

EE123 Lab 1-Time Domain Sonar

February 12, 2018

1 EE 123 Lab1 - Time Domain Sonar Lab

1.0.1 Written by Miki Lustig and Frank Ong 2016

Edited, debugged and ported to Raspberry Pi Nick Antipa, Li-Hao Yeh and Miki Lustig 2018
[Celine Veys](#), [Mia Mirkovic](#)

In this lab, we will interact with physical time-domain signals. The first part will involve generating and recording sounds on the Raspberry-Pi. We will use the chirp signal to characterize the response of the speaker-microphone system and look at detecting signals using cross-correlation. In the second part, we will build on part one and use the speaker-microphone system to develop a simple sonar.

When using the raspberry pi – The RaspberryPi does not have a microphone input. In order to use both audio input and output, we will use a USB soundcard. This USB soundcard is in fact an Analog-to-Digital and Digital-to-Analog device.

- please connect the USB audio to the pi
- connect the microphone to the mic input of the USB audio
- connect the speaker to the speaker output of the USB audio
- make sure the speaker is powered – e.g. connected to a USB either on your computer, or the pi

```
In [8]: # Import functions and libraries
import numpy as np, matplotlib.pyplot as plt
import threading, time, queue, pyaudio
from matplotlib.pyplot import *
import matplotlib.cm as cm
from scipy import signal
from numpy import *
from threading import Lock

%matplotlib inline
```

1.1 Part 1: Chirping!

For this assignment you will use the the raspberry-pi equipped with a USB sound card, a USB powered speaker and a microphone. When playing a sound and recording, the signal goes

through several systems. In particular it goes through the response of the USB soundcard output, the speaker, the room we are in and the response of the microphone and receive part of the USB soundcard.

A chirp is a signal in which the frequency increases linearly with time. In this assignment we will generate a chirp signal and use it to measure the amplitude of the frequency response of our speaker-room-microphone system. This lab will work best in a quiet environment – We recommend that you execute the lab at home or in a quiet place before submitting it.

A simultaneous frequency is defined as the derivative of the phase of a signal, $f = \frac{1}{2\pi} \frac{d\phi(t)}{dt}$. For example, the simultaneous frequency of $\cos(\phi(t)) = \cos(2\pi f_0 t)$ is

$$f = \frac{d\phi(t)}{2\pi dt} = f_0$$

For a linear chirp, the frequency changes linearly over time. The simultaneous frequency is therefore defined as

$$f(t) = f_0 + kt.$$

So,

$$x(t) = \sin(2\pi \int_0^t f(t') dt') = \sin(2\pi \int_0^t (f_0 + kt') dt') = \sin(2\pi(f_0 + \frac{k}{2}t)t)$$

1.1.1 Part I Task I: Generating the Chirp

Generate a 10 seconds long chirp signal, sampled at 48,000[Hz] with a frequency range of 20[Hz] to 20,000[Hz]. Set the magnitude of the chirp to 0.5. This will help prevent non-linearities when we play the sound later.

- Given T =total time length, f_0 =start frequency, f_1 = end frequency, derive a formula $f(t)$ for the frequency sweep.
- Find the formula for the phase by integrating $\phi(t) = \int_0^T f(t)dt$ to get the phase function.

Now, * Set the sample-rate frequency $f_s = 48000$ Hz * Generate a time index from $t=0$ to $t=10$ with sampling rate of 48000 Hz * Generate a vector of frequency vs time: $\phi_{of_t} (\phi(t))$ * Generate the chirp function s_chirp with amplitude of 0.5 by plugging the phase into a sinusoid.

```
In [137]: fs = 48000
          f0 = 20
          f1 = 20000
          T = 10

          # generate time index and phase
          t = np.r_[0:T:1/fs]
          f_sweep = t*(f1-f0)/T + f0
          phi = (t**2)*((f1-f0)/(2*T)) + t*f0

          def generate_chirp (num_samps, f0, f1, fs=48000):
              T = num_samps/fs
              t = np.r_[0.0:T:1/fs]
```

```

f_sweep = t*(f1-f0)/T + f0
phi = (t**2)*((f1-f0)/(2*T)) + t*f0
s_chirp = 0.5*np.sin(2*np.pi*(t+phi))
a_chirp = exp(2*1j*np.pi*phi)
#h_chirp = signal.hilbert(s_chirp)
return [s_chirp, a_chirp]

def plot_fr(sig, a_or_h, title='Frequency Response of Chirp', f0=0.0, f1 = np.pi, fs =
    F = f1-f0
    fs = 48000
    fs1 = 512
    t = len(sig)
    f = np.r_[f0:f1:F/t]
    f1 = np.r_[f0:f1:F/fs1]

    width, height = plt.figaspect(0.2)
    fig = plt.figure(figsize=(width, height))
    plt.title(title)
    plt.xlabel(xunits)
    if (a_or_h == 'h'):
        w, h = signal.freqz(sig)
        plt.plot(f1, abs(h))
    elif (a_or_h == 'a'):
        plt.plot(f, abs(sig))
    plt.show()
    return 1

def plot_hil(sig, env, title, f0=0, f1=np.pi, fs = 512, xunits='f [Hz]'):
    F = f1-f0
    fs = 48000
    fs1 = 512
    t = len(sig)
    f = np.r_[f0:f1:F/t]
    f1 = np.r_[f0:f1:F/fs1]
    width, height = plt.figaspect(0.2)
    fig = plt.figure(figsize=(width, height))
    plt.title(title)
    plt.xlabel(xunits)
    h = signal.hilbert(sig)
    if env:
        plt.plot(f, abs(h))
    else:
        plt.plot(f1, h)
    plt.show()
    return 1

def acorr(sig, title='Autocorrelation of Chirp', xunits='time[ms]'):
    acorr = signal.fftconvolve(sig, sig[::-1], mode="full")

```

```

width, height = plt.figaspect(0.2)
fig = plt.figure(figsize=(width, height))
plt.plot(abs(acorr))
plt.title(title)
plt.xlabel(xunits)
plt.show
return acorr

def xcorr(sig, rsig, t0=0, title='Cross-Correlation', xunits='time[ms]'):
    xcorr = signal.fftconvolve(sig, rsig[::-1], mode="full")
    width, height = plt.figaspect(0.3)
    fig = plt.figure(figsize=(width, height))
    t1=(len(sig)+len(rsig)- 1)
    plt.plot(abs(xcorr))
    plt.title(title)
    plt.xlim(t0, t1)
    plt.xlabel(xunits)
    plt.show()
    return xcorr
# generate chirp signal

```

```
s_chirp = generate_chirp(48000*10, 20, 20000)
```

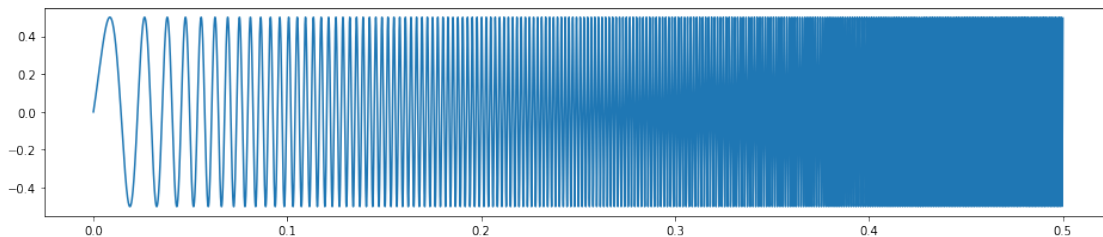
- Plot the first $\frac{1}{2}$ second of the chirp (s_chirp), you will notice that the carrier frequency increases and that the chirp has a constant envelope. To get a nice figure, make sure the aspect ratio of the figure is height/width = 0.2 . Label the axis and figure appropriately.

In [60]: *# Set the aspect ratio such that the image is wide*

```
width, height = plt.figaspect(0.2)
plt.figure(figsize=(width, height))
```

#Your code below:

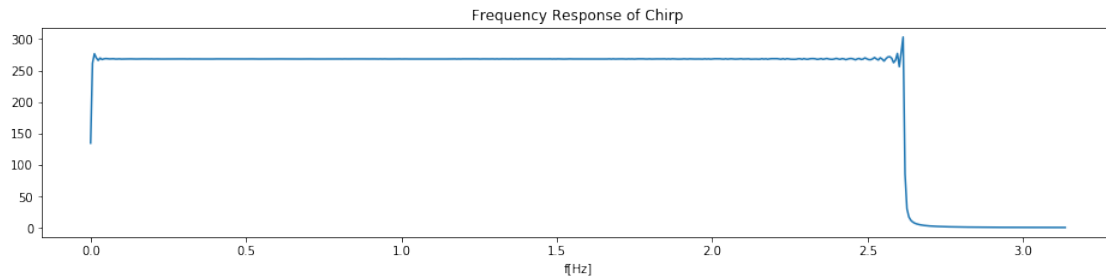
```
plt.plot([n for n in t if n < 0.5], s_chirp[0][0:len([n for n in t if n < 0.5])])
plt.show()
```



- Plot the magnitude frequency response of the sequence from 0 to π using the function `signal.freqz`. Note, that the digital frequency range represents a physical frequency range of 0[hz] to 24000[Hz]. To get a nice figure, make sure the aspect ratio of the figure is height/width = 0.2. Label the axis and figure appropriately.

The `signal.freqz` function on the Pi is very slow– be patient.

```
In [61]: plot_fr(s_chirp[0], 'h')
```



```
Out [61]: 1
```

Explain why the chirp is an appropriate signal to measure the magnitude frequency response of a system.

Your answer here: It allows us to characterize the system's response to the frequencies it sweeps by linearly relating frequency to time.

1.1.2 Part I Task II: Playing and Recording the Chirp

Now, we will play the sound of the chirp on our computer speaker and simultaneously record using the microphone.

Instructions for the Raspberry pi (EE123 2018):

- To setup the output and input volume, in a terminal, run `alsamixer`. Make sure the card showing is: USB PnP Sound Device.

You can use arrows to move between Speaker, Mic and auto gain, and also change the gain. Use the M key to toggle between on (00) and Mute (MM). Use the Keys Q and Z to change the gain of the left speaker, and E and C to change the gain of the right speaker.

For the sonar to work well we will need to use either the right or left speaker, but not both. Make sure the gain for one of them is 0. Also, Make sure that the Auto-Gain-Control is disabled.

Instruction for laptops (for anyone wanting to try this also on a laptop not EE123)

- On Apple computers it is recommended that you turn off the ambient noise reduction by going to system-preferences, selecting sound, choose the input tab and make sure that the "Use ambient noise reduction" box is unchecked. In some windows system there's ambient noise reduction as well. Make sure it is also turned off.
- Your laptop most likely has two speakers. It is best if we work only with one. Go to the operating system's sound settings and change the stereo settings such that the speaker that is closest to the microphone is active. Your result will be much better that way.

- Make sure your output volume is at 70-80% and that the laptop's microphone is on, again to avoid non-linear distortions.
- We will record 12 seconds just to make sure we capture the entire sequence.

The code below defines some functions to use with pyaudio – a multi-platform audio python interface.

```
In [65]: def play_audio( Q, p, fs , dev=None):
    # play_audio plays audio with sampling rate = fs
    # Q - A queue object from which to play
    # p - pyAudio object
    # fs - sampling rate
    # dev - device number

    # Example:
    # fs = 44100
    # p = pyaudio.PyAudio() #instantiate PyAudio
    # Q = Queue.queue()
    # Q.put(data)
    # Q.put("EOT") # when function gets EOT it will quit
    # play_audio( Q, p, fs,1 ) # play audio
    # p.terminate() # terminate pyAudio

    # open output stream
    ostream = p.open(format=pyaudio.paFloat32, channels=1, rate=int(fs),output=True,out
    # play audio
    while (1):
        data = Q.get()
        if data is "EOT" :
            break
        try:
            ostream.write( data.astype(np.float32).tostring() )
        except:
            break

def record_audio( queue, p, fs ,dev=None,chunk=2048,lock=None):
    # record_audio records audio with sampling rate = fs
    # queue - output data queue
    # p - pyAudio object
    # fs - sampling rate
    # dev - device number
    # chunk - chunks of samples at a time default 1024
    #
    # Example:
    # fs = 44100
    # Q = Queue.queue()
    # p = pyaudio.PyAudio() #instantiate PyAudio
    # record_audio( Q, p, fs, 1) #
```

```

# p.terminate() # terminate pyAudio

istream = p.open(format=pyaudio.paFloat32, channels=1, rate=int(fs), input=True, input_device_index=device_index)

# record audio in chunks and append to frames
frames = [];
while (1):
    try: # when the pyaudio object is destroyed, stops
        with lock if lock is not None else 1:
            data_str = istream.read(chunk, exception_on_overflow=False) # read a chunk of audio
    except:
        break
    data_flt = np.fromstring( data_str, 'float32' ) # convert string to float
    queue.put( data_flt ) # append to list

def xciever(sig, fs):
    # function takes a signal and a sampling frequency
    # it then plays and records at the same time. The function returns
    # the recorded sound.

    rcv = [];

    # create an input output FIFO queues
    Qin = queue.Queue()
    Qout = queue.Queue()

    #lock for controlling access to shared resources
    lock = Lock()

    # create a pyaudio object
    p = pyaudio.PyAudio()

    # initialize a recording thread.
    t_rec = threading.Thread(target = record_audio, args = (Qin, p, fs ), kwargs={'loop':True})
    t_play_audio = threading.Thread(target = play_audio, args = (Qout, p, fs ))

    # start the recording and playing threads
    t_rec.start()
    t_play_audio.start()

    Qout.put( sig );
    Qout.put( "EOT" );

    # pause for RECORD_SECS seconds
    RECORD_SECS = len(sig)/fs + 2.0
    time.sleep( RECORD_SECS )
    # terminate pyAudio
    with lock:

```

```

p.terminate()

# append to output
while ( not Qin.empty() ) :
    data = Qin.get()
    rcv = np.append( rcv, data )

return rcv

```

Playing and recording audio:

- Run the following code. It is an example of how to play and record sound at the same time and uses threading for the play and record threads.

The resulting received sequence will be stored in the variable `rcv_chirp`.

In [63]: *## Play and record chirp at the same time*

```

fs = 48000 # sampling rate = 48000 Hz

rcv_chirp = xciever(s_chirp[0], fs)
Qin = queue.Queue()
Qout = queue.Queue() #!/ Note: queue instead of Queue for python3

```

- Plot the frequency response of the received sequence.
- Also, plot the absolute value of the received signal. Plotting the absolute value (sort of) displays the envelope of the chirp.

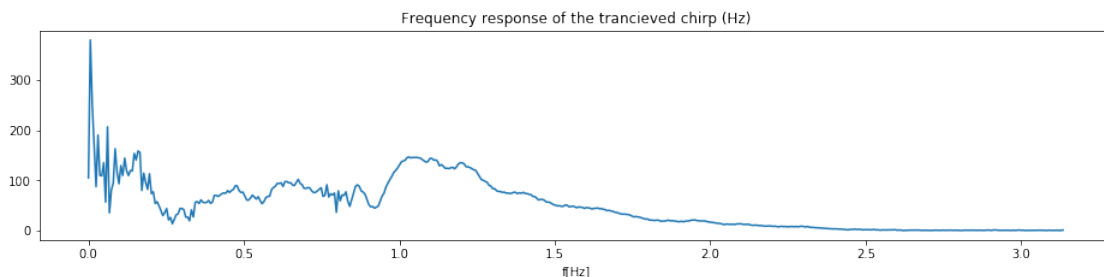
Label the figures and use an aspect ratio of Height/Width = 0.2

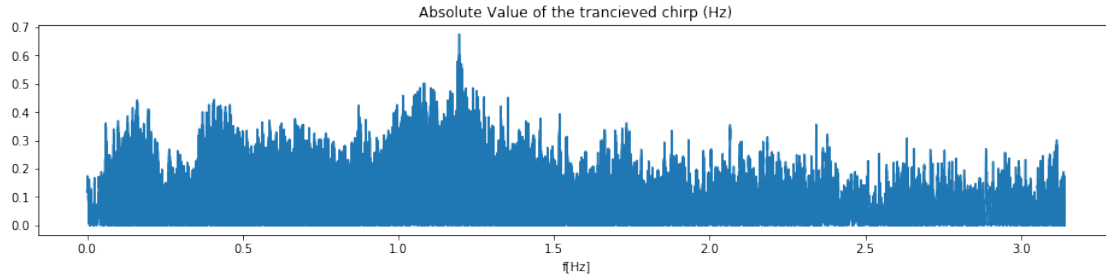
In [64]: *## Plot chirp response*

```

plot_fr(rcv_chirp, 'h', title='Frequency response of the trancieved chirp (Hz)')
plot_fr(rcv_chirp, 'a', title='Absolute Value of the trancieved chirp (Hz)')

```





Out [64]: 1

- Comment on the results you got. In addition, what is the implicit assumption we are making in order to claim that the result is a frequency response? (HINT: consider the case when the chirp was very short)

Answers here: Though we expected a constant frequency response over the swept range, the spectrum of the received chirp shows a large spike in the low-frequency range and tapers out at high frequencies. This is likely because the higher-frequency signal doesn't have as much energy as the low-frequency signal and therefore does not exert as much force on the microphone. The implicit assumption is that the system is able to sample and respond to each frequency in the swept range. With a short chirp, the system isn't able to sample as many frequencies since the sweep is faster and the sampling rate of the microphone doesn't change.

1.1.3 Part I, Task III: Envelope detection with Hilbert transform.

The absolute value of the result "sort of" displays the envelope, however it is still modulated by the (now rectified) frequency sweep carrier. If we write down the response, it can be expressed approximately as

$$y[n] = |H[n]| \sin(2\pi(f_0 + k[n * T])nT + \angle H[n])$$

where $|H[n]|$ is the frequency response for the instantaneous frequency at the n th sample and $\angle H[n]$ is its phase response.

The reason that it is only an approximation is that there is an inherent assumption that we do not look at transient effects, only steady state effect for each frequency. This is a good approximation because our chirp is very slow compared to the propagation of sound in the room.

One way to get the envelope $|H[n]|$ is to convert it to its analytic signal. The analytic signal $x_a(t)$ of signal $x(t)$ is:

$$x_a = F^{-1}(F(x) \cdot 2U) = x + jy$$

where F is the Fourier transform, U the unit step function, and y "is" the Hilbert transform of x . In other words, the negative half of the frequency spectrum is zeroed out, turning the real-valued signal into a complex signal. This is similar to the question in HW2!

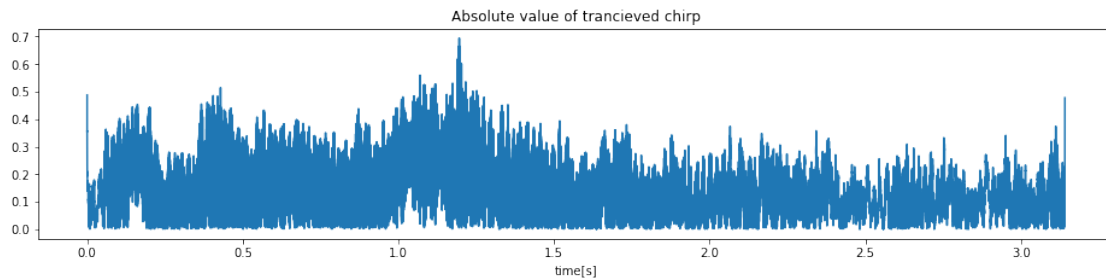
The analytic signal of the received chirp will then be:

$$y_a[n] = |H[n]| e^{j2\pi(f_0 + k[n * T])nT + \angle H[n]}$$

The envelope can be detected by taking the magnitude. (analytic function of y seems to have a one more phase shift $\pi/2$ since it is a sin function)

- Compute the analytic signal by using the function `signal.hilbert` and plot its absolute value. Note that the discrete hilbert transform is not perfect, since it uses FIR filtering. This will show up as ripple in the envelope.
- Label the figures and use an aspect ration of Height/Width = 0.2

In [67]: `plot_hil(rcv_chirp, 1, title='Absolute value of tranceived chirp', xunits = 'time[s]')`



Out [67]: 1

1.1.4 Part I, Task IV: Auto-correlation Properties of the Chirp:

In part II of the lab, we will be sending and receiving chirp pulses to estimate delays between the tranceived pulses. This is done by cross correlating / matched filtering the received signal with the known chirp pulse to detect the echoes. In this task, we will investigate the correlation properties of the chirp.

A cross correlation is defined as:

$$R_{xy}[n] = \sum_{m=-\infty}^{\infty} x[m]y^*[m-n] = (x[m] * y^*[-m])[n]$$

, where $y^*[-m]$ is the complex conjugat of $y[-m]$. This similar to a convolution, without flipping one of the signals. It can be implemented using a convolution as shown above. In general, the more correlated the two signals is at position n , the higher the value will be. That's why it is useful in a sonar system.

Matched filter When we look for a very specific shape in a signal, we can comput a cross correlation between the signal and the shape we are interested in. In that case, the operation of the cross correlation is also called a matched filter – i.e. correlating with a filter that is matched to the shape we look for.

Because we will be doing cross-correlations between a chirp pulse and its echoes, it is useful to look at the auto-correlation, which is basically a cross correlation of the signal with itself. A discrete autocorrelation of a signal is defined as:

$$R_{xx}[n] = \sum_{m=-\infty}^{\infty} x[m]x^*[m-n] = (x[m] * x^*[-m])[n]$$

The chirp has a very nice property that its auto-correlation is very narrow. Since the spread of the resulting correlation determines how fast you can detect, the width of the auto-correlation is important. This property is called pulse compression and is widely considered in radar design. Random noise and some other pseudo-random like sequences also possess this property.

- Generate a 512 sample chirp pulse with a frequency sweep from 17KHz-18KHz and sampling rate fs=48000.
- Validate its frequency response by plotting it.

```
In [69]: %matplotlib inline
        ## Your beautiful code here:

        fs = 48000.0

        f0 = 17000.0
        f1 = 18000.0
        f = np.r_[f0:f1:(f1-f0)/512.0]

        # generate time index and
        T = 512.0/fs

        t = r_[0.0:T:1/fs]
        f_sweep = t*(f1-f0)/T + f0
        phi = (t**2)*((f1-f0)/(2*T)) + t*f0

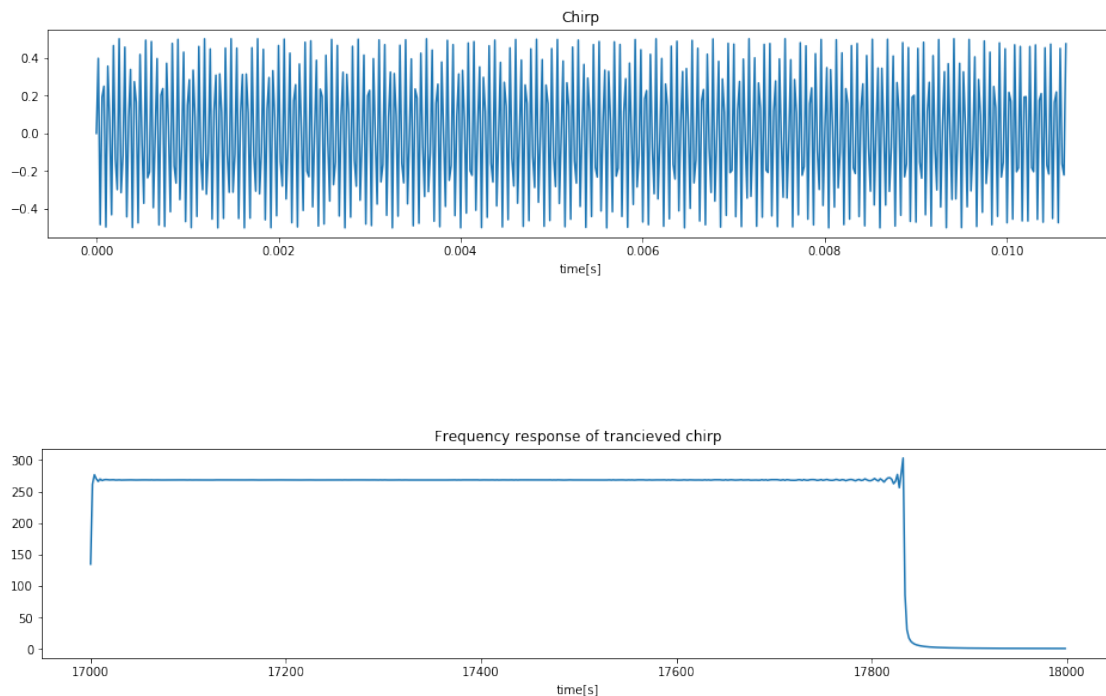
        # generate chirp signal
        new_chirp = 0.5*np.sin(2*np.pi*(t+phi))
        w, h = signal.freqz(s_chirp[0])
        # generate frequency response of chirp

        # plot
        width, height = figaspect(0.2)
        fig1 = figure(figsize=(width,height))

        plt.plot(t, new_chirp)
        plt.title('Chirp');
        plt.xlabel('time[s]')
        plt.show()

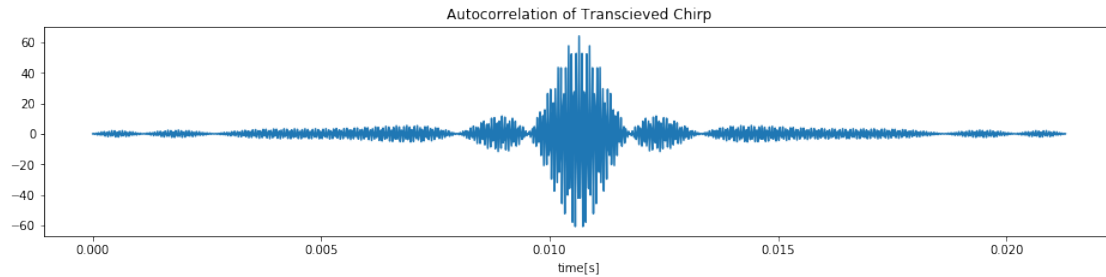
        width, height = figaspect(0.2)
        fig1 = figure(figsize=(width,height))
        plt.plot(f, abs(h))
```

```
plt.title('Frequency response of transcieved chirp');
plt.xlabel('time[s]')
plt.show()
```



- Compute the autocorrelation of the chirp “using” discrete convolution, either with `signal.convolve` or `signal.fftconvolve`. Remember that you have to flip the signal since convolution does that already. You can flip a signal `x` by doing `x[::-1]`. Use `mode="full"` for convolution.
- Plot the autocorrelation. Your plot should be spiky because we did not do envelope detection yet. Use milliseconds as the x-axis

```
In [70]: ## Your fantastic code here:
width, height = figaspect(0.2)
fig1 = figure(figsize=(width,height))
rev_chirp = new_chirp[::-1]
autocorr = signal.fftconvolve(new_chirp, rev_chirp, mode="full")
t2 = r_[0.0:1023]/fs
plt.plot(t2, autocorr)
plt.title('Autocorrelation of Transcieved Chirp');
plt.xlabel('time[s]')
plt.show()
```



In a similar way as we did before, it is possible to recover the envelope of the autocorrelation by performing a cross-correlation with the analytic signal and then taking the absolute value. In this case, we know exactly what the analytic function is!

- Generate `s_chirp_a`, the analytic function of the chirp by computing: `s_chirp_a = exp(1j*phi_of_t)`. Perform cross correlation between `s_chirp_a` and `s_chirp` and show the envelope. As stated before, this could also be called a matched filter.
- Measure the full-width at half max (FWHM) of the main lobe of the autocorrelation.
- Comment on the FWHM of the main lobe of the matched-filter with respect to the length of the pulse. That ratio is also called pulse compression. For simplicity, normalize the plot such that the maximum is 1, but record the maximum value of the autocorrelation and display it in the title of the figure.

Use the pragma `%matplotlib notebook` for making the figure interactive, and `plt.grid('on')` for displaying a grid.

Use milliseconds as the x-axis

```
In [93]: #matplotlib notebook
         #matplotlib notebook
         %matplotlib inline
         fs = 48000
         f0 = 20
         f1 = 20000
         f = np.r_[f0:f1:(f1-f0)/512]

         # generate time index and
         T = 512/fs
         t = r_[0.0:T:1/fs]
         f_sweep = t*(f1-f0)/T + f0

         # generate time index and phase
         phi_new = (t**2)*((f1-f0)/(2*T)) + t*f0

         # generate chirp signal
         s_chirp_new = 0.5*np.sin(2*np.pi*(t +phi_new))
```

```

s_chirp_a = exp(1j*2*np.pi*phi_new)
s_chirp_a_reverse = s_chirp_a[::-1]
corr = signal.fftconvolve(s_chirp_new, s_chirp_a_reverse, mode="full")

print("s_chirp: " + str(len(s_chirp_new)))
print(len(s_chirp_a_reverse))
width, height = figaspect(0.2)
fig1 = figure(figsize=(width,height))

t2 = r_[0.0:1022.5/fs:1/fs]
print(len(corr))
print(len(t2))
plt.plot(t2, abs(corr))
plt.title('Cross-correlation of Chirp and Analytic Chirp');
plt.xlabel('time[s]')
plt.xlim(0.009, 0.012)
plt.show()
# your nice script to produce beautiful chirps, xcorrelations and figures here:

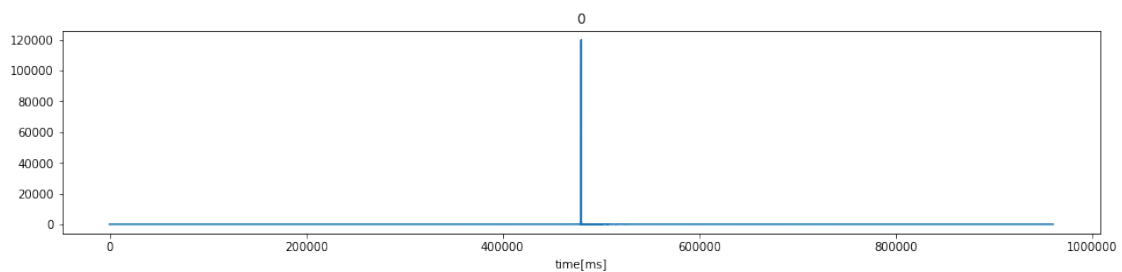
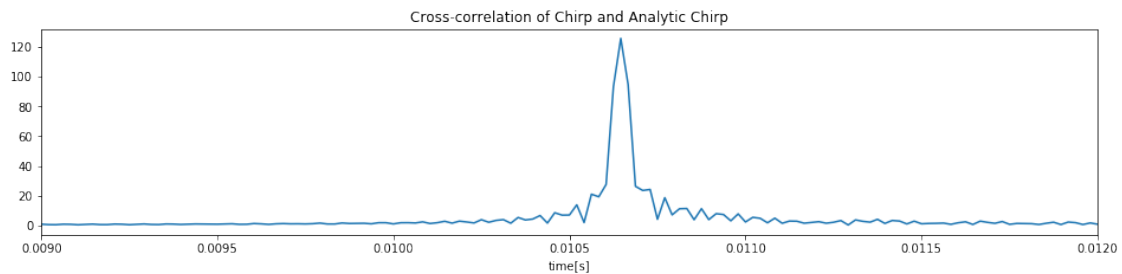
xc = xcorr(s_chirp[0], s_chirp[1], 0)

```

```

s_chirp: 512
512
1023
1023

```



Your answer here: The longer the pulse, the narrower the main lobe. The FWHM is about 60.

Now we will look at why the chirp pulse is better for cross-correlation detection than a pure tone. - Repeat Task “III” <- “IV” for: 1. A constant frequency of 17000Hz, 512 samples in length. 2. A chirp with a frequency sweep from 16500Hz - 17500Hz (1KHz Bandwidth), 512 in length. 3. A chirp with a frequency sweep from 15000Hz - 19000Hz (4KHz Bandwidth), 512 in length - Compare the size of the main lobes (full width at half max). How much “Pulse Compression” are you getting by using a chirps for detection compared to a single frequency pulse? - What is the approximate bandwidth of the pure frequency pulse and what is the bandwidth of the chirp pulses? Comment on the tradeoff between bandwidth and pulse compression - What is the maximum autocorrelation for each pulse?

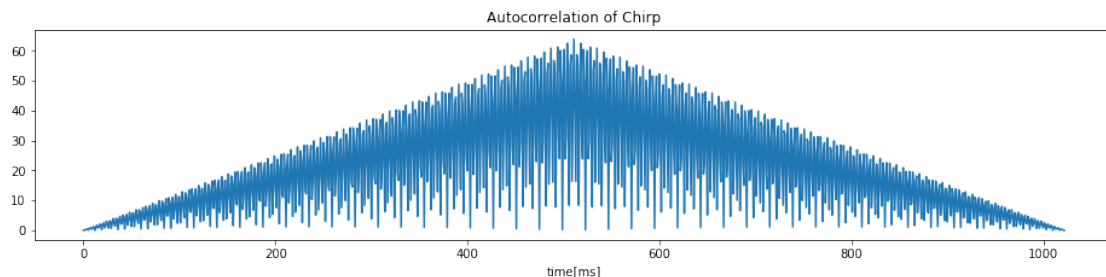
```
In [94]: # your solution here
```

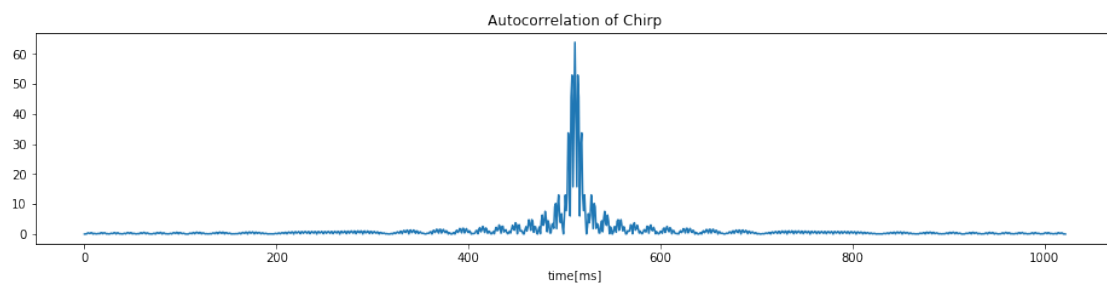
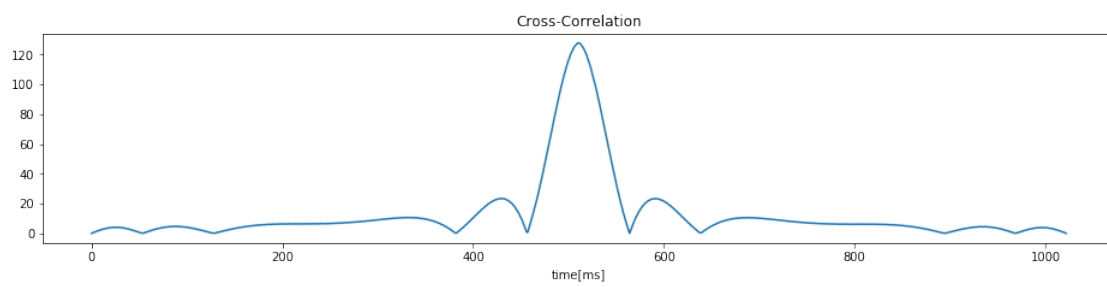
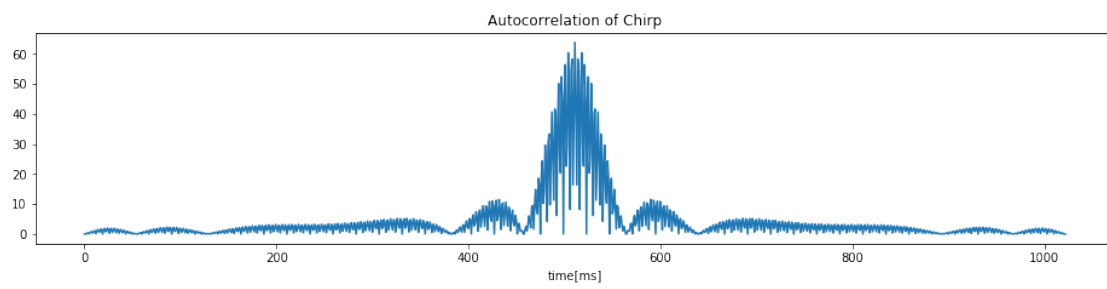
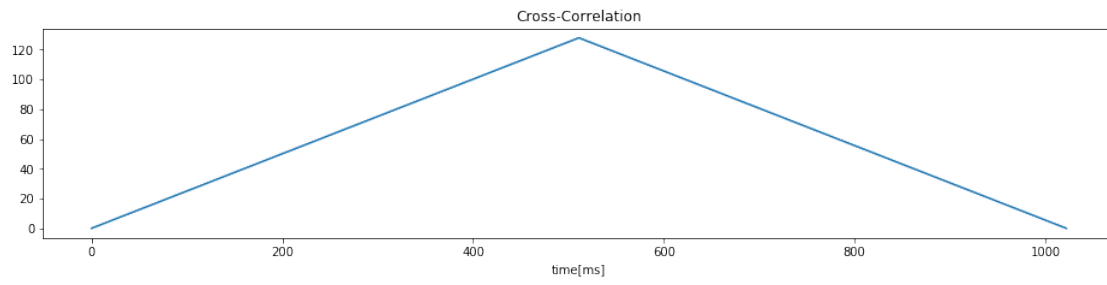
```
fs = 48000
T = 512/fs
t = r_[0.0:512]/fs

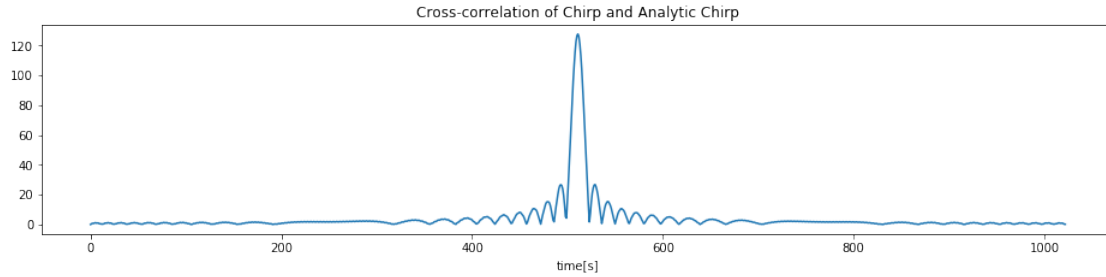
sig_1 = generate_chirp(512, 17000, 17000)
sig_2 = generate_chirp(512, 16500, 17500)
sig_3 = generate_chirp(512, 15000, 19000)
```

```
In [96]: acorr(sig_1[0])
xcorr(sig_1[0], sig_1[1])
acorr(sig_2[0])
xcorr(sig_2[0], sig_2[1])
acorr(sig_3[0])
xcorr(sig_3[0], sig_3[1])

plt.title('Cross-correlation of Chirp and Analytic Chirp');
plt.xlabel('time[s]')
plt.show()
```







Your answers and interpretations here: The autocorrelation of a constant frequency signal takes up is 11.6ms, the main lobe for the 1Khz while chirp is ~1.24ms and for the 4Khz chirp is ~0.3ms. Their pulse compression factors are about 9.6 and 38.6 respectively. At the same time, the bandwidth of the constant frequency pulse is approximately $1/(\text{its length}) = 86 \text{ Hz}$, and the bandwidth for the chirps are 1KHz and 4KHz – these are inversly proportional to the main lobe width! This exemplifies the time-frequency uncertainty principle: compression in one domain corresponds to expansion in the other.

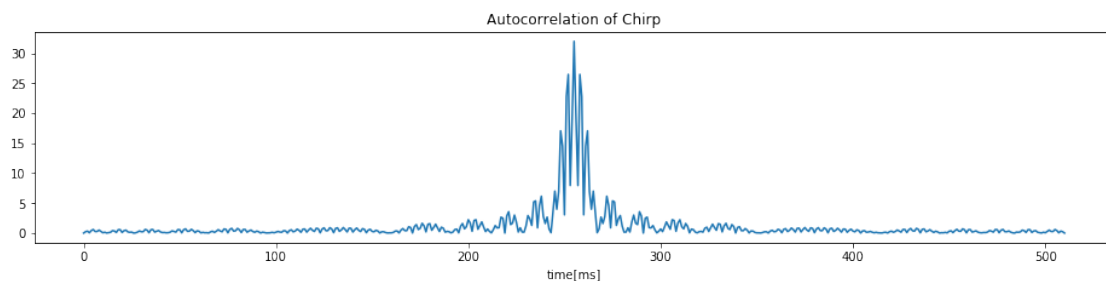
- Now, repeat task “III” <- “IV” for

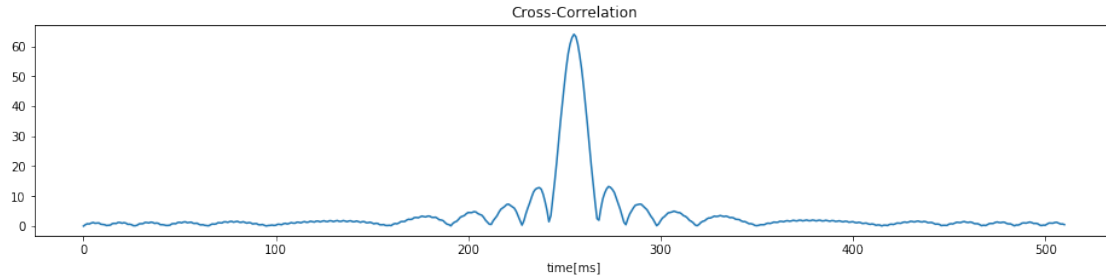
1. a chirp with a frequency sweep from 15000Hz - 19000Hz, 256 in length

- Compare the size of the main lobe (full width at half max) to the previous case of 15000Hz - 19000Hz, 512 in length.
- Compare the maximum autocorrelation as well.

What's the effect of having more bandwidth? what's the effect of having longer/shorter pulses?

```
In [98]: # your solution here
sig_4 = generate_chirp(256, 15000, 19000)
a_corr = acorr(sig_4[0])
x_corr = xcorr(sig_4[0], sig_4[1])
```





Your answer below: The pulse compression is proportional to the bandwidth. The peak correlation is proportional to the length of the pulse.

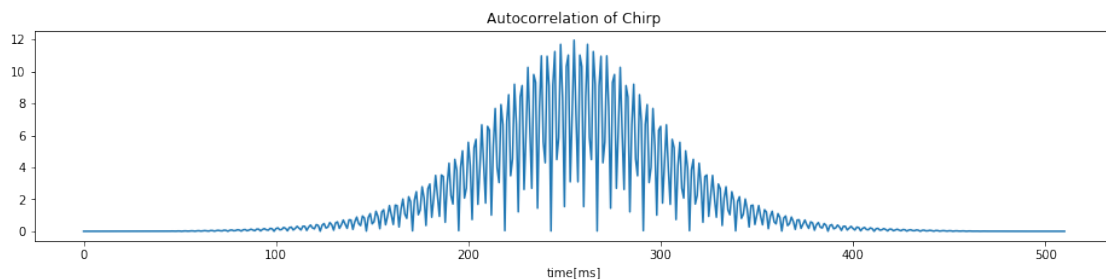
1.1.5 Dealing with sidelobes

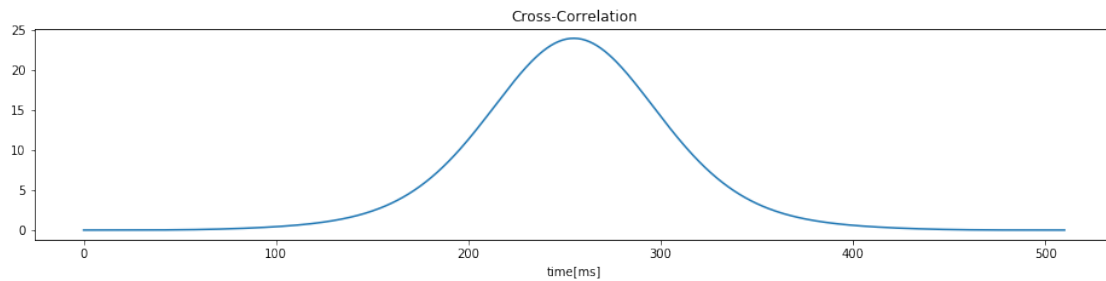
As you can see, the chirp provides good pulse compression of the main-lobe. However, there exists very strong sidelobes. This is because the chirp is multiplied with a rect function, that is abrupt. Instead, we will window the chirp with one of the smooth window functions to taper off the sidelobes.

- Repeat the above for a chirp with a sweep from 16.5KHz to 17.5KHz, and from 15KHz to 19KHz. This time, multiply the chirp (and its analytic function) with a hanning window. You will find the function `np.hanning` useful.
- plot the normalized autocorrelations (in the same figure)
- Comment on the magnitude of the side-lobes?
- Comment on the width of the main lobes?
- What's the tradeoff?

```
In [99]: # your solution here
w_hanning = np.hanning(256)

sig_5 = generate_chirp(256, 16500, 17500)
a_corr = acorr(w_hanning*sig_5[0])
x_corr = xcorr(w_hanning*sig_5[0], w_hanning*sig_5[1])
```





Your answers here: The width of the main lobe is doubled and the maximum autocorrleation is reduced. But, the sidelobes are significantly smaller!

1.1.6 You are now ready to proceed to the Sonar Lab

1.2 Part 2: Sonar

In this part of the lab we will write a simple application that implements a sonar using the laptop internal speaker and microphone.

The basic idea is very simple and is the basis of sonar and ultrasound images – Objects reflect sound waves. If we send a pulse of sound, we will get reflection echoes of that pulse. Detecting the echos and their time-of-flight will reveal their distance from the source, based on the speed of sound in air.

The way we are going to implement the sonar is to generate a series of rapid pulses, and use matched filtering to detect the source and the returning echos. There are many parameters in this lab that can be tweaked to get different results. We encourage you to experiment. We enjoyed very much making the lab and played quite a bit! We hope you enjoy it too.

Instructions for RaspberryPi: The microphone and speaker you have are somewhat directional. Make sure that the microphone and speaker point in the same direction. You will get the best quallity in a quiet room, without interference from other sources of noise – especially from your fellow students playing chirp pulses at the same time as you are!

Instructions for other laptops: Unfortunately, the quallity of the sonar system is going to be highly dependent on your laptop quallity, and the position of the speakers and microphone. It is recommended that you adjust the sound settings on your system such that only the speaker that is closer to the microphone is active. For example, MacBookAirs have the microphone on the side of the computer – so you should set adjust the audio settings to left speaker only. Also, it is recommended that the speaker volume be set to half of its maximum to avoid non-linear distortions.

If you are getting poor results, please consult with us.

This lab was inspired from an iphone app called active-radar.

1.2.1 Part II, Task I: Generating Chirp Pulses

Recall from Part I, that the width of the main lobe of the autocorrelation depends on the bandwidth of the pulse. For a constant frequency pulse, the bandwidth will be inversely proportional to its length. Short pulses are localized in time, and therefore we will be able to separate echoes from targets that are close. However, short pulses carry less energy (for the same amplitude) and this will reduce our signal to noise ratio (SNR) in the detection and reduce our ability to detect the targets at all. So, in summary: for constant frequency pulse, there's an inherent tradeoff between the resolution of the sonar (distinguish between close targets) and the signal to noise ratio.

If we use a chirp pulse, we can increase the length of the pulse while also increasing the bandwidth. This will enable us to improve our signal to noise ratio as well as keeping the resolution of our sonar (by preserving the BW).

In our implementation we are going to design a pulsed sonar system in which we repeatedly send pulses and then listen to the returning echoes. The arrival time of the echoes will correspond to double the time-of-flight of sound propagation from our system to the target.

- Write a function that generates a chirp pulse: `pulse = genChirpPulse(Npulse, f0, f1, fs)`

The function will accept: `Npulse` = number of samples, `f0,f1` = starting and ending frequency and `fs` = sampling frequency. The function will return the analytic function of the chirp $\exp(j2\pi \int_0^t f(t)dt)$ with amplitude 1.

```
In [100]: def genChirpPulse(Npulse, f0, f1, fs):
#         Function generates an analytic function of a chirp pulse
#         Inputs:
#             Npulse - pulse length in samples
#             f0      - starting frequency of chirp
#             f1      - end frequency of chirp
#             fs      - sampling frequency
#         Output:
#             pulse - chirp pulse
t = np.r_[0:Npulse]/fs
phi_of_t = (t**2)*((f1-f0)/(2*Npulse/fs)) + t*f0
phase = 2*np.pi*phi_of_t
return np.exp(1j*phase)
```

- To validate that the function works display the pulse generated with `Npulse = 200`, `f0=1000`, `f1 = 8000`, `fs = 48000`. Remember the pulse is complex, so plot the real and imaginary part separately.

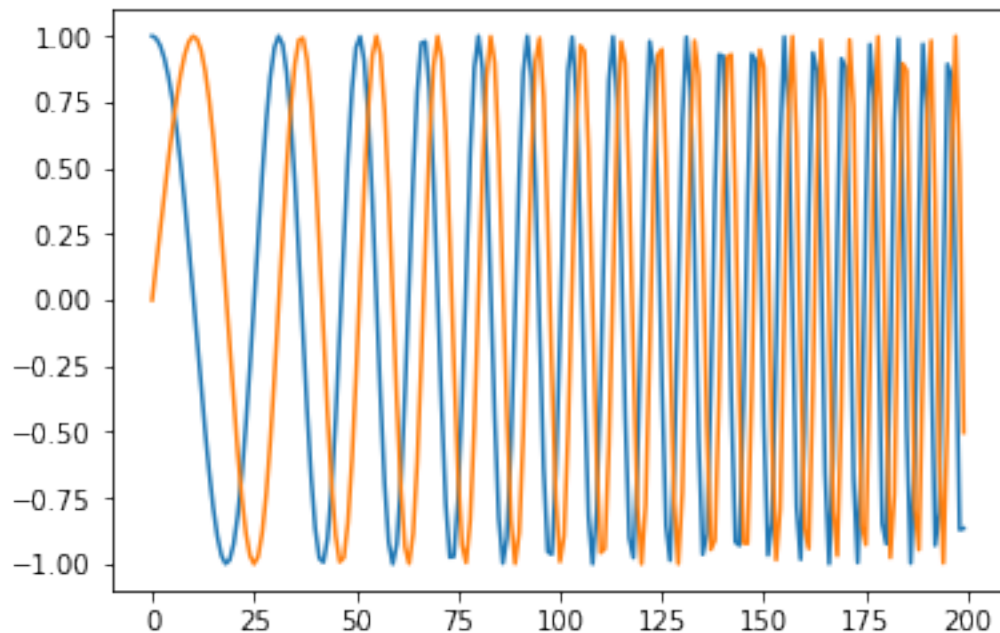
```
In [101]: %matplotlib inline
%matplotlib inline
%matplotlib inline
```

```

# your code here:
pulse = genChirpPulse(200, 1000, 8000, 48000)
# plot
plt.plot(np.real(pulse))
plt.plot(np.imag(pulse))

```

Out[101]: [<matplotlib.lines.Line2D at 0x6ddfc310>]



Generate Pulse Trains

Next, we will use the pulse generated by `genChirpPulse` and generate a pulse train.

- Write a new function `ptrain = genPulseTrain(pulse, Nrep, Nseg)` The function accepts `pulse` = a pulse generated by `genChirpPulse`, `Nrep` = number of pulse repetitions and `Nseg` = length of each pulse train segment (which is \geq to the length of `pulse`).

The function returns `ptrain` which is a vector of length `Nrep x Nseg` (Hint: use `np.tile`)

```

In [102]: def genPulseTrain(pulse, Nrep, Nseg):
            # Function generates a pulse train from a pulse.
            #Inputs:
            # pulse = the pulse generated by genChirpPulse
            # Nrep = number of pulse repetitions
            # Nseg = Length of pulse segment  $\geq$  length(pulse)
            zpad = array(np.zeros(Nseg - len(pulse)))

            return np.tile(np.hstack((np.array(pulse), zpad)), Nrep)

```

1.2.2 Part II, Task II: Echos in with Chirp pulse train

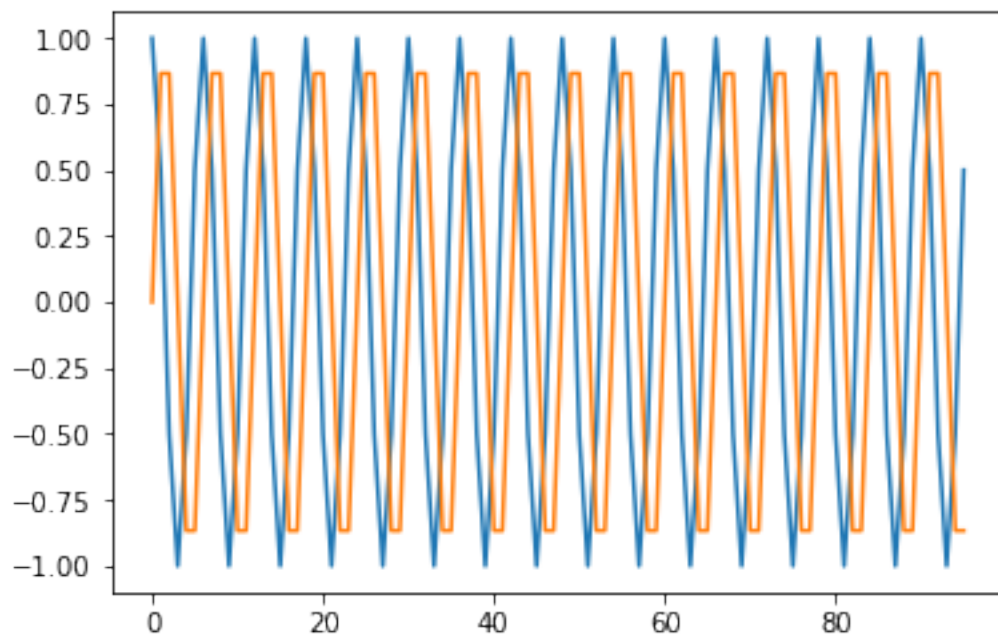
We now have components to generate pulses, generate a pulse train, play and record it. Lets see what we get! We will start with very short pulses with a single carrier frequency. Rectangular pulses are difficult for the speaker to produce as they exhibit discontinuities in the beginning and the end of the pulse. Therefore we will multiply the pulses with a smooth window. Here, we will use a hanning window.

- Generate a $f_0=f_1=8\text{KHz}$, $N_{\text{pulse}}=96$ pulse with $f_s=48000$. Window the pulse with a hanning window. This will result in a pulse length of 2ms. You should be able to hear this tone.
- Plot the real and imaginary part of the pulse

```
In [163]: fs = 48000
          f0 = 8000
          f1 = 8000
          Npulse = 96

          # your code here:
          w_hanning = np.hanning(Npulse)
          generated_pulse = genChirpPulse(Npulse, f0, f1, fs)
          windowed = generated_pulse*w_hanning
          plt.plot(np.real(generated_pulse))
          plt.plot(np.imag(generated_pulse))
```

```
Out[163]: [<matplotlib.lines.Line2D at 0x6dd500d0>]
```



- Use the real part of the pulse to generate a pulse train of $N_{\text{rep}}=15$ pulses, $N_{\text{seg}}=4096$ samples

```
In [167]: # your code here:
```

```
ptrain = genPulseTrain(np.real(generated_pulse), 15, 4096)
```

- Play and record the pulse train. Scale the amplitude of the pulses to 1/2. Make sure your volume is set to maximum of 70% and look at the plot with the input pulse train and the received pulse train.

Use the pragma `%matplotlib notebook` for interactive plots, so you can zoom into the result.

```
In [168]: rcv = xciever(ptrain/2.0 , fs)
```

```
In [169]: %matplotlib notebook
          %matplotlib notebook
          %matplotlib notebook
          %matplotlib notebook
```

```
width, height = figaspect(0.2)
fig = figure(figsize=(width,height))
```

```
plt.plot(ptrain)
plt.plot(rcv)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

- Extract a single pulse from the received pulse train. You can find the pulse index from the interactive plot. Extract at least 2 Npulse samples before the pulse and 20 Npulse samples after using `rcv_pulse = rcv[idx-2*Npulse:idx+Npulse*20]`
- Plot the received pulse. Can you see any echoes?

You can disable interactivity by the pragma `matplotlib inline`

```
In [170]: %matplotlib notebook
          # your code here:
          # find index of start pulse
          width, height = figaspect(0.3)
          fig = figure(figsize=(width,height))
          idx = 920
          rcv_pulse = rcv[idx-2*Npulse:idx+Npulse*20]
          plt.plot(rcv_pulse)
          # your code here
          # find index of start pulse
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[170]: [matplotlib.lines.Line2D at 0x6dc89870]
```

There is an echo about 10 samples after the end of the first received pulse.

Matched Filtering The strong pulses we see are a result of direct feed-through from the transmitter to the receiver that do not scatter off targets. The echoes we see are a result of echoes from reflecting surfaces. The problem in our setup is that we don't know the exact delay between the transmitter and the receive hardware (in PyAudio). Instead, we will assume that the travel time for sound between the speaker and the microphone is negligible and much smaller than scattering targets. We can then detect when the pulses start based on the direct feedthrough signal. This assumption is very good as long as your speaker is close to the microphone!

We will detect both the feedthrough and echoes using matched filtering.

- Write a function `Xrcv = crossCorr(rcv, pulse_a)` to calculate the cross correlation (matched filter) of the received signal with the analytic function of the pulse. You can use `signal.fftconvolve`
- Take the absolute value of `Xrcv` to recover its envelope. Call the result `Xrcv_a`.

Make sure the plot is interactive with `matplotlib notebook`

```
In [171]: Xrcv = xcorr(rcv, pulse)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

- Again, extract a single pulse from the received pulse train using the same index. Extract at least 2 Npulse samples before the pulse and 20 Npulse samples after. Plot the received pulse. Can you see any echoes?

The echo is much harder to detect in this case.

```
In [173]: %matplotlib notebook
          # find index of start pulse

          width, height = figaspect(0.3)
          fig = figure(figsize=(width,height))
          idx = 920
          Xrcv_a = Xrcv[idx-2*Npulse:idx+Npulse*20]
          plt.plot(np.real(Xrcv_a))
          # plot
```


<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Out[173]: [<matplotlib.lines.Line2D at 0x6dcc59b0>]

Sonar System In order to automate the system and visualize the results we need a few more components. To extract the pulses we need to know the position of the first feedthrough pulse.

- Write a function `idx = findDelay(Xrcv_a, Nseg)` that takes the result of the matched filtering and finds the index of the first feedthrough pulse. Try testing on the actual signal to check whether the function is correct. There are multiple ways of doing it. `Nseg` is not necessarily required.

```
In [174]: def findDelay(Xrcv, Nseg):
           # finds the first pulse
           # Inputs:
           #         Xrcv - the received matched filtered signal
           #         Nseg - length of a segment
           # Output:
           #         idx - index of the beginning of the first pulse
           return np.argmax(Xrcv[:Nseg])
```

```
In [175]: idx = findDelay(Xrcv_a, 1250)
           print(idx)
```

225

We now can correct for delays and detect echoes. The only thing left now is to convert the time between echoes into actual distance.

If we send a pulse of sound, we will get reflection echoes of that pulse. Detecting the echos and their time-of-flight will reveal their distance from the source, based on the speed of sound in air. The speed of sound in air is given by the following equation:

$$v_s = 331.5\sqrt{1 + T/273.15} \text{ m/s},$$

where T is the temperature in degree celcius.

- Create a function `t = dist2time(dist, temperature)` that takes in the distance to the target in cm and converts it into the time in seconds between the transmitted pulse and its echo. Remember the arrival time include the time to the target and back and therefore the time should be doubled. For example, for temperature = 20 celcius and dist = 400 cm, the time it takes is 0.023 secs.

- Create a function `dist = time2dist(t, temperature)` that takes in the time to the target in seconds and converts it into the distance in cm between the transmitted pulse and its echo. Remember the arrival time include the time to the target and back and therefore the time should be halved.

```
In [176]: def dist2time( dist, temperature=21):
           # Converts distance in cm to time in secs
           # Inputs:
           # dist          - distance to object in cm
           # temperature - in celcius
           # Output:
           # t             - time in seconds between transmitted pulse and echo
           vs = 331.5*np.sqrt(1+temperature/273.15)
           t = dist/vs
           return t

           def time2dist(t,temperature=21):
               # Converts time in seconds to distance in cm
               # Inputs:
               # t           - time of echo
               # temperature - in celcius
               # Output:
               # dist        - distance in cm of the target
               vs = 331.5*np.sqrt(1+temperature/273.15)
               d = vs*t
               return d*100
```

1.3 A sonar (almost)

- The following function will use your functions to generate pulses and display the matched filtering of each pulse as intensity of a horizontal line in an image. If nothing is moving, you will be able to see constant vertical lines representing echos. If something is moving, you will be able to track the object's distance.

```
In [177]: # some code for you

def sortOfASonar(Npulse, f0, f1,fs, Nrep, Nseg):
    pulse_a = genChirpPulse(Npulse, f0, f1, fs)*np.hanning(Npulse)
    pulse = pulse_a.real
    ptrain = genPulseTrain(pulse, Nrep, Nseg)
    rcv = xciever(ptrain/2.0 , fs)
    Xrcv_a = xcorr(rcv, pulse_a)

    idx = findDelay(Xrcv_a,Nseg)
    img = np.zeros((Nrep,Nseg))
```

```

img[0,:]=Xrcv_a[idx:idx+Nseg]

# Look for peak in each pulse in the pulse train to avoid drift between xmit and r
for n in range(1,Nrep):
    idxx = findDelay(Xrcv_a[idx+Nseg//2:idx+Nseg//2+Nseg],Nseg)
    idx = idx + idxx+Nseg//2
    img[n,:]=Xrcv_a[idx:idx+Nseg]

return img

```

Now, use the function above to:

- Generate a pulse train of 100 pulses. Each (hamming windowed) pulse should be length of 72 samples (1.5ms) and constant frequency of 8KHz. The spacing between pulses should be 0.1 seconds (Nseg=4800).
- Display the image

```

In [178]: Npulse = 72
          f0 = 8000
          f1 = 8000
          fs = 48000
          Nrep = 100
          Nseg = 4800
          img = sortOfASonar(Npulse, f0, f1,fs, Nrep, Nseg)

```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

```

/usr/local/lib/python3.5/dist-packages/ipykernel_launcher.py:12: ComplexWarning: Casting complex
if sys.path[0] == '':
/usr/local/lib/python3.5/dist-packages/ipykernel_launcher.py:18: ComplexWarning: Casting complex

```

Display the result. Pay attention to the width of the echos.

```

In [179]: %matplotlib inline
          %matplotlib inline

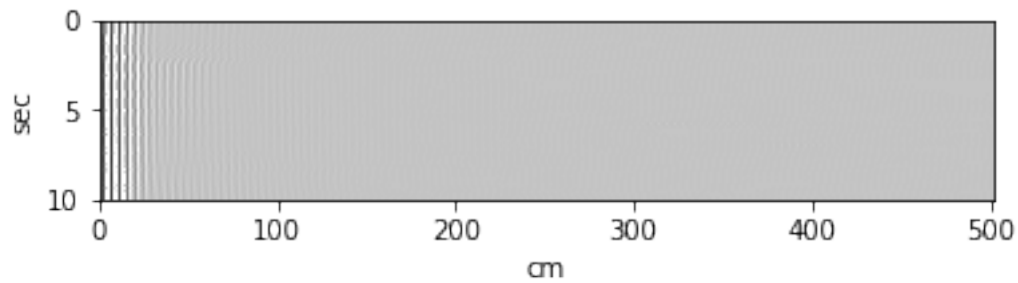
# display up to 2.5m approximately 700 samples at 21 degrees C in 48000 sampling rate

vmax = 0.3 # threshold -- lower will be able to see smaller echos

plt.imshow(img[:,0:700]/max(img.ravel()),vmax=vmax, aspect=10,cmap='gray',interpolation='nearest')
plt.xlabel('cm')
plt.ylabel('sec')

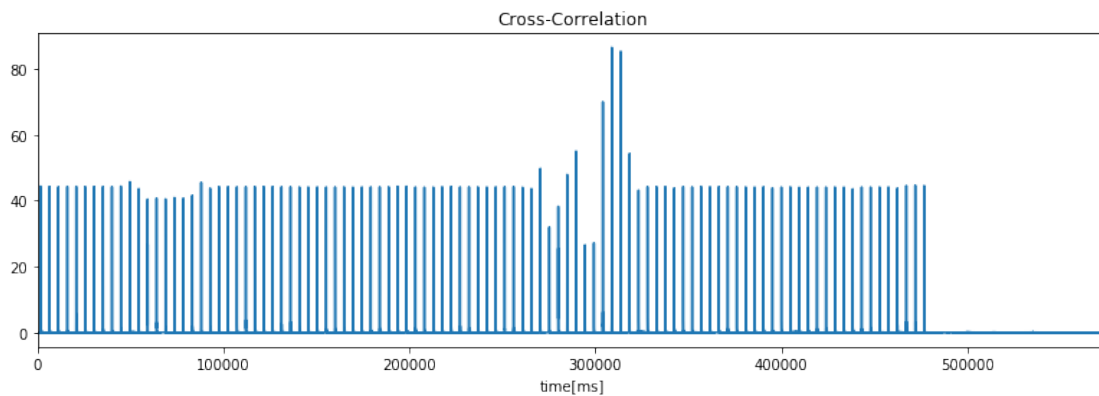
```

Out[179]: <matplotlib.text.Text at 0x6d144590>



Now, repeat the experiment with a chirp length of Nseg = 360 samples, and a frequency sweep from KHz to 12KHz. Pay attention to the resolution of the lines.

```
In [159]: Npulse = 72*5
          f0 = 6000
          f1 = 12000
          fs = 48000
          Nrep = 100
          Nseg = 4800
          img = sortOfASonar(Npulse, f0, f1, fs, Nrep, Nseg)
```



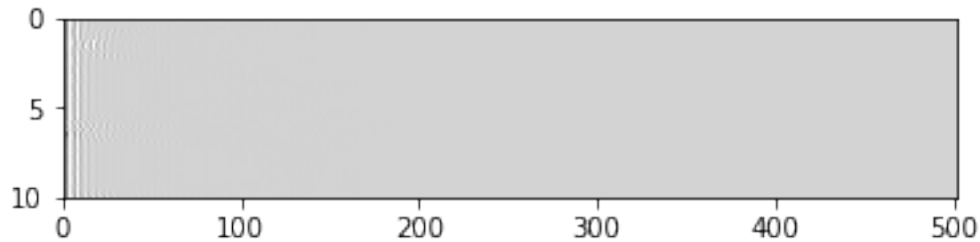
```
/usr/local/lib/python3.5/dist-packages/ipykernel_launcher.py:12: ComplexWarning: Casting complex
  if sys.path[0] == '':
/usr/local/lib/python3.5/dist-packages/ipykernel_launcher.py:18: ComplexWarning: Casting complex
```

```
In [160]: %matplotlib inline
          %matplotlib inline
          # display up to 2.5m approximately 700 samples at 21 degrees C in 48000 sampling rate
```

```
vmax = 0.3 # threshold -- lower will be able to see smaller echos
```

```
plt.imshow(img[:,0:700]/max(img.ravel()),vmax=0.2, aspect=10,cmap='gray',interpolation
```

Out[160]: <matplotlib.image.AxesImage at 0x6dd1efb0>



Feel free to repeat while moving a target– can you see the echoes changing? Try playing with different parameters.

1.4 A Real (time) Sonar

You now have a working sonar! It would be much easier though to play with different parameters if we automate things, so we created some wrappers for real-time plotting in a separate notebook (lab1-RealTime-Sonar).

- Copy-and-paste the 5 functions you created, including `genPulseTrain()`, `genChirpPulse()`, `crossCorr()`, `findDelay()`, and `dist2time()`, to the specified code cell in the real-time Sonar lab.

1.4.1 You are now ready to proceed to the Real-Time Sonar Lab