University of Washington ECE Department

EE 242 Lab 1 – Time & Amplitude Operations on Signals
Background Material

1.0 Introduction

The goals of Lab 1 are to develop an understanding of discrete representations of signals, learn how to read, write and play audio signals, and explore amplitude and time transformations of signals. This document will provide background information and useful tips for the different exercises in Lab 1.

This lab will make use of the numpy package to represent signals as arrays and matplotlib.pyplot for plotting signals, as well as Python concepts such as functions and objects. Students with previous Python experience should be familiar with these concepts. Students taking EE241 concurrently with EE242 will be learning about numpy concurrently. For purposes of Lab 1, the numpy functions required will be provided for you. Time samples are needed for creating signals and for plotting, as will be discussed further below.

2.0 Computer Representations of Signals

In class, we work with real-valued signals $x(t)$ where the independent variable (time) is also real-valued, i.e. $x(t) \in \Re$ and $t \in \Re$ or $-\infty < t < \infty$. A signal represented on a digital computer has to have discrete time samples and be finite in length. This means we can only define a signal over a finite range like $-2 \leq t \leq 10$, and we can only keep time samples at discrete locations. We can only approximate continuous-time signals and systems in software.

The simplest approach is to use time samples taken at fixed intervals, which is referred to as the **sampling period**. Taking samples at fixed intervals over a finite time period means that our signals can be represented as arrays. It is important to keep track of the sampling period in order to relate the digital signal to the real world. Often it is more useful to keep track of the **sampling frequency**, which is inversely related to the sampling period. In Hertz (Hz), the sampling frequency is $f_s=1/T_s$ where T_s is the sampling period in seconds. A sampling frequency of 100 Hz means that we are taking 100 samples per second, which corresponds to a sampling period of $T=1/100=.01$ second (or 10 ms). As you might expect, if samples are not spaced closely enough, then the approximation will not be good. There are theoretical results saying how fast we need to sample, but we'll learn that towards the end of the quarter. In the meantime, you need to be careful to use the specified sampling frequency (or sampling period).

A signal is digitized in amplitude as well as time, characterized by bytes per sample. Different audio formats will have different amplitude digitization methods, and there may be multiple channels. This information is handled by the file I/O function.

If you want to create a digital signal, you first create a vector with the sequence of times in the specified range, and then implement the time function. The *numpy* **arange** function is useful for defining a time samples vector

t = numpy.arange(start, stop, ts) # ts = sampling period (step size)

The stop value is exclusive, so you need the stop time to be greater than the actual end time you want. Time samples are needed for creating signals and for plotting. In the example below, we use *numpy* (imported as np) to specify a sequence of times which are then used to create the signal x(t) = t/2 over the range $0 \le t \le 4$ with a sampling period of T=0.2.

```
# x(t)=t/2  over the range [0,4] with a sampling period of T=0.2

t=np.arange(0,4.1,.2)       # time samples vector
x=t/2                       # signal vector
print('\n t = ',t)
print('\n x = ',x)
```

To plot the sound samples, we need the time samples vector to map each sample to time. Here's an example where we create time vector for an audio file that we read in and stored as "data1".

```
timeArray = np.arange(0, points, 1)
timeArray = timeArray / fs * 1000    # Scaling to time in milliseconds

data1 = data[:, 0]    # Use only one channel, zero out the other
plt.plot(timeArray, data1, color='k')
plt.ylabel('Amplitude')
plt.xlabel('Time (ms)')
```

The elements in a digital signal correspond to times $t=n*Ts$ in the continuous-time signal, for integer n and sampling period *Ts*. If we want to find the digital time *n* that corresponds to a particular time *t*, you would use **n=int(t/Ts)** or **n=int(t*fs)**, where the **int** function rounds to the nearest integer. Since we can't represent the whole continuous time signal, you can't get a result for arbitrary times *t*.

3.0 Reading, Writing and Playing Audio Files

There are multiple packages available for working with audio files. We will use both *simpleaudio* and *scipy.io*. Whatever package you use, there are two important issues to be aware of.

First, as noted above, the digital file must always be associated with a sampling frequency fs=1/Ts. This means that when you read an audio file, you will get fs with it. Conversely, you need to specify fs when you save a file or play the audio. If you play the file with a different fs than it was saved with, you will change the sound of the signal. (We'll try this in the lab.)

Second, audio files may be mono (1 channel) or stereo (2 channels). If the file is mono, then the audio data you read in will come in the form of a vector or 1-dimensional array. If it is stereo, then the data will have a pair of such vectors (one for each speaker), which together form a 2-dimensional array.

It is important to keep track of this information. For example, in creating new sounds, we might want to add, multiply or concatenate different sounds. In order to get the desired result, you need all component sounds to have the same sample rate and the same number of channels.

You can read and write .wav files using the **wavfile** module from **scipy.io**, and you can read and play .wav files using the **simpleaudio** module. To do all three, we'll use both. You will need to install simpleaudio. In your terminal or Anaconda prompt, type `pip install simpleaudio` (do not put a period at the end). Documentation on the *simpleaudio* package can be found at: https://simpleaudio.readthedocs.io/en/latest/. You might have to install few dependencies to install **simpleaudio** successfully. If you are on a Windows machine, you will need to install Microsoft Visual C++ 14.0 or greater (https://visualstudio.microsoft.com/visual-cpp-build-tools/) and restart your PC to run pip install simpleaudio successfully. If you're on a Linux machine, you can install the dependencies using sudo apt-get install -y python3-dev libasound2-dev.

Once installation is complete, you may exit your terminal or Anaconda prompt. With **simpleaudio,** the sampling frequency, channel and other information is bundled together with the data in a special WaveObject. With scipy.io, you have to keep track of the sampling frequency separately.

Let's start with **simpleaudio.** The easy way to read a file creates an object that has the data, the sampling frequency, number of channels, and bytes per sample as attributes. (The signal is digitized in amplitude as well as time, characterized by bytes per sample.) Assuming that you used `import simpleaudio as sa`, you can extract this information and play the file as follows:

```
# Test the audio read and play functions
wav_obj = sa.WaveObject.from_wave_file('train32.wav')
fs = wav_obj.sample_rate
channels = wav_obj.num_channels
print('Sampling rate =', fs)
print('Number of channels =', channels)
play_obj = wav_obj.play()
play_obj.wait_done()
```

If you are playing more than one file, be sure to follow the play command with:

`play_obj.wait_done()`

If you read .wav files using the **wavfile** module from **scipy.io,** then the sample rate is returned separately, as shown below for a stereo audio file. (Note that data.shape won't work for mono.)

```
# use scipy.io to read audio files
from scipy.io import wavfile as wav
fs1, data1 = wav.read('train32.wav')
print('Train whistle has: sampling rate', fs1, ', # of samples', len(data1), ', type', data1.dtype)
fs2, data2 = wav.read('tuba11.wav')
len2, ch2 = data2.shape
print('Tuba has: sampling rate', fs2, ', # of samples', len2, ', # of channels', ch2, ', type', data2.dtype)
```

If you have two-channel audio, you can extract the separate channels using data[:,ch] where ch is 0 or 1. Now let's create a new file that plays the two audio channels separately in sequence with a pause in between, and save the result as a new audio file.

```
pause = np.zeros(int(2*fs))                    # create a 2-sec pause
data0 = data[:,0]                              # extract channel 0
data1 = data[:,1]                              # extract channel 1
ptuba_data = np.concatenate([data0, pause, data1])   # insert pause between tubas
outfile = 'ptuba.wav'
wav.write(outfile,fs,ptuba_data.astype('int16'))# write wav file
```

4.0 Implementation of Time Operations

In class, we learned about two simple time transformations of signals: linear time scaling and time shifting. For continuous-time signals, the time argument takes on real values over an infinite (positive and negative) time range. On a computer, signals are finite length and discrete (sampled) time. This complicates time transformations in a few ways. First, we cannot use arbitrary time transformation factors (a and b in $y(t)=x(at+b)$), just as we cannot access signal values for arbitrary times. Second, we need to make assumptions about and keep track of the time window. We discuss these issues separately for time scaling and then shifting below. In considering these issues, we'll see that we cannot count on the transformation being invertible.

To speed up signals in discrete-time, you throw away time samples, and to slow them down, you insert time samples using interpolation. For example, if $y[n]=x[2n]$, then you throw away all the odd samples of $x[n]$. The speeding up and slowing down involves integer factors, so to get non-integer scaling factors, you need a combination of integer upsampling (inserting) and downsampling (discarding). Basically, this means that you can't implement an arbitrary scaling factor a – we use a close approximation that is rational.

To implement time scaling, we'll use a new package (decimal) in order to get the integer values needed to do the upsampling and downsampling. The only thing we need from that package is the as_integer_ratio method. For interpolation, we'll use the resampling function from the signal

processing module (signal) of scipy. You can simply use these functions as specified below. If you are interested, documentation is at:

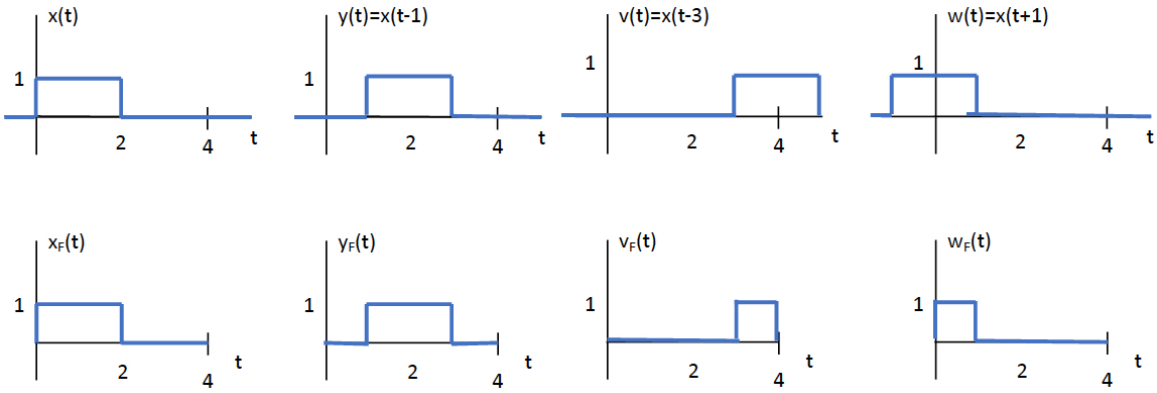
https://docs.python.org/3/library/decimal.html
https://docs.scipy.org/doc/scipy/reference/signal.html.

With these two packages (which will need to be imported in your import cell), we can now write a function that implements y(t)=x(at).

```
def timescale(x,fs,a):
    n,d = decimal.Decimal(a).as_integer_ratio()
    y= sig.resample_poly(x,d,n)
    t=np.arange(0,len(y),1)*(1/fs)
    return y,t
```

This function takes inputs: a vector x, sampling rate fs and scaling factor a. It returns two vectors: the new time signal in vector y and the new times associated with the samples in vector t. In this function, the input and output are not constrained to be the same length in time. If you can use the new time window, then you will need to create a new time vector to go with it. Sometimes you are constrained to keep the same time window as the original signal (e.g. for adding two signals), then you need to add or discard samples from the transformed signal to match the desired window. For time scaling a signal that starts at time 0 (a reasonable scenario for audio signals), we can just modify the end of the signal: concatenate a zero vector (for a>1) or discard the last samples (for 0<a<1). (It is standard to assume that any samples outside the original time window have value zero.)

Just as we cannot get values for a signal at arbitrary continuous times t, we cannot implement arbitrary time shifts *b*. With discrete signals, you can only do integer time shifts, so you need to find the time shift *b* in the same way. Time shifts also affect the time window. If we delay a signal by t0, we might concatenate a vector of n0 zeroes to the start of the signal and increase its length. A time advance would change the start time. To shift the signal in time with a fixed window, we concatenate zeroes on one end and discard samples on the other, depending on the direction of the shift. For example, let's say we have a signal *x(t)* that takes on value 1 for *t* in [0,2] and 0 elsewhere, and on the computer we only represent the time window [0,4]. The figure below shows 3 different time shifts of this signal, with the ideal infinite time case in the first row and the finite-window case in the second row, using the F subscript to indicate that it is the finite-length version. Note that information can be lost in some cases, so it is important to choose the time window carefully.



Another possibility is a hybrid: allow the signal to extend in positive time but keep the start time at zero.

For the time shift function that you will write yourself in the lab, we will constrain the input and output to both span the same time window. This means that:

- When you advance the signal (b>0), some early samples will be lost, and the same number of samples will be added as zeroes at the end.
- When you delay the signal (b<0), some ending samples will be lost, and the same number will be added as zeroes at the beginning.

Because you have differences for these conditions, you will need conditional control in this function.

In working with ideal continuous-time signals, linear time transformations are invertible. In other words, if you transform *x(t)* using *y(t)=x(at+b)*, then you can always undo that transformation to recover *x(t)*. Specifically, you would use

$$v(t) = y((t-b)/a) = x(a(t-b)/a + b) = x(t).$$

On a computer, signals are finite length and discrete (sampled) time, and you cannot always undo a time transformation. As you can see from the discussion above, it is possible that information will be lost in the time transformation for multiple reasons: fixed time window, approximation of the time transformation factors, and through the process of time sampling. For example, if *y[n]=x[2n],* then you throw away all the odd samples of *x[n].* Clearly, you can't recover the original signal if you've thrown part of it away. The best way to counteract this problem is to choose your time windows with some knowledge of the signal, and to use a sufficiently high sampling rate **fs** such that the samples thrown away don't matter.