1 3097S Milestone 1: Sonar chirp pulse simulation and processing

1.1 Part 1

This report is split into two components. The first componment dealt with a similiar scenario to that of the actual project. The second component dealt with adapting the tasks in part one to that which was relevant the design compenent. This was done to familiarise with techniques in sonar simulation and processing.

1.1.1 Step 1

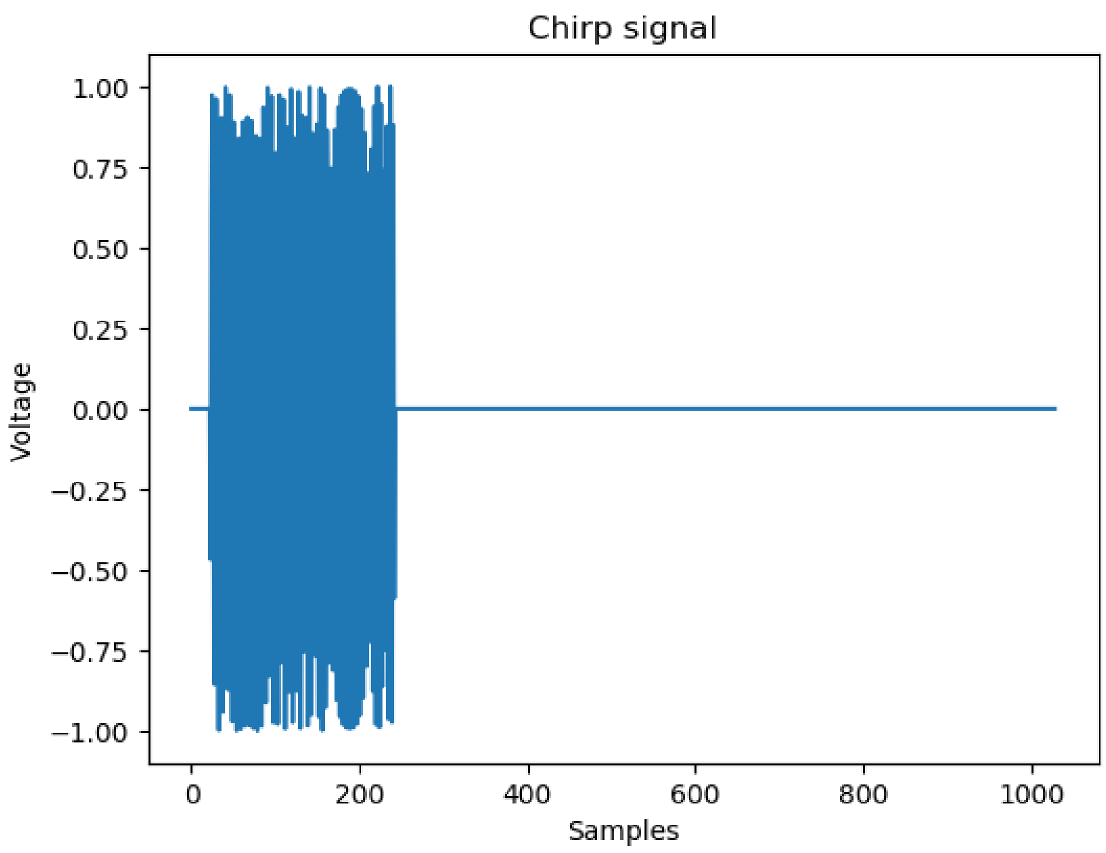
In order to simulate the sonar chirp pulse a chirp pulse was simulated using the rect function. This function had a Pulse length of 5ms, a bandwidth of 4kHz and a centre frequency of 10kHz.

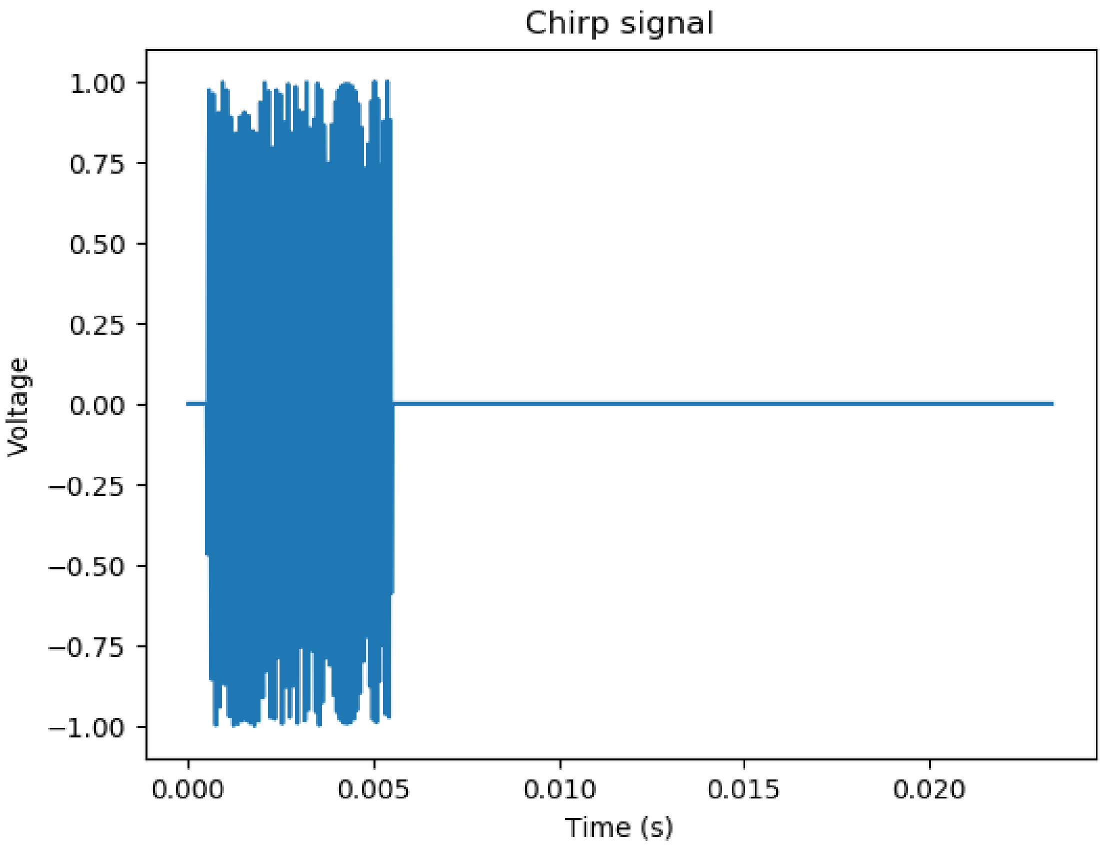
[116]:
```
#STEP 1: Chirp pulse creation

c = 343;  # Speed of sound in air in m/s
fs = 44100; # This is the sample rate of the sonar.
dt = 1/fs; # This is the sample spacing
r_max = 4; # Maximum range in metres to which to simulate.
t_max = 2*r_max/c; # Time delay to max range

# Create an array containing the time values of the samples
t = collect(0:dt:t_max); # t=0:dt:t_max defines a range.
# Create an array containing the range values of the samples
r = c*t/2;
# NOW create the chirp pulse, shifted by an amount td, to start at
# some time td-T/2>0.
f0 = 10000; # Centre frequency is 10 kHz
B = 4000; # Chirp bandwidth
T = 5E-3; # Chirp pulse length
K = B/T; # Chirp rate
# Define a simple a rect() function which returns for -0.5<=t<=0.5 or 0.
# The function will work if t is an array of values.
rect(t) = (abs.(t) .<= 0.5)*1.0
# rect(t/T) spans the interval [-T/2,T/2]
# We must therefore delay the chirp pulse so that it starts after t=0.
# Shift the chirp pulse by 0.6T units to the right, so that it starts at
```

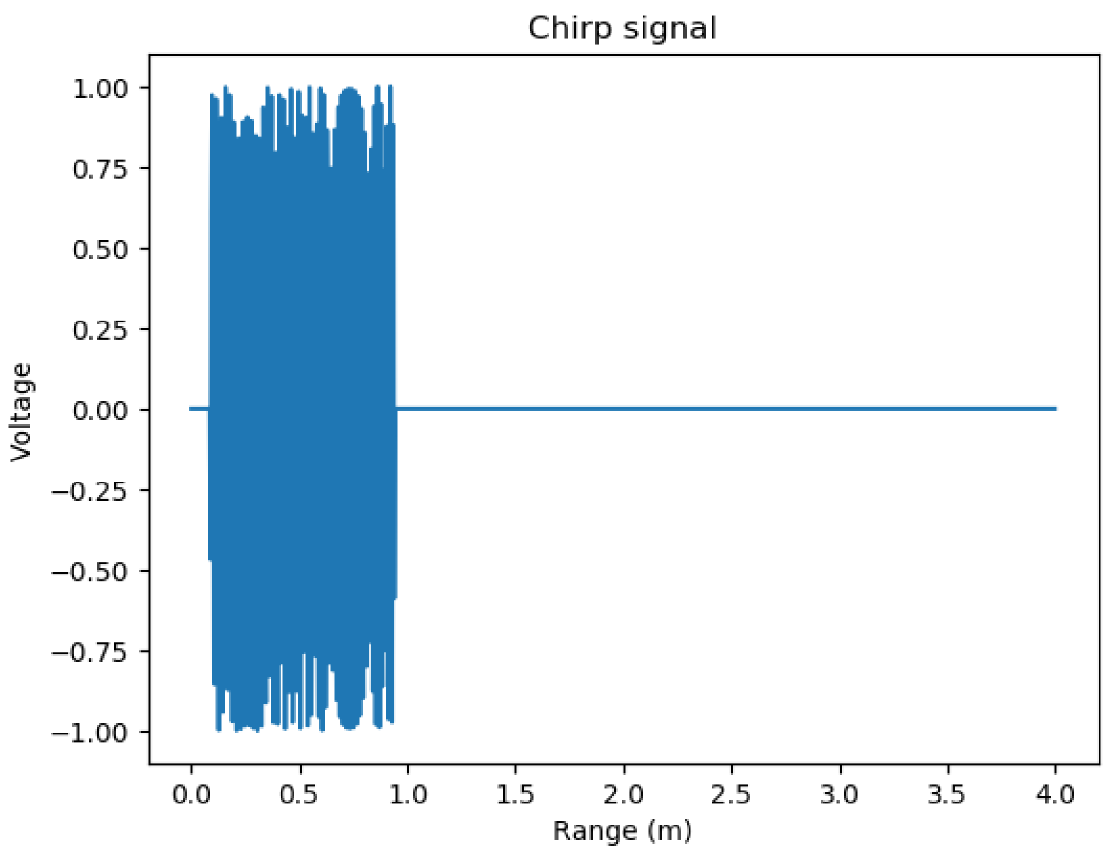
```
0.1*T
td = 0.6*T; # Chirp delay
# Note: one can use the macro @. to avoid having to put . for arrays:
# @. v_tx = cos( 2*pi*(f0*(t-td) + 0.5*K*(t-td).^2) ).*rect((t-td)/T);
v_tx = cos.( 2*pi*(f0*(t .- td) + 0.5*K*(t .- td).^2) ) .* rect.((t .-td)/T);
```

The chirp signal was represnted in both the time and frequecny domains. These plots were zoomed into and examined using the "pygui(true)" command which launched windows of the plots. The frequency plots were made using the the FFTW library within Julia.

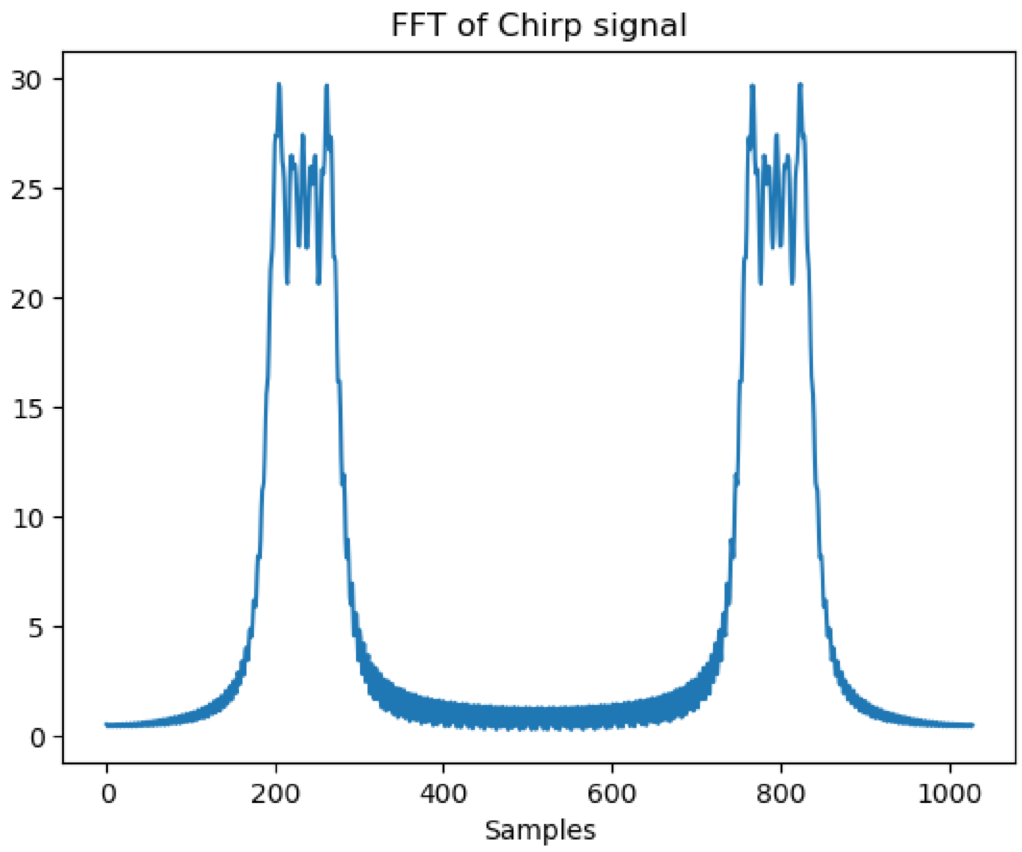
[117]:
```
using PyPlot; pygui(false) # import plot library
# If not installed, add the package via: using Pkg; Pkg.add("PyPlot");
figure() # Create a new figure
plot(v_tx) # Basic plot, axis labeled in samples
title("Chirp signal")
xlabel("Samples");
ylabel("Voltage");

figure() # Create a new figure
plot(t,v_tx) # Put time on x-axis
title("Chirp signal")
xlabel("Time (s)");
ylabel("Voltage");

figure() # Create a new figure
plot(r,v_tx) # Put range on x-axis
title("Chirp signal")
xlabel("Range (m)");
ylabel("Voltage");
```


Chirp signal
Voltage
Samples
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
200
400
600
800
1000


Chirp signal
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Voltage
0.000
0.005
0.010
0.015
0.020
Time (s)

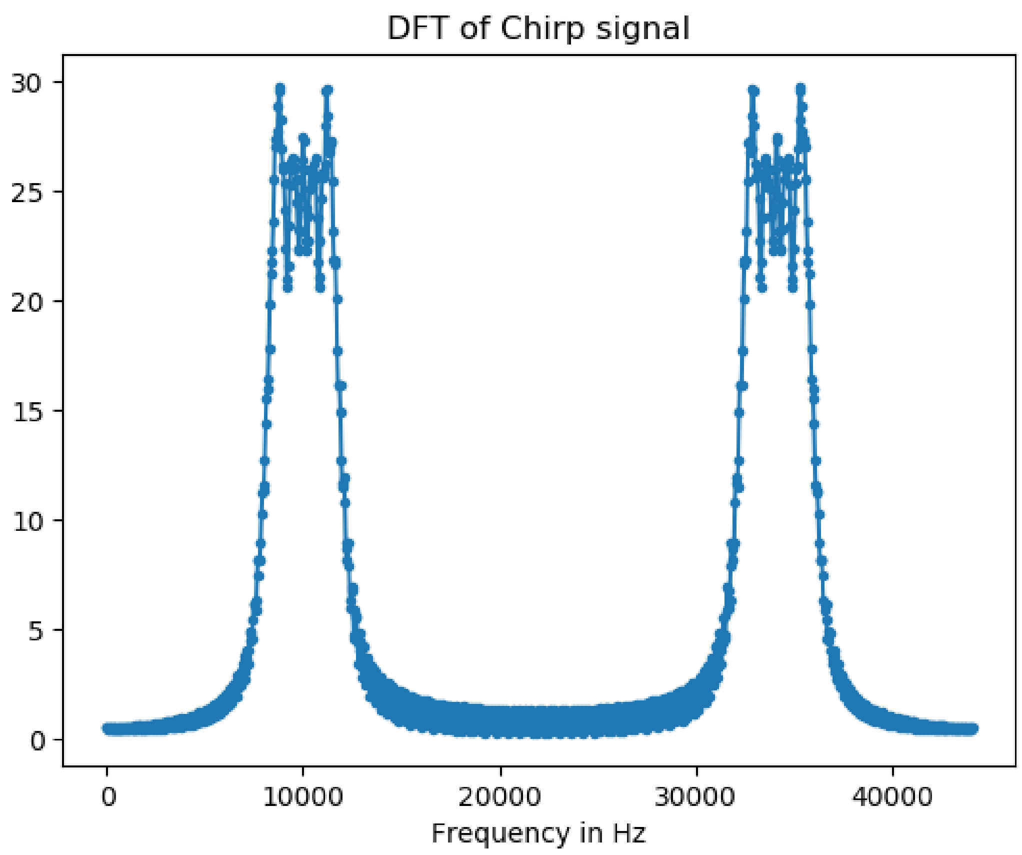


[118]:
```
using FFTW # If not installed do: using Pkg; Pkg.add("FFTW");

V_TX = fft(v_tx);
figure()
plot( abs.(V_TX) )
title("FFT of Chirp signal");
xlabel("Samples");
```



A discrete Fourier transform plot was also created. The right hand side represented the negative components.

[119]:
```
#LABEL frequency axis
N = length(t);
f = 1/(N*dt) # spacing in frequency domain
#create array of freq values stored in f_axis. First element maps to OHz
f_axis = (0:N-1)*f;
figure();
plot(f_axis, abs.(V_TX),".-");
title("DFT of Chirp signal");
xlabel("Frequency in Hz");
```

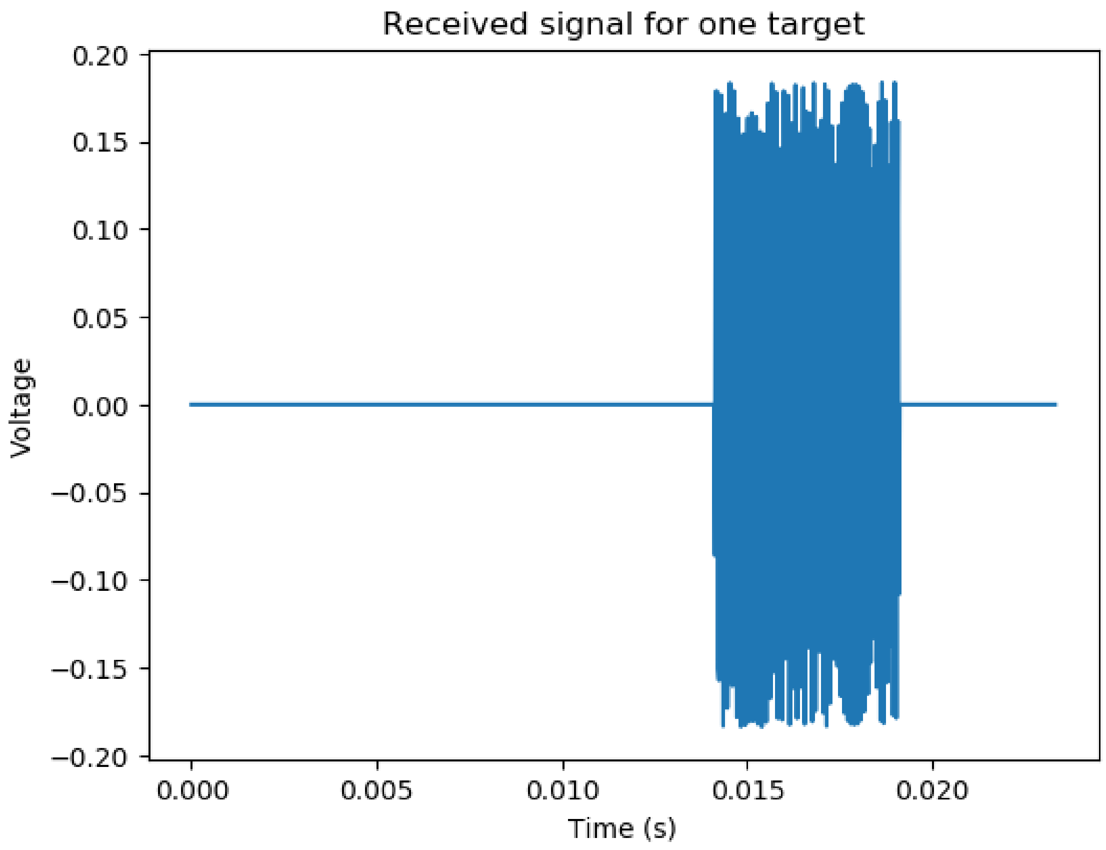


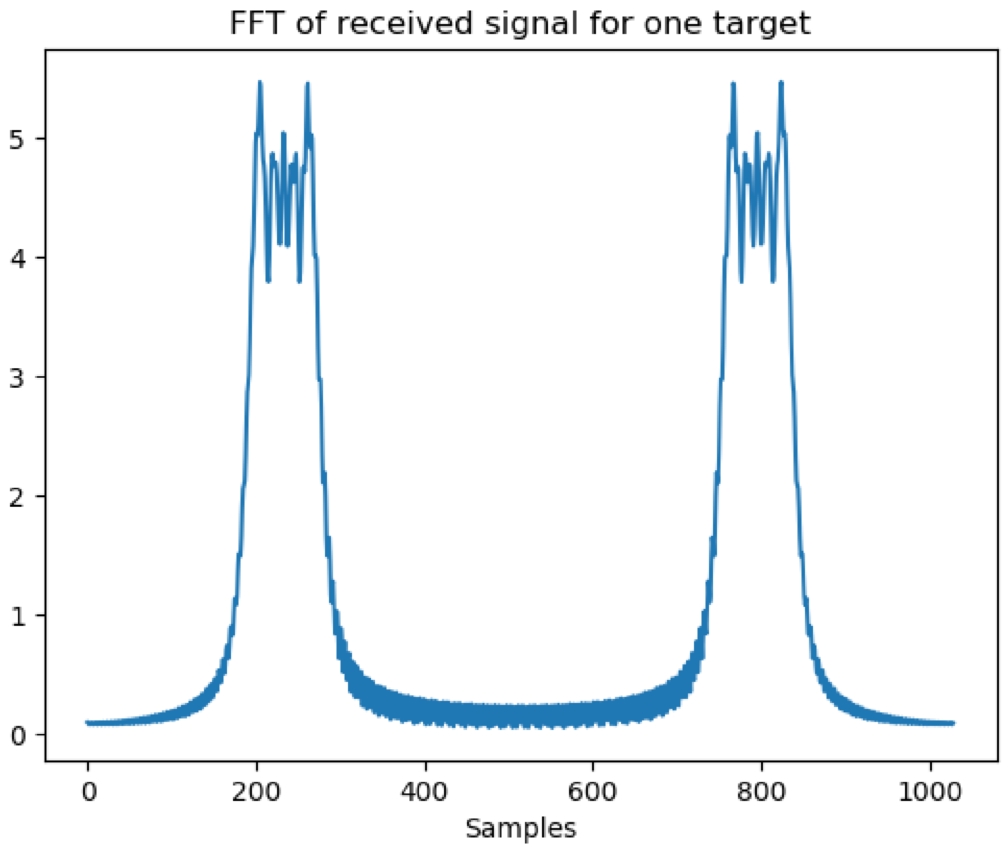
1.1.2 Step 2

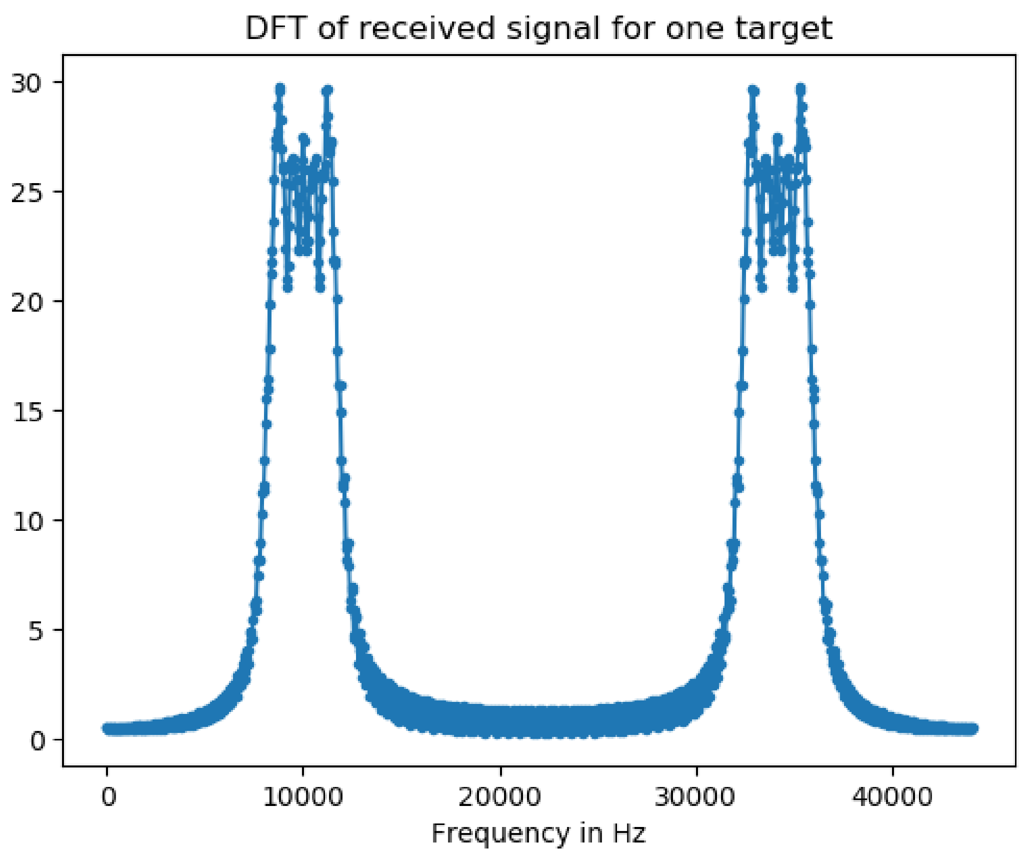
Two targets were simulated. The first was at a distance of 2.33m (birthday in February). The second was taking at an arbitrary distance using October in the R1 equation. From this time and frequency domain plots were created. First for the first target and then for both targets added superimposed.

[120]:
```
#STEP 2: Point target simulation

R1 = 1.5 + (12-2)/12 # 2.33m - range to target.
td1 = 2*R1/c; # two way delay to target.
A1 = 1/R1^2; # echo voltage signal proportional to 1/R^2
v_rx = A1*cos.( 2*pi*(f0*(t.-td.-td1) + 0.5*K*(t.-td.-td1).^2) ) .* rect((t.-
td.-td1)/T);

V_RX = fft(v_rx);

figure() # Create a new figure
plot(t,v_rx) # Put time on x-axis
title("Received signal for one target")
```

```
xlabel("Time (s)");
ylabel("Voltage");

figure() # Create a new figure
plot(abs.(V_RX) )
title("FFT of received signal for one target");
xlabel("Samples");

figure();
plot(f_axis, abs.(V_TX),".-");
title("DFT of received signal for one target");
xlabel("Frequency in Hz");
```

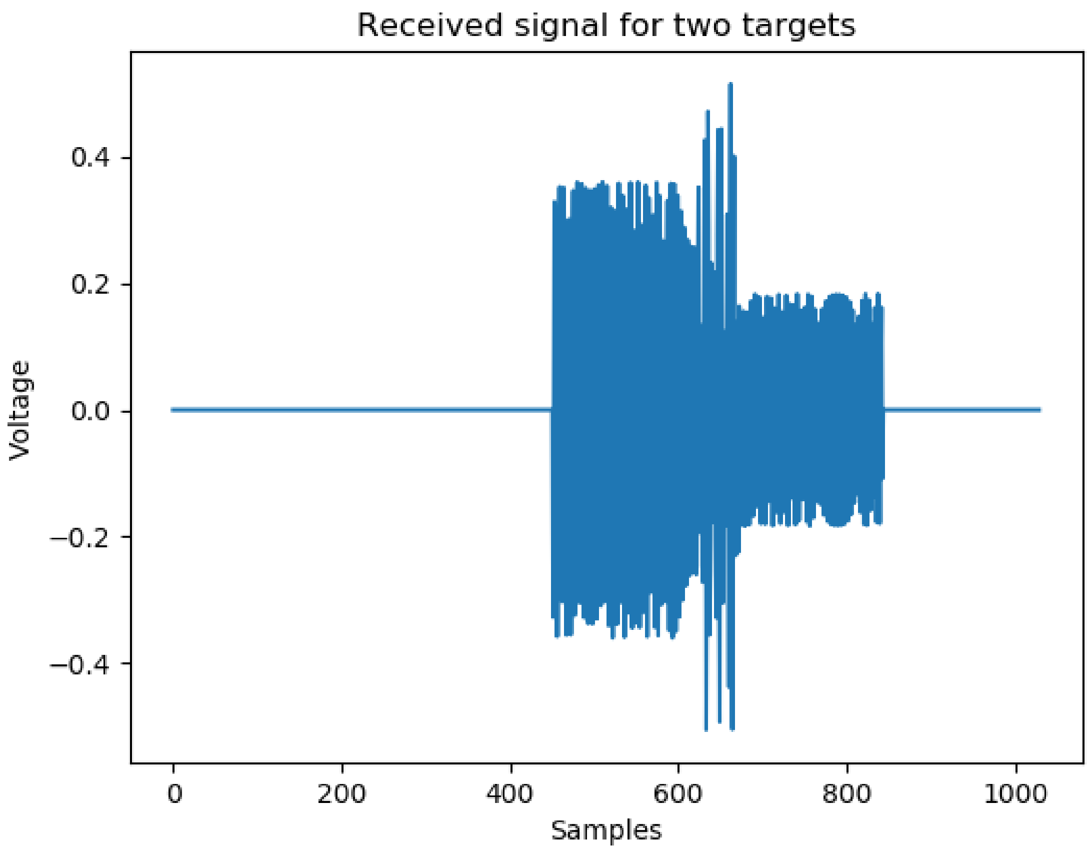


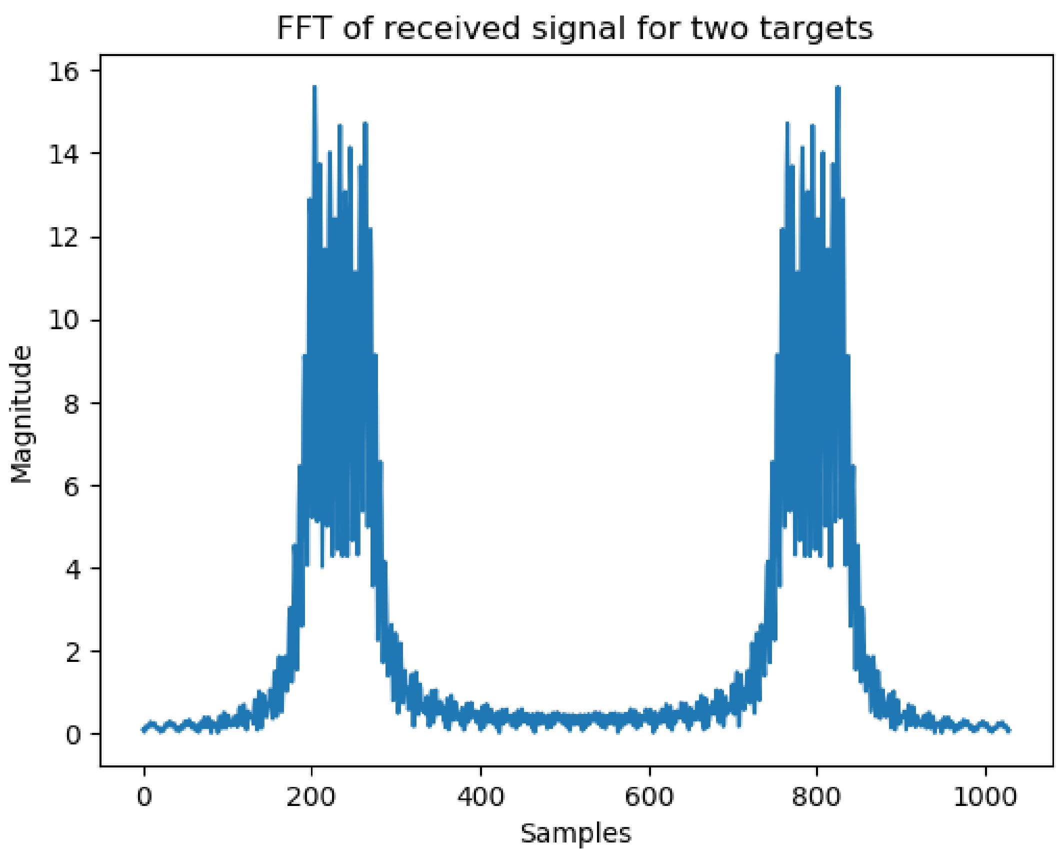

FFT of received signal for one target
0
1
2
3
4
5
0
200
400
600
800
1000
Samples



[121]:
```
#second target
R2 =  1.5 + (12-10)/12# 1.67m - range to target.
td2 = 2*R2/c; # two way delay to target.
A2 = 1/R2^2; # echo voltage signal proportional to 1/R^2
v_rx = A1*cos.(2*pi*(f0*(t.-td.-td1) + 0.5*K*(t.-td.-td1).^2) ) .* rect((t.-
td.-td1)/T) + A2*cos.(2*pi*(f0*(t.-td.-td2) + 0.5*K*(t.-td.-td2).^2) ) .*␣
 ↪rect((t.-
td.-td2)/T);
V_RX = fft(v_rx);

figure() # Create a new figure
plot(v_rx) # Put time on x-axis
title("Received signal for two targets")
xlabel("Samples");
ylabel("Voltage");

figure() # Create a new figure
plot(abs.(V_RX) )
title("FFT of received signal for two targets");
xlabel("Samples");
```

```
ylabel("Magnitude");
```



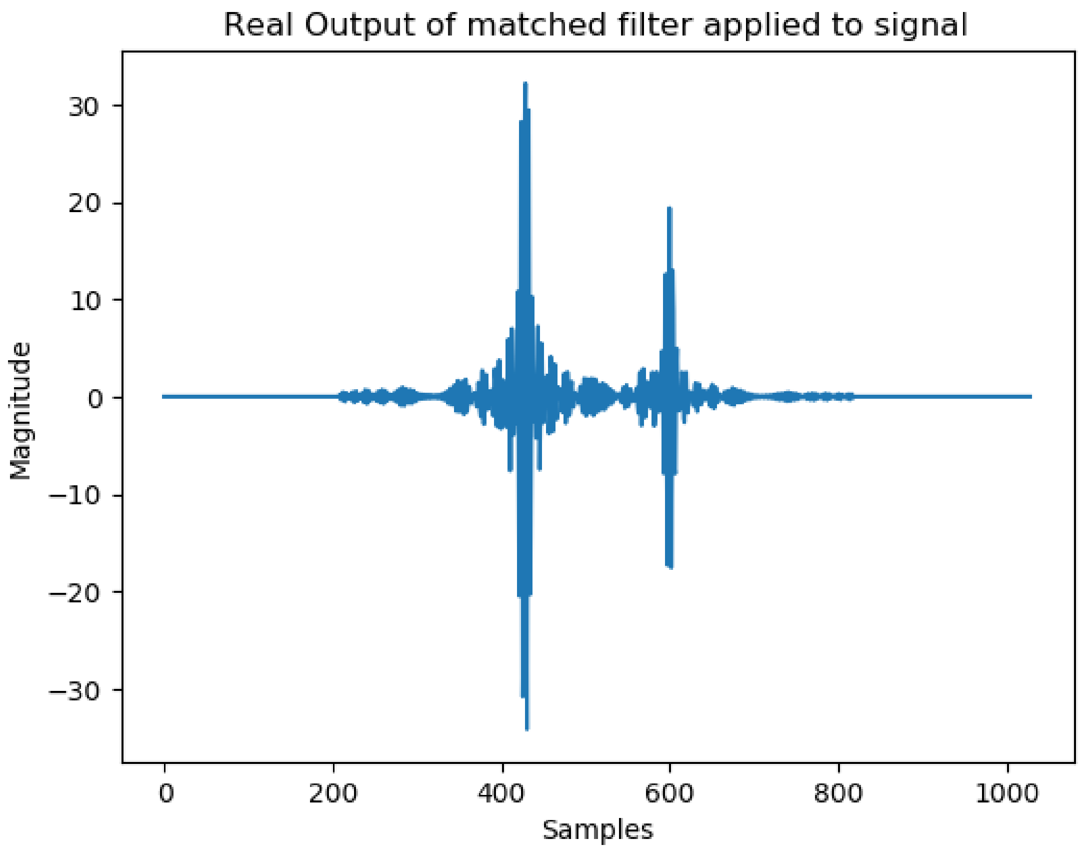


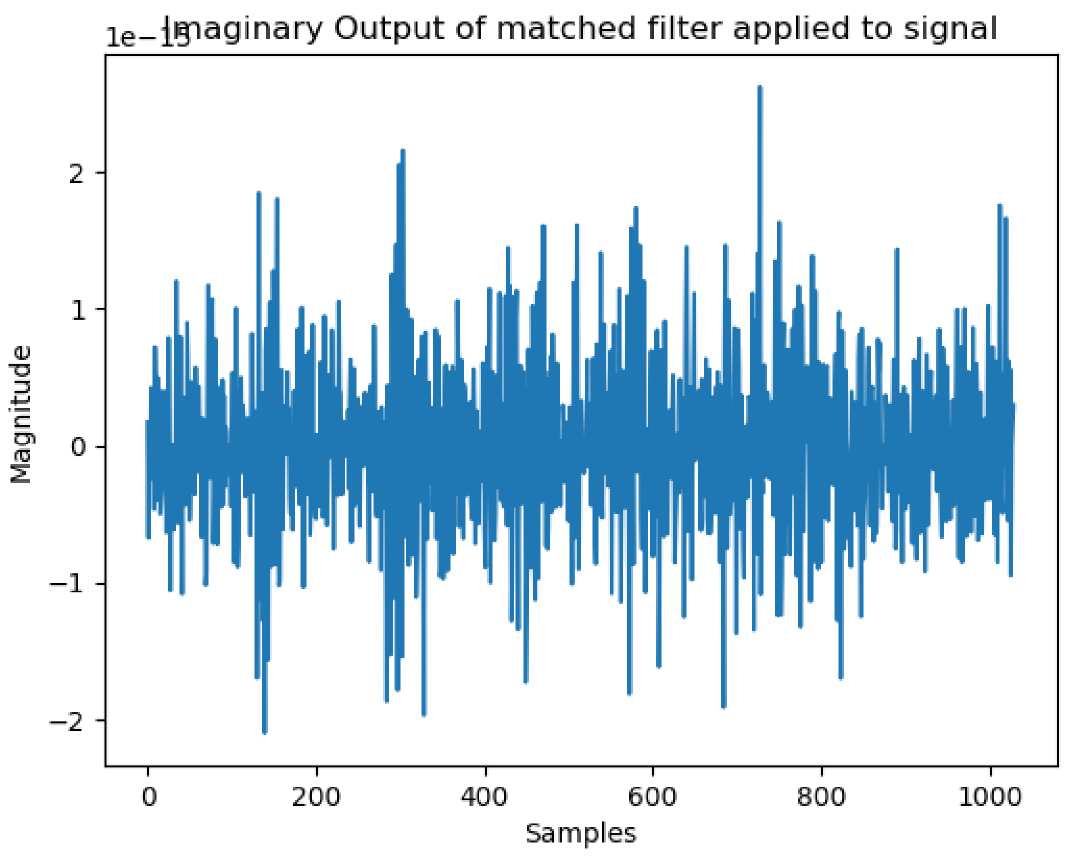
1.1.3 Step 3

In order to maximise the signal to noise ratio the match filter method was applied. This resulted in a real signal as the imaginary signal had a neglible magnitude, smaller the 1E-14. This resulted in pulse compression.

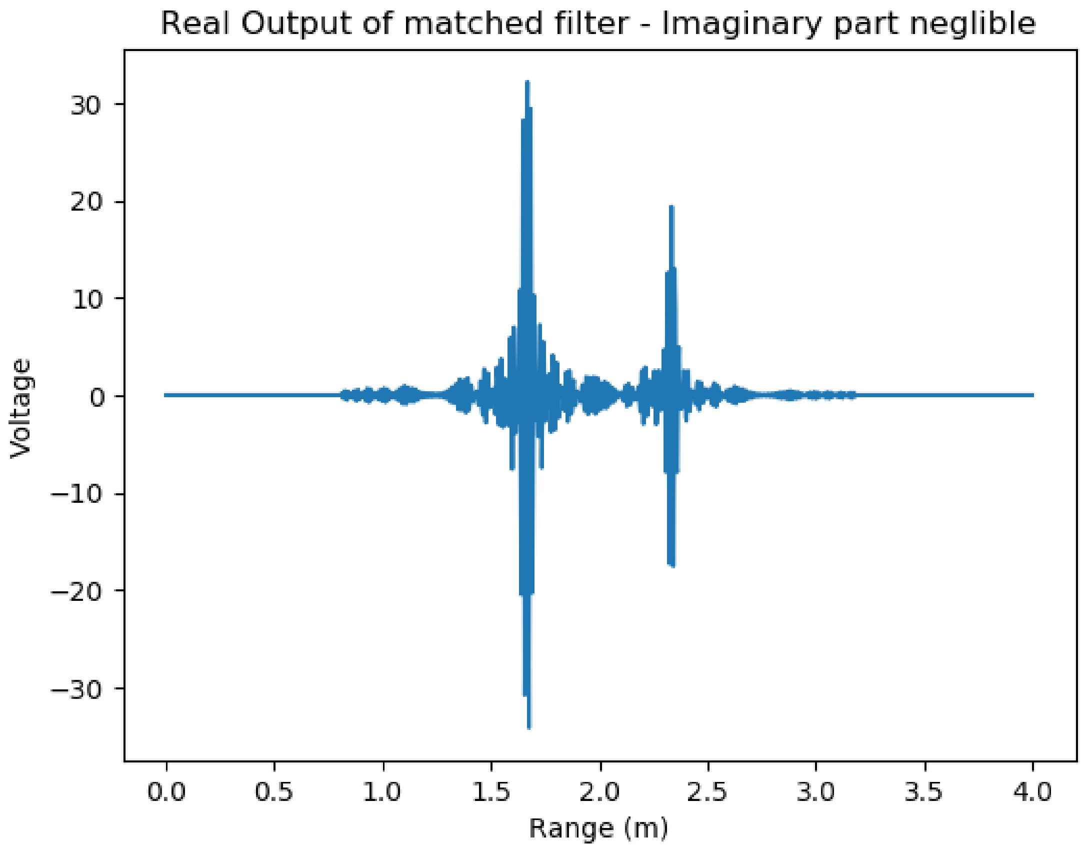
[122]:
```
#STEP 3: Matched filtering

H = conj( V_TX);
V_MF = H.*V_RX;
v_mf = ifft(V_MF);

figure() # Create a new figure
plot(real(v_mf));
title("Real Output of matched filter applied to signal")
xlabel("Samples");
ylabel("Magnitude");

figure() # Create a new figure
plot(imag(v_mf));
```

```
title("Imaginary Output of matched filter applied to signal")
xlabel("Samples");
ylabel("Magnitude");

#Negligble Imaginary part <1E-14
```





[123]:
```
v_mf = real(v_mf);
figure() # Create a new figure
plot(r,v_mf) # To see the detail  zoom in to have a good look.
title("Real Output of matched filter - Imaginary part neglible")
xlabel("Range (m)");
ylabel("Voltage");


# Take note of the shape of the envelope, as well as the internal detail.
# effect of pulse compression
```

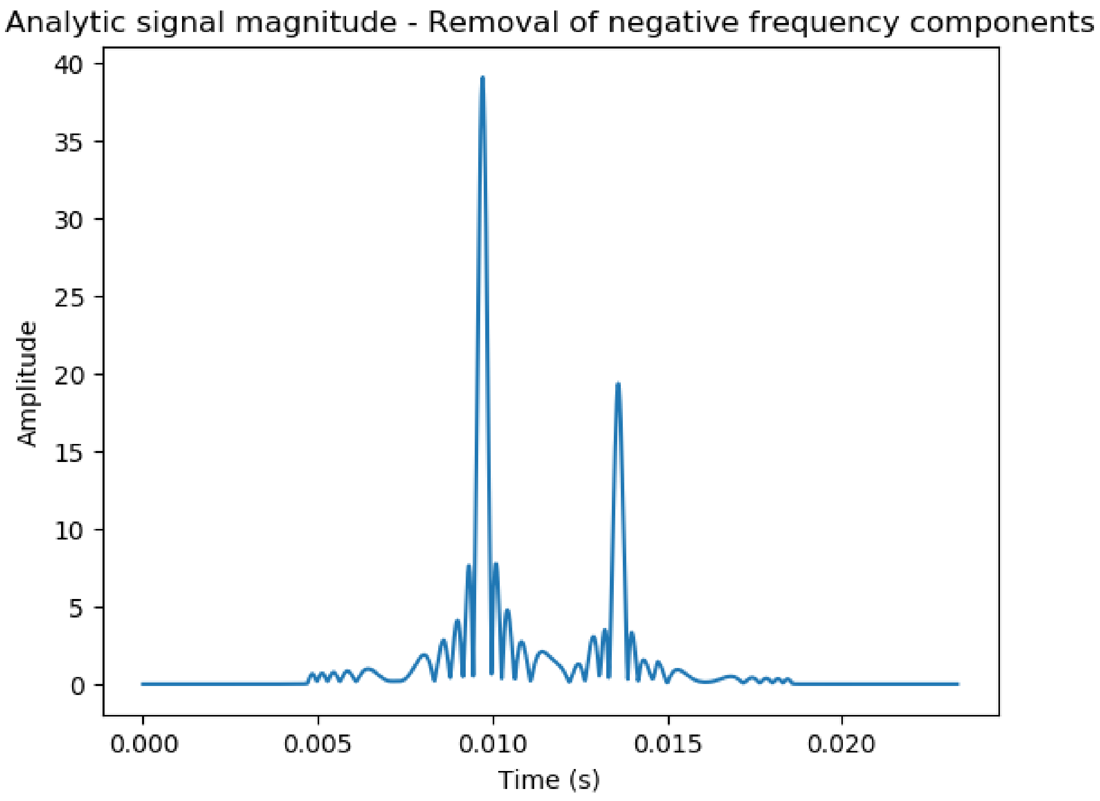


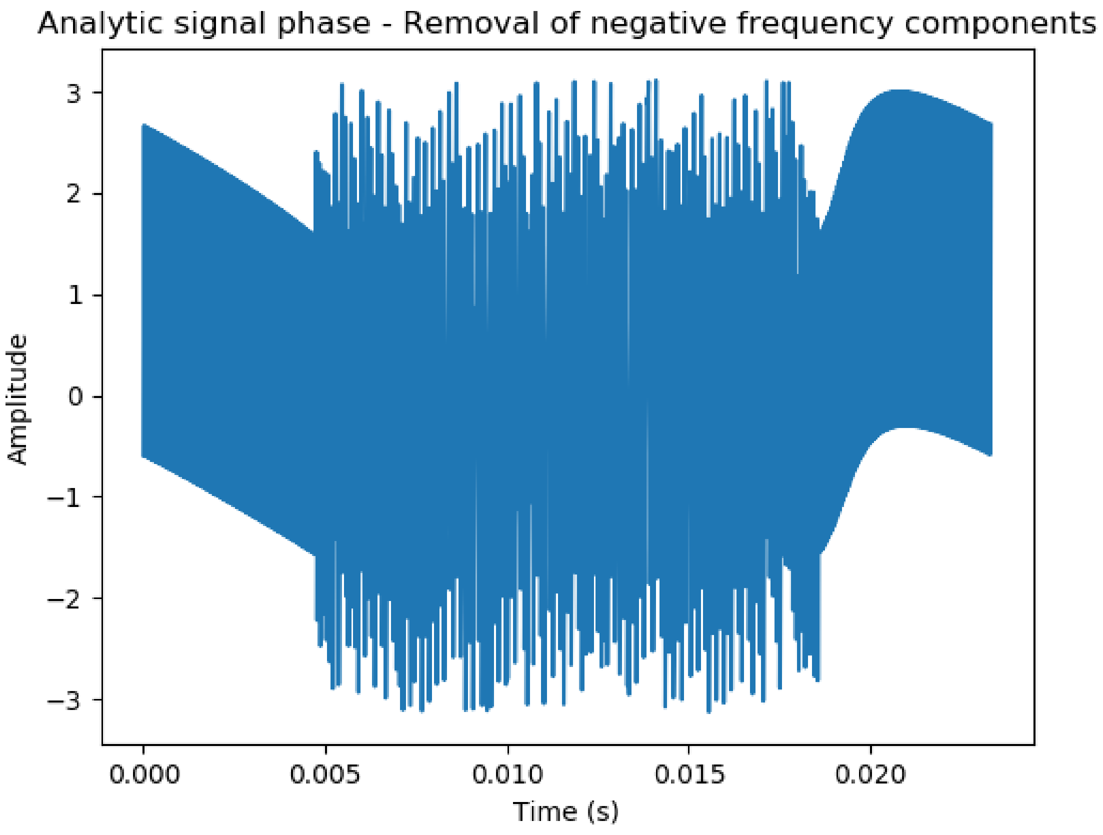
1.1.4 Step 4

In order to form the analytic signal, the negative frequency components were removed. This was done by zeroing the second half of the array. The analytic signal was obtained from the inverse fourier transform. The magnitude and phase of the analytic signal were plotted in the time domain.

[124]:
```
#STEP 4: Forming an analytic signal

V_ANAL = 2*V_MF; # make a copy and double the values
N = length(V_MF);
if mod(N,2)==0 # case N even
 neg_freq_range = Int(N/2):N; # Define range of neg-freq components
else # case N odd
 neg_freq_range = Int((N+1)/2):N;
end
V_ANAL[neg_freq_range] .= 0; # Zero out neg components in 2nd half of array.
v_anal = ifft(V_ANAL);

figure() # Create a new figure
plot(t,abs.(v_anal)) # To see the magnitude  zoom in to have a good look.
```

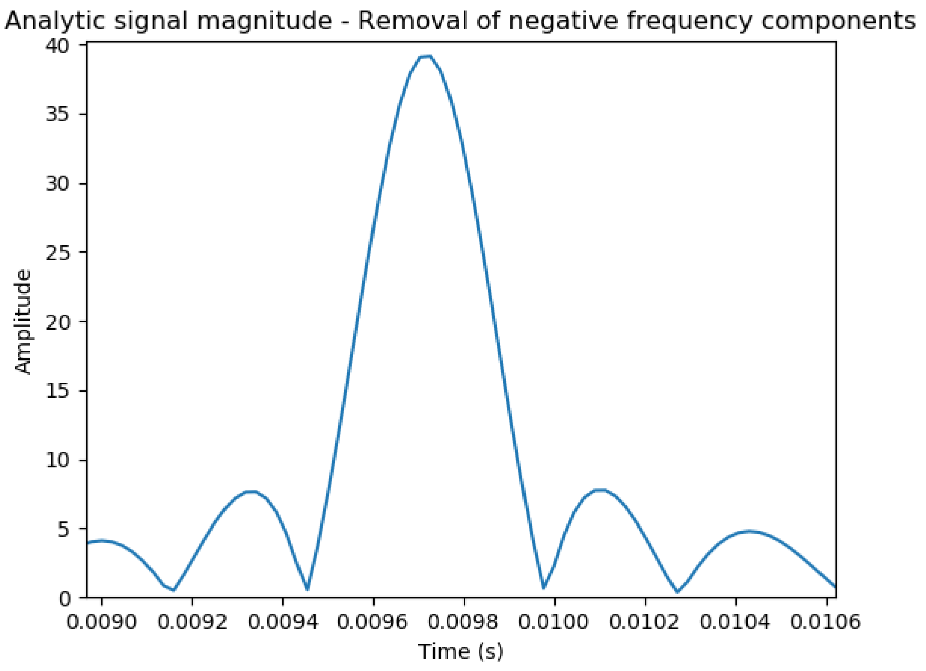
```
title("Analytic signal magnitude - Removal of negative frequency components")
xlabel("Time (s)");
ylabel("Amplitude");

figure() # Create a new figure
plot(t,angle.(v_anal))
title("Analytic signal phase - Removal of negative frequency components")
xlabel("Time (s)");
ylabel("Amplitude");
```





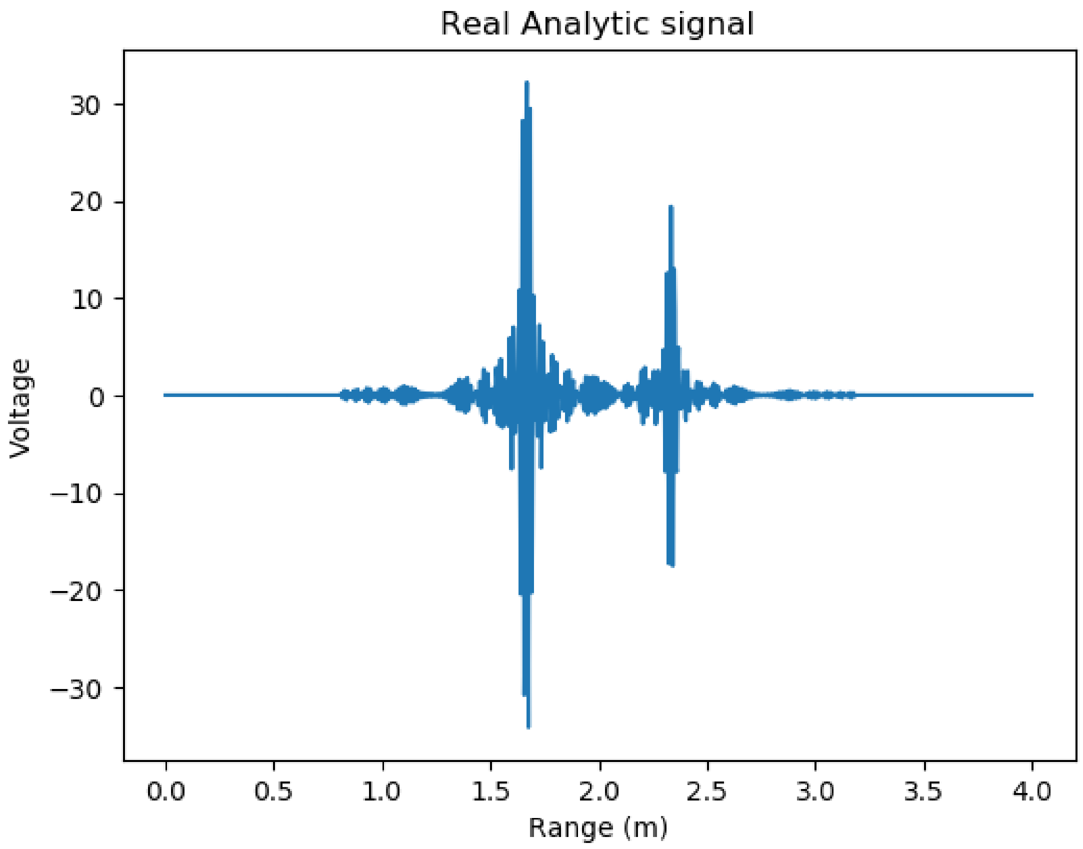
These graphs agree with the theory. The re en y domain graphs contain conjugate symme-try which implies its a purely real signal. This has been represented in the graphs plotted above which shows a negligible imaginary component.

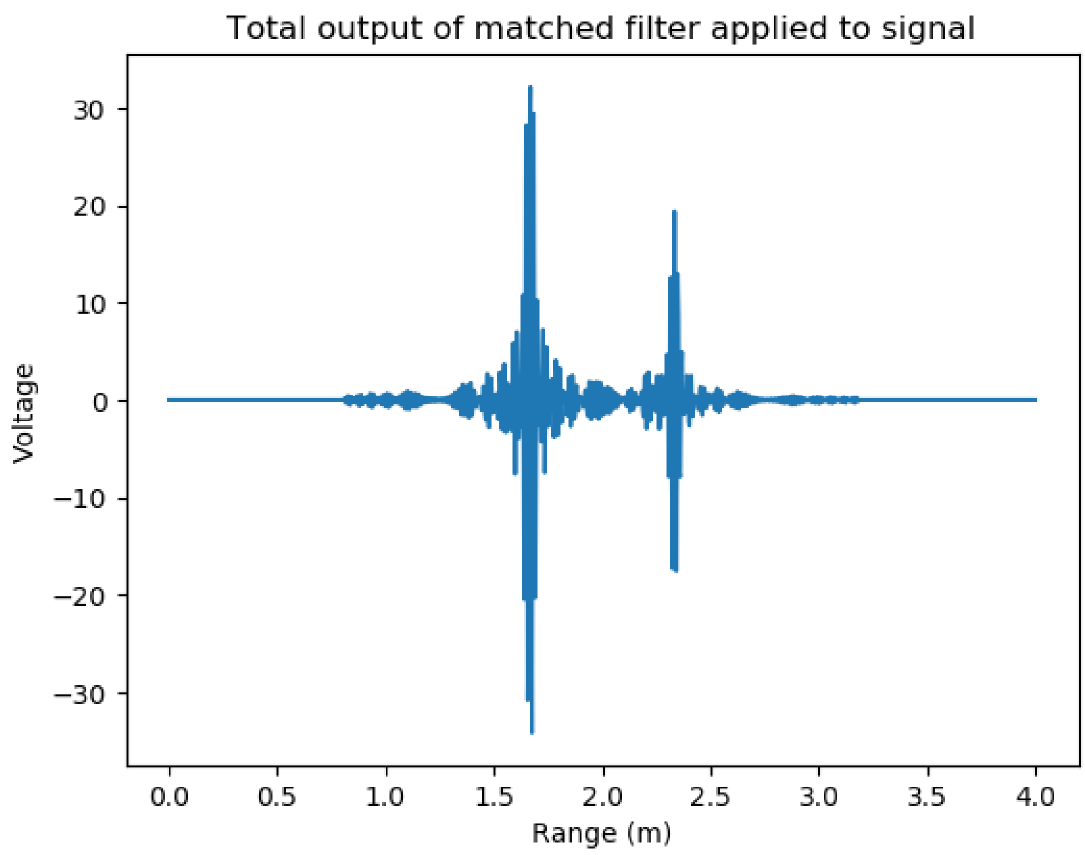
his implies the negative spectral components can be removed as they can be constructed from the p siti e components.
The 3dB point was at 39.2*0.707 =27.75
From this it could be seen that dt 0.00982712 -0.00960746= 219.66E-6 s. The theory indicates this value s ld be: dt 1/B = 1/4000 = =250E-6 s This results in a 12% error from theory.



n rder t red e t e sidel bes a ind ing te ni e ld be sed is te ni e ld
res lt in red ti n in t e l er side l bes b t a ider mainl be

[125]:
```
figure() # Create a new figure
plot(r,real(v_anal));  #compare to v_mf
title("Real Analytic signal")
xlabel("Range (m)");
ylabel("Voltage");

 figure() # Create a new figure
 plot(r,v_mf);
 title("Total output of matched filter applied to signal")
 xlabel("Range (m)");
 ylabel("Voltage");
```





The above graphs show that v_anal is the same as v_mf.

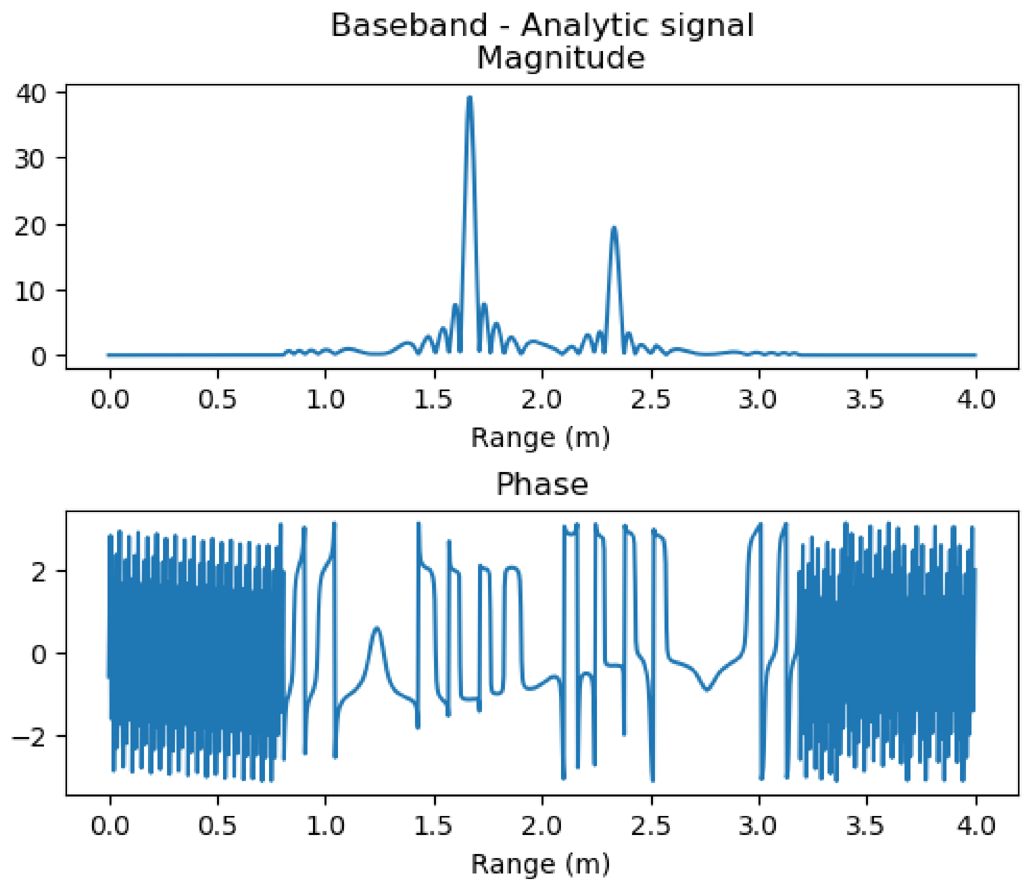
1.2 Step 5

This does agree with the theory. Multiplying by the exp(-j2pi*f*0t) causes the signal to be doubled in magnitude is also shifted to two locations, the origin and -2f0. The FFT of the baseband shows that the signal has been shifted to 0 Hz as predicted. Over the main-l be the the phase is constant.

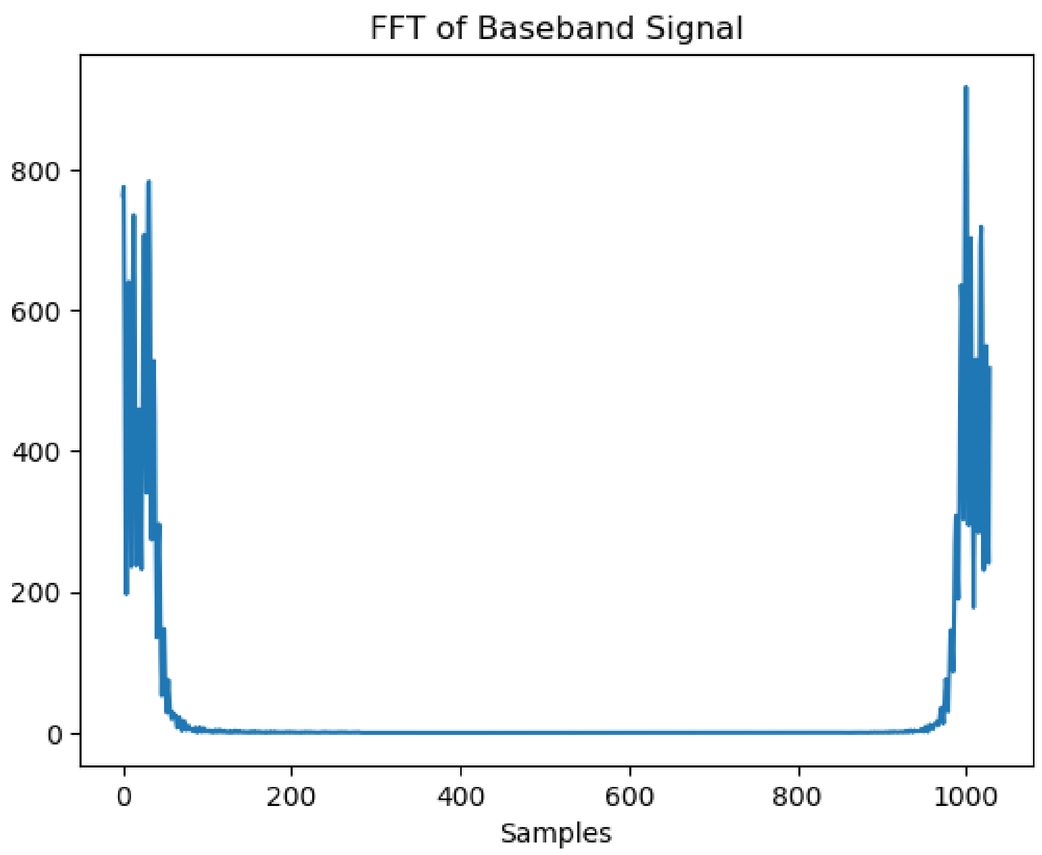
[126]:

```
#STEP 5: Translating the signal to baseband

j=im; # Assign j as sqrt(-1) (im in julia)
v_bb = v_anal.*exp.(-j*2*pi*f0*t);

fig = figure() # Create a new figure

subplot(2,1,1)
plot(r,abs.(v_bb))
```

```
title("Baseband - Analytic signal
    Magnitude")
xlabel("Range (m)");
fig.subplots_adjust(hspace=.5)
subplot(2,1,2)
plot(r,angle.(v_bb))
title("
Phase")
xlabel("Range (m)");
```



[127]:
```
V_BB = fft(v_bb);
figure() # Create a new figure
plot(abs.(V_BB))
title("FFT of Baseband Signal");
xlabel("Samples");
```

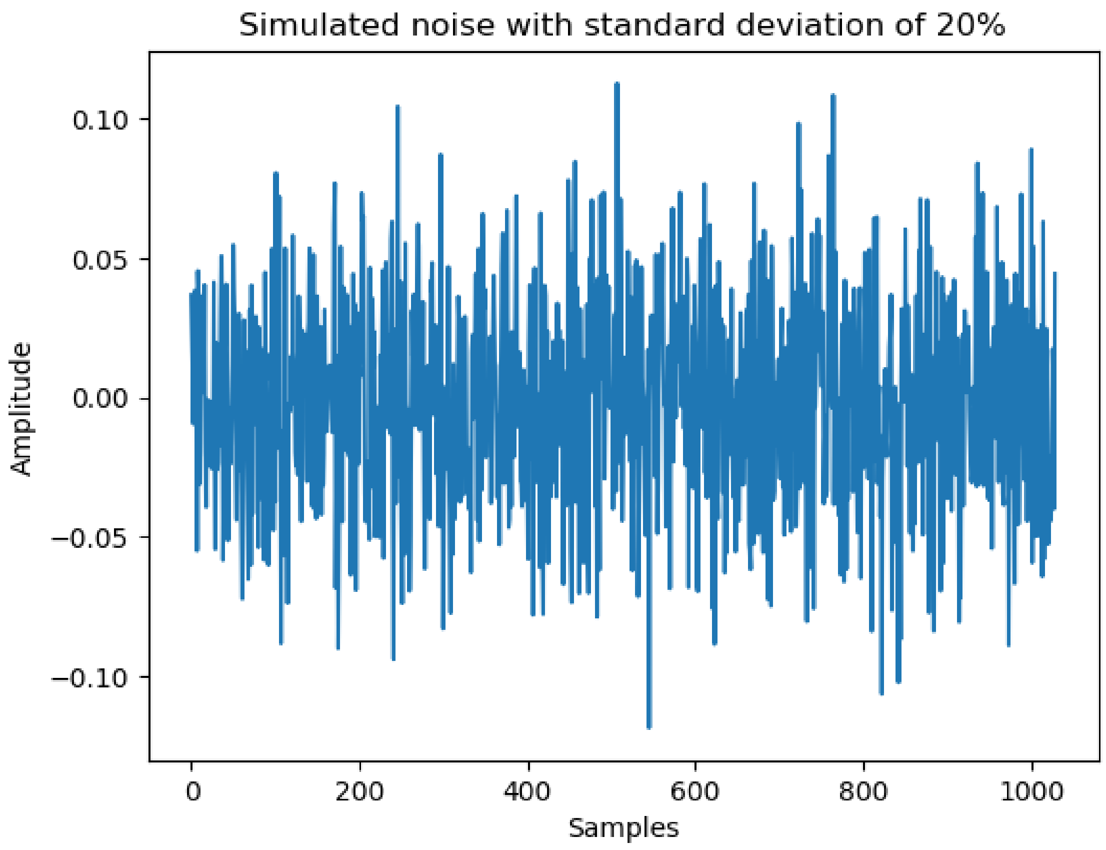


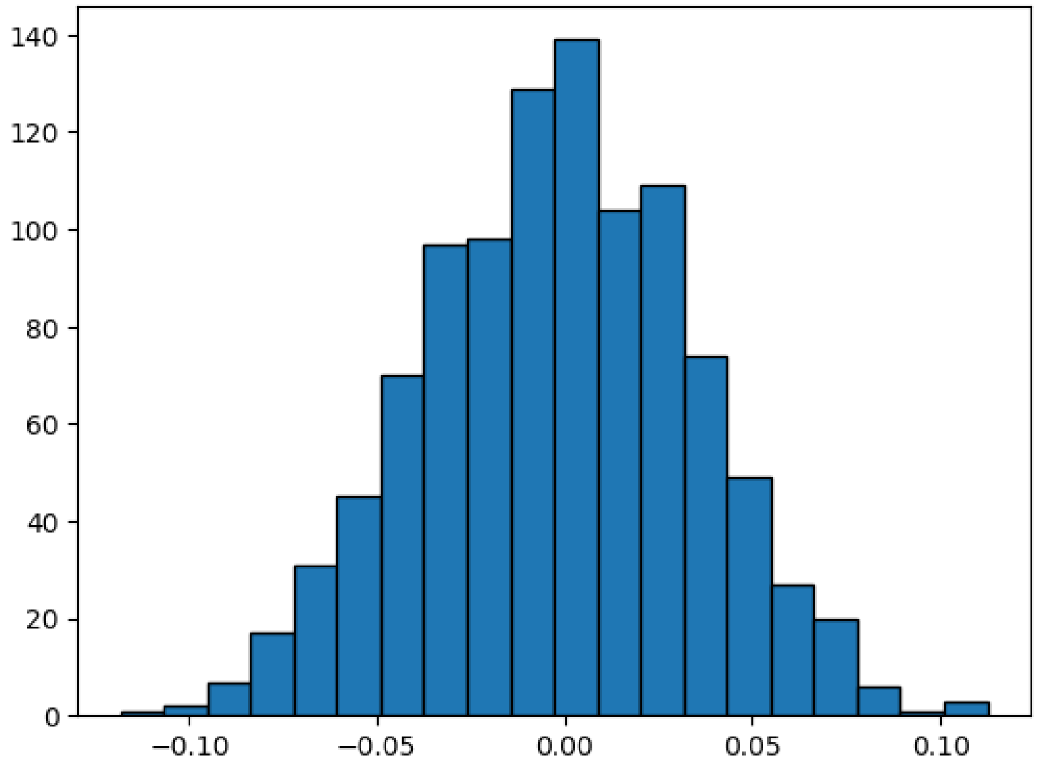
1.2.1 Step 6

Noise was added to the system and the PSD of the noise was found. The previous steps were also completed with the noise and the match filter was found output a similiar result. This was expected as the match filter outputs the peak signal to noise ratio.

[128]:
```
#STEP 6: Adding noise to the simulation

sigma = 0.2 * A1;
noise_signal = sigma * randn(N);
figure() # Create a new figure
plot(noise_signal)
title("Simulated noise with standard deviation of 20%");
xlabel("Samples");
ylabel("Amplitude");

figure() # Create a new figure
nbins=20
hist(noise_signal,nbins,edgecolor = "black")
using Statistics # import basic stats functions
```

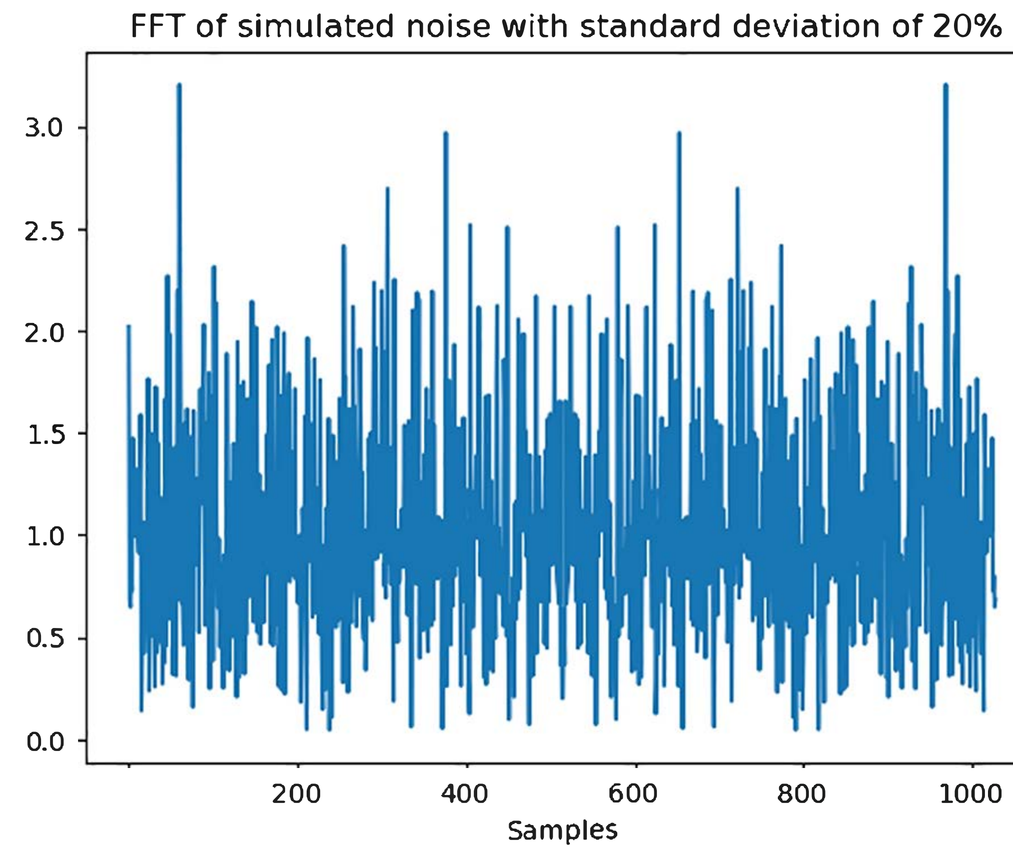
```
println("Calculated std dev = ", std(noise_signal))
```

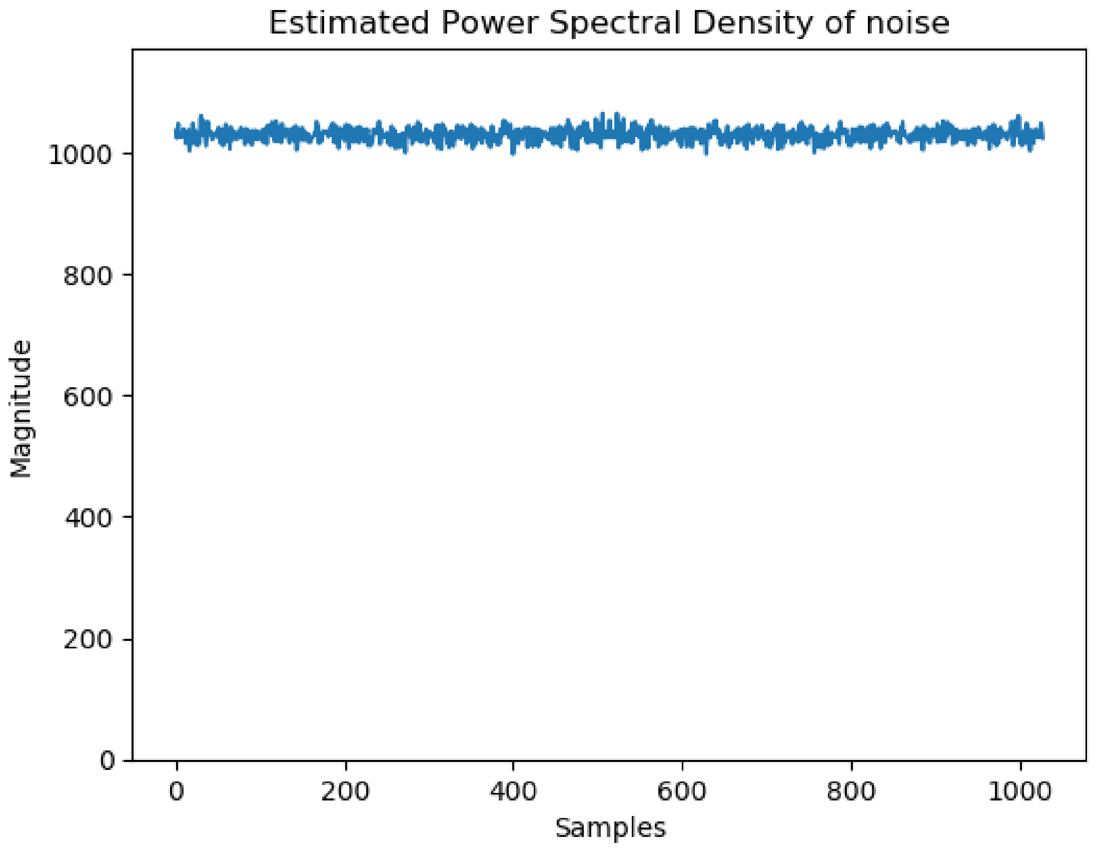




```
Calculated std dev = 0.036318872046987354
```

[129]:
```
figure() # Create a new figure
NS = fft(noise_signal);
plot(abs.(NS))
title(" FFT of simulated noise with standard deviation of 20%");
xlabel("Samples");

#PSD

N_ave = 10000; # Try different values of Nave 10, 100, 1000 ...
PSD_sum = zeros(N);
for n=1:N_ave;
global PSD_sum = PSD_sum .+ ( abs.(fft( randn(N) )) ).^2;
end
PSD_estimated = PSD_sum/N_ave;
figure();
plot(PSD_estimated);
title("Estimated Power Spectral Density of noise");
xlabel("Samples");
ylabel("Magnitude");
ylim([0,1.1*maximum(PSD_estimated)])
```





```
(0.0, 1170.9484182461692)
```

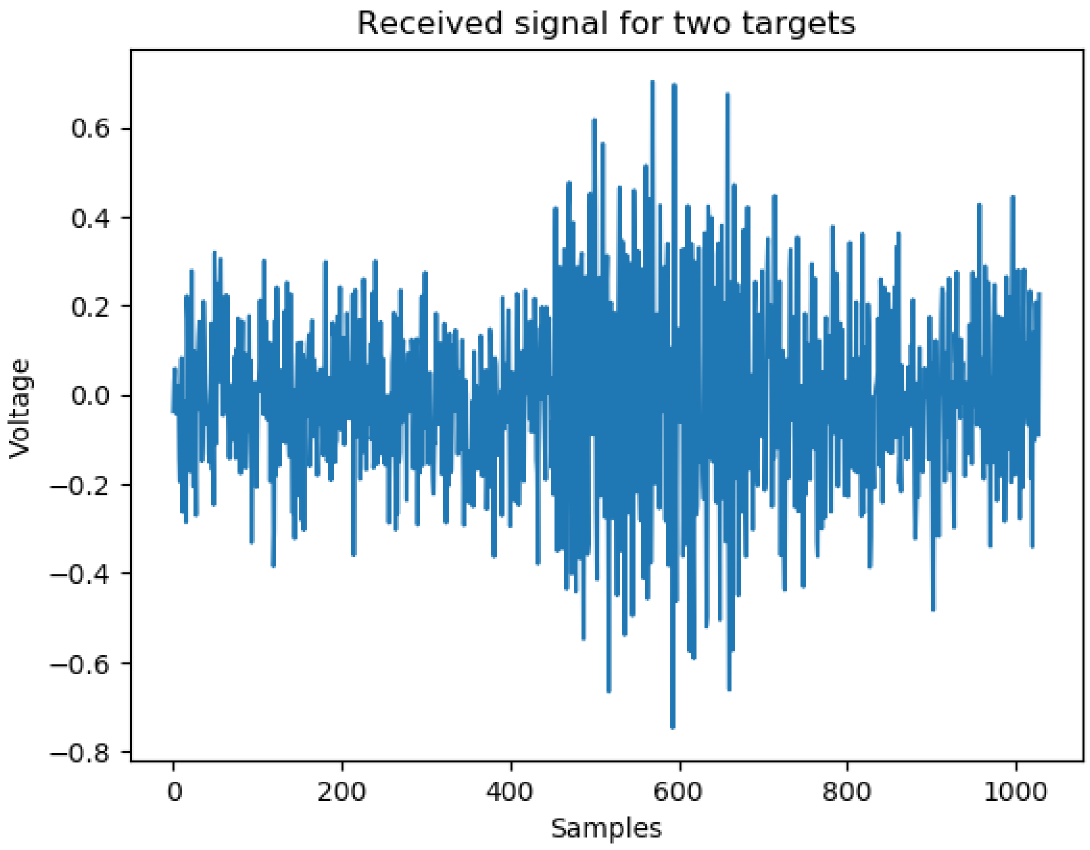
[129]:

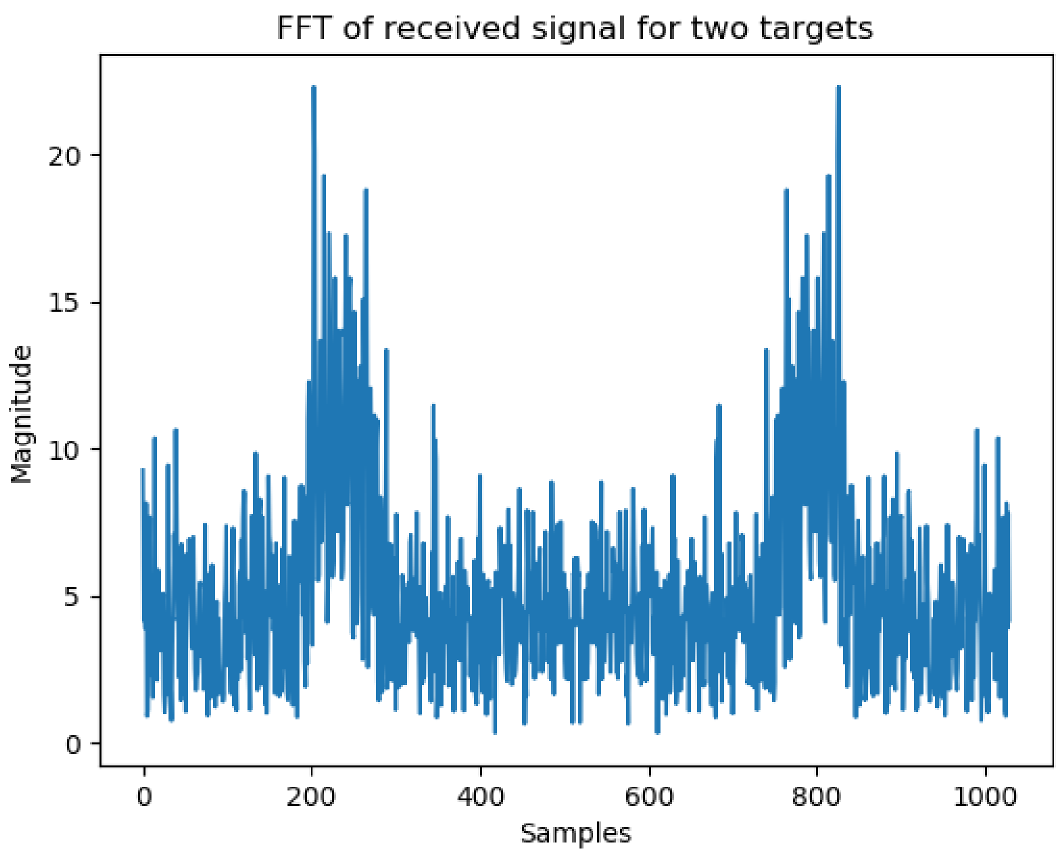
From the power spectral density graph above it can be seen that the noise is indeed white noise. This noise is constant over the entrie duration however, is not band-limited. This is due the simulation not representing the band limiting of the receiver

The standard deviation of the noise was increased to 80% on A1 to test the effectiveness of the match filter.

[28]:
```
#STEP 1: Chirp pulse creation

c = 343;  # Speed of sound in air in m/s
fs = 44100; # This is the sample rate of the sonar.
dt = 1/fs; # This is the sample spacing
r_max = 4; # Maximum range in metres to which to simulate.
t_max = 2*r_max/c; # Time delay to max range

# Create an array containing the time values of the samples
t = collect(0:dt:t_max); # t=0:dt:t_max defines a range.
# Create an array containing the range values of the samples
r = c*t/2;
# NOW create the chirp pulse, shifted by an amount td, to start at
# some time td-T/2>0.
f0 = 10000; # Centre frequency is 10 kHz
B = 4000; # Chirp bandwidth
T = 5E-3; # Chirp pulse length
K = B/T; # Chirp rate
# Define a simple a rect() function which returns for -0.5<=t<=0.5 or 0.
# The function will work if t is an array of values.
rect(t) = (abs.(t) .<= 0.5)*1.0
# rect(t/T) spans the interval [-T/2,T/2]
# We must therefore delay the chirp pulse so that it starts after t=0.
# Shift the chirp pulse by 0.6T units to the right, so that it starts at
0.1*T
td = 0.6*T; # Chirp delay
# Note: one can use the macro @. to avoid having to put . for arrays:
# @. v_tx = cos( 2*pi*(f0*(t-td) + 0.5*K*(t-td).^2) ).*rect((t-td)/T);
v_tx = cos.( 2*pi*(f0*(t .- td) + 0.5*K*(t .- td).^2) ) .* rect.((t .-td)/T);

#LABEL frequency axis
N = length(t);
f = 1/(N*dt) # spacing in frequency domain
#create array of freq values stored in f_axis. First element maps to 0Hz
f_axis = (0:N-1)*f;
```

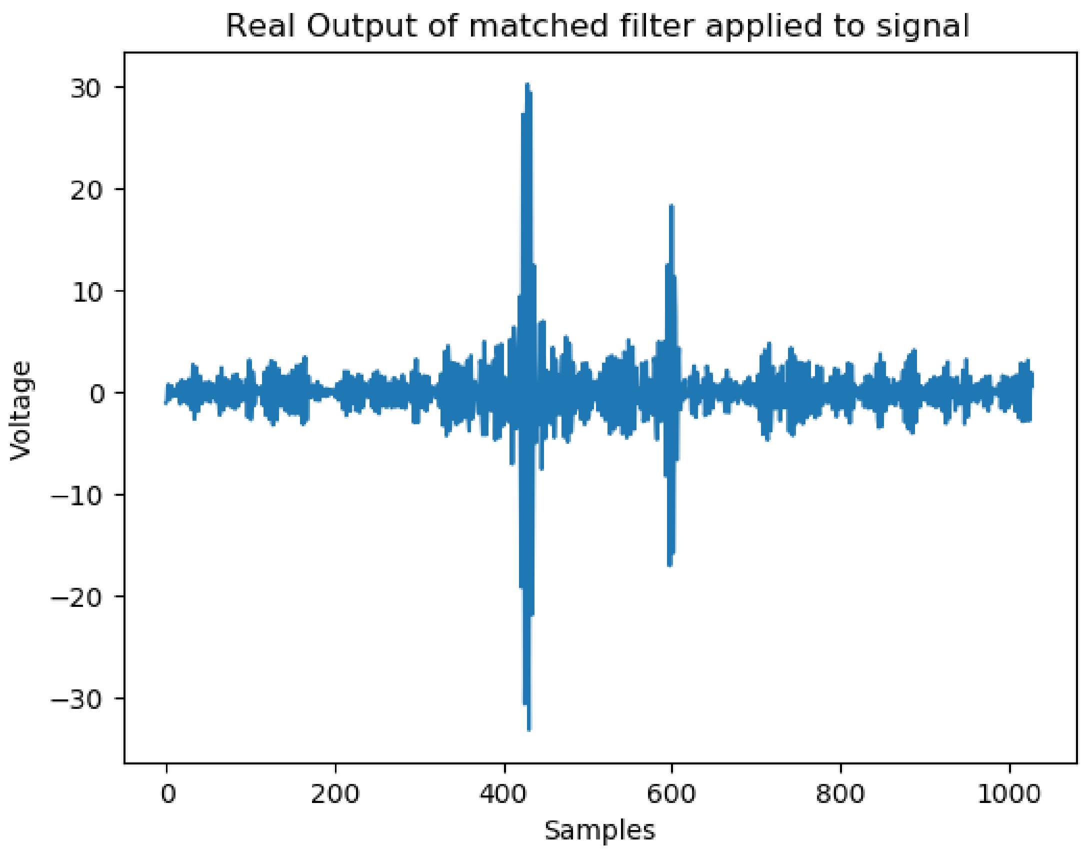
```julia
using PyPlot; pygui(false) # import plot library
using FFTW

#1st Target
R1 = 1.5 + (12-2)/12 # 2.33m - range to target.
td1 = 2*R1/c; # two way delay to target.
A1 = 1/R1^2; # echo voltage signal proportional to 1/R^2
v_rx = A1*cos.( 2*pi*(f0*(t.-td.-td1) + 0.5*K*(t.-td.-td1).^2) ) .* rect((t.-
td.-td1)/T);


#second target
R2 =  1.5 + (12-10)/12# 1.67m - range to target.
td2 = 2*R2/c; # two way delay to target.
A2 = 1/R2^2; # echo voltage signal proportional to 1/R^2
v_rx = A1*cos.(2*pi*(f0*(t.-td.-td1) + 0.5*K*(t.-td.-td1).^2) ) .* rect((t.-
td.-td1)/T) + A2*cos.(2*pi*(f0*(t.-td.-td2) + 0.5*K*(t.-td.-td2).^2) ) .*␣
 ↪rect((t.-
td.-td2)/T);
V_RX = fft(v_rx);

sigma = 0.8 * A1;
noise_signal = sigma * randn(N);

v_rx = A1*cos.(2*pi*(f0*(t.-td.-td1) + 0.5*K*(t.-td.-td1).^2) ) .* rect((t.-
td.-td1)/T) + A2*cos.(2*pi*(f0*(t.-td.-td2) + 0.5*K*(t.-td.-td2).^2) ) .*␣
 ↪rect((t.-
td.-td2)/T) + noise_signal;
V_RX = fft(v_rx);

figure() # Create a new figure
plot(v_rx) # Put time on x-axis
title("Received signal for two targets")
xlabel("Samples");
ylabel("Voltage");

figure() # Create a new figure
plot(abs.(V_RX) )
title("FFT of received signal for two targets");
xlabel("Samples");
ylabel("Magnitude");
```

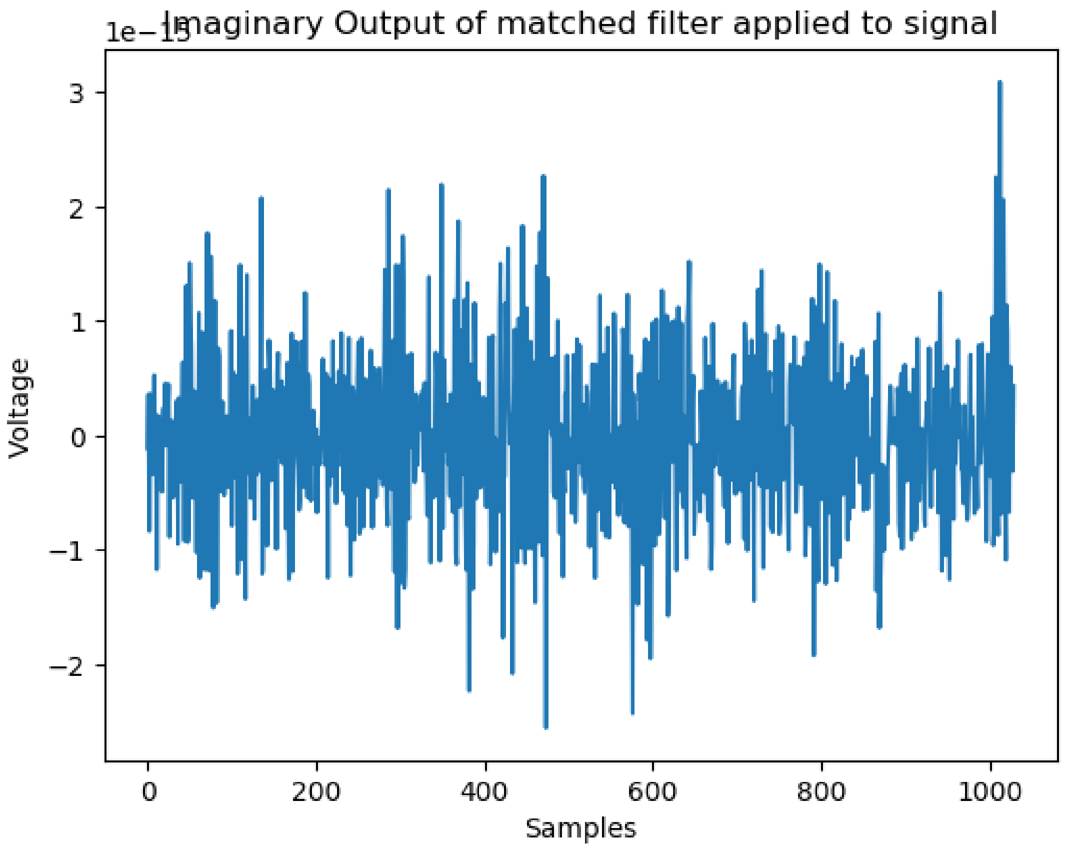

Received signal for two targets
Voltage
Samples
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
0
200
400
600
800
1000



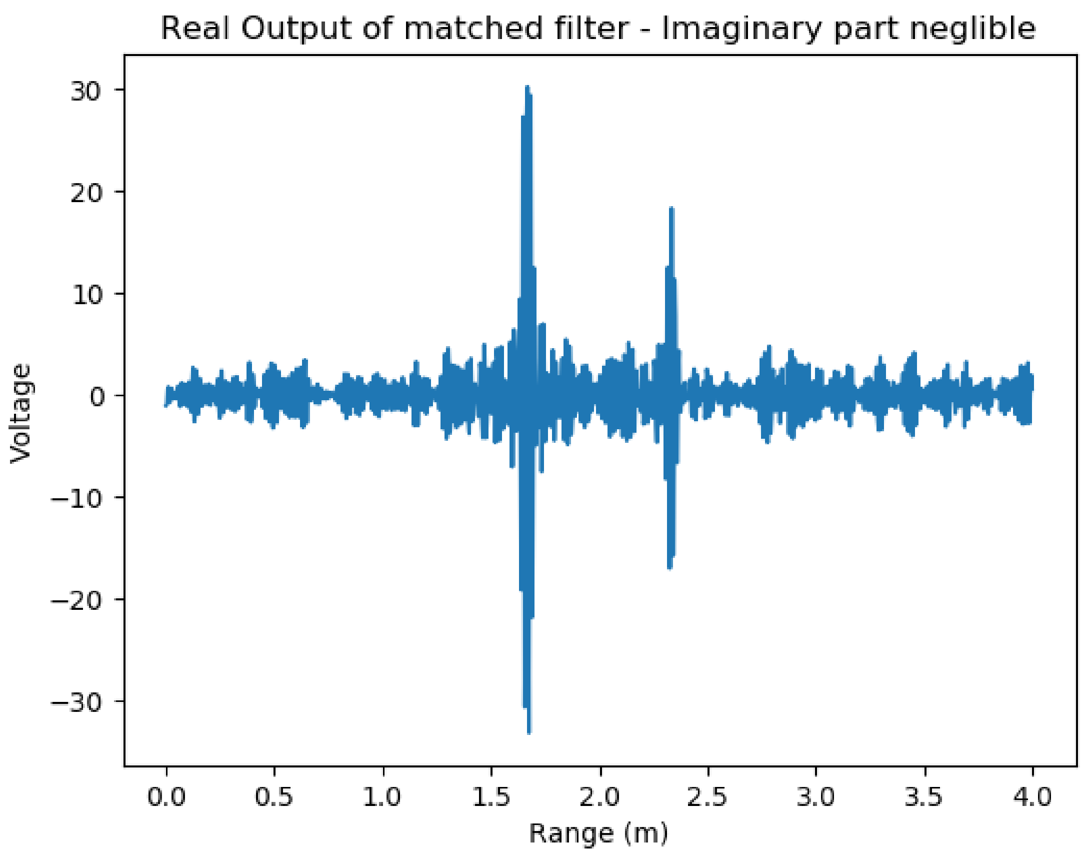
[29]:
```
#STEP 3: Matched filtering
V_TX = fft(v_tx);
H = conj(V_TX);
V_MF = H.*V_RX;
v_mf = ifft(V_MF);

figure() # Create a new figure
plot(real(v_mf));
title("Real Output of matched filter applied to signal")
xlabel("Samples");
ylabel("Voltage");

figure() # Create a new figure
plot(imag(v_mf));
title("Imaginary Output of matched filter applied to signal")
xlabel("Samples");
ylabel("Voltage");

#Negligble Imaginary part <1E-14
```

```
v_mf = real(v_mf);
figure() # Create a new figure
plot(r,v_mf) # To see the detail  zoom in to have a good look.
title("Real Output of matched filter - Imaginary part neglible")
xlabel("Range (m)");
ylabel("Voltage");


# Take note of the shape of the envelope, as well as the internal detail.
# effect of pulse compression
```



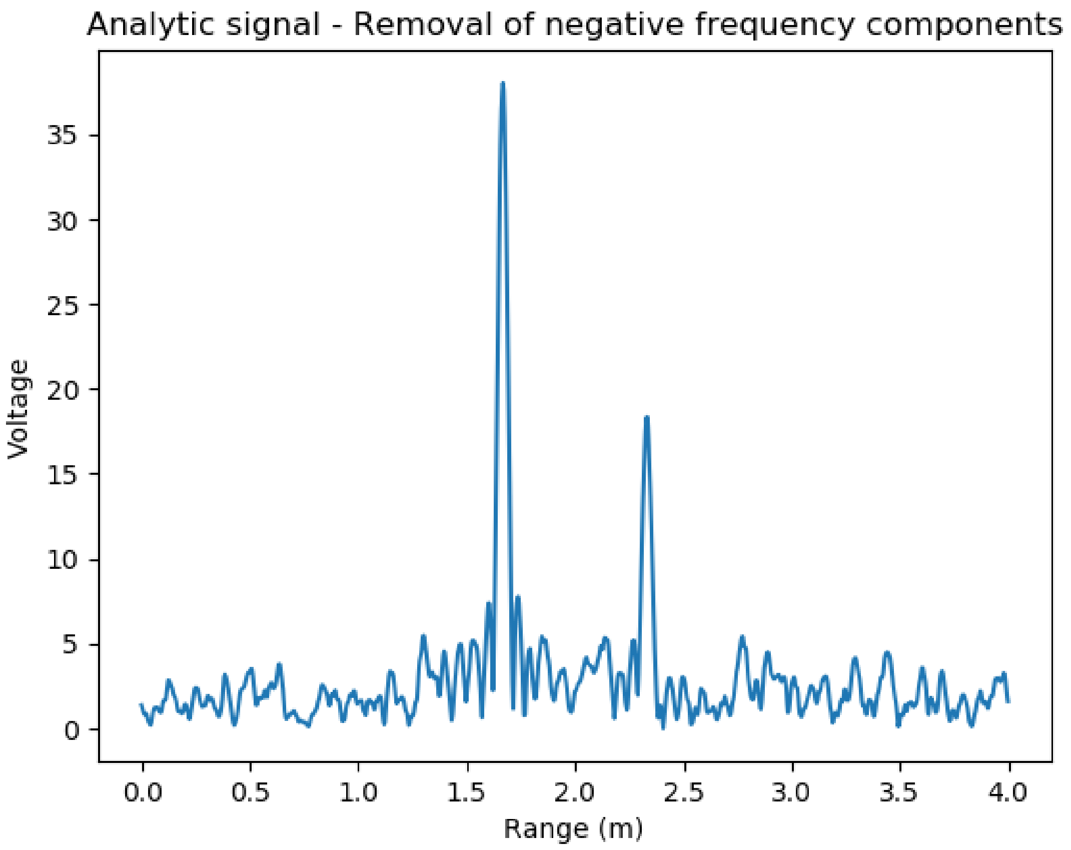


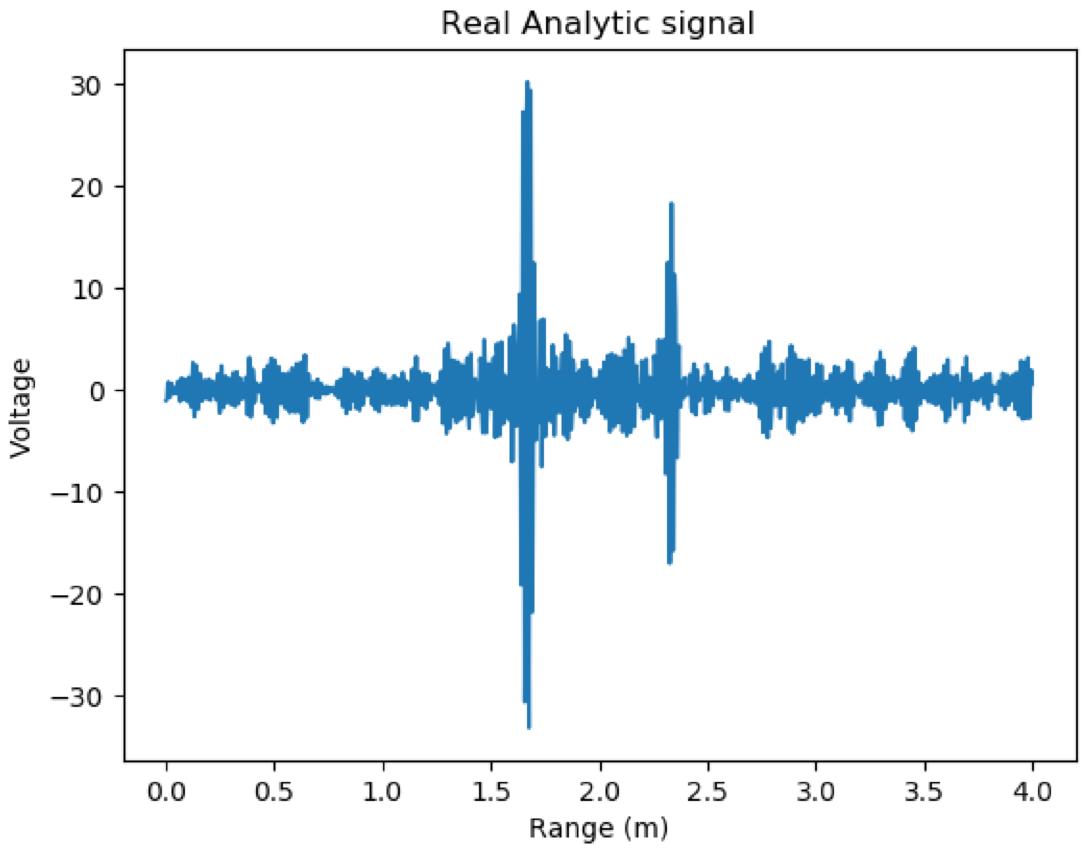
Imaginary Output of matched filter applied to signal
1e−15
3
2
1
0
−1
−2
Voltage
0
200
400
600
800
1000
Samples



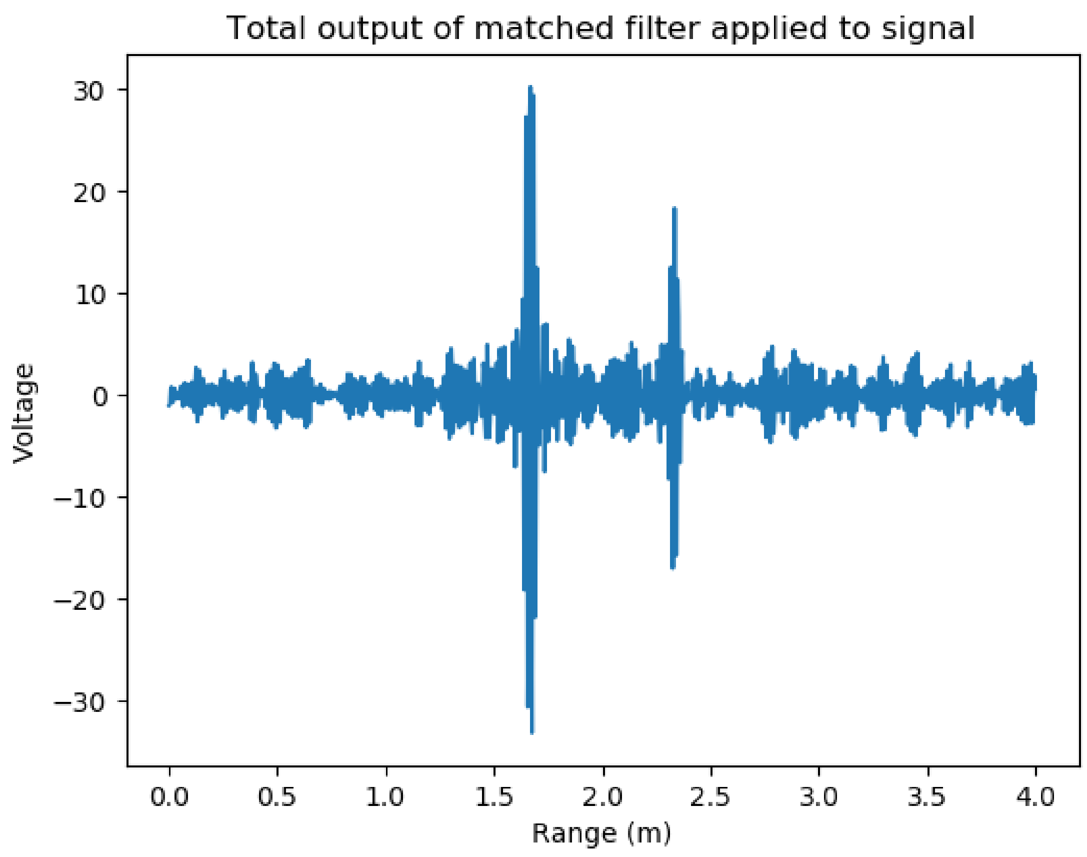
[30]:
```
#STEP 4: Forming an analytic signal

V_ANAL = 2*V_MF; # make a copy and double the values
N = length(V_MF);
V_ANAL = 2*V_MF; # make a copy and double the values
N = length(V_MF);
if mod(N,2)==0 # case N even
 neg_freq_range = Int(N/2):N; # Define range of neg-freq components
else # case N odd
 neg_freq_range = Int((N+1)/2):N;
end
V_ANAL[neg_freq_range] .= 0; # Zero out neg components in 2nd half of array.
v_anal = ifft(V_ANAL);

figure() # Create a new figure
plot(r,abs.(v_anal)) # To see the magnitude  zoom in to have a good look.
title("Analytic signal - Removal of negative frequency components")
xlabel("Range (m)");
ylabel("Voltage");
```

```
figure() # Create a new figure
plot(r,real(v_anal));  #compare to v_mf
title("Real Analytic signal")
xlabel("Range (m)");
ylabel("Voltage");

figure() # Create a new figure
plot(r,v_mf);
title("Total output of matched filter applied to signal")
xlabel("Range (m)");
ylabel("Voltage");
```



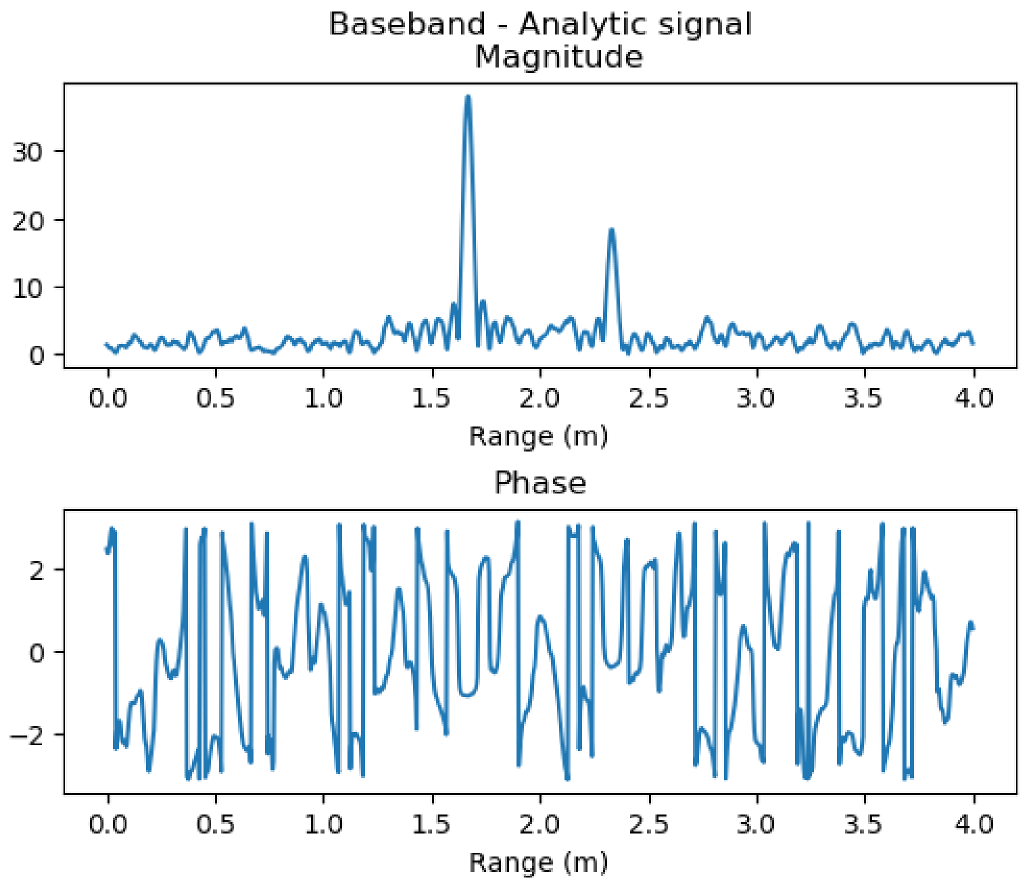


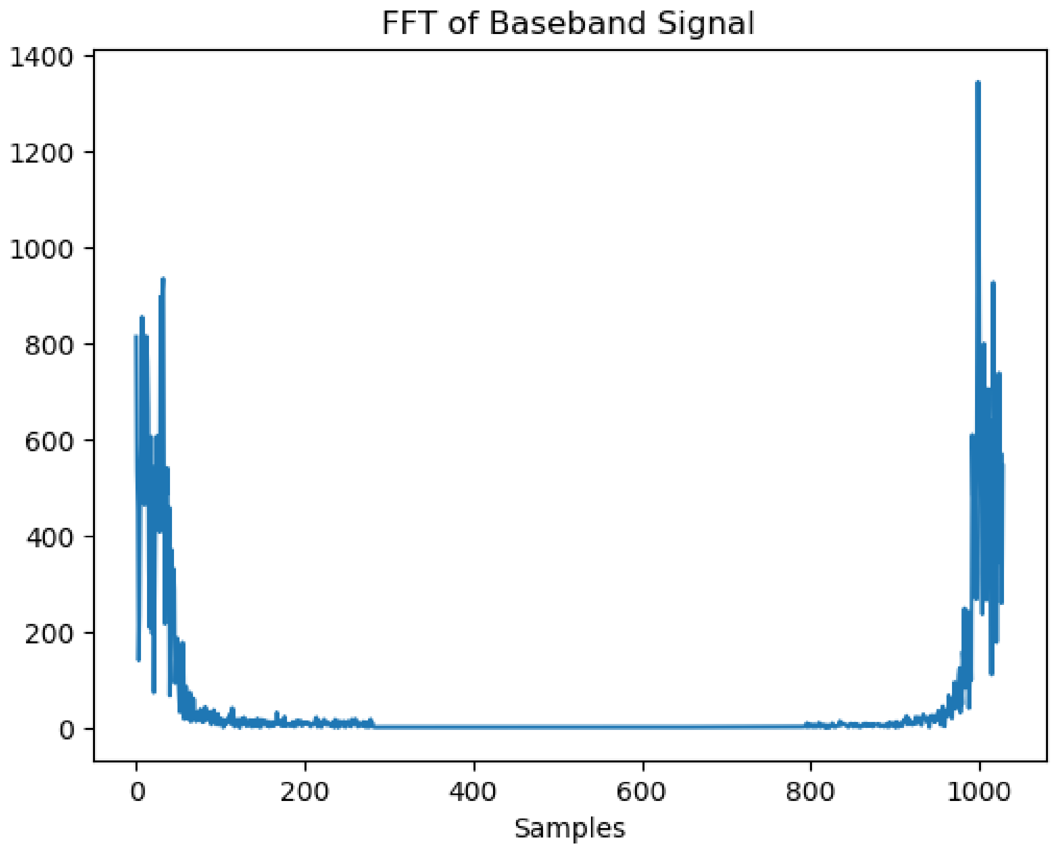
Real Analytic signal
Voltage
Range (m)
30
20
10
0
−10
−20
−30
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0



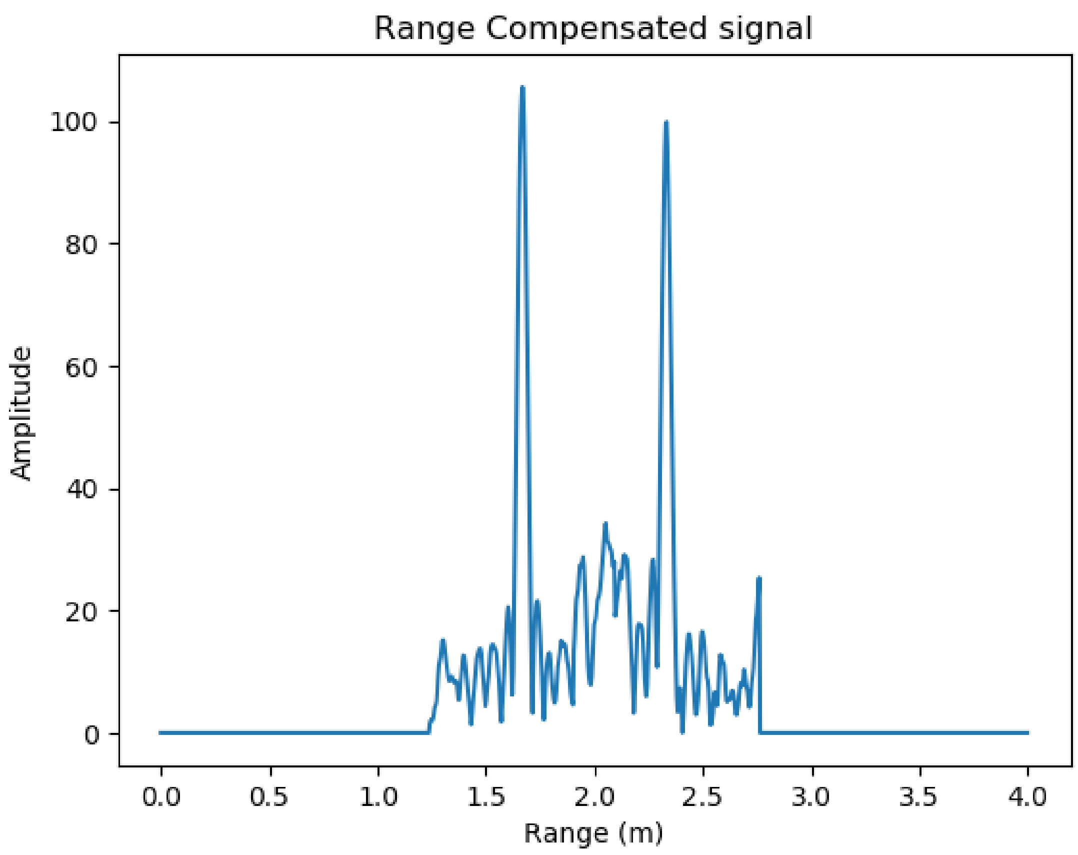
[31]:
```
#STEP 5: Translating the signal to baseband

j=im; # Assign j as sqrt(-1) (im in julia)
v_bb = v_anal.*exp.(-j*2*pi*f0*t);

fig = figure() # Create a new figure

subplot(2,1,1)
plot(r,abs.(v_bb))
title("Baseband - Analytic signal
    Magnitude")
xlabel("Range (m)");
fig.subplots_adjust(hspace=.5)
subplot(2,1,2)
plot(r,angle.(v_bb))
title("
Phase")
xlabel("Range (m)");
```

```
V_BB = fft(v_bb);
figure() # Create a new figure
plot(abs.(V_BB))
title("FFT of Baseband Signal");
xlabel("Samples");
```





[32]:
```
#Part 7

# Define a simple a rect() function which returns for -0.25<=t<=0.25 or 0.
# The function will work if t is an array of values.
rect2(t) = (abs.(t) .<= 0.5)*1.0
R1_func = R1^2
R2_func = R2^2
v_rc = v_bb.* rect2.((t.-td1)/T)*R1_func .+ v_bb.* rect2.((t .-td2)/T)*R2_func;

figure() # Create a new figure
plot(r,real(abs.(v_rc)));
title("Range Compensated signal")
xlabel("Range (m)");
ylabel("Amplitude");
```



From the compensation profile it appears the noise diminishes at is gets further way from the mainlobes.

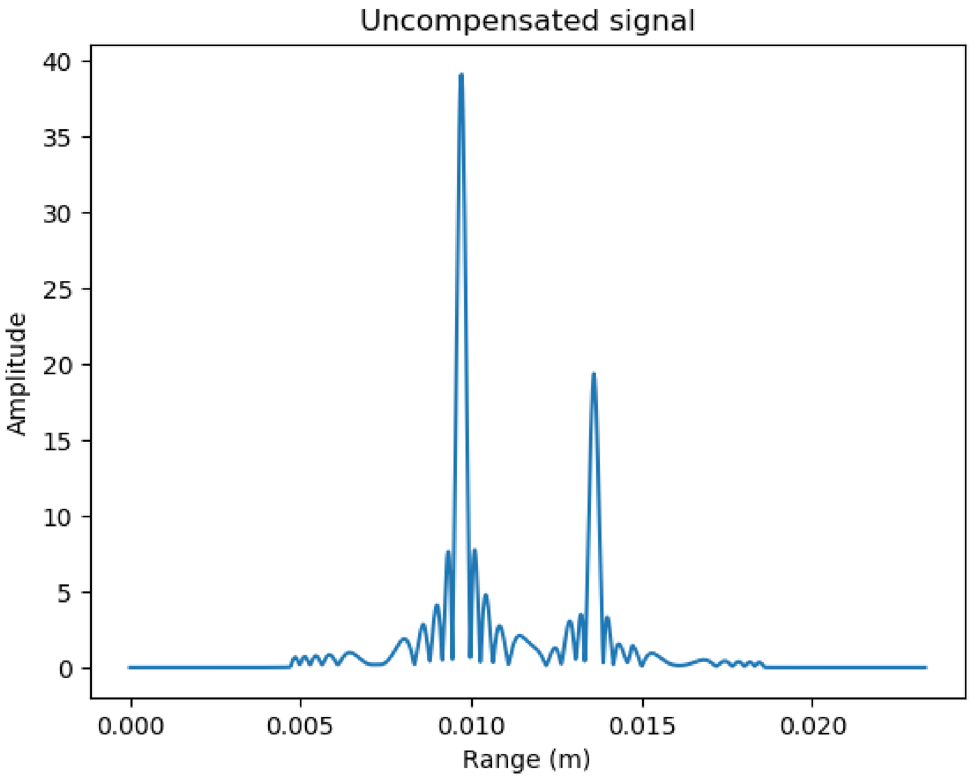
From the above graphs it can be clearly seen that the match filter worked as expected. This is shown by the similiar result even with extremely high noise ~ 80%

1.2.2 Step 7

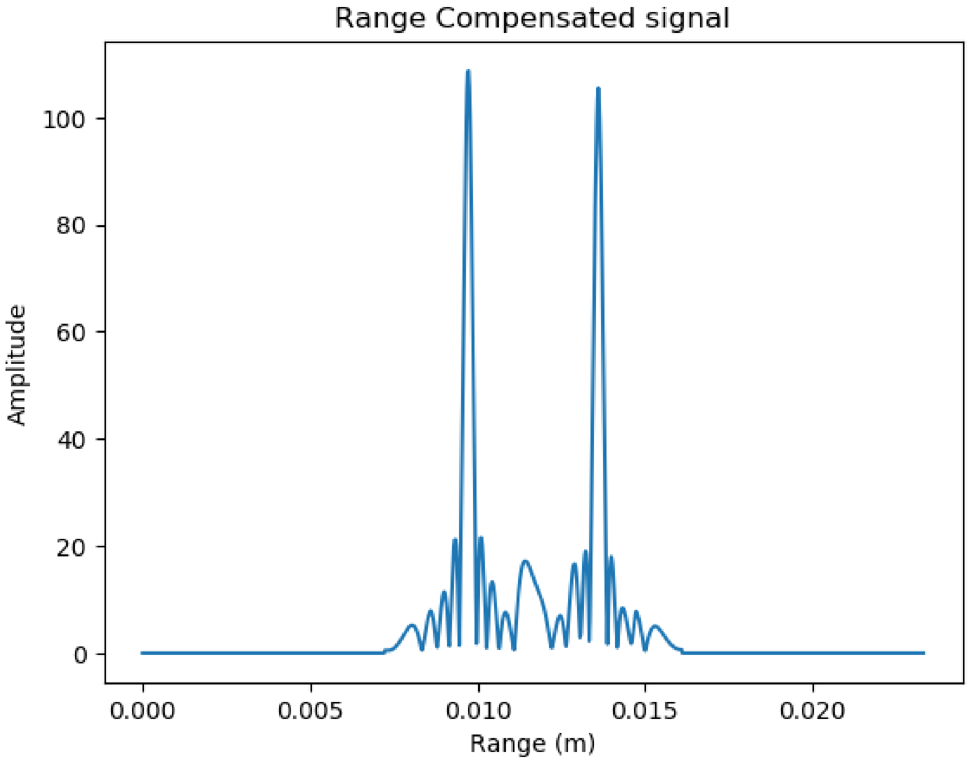
In order to compensate for the inverse square relationship between the recived signal and the distance, range compensation was performed. This was done using two rect functions at each of the targets locations and multiplying by a factor of Rˆ2. The figures below show that this compensation corrected the smaller peak of the target located further away.

[130]:
```
#Part 7

# Define a simple a rect() function which returns for -0.25<=t<=0.25 or 0.
# The function will work if t is an array of values.
rect2(t) = (abs.(t) .<= 0.25)*1.0
R1_func = R1^2
R2_func = R2^2
v_rc = v_bb.* rect.((t.-td1)/T)*R1_func .+ v_bb.* rect.((t .-td2)/T)*R2_func;
```

```
figure() # Create a new figure
plot(t,real(abs.(v_bb)));
title("Uncompensated signal")
xlabel("Range (m)");
ylabel("Amplitude");

figure() # Create a new figure
plot(t,real(abs.(v_rc)));
title("Range Compensated signal")
xlabel("Range (m)");
ylabel("Amplitude");
```



Uncompensated signal
Amplitude
Range (m)
0
5
10
15
20
25
30
35
40
0.000
0.005
0.010
0.015
0.020



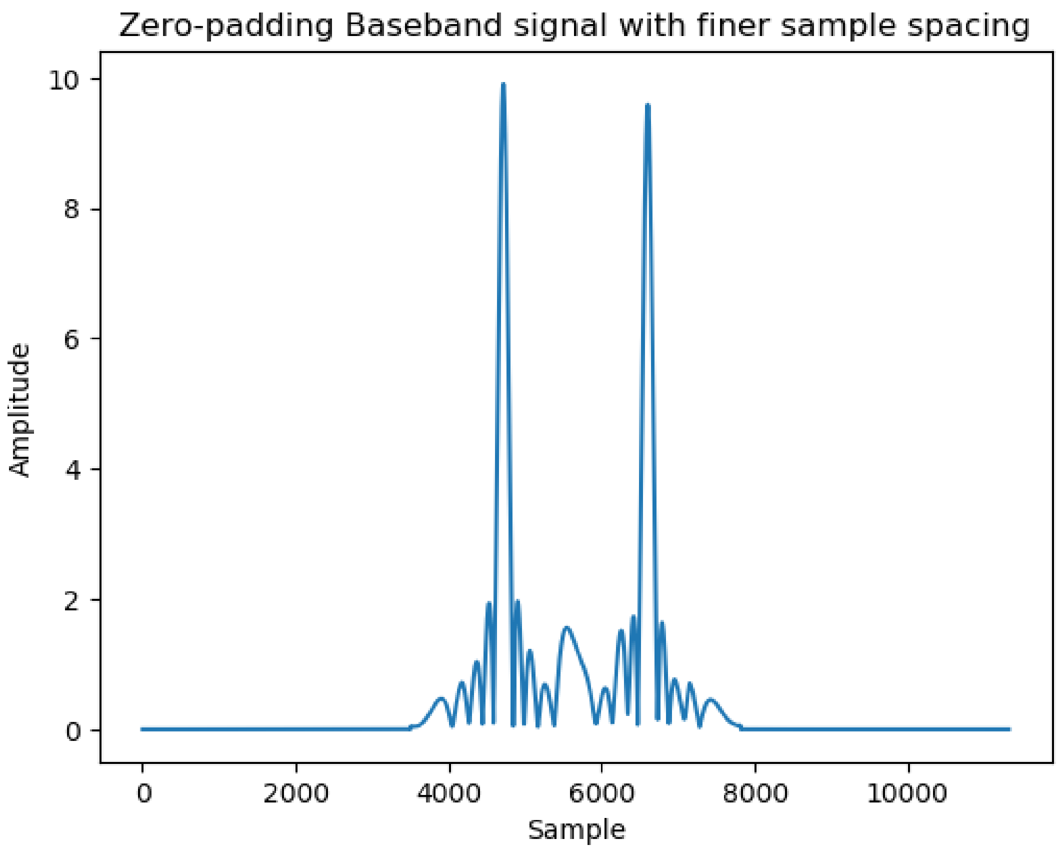
Range Compensated signal
Amplitude
Range (m)
0
20
40
60
80
100
0.000
0.005
0.010
0.015
0.020

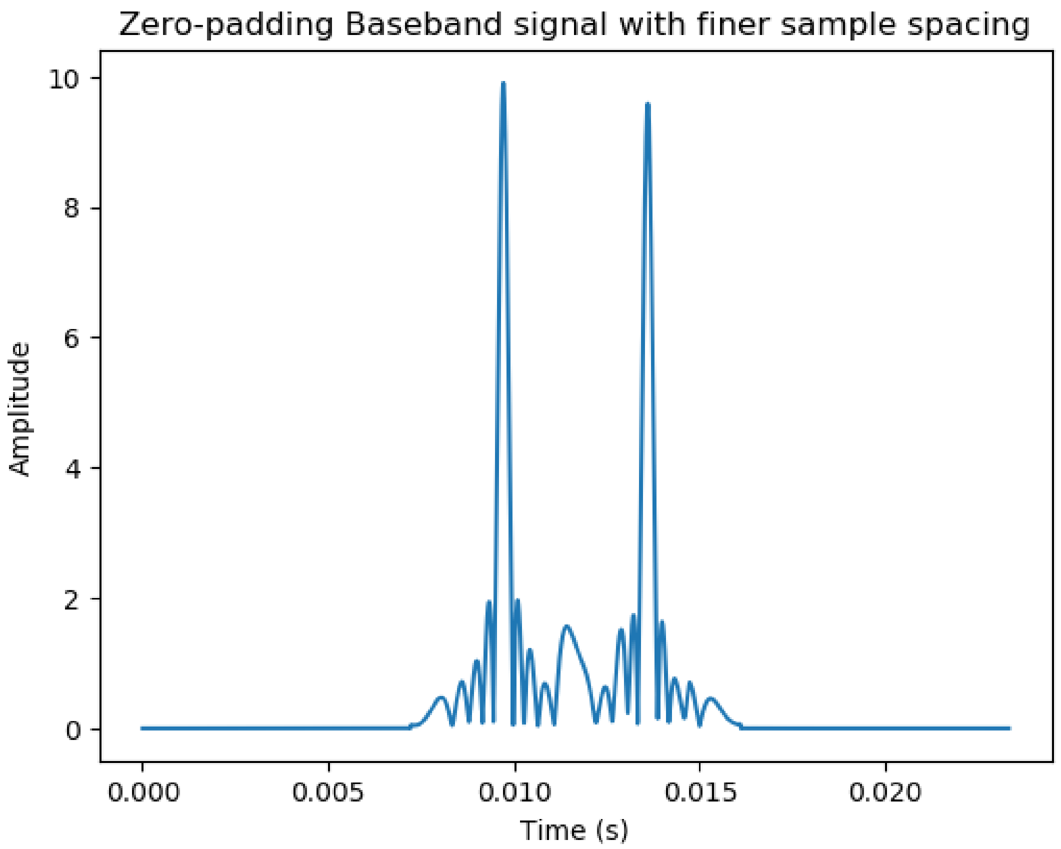
1.3 Step 8

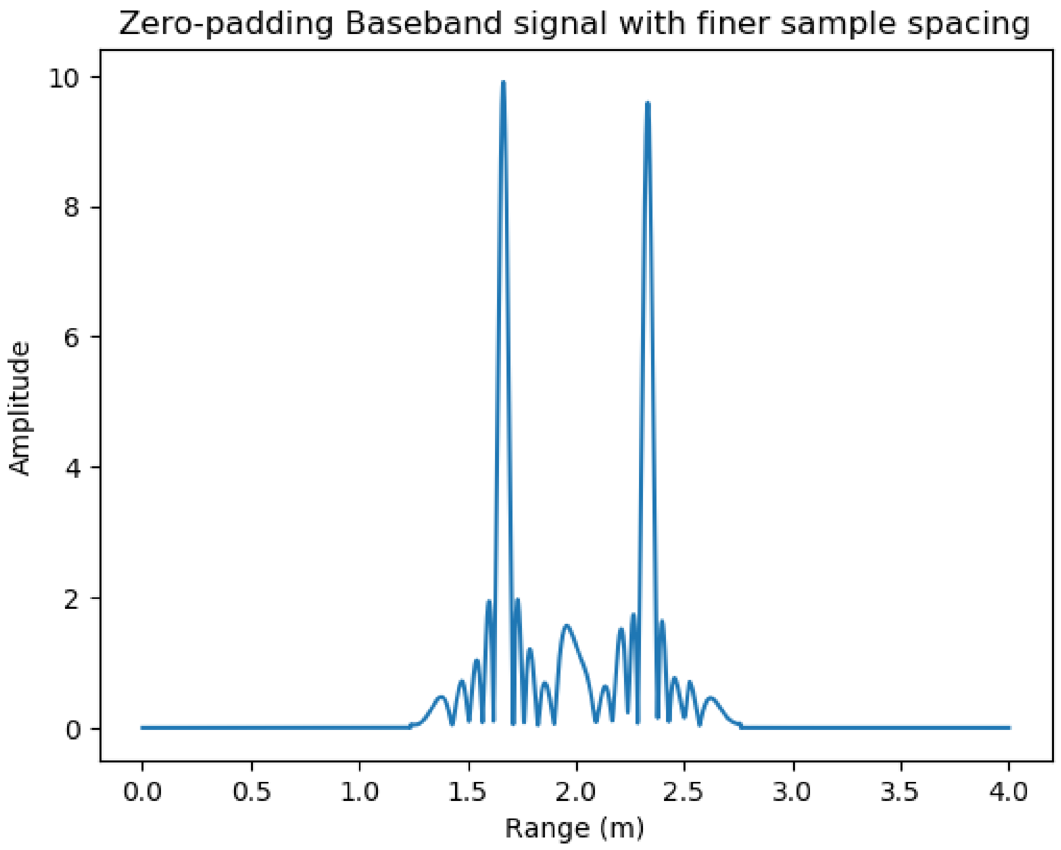
The zero padding technique was performed on the compensated signal in order to smooth out the waveforms and allow for closer spaced samples. The sample spacing for the time and range had to be recalculated.

[131]:
```
#Part 8
Empty  = zeros(10*N)
p = floor(Int32, N/2)
V_RC_Padded = fft(v_rc)
temp = vcat(V_RC_Padded[1:p-1], Empty)
X = vcat(temp, V_RC_Padded[p:end])
x = ifft(X);
time_new = collect(0:dt/11.004:t_max);
range_new = c*time_new/2;

figure()
plot(real(abs.(x)));
title("Zero-padding Baseband signal with finer sample spacing")
xlabel("Sample");
ylabel("Amplitude");
```

```

using PyPlot;
figure()
plot(time_new,real(abs.(x)));
title("Zero-padding Baseband signal with finer sample spacing")
xlabel("Time (s)");
ylabel("Amplitude");


figure()
plot(range_new,real(abs.(x)));
title("Zero-padding Baseband signal with finer sample spacing")
xlabel("Range (m)");
ylabel("Amplitude");
```


Zero-padding Baseband signal with finer sample spacing
Amplitude
Sample
0
2
4
6
8
10
0
2000
4000
6000
8000
10000


Zero-padding Baseband signal with finer sample spacing
Amplitude
Time (s)
0
2
4
6
8
10
0.000
0.005
0.010
0.015
0.020


Zero-padding Baseband signal with finer sample spacing
Amplitude
Range (m)
0
2
4
6
8
10
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

0.0.1 Part 9 - Sampling below the usual Nyquist rate

The above steps were repeated but using a lower sample rate of 4*B. This was done in order to check if the information was preserved.

[171]:
```
#STEP 1: Chirp pulse creation

c = 343;  # Speed of sound in air in m/s
B = 4000; # Chirp bandwidth
fs = 4*B; # This is the sample rate of the sonar.
dt = 1/fs; # This is the sample spacing
r_max = 4; # Maximum range in metres to which to simulate.
t_max = 2*r_max/c; # Time delay to max range

# Create an array containing the time values of the samples
t = collect(0:dt:t_max); # t=0:dt:t_max defines a range.
# Create an array containing the range values of the samples
r = c*t/2;
# NOW create the chirp pulse, shifted by an amount td, to start at
# some time td-T/2>0.
f0 = 10000; # Centre frequency is 10 kHz
T = 5E-3; # Chirp pulse length
K = B/T; # Chirp rate
# Define a simple a rect() function which returns for -0.5<=t<=0.5 or 0.
# The function will work if t is an array of values.
rect(t) = (abs.(t) .<= 0.5)*1.0
# rect(t/T) spans the interval [-T/2,T/2]
# We must therefore delay the chirp pulse so that it starts after t=0.
# Shift the chirp pulse by 0.6T units to the right, so that it starts at
0.1*T
td = 0.6*T; # Chirp delay
# Note: one can use the macro @. to avoid having to put . for arrays:
# @. v_tx = cos( 2*pi*(f0*(t-td) + 0.5*K*(t-td).^2) ).*rect((t-td)/T);
v_tx = cos.( 2*pi*(f0*(t .- td) + 0.5*K*(t .- td).^2) ) .* rect.((t .-td)/T);


using PyPlot; pygui(false) # import plot library
# If not installed, add the package via: using Pkg; Pkg.add("PyPlot");
```

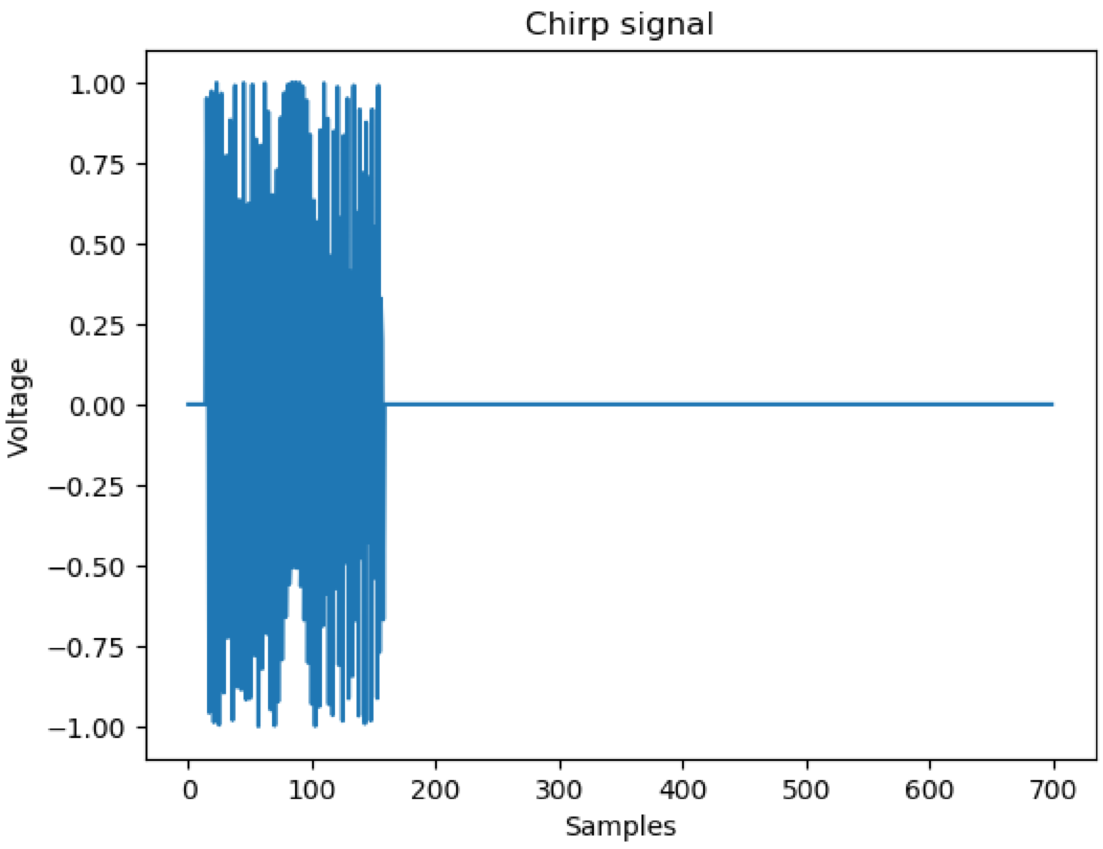
1 3097S Milestone 1: Sonar chirp pulse simulation and processing

2 Part 2 - 40 kHz ultrasound transducer

The bandwidth of the 40 kHz transducer was found to be 2 or 2.5 kHz depending if it was a transmitting or recieving transducer. For this report the bandwidth of 2 kHz was used. A value of 6 was used for k as the value 4 did not provide good results.

[313]:
```
#STEP 1: Chirp pulse creation

c = 343;  # Speed of sound in air in m/s
B = 2000; # Chirp bandwidth
k = 6;
fs = k*B; # This is the sample rate of the sonar.
dt = 1/fs; # This is the sample spacing
r_max = 10; # Maximum range in metres to which to simulate.
t_max = 2*r_max/c; # Time delay to max range

# Create an array containing the time values of the samples
t = collect(0:dt:t_max); # t=0:dt:t_max defines a range.
# Create an array containing the range values of the samples
r = c*t/2;
# NOW create the chirp pulse, shifted by an amount td, to start at
# some time td-T/2>0.
f0 = 40000; # Centre frequency is 40 kHz
T = 12E-3; # Chirp pulse length
K = B/T; # Chirp rate
# Define a simple a rect() function which returns for -0.5<=t<=0.5 or 0.
# The function will work if t is an array of values.
rect(t) = (abs.(t) .<= 0.5)*1.0
# rect(t/T) spans the interval [-T/2,T/2]
# We must therefore delay the chirp pulse so that it starts after t=0.
# Shift the chirp pulse by 0.6T units to the right, so that it starts at
0.1*T
td = 0.6*T; # Chirp delay
# Note: one can use the macro @. to avoid having to put . for arrays:
# @. v_tx = cos( 2*pi*(f0*(t-td) + 0.5*K*(t-td).^2) ).*rect((t-td)/T);
```

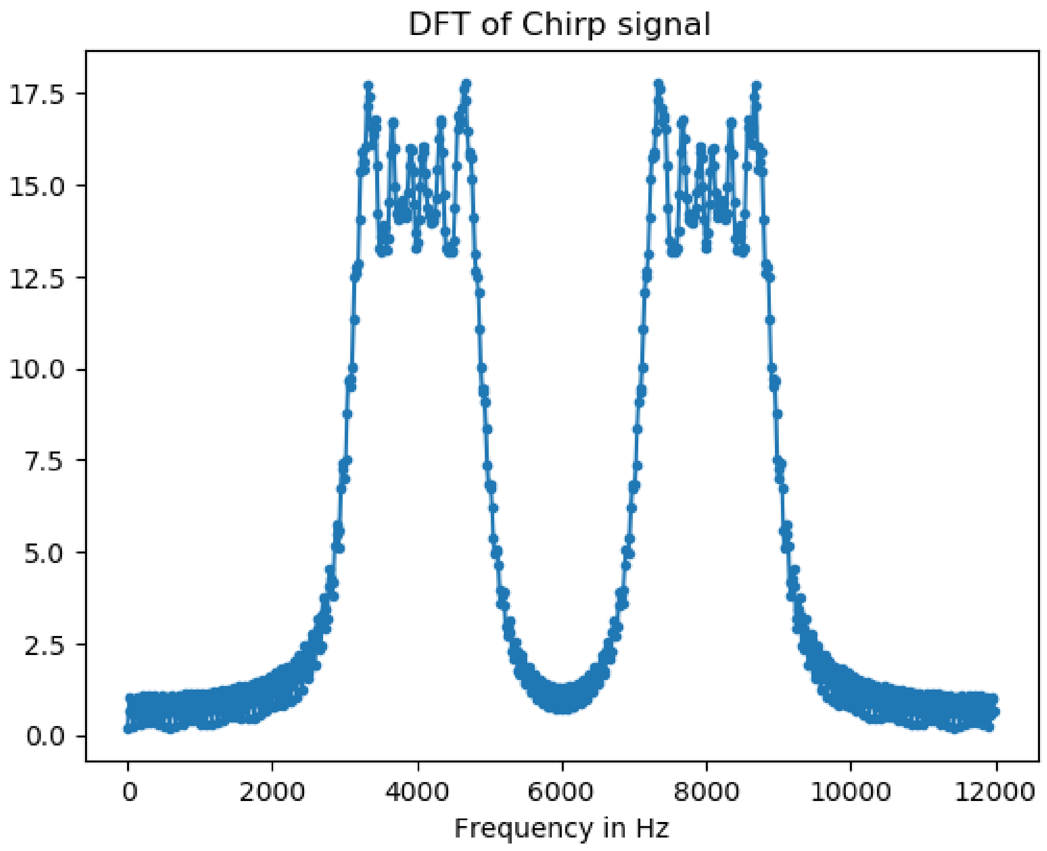
```
v_tx = cos.( 2*pi*(f0*(t .- td) + 0.5*K*(t .- td).^2) ) .* rect.((t .-td)/T);


using PyPlot; pygui(false) # import plot library
# If not installed, add the package via: using Pkg; Pkg.add("PyPlot");
figure() # Create a new figure
plot(v_tx) # Basic plot, axis labeled in samples
title("Chirp signal")
xlabel("Samples");
ylabel("Voltage");
```



[314]:
```
using FFTW # If not installed do: using Pkg; Pkg.add("FFTW");

V_TX = fft(v_tx);
```

[315]:
```
#LABEL frequency axis
N = length(t);
f = 1/(N*dt) # spacing in frequency domain
#create array of freq values stored in f_axis. First element maps to 0Hz
f_axis = (0:N-1)*f;
figure();
```

```
plot(f_axis, abs.(V_TX),".-");
title("DFT of Chirp signal");
xlabel("Frequency in Hz");
```



[316]:
```
R1 was chosen to be the same as part 1. R2 was chosen to be 5m away.
```

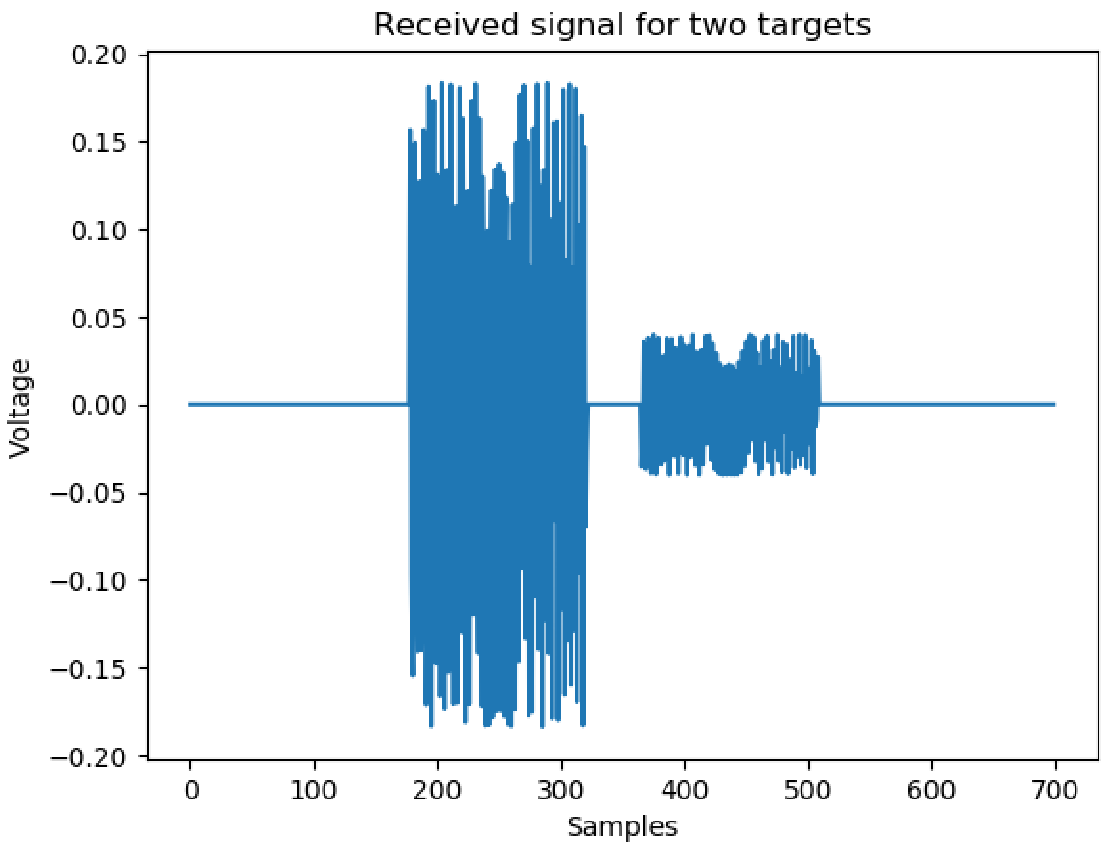
```
syntax: extra token "was" after end of expression
```

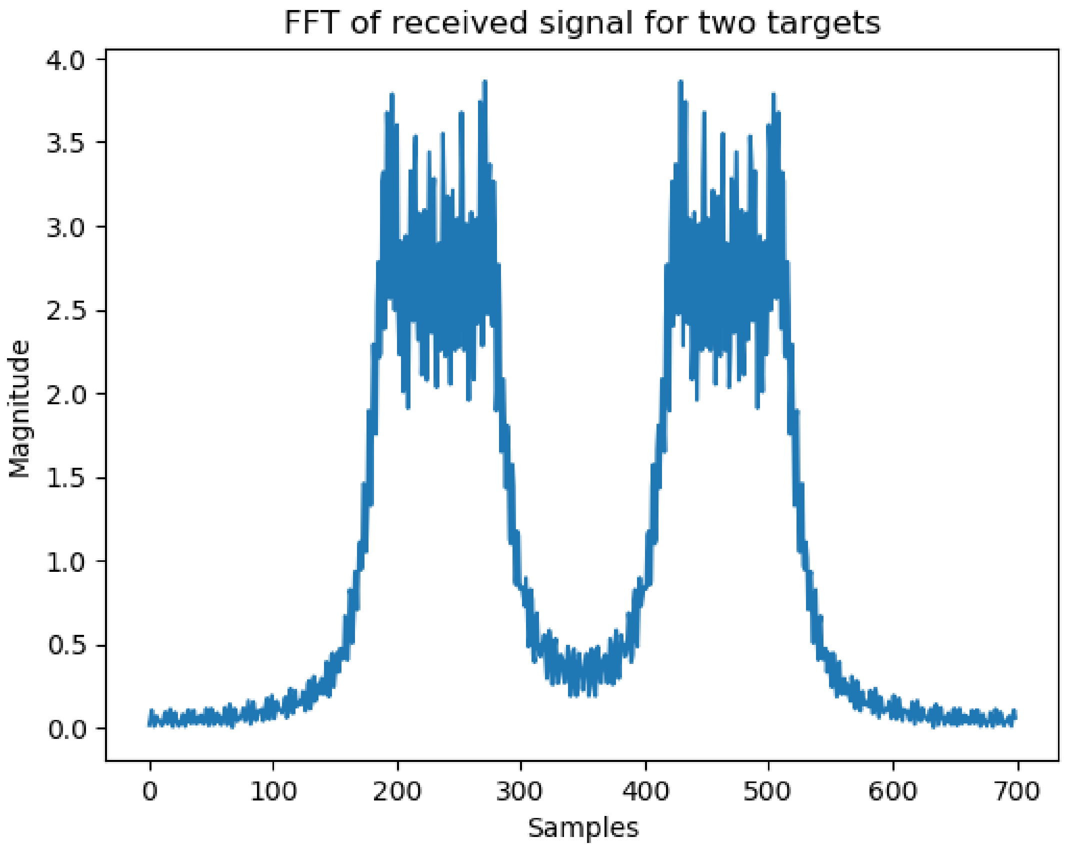
[317]:
```
#STEP 2: Point target simulation

R1 = 1.5 + (12-2)/12 # 2.33m - range to target.
td1 = 2*R1/c; # two way delay to target.
A1 = 1/R1^2; # echo voltage signal proportional to 1/R^2
v_rx = A1*cos.( 2*pi*(f0*(t.-td.-td1) + 0.5*K*(t.-td.-td1).^2) ) .* rect((t.-
td.-td1)/T);
```

```
V_RX = fft(v_rx);
```

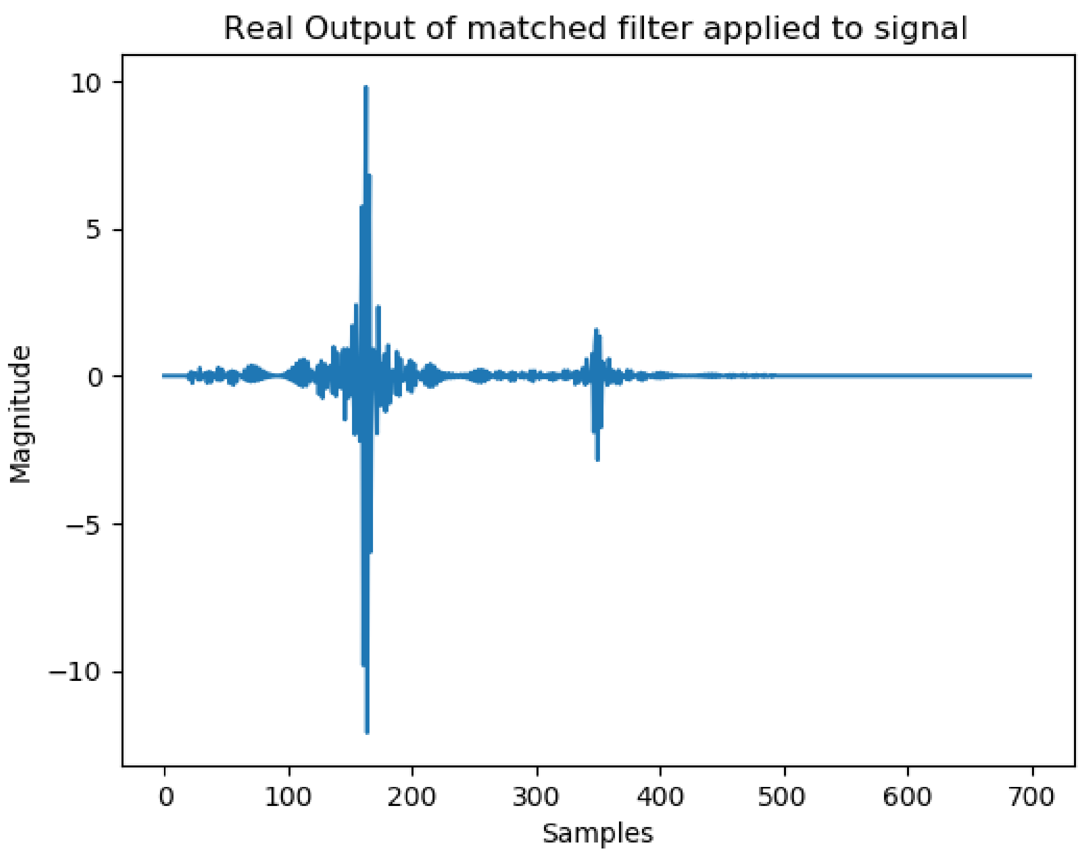
[318]:

```
#second target
R2 = 5  # 5m - range to target.
td2 = 2*R2/c; # two way delay to target.
A2 = 1/R2^2; # echo voltage signal proportional to 1/R^2
v_rx = A1*cos.(2*pi*(f0*(t.-td.-td1) + 0.5*K*(t.-td.-td1).^2) ) .* rect((t.-
td.-td1)/T) + A2*cos.(2*pi*(f0*(t.-td.-td2) + 0.5*K*(t.-td.-td2).^2) ) .*␣
 ↪rect((t.-
td.-td2)/T);
V_RX = fft(v_rx);

figure() # Create a new figure
plot(v_rx) # Put time on x-axis
title("Received signal for two targets")
xlabel("Samples");
ylabel("Voltage");

figure() # Create a new figure
plot(abs.(V_RX) )
title("FFT of received signal for two targets");
xlabel("Samples");
ylabel("Magnitude");
```

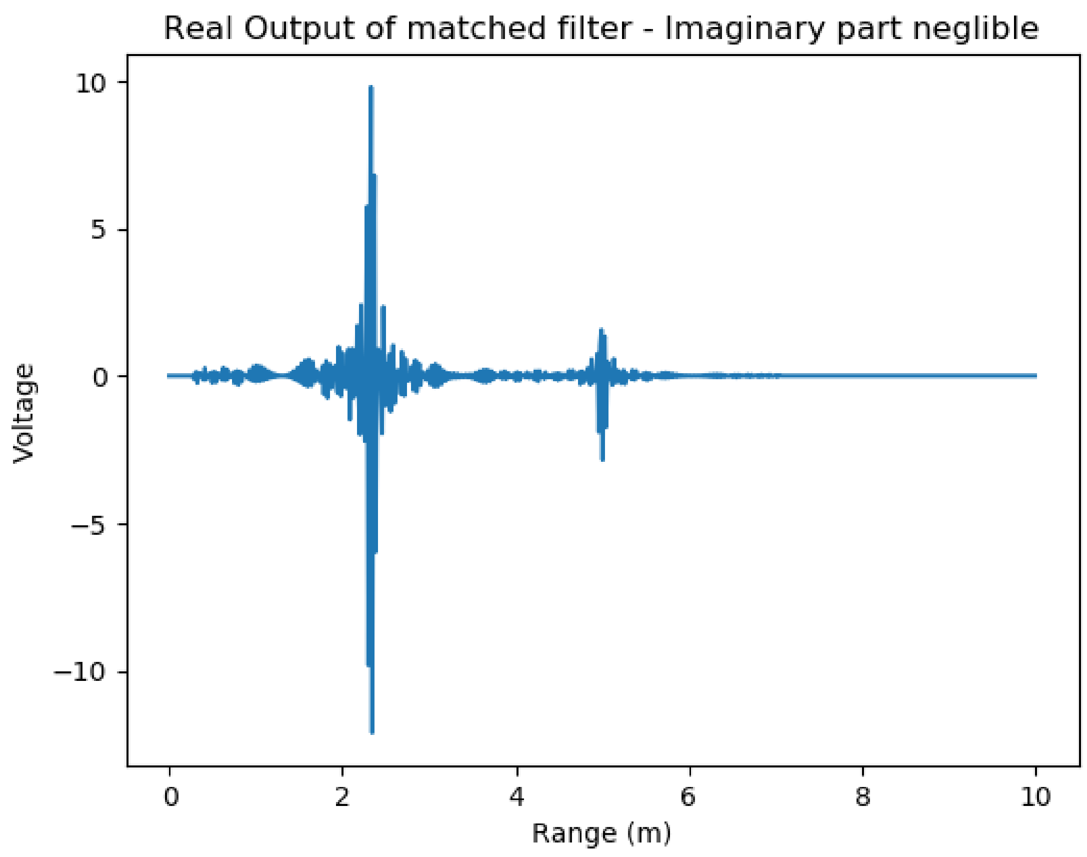




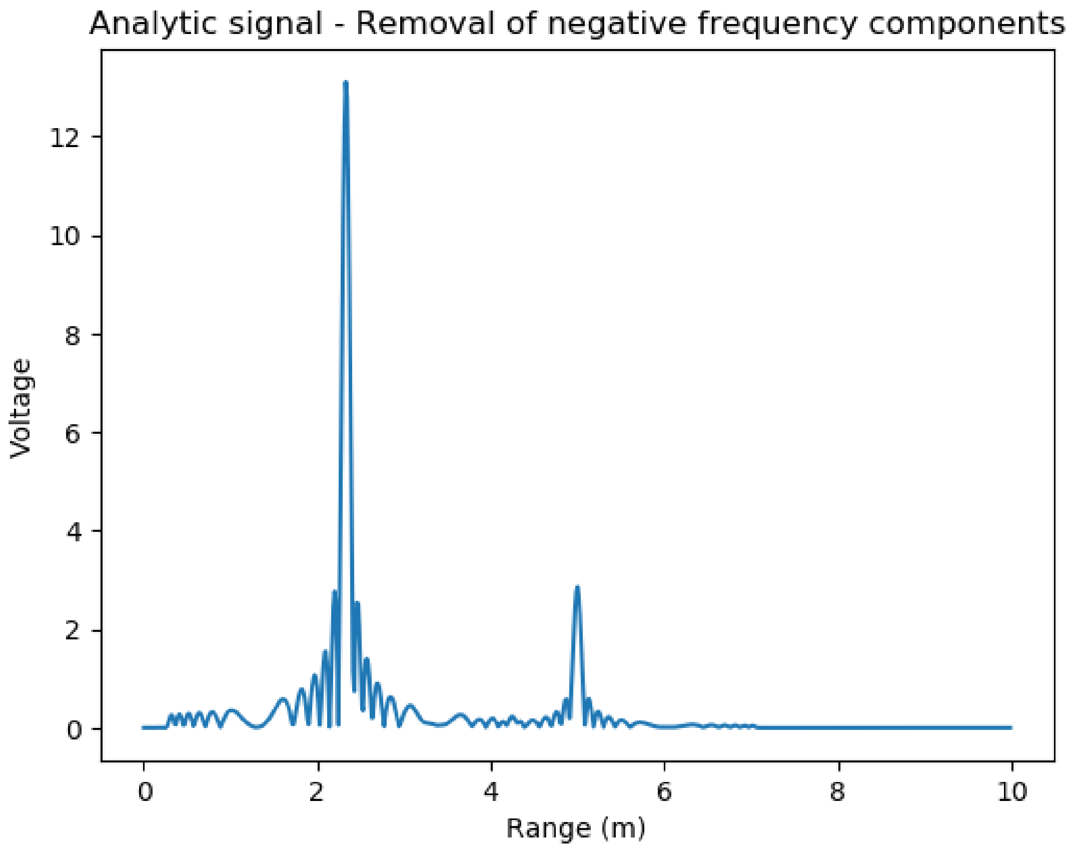
[319]:
```
#STEP 3: Matched filtering

H = conj( V_TX);
V_MF = H.*V_RX;
v_mf = ifft(V_MF);

figure() # Create a new figure
plot(real(v_mf));
title("Real Output of matched filter applied to signal")
xlabel("Samples");
ylabel("Magnitude");

#Negligble Imaginary part <1E-14
```



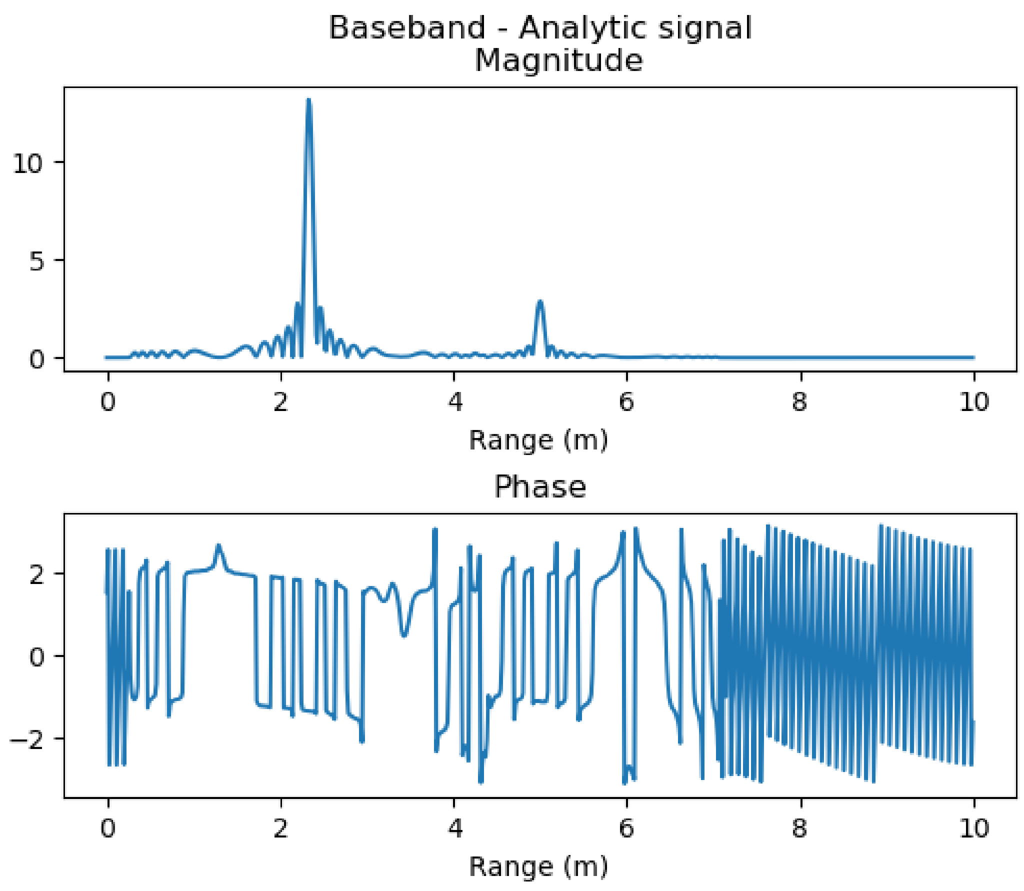
[320]:
```
v_mf = real(v_mf);
figure() # Create a new figure
plot(r,v_mf) # To see the detail  zoom in to have a good look.
title("Real Output of matched filter - Imaginary part neglible")
xlabel("Range (m)");
ylabel("Voltage");


# Take note of the shape of the envelope, as well as the internal detail.
# effect of pulse compression
```



[321]:
```
#STEP 4: Forming an analytic signal

V_ANAL = 2*V_MF; # make a copy and double the values
N = length(V_MF);
V_ANAL = 2*V_MF; # make a copy and double the values
N = length(V_MF);
if mod(N,2)==0 # case N even
 neg_freq_range = Int(N/2):N; # Define range of neg-freq components
else # case N odd
 neg_freq_range = Int((N+1)/2):N;
end
V_ANAL[neg_freq_range] .= 0; # Zero out neg components in 2nd half of array.
v_anal = ifft(V_ANAL);

figure() # Create a new figure
plot(r,abs.(v_anal)) # To see the magnitude  zoom in to have a good look.
title("Analytic signal - Removal of negative frequency components")
xlabel("Range (m)");
ylabel("Voltage");
```

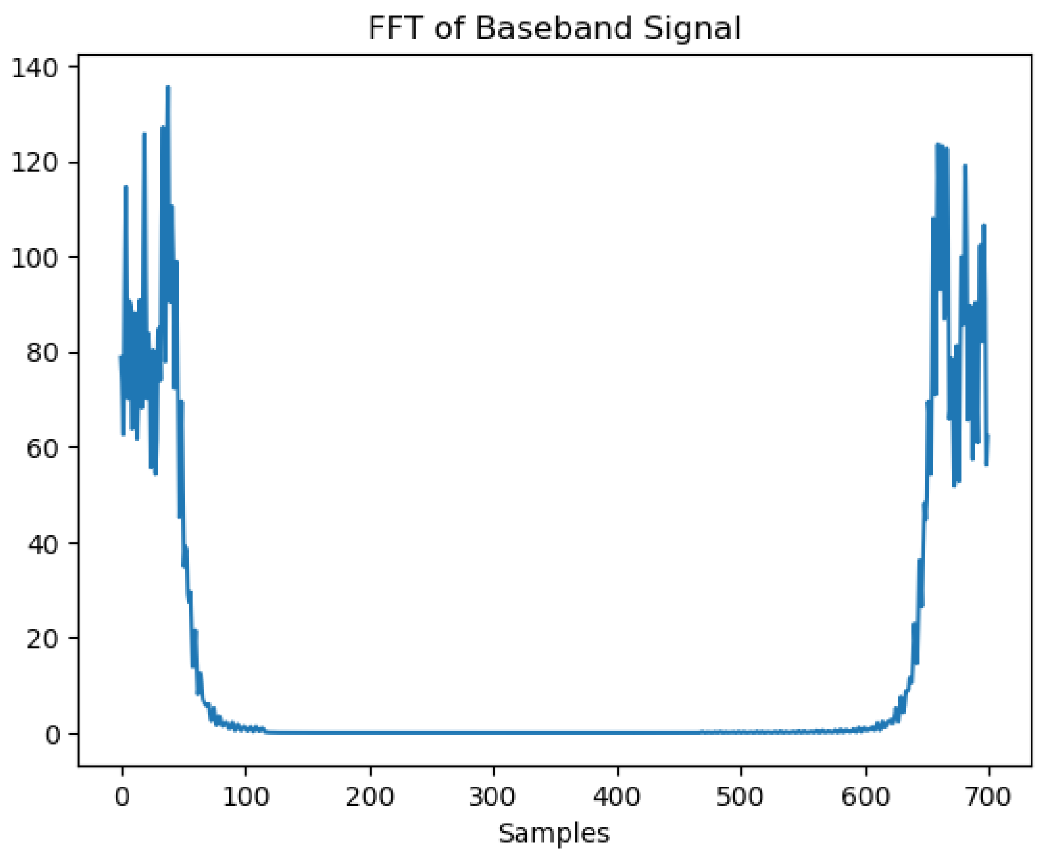


[322]:
```
#STEP 5: Translating the signal to baseband

j=im; # Assign j as sqrt(-1) (im in julia)
v_bb = v_anal.*exp.(-j*2*pi*f0*t);

fig = figure() # Create a new figure

subplot(2,1,1)
plot(r,abs.(v_bb))
title("Baseband - Analytic signal
    Magnitude")
xlabel("Range (m)");
fig.subplots_adjust(hspace=.5)
subplot(2,1,2)
plot(r,angle.(v_bb))
title("
Phase")
xlabel("Range (m)");
```



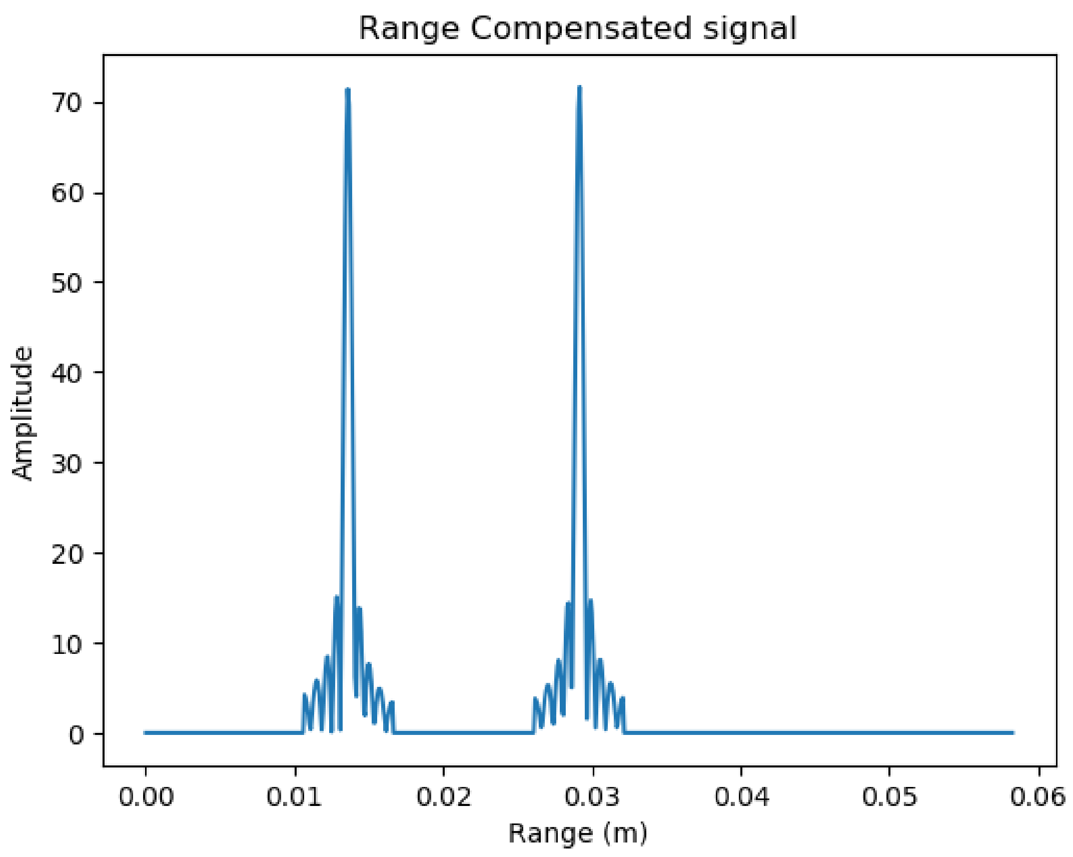
[323]:

```
V_BB = fft(v_bb);
figure() # Create a new figure
plot(abs.(V_BB))
title("FFT of Baseband Signal");
xlabel("Samples");
```



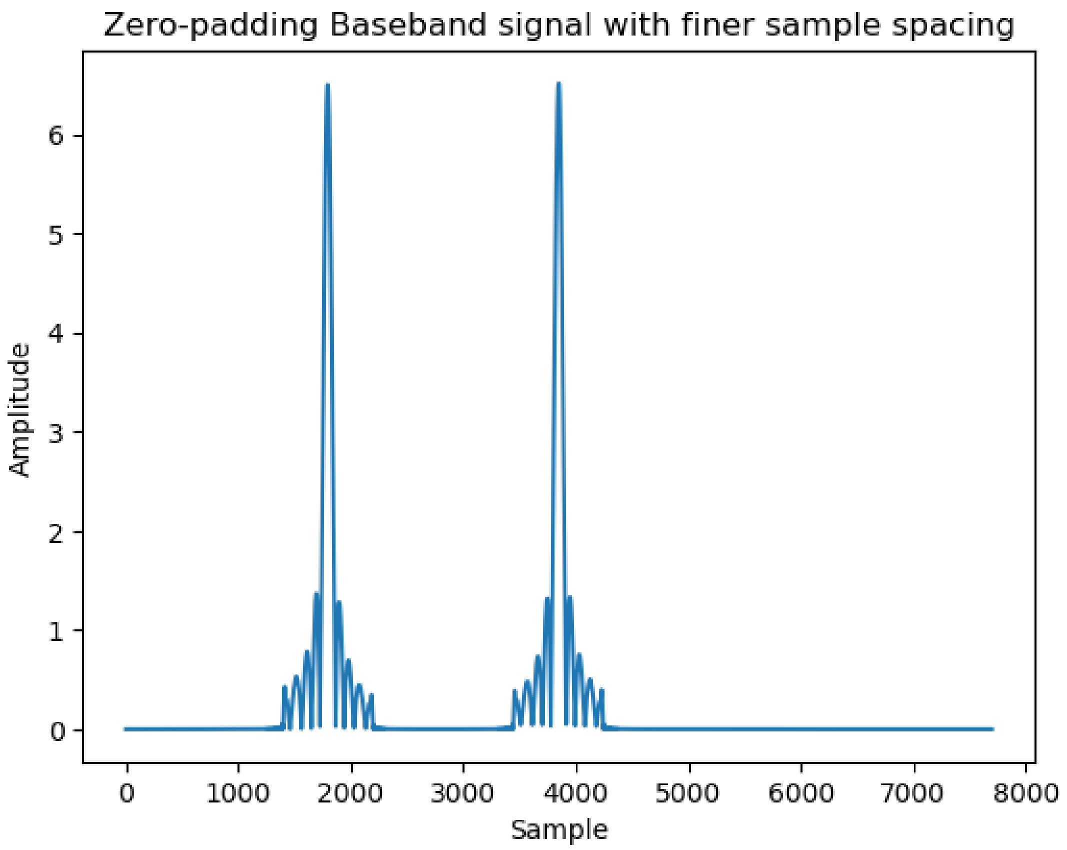
[324]:
```
#STEP 6: Adding noise to the simulation

sigma = 0.2 * A1;
noise_signal = sigma * randn(N);
```

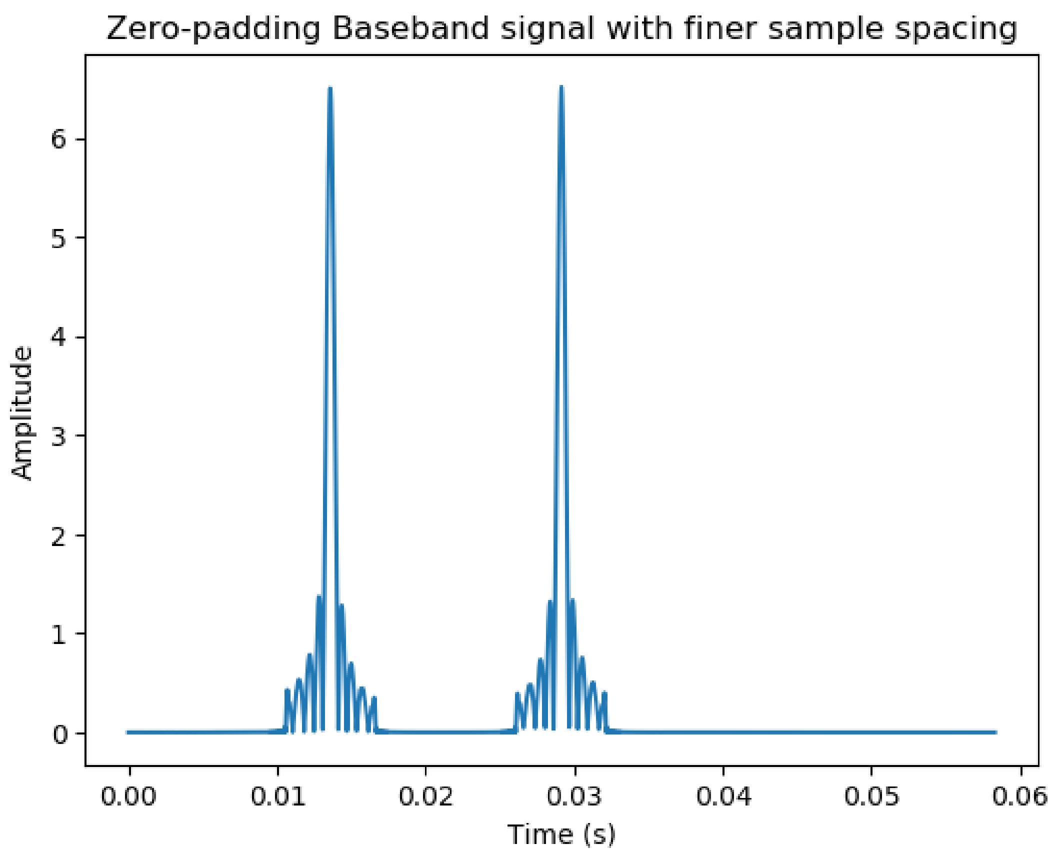
[325]:
```
#Part 7

# Define a simple a rect() function which returns for -0.25<=t<=0.25 or 0.
# The function will work if t is an array of values.
rect2(t) = (abs.(t) .<= 0.25)*1.0
R1_func = R1^2
R2_func = R2^2
v_rc = v_bb.* rect2.((t.-td1)/T)*R1_func .+ v_bb.* rect2.((t .-td2)/T)*R2_func;

figure() # Create a new figure
plot(t,real(abs.(v_rc)));
title("Range Compensated signal")
xlabel("Range (m)");
ylabel("Amplitude");
```



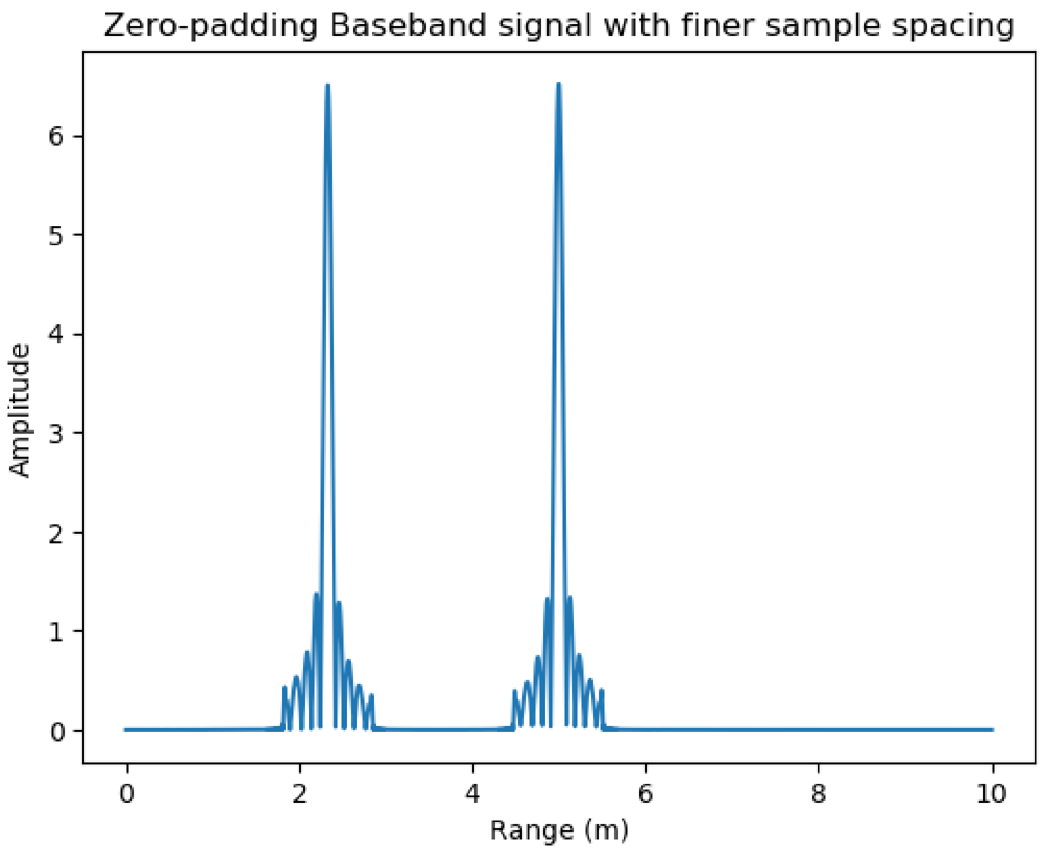
[326]:
```
Empty  = zeros(10*N)
p = floor(Int32, N/2)
V_RC_Padded = fft(v_rc)
temp = vcat(V_RC_Padded[1:p-1], Empty)
X = vcat(temp, V_RC_Padded[p:end])
x = ifft(X);
time_new = collect(0:dt/11.004:t_max);
range_new = c*time_new/2;

figure()
plot(real(abs.(x)));
title("Zero-padding Baseband signal with finer sample spacing")
xlabel("Sample");
ylabel("Amplitude");

using PyPlot;
figure()
plot(time_new,real(abs.(x)));
title("Zero-padding Baseband signal with finer sample spacing")
xlabel("Time (s)");
```

```
ylabel("Amplitude");


figure()
plot(range_new,real(abs.(x)));
title("Zero-padding Baseband signal with finer sample spacing")
xlabel("Range (m)");
ylabel("Amplitude");
```





Zero-padding Baseband signal with finer sample spacing
Amplitude
Time (s)
0
1
2
3
4
5
6
0.00
0.01
0.02
0.03
0.04
0.05
0.06

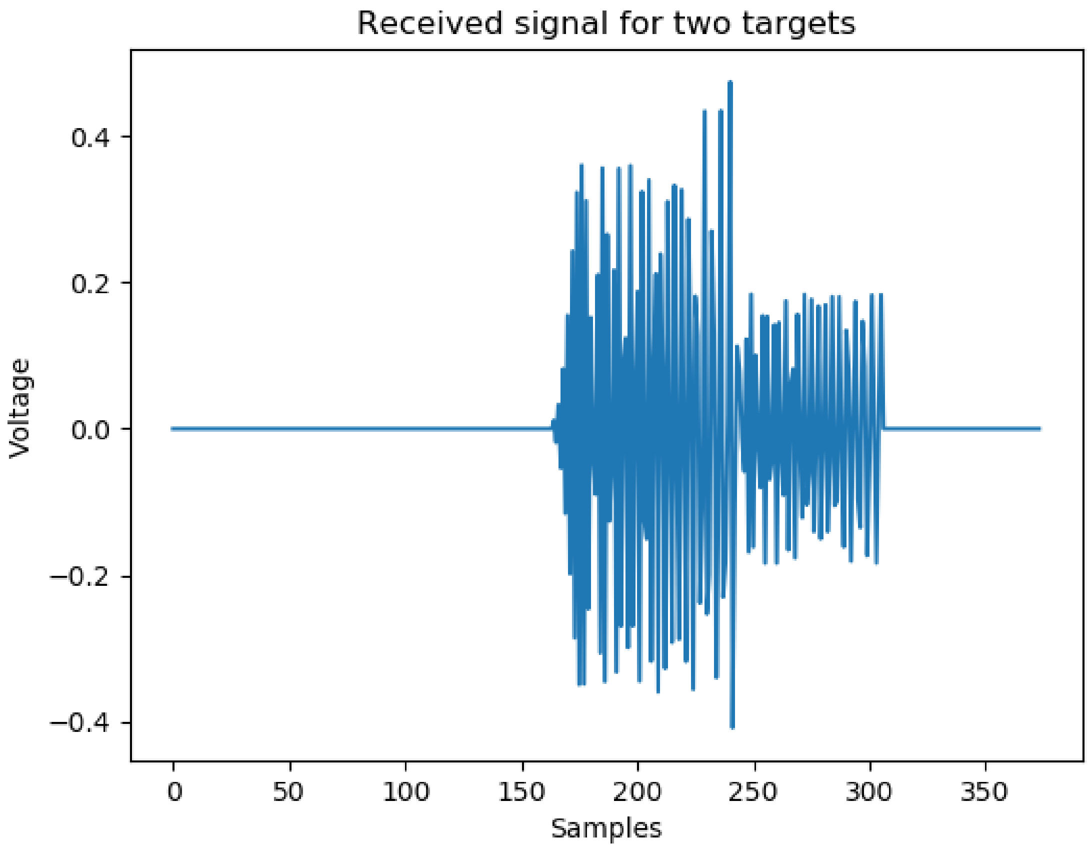

Zero-padding Baseband signal with finer sample spacing
Amplitude
Range (m)
0
1
2
3
4
5
6
0
2
4
6
8
10

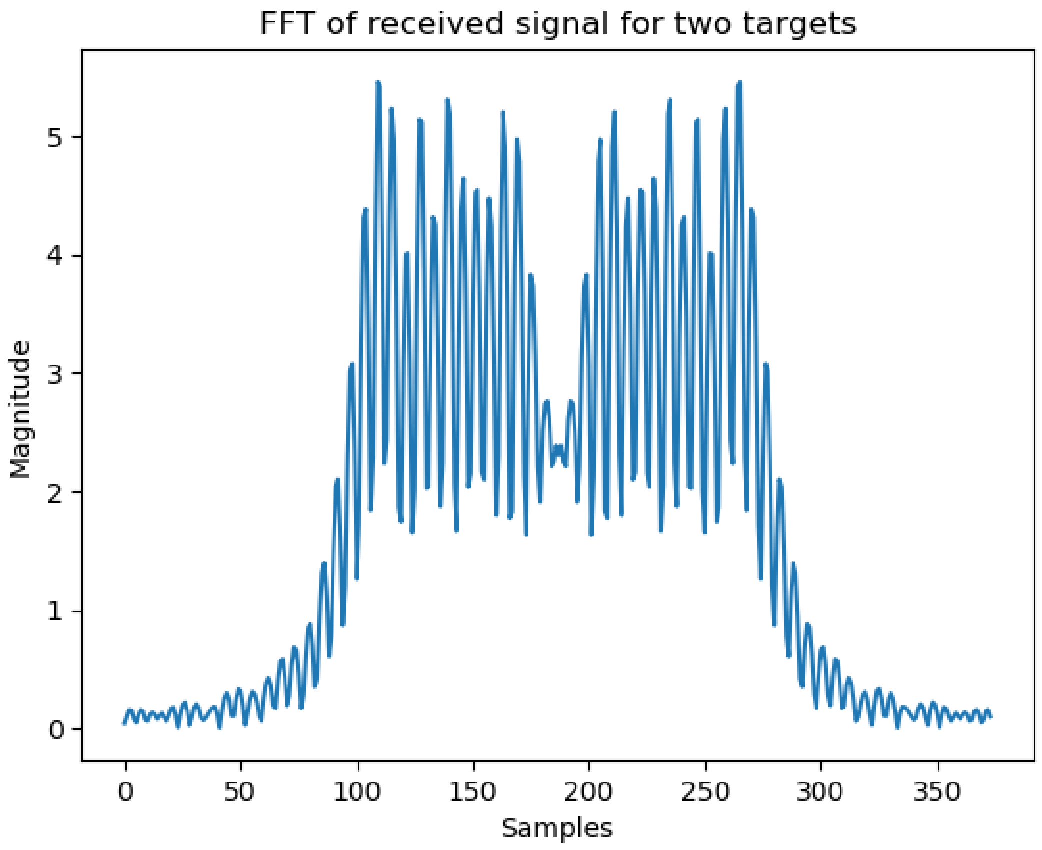
[172]:
```
using FFTW # If not installed do: using Pkg; Pkg.add("FFTW");

V_TX = fft(v_tx);
```

[173]:
```
#LABEL frequency axis
N = length(t);
f = 1/(N*dt) # spacing in frequency domain
#create array of freq values stored in f_axis. First element maps to 0Hz
f_axis = (0:N-1)*f;
```

[174]:
```
#STEP 2: Point target simulation

R1 = 1.5 + (12-2)/12 # 2.33m - range to target.
td1 = 2*R1/c; # two way delay to target.
A1 = 1/R1^2; # echo voltage signal proportional to 1/R^2
v_rx = A1*cos.( 2*pi*(f0*(t.-td.-td1) + 0.5*K*(t.-td.-td1).^2) ) .* rect((t.-
td.-td1)/T);

V_RX = fft(v_rx);
```

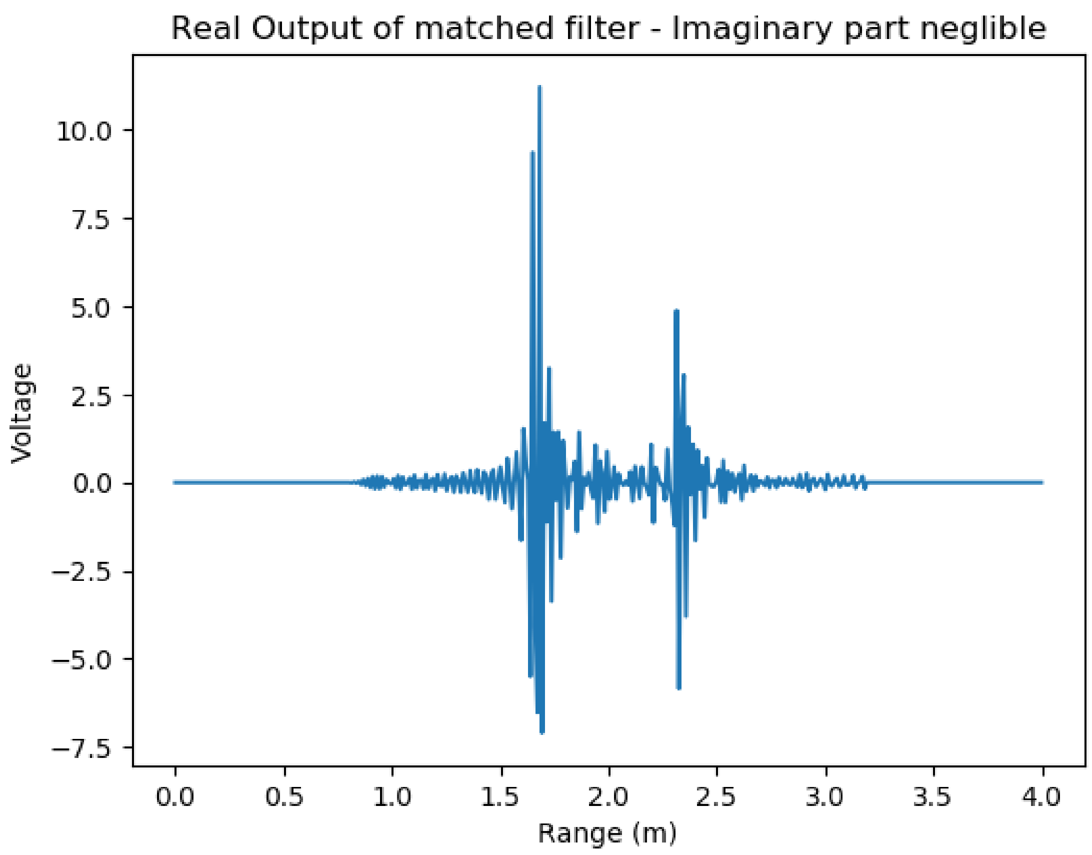
[175]:
```
#second target
R2 =  1.5 + (12-10)/12# 1.67m - range to target.
td2 = 2*R2/c; # two way delay to target.
A2 = 1/R2^2; # echo voltage signal proportional to 1/R^2
v_rx = A1*cos.(2*pi*(f0*(t.-td.-td1) + 0.5*K*(t.-td.-td1).^2) ) .* rect((t.-
td.-td1)/T) + A2*cos.(2*pi*(f0*(t.-td.-td2) + 0.5*K*(t.-td.-td2).^2) ) .*␣
 ↪rect((t.-
td.-td2)/T);
V_RX = fft(v_rx);

figure() # Create a new figure
plot(v_rx) # Put time on x-axis
title("Received signal for two targets")
xlabel("Samples");
ylabel("Voltage");

figure() # Create a new figure
plot(abs.(V_RX) )
title("FFT of received signal for two targets");
xlabel("Samples");
ylabel("Magnitude");
```


Received signal for two targets
Voltage
Samples
0.4
0.2
0.0
−0.2
−0.4
0
50
100
150
200
250
300
350

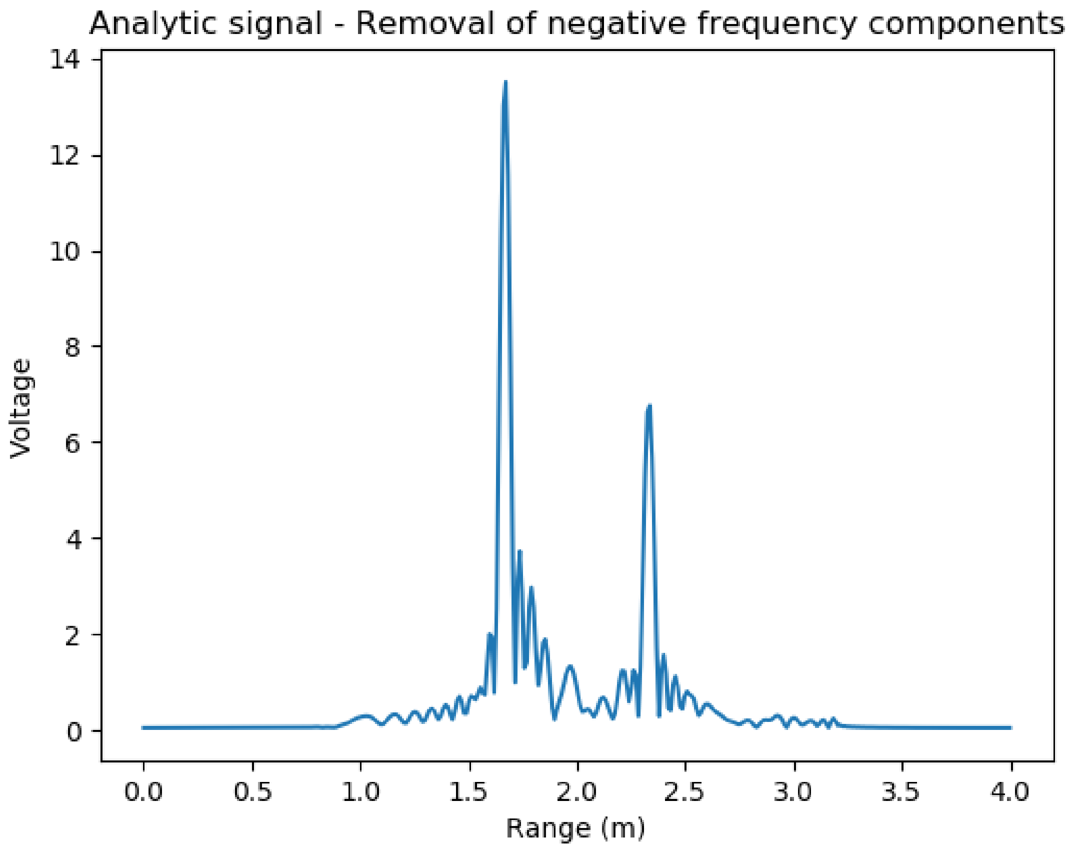


[176]:
```
#STEP 3: Matched filtering

H = conj( V_TX);
V_MF = H.*V_RX;
v_mf = ifft(V_MF);

#Negligble Imaginary part <1E-14
```

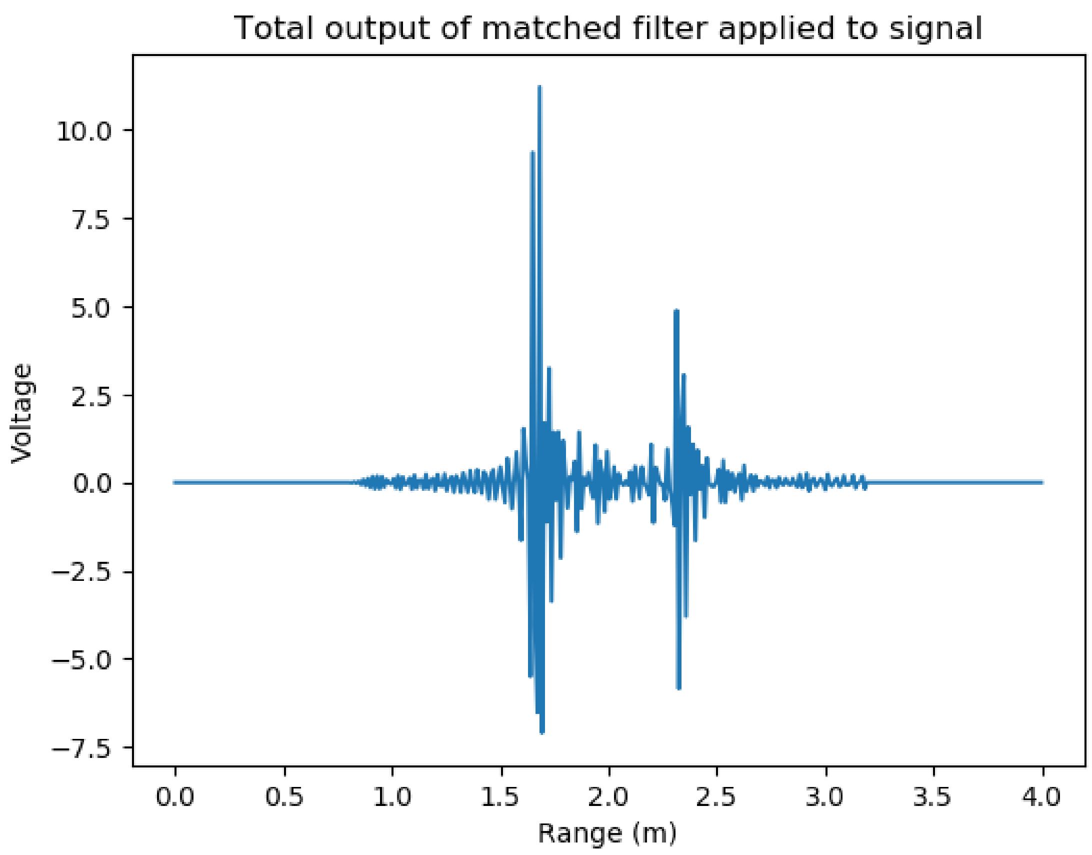
[177]:
```
v_mf = real(v_mf);
figure() # Create a new figure
plot(r,v_mf) # To see the detail  zoom in to have a good look.
title("Real Output of matched filter - Imaginary part neglible")
xlabel("Range (m)");
ylabel("Voltage");


# Take note of the shape of the envelope, as well as the internal detail.
# effect of pulse compression
```



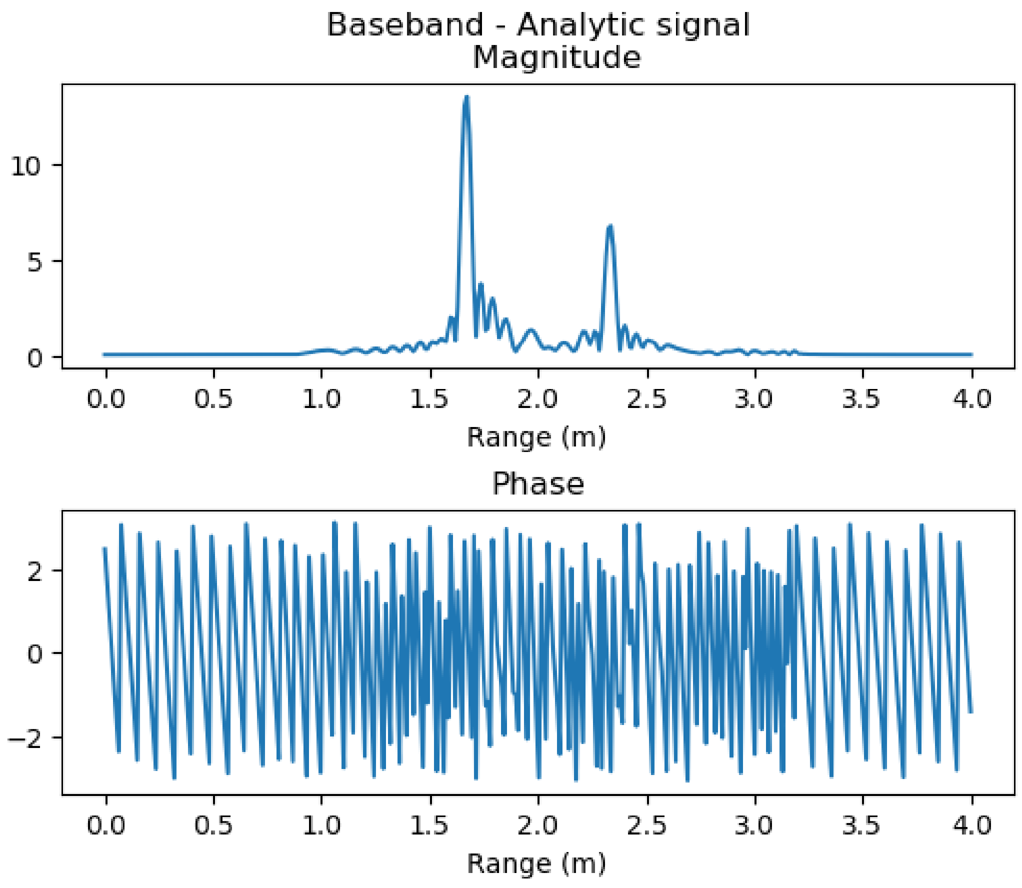
[178]:
```
#STEP 4: Forming an analytic signal

V_ANAL = 2*V_MF; # make a copy and double the values
N = length(V_MF);
V_ANAL = 2*V_MF; # make a copy and double the values
N = length(V_MF);
if mod(N,2)==0 # case N even
 neg_freq_range = Int(N/2):N; # Define range of neg-freq components
else # case N odd
 neg_freq_range = Int((N+1)/2):N;
end
V_ANAL[neg_freq_range] .= 0; # Zero out neg components in 2nd half of array.
v_anal = ifft(V_ANAL);

figure() # Create a new figure
plot(r,abs.(v_anal)) # To see the magnitude  zoom in to have a good look.
title("Analytic signal - Removal of negative frequency components")
xlabel("Range (m)");
ylabel("Voltage");
```



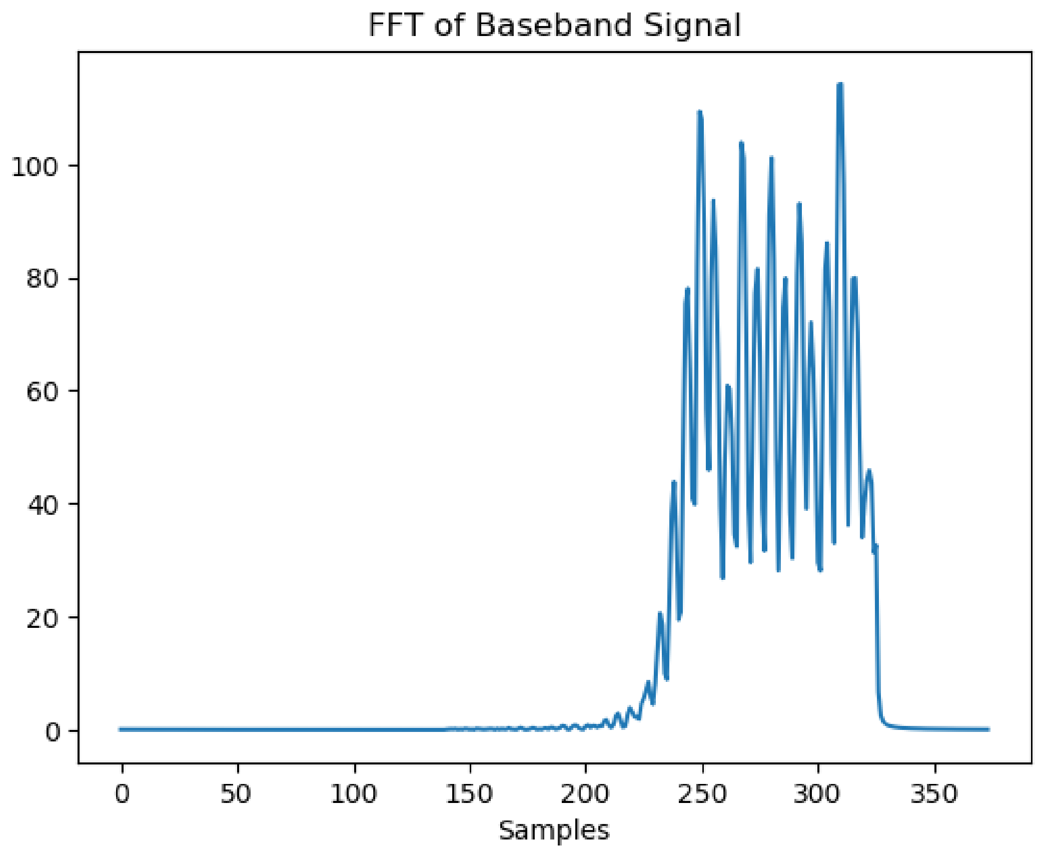
[179]:
```
figure() # Create a new figure
plot(r,v_mf);
title("Total output of matched filter applied to signal")
xlabel("Range (m)");
ylabel("Voltage");
```



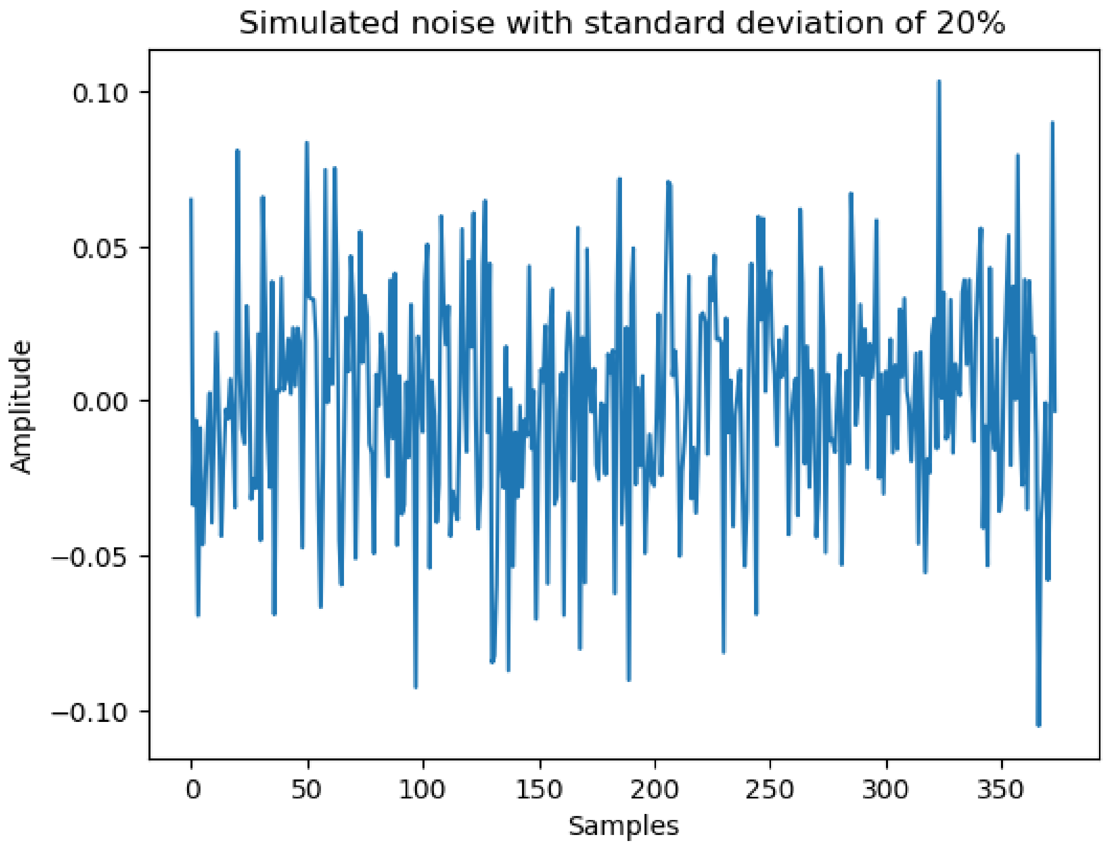
[180]:
```
#STEP 5: Translating the signal to baseband

j=im; # Assign j as sqrt(-1) (im in julia)
v_bb = v_anal.*exp.(-j*2*pi*f0*t);

fig = figure() # Create a new figure

subplot(2,1,1)
plot(r,abs.(v_bb))
title("Baseband - Analytic signal
    Magnitude")
xlabel("Range (m)");
fig.subplots_adjust(hspace=.5)
subplot(2,1,2)
plot(r,angle.(v_bb))
title("
Phase")
xlabel("Range (m)");
```



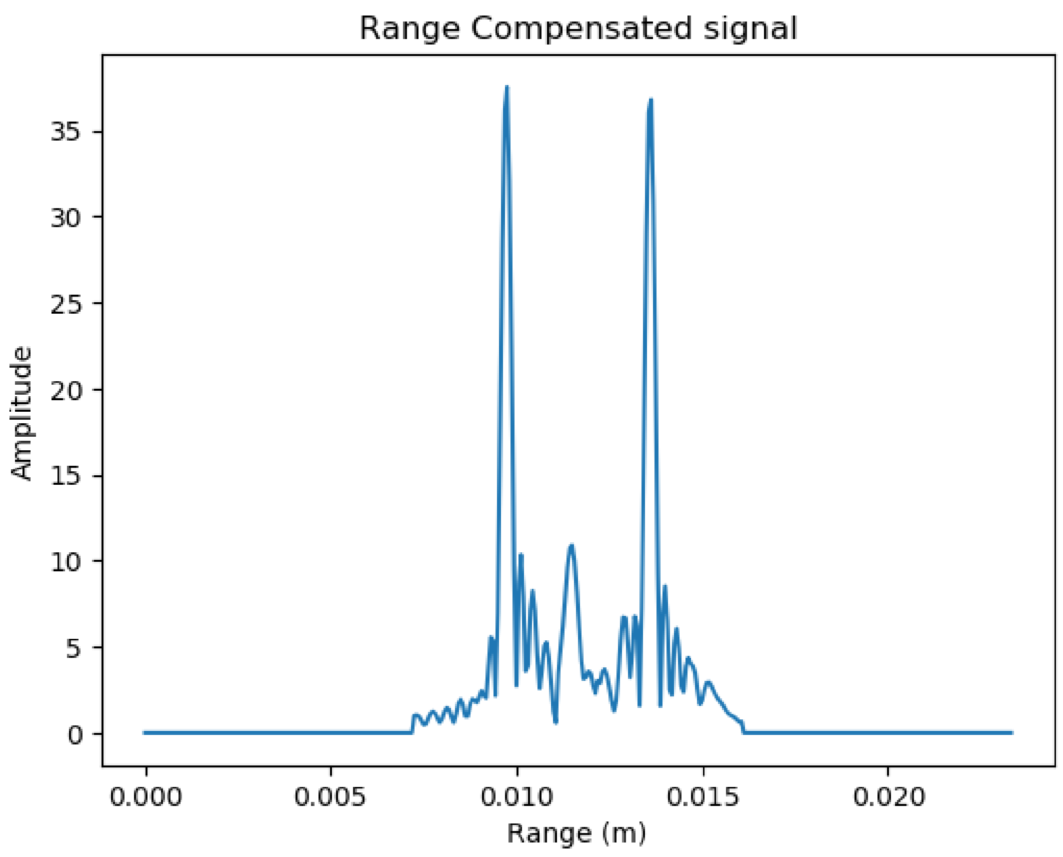
[181]:
```
V_BB = fft(v_bb);
figure() # Create a new figure
plot(abs.(V_BB))
title("FFT of Baseband Signal");
xlabel("Samples");
```



[182]:
```
#STEP 6: Adding noise to the simulation

sigma = 0.2 * A1;
noise_signal = sigma * randn(N);
figure() # Create a new figure
plot(noise_signal)
title("Simulated noise with standard deviation of 20%");
xlabel("Samples");
ylabel("Amplitude");
```

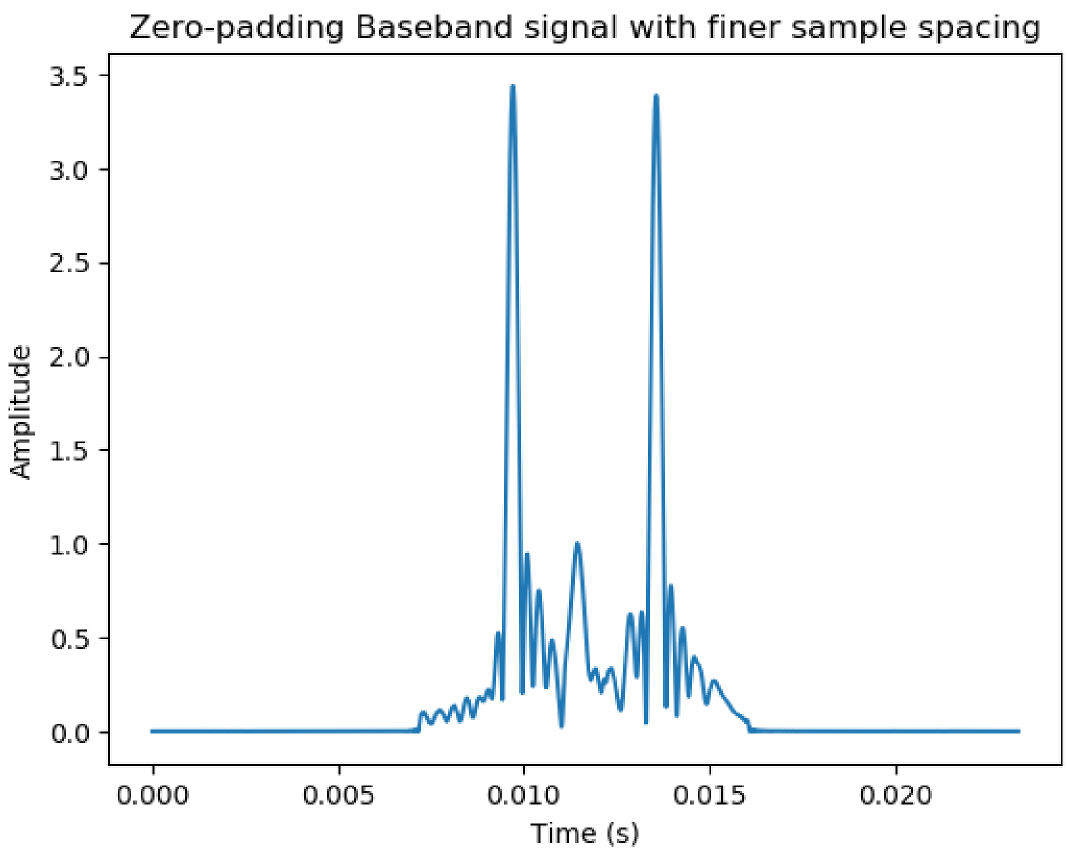


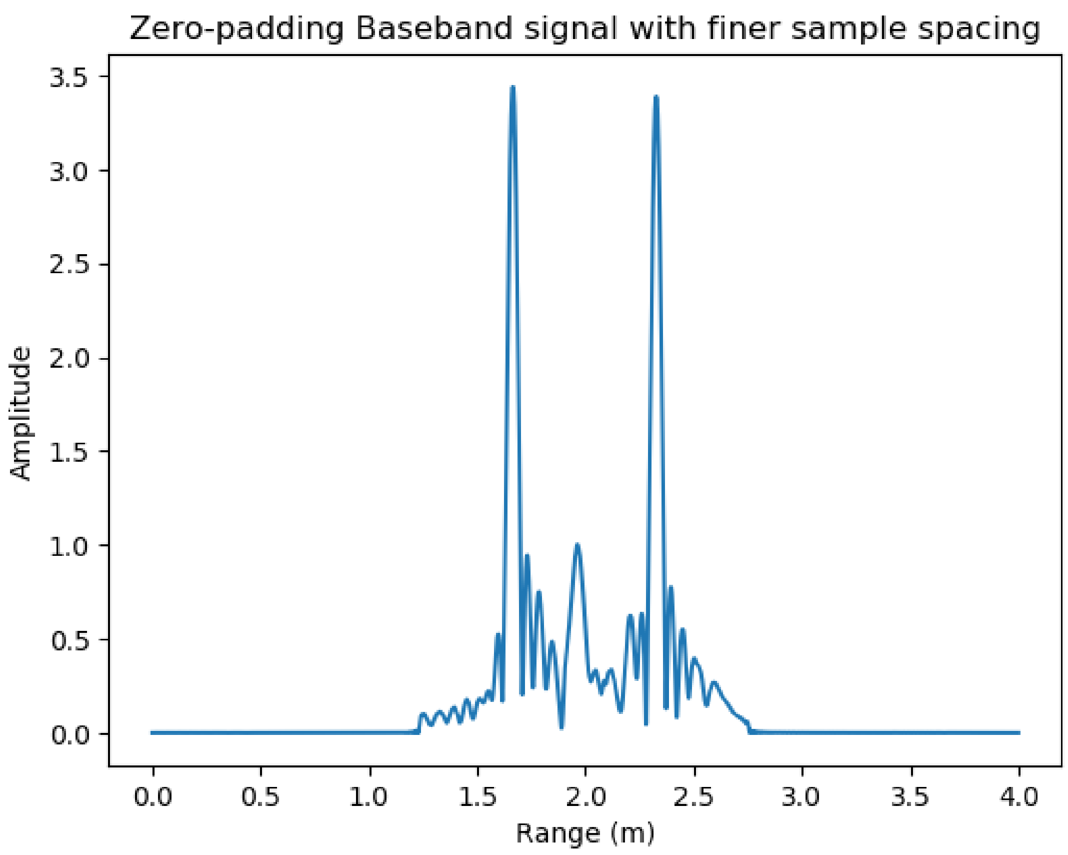
[183]:
```
#Part 7

# Define a simple a rect() function which returns for -0.25<=t<=0.25 or 0.
# The function will work if t is an array of values.
rect2(t) = (abs.(t) .<= 0.25)*1.0
R1_func = R1^2
R2_func = R2^2
v_rc = v_bb.* rect.((t.-td1)/T)*R1_func .+ v_bb.* rect.((t .-td2)/T)*R2_func;

figure() # Create a new figure
plot(t,real(abs.(v_rc)));
title("Range Compensated signal")
xlabel("Range (m)");
ylabel("Amplitude");
```



[184]:
```
#Part 8
Empty  = zeros(10*N)
p = floor(Int32, N/2)
V_RC_Padded = fft(v_rc)
temp = vcat(V_RC_Padded[1:p-1], Empty)
X = vcat(temp, V_RC_Padded[p:end])
x = ifft(X);
time_new = collect(0:dt/11.024:t_max);
range_new = c*time_new/2;

figure()
plot(time_new,real(abs.(x)));
title("Zero-padding Baseband signal with finer sample spacing")
xlabel("Time (s)");
ylabel("Amplitude");


figure()
plot(range_new,real(abs.(x)));
title("Zero-padding Baseband signal with finer sample spacing")
```

```
xlabel("Range (m)");
ylabel("Amplitude");
```





From the above graphs it can be seen that all the information is preserved.