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 be using the PC sound system and Python wave file processing functions for 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.

To record using the Jupyter notebook you can use the module `pyaudio_helper.py` of scikit-dsp-comm, which is described in detail in a Scipy 2018 paper. A microphone recording example can be found in the sample notebook on the Web Site (see `Audio_Record_Playback.zip`). At a high level the `pyaudio_helper` interface abstracts the OS audio interface to the block diagram of Figure 1. To get started with Pyaudio helper you have to:

1. Establish valid microphone and speaker/headphone device indices using `available_devices()`.
2. Define a simple GUI interface to control parameters associated with the running code.
3. Write a *call back* function for processing input samples into output samples
4. Instantiate a `DSP_io_stream` object object a call the method `interactive_stream()`

Fully runnable code can be found in the sample notebook. All you have to do is discover the input/out indices for your PC. An example of devices on my Windows laptop is provided in Listing 1 below

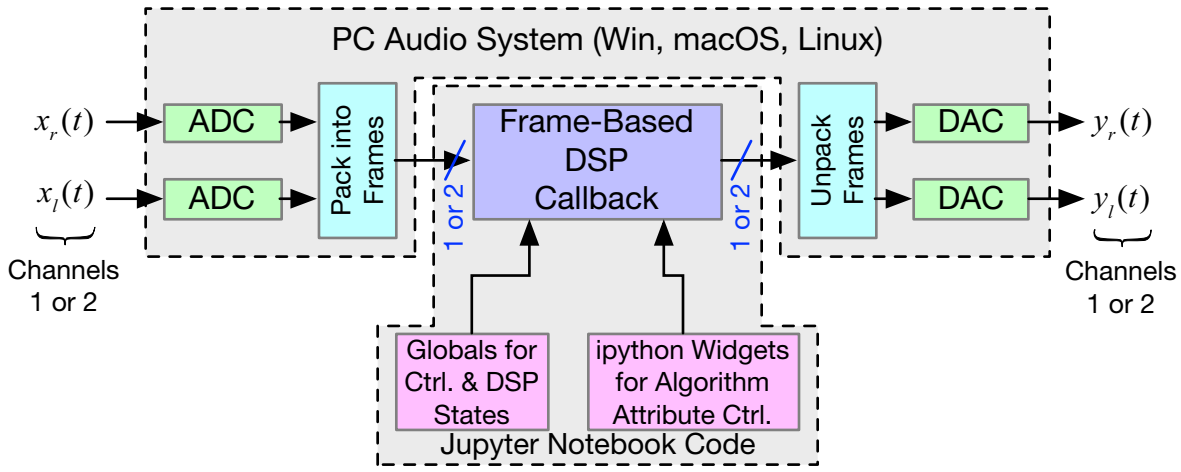


Figure 1: Block diagram of the pyaudio_helper interface, valid for Windows, macOS, and Linux.

Listing 1: A typical Windows10 listing of available devices

```
pah.available_devices()
```

```
{0: {'name': 'Microsoft Sound Mapper - Input', 'inputs': 2, 'outputs': 0},
 1: {'name': 'Microphone (5- USB Audio Device', 'inputs': 1, 'outputs': 0},
 2: {'name': 'Microphone (Realtek Audio)', 'inputs': 2, 'outputs': 0},
 3: {'name': 'Microsoft Sound Mapper - Output', 'inputs': 0, 'outputs': 2},
 4: {'name': 'Speakers / Headphones (Realtek ', 'inputs': 0, 'outputs': 6},
 5: {'name': 'Speakers (5- USB Audio Device)', 'inputs': 0, 'outputs': 2},
 6: {'name': 'Microphone (Realtek HD Audio Mic input)',
  'inputs': 2,
  'outputs': 0},
 7: {'name': 'Stereo Mix (Realtek HD Audio Stereo input)',
  'inputs': 2,
  'outputs': 0},
 8: {'name': 'Speakers 1 (Realtek HD Audio output with SST)',
  'inputs': 0,
  'outputs': 2},
 9: {'name': 'Speakers 2 (Realtek HD Audio output with SST)',
  'inputs': 0,
  'outputs': 6},
 10: {'name': 'PC Speaker (Realtek HD Audio output with SST)',
  'inputs': 2,
  'outputs': 0},
 11: {'name': 'Speakers (USB Audio Device)', 'inputs': 0, 'outputs': 2},
 12: {'name': 'Microphone (USB Audio Device)', 'inputs': 1, 'outputs': 0}}
```

To use the internal microphone on this laptop device index 2 works, or when using a USB audio interface (USB Audio Device in the listing) with external microphone, index 1 works. The output device can be laptop speakers, index 4, or the USB Audio Device headphone jack, index 5. On a macBook you will typically see just two devices, a stereo mic input, index 0, and stereo speaker/ headphone jack output, index 1. The recording app found in the Jupyter notebook is shown in Figure 2. Note you do not have to use $f_s = 11025$ Hz. If you choose to use the CD sampling rate of

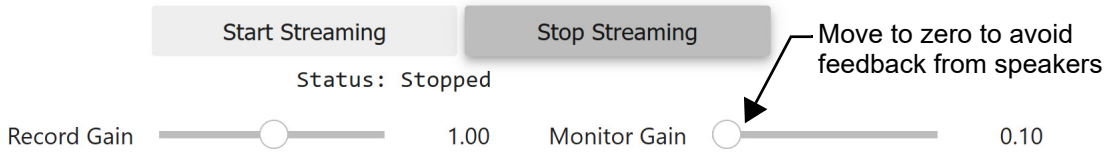
44,000 Hz that is OK. All of the block lengths will increase by a factor of 4 however.

```
fs = 11025
# 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(callback1,1,4,fs=fs,Tcapture=30)
DSP_IO_1.interactive_stream(Tsec=30)
widgets.HBox([Record_gain,Monitor_gain])
```

Choose input/output indices = 1/4

Make the capture buffer 30s

Run the stream for just 30s



Move to zero to avoid feedback from speakers

Figure 2: Microphone recording app build using `pyaudio_helper` and `ipy-widgets` for the GUI controls.

For playback from the Jupyter notebook you can use `pyaudio_helper` or there is a very nice audio *widget* available that allows direct playback via your PC sound system. Audio interface functions for Python (see Figure 1) are available in module `sigsys.py` of scikit-dsp-comm, e.g.,

- Save mono audio capture buffer in a `*.wav` file:

```
ss.to_wav('my_record_11k.wav',fs,
          DSP_IO_1.data_capture/max(abs(DSP_IO_1.data_capture)))
```

- After the recording is saved to `*.wav` you can restore it:

```
fs, x_rec = ss.from_wav('my_record_11k.wav')
```

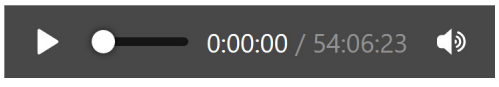
- Loop the audio recorded audio into a longer playback vector denoted `x_PB`

```
Tplay = 10
x_PB = zeros(Tplay*fs)
x_loop = pah.loop_audio(x_rec)
N_save = 0
while N_save < len(x_PB):
    x_PB[N_save:N_save+fs] = x_loop.get_samples(fs)
    N_save += fs
```

- Playback with $f_s = 11025$ sps

This function can also work with wave files directly by including the file name as a string

```
Audio(x_PB,rate=11025) # play the mono channel
```



Saving and restoring speech a vector from the note to `*.wav` files is possible using the functions shown above from found in the module `sigsys.py` of `scikit-dsp-comm`. The functions are documented in Table 1. Note these functions are simply wrapper functions of wave file interface

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

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

functions in `scipy.io`.

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 4. 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.

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. See Figure 5 for details. If the pattern is say 45 ms, repeat pre-

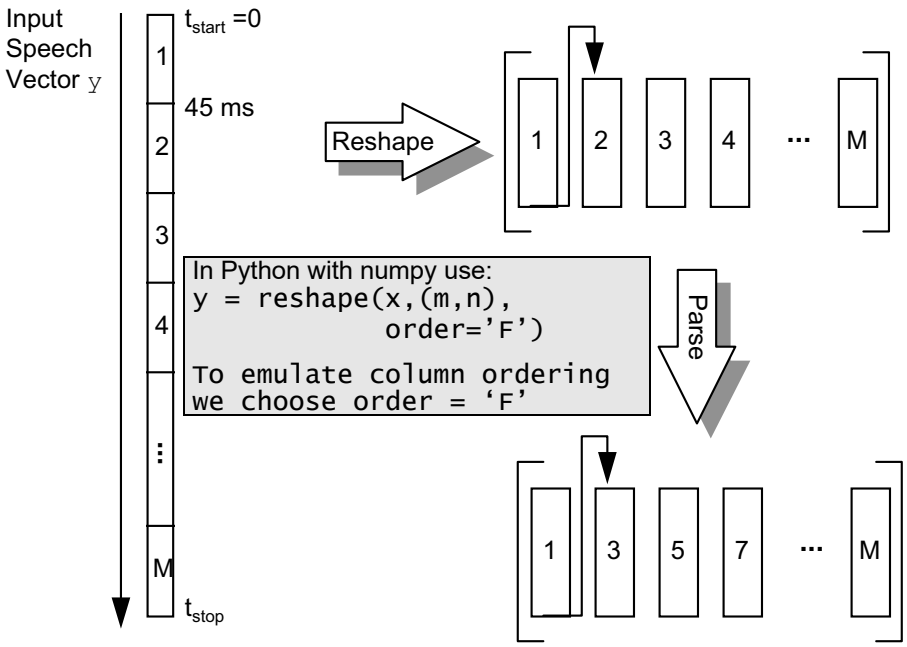


Figure 4: Butt splicing to decreasing playing time yet maintain sound pitch.

vious 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.

```
[3, 4]])
```

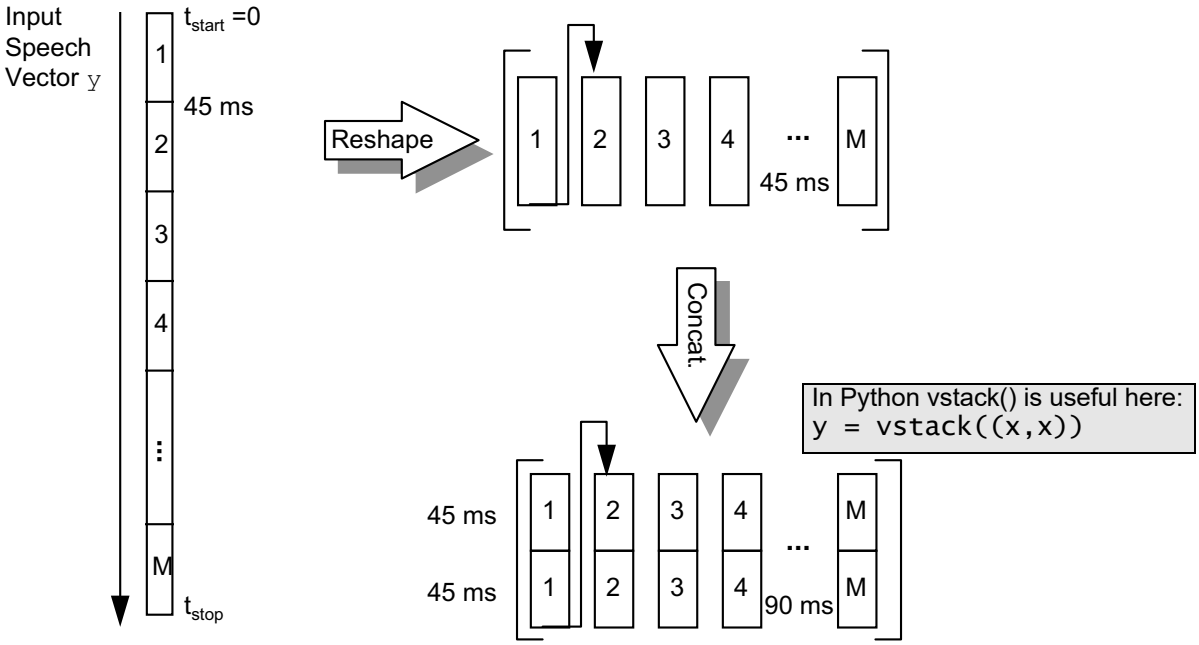


Figure 5: Butt splicing to increase playing time yet maintain sound pitch.

Understanding Numpy's `reshape()`, Transpose (`.T`), and `flatten()`

Before working with speech it is instructive to explore the numpy functions `reshape()`, transpose (`.T`), and `flatten()` or use flatten('f') with a simple integer sequence.

```
x_t = arange(20)
x_t
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

```
reshape(x_t,(4,5))
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

- The values are increasing by rows
- See what happens when we flatten the 2D array

```
reshape(x_t,(4,5)).flatten()
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

- The flattened values are ready for playback in the correct order
- Next consider what happends when we transpose:

```
reshape(x_t,(4,5)).T
```

```
array([[ 0,  5, 10, 15],
       [ 1,  6, 11, 16],
       [ 2,  7, 12, 17],
       [ 3,  8, 13, 18],
       [ 4,  9, 14, 19]])
```

- The values increase by column
- When flattened we expect the values to become scrambled:

```
reshape(x_t,(4,5)).T.flatten()
```

```
array([ 0,  5, 10, 15,  1,  6, 11, 16,  2,  7, 12, 17,  3,  8, 13, 18,  4,
        9, 14, 19])
```

- Indeed, the values are scrambled

Experiments

1. Create a short wave file using 16 bits per sample resolution, $F_s = 11.025$ ksps, 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 ZIP file under Lab 2 entitled `Audio_Record_Playback.zip`.
2. Import the wave file into the IPython/Jupyter notebook workspace using the function

   ```
   fs,y = ssd.from_wav('test_file.wav')
   ```

3. Beyond using `pyaudio_helper`, as described in experiment 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')`.
4. Verify that Python numpy plays a matrix by first flattening the matrix to a 1D array and this is done row wise. First convert the vector y into a matrix using

   ```
   >> y = reshape(x, M, N)
   >> z = y.flatten() # or simply z = reshape(x, M, N).flatten()
   ```

   where `M*N = len(x)`. Play the sound using row wise flattening. Verify that scrambled speech is heard if you play the transpose of your matrix, that is you flatten `y.T`, i.e., `y.T.flatten()`.
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 function

   ```
   >> specgram(y,FFTLENGTH,Fs) # using the minimum parameters is OK
   ```

   Recall from ECE 2610 that a spectrogram is a frequency spectrum versus time plot. For a sampling rate of 11025 Hz a good set of values is

   ```
   >> specgram(y,512,11025) # for fs = 11025 Hz
   >> specgram(y,2048,44100) # for fs = 44100 Hz
   ```

   Sample results for the original captured speech at $f_s = 11025$ Hz is shown in Figure 6.

```
specgram(x_PB,512,fs);
title(r'Normal')
ylabel(r'Frequency')
xlabel(r'Time (s)')
ylim([0,1000])
grid();
```

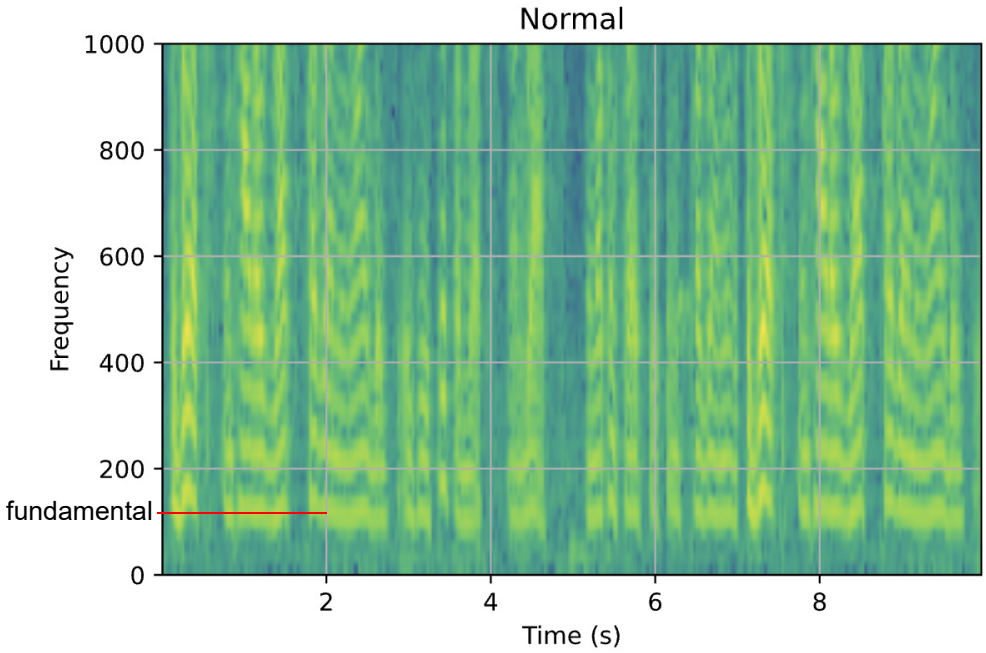


Figure 6: Sample spectrogram for a 11025 Hz capture zoomed to a 0-1000 Hz frequency range

9. Summarize your findings.
10. Prepare for instructor demo.

Bibliography/References

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