

ECE 4680 DSP Laboratory 2: Speech Signal Processing Using Python Sound Functions

This lab is to be worked in teams of most two. Each team will submit a lab report describing their results. An E-mail ZIP package containing .wav files will also be turned in or a lab demo to the instructor, to allow evaluation of the processing algorithms.

Introduction

The basic investigation for this experiment will be the use of *butt splicing* to either speed up or slow down the playing time (delivery) of a recorded speech signal (in Python/Numpy an ndarray), without altering the pitch. Think about this for a moment. If a signal is played back at a sampling rate of twice the original record value, the play back time is cut in half, and the pitch is doubled. Likewise if a signal is played back at a sampling rate of one half the original record value, the play back time is doubled, and the pitch is halved. Using butt splicing of speech segments it is possible to maintain the recorded pitch while either slowing down or speeding up the play back time.

To implement the above algorithms you will be using the PC sound system and Python (~~MATLAB~~) audio processing functions for recoding, storage, and playback of recorded speech segments. Python in the Jupyter notebook offers approaches for both *real-time* and *near real-time* audio signal processing. In this experiment you will become familiar with both approaches. Ultimately you will be working with your own speech segment recorded using your own voice via a microphone. This will be archived using a *.wav file so it will be easy to re-load your speech test vector as you develop algorithms. The details can be found in Appendix A of this lab document, which is a PDF rendering of a corresponding Jupyter notebook found on the ECE 4680 and ECE 4650/5650 Web Site as Audio_Record_Playback.zip.

As brief overview of the tools, for recording audio in the Jupyter notebook you can use pyaudio_helper.py. The resulting audio capture can be immediately played in the notebook using either pyaudio_helper or the Ipython.display.Audio() function which is a notebook *widget control*. It is best to archive the audio recording in a wave file using the functions of Table 1 from the module sigsys.py of scikit-dsp-comm. A Jupyter notebook found in the Audio_Record_Playback.zip contains complete examples.

Digital Signal Processing and Windows Wave Files

The multimedia capability Windows brings the ability to handle digitized audio signals in what is known as the wave (*.wav) file format. A Wave audio file is a binary file format for storing continuous-time audio source material, in sampled data form, as a result analog-to-digital (A/D) con-

version [1].

Table 1: Python sound functions for Win/Mac/Linux OS's.

Python Sound Related Functions	
<code>to_wav(filename, rate, x)</code>	Write a wave file. A wrapper function for <code>scipy.io.wavfile.write</code> that also includes int16 scaling and conversion. Assume input <code>x</code> is [-1,1] values.
<code>from_wav(filename)</code>	Read a wave file. A wrapper function for <code>scipy.io.wavfile.read</code> that also includes int16 to float [-1,1] scaling.

A PC sound system also includes two digital-to-analog (D/A) converters for playback. The block diagram of Figure 1 depicts the DSP capability that a sound system equipped PC typically has. In Appendix A you will also see a system block diagram from the PyAudio point of view. A hyper-link in the appendix also takes you to a 2018 Scipy paper describing how full input/output real-time DSP can be performed. This lab does not use all of the capability of PyAudio.

Table 1. To play a wave file you use the `Audio()` function available in the Jupyter notebook (see

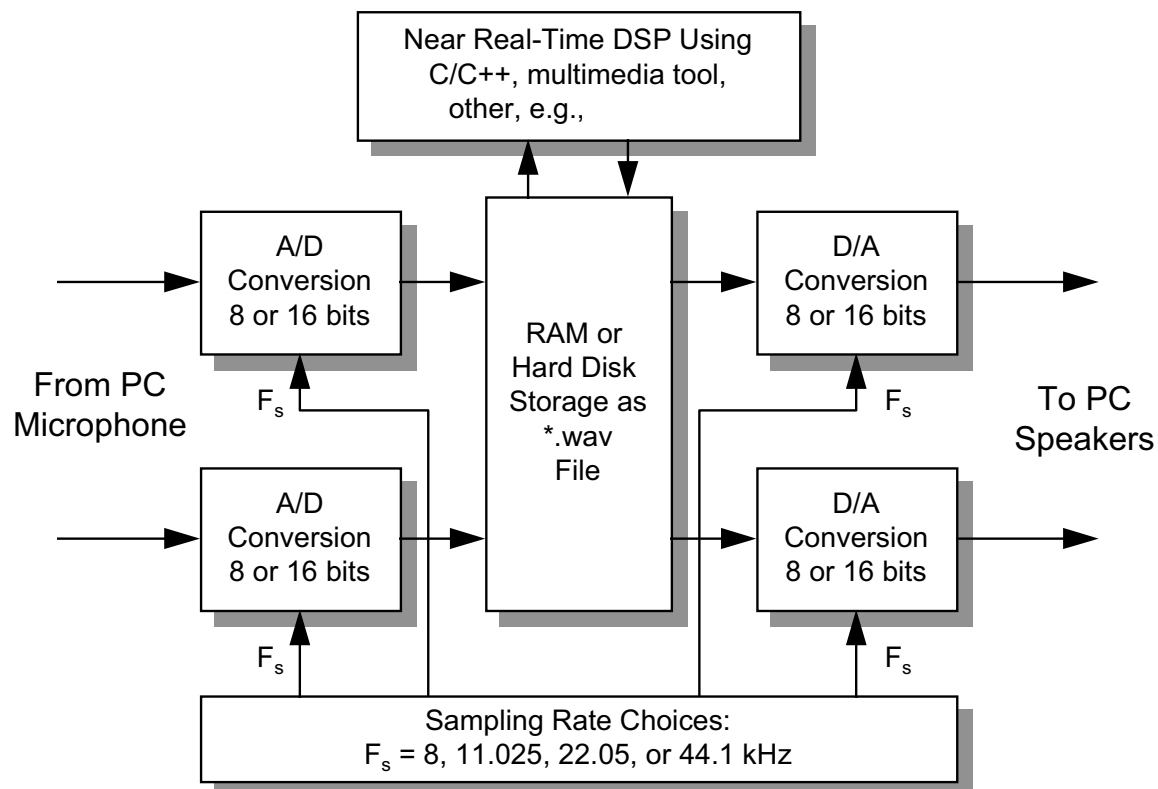


Figure 1: DSP capability of the typical sound card equipped PC.

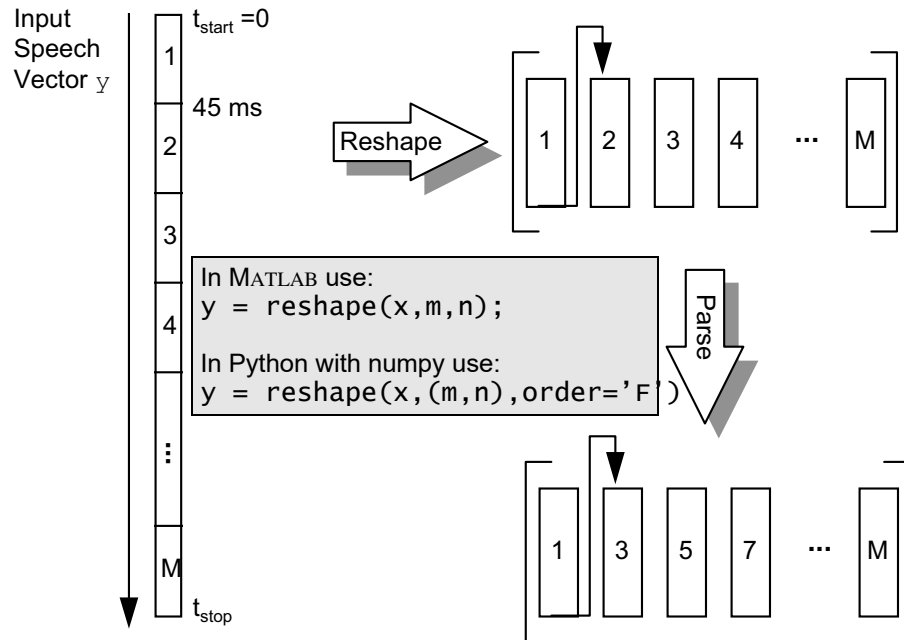


Figure 2: Butt splicing to decreasing playing time yet maintain sound pitch.

Figure 1) or use the audio player of your OS to play the wave file directly.

Simple Rate Changing Independent of Pitch

Without getting too detailed this section explains a simple technique for either increasing or decreasing the playing time of a sound vector, without altering the sound pitch. Increasing the rate refers to decreasing the playing time, while decreasing the rate implies increasing the playing time. Both operations can be solved in an approximate way by *butt splicing* speech segments of say 45 ms or so. Butt splicing in DSP is analogous to adding or removing segments of magnetic recording tape by end-to-end taping them together to form a new edited tape. Butt splicing of speech sequences thus implies that no transition smoothing is used, just a hard end-to-end connection.

Decreasing Playback Time

To decrease the playback time, yet retain the proper pitch, all we need do is to periodically remove short segments of the original speech vector, butt splice the remaining pieces back together, then play it back at the original recording rate. If the pattern is save 45 ms, discard 45 ms, save 45 ms, etc., the new sound vector will be half as long as the original, thus it will play in half the time. A graphical description of the operation in terms of Python ndarrays is shown in Figure 2. The segment length may need to be adjusted for best sound quality. Note for the case of Python, the option `order='F'` insures that Fortran ordering is taken in the reshape. This also matches the way MATLAB does things.

Increasing Playback Time

To increase the playback time, yet retain the proper pitch, all we need do is to periodically repeat short segments of the original speech vector, again using a butt splicing technique, then play it back at the original recording rate. If the pattern is say 45 ms, repeat previous 45 ms, save next 45 ms, etc., the new sound vector will be twice as long as the original, thus it will play in twice the time. Python Hint (note the `.T` operator forms the matrix transpose):

```
In [226]: A = array([[1,2],[3,4]]) # make sure to create an ndarray
```

```
In [227]: hstack((A,A))
```

```
Out[227]:
```

```
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
```

```
In [228]: hstack((A.T,A.T))
```

```
Out[228]:
```

```
array([[1, 3, 1, 3],
       [2, 4, 2, 4]])
```

```
In [229]: hstack((A.T,A.T)).T
```

```
Out[229]:
```

```
array([[1, 2],
       [3, 4],
       [1, 2],
       [3, 4]])
```

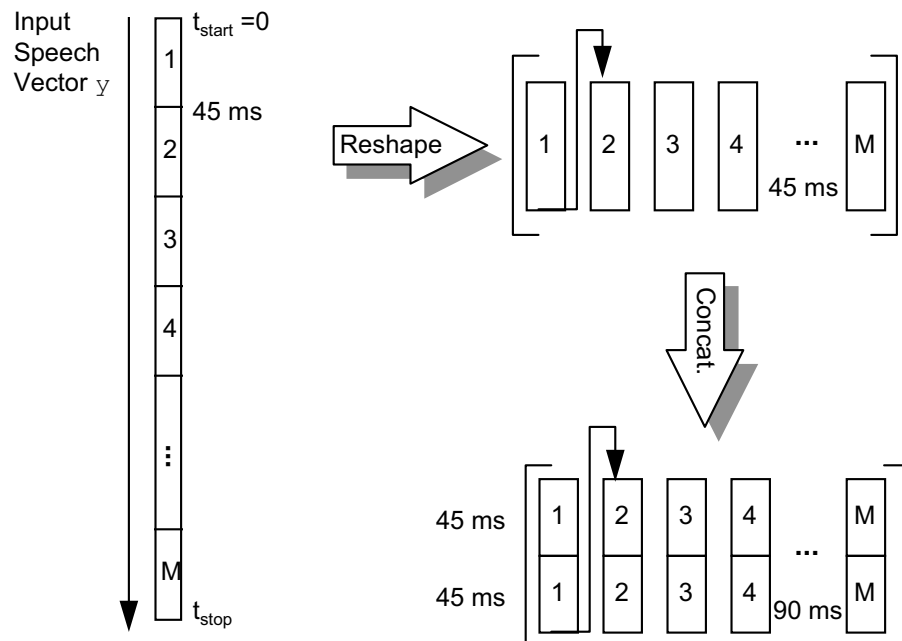


Figure 3: Butt splicing to increase playing time yet maintain sound pitch.

Tasks

1. Create a short wave file using 16 bits per sample resolution, $F_s = 11.025$ ksp/s, and duration of about 30 seconds, using the module `pyaudio_helper` running in a Jupyter notebook or a similar wave recording application. A microphone will be needed as well. See the Appendix A of this document and the ZIP file under Lab 2 entitled `Audio_Record_Playback.zip`. Save the file using

```
ss.to_wav('mytest_file.wav', 11025, x_1D_ndarray)
```

2. Import the wave file into the IPython/Jupyter notebook workspace using the function


```
fs, y = ss.from_wav('mytest_file.wav')
```
3. Beyond using `pyaudio_helper`, as described in Task 1, verify that `Audio(y, rate=Fs)`, where `y` is a numpy ndarray, plays one channel of audio through the sound system, and that raising or lowering `Fs` changes both the duration and pitch of the message. Note `Audio` only works in the Jupyter notebook. Using the length of `y` and `Fs` determine the exact time duration of the played sound vector. Note for future reference left and right audio channels can also be played by using two equal length arrays using `Audio([y_r, y_l], rate=Fs)` or using a wave file as `Audio('audio.wav')`. `Pyaudio_helper` with the looping feature, as shown in Appendix A, can of course also be used, then you can hear the results over and over.
4. Reshape the 1D sound array `y` into a $100 \times M$ 2D array, say `y_rs`, such that M allows the maximum number of full columns to be filled from the length L of `y`. You will need to trim the length of `y` to be integer compatible. See Appendix B, which discusses 1D and 2D arrays, for details. Listen to the playback of the reshaped using `flatten()`. Then listen to the array that is first transposed and then flattened. The transposed version should be scrambled.
5. Butt splice `y` by removing alternate 45 ms speech segments to halve the delivery time and maintain the same pitch. Export your modified speech vector back into a wave file and verify that it is playable.
6. Butt splice `y` by inserting redundant 45 ms speech segments to double the delivery time and maintain the same pitch. Export your modified speech vector back into a wave file and verify that it is playable.
7. Try variations on 5 and 6 to improve the quality, or further alter the rate change.
8. Analyze your speech waveforms using the spectrogram (`specgram`) function


```
>> specgram(y, FFTLENGTH=256, Fs=fs);
```

 Recall from ECE 2610 that a spectrogram is a frequency spectrum versus time plot.
9. Summarize your findings.
10. Prepare for instructor demo.

Bibliography/References

- [1] Tim Kientzle, *A Programmers Guide to Sound*, Addison-Wesley, Reading, Ma, 1998.

Appendix A

Audio Recording and Playback in Jupyter Notebook

```
In [1]: %pylab inline
import sk_dsp_comm.sigsys as ss
import sk_dsp_comm.pyaudio_helper as pah
import imp
import sk_dsp_comm.fir_design_helper as fir_d
import scipy.signal as signal
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
from IPython.display import Audio, display
from IPython.display import Image, SVG
```

Populating the interactive namespace from numpy and matplotlib

```
In [2]: pylab.rcParams['savefig.dpi'] = 100 # default 72
        pylab.rcParams['figure.figsize'] = (6.0, 4.0) # default (6,4)
        %config InlineBackend.figure_formats=['png'] # default for inline viewing
        %config InlineBackend.figure_formats=['svg'] # SVG inline viewing
        %config InlineBackend.figure_formats=['pdf'] # render pdf figs for LaTeX
```

1 Introduction

In this notebook (also the [Appendix for ECE 4680 Lab 2](#)) we consider audio processing, real-time and near real-time, in the Jupyter notebook. If you are reading the PDF rendering of this notebook, it was first exported from the notebook version as $\text{\LaTeX}$ and then the .tex file was edited in [VS code](#) using the LaTeX Workshop extension, to include screenshots of GUI controls from the running Jupyter notebook. Finally a PDF file was produced.

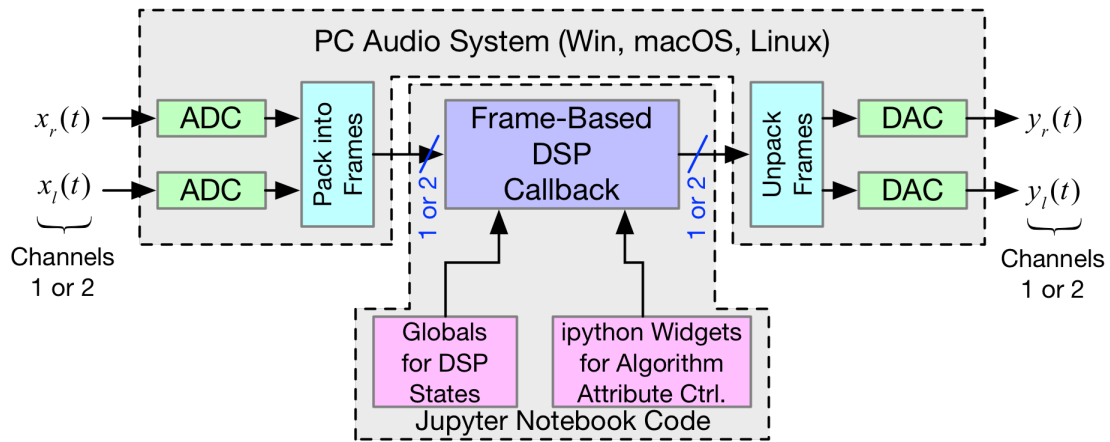
There are two topics to be covered here:

1. Using `pyaudio_helper.py` from [Scikit-DSP-Comm](#) for audio recording and playback/streaming via audio looping
2. Using the `IPython.display` function `Audio()` to playback existing wav files or ndarray 1D signals

1.1 Pyaudio_helper for Recording and Playback

In this first subsection we consider the use of PyAudio via `scikit-dsp-comm.pyaudio_helper` to record and playback audio samples streams is described. The `ipywidgets` or `jupyterwidgets` are used to provide GUI controls inside the notebook. The widgets must be installed on your system. In particular when using Jupyter Lab, a recently released upgrade to the older Jupyter Notebook interface, requires additional installation steps.

A simplified block diagram of PyAudio *streaming-based* (nonblocking) signal processing when using `pyaudio_helper` and `ipython` widgets is shown in the figure below.



Once `pyaudio_helper` is imported (here via `import sk_dsp_comm.pyaudio_helper as pah`) you can see what audio devices are available on your or lab system you are in front of:

```
In [3]: pah.available_devices()
```

```
Out[3]: {0: {'name': 'Microsoft Sound Mapper - Input', 'inputs': 2, 'outputs': 0},
        1: {'name': 'Microphone (Realtek Audio)', 'inputs': 2, 'outputs': 0},
        2: {'name': 'Microsoft Sound Mapper - Output', 'inputs': 0, 'outputs': 2},
        3: {'name': 'Speakers / Headphones (Realtek ', 'inputs': 0, 'outputs': 2}}
```

Note if you plugin additional USB audio devices or you PC is *docked* in a docking station, additional devices will be displayed. If devices are added after the Python kernel starts, you will have stop and restart the kernel for the new devices to be identified.

1.1.1 Single Channel Recording with Record Monitor Level Control

Here we make a single channel recording (mono) using an external mic input, then save the output to a wav file for later playback in a repeating loop. The raw capture samples could also be manipulated in Python statically before playback or in real-time while being played back.

To aid the recording process to slider widgets are configured: (1) To set the record level into the capture buffer `DSP_IO_1.data_capture` and (2) to set the playback level so that if output device 5, in this case, can be brought down to avoid unwanted feedback getting into the mic input. Due to the latency with the PyAudio processing, feedback from output input will result in echo.

```
In [4]: Record_gain = widgets.FloatSlider(description = 'Record Gain',
        continuous_update = True,
        value = 1.0,
        min = 0.0,
        max = 2.0,
        step = 0.01,
        orientation = 'horizontal')
        Monitor_gain = widgets.FloatSlider(description = 'Monitor Gain',
        continuous_update = True,
        value = 0.1,
        min = 0.0,
```

```

max = 2.0,
step = 0.01,
orientation = 'horizontal')

```

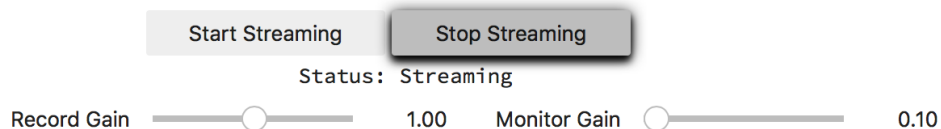
```

In [5]: # define callback (3)
# Here we configure the callback to play back a wav file
def callback1(in_data, frame_count, time_info, status):
    global DSP_IO_1
    DSP_IO_1.DSP_callback_tic()

    # convert byte data to ndarray
    in_data_nda = np.frombuffer(in_data, dtype=np.int16)
    # Ignore in_data when generating output only
    #*****
    x = in_data_nda.astype(float32)
    y = Record_gain.value*x
    # Note wav is scaled to [-1,1] so need to rescale to int16
    #y = 32767*x.get_samples(frame_count)
    # Perform real-time DSP here if desired
    #
    #*****
    # Save data for later analysis
    # accumulate a new frame of samples
    DSP_IO_1.DSP_capture_add_samples(y)
    #*****
    # Monitor level control
    y *= Monitor_gain.value
    # Convert from float back to int16
    y = y.astype(int16)
    DSP_IO_1.DSP_callback_toc()
    return y.tobytes(), pah.pyaudio.paContinue

In [6]: # fs, x_wav2 = ss.from_wav('Music_Test.wav')
# x_wav = (x_wav2[:,0] + x_wav2[:,1])/2
# x = pah.loop_audio(x_wav)
DSP_IO_1 = pah.DSP_io_stream(callback1,1,3,fs=44100,Tcapture=10)
DSP_IO_1.interactive_stream(Tsec=5)
widgets.HBox([Record_gain,Monitor_gain])

```



1.1.2 Save Captured Samples to a *wave* File

Before saving to a wav file the short capture made above, at $f_s = 44100$ Hz, was first trimmed to remove a startup recording glitch and then dead-air at the back end was removed by setting

the sample length to just $N_{\text{samples}} = 100,000$ less $N_{\text{trim}} = 6000$. For Lab 2 when sampling at just $f_s = 22050$ Hz, trimming is optional, and clearly the total length will be greater than 100,000 samples, as $30 \times 22,050 = 661,500$ samples.

```
In [20]: # Scale values to [-1,1]
```

```
Ntrim = 6000
```

```
Nsamps = 100000
```

```
ss.to_wav('my_record.wav', 44100,  
         DSP_IO_1.data_capture[Ntrim:Nsamps]/  
         max(abs(DSP_IO_1.data_capture[Ntrim:Nsamps])))
```

```
In [25]: # mwickert saying "Hello there, is this really working?"
```

```
# A little echo is present due to the monitor gain being 0.1. on a MacBook
```

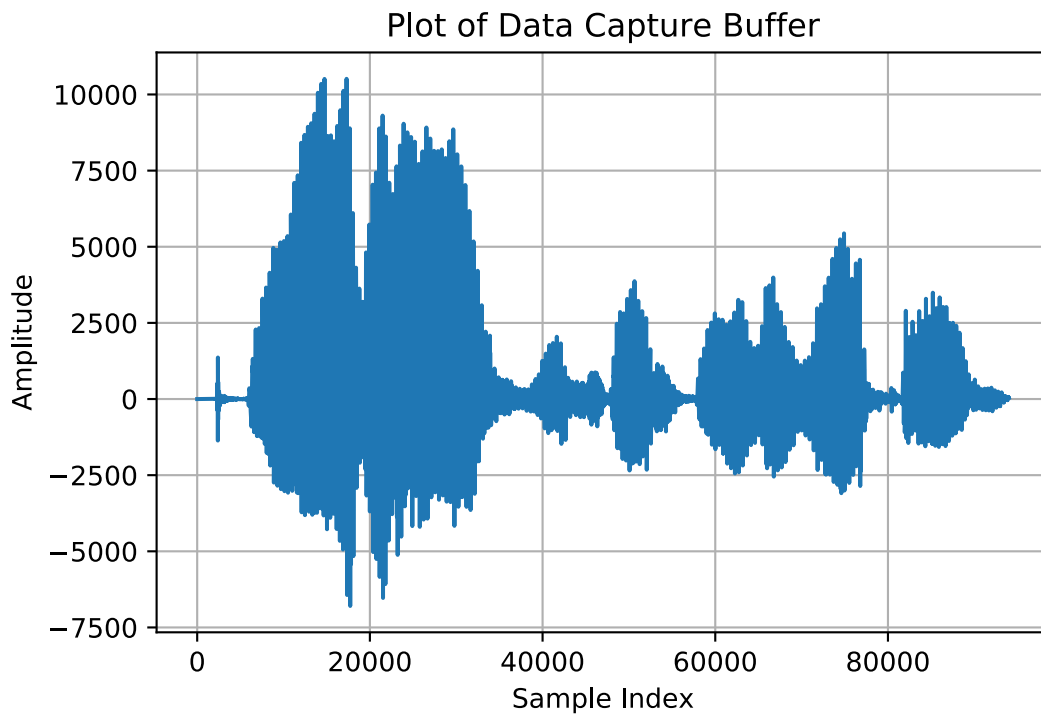
```
plot(DSP_IO_1.data_capture[Ntrim:Nsamps])
```

```
title(r'Plot of Data Capture Buffer')
```

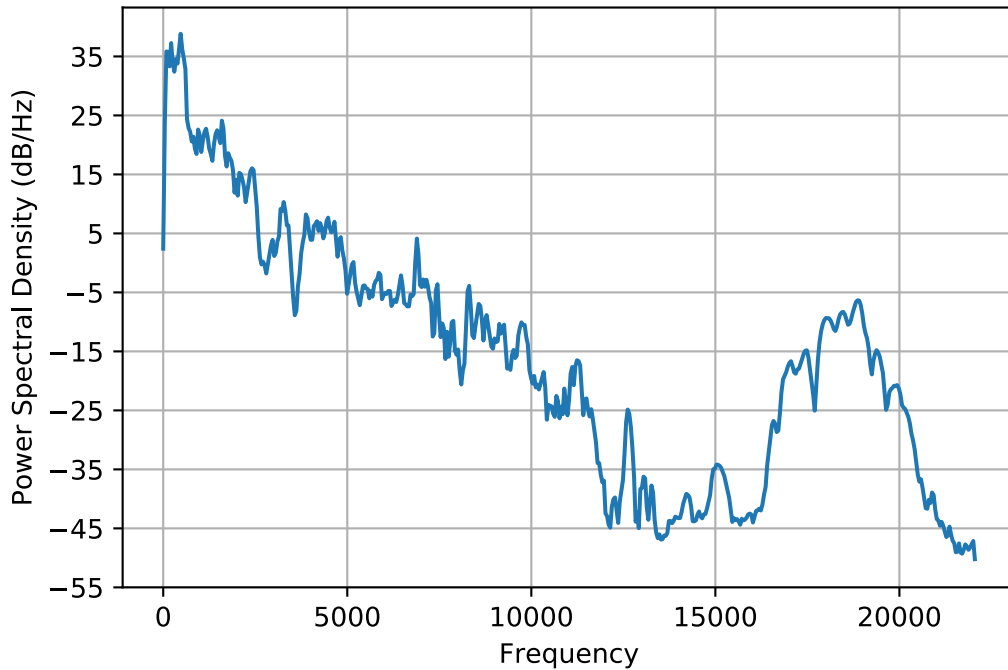
```
xlabel(r'Sample Index')
```

```
ylabel(r'Amplitude')
```

```
grid();
```



```
In [22]: psd(DSP_IO_1.data_capture[Ntrim:Nsamps], 2**10, 44100);
```



1.1.3 Single Channel Loop Playback

Here we take the captured samples that have been saved in a wavefile and play them back in a repeating audio loop. The samples could also be manipulated first before playback.

No audio controls are used for the playback other than the standard start and stop buttons.

```
In [7]: # define callback (3)
# Here we configure the callback to play back a wav file
def callback2(in_data, frame_count, time_info, status):
    global DSP_IO, x_loop
    DSP_IO.DSP_callback_tic()

    # convert byte data to ndarray
    #in_data_nda = np.frombuffer(in_data, dtype=np.int16)
    # Ignore in_data when generating output only
    #*****
    #x = in_data_nda.astype(float32)
    #y = Record_gain.value*x
    # Note wav is scaled to [-1,1] so need to rescale to int16
    y = 32767*x_loop.get_samples(frame_count)
    # Perform real-time DSP here if desired
    #
    #*****
    # Save data for later analysis
    # accumulate a new frame of samples
```

```

DSP_IO.DSP_capture_add_samples(y)
*****
# Convert from float back to int16
y = y.astype(int16)
DSP_IO.DSP_callback_toc()
return y.tobytes(), pah.pyaudio.paContinue

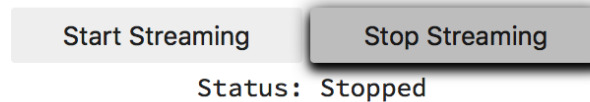
```

In the below cell before the DSP_IO object is created we first load the wav file into a loop_audio object and then pass that object, x_loop, into the above callback function, callback2. This then allows infinite looping of the original wav file to output device 5.

```

In [8]: fs, x_rec = ss.from_wav('my_record.wav')
        x_loop = pah.loop_audio(x_rec)
        DSP_IO = pah.DSP_io_stream(callback2,1,3,fs=44100,Tcapture=0)
        DSP_IO.interactive_stream(Tsec=0)

```



1.1.4 Stereo Gain Sliders with a Stereo Audio Loop

In this example two audio channels are employed (stereo) to loop the wav file Music_test.wav. Two slider widgets are used to independently control the gain of the left and right audio paths sent to output device 5.

```

In [9]: L_gain = widgets.FloatSlider(description = 'L Gain',
                                     continuous_update = True,
                                     value = 1.0,
                                     min = 0.0,
                                     max = 2.0,
                                     step = 0.01,
                                     orientation = 'vertical')
        R_gain = widgets.FloatSlider(description = 'R Gain',
                                     continuous_update = True,
                                     value = 1.0,
                                     min = 0.0,
                                     max = 2.0,
                                     step = 0.01,
                                     orientation = 'vertical')

        #widgets.HBox([L_gain, R_gain])

In [10]: # L and Right Gain Sliders
        def callback(in_data, frame_count, time_info, status):
            global DSP_IO_3, L_gain, R_gain, x_loop_stereo
            DSP_IO_3.DSP_callback_tic()
            # convert byte data to ndarray

```

```

in_data_nda = np.frombuffer(in_data, dtype=np.int16)
# separate left and right data
#x_left,x_right = DSP_IO.get_LR(in_data_nda.astype(float32))
# Use a loop object as a source of stereo samples
# Note since wave files are scaled to [-1,1] we scaling to
# rescale to the dynamic range of int16
new_frame = x_loop_stereo.get_samples(frame_count)
x_left = 20000*new_frame[:,0]
x_right = 20000*new_frame[:,1]
*****
# DSP operations here
y_left = x_left*L_gain.value
y_right = x_right*R_gain.value

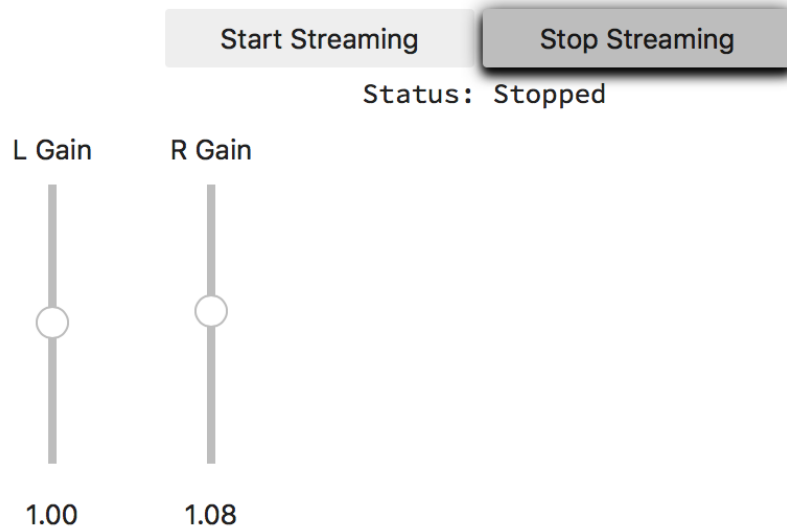
*****
# Pack left and right data together
y = DSP_IO_3.pack_LR(y_left,y_right)
# Typically more DSP code here
*****
# Save data for later analysis
# accumulate a new frame of samples
DSP_IO_3.DSP_capture_add_samples_stereo(y_left,y_right)
*****
# Convert from float back to int16
y = y.astype(int16)
DSP_IO_3.DSP_callback_toc()
# Convert ndarray back to bytes
#return (in_data_nda.tobytes(), pyaudio.paContinue)
return y.tobytes(), pah.pyaudio.paContinue

```

```

In [11]: fs, x_in_stereo = ss.from_wav('Music_Test.wav')
x_loop_stereo = pah.loop_audio(x_in_stereo)
DSP_IO_3 = pah.DSP_io_stream(callback,1,3,fs=44100,Tcapture=0)
DSP_IO_3.interactive_stream(0,2)
widgets.HBox([L_gain, R_gain])

```



1.1.5 Real-Time DSP

To implement one or two channel real-time DSP in the Jupyter notebook, see the 2018 Scip Conference paper [Real-Time Digital Signal Processing Using pyaudio_helper and the ipywidgets](#).

1.2 Using IPython.display.Audio to Stream Wavefiles

In this subsection we provides examples of how to use the Audio function that is imported from IPython.display at the top of the notebook. The module `sk_dsp_comm.sigsys`, i.e.,

```
import sk_dsp_comm.sigsys as ss
```

is also useful as it contains the functions

```
fs, x = ss.from_wave('file.wav')
```

and

```
ss.to_wave('file_name',fs,x)
```

for importing and exporting wav files to the Python workspace.

1.2.1 Using the Test File *zero.wav*

Included in the ZIP of this notebook is a stereo test file of a voice staying the word *zero*. We use that file in a few experiments below.

```
In [12]: fs,x = ss.from_wav('zero.wav')
```

```
In [16]: Audio('zero.wav')
```

```
Out[16]:
```



From the Audio function API we can also feed it with one or two numpy ndarrays. The `zero.wav` test file is stereo, so `x` is a 2D array containing two columns. We must first reduce the 2D array to a 1D array using slicing.

```
In [20]: x.shape
```

```
Out[20]: (56568, 2)
```

- Direct mono playback:

```
In [14]: Audio(x[:,0],rate=fs) # play the left channel
```

```
Out[14]:
```



- Direct stereo playback:

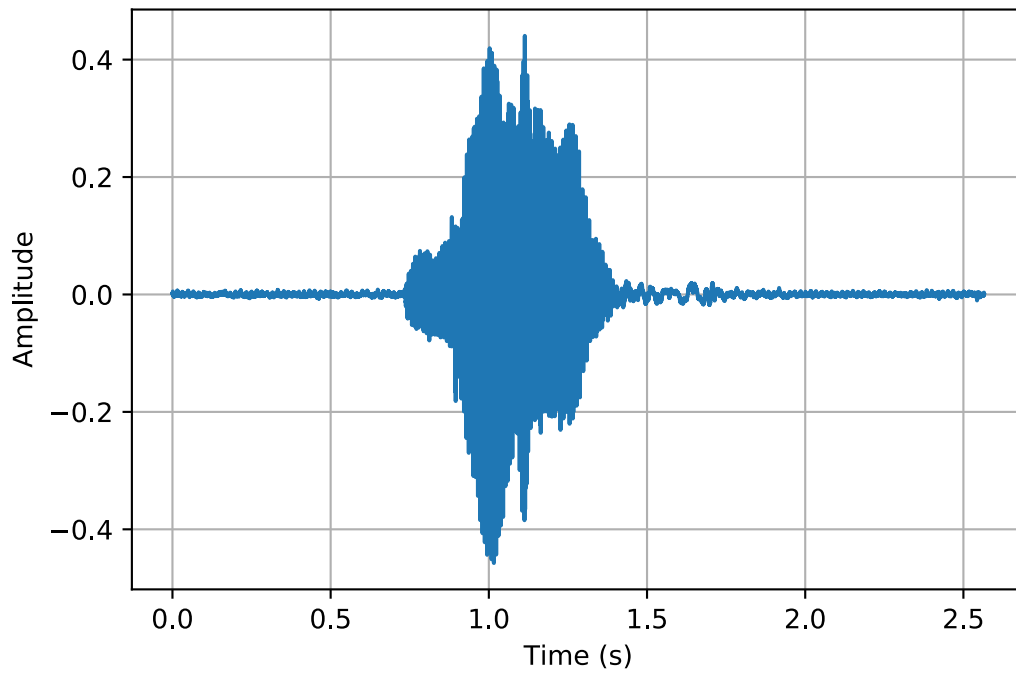
```
In [15]: Audio([x[:,0],x[:,1]],rate=fs) # play the left and right channels
```

```
Out[15]:
```



- Take a look at the waveform and its scaling relative to the normalized wavefile amplitude range of $[-1, 1]$

```
In [17]: # mwickert saying the word "Zero"
t = arange(len(x))/float(fs)
plot(t,x[:,0])
xlabel('Time (s)')
ylabel('Amplitude')
grid();
```



Appendix B

Working with 1D and 2D ndarrays

```
In [1]: %pylab inline
import sk_dsp_comm.sigsys as ss
import sk_dsp_comm.pyaudio_helper as pah
import imp
import sk_dsp_comm.fir_design_helper as fir_d
import scipy.signal as signal
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
from IPython.display import Audio, display
from IPython.display import Image, SVG
```

Populating the interactive namespace from numpy and matplotlib

```
In [2]: pylab.rcParams['savefig.dpi'] = 100 # default 72
        pylab.rcParams['figure.figsize'] = (6.0, 4.0) # default (6,4)
        %config InlineBackend.figure_formats=['png'] # default for inline viewing
        %config InlineBackend.figure_formats=['svg'] # SVG inline viewing
        %config InlineBackend.figure_formats=['pdf'] # render pdf figs for LaTeX
```

1 Introduction

The numpy ndarray is very flexible. Most of the time we work with 1D arrays which is just a list of number, similar to the Python type `list` when it contains float or integer values. The 1D ndarray is special because numpy allows array-wide operations to be performed easily. The 2D ndarray is a step up from the 1D case, giving you the opportunity thing in terms of matrices, in particular linear algebra. For Lab 2 linear algebra is not needed, we do need reshaping, which requires a 2D array.

1.1 Creating a 2D Array

Begin by creating a 2×2 matrix

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
In [18]: A = array([[1,2],[3,4]])
A
```

```
Out[18]: array([[1, 2],
               [3, 4]])
```

```
In [33]: # A small side note showing a little linear algebra:
print(A+A) # matrix add
print('-----')
print(2*A) # matrix scale
```

```

print('-----')
print(A@A) # matrix multiply using @
print('-----')
print(A-A) # matrix subtract
print('-----')
print(inv(A)) # matrix inverse

```

```

[[2 4]
 [6 8]]
-----
[[2 4]
 [6 8]]
-----
[[ 7 10]
 [15 22]]
-----
[[0 0]
 [0 0]]
-----
[[-2.  1. ]
 [ 1.5 -0.5]]

```

```

In [19]: # Also starting from a 1D array B form C
        B = array([1,2,3,4])
        C = reshape(B,(2,2))
        C

```

```

Out[19]: array([[1, 2],
               [3, 4]])

```

1.2 Matrix Stacking using hstack and vstack

Moving beyond the reshape() function there are functions for combining ndarrays. Here I briefly consider stacking 2D arrays horizontally with hstack().

```

In [29]: D = hstack((C,C,C.T))
        D

```

```

Out[29]: array([[1, 2, 1, 2, 1, 3],
               [3, 4, 3, 4, 2, 4]])

```

1.3 Being Careful with Integer Division

There is a catch in reshaping a 1D array into a 2D array. The length of B ($\text{len}(B)$) must be the product of the row size N and column size M . To the desired $N \times M$ size means you will have to do some trimming. Remember that in Python 3 integer division is performed with $\text{len}(B)//N$, as $\text{len}(B)/N$ will give you a float value.

```
In [25]: # Compare division:
L = 235765
print(L//20,L/20)
```

```
11788 11788.25
```

Suppose you have a 1D array of length L and you want to reshape the matrix into N rows and fill as many full columns as possible. How do you do it?

```
In [40]: x = rand(L)
M = 20
L_by_M = L//M
# Trim x to accomodate N = L_by_M full columns
xt = x[:L_by_M*M]
print(xt.shape)
reshape(xt, (M,L_by_M)).shape
# Below we see the reshaping operation creates a 20 x 11788 2D matrix
```

```
(235760,)
```

```
Out[40]: (20, 11788)
```

```
In [41]: # The new total length, after flattening as described below is:
reshape(xt, (M,L_by_M)).flatten().shape
```

```
Out[41]: (235760,)
```

1.4 Flatten a 2D Array to a 1D Array

For audio playback you will need to first return 2D arrays (matrices) back to 1D arrays. With numpy this is best done using `flatten()`, which takes an optional argument to control how the flattening is done. The default is row-major, which is how C stores arrays.

```
In [20]: # Use flatten() to take a 2D array back to 1D
# By default flatten is a row-major operation (C-style), e.g., flatten('C')
# flatten('F') is a column major operation (FORTRAN-style)
A.flatten()
```

```
Out[20]: array([1, 2, 3, 4])
```

```
In [23]: # Note we can chain operations and say transpose and then flatten
# Now the result is scrambled unless we use flatten('F')
A.T.flatten()
```

```
Out[23]: array([1, 3, 2, 4])
```

```
In [26]: # We can check the changes in shape along the way:
A.shape
```

```
Out[26]: (2, 2)
```

```
In [27]: A.T.shape
```

```
Out[27]: (2, 2)
```

```
In [28]: A.T.flatten().shape
```

```
Out[28]: (4,)
```

The flattened 2D to 1D array can now be played back in the notebook using either `Audio()` or a `DSP_io_stream` object from `pyaudio_helper`.