EE 242 Lab 1a – Modifying Signals - Sampling and Amplifying

Yehoshua Luna, Ben Seinhart, Aaron McBride, Team AE01

This lab has 2 exercises to be completed as a team. Each should be given a separate code cell in your Notebook, followed by a markdown cell with report discussion. Your notebook should start with a markdown title and overview cell, which should be followed by an import cell that has the import statements for all assignments. For this assignment, you will need to import: numpy, the wavfile package from scipy.io, and matplotlib.pyplot.

```
In [ ]: # We'll refer to this as the "import cell." Every module you import should be impor
import numpy as np
import matplotlib
import scipy.signal as sig
import scipy.io.wavfile as wav
import matplotlib.pyplot as plt

# import whatever other modules you use in this lab -- there are more that you need
from IPython import display
from IPython.display import display, Audio
```

Summary

In this lab, you will work through a series of exercises to introduce you to working with audio signals and explore the impact of different amplitude and time operations on signals. This is a two-week lab. You should plan on completing the first 2 assignments in the first week.

Lab 1a turn in checklist

• Lab 1a Jupyter notebook with code for the 2 exercises assignment in separate cells. Each assignment cell should contain markdown cells (same as lab overview cells) for the responses to lab report questions. Include your lab members' names at the top of the notebook.

Please submit the report as PDF

Assignment 1 -- Working with sound files

Start a new cell following the guidelines in the Lab 1a template, dividing it into Parts A-C.

A. Download the train.wav sound file provided. Read in the file saving the audio vector and sampling frequency in variables $x_1(t)$ and f_{s_1}, respectively. Print the sampling rate f_{s_1}

(which should be 32kHz) and the shape of $x_1(t)$, which will tell you the length and number of channels.

B. Write out two new versions of the file in wav format using different sampling rates: $f_{s_2} = f_{s_1}/2$ (16 kHz) and $f_{s_3} = 1.5 f_{s_1}$. Basically create $x_2(t) = x_1(t/2)$ and $x_3(t) = x_1(2t)$

C. Read in the three different versions of the train sound file and play each one. ($x_1(t), x_2(t), x_3(t)$, don't change the f_s while playing)

In [8]:
```
# Assignment 1 - Time Scaling Function
# Start with a comment section that explains what the input and output variables ar

# x: input signal vector
# fs: sampling rate (in Hz)
# a: scaling parameter. This has to be a decimal value for as_integer_ratio to work
# So, explicitly casting it into a float or a double or any fractional data type wi
# Returns t: time samples vector corresponding to y: scaled signal
def timescale(x, fs, a):
    [n, d] = (np.double(a)).as_integer_ratio()
    y = sig.resample_poly(x, d, n)
    t = np.arange(0, y.size,1) * (1.0/fs)
    return y, t

# x: input signal vector
# fs: sampling rate (in Hz)
# a: scaling parameter
# Returns t: time samples vector corresponding to y: scaled signal
def linear_timescale(x, fs, a):
    # Implements the linear time scaling that we learned in class
    indices = np.arange(0, x.size, a)
    y = np.interp(indices, np.arange(x.size), x)
    t = np.arange(0, y.size, 1) * (1.0/fs)
    return y, t
```

In [20]:
```
# Assignment 1 - Playing and Plotting Time Scaled Audio Files

# Part A
# 1. Read the train.wav file
f1, x1 = wav.read("train32.wav")

# 2. Print the sampling rate and shape of signal
print("Sample rate: " + str(f1))
print("Signal shape: " + str(original.shape[0]))

# Part B
# 1. Use both timescale and linear_timescale functions above to resample the signal
low_poly, dummy = timescale(original, f1, 0.5)
high_poly, dummy = timescale(original, f1, 2)

low_linear, dummy = linear_timescale(original, f1, 0.5)
high_linear, dummy = linear_timescale(original, f1, 2)

# 2. Store the files as train_(high/low)_poly.wav and train_(high/low)_linear.wav (
wav.write('train_low_poly.wav', f1, low_poly.astype(np.int16))
```

```python
wav.write('train_high_poly.wav', f1, high_poly.astype(np.int16))

wav.write('train_low_linear.wav', f1, low_linear.astype(np.int16))
wav.write('train_high_linear.wav', f1, high_linear.astype(np.int16))

# Part C
# 1. Play the files
print("Original: ")
display(Audio('train32.wav'))

print("Low poly: ")
display(Audio('train_low_poly.wav'))

print("High poly: ")
display(Audio('train_high_poly.wav'))

print("Low linear: ")
display(Audio('train_low_linear.wav'))

print("High linear: ")
display(Audio('train_high_linear.wav'))
```

```
Sample rate: 32000
Signal shape: 50313
Original:
```

0:00 / 0:01

```
Low poly:
```

0:00 / 0:03

```
High poly:
```

0:00 / 0:00

```
Low linear:
```

0:00 / 0:03

```
High linear:
```

0:00 / 0:00

Discussion

Comment on how the audio changes when the incorrect sampling frequency is used.

- When the sampling frequency used for playback is *less* than the original sampling frequency:
  - **The speed of the audio *decreases*.** This is because the samples get played back at a slower rate then they were recorded, which increases the duration between tones in the output.
  - **The pitch of the audio *decreases*.** This is because the samples get played back at a slower rate then they were recorded, which increases the wavelength of tones in the audio (due to more time between samples) - thus reducing their pitch.
  - **The duration of the audio clip is *extended*.** This is because the same number of samples are played back at a slower rate, which means more time is required to play back all the samples.
- When the sampling frequency used for playback is *greater* than the original sampling frequency:
  - **The speed of the audio *increases.*** This is because the samples get played back at a faster rate then they were recorded, which decreases the duration between tones in the output.
  - **The pitch of the audio *increases*.** This is because the samples get played back at a faster rate then they were recorded, which decreases the wavelength of tones in the audio (due to less time between samples) - thus increasing their pitch.
  - **The duration of the audio clip *decreases*.** This is because the same number of samples are played back at a faster rate, which means less time is required to play back all the samples.

Assignment 2 -- Amplitude Operations on Signals

Again, following the guidelines in the Lab 1 template, start a new cell and write a script to meet the following specifications. This assignment will have three parts, A-C, each of which should be indicated with comments.

A.1. Create a discrete time signal $s_1(t)$ that is the same length as $x_1(t)$ and has value 1 for t=[0,0.5] and value 0.2 for $t > 0.5$.

You can use the command below where len is the length of $x_1(t)$ and n_0 is the index corresponding to t=0.5, s1 = np.concatenate((np.ones(n0),0.2*np.ones(len(x1)-n0))

A.2. Multiply x1 with s1 to create v1. Save this signal to a wav file.

B.1 Create a discrete-time decaying ramp signal r1, that is the same length as x1. The signal should have value 1 at time 0 and linearly decay to value 0. (Hint: use numpy.arange.)

B.2 Multiply x1 with r1 to create v2. Save this signal to a wav file.

C. Read in v1 and v2 and play the two different modifications together with the original, to verify that the volume of the second whistle is reduced.

Ensure to play all the signals for the TAs

In [22]:
```python
# Assignment 2 - Amplitude Operations on Signals

# Part A
# 1. Create a signal s1 which 1 from 0s to 0.5s and 0.2 from 0.5s onward
s1 = np.concatenate((np.ones(int(0.5 * f1)), 0.2 * np.ones(int(x1.size - 0.5 * f1))

# 2. Multiply s1 with x1 sample by sample to create v1
v1 = s1 * x1

# Part B
# 1. Create a signal r1 which goes down linearly from 1 to 0 across the length of t
r1 = np.arange(1, 0, -1/x1.size)

# 2. Multiply r1 with x1 sample by sample to create v2
v2 = r1 * x1

# Part C
# 1. Store and play v1 and v2
wav.write('train_step.wav', f1, v1.astype(np.int16))
wav.write('train_ramp.wav', f1, v2.astype(np.int16))

print("Original: ")
display(Audio('train32.wav'))

print("Stepped: ")
display(Audio('train_step.wav'))

print("Ramped: ")
display(Audio('train_ramp.wav'))
```

```
Original:
```

0:00 / 0:01

```
Stepped:
```

0:00 / 0:01

```
Ramped:
```

0:00 / 0:01

Discussion

Discuss the differences that the two modifications have on the signal. What would happen if you defined s1 to take value 2 for the [0,0.5] range? If you wanted a smooth but faster decay in amplitude, what signal might you use?

- **If s1 took the value 2 for the [0, 0.5] range:** The audio signal would be *two times louder* than the original for the first half of the clip. This is because all samples within the range would be multiplied by 2, which would increase their magnitude and thus overall volume. For the second half of the clip the audio would still be 20% as loud as the original clip (asuming the configuration for s1 is not otherwise altered).
- **If you wanted a smooth but faster decay in amplitude:** You could use a ramp signal with a *greater negative slope*. This would cause the amplitude of the resulting signal to decrease faster because each subsequent sample would be multiplied by a value which is decreasing at a faster rate (recall, greater negative slope indicates the value decreases at a faster rate). Note, you could also use an exponentially decreasing signal - althought that would only decrease the volume faster in the begining.