(0, float(numpydata.shape[0]), 1) / RATE
#plt.plot(time, numpydata)

# SciPy Import
kernelRate,kernelAudData=scipy.io.wavfile.read('Kernel_PulseTrain_chirp_4000_025_005.wav')

#the sample rate is the number of bits of infomration recorded per second
print(kernelRate)
print(kernelAudData)

#wav bit type the amount of information recorded in each bit often 8, 16 or 32 bit
kernelAudData.dtype

print("Data Shape:")
print(np.shape(kernelAudData))

#wav length
kernel_len_PRI = kernelAudData.shape[0] / rate

#Final kernel
kernel = np.trim_zeros(kernelAudData,'fb')
plt.plot(kernel)
```

```
80000
[    0.         31855.62760586 29148.73274129 ...     0.
     0.             0.        ]
Data Shape:
(20000,)
```

Out[6]: [<matplotlib.lines.Line2D at 0x26f3db5af08>]



Sample Padding & Gating

Almost every group will probably mention that they ran into all sorts of plotting issues. This was fixed by padding out our array.

We also do our reshaping and summing at this point.

In [7]:
```
# Padding
print('Channel 1 Length:'), print (len(channel1))
print('NSAMPS Length:'), print (NSAMPS)

while np.mod(NSAMPS,RECORD_SECONDS) != 0:
    NSAMPS += 1
    print(NSAMPS)

if len(channel1) > NSAMPS:
    mod_channel1 = channel1[NSAMPS - len(channel1)]
    print("* Shortened *")
    print('NEW Channel 1 Length:'), print (len(mod_channel1))
if len(channel1) < NSAMPS:
    mod_channel1 = np.pad(channel1, (0,NSAMPS-len(channel1)), 'median')
    print("* Padded *")
    print('NEW Channel 1 Length:'), print (len(mod_channel1))
if len(channel1) == NSAMPS:
    mod_channel1 = channel1


# N Observations Effect
# provide n observation effects
channel1_env = mod_channel1.reshape(RECORD_SECONDS, int(NSAMPS/RECORD_SECONDS))
# add them all together
n_obs_channel1_env = channel1_env.sum(axis=0)

print('Summed Length:'),print(len(n_obs_channel1_env))

# Turn off Shaping
#n_obs_channel1_env = channel1
```

```
Channel 1 Length:
382976
NSAMPS Length:
382976
Summed Length:
95744
```

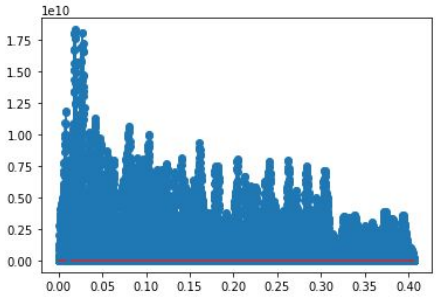
Cross Correlation

Now that we have a proper array of one PRI we can finally do some cross correlation with our FM chirp kernel. For this we used scipy's correlate function. We also window the returns to only look at the first 0.40 seconds as we are working in smaller rooms with low output power.

In [9]:
```
# Cross Correlation:
from scipy.signal import correlate

PRI_index = int(len(n_obs_channel1_env)/CHANNELS)
zoom = 10
PRI_index_zoom = int(PRI_index/zoom)

xcorr_array = np.absolute(correlate(n_obs_channel1_env[0:PRI_index_zoom],kernel,'full'))
xcorr_Len = len(xcorr_array)
print('Original array length: {}'.format(len(n_obs_channel1_env[0:PRI_index_zoom])))
print('New array length: {}'.format(xcorr_Len))

time3 = np.arange(0, xcorr_Len, 1) / rate*RECORD_SECONDS*CHANNELS
time4 = np.arange(0, len(kernel), 1) / rate*RECORD_SECONDS*CHANNELS

plt.stem(time3,xcorr_array,use_line_collection=True)
plt.stem(time3,n_obs_channel1_env[0:xcorr_Len],use_line_collection=True)
plt.stem(time4,-1*kernel,use_line_collection=True)
```

```
Original array length: 4787
New array length: 4885
```

Out[9]:
```
<StemContainer object of 3 artists>
```



Target Detection & Ranging

We have finally reached the point where we can detect targets. For most of the term we were using a multiple of the median as our threshold and find peaks with distance turned on to locate targets. Although custom thresholding was built for the radarsim, it hasn't yet been implemented into our actual radar build.

This gives us our list of targets with distance and return strength.

In [47]:
```
# Target Detection
# Find Peaks still used as threshold function was built later

from scipy.signal import find_peaks
R_unamb_zoom = R_unamb
scalar = np.median(np.absolute(xcorr_array[0:int(len(xcorr_array)/10)]))*2.2
peaks, _ = find_peaks(xcorr_array, height=scalar, distance=60)

# make the distance vector
nsamps = int(len(xcorr_array))
x = np.linspace(0,R_unamb_zoom, nsamps)
dx = R_unamb_zoom/nsamps

# Pandas Table
import pandas as pd
# intialise data of lists.
data = {'Distance':peaks*dx,
        'Return Strength':xcorr_array[peaks]}

# Create DataFrame
df = pd.DataFrame(data)
df = df.style.set_properties(**{
    'font-size': '10pt'})

# Print the output.
display(df)
print('Total Targets: {}' .format(len(peaks)))
```

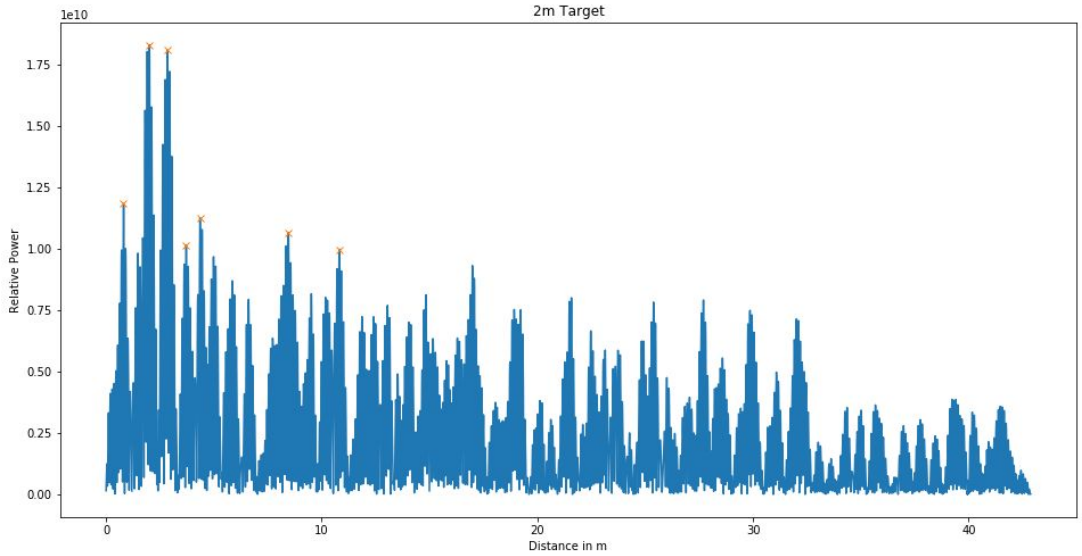
	Distance	Return Strength
0	0.807472	1.18169e+10
1	2.0099	1.82807e+10
2	2.84371	1.80694e+10
3	3.71262	1.01325e+10
4	4.35333	1.12515e+10
5	8.45212	1.06477e+10
6	10.8131	9.96702e+09

```
Total Targets: 7
```

Radar A-Scope

Finally we have our last plot with our returns and peaks determined. This plot is known as a Radar A-Scope. Our next, but sadly unfinished goal was to make the Radar B-Scope for azimuth (angle.) This would involve running the entire program against all 8 channels; before that we would prefer to have further developed everything as functions as clean things up further.

```
In [48]: # Distance Plot

         plt.figure(figsize=(16,8))
         plt.plot(x,xcorr_array[0:nsamps])
         plt.plot(peaks*dx,xcorr_array[peaks], 'x')
         plt.title('2m Target')
         plt.xlabel('Distance in m')
         plt.ylabel('Relative Power')

Out[48]: Text(0, 0.5, 'Relative Power')
```



This is the end of our program in its current state. It works on both our laptops and on the Raspberry Pi. On the Pi it generally just needs some light modification for plot and data display. It was interesting learning to use Python, and although Jupyter Notebooks worked out I think using Spyder would have made exchanging code between Windows and Rpi a bit smoother.

G. References

[1] Louis N. Ridenour. *Radar System Engineering*, volume 1 of *MIT Radiation Laboratory Series*. McGraw-Hill, New York, 1947.

[2] Wolff, Christian. "Radar Basics," Radartutorial.eu, 2020, https://www.radartutorial.eu/.

[3] Skolnik, Merrill I. *Radar Handbook,* 3rd ed., Mcgraw-Hill, 2008.

H. Appendix

Full Python Code

See Github Jupyter Notebook Current_Build_v02

Full RadarSim Code

See Github Jupyter Notebook RadarSimJupyter_team1

RadarSim Questions & Answers

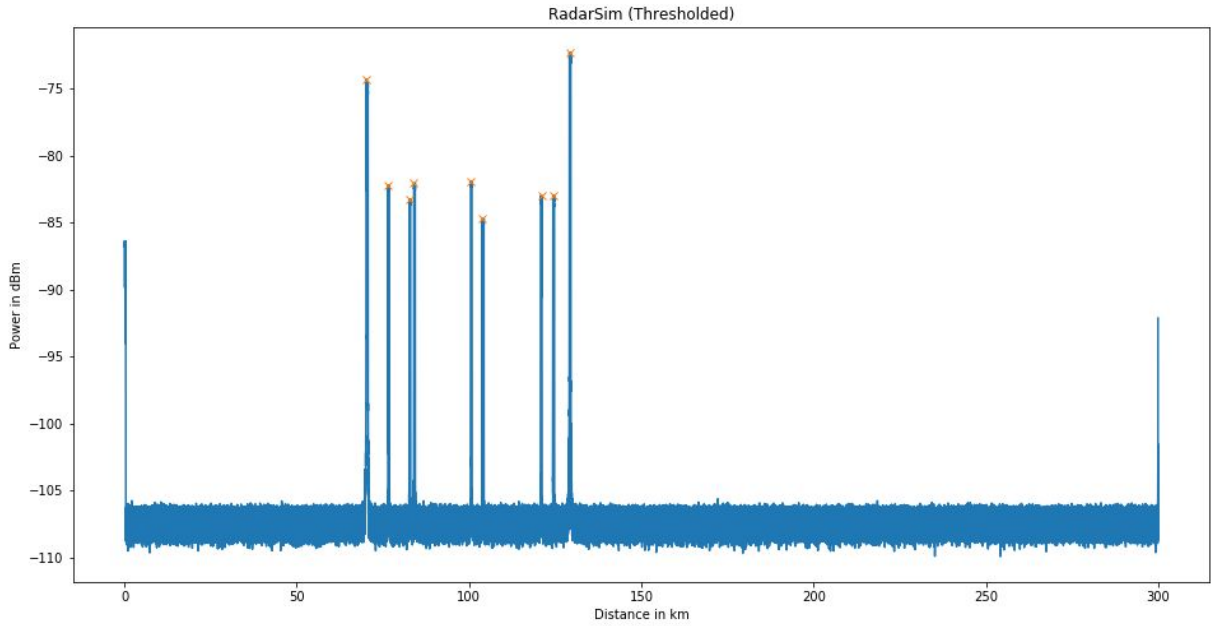


Figure 5 - RadarSim Final Plot

1. **Add probability of detection function**

```
def prob_detection(var1, var2):

    SNR = var1 #SNR obtained from the difference between noise floor power and peak power
    num_observations = var2

    false_detection = np.exp(-(SNR)*np.sqrt(num_observations))

    prop_detection = 1 - false_detection
    return prob_detection
```

2. **Add function to detect/generate new noise floor after gating and summing**

```
def new_noise_floor(var1):

    sum_and_gated = var1

    n_l = np.median(sum_and_gated)

    return n_l
```

3. **Use probability of detection function to determine threshold**

```
#SNR calculation and probability of false detect

noise_level = new_noise_floor(n_obs_main_trace_env[20000:30000])
print(noise_level)

noise_level = noise_level*330 #Noise level multiplier
print(noise_level)

snr_level = 10 * np.log10(n_obs_main_trace_env[peaks]/noise_level)
print(snr_level)

p = prob_detection(snr_level,K_pulses)

print(len(p))

count = 0

for i in range(0,len_p):
        if p[i] < 0:
                p[i] = 0
        elif p[i] > .6:
                count = count + 1
print(p)
print(count)
```

4. **What is the last peak?**

   The last peak is a bit of aliasing of the first transmit pulse peak.

5. **Convert plot to dBm**

   Added line:

   n_obs_main_trace_env_dbm = 10*np.log10(n_obs_main_trace_env/1e-3)

6. **Replace find_peaks with a thresholding function**

   See question 3 as it's a combined function.