ECE 438 - Laboratory 9b
Speech Processing (Week 2)

Last Updated on April 5, 2022

In [1]:
```
import sys
import json
import numpy as np
import scipy as scp
import scipy.signal
import matplotlib.pyplot as plt
import soundfile as sf
import IPython.display as ipd
```

In [2]:
```
# make sure the plot is displayed in this notebook
%matplotlib inline
# specify the size of the plot
plt.rcParams['figure.figsize'] = (16, 6)

# for auto-reloading extenrnal modules
%load_ext autoreload
%autoreload 2
```

1. Introduction

This is the second part of a two week experiment. During the first week we discussed basic properties of speech signals, and performed some simple analyses in the time and frequency domain.

This week, we will introduce a system model for speech production. We will cover some background on **linear predictive coding**, and the final exercise will bring all the prior material together in a speech coding exercise.

1.1. A Speech Model

From a signal processing standpoint, it is very useful to think of speech production in terms of a model, as in Figure 1. The model shown is the simplest of its kind, but it includes all the principal components. The excitations for voiced and unvoiced speech are represented by an impulse train and white noise generator, respectively. The pitch of voiced speech is controlled by the spacing between impulses, T_p, and the amplitude (volume) of the excitation is controlled by the gain factor G.

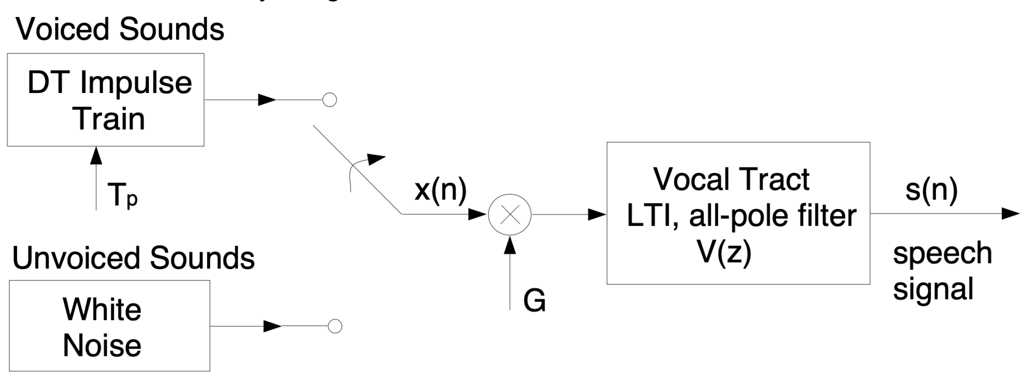

Figure 1: Discrete-Time Speech Production Model

As the acoustical excitation travels from its source (vocal cords, or a constriction), the shape of the vocal tract alters the spectral content of the signal. The most prominent effect is the formation of resonances, which intensifies the signal energy at certain frequencies (called formants). As we learned in the Digital Filter Design lab, the amplification of certain frequencies may be achieved with a linear filter by an appropriate placement of poles in the transfer function. This is why the filter in our speech model utilizes an all-pole LTI filter. A more accurate model might include a few zeros in the transfer function, but if the order of the filter is chosen appropriately, the all-pole model is sufficient. The primary reason for using the all-pole model is the distinct computational advantage in calculating the filter coefficients, as will be discussed shortly.

Recall that the transfer function of an all-pole filter has the form

$$V(z) = \frac{1}{1 - \sum_{k=1}^{P} a_k z^{-k}} \tag{1}$$

where P is the order of the filter. This is an IIR filter that may be implemented with a recursive difference equation. With the input $G \cdot x[n]$, the speech signal $s[n]$ may be written as

$$s[n] = \sum_{k=1}^{P} a_k s[n-k] + G \cdot x[n] \tag{2}$$

Keep in mind that the filter coefficients will change continuously as the shape of the vocal tract changes, but speech segments of an appropriately small length may be approximated by a time-invariant model.

This speech model is used in a variety of speech processing applications, including methods of speech recognition, speech coding for transmission, and speech synthesis. Each of these applications of the model involves dividing the speech signal into short segments, over which the filter coefficients are almost constant. For example, in speech transmission the bit rate can be significantly reduced by dividing the signal up into segments, computing and sending the model parameters for each segment (filter coefficients, gain, etc.), and re-synthesizing the signal at the receiving end, using a model similar to Figure 1. Most telephone systems use some form of this approach. Another example is speech recognition. Most recognition methods involve comparisons between short segments of the speech signals, and the filter coefficients of this model are often used in computing the “difference” between segments.

Exercise 1.2: Synthesis of Voiced Speech

1. Use the following code to load three sets of filter coefficients: `A1` , `A2` , and `A3` , respectively, for the vocal tract model in equations (1) and (2). Each vector contains coefficients $\{a_1, a_2, \dots, a_{15}\}$ for an all-pole filter of order 15.

In [3]:
```
coeff = json.load(open("coeff.json", 'r'))
A1 = np.array(coeff["A1"])
A2 = np.array(coeff["A2"])
A3 = np.array(coeff["A3"])
```

2. Complete the function below to create a length `N` excitation for voiced speech, with a pitch period of `Np` samples. The output vector `x` should contain a discrete-time impulse train with period `Np` (e.g., $[1, 0, 0, \cdots, 0, 1, 0, 0, \cdots]$).

In [4]:
```
def exciteV(N, Np):
    """
    Parameters
    ---
    N: the length of excitation
    Np: pitch period in number of samples

    Returns
    ---
    x: a discrete-time impulse train with period Np
    """

    x = None
    return x
```

3. Assuming a sampling frequency of 8 kHz (0.125 ms/sample), create a 40 millisecond-long excitation with a pitch period of 8 ms, and filter it using equation (2) for each set of coefficients.

You may use the command:

```
s = scp.signal.lfilter(np.array([1]), np.insert(-A, 0, 1), x)
```

where `A` is the row vector of filter coefficients.

`scp.signal.lfilter()` (https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.lfilter.html) filter data along one-dimension with an IIR or FIR filter.

`np.insert(arr, 0, 1)` (https://numpy.org/doc/stable/reference/generated/numpy.insert.html) insert the value 1 at the beginning of `arr` .

In [5]:
```
# insert your code here
```

4. Plot each of the three filtered signals.

In [6]:
```
# insert your code here
```

We will now compute the frequency response of each of these filters. The frequency response may be obtained by evaluating Eq. (1) at points along $z = e^{j\omega}$.

5. Use the following command to obtain the frequency response of these filters.

```
w, h = scp.signal.freqz(np.array([1]), np.insert(-A, 0, 1), 512)
```

where `A` is the vector of coefficients.

In [7]:
```
# insert your code here
```

6. Plot the magnitude of each response versus frequency in Hertz. Make sure to label the frequency axis in units of Hertz.

In [8]:
```
# insert your code here
```

7. The location of the peaks in the spectrum correspond to the formant frequencies. For each vowel signal, estimate the first three formants (in Hz) and list them.

insert your answer here

8. Generate the three signals again, but use an excitation which is 1-2 seconds long. Listen to the filtered signals.

```
In [9]: # insert your code here
```

9. Can you hear qualitative differences in the signals generated in Q8? Can you identify the vowel sounds?

insert your answer here

2. Linear Predictive Coding

The filter coefficients which were provided in the previous section were determined using a technique called **linear predictive coding** (LPC). LPC is a fundamental component of many speech processing applications, including compression, recognition, and synthesis.

In the following discussion of LPC, we will view the speech signal as a discrete-time random process.

2.1. Forward Linear Prediction

Suppose we have a discrete-time random process $\{\ldots, S_{-1}, S_0, S_1, S_2, \ldots\}$ whose elements have some degree of correlation. The goal of **forward linear** prediction is to predict the sample S_n using a linear combination of the previous P samples.

$$\hat{S}_n = \sum_{k=1}^{P} a_k S_{n-k} \tag{3}$$

P is called the *order* of the predictor. We may represent the error of predicting S_n by a random sequence e_n.

$$e_n = S_n - \hat{S}_n \tag{4}$$
$$e_n = S_n - \sum_{k=1}^{P} a_k S_{n-k} \tag{5}$$

An optimal set of prediction coefficients a_k for (5) may be determined by minimizing the mean-square error $\mathbb{E}[e_n^2]$. Note that since the error is generally a function of n, the prediction coefficients will also be functions of n. To simplify notation, let us first define the following column vectors.

$$\mathbf{a} = \begin{bmatrix} a_1 & a_2 & \cdots & a_P \end{bmatrix}^T$$
$$\mathbf{S}_{n,P} = \begin{bmatrix} S_{n-1} & S_{n-2} & \cdots & S_{n-P} \end{bmatrix}^T$$

Then,

$$\begin{aligned}
\mathbb{E}[e_n^2] &= \mathbb{E}\left[\left(S_n - \sum_{k=1}^{P} a_k S_{n-k}\right)^2\right] && (6)\\
&= \mathbb{E}\left[\left(S_n - \mathbf{a}^T\mathbf{S}_{n,P}\right)^2\right] && (7)\\
&= \mathbb{E}\left[S_n^2 - 2S_n\mathbf{a}^T\mathbf{S}_{n,P} + \mathbf{a}^T\mathbf{S}_{n,P}\mathbf{a}^T\mathbf{S}_{n,P}\right] && (8)\\
&= \mathbb{E}\left[S_n^2\right] - 2\mathbf{a}^T\,\mathbb{E}\left[S_n\mathbf{S}_{n,P}\right] + \mathbf{a}^T\,\mathbb{E}\left[\mathbf{S}_{n,P}\mathbf{S}_{n,P}^T\right]\mathbf{a} && (9)
\end{aligned}$$

The second and third terms of equation (9) may be written in terms of the autocorrelation sequence $r_{SS}[k,l]$.

$$\mathbb{E}\left[S_n\mathbf{S}_{n,P}\right] = \begin{bmatrix} \mathbb{E}[S_nS_{n-1}] \\ \mathbb{E}[S_nS_{n-2}] \\ \vdots \\ \mathbb{E}[S_nS_{n-P}] \end{bmatrix} = \begin{bmatrix} r_{SS}[n,n-1] \\ r_{SS}[n,n-2] \\ \vdots \\ r_{SS}[n,n-P] \end{bmatrix} \equiv \mathbf{r}_S \tag{10}$$

$$\begin{aligned}
\mathbb{E}\left[\mathbf{S}_{n,P}\mathbf{S}_{n,P}^T\right] &= \mathbb{E}\begin{bmatrix} S_{n-1}S_{n-1} & S_{n-1}S_{n-2} & \cdots & S_{n-1}S_{n-P} \\ S_{n-2}S_{n-1} & S_{n-2}S_{n-2} & \cdots & S_{n-2}S_{n-P} \\ \vdots & \vdots & \ddots & \vdots \\ S_{n-P}S_{n-1} & S_{n-P}S_{n-2} & \cdots & S_{n-P}S_{n-P} \end{bmatrix} \\
&= \begin{bmatrix} r_{SS}[n-1,n-1] & r_{SS}[n-1,n-2] & \cdots & r_{SS}[n-1,n-P] \\ r_{SS}[n-2,n-1] & r_{SS}[n-2,n-2] & \cdots & r_{SS}[n-2,n-P] \\ \vdots & \vdots & \ddots & \vdots \\ r_{SS}[n-P,n-1] & r_{SS}[n-P,n-2] & \cdots & r_{SS}[n-P,n-P] \end{bmatrix} \equiv \mathbf{R}_S && (11)
\end{aligned}$$

Substituting into equation (9), the mean-square error may be written as

$$\mathbb{E}\left[e_n^2\right] = \mathbb{E}\left[S_n^2\right] - 2\mathbf{a}^T\mathbf{r}_S + \mathbf{a}^T\mathbf{R}_S\mathbf{a} \tag{12}$$

Note that while $\mathbf{a}$ and $\mathbf{r}_S$ are vectors, and $\mathbf{R}_S$ is a matrix, the expression in (12) is still a scalar quantity.

To find the optimal a_k coefficients, which we will call $\hat{\mathbf{a}}$, we differentiate equation (12) with respect to the vector $\mathbf{a}$ (compute the gradient), and set it equal to the zero vector.

$$\nabla_{\mathbf{a}}\mathbb{E}\left[e_n^2\right] = -2\mathbf{r}_S + 2\mathbf{R}_S\hat{\mathbf{a}} \equiv 0 \tag{13}$$

Solving,

$$\mathbf{R}_S\hat{\mathbf{a}} = \mathbf{r}_S \tag{14}$$

The vector equation in (14) is a system of P scalar linear equations, which may be solved by inverting the matrix $\mathbf{R}_S$.

Note from (10) and (11) that $\mathbf{r}_S$ and $\mathbf{R}_S$ are generally functions of n. However, if S_n is wide-sense stationary, the autocorrelation function is only dependent on the difference between the two indices, $r_{SS}[k,l] = r_{SS}[|k-l|]$. Then $\mathbf{R}_S$ and $\mathbf{r}_S$ are no longer dependent on n, and may be written as follows.

$$\mathbf{r}_S = \begin{bmatrix} r_{SS}[1] \\ r_{SS}[2] \\ \vdots \\ r_{SS}[P] \end{bmatrix} \tag{15}$$

$$\mathbf{R}_S = \begin{bmatrix} r_{SS}[0] & r_{SS}[1] & \cdots & r_{SS}[P-1] \\ r_{SS}[1] & r_{SS}[0] & \cdots & r_{SS}[P-2] \\ r_{SS}[2] & r_{SS}[1] & \cdots & r_{SS}[P-3] \\ \vdots & \vdots & \ddots & \vdots \\ r_{SS}[P-1] & r_{SS}[P-2] & \cdots & r_{SS}[0] \end{bmatrix} \tag{16}$$

Therefore, if S_n is wide-sense stationary, the optimal a_k coefficients do not depend on n. In this case, it is also important to note that $\mathbf{R}_S$ is a Toeplitz (constant along diagonals) and symmetric matrix, which allows (14) to be solved efficiently using the Levinson-Durbin algorithm (see [2]). This property is essential for many real-time applications of linear prediction.

2.2. Linear Predictive Coding of Speech

An important question has yet to be addressed. The solution in (14) to the linear prediction problem depends entirely on the autocorrelation sequence. How do we estimate the autocorrelation of a speech signal? Recall that the applications to which we are applying LPC involve dividing the speech signal up into short segments and computing the filter coefficients for each segment. Therefore we need to consider the problem of estimating the autocorrelation for a short segment of the signal. In LPC, the following biased autocorrelation estimate is often used.

$$\hat{r}_{SS}[m] = \frac{1}{N} \sum_{n=0}^{N-m-1} s[n]s[n+m], \quad 0 \le m \le P \tag{17}$$

Here we are assuming we have a length N segment which starts at $n = 0$. Note that this is the single-parameter form of the autocorrelation sequence, so that the forms in (15) and (16) may be used for $\mathbf{r}_S$ and $\mathbf{R}_S$.

Exercise 2.3: LPC

1. Complete the function below to compute the order-P LPC coefficients for the vector x, using the autocorrelation method. Consider the input vector `x` as a speech segment, in other words do not divide it up into pieces. The output vector `coef` should be a vector containing the `P` coefficients $\{\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_P\}$. In your function you should do the following:

- Compute the biased autocorrelation estimate of equation (17) for the lag values $0 \le m \le P$.
- Form the $\mathbf{r}_S$ and $\mathbf{R}_S$ vectors as in (15) and (16). Hint: Use the `scp.linalg.toeplitz()` (https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.toeplitz.html) function to form $\mathbf{R}_S$.
- Solve the matrix equation (14) for $\hat{\mathbf{a}}$, which can be solved by inverting the matrix $\mathbf{R}_S$ using `np.linalg.inv()` (https://numpy.org/doc/stable/reference/generated/numpy.linalg.inv.html).

In [10]:
```
def mylpc(x, P):
    """
    Parameters
    ---
    x: the speech segment
    P: the order

    Returns
    ---
    coef: the order-P LPC coefficients for x
    """

    coef = None
    return coef
```

2. Load `test.json` using the code below. This file contains two vectors: a signal `x` and its order-15 LPC coefficients `a`. Use your function to compute the order-15 LPC coefficients of `x`, and compare the result to the vector `a`.

Note: To check if two vectors are close, use the command below. This function will raise error if the two arrays are not equal up to desired tolerance. In the case below, both the absolute tolerance and the relative tolerance are 10^{-10}.

```
np.testing.assert_allclose(np.array(coef), np.array(a), atol=1e-10, rtol=1e-10)
```

In [11]:
```
test = json.load(open("test.json", 'r'))
x = np.array(test['x'])
a = np.array(test['a'])
```

In [12]:
```
# insert your code here
```

3. Speech Coding and Synthesis

One very effective application of LPC is the compression of speech signals. For example, an LPC vocoder (voice-coder) is a system used in many telephone systems to reduce the bit rate for the transmission of speech. This system has two overall components: an analysis section which computes signal parameters (gain, filter coefficients, etc.), and a synthesis section which reconstructs the speech signal after transmission.

Since we have introduced the speech model in section 1.1, and the estimation of LPC coefficients in section 2, we now have all the tools necessary to implement a simple vocoder. First, in the analysis section, the original speech signal will be split into short time frames. For each frame, we will compute the signal energy, the LPC coefficients, and determine whether the segment is voiced or unvoiced.

Exercise 3.1

1. Load and play the audio file `phrase.au`. This speeech signal is sampled at a rate of 8000 Hz.

In [13]:
```
# insert your code here
```

2. Divide the original speech signal into 30ms non-overlapping frames. Place the frames into L consecutive rows of a matrix S (use `np.reshape()` (https://numpy.org/doc/stable/reference/generated/numpy.reshape.html)). If the samples at the tail end of the signal do not fill an entire column, you may disregard these samples.

Hint: Say the original signal is of length `N`, and only the first `M` (is divisible by 10 and is as large as possible) samples are needed, we can calculate `M` easily by

```
M = N // 10 * 10
```

where `//` is the floor division operator in Python.

In [14]:
```
# insert your code here
```

3. For each frame of the original word (i.e., each row of `S`), do the following:

- Compute the energy of each frame of the original word, and place these values in a length L vector called `energy`.
- Determine whether each frame is voiced or unvoiced. Use your zero cross function from the first week to compute the number of zero-crossings in each frame. For length N segments with less than $\frac{N}{2}$ zero-crossings, classify the segment as voiced, otherwise unvoiced. Save the results in a vector `VU` which takes the value of `1` for voiced and `0` for unvoiced.
- Use your `mylpc(x, P)` function to compute order-15 LPC coefficients for each frame. Place each set of coefficients into a column of a $L \times 15$ matrix `A`.

In [15]:
```
# insert your code here
```

4. To see the reduction in data, add up the total number of bytes Python uses to store the encoded speech in the arrays `A`, `VU`, and `energy` (use the `sys.getsizeof()` function). Compute the compression ratio by dividing this by the number of bytes Python uses to store the original speech signal. Note that the compression ratio can be further improved by using a technique called vector quantization on the LPC coefficients, and also by using fewer bits to represent the gain and voiced/unvoiced indicator.

In [16]:
```
# insert your code here
```

5. Now the computed parameters will be used to re-synthesize the phrase using the model in Figure 1. Similar to your `exciteV()` function from Section 1.2, complete the function below that returns a length N excitation for unvoiced speech (generate a `np.random.normal(0, 1)` sequence).

In [17]:
```
def exciteUV(N):
    """
    Parameters
    ---
    N: the length of excitation

    Returns
    ---
    x: the excitation of length N
    """

    x = None
    return x
```

6. Initialize an empty NumPy array `output`. Then, for each encoded frame, do the following:

- **Check if current frame is voiced or unvoiced.**
- **Generate the frame of speech by using the appropriate excitation into the filter specified by the LPC coefficients (you did this in Section 1.2). For voiced speech, use a pitch period of 7.5 ms. Make sure your synthesized segment is the same length as the original frame.**
- **Scale the amplitude of the segment so that the synthesized segment has the same energy as the original.**
- **Append `frame` to the end of the `output` array by using `output = np.append(output, frame)`.**

In [18]:
```
# insert your code here
```

7. Plot both the original and synthesizsed words.

In [19]:
```
# insert your code here
```

8. Listen to the original and synthesized phrase. Comment on the quality of your synthesized signal. How might the quality be improved?

In [20]:
```
# insert your code here
```

4. References

[1] J. R. Deller, Jr., J. G. Proakis, J. H. Hansen, Discrete-Time Processing of Speech Signals, Macmillan, New York, 1993.
[2] J. G. Proakis and D. G. Manolakis, Digital Signal Processing, 3rd edition, Prentice-Hall, Englewood Cliffs, New Jersey, 1996.