ECE 5650/4650 Python Project 1

This project is to be treated as a take-home exam, meaning each student is to due his/her own work. The exception to this honor code is for ECE 4650 students I will allow you to work in teams of two if desired. Still, teams do not talk to other teams. The project due date is no later than 5:00 PM Tuesday, November 20, 2018 (Thanksgiving week). Getting it completed earlier is recommended. To work the project you will need access to the Jupyter qtconsole or Jupyter Notebook (preferred). All the needed functions can be found in numpy, the signal module of scipy, sk_dsp_comm, and detect_peaks.py, found in the project ZIP package set1p.zip.

Introduction

In this project you are introduced to using Python’s Scipy Stack, which is a collection of Python packages for doing engineering and scientific computing. The core portion of the SciPy stack is known as PyLab. This Python DSP project will get you acquainted with portions of the Python language and PyLab and then move into the exploration of

- LCCDEs with non-zero initial conditions
- Multi-rate sampling theory
- Polyphase rate changing class
- Studying an IIR notch filter
- PyAudio_helper real-time DSP apps (LinearChirp, speech with noise and tone jamming)

It is the student’s responsibility to learn the very basics of Python from one of the various tutorials on the Internet, such as Python Basics (see the link below).

Python Basics with NumPy and SciPy

To get up and running with Python, or more specifically the PyLab (numpy and matplotlib loaded into the environment) for engineering and scientific computing, please read through the tutorial I have written in a Jupyter notebook. The PDF version of the notebook can be found at http://www.eas.uccs.edu/wickert/ece5650/notes/PythonBasics.pdf.

Problems

Causal Difference Equation Solver with Non-Zero Initial Conditions

1. In this problem you gain some experience in Python coding by writing a causal difference equation solver. You will actually be writing a Python function (def) that you will input filter coefficients \([b_0, b_1, \ldots]\) and \([1, a_1, a_2, \ldots]\), the input signal \(x[n]\), and initial conditions, \(x_i\) and \(y_i\), to then obtain the output, \(y[n]\). The starting point is
(1)

\[
\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]
\]

\[
y[n] = \frac{1}{a_0} \left[ \sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k] \right]
\]

The second line above contains the LCCDE form of interest for working this problem. There are two sum-of-products (SOP) that must be calculated for each new sample input to the system. Your responsibility is to writing the main number crunching algorithm. Adhere to the Listing 1 code template (found in a sample Jupyter notebook):

**Listing 1:** Code cell template for writing the function LCCDE.

```python
def LCCDE(b,a,x,yi=[0],xi=[0]):
    
    y = LCCDE(b,a,x,yi,xi)
    Causal difference equation solver which includes initial conditions/states on x[n] and y[n] for n < 0, but consistent with the length of the b and a coefficient arrays.

    b = feedforward coefficient array
    a = feedback coefficient array
    x = input signal array
    yi = output state initial conditions, if needed
    xi = input state initial conditions, if needed
    y = output/returned array corresponding to x

    Examples
    ========
    >> n = arange(20)
    >> x = ss.dimpulse(n)
    >> #x = ss.dstep(n)
    >> y = LCCDE([0,1/3],[1,-5/6,1/6],x,[1],[0])
    >> stem(n,y)
    >> grid();
    >> print(y)

    >> n = arange(20)
    >> x = ss.dimpulse(n)
    >> #x = ss.dstep(n)
    >> y = LCCDE(ones(5)/5,[1,-1],x,[5],[-1,-1])
    >> stem(n,y)
    >> grid();
    >> print(y)

    # Make sure the input list is converted to an ndarray
    a = array(a)
    b = array(b)
    # Initialize the output array
    y = zeros(len(x))
    # Initialize the input/output state arrays to zero
    x_state = zeros(len(b))
```

Problems
```python
y_state = zeros(len(a)-1)
# Load the input initial conditions into the
# input/output state arrays
if len(a) > 1:
    for k in range(min(len(yi),len(a)-1)):
        y_state[k] = yi[k]
if len(b) > 1:
    for k in range(min(len(xi),len(b)-1)):
        x_state[k+1] = xi[k]

# Process sample-by-sample in a loop

return y
```

In writing the main loop code consider how `x_state` and `y_state` are used:

<table>
<thead>
<tr>
<th><code>x_state</code></th>
<th><code>y_state</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x[n] )</td>
<td>( y[n-1] )</td>
</tr>
<tr>
<td>( x[n-1] )</td>
<td>( y[n-2] )</td>
</tr>
<tr>
<td>( x[n-2] )</td>
<td>( y[n-3] )</td>
</tr>
<tr>
<td>( \cdots )</td>
<td>( \cdots )</td>
</tr>
<tr>
<td>( x[n-M] )</td>
<td>( y[n-N] )</td>
</tr>
</tbody>
</table>

To update the state arrays on each loop iteration consider using the Numpy functions `roll()` and `sum()` for ndarrays.

a) Complete the function `LCCDE()`. Feel free to do all of your code development right inside an Jupyter notebook.

b) Test the code with the two examples given in the doc string of the code template. The printed listing of output values needs to be included in your report for grading purposes. Feel free to validate your code by hand calculations or other means you can devise.

c) Compare the execution speed `LCCDE()` with `signal.lfilter()` under zero initial conditions using the IPython magic `%%timeit`. An example of the setup for `LCCDE()` is shown below:

```python
n = range(20)
x = ss.dimpulse(n)
%%timeit y = LCCDE([[0,1/3],[1,-5/6,1/6]],x)
```
Multirate Systems with Python Using PyLab

2. In this first task you will do some basic signals and systems problem solving using PyLab with SciPy and the code the modules `sigsys.py` and `detect_peaks.py` found in the project ZIP package. **Note:** In Task 1 I will be guiding you step-by-step. In the remaining tasks things become more open-ended.

   a) Generate 10,000 samples of \( x[n] \). The large number of samples insures a high resolution spectral estimate.

   b) Plot \( x[n] \) for \( 0 \leq n < 50 \) using `plot(x, y)`. Label your axis accordingly.

   c) Plot the power spectrum of \( x[n] \) using

   ```
   f, Sx = ss.simple_SA(x, NFFT=2048, fs=1, window='hanning')
   ```

   Then plot in dB, that is plot \( 10 \log_{10}(S_x) \). **Note:** Here the power spectrum as defined in notes Chapter 4, \( P_{xx}(\omega) = P_{xx}(2\pi f) \), is related to \( |X(e^{j\omega})|^2 \) with additional scaling and averaging. Also, setting \( fs=1 \) means the frequency axis array \( f \) corresponds to \( \omega/(2\pi) \), which convenient for the present analysis.

   d) Next verify that the PSD spectral peaks are where you expect based on theory. To numerically find the peaks use `index = detect_peaks(Sx)` (the ZIP package contains `detect_peaks.py`), as shown in Listing 2.

**Listing 2:** Code snippet for listing the peaks found by `simple_SA`.

```python
from detect_peaks import detect_peaks
f, Sx = ss.simple_SA(x, NFFT=2048, fs=1, window='hanning')
plot(f, 10*log10(Sx))
grid();
xlabel(r'Normalized Frequency $\omega/(2\pi)$')
ylim([-30, 0])
ylabel(r'Power Spectrum $P_{xx}(\omega)$ (dB)')
idx = detect_peaks(Sx) # with threshold (Sx, mph=10**(-40/10))
    # where mph <=> maximum peak height
for i in idx:
    print('Spectrum peak at %6.4fHz of %4.2fdB ' % (f[i], 10*log10(Sx[i])))
```

**Note:** In the above code snippet, the \ is Python’s line continuation character. It is possi-
e) Using \( z_1 = ss\text{.downsample}(x,M) \) produce \( z_1[n] \) and repeat parts (b) – (d) for \( z_1 \). You need to compare the experimental results for the spectral frequency locations with theory. The amplitude values are not a concern at this point. The fact that you generated sinusoids at two different amplitudes should help you keep track of which frequency is which, in spite of any aliasing that may be present.

f) Using \( z_2 = ss\text{.upsample}(z1,L) \) produce \( z_2[n] \) and repeat parts (b) – (d) for \( z_2 \). As a result of the upsampling, modify the plot range from part (b) to \( 0 \leq n < 100 \).

g) Design the lowpass interpolation filter as indicated in the block diagram. Choose the cutoff frequency accordingly. The \texttt{firwin} function is in the SciPy signal module:

\[
b = \text{signal\text{.firwin}(numtaps, cutoff)}
\]

where here \( numtaps=65 \) and \( cutoff=2*fc/fs \), where \( f_c = \omega_c/(2\pi) \) and the sampling rate \( f_s = 1 \) at this point. Plot the frequency response magnitude and phase of this filter. Also obtain a pole-zero plot. The relevant Python functions are shown in Listing 3.

**Listing 3:** Plotting frequency response from the ground up and also plotting a pole-zero plot.

```python
f = arange(0,0.5,.001)
w,H = signal.freqz(b,1,2*pi*f)
plot(f,20*log10(abs(H)))
grid();
xlabel(r'Normalized Frequency $\omega/(2\pi)$')
ylabel(r'Magnitude Response $|H(e^{j\omega})|$ (dB)');
figure()
plot(f,angle(H))
grid();
xlabel(r'Normalized Frequency $\omega/(2\pi)$')
ylabel(r'Phase Response $\angle H(e^{j\omega})$ (rad)');
ss.zplane(b,[1],False,1.5) #Turn off autoscale if a bad root appears
```

h) Finally, process \( z_2[n] \) through the lowpass interpolation filter to produce \( y[n] \). The relevant Python function to perform filtering is:

\[
y = \text{signal\text{.lfilter}(b,a,x)}
\]

where for an FIR filter \( a=1 \). With \( y \) in hand, repeat parts (b) – (d). To compensate for the FIR filter delay, in part (b) change the plot range to \( 50 \leq n < 100 \). Comment on your final results. Note: Your lowpass filter will push the amplitudes of the upsampling images down, but the \texttt{detect\_peak} function will still find them. The argument \( mph \) can be used to ignore peaks below the minimum peak height (mph).

### A Polyphase Decimator Class

3. In this problem you will develop and test a polyphase decimator class in Python. The starting point will be a related Python class that performs FIR filtering without the use of \texttt{signal\text{.firwin}}.
nal.1.filter. Recall that a standard decimate by $M$ operation first filters the input signal, $x[n]$, with a lowpass filter having a lowpass bandwidth of $\omega_c = \pi/M$, followed by a downsample by $M$ as shown in Figure 2:

![Figure 2](image-url)  

**Figure 2:** A decimate by $M$ system implemented as a cascade of a lowpass and a downsampler.

The block diagram of a polyphase decimator can be found on text page 200, Figure 4.40 or on notes Chapter 4 page 55 (also see Figure 3 below),

![Figure 3](image-url)  

**Figure 3:** Polyphase decimate by $M$ where the $E_i(z)$ filter coefficients are related to the FIR coefficients of $H(z)$ by decimation with index phase shifting.

where the FIR filters $E_i(z)$ have impulse responses $e_i[n]$ which are obtained from the FIR filter coefficients held in array $b$ as follows:

$$ e_i[n] = b[nM + i], i = 0, ..., M - 1 $$  \hspace{1cm} (2) 

where $n$ runs over the full extent of the array $b$. In the end each FIR filter $E_i(z)$ has coefficients that are a decimated and time shifted version of the original FIR coefficients, and all the coefficients are unique across the $M$ filters. The filter design and corresponding coefficients can come from an FIR design tool such as `fir_design_helper` shown in Listing 4.

**Listing 4:** Design an equal-ripple FIR using `fir_design_helper`.

```python
import sk_dsp_comm.fir_design_helper as fir_d
M = 5
b = fir_d.fir_remez_lpf(0.8*1/(2*M),1.2*1/(2*M),0.1,60,1.0)
Remez filter taps = 67.
```

In the example above the filter order is low because the transition around the desired cutoff runs from 0.8 to 1.2 of $\omega_c$ or $\pm 20\%$. For this problem the $\pm 20\%$ transition width is OK, but you may want to experiment on your own with smaller values.
Your job is to complete the class of Listing 5 that has an init method or constructor and a filtering method.

**Listing 5:** Polyphase decimator object.

```python
class polyphase_deci(object):
    
    Polyphase decimator built from the ground up

    Mark Wickert October 2017

    def __init__(self,b,M):
        
        Initialize the object with decimated filter coefficient
        and array of sample-by-sample FIR filtering object

        Mark Wickert October 2017
        
        self.b = b
        self.M = M
        Nb = int(ceil(len(b)/M))
        # E is a 2D array used to hold the M polyphase filters in rows
        self.E = zeros((M,Nb))
        # Create a Python list to hold the M fir_filt objects used to
        # implement the polyphase filter. Here we start with an empty list
        self.E_filter_obj_list = []
        
        # Initialize the M fir_filt objects with the correct coefficients
        # You add new elements, e.g., objects here, to a list using .append()
        for k in range(M):
            
            # Write code to parse the input coefficients in self.b into
            # the 2D array E. Then use rows of E when you instantiate each
            # of the filter objects held in self.E_filter_obj_list.
            # Note __init__ does not need to return anything, it is just
            # initializing the attributes of the class for use later.

    def polyphase_decim(self,xM):
        
        Polyphase filter using the fir_filt class.
        Input M samples to produce one output

        Mark Wickert October 2017
        
        # Write code here to process M input samples through the
        # M fir_filt objects created and initialized by __init__
        
        return y
```

As an example of a completed class, consider fir_filt, of Listing 6, which is used in polyphase_dec.

**Listing 6:** The class fir_filt which performs FIR filtering one sample at a time to mimic the behavior of the filtering inside the polyphase decimator.
class fir_filt(object):
    """
    FIR filtering object that does not use signal.lfilter.
    Filter one sample at a time. numpy is used for the FIR
    sum of products calculation.
    Mark Wickert October 2017
    """
    def __init__(self,b):
        """
        Initialize the object with the b coefficients and zero the state array
        """
        self.b = b
        self.state = zeros(len(b))
    def filter_sample(self,x):
        """
        Filter one sample of the input to produce one output sample.
        The state array is also updated by restacking. A circular buffer
        would be better.
        """
        self.state = hstack((array([x]),self.state[:-1]))
        # Form a sum of products
        y = sum(self.b*self.state)
        return y

a) Complete the class polyphase_dec as described above, include also in-line comments in
   your code that is placed in the Jupyter Notebook cell provided in the project sample
   notebook.

b) Complete a reference design that implements the design of Figure 2 using an fir_filt
   object to filter one sample at a time followed by a downsample by \( M \). Yes, you will be
   throwing away \( M - 1 \) samples with this approach.

c) Test the implementations of Figure 2 and Figure 3 using the 500 sample test signal
   described below.

**Listing 7:** Generating a 500 sample test signal.

\[
M = 5
\]
\[
n = \text{arange}(0,500)
\]
\[
#x = \cos(2^\pi/100*n)
\]
\[
x = \text{ss.dimpulse}(n-100) - \text{ss.dimpulse}(n - 300)
\]
Set \( M = 5 \) and compare the two outputs to see that identical values (to within numeri-
   cal precision) are obtained.

d) Finally, compare execution time between the two implementations above, along with the
   simple Numpy/Scipy version:

\[
y3 = \text{ss.downsample}(\text{signal.lfilter}(b,1,x),M)
\]
For code timing you will use the *cell magic* capability of the Jupyter notebook. On my
3.5 year old MacBook Pro timing of the `signal.lfilter` version gives:

```
In [211]: %timeit y3 = ss.downsample(signal.lfilter(b,1,x),M)
85.7 μs ± 796 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Note you can also time a single line of code using a line magic:

```
In [213]: %timeit y3 = ss.downsample(signal.lfilter(b,1,x),M)
84.8 μs ± 569 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

On your machine the numbers will be different. Expect the sample-by-sample versions to yield much slower results. Why? Are you disappointed that the polyphase version is not faster? Keep in mind that the speed of pure Python and Numpy Python code be improved tools we are not taking the time study in problem, e.g., Cython, Numba, Xtension, and others. You can also view this exercise the opportunity to quickly prototype an algorithm that you plan to move to pure C/C++.

**Difference Equations, Frequency Response, and Filtering**

4. In this problem we consider the steady-state response of an IIR notch filter. To begin with we know that when a sinusoidal signal of the form

\[ x[n] = A \cos(\omega_0 n + \phi), -\infty < n < \infty \quad (3) \]

is passed through an LTI system having frequency response \( H(e^{j\omega}) \), the system output is of the form

\[ y[n] = A |H(e^{j\omega_0})| \cos[(\omega_0 n + \phi) + \angle H(e^{j\omega_0})], -\infty < n < \infty \quad (4) \]

In this problem the input will be of the form \( x[n] = A \cos(\omega_0 n + \phi) u[n] \), so the output will consist of both the transient and steady-state responses. For \( n \) large the output is in steady-state, and we should be able to determine the magnitude and phase response at \( \omega_0 \) from the waveform. The filter of interest is

\[
H(e^{j\omega}) = \frac{1 - 2 \cos(\omega_0) e^{-j\omega} + e^{-j2\omega}}{1 - 2r \cos(\omega_0) e^{-j\omega} + r^2 e^{-j2\omega}}
\]

where \( \omega_0 \) controls the center frequency of the notch and \( r \) controls the bandwidth.

a) Using `signal.freqz()` plot the magnitude and phase response of the notch for \( \omega_0 = \pi/2 \) and \( r = 0.9 \). Plot the magnitude response as both a linear plot, \( |H(e^{j\omega})| \), and in dB, i.e., \( 20 \log_{10}(|H(e^{j\omega})|) \).

b) Using `signal.lfilter()`, input \( \cos(\pi/3 \cdot n) \) for \( 0 \leq n \leq 50 \). Determine from the output sequence plot approximate values for the filter gain and phase shift at \( \omega = \pi/3 \). To do this you look at the change in amplitude and the change in zero crossing location. Note
that the filter still has center frequency $\omega_0 = \pi/2$. Also, plot the transient response as a separate plot by subtracting the known steady-state response from the total response. To do this note your simulation gives you the total response, so to get to just the transient you have to subtract the known from theory steady-state response from the total response.

c) Assume that the filter operates within an A/D-$H(e^{i\omega})$-D/A system. An interfering signal is present at 120 Hz and the system sampling rate is set at 2000 Hz. Determine $\omega_0$ so that the filter removes the 120 Hz signal. Simulate this system by plotting the filter output in response to the interference signal input. Determine in ms, how long it takes for the filter output to settle to $|y[n]| < 0.01$, assuming the input amplitude is one.

**Real-Time DSP Using pyaudio_helper**

5. In this problem you will process a speech files in real-time using the `scikit-dsp-comm` module `pyaudio_helper`. Two important references to get started with `pyaudio_helper` are the Scipy 2018 paper [4] and ECE 4680 lab materials also on the ECE 5650 Web Site. With PyAudio and the use of the `pyaudio_helper` module, a DSP algorithm developer can implement frame-based real-time DSP as depicted in Figure 4, below. To make effective use of

![Diagram](image)

**Figure 4**: Real-time DSP-I/O as seen through the eyes of `pyaudio_helper`.

true DSP I/O a USB audio dongle is needed, such as the $7.49$ Sabarent (mono mic input stereo headphone output) or $39.99$ Griffen iMic (full stereo input/output). A collection of these devices is not available to loan out and there is currently no expectation that student teams need to purchase one. So, for this problem you will instead use recorded audio tracks and the `loop_audio` class to produce a continuous stream of samples that can be heard on your PCs audio output (build-in speakers and/or headphones).

To get an understanding of the `pyaudio_helper` application programming interface (API) and the use of Jupyter widgets (ipywidgets), please start by reading through the Scipy 2018 paper of reference [4]. Moving forward, developing and running a `pyaudio_helper` app has three basic steps plus understanding your computer’s device configuration. Here we consider a two-channel (stereo) loop playback where the audio inputs replaced with an audio
source derived from a wave file:

- **Step 0**: First check devices available using `pyaudio_helper.devices_available()`. You are looking to see at minimum two input channels, usually microphone inputs from somewhere on a laptop lid and two Speaker/Headphone outputs. In Listing 8 below screen shot device number 1 has two inputs and device 3 has two outputs. We use these for I/O in the notebook examples as this is the configuration of the Dell laptop I am authoring this document on.

**Listing 8**: A code cell showing how to list the available audio devices.

```python
def pah.available_devices():
    {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 plug additional USB audio devices or your 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.

- **Step 1**: Define Jupyter widgets (ipywidgets) for interactive parameter control in a real-time PyAudio app; in particular we make heavy use of float sliders, but may other widgets and widget containers are available. Syntax for creating two vertically stacked sliders is shown in Listing 9.

**Listing 9**: A code cell showing how to create vertical float sliders.

```python
L_gain = widgets.FloatSlider(description = 'L Loop Gain',
                           continuous_update = True,
                           value = 1.0,
                           min = 0.0,
                           max = 2.0,
                           step = 0.01,
                           orientation = 'vertical')

R_gain = widgets.FloatSlider(description = 'R Loop Gain',
                           continuous_update = True,
                           value = 1.0,
                           min = 0.0,
                           max = 2.0,
                           step = 0.01,
                           orientation = 'vertical')

#widgets.HBox([L_gain, R_gain])
```

- **Step 2**: Write a **callback function** (in the below it is named `callback`) that performs the real-time processing using frames of signal samples (see [4]). An example is given in Listing 10.
Listing 10: A sample PyAudio callback function that happens to be named callback

```python
def callback(in_data, frame_count, time_info, status):
    global DSP_IO, L_gain, R_gain, x_loop_stereo
    DSP_IO.DSP_callback_tic()  # sets the start time when entering the callback
    in_data_nda = np.frombuffer(in_data, dtype=np.int16)
    # breakout input left and right signal sample frames
    # Note here the inputs are not used, but could be mixed with the loop
    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 use scaling to
    # increase the dynamic range to that of an int16 (16-bit signed int)
    new_frame = x_loop_stereo.get_samples(frame_count)
    x_left = 20000*new_frame[:,0]
    x_right = 20000*new_frame[:,1]
    # DSP operations here (here just gain control)
    y_left = x_left*L_gain.value
    y_right = x_right*R_gain.value
    # Pack left and right data together
    y = DSP_IO.pack_LR(y_left, y_right)
    # Scale float arrays
    # DSP, just apply slider gain
    # Save data for later analysis
    # accumulate a new frame of samples
    DSP_IO.DSP_capture_add_samples_stereo(y_left, y_right)
    # Convert from float back to int16
    y = y.astype(int16)
    DSP_IO.DSP_callback_toc()  # sets the stop time when returning from the callback
    # Convert ndarray back to bytes
    return (in_data_nda.tobytes(), pyaudio.paContinue)
return y.tobytes(), pa.event_ID, paContinue
```

- **Step 3:** Finally you create a DSP_iostream object (details in [4]) configured to use as a callback the function callback, input device 1 and output device 3, and a sampling rate
of 44.1 kHz. Figure 5 shows the code cell and an screen shot of the widgets as rendered

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

Figure 5: A code cell example for Step 3 also include a screen shot of the resulting widgets that are created when the cell is executed. Note when you first open a notebook the widgets are not displayed and you likely will see:

Error creating widget: could not find model
Error creating widget: could not find model

in the notebook. The first line of code brings in a stereo wave file and the second line instantiates an audio looping object. When instantiating the DSP_IO object it is best to set the size of the capture buffer to 0s, this avoids memory issues when in combination setting the interactive stream to run indefinitely. Conversely, when setting Tcapture > 0, it is best to set Tsec to a similar value, but not less than Tcapture.

The Jupyter notebook `Project1_pyaudio_helper_sample.ipynb`, in the project ZIP package, contains the stereo audio loop example discussed in the steps 0–3 above. The framework for parts (a), (b), and (c) of Project Problem 5 can also be found in this notebook.

When you run the three cells and then click the start streaming button, hearing is believing. What I mean is you have to listen through the playback device (speaks or headphones) to know things are really working. Actually, by analyzing the capture buffer you plot in the time and frequency domain, or both via the spectrogram (`specgram` in matplotlib). The `Project1_pyaudio_helper_sample` notebook discusses how to fill the capture buffer `DSP_IO.data_capture_left`, `DSP_IO.data_capture_left` for `numChan = 2`, or `DSP_IO.data_capture` in the case of `numChan = 1`. Screen shorts of the code cells and resulting graphics of the waveforms are given in Figure 6.
# create a time axis

```python
fs, x_in_stereo = ss.from_wav('Music_Test.wav')
x_loop_stereo = pah.loop_audio(x_in_stereo)
DSP_IO = pah.DSP_io_stream(callback,1,3,fs=44100,Tcapture=5)
DSP_IO.interactive_stream(10,2)
widgets.HBox([L_gain, R_gain])
```

```python
# create a time axis

```python
t = arange(len(DSP_IO.data_capture_left))/44100
subplot(211)
plot(t,DSP_IO.data_capture_left)
title(r'Left')
xlabel(r'Time (s)')

subplot(212)
plot(t,DSP_IO.data_capture_right)
title(r'Right')
xlabel(r'Time (s)')
tight_layout()
```

**Figure 6:** Working with the capture DSP_io_stream.data_capture buffer when 
numChan = 2 and displaying the left and right channel waveforms.

A spectrogram example can be found in Project1_pyaudio_helper_sample as well a quick look at stream callback statistics.

Now its time to write some code.

## Using the LinearChirp Class in pyaudio_helper

**a)** In this first exercise you are going to interface a pre-made software component in a 
pyaudio_helper callback to create a parameterizable linear frequency chirp on the left 
channel and a variable frequency sinusoid on the right channel. The component is 
included in Project1_pyaudio_helper_sample under Section 5a.

A linear chirp signal is created when you sweep the frequency of a sinusoidal waveform
linearly from a starting frequency, $f_{\text{start}}$, to and ending frequency, $f_{\text{stop}}$, over a time interval $T_p$. The process repeats at rate $R_p = 1/T_p$ with a sawtooth shaped waveform, i.e.,

$$\begin{align*}
\text{Figure 7:} & \quad \text{The instantaneous frequency of a linear chirp sinusoid.}
\end{align*}$$

In continuous time the signal takes the form

$$x(t) = A \cos[2\pi f_{\text{start}}t + 2\pi \mu t^2 + \theta] \tag{6}$$

which has instantaneous frequency

$$f_i(t) = \frac{1}{2\pi} \frac{d}{dt}[2\pi f_{\text{start}}t + 2\pi \mu t^2 + \theta] = f_{\text{start}} + 2\mu t \text{ Hz} \tag{7}$$

A discrete-time implementation of $x(t)$ is found in the class `LinearChirp`, Listing 11 below.

**Listing 11**: Code listing of the `LinearChirp` class which is used in Problem 5a.

class LinearChirp(object):
    
    """
    The class for creating a linear chirp signal that can be used in frame-based processing, such as PyAudio.
    """

    Mark Wickert November 2018
    """

    def __init__(self, f_start, f_stop, period=1.0, frame_length = 1024, fs=48000):
        """
        Instantiate a chirp object
        """
        self.f_start = f_start
        self.f_stop = f_stop
        self.period = period
        self.fs = fs
        self.frame_length = frame_length
        # State variables
        self.theta = zeros(self.frame_length)
        self.theta_old = 0
        self.f_ramp_old = 0
        if self.period > 0:
            self.Df = self.f_stop - self.f_start
            self.f_step = self.Df/(self.period*self.fs)

    def generate(self):
        """
        Generate an array of samples
        """
omega = 2*pi*self.f_start/self.fs
for n in range(self.frame_length):
    self.theta[n] = mod(omega + 2*pi*self.f_ramp_old/self.fs +
                        self.theta_old,2*pi)
    # Update frequency accumulator state if chirping
    if self.period > 0 and self.Df != 0:
        self.f_ramp_old = mod(self.f_step + self.f_ramp_old,self.Df)
    # Update phase accumulator state
    self.theta_old = self.theta[n]
return self.theta

def set_f_start(self,f_start):
    ""
    Change the start frequency
    """
    self.f_start = f_start
    if self.period > 0:
        self.Df = self.f_stop - self.f_start
        self.f_step = self.Df/(self.period*self.fs)

def set_f_stop(self,f_stop):
    ""
    Change the stop frequency
    """
    self.f_stop = f_stop
    if self.period > 0:
        self.Df = self.f_stop - self.f_start
        self.f_step = self.Df/(self.period*self.fs)

def set_period(self,period):
    ""
    Change the chirp period
    """
    self.period = period
    self.f_step = self.Df/(self.period*self.fs)

The Project1_pyaudio_helper_sample notebook provides a nice example of using the
LinearChirp class to statically create a chirp and a constant frequency sinusoid.

**Listing 12:** A static example of creating a linear chirp object, modifying it, then
capturing a buffer of signal samples and converting from phase to a cosine wave,
finally plotting a spectrogram.

```
# The five inputs: f_start, f_stop, period, frame_length, fs
LFM = LinearChirp(2000,5000,0.2,12000,10000)
# I then overwrite the initial stop frequency to 3 kHz and the
# period to 0.4s, and finally produce a spectrogram of the
# 12000 sample frame
LFM.set_f_stop(3000)
LFM.set_period(0.4)
x = cos(LFM.generate()) # output is phase so wrap with cos()
spectrogram(x,256,100000);
title(r'1000 to 3000 Hz Linear Chirp over 0.4s')
ylabel(r'Frequency (Hz)')
xlabel(r'Time (s)')
grid()
```
Figure 8 shows the result spectrogram is an indeed a linear chirp running from 2 kHz to 3 KHz with a period of 400 ms.

Figure 8: Spectrogram of a statically created linear chirp created using a 12000 sample frame.

Moving on, the Problem 5a task is to implement a two channel real-time audio signal generator with the left channel producing a linear chirp and the right channel producing a sinusoid, both tunable. Figure 9 shows the Step 3 code cell and widgets.

Figure 9: The Step 3 code cell and a screen shot of the widgets required for Problem 5a.

- L/R gain min = 0, max = 2, step = 0.01
- Chirp rate \( (1/T_p) \) min 0.5 Hz, max = 100 Hz, step = 0.1 Hz
- The three frequency sliders min = 10 Hz, max = 12000 Hz, step = 1.0 Hz
Test the signal generator by listening to the outputs as the sliders are varied over their ranges. I suggest setting the left right gains to zero independently to better hear the chirp in the left channel and the variable tone signal in the right. Validate the settings shown in Figure 10 by configuring the capture buffer with $T_{\text{capture}} = 5$ and $T_{\text{sec}} = 10$ and then plotting the spectrogram of the left and right channel signals. Examples of using the LinearChirp class can be found in the Project1_pyaudio_helper_sample notebook.

**Figure 10:** Linear chirp slider settings to be used in the capture buffer experiment.

**Mitigating Additive Noise on Speech Using a Tunable Lowpass Filter**

b) In this problem you investigate the use of an adjustable FIR lowpass filter to mitigate additive noise in a speech signal. The block diagram of Figure 11 shows how white noise of variance $\sigma_w^2$ is summed in with the 8 kmps speech loop and then passed through a 63-tap FIR lowpass filter, inside a single channel callback. To help you learn how to configure a linear filter the Project1_pyaudio_helper_sample notebook contains a complete real-time filter example for a tunable bandpass filter (BPF).

The steps to configure a pyaudio_helper app with a linear filter are expanded slightly from the 1, 2, 3 list. Step 1 still defines the slider widgets. Step 2 is now in two parts. Step 2a is used to obtain the initial $b$ and $a$ coefficient arrays for the filter and set up the filter initial conditions state array $z_i_{\text{BPF}}$ as shown in Listing 13. Step 2a also obtains the speech loop standard deviation, $\sigma_{\text{speech}}$, so the signal-to-noise ratio (SNR), defined as $\text{SNR}_{\text{dB}} = 10 \log_{10} \left( \frac{\sigma_{\text{speech}}}{\sigma_w} \right)$.

**Figure 11:** Improving the intelligibility of noisy speech using a 63-tap FIR lowpass inside the callback of a pyaudio_helper app.

- Initial LPF settings: $f_c = 2000$ Hz, min = 200 Hz, max = 3200 Hz, step = 1.0 Hz
- $f_s = 8$ kHz
- $\sigma_{w}^2 = \sigma_{\text{speech}}^2 / 10^{\text{SNR}_{\text{dB}} / 10}$
SNR = \frac{\sigma_{speech}^2}{\sigma_{w}^2}, is properly calibrated on-the-fly in the callback. The details can be found in Listing 14 and in project1_pyaudio_helper_sample.

**Listing 13:** Step 2a code cell listing for the BPF example, showing in particular the initial filter design and setting up the filter initial condition array which is used to maintain filter state continuity when entering and departing the callback.

```python
fs, x_rec = ss.from_wav('speech_8k.wav')
std_x = std(x_rec) # For setting SNR
# Design a bandpass filter
b_BPF, a_BPF = H_BP(8000,1000)
zi_BPF = signal.lfiltic(b_BPF, a_BPF,[0])
ss.zplane(b_BPF, a_BPF) # take a look at the filter pole-zero plot
```

Set 2b is devoted to the callback function, which now is expanded to include not only real-time filtering, but on-the-fly redesign of the filter should the GUI sliders be tweaked while code is running in real-time. Note five new global variables are needed to support the BPF: (2) GUI sliders and (3) filter related; two filter coefficient arrays and the initial conditions array. The details are in the code highlights of Listing 14.

**Listing 14:** Code cell highlights for Step 2b, the callback, for the tunable bandpass filter.

```python
global DSP_IO, x_loop, Gain, std_x, SNR
global BOF_f0, BPF_Df # remove/add globals for slider widgets
global b_BPF, a_BPF, zi_BPF #remove/add globals for filter related variables
.
.
# Note wav is scaled to [-1,1] so need to rescale to int16
x = 32767*x_loop.get_samples(frame_count)
x += 32767*std_x/10**(SNR.value/20) * randn(frame_count)
# Perform real-time DSP, e.g. a linear filter
b_BPF , a_BPF = H_BP(BPF_f0.value,BPF_Df.value)
y, zi_BPF = signal.lfilter(b_BPF,a_BPF,x,zi=zi_BPF)
```

Moving on to the actual lowpass design of Problem 5b, you begin by implementing the 63-tap lowpass and allow for on-the-fly changes in the cutoff frequency \( f_c \) via a GUI slider. To start with, the filter design is implemented using the scipy.signal function \( b = \text{signal.firwin(numtaps, cutoff, fs=8000)} \), with all frequencies in Hz. To get things started an initial filter design, based the GUI slider initial values, is designed in Step 2b along with an initial state array \( zi_{LPF} \). In Step 2b you remove/add globals for the sliders and add/remove filter related variables, then write filter redesign code, and filter using initial conditions. Finally in Step 3 you implement a code cell identical to the BPF example above. Figure 12 shows the GUI that follows the Step 3 code cell. The attributes of the \( f_c \) slider can be Found in Figure 11.
While the code is running, adjust the SNR to 0 dB (or lower if you wish) to then carefully listen to the intelligibility of speech loop audio as $f_c$ is varied. Stop the streaming and document your filter cutoff frequency and plot $|H(e^{j2\pi f_c/f_s})|$ in dB. Consider using the function `freqz_respond_list()`, which is found in both `fir_design_helper` and `iir_design_helper`. Code cell placeholders can be found in the `Project1_pyaudio_helper_sample` notebook.

**Mitigating Tone Jamming on Speech Using IIR Notch Filters**

c) In this third PyAudio problem you loop a single channel audio file, `speech_jam_8k.wav`, to provide a continuous speech signal that contains tone jamming. The objective is to remove the jamming tones using a cascade of two IIR notch filters like those studied in Problem 4 of the project. The slider controls are used to control the notch center frequency so you can interactively tune the filters while real-time filtering is taking place. You hear the impact of the filter adjustment and tune to remove the annoying jamming tones. The system block diagram is shown in Figure 13. The task is to implement the

**Figure 12:** A screen shot of the GUI following the Step 3 code cell, which most importantly has the $f_c$ slider for changing the lowpass filter cutoff frequency.

Figure 13: System block diagram for notch filtering tone jammed speech in a `pyaudio_helper` app.

block diagram in a single channel callback that provides sliders for adjusting the notch

---

**Figure 13:** System block diagram for notch filtering tone jammed speech in a `pyaudio_helper` app.
filter center frequencies. Additive noise is still present in this model, but during your experiments leave the SNR set to 40 dB. Once again a single channel callback framework is provided in `Project1_pyaudio_helper_sample` notebook. You must provide additions to Step 1, the slider controls, Step 2a where you will initialize a cascade of two instances of the pre-build notch filter found in the function

\[ b, a = ss.fir_iir_notch(f0, fs, r=0.95) \]

Since you have two notch filter to implement, I suggest using cascading them into one `b` and one `a` array using the function

\[ b, a = ss.cascade_filters(b1, a1, b2, a2) \]

This will make it easier to handle the initial conditions array. In Step 2b you modify globals and rework the filter redesign code and the filtering code. Finally your Step 3 is like 5b, except you are using the wave file `speech_jam_8k.wav`. The app GUI is shown in Figure 14.

![Figure 14](image)

**Figure 14:** A screen shot of the GUI following the Step 3 code cell, which most importantly has the two sliders for changing the notch center frequencies.

To wrap this task up run the app and carefully tune the sliders to eliminate the two jamming tones. Without further modification of notch sliders, using `fir_d.freqz_response_list([b_Notch],[a_Notch],'db',fs,Npts=4096)` or a method of your choosing, to plot the notch filter cascade frequency response as dB gain, versus frequency in Hz. These setting should reflect the slider setting you arrived at during your listening test.

Finally using `simple_SA()` and `detect_peaks()`, that was used in Problem 2, determine the location of the jamming tones in `speech_jam_8k.wav`. How close did you come by just listening and adjusting the sliders? For additional details see the placeholder cells in `Project1_pyaudio_helper_sample` notebook.
Bibliography/References


