EE-679 Assignment 2

Linear Predictive Analysis

Soham Naha

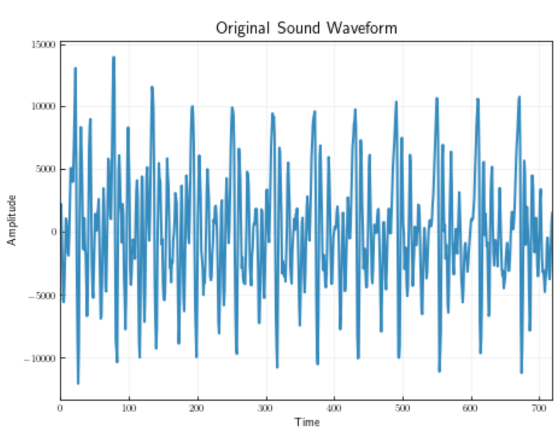
Roll : 193079003

November 1, 2020

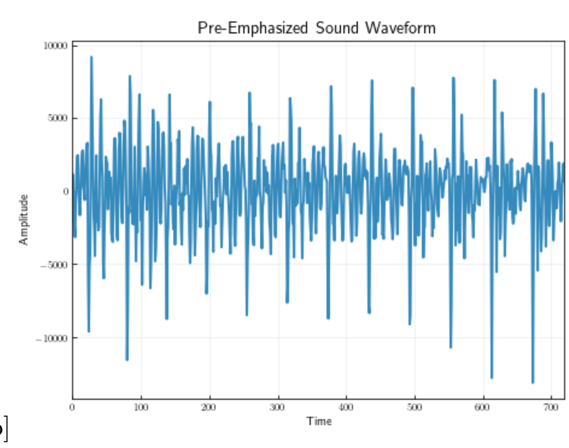
Given the speech segment *(aa.wav)* extracted from the word *"pani"* in *"machali.wav"* (male voice), sampled at 8 kHz, do the following.

1. **Apply pre-emphasis to the signal.**
   Pre-emphasis is applied as a high-pass filter, $P(z) = 1 - \alpha z^{-1}$, where α is a free-parameter, to make the spectrum of the speech signal a bit uniform. Here, α is chosen to be 0.95 by default.

```python
def pre_emphasize(sound,alpha=0.95,verbose=False):
    """Pre-emphasize the input sound signal
    :param sound : the speech signal to be pre-emphasized
    :param alpha : pre-emphasis parameter (0.9 <alpha < 0.99)
                   (y[n] = x[n] - alpha*y[n-1])
                    (default alpha = 0.95)
    :param verbose : if True saves the pre-emphasized waveform
    :output pre-emphasis : output of pre-emphasizing sound
    """
    alpha_pre_emphasis = alpha
    pre_emphasis = np.zeros_like(sound)
    pre_emphasis[0] = sound[0]
    for i in range(1,len(pre_emphasis)):
        pre_emphasis[i] = sound[i] - alpha_pre_emphasis*sound[i-1]

    if verbose:
        plt.figure()
        plt.title("Pre-Emphasized Sound Waveform")
        plt.plot(pre_emphasis)
        plt.grid(color='0.9', linestyle='-')
        plt.tight_layout()
        plt.xlim(xmin=0)
        plt.ylabel("Amplitude")
        plt.xlabel("Time")
        plt.autoscale(enable=True, axis='x', tight=True)
        plt.savefig("../plots/PreEmphasizedSound.png",bbox_inches="tight")

return pre_emphasis
```



[a]

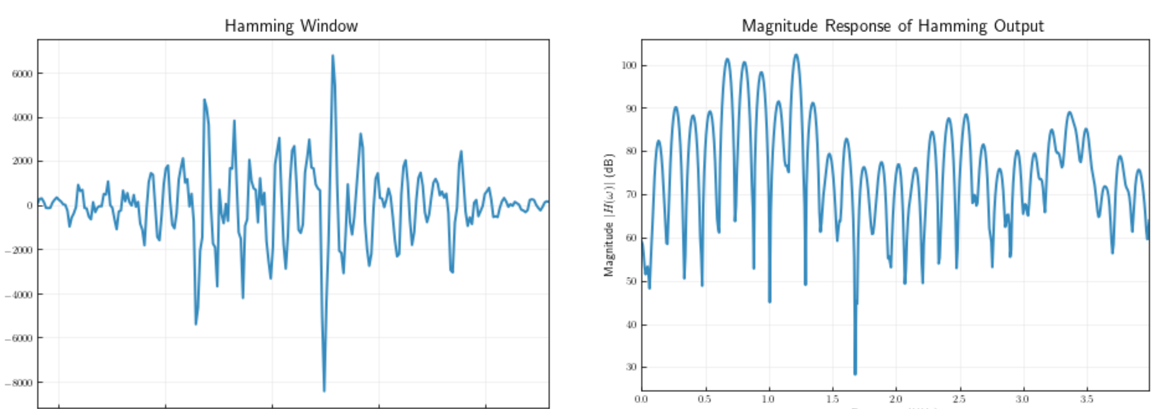


[b]

Figure 1: (a)/*aa*/ from aa.wav (b)Pre-Emphasized /*aa*/ from aa.wav

2. **Compute and plot the narrowband magnitude spectrum slice using a Hamming window of duration = 30 ms on a segment near the centre of the given audio file.**

```python
def hamming_window(sound,duration=30,center=True,verbose=True):
        """Calculate the hamming windowed segment of the speech signal
        :params sound : the speech signal to be windowed
        :params duration : the hamming window duration (default = 30ms)
        :params center : take the window from the center of the speech segmen if True
                         else take the segment from the starting of the speech segment
                         (default = True)
        :params verbose        : Plot the segmented waveform if True
        :outputs hamming_output : the windowed signal and saves the magnitude spectrum of the window
        """
    window_duration = int((duration/1000)*Fs)
    if center:
        center = len(sound)//2
        windowed_sound = sound[center-window_duration//2:center+window_duration//2]
        xticks = np.linspace(center-window_duration//2,center+window_duration//2,window_duration)
    else:
        windowed_sound = sound[:window_duration]

    #print(windowed_sound.shape)
    #print(window_duration)

    hamming_output = np.hamming(window_duration)*windowed_sound

    # Magnitude Response
    w,h_window = signal.freqz(hamming_output)
    plt.figure()
    plt.plot((Fs*w/(2*np.pi))/1000,20*np.log10(h_window))
    plt.grid(color='0.9', linestyle='-')
    plt.title("Magnitude Response of Hamming Output")
    plt.xlabel("Frequency (KHz)")
    plt.ylabel(r"Magnitude $|H(\omega)|$ (dB)")
    plt.tight_layout()
    plt.autoscale(enable=True, axis='x', tight=True)
    plt.savefig("../plots/MagnitudeResponseHamming_"+str(duration)+"ms.png",bbox_inches="tight")

    if verbose:
        plt.figure()
        if center:
            plt.plot(xticks,hamming_output)
            plt.xlim(xmin=xticks[0])
        else:
            plt.plot(hamming_output)
            plt.xlim(xmin=0)
        plt.grid(color='0.9', linestyle='-')
        plt.title("Hamming Window")
        plt.autoscale(enable=True, axis='x', tight=True)
        plt.tight_layout()
        plt.savefig("../plots/HammingOut.png",bbox_inches="tight")

    return hamming_output
```

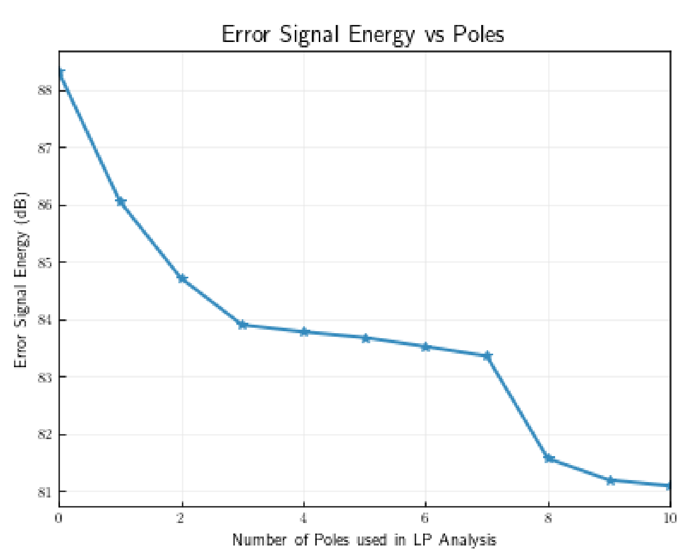


[a] [b]

Figure 2: (a)Windowed Signal with window duration 30ms (b)Magnitude spectrum of the windowed signal

3. **With the same 30 ms segment of part 2, compute the autocorrelation coefficients required for LPC calculation at various $p = 2, 4, 6, 8, 10$. Use the Levinson-Durbin recursion to compute the LP coefficients from the autocorrelation coefficients. Plot error signal energy (i.e. square of gain) vs p.**

```python
def LPanalysis(signal, p):
    """Calculates the coefficients and Gains of the filter using Levinson Durbin Recursion
        where P = LP order
    :params signal : windowed signal for parameter estimation
    :params p : order of the filter
    :outputs E: residual energy of filters upto order p
    :outputs G: Gains of the filters upto order p
    :outputs a: filter coefficients of filters upto order p
    """
    R = np.correlate(signal,signal, mode = 'full')        #Autocorrelation
    R = R[-(len(signal)):]        #Keep autocorrelation values for positive values of i
    # Levinson-Durbin Recursion Algorithm
    E = np.zeros(p+1)        #Vector to store error values
    a = np.zeros((p+1,p+1))
    G = np.zeros(p+1)
    E[0] = R[0]        #Initial Condition
    for i in range(1, p+1):                  # 1 <= i <= p
        if i==1:
            k = R[1]/E[0]
            a[1][1] = k
            E[1] = (1-k**2)*E[0]
            a[1][0] = 1
            G[1] = np.sqrt(E[1])
        else:           #sum_{j=1}^{i-1} \alpha_j^{i-1}*r[i-j] calculation
            temp = 0
            for j in range(1, i):        # 1 <= j <= i-1
                temp += a[i-1][j] * R[i-j]
            k = (R[i] - temp)/E[i-1]
            a[i][i] = k
            for j in range(1, i):        # 1<=j<=i-1
                a[i][j] = a[i-1][j] - k * a[i-1][i-j]

            E[i] = (1 - k**2) * E[i-1]
            G[i] = np.sqrt(E[i])
            a[i][0] = 1
    return(E, G, a)

def plot_error_signal_energy(E,p=10):
    x = [i for i in range(p+1)]
    plt.figure()
    plt.plot(x,10*np.log10(E),marker="*")
    plt.xlabel("Number of Poles used in LP Analysis")
    plt.ylabel("Error Signal Energy (dB)")
    plt.title("Error Signal Energy vs Poles")
    plt.grid(color='0.9', linestyle='-')
    plt.autoscale(enable=True, axis='x', tight=True)
    plt.savefig("../plots/ErrorSignalEnergy_vs_numPoles.png",bbox_inches="tight")
```



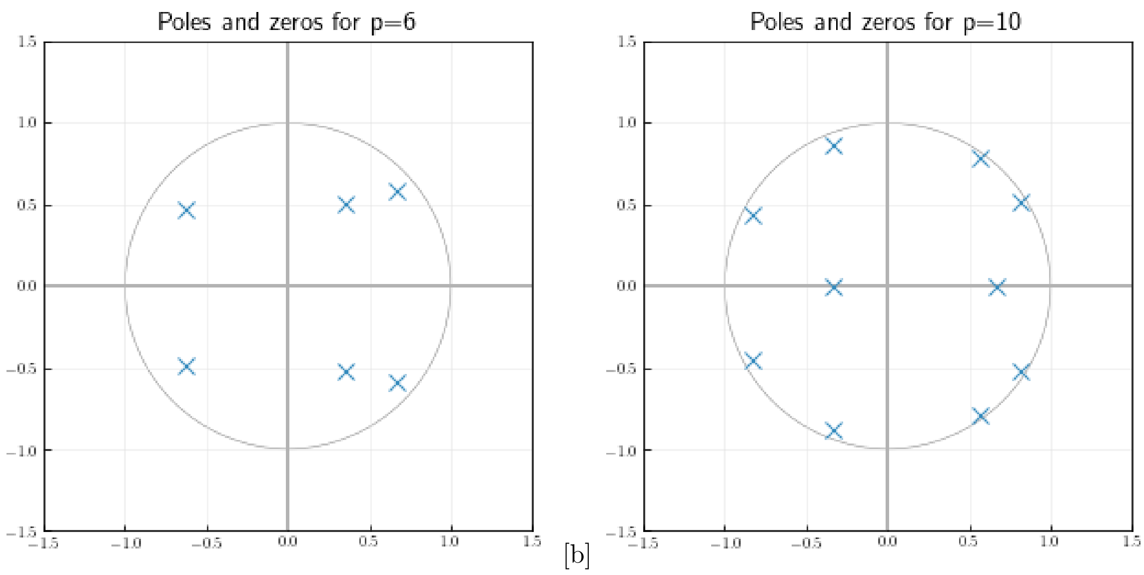
[a]

Figure 3: (a) Residual Energy signal (dB) vs p

4. **Show the pole-zero plots of the estimated all-pole filter for $p = 6, 10$; Comment.**

```
def zplane(ax, z, p, filename=None):
    """Plot the complex z-plane given zeros and poles.
    """

    # Add unit circle and zero axes
    unit_circle = patches.Circle((0,0), radius=1, fill=False,
                                 color='black', ls='solid', alpha=0.5)
    ax.add_patch(unit_circle)
    plt.axvline(0, color='0.7')
    plt.axhline(0, color='0.7')

    # Plot the poles and set marker properties
    poles = plt.plot(p.real, p.imag, 'x', markersize=9)

    # Plot the zeros and set marker properties
    zeros = plt.plot(z.real, z.imag,  'o', markersize=9,
             color='none',
             markeredgecolor=poles[0].get_color(), # same color as poles
             )

    # Scale axes to fit
    r = 1.5 * np.amax(np.concatenate((abs(z), abs(p), [1])))
    plt.axis('scaled')
    plt.axis([-r, r, -r, r])

    """
    If there are multiple poles or zeros at the same point, put a
    superscript next to them.
    TODO: can this be made to self-update when zoomed?
    """
    # Finding duplicates by same pixel coordinates (hacky for now):
    poles_xy = ax.transData.transform(np.vstack(poles[0].get_data()).T)
    zeros_xy = ax.transData.transform(np.vstack(zeros[0].get_data()).T)

    # dict keys should be ints for matching, but coords should be floats for
    # keeping location of text accurate while zooming

    # TODO make less hacky, reduce duplication of code
    d = defaultdict(int)
    coords = defaultdict(tuple)
    for xy in poles_xy:
        key = tuple(np.rint(xy).astype('int'))
        d[key] += 1
        coords[key] = xy
    for key, value in d.items():
        if value > 1:
            x, y = ax.transData.inverted().transform(coords[key])
            plt.text(x, y,
                        r' ${}^{' + str(value) + '}$',
                        fontsize=13,
                        )

    d = defaultdict(int)
    coords = defaultdict(tuple)
    for xy in zeros_xy:
        key = tuple(np.rint(xy).astype('int'))
        d[key] += 1
        coords[key] = xy
    for key, value in d.items():
        if value > 1:
            x, y = ax.transData.inverted().transform(coords[key])
            plt.text(x, y,
                        r' ${}^{' + str(value) + '}$',
                        fontsize=13,
                        )


def plot_poles_and_zeros(req,a,G):
    """Uses a helper function zplane to plot poles and zeros"""
    for p_ in req:
        poles = [a[p_][0],*(-a[p_][1:p_+1])]
        gain = G[p_]
        z,p,k = signal.tf2zpk(gain,poles)
```

```python

        fig = plt.figure()
        ax = fig.add_subplot(111)
        zplane(ax, z, p)
        plt.grid(True, color='0.9', linestyle='-', which='both', axis='both')
        plt.title('Poles and zeros for p='+str(p_))
        plt.savefig("../plots/PoleZeroPlot_p_"+str(p_)+".png",bbox_inches="tight")
        plt.clf()
```



[a] [b]

Figure 4: (a) Pole-Zero Plot for p=6 (b) Pole-Zero Plot for p=10

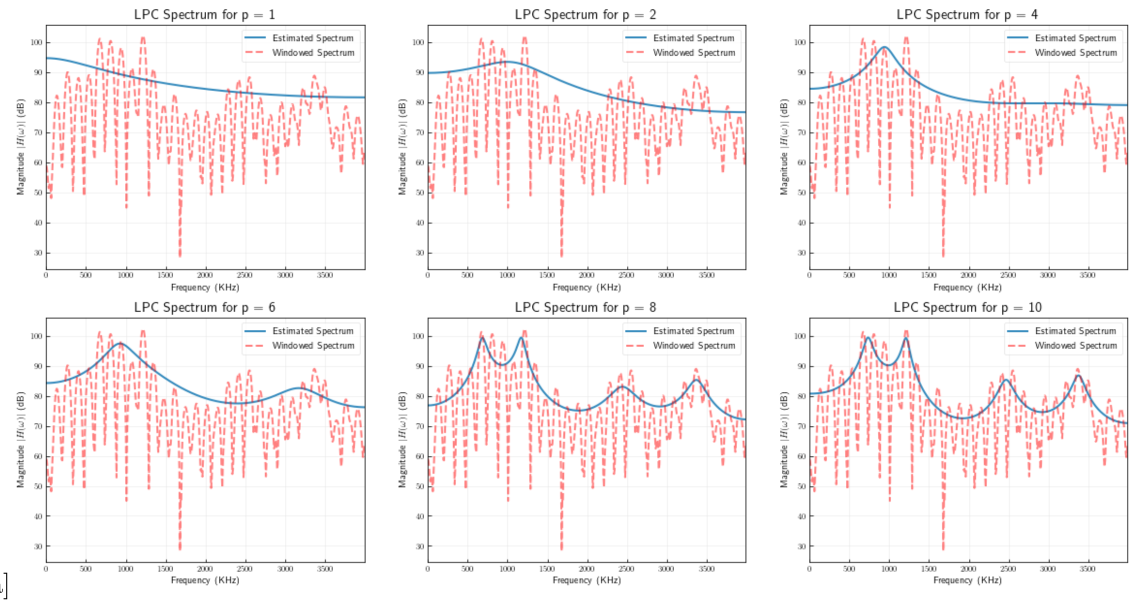
Discussion :

- From the relationship $z = re^{j\theta}$ where $r = e^{-\pi B T_s}$ and $\theta = 2\pi F T_s$ (B is the bandwidth of formant frequency F, T_s is the sampling frequency), the pole-zero plot can be a good indication of the numer of formant frequencies estimated (one formant frequency would result in two conjugate symmetric poles).
- For order $p = 6$, we see that the number of poles are 6, so it has determined three formants, whose phase and magnitude can be used to recalculate the formant frequencies and bandwidths of these formants. Here, the radius of the z-plane poles are less close to the unit-circle, and so these poles would have more bandwidth spread.
- For order $p = 10$, there are more poles estimated, as in LPAnalysis, greater the order p, the more are the poles and the more number of formants estimated. So, for 10^{th} order filter the poles are all nearly coinciding on the unit circle, pointing to the fact that the bandwidths corresponding the poles of the filter $\frac{G}{A(z)}$, are estimated quite precisely.

5. **Compute the gain and plot the LPC spectrum magnitude (i.e. the dB magnitude frequency response of the estimated all-pole filter) for each order "p". Comment on the characteristics of the spectral envelope estimates. Comment on their shapes with reference to the short-time magnitude spectrum computed in part 2.**

```python
def plot_LPC_Spectrum(a,G,Fs,h_w,p=[1,2,4,6,8,10]):
    """Plot the LPC Magnitude Spectrum
    :params a: Poles of the filters for order upto p=10
    :params G: Gains of the filters for orser upto p=10
    :params Fs: sampling frequency of the sound sample
    :params h_w: the magnitude spectrum of the windowed speech signal
    :params p: orders of the filter to be considered (default = [1,2,4,6,8]
    """
    n = len(p)
    plt.figure(figsize=(20,10))
    for i in range(n):
        poles = [a[p[i]][0],*(-a[p[i]][1:p[i]+1])]
        w,h = signal.freqz(G[p[i]],poles)
        w_ham,h_ham = signal.freqz(h_w)
        plt.suptitle("")
        plt.subplot(2,3,i+1)
        plt.plot((Fs*w/(2*np.pi)),20*np.log10(abs(h)),label="Estimated Spectrum")
        plt.plot((Fs*w_ham/(2*np.pi)),20*np.log10(abs(h_ham)),"r",linestyle='dashed',alpha=0.5,
                 label="Windowed Spectrum")
        plt.title("LPC Spectrum for p = {}".format(p[i]))
```

```python
        plt.grid(color='0.9', linestyle='-')
        plt.xlim(xmin=-5)
        plt.legend(loc="best")
        plt.xlabel("Frequency (KHz)")
        plt.ylabel(r"Magnitude $|H(\omega)|$ (dB)")
        plt.autoscale(enable=True, axis='x', tight=True)

    plt.savefig("../plots/LPCSpectrum.png",bbox_inches="tight")
```



[a]

Figure 5: (a) LPC Magnitude Spectrum for poles p=[1,2,4,6,8,10]

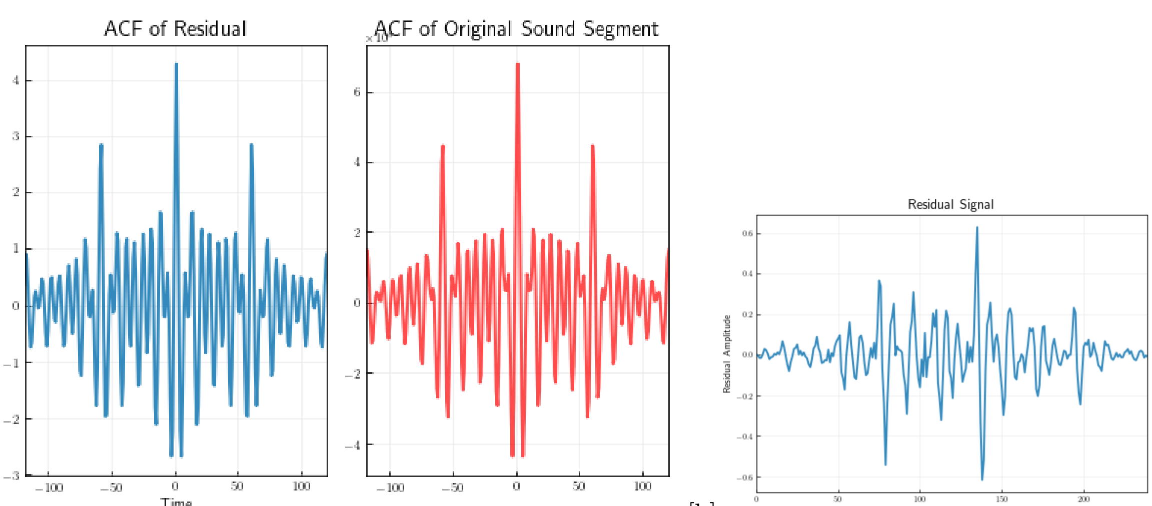
Discussions :

- The Gains recorded are :
  - Gain at order p = 2 is 17194.620917936038
  - Gain at order p = 4 is 15439.956256945918
  - Gain at order p = 6 is 14991.367557189498
  - Gain at order p = 8 is 11976.991688357673
  - Gain at order p = 10 is 11336.29152178067
- The plots above can be easily interpreted, so see that as the order p of the LP Coefficients increases the magnitude spectrum of the estimated filter gets closer and closer to the original magnitude spectrum of the windowed segment.
- For order $p \in \{1, \ldots, 4\}$, there is only one formant estimated, while for $p = 6$, we see two of the formant frequencies are estimated. For $p \in \{8, \ldots, 10\}$ upto four formants are estimated more or less accurately.
- As the order increase we see that gain, $G = \sqrt{ACF[0] - \sum_{k=1}^{p} a_k ACF[k]}$ decreases as was evident from theory as well as the Error Signal Energy vs p plot.

6. **Based on the 10^{th}-order LP coefficients, carry out the inverse filtering of the /a/ vowel segment to obtain the residual error signal. Can you measure the pitch period of the voiced sound from the residual waveform? Use the acf to detect the pitch. Compare the acf plots of the original speech and residual signals.**

```python
def autocorrelate(gain,poles,segment_signal,duration=30,verbose=True):
    """Calculate F0 from the autocorrelation of windowed signal using inverse filtering
    :param gain: Gain of the 10th order LP Coefficient
    :param poles: the 10th order LP Coefficients
    :param segment_signal: windowed signal
    :param duration: windowed signal duration (Here 30ms)
    :param verbose: if True plot the residual signal waveform
    :output f0: returns the calculated fundamental frequency of the signal
    """
    window_duration = int((duration/1000)*Fs)

    inverse_filter = np.zeros_like(segment_signal)
    for i in range(segment_signal.shape[0]):
        inverse_filter[i] = segment_signal[i]
        for j in range(len(poles)):
            if (i-j)>=0:
```

```
                inverse_filter[i] -= poles[j]*segment_signal[i-j]
        inverse_filter[i] /= gain

    if verbose:
        plt.figure()
        plt.plot(inverse_filter)
        plt.title("Residual Signal")
        plt.grid(color='0.9', linestyle='-')
        plt.xlabel("Time")
        plt.ylabel("Residual Amplitude")
        plt.tight_layout()
        plt.autoscale(enable=True, axis='x', tight=True)
        plt.savefig("../plots/ResidualPlot.png",bbox_inches="tight")

    autocorrelate = np.correlate(inverse_filter,inverse_filter,mode="same")
    signal_autocorrelate = np.correlate(segment_signal,segment_signal,mode="same")

    maxima = np.argmax(autocorrelate)
    second_maxima = np.argmax(autocorrelate[autocorrelate<0.7*np.max(autocorrelate)])

    print("Index of maxima",maxima)
    print("Index of second maxima",second_maxima)

    F0 = (Fs/(maxima - second_maxima))
    print("F0 detected from ACF : ",F0,"Hz")

    plt.figure()
    plt.subplot(121)
    plt.title("ACF of Residual")
    plt.plot(autocorrelate)
    plt.grid(color='0.9', linestyle='-')
    plt.xlabel("Time")
    plt.autoscale(enable=True, axis='x', tight=True)
    plt.tight_layout()

    plt.subplot(122)
    plt.title("ACF of Original Sound Segment")
    plt.plot(signal_autocorrelate,"r",alpha=0.7)
    plt.grid(color='0.9', linestyle='-')
    plt.autoscale(enable=True, axis='x', tight=True)
    plt.tight_layout()
    plt.savefig("../plots/ComparisonBetweenAutocorrelation.png",bbox_inches="tight")


    return F0
```



[a] [b]

Figure 6: (a) AutoCorrelation Plot for Windowed Signal and Residual Signal (b) Residual Signal Plot

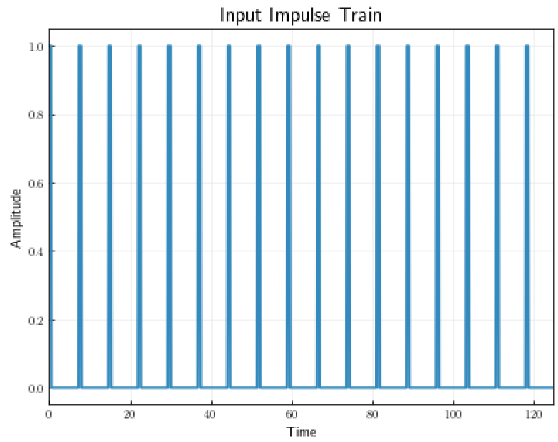
Discussion :

- Yes, the fundamental frequency of the speech segment can be calculated from the Auto-Correlation of the Residual signal, obtained by inverse filtering the segmented_signal using the LPAnalysis coefficients of order $p = 10$, by calculating the distance between the peaks and calculating the frequency from time-scale.

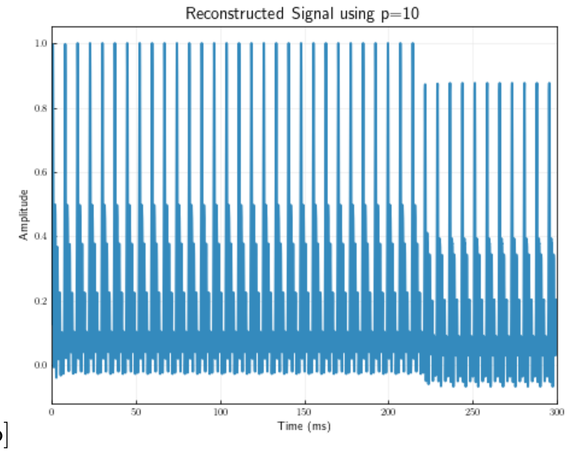
- The Auto-Correlation plot of the Residual signal resembles that of the Original speech segment, only that the residual signal has less magnitude than that of the original signal.
- So, the residual signal's ACF can be used to calculate the fundamental frequency F_0 of the input speech segment, by first calculating the time spacing between the two peaks (in my calculation I used a threshold of 0.7, ie. the second maximum peak amplitude should be less than 0.7*maximum peak amplitude), so that only the necessary peaks are captured.
- The fundamental frequency F_0 of the speech segment was calculated as 135.59 Hz.

7. **(Optional for bonus marks) LP re-synthesis: We analysed a natural speech sound /a/ above. Using a suitable set of parameter estimates as obtained there, we wish to reconstruct the sound. That is, use the best estimated LP filter with an ideal impulse train of the estimated pitch period as source excitation. Carry out de-emphasis on the output waveform. Set the duration of the synthesized sound to be 300 ms at 8 kHz sampling frequency and view the waveform as well as listen to your created sound.**
Comment on the similarity with the original sound. What would be a good application for this analysis-and-synthesis system, and how exactly does it help?

```python
def reconstruct(gain,poles,F0,Fs,p=10,duration=300):
    """Reconstruc the original speech signal from the LP coefficients, gain and F0 estimation
    using impulse trains as the input to the reconstruction filter
    :param gain: Gain of the estimated LP coefficients of order 10
    :param poles: LP coefficient of order 10
    :param F0: estimated fundamental frequency of the speech segment
    :param Fs: sampling frequency of the speech segment
    :param p: order or LP Coeffs (default=10)
    :param duration: duration of the output signal
    :output output_signal: the reconstructed signal
    """
    period = F0/1000
    total = int((duration/1000)*Fs)
    t = np.linspace(0,duration,total)
    impulse_train = (signal.square(2 * np.pi *period * t,duty=0.08)+1)/2
    plt.figure()
    plt.title("Input Impulse Train")
    plt.xlabel("Time")
    plt.ylabel("Amplitude")
    plt.plot(t[:1000], impulse_train[:1000])
    plt.grid(color='0.9', linestyle='-')
    plt.autoscale(enable=True, axis='x', tight=True)
    plt.savefig("../plots/ReconstructionImpulse(segment).png",bbox_inches="tight")

    output_signal = np.zeros_like(impulse_train)
    for i in range(len(impulse_train)):
        output_signal[i] = gain*impulse_train[i]
        for j in range(len(poles)):
            if (i-j)>=0:
                output_signal[i] += poles[j]*output_signal[i-j]

    #de-emphasize
    de_emphasis = np.zeros_like(output_signal)
    de_emphasis[0] = output_signal[0]
    for i in range(1,len(de_emphasis)):
            de_emphasis[i] = output_signal[i] + de_emphasis[i-1]*alpha
    de_emphasis = de_emphasis/np.max(de_emphasis)


    plt.figure()
    plt.plot(t,de_emphasis)
    plt.grid(color='0.9', linestyle='-')
    plt.autoscale(enable=True, axis='x', tight=True)
    plt.tight_layout()
    plt.xlabel("Time (ms)")
    plt.ylabel("Amplitude")
    plt.title("Reconstructed Signal using p="+str(p))
    plt.savefig("../plots/ReconstructedSignal_p"+str(p)+".png",bbox_inches="tight")

    wav.write("../wavfiles/Reconstructed_Wave_p"+str(p)+".wav",8000,de_emphasis)
    return output_signal
```



[a]



[b]

Figure 7: (a) Impulse Input to the Reconstruction Filter (b) Reconstructed Speech Signal

Discussion :

- De-emphasis is used to remove the effect of pre-emphasis filter, using the same parameter α as the free parameter.
- The reconstructed signal from G, a, F_0 calculated by passing a impulse train of frequency F_0 through a filter with parameters (G, a) of the p^{th} order LP Coefficients, is more or less a replica of the original signal.
- The reconstructed *.wav* file are stored in directory *wavfiles*.
- A good application of such a technique of reconstructio, is in telecommunications where the Bit Rate is a valuable resource. As the number of parameters sent are quite less than the number of parameters to needed if the complete signal was to be sent, it helps in Bandwidth saving, and increased Bit Rate.

The Main Function and the helper function to Read the Wavfile.

```
def read_file(filename="../wavfiles/aa.wav",verbose=False):
	"""Reads the wavfile and returns F0 and the signal
	:param filename: path to the wavfile
	:param verbose: if True plots teh original speech signal
	:output sound: the speech signal
	:output Fs: the sampling frequency of the speech signal
	"""
	input_wav = wav.read(filename)
	sound = input_wav[1]
	Fs = input_wav[0]
	if verbose:
		plt.figure()
		plt.title("Original Sound Waveform")
		plt.plot(sound)
		plt.grid(color='0.9', linestyle='-')
		plt.tight_layout()
		plt.xlim(xmin=0)
		plt.ylabel("Amplitude")
		plt.xlabel("Time")
		plt.autoscale(enable=True, axis='x', tight=True)
		plt.savefig("../plots/OriginalSound.png",bbox_inches="tight")

	return sound,Fs

if __name__ == '__main__':
	filename="../wavfiles/aa.wav"
	sound,Fs = read_file(filename,verbose=True)
	# Question 1 : PRE-EMPHASIS USING ALPHA=0.95
	pre_emphasis = pre_emphasize(sound,verbose=True)
	hamming_output = hamming_window(pre_emphasis)
	E,G,a = LPanalysis(hamming_output,10)
	plot_error_signal_energy(E)
	plot_poles_and_zeros([6,10],a,G)
	plot_LPC_Spectrum(a,G,Fs,hamming_output)
	F0 = autocorrelate(G[10],a[10],hamming_output)
	reconstructed_signal = reconstruct(G[10],a[10],F0,Fs)
```