ECE 5650/4650 Python Project 1

This project is to be treated as a take-home exam, meaning each student is to do their 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 Wednesday, November 26, 2024 (Thanksgiving week). Getting it completed earlier is recommended. To work the project you will need access to Jupyter Lab, `pyaudio_helper` and `ipywidgets` for making GUIs in Jupyter Lab. All the needed functions can be found in `numpy`, the `signal` module of `scipy` and `scikit-dsp-comm`. The project ZIP package `set1p.zip` contains sample notebooks and graphics that display in the notebooks.

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,* which imports the `numpy` and `matplotlib`. This Python DSP project will get you acquainted with portions of the Python language and then move into the exploration of

- LCCDEs with non-zero initial conditions
- Multi-rate sampling theory
- DSP components: Phase accumulators and numerically controlled oscillators
- Studying an IIR notch filter
- PortAudio in Python via PyAudio and `pyaudio_helper_old.py` for real-time DSP apps (linear chirp, speech with noise, and tone jamming mitigation with a notch filter cascade)

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 an Jupyter notebook. The PDF version of the notebook can be found at https://faculty.uccs.edu/mwickert/wp-content/uploads/sites/58/2024/07/Python_Basics_2019.pdf.

Problems

Causal Difference Equation Solver with Non-Zero Initial Conditions

1. In this Task 1 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

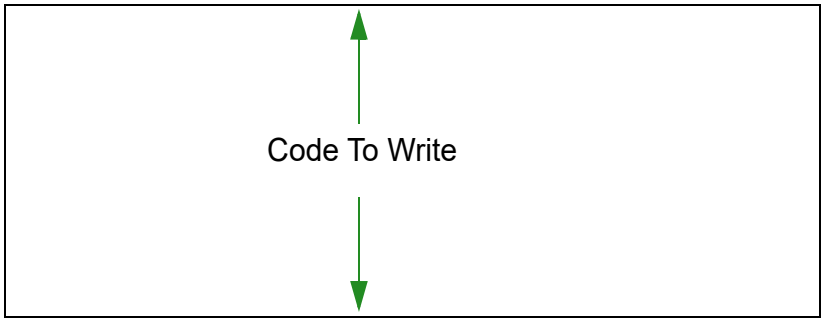
$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$
$$y[n] = \frac{1}{a_0}\left[\sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k]\right] \quad (1)$$

   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`.

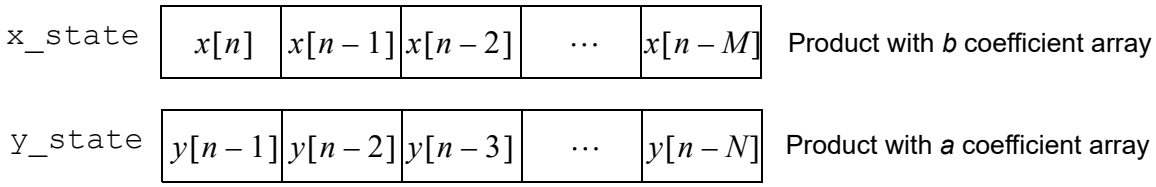
```
def LCCDE(b,a,x,yi=[0],xi=[0]):
    """
    y = LCCDE(b,a,x,yi,xi)
    Causal difference equation solver which
    includes initial conditions/states on
    x[n] and y[n] for n < 0, but consistent
    with the length of the b and a coefficient arrays.

     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))
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
 for n, x_n in enumerate(x):
```




```
return y
```

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



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:

```
n = arange(20)
x = ss.dimpulse(n)
%timeit y = LCCDE([0,1/3],[1,-5/6,1/6],x)
```

Multirate Systems Time and Frequency Domain Characterization

2. In this third 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 3 I will continue to guide you step-by-step. In the remaining tasks things become more open-ended.

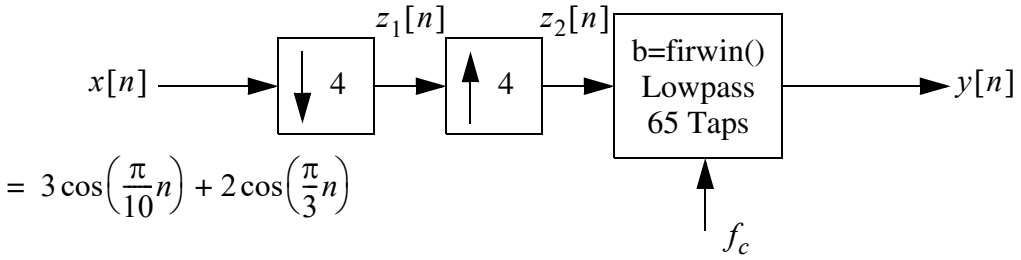


Figure 1: Downsampler/upsampler with lowpass interpolation filter block diagram.

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

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

c) Plot the power spectrum of $x[n]$ using (for now use the new module `simple_sa_new.py`)

```
f,Sx = sa_new.simple_sa(x, NS, NFFT, fs, window='hann')
```

with `NS=NFFT=2048`. 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 the scipy.signal function `index=detect_peaks(Sx)`, as shown in Listing 2.

Listing 2: Code snippet for listing the peaks found by `simple_sa` with `find_peaks`

```
n = arange(10000)
x = cos(2*pi*n/20) + 3*cos(2*pi*n/5)

#       f,Sx = ss.simple_sa(x,2048,2048,1,window='hann') # when scikit is updated
f,Sx = sa_new.simple_sa(x,2048,2048,1,window='hann')
plot(f,10*log10(Sx))
grid();
```

```
xlabel(r'Normalized Frequency $\omega/(2\pi)$')
ylim([-30,0])
ylabel(r'Power Spectrum $P_{x}(\omega)$ (dB)');
# argument height = -15 sets the detection threshold at -15 dB
idx_peaks,pk_prop = signal.find_peaks(10*log10(Sx),height=-15)
for k in idx_peaks:
    print('Spectrum peak of {:4.2f} Hz at {:8.4f}dB.'\
          .format(f[k],10*log10(Sx[k])))

Spectrum peak of 0.05 Hz at -12.9502 dB.
Spectrum peak of 0.20 Hz at  -3.4078 dB.
```

Note: In the above code snippet, the \ is Python's line continuation character. It is possible to break lines on , (commas) too.

e) Using `z1 = ss.downsample(x,M)` produce $z_1[n]$ and repeat parts (b) – (d) for `z1`. 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 frquency is which, in spite of any aliasing that may be present.

f) Using `z2 = ss.upsample(z1,L)` produce $z_2[n]$ and repeat parts (b) – (d) for `z2`. As a result of the upsampling, modify the plot range from part (b) to $0 \le n < 100$.

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

```
b = signal.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.

```
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 = signal.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 $50 \le n < 100$. Comment on your final results. Note: Your lowpass filter will push the amplitudes of the upsampling images down, but the `find_peaks` function will still find them. The argument `height=` can be used to ignore peaks below the height threshold.

Phase Accumulator for Periodic Signal Generation

3. **Omitted for Fall 2024.** In this problem you will be introduced to a signal processing component(s) frequently used in FPGA implementations where a clock source is used to increment a digital word accumulator of bit width B-bits. Here we will consider a 48-bit accumulator as the AMD FPGA product line features the DSP Slice as code signal processing block for implementing multiply and accumulate operations. Numpy has unsigned integer array types found in C/C++: `uint8, uint16, uint32,` and `uint64`. We use 48-bits of a `uint64` for our (*phase*) accumulator as follows:

```
class NCO48(object):

    def __init__(self,fcenter,fs,kc=1.0,state_init_k = np.uint64(0)):
        """
        NCO48(fcenter, fs, kc=1.0, state_init_k = 0)
        Initialize the NCO with a center frequency in Hz, the
        sampling rate, the gain kc, and initial accumulator state
        of the accumulator as k_hat ? [0,2**48 - 1).
        Note: f0 = k0 * fs/2**48 or k0 = uint64(rint(f0*2**48/fs))
        """
        self.fcenter_hat = fcenter # desired NCO center frequency
        self.fs = fs # sampling rate
        self.delta_k = np.uint64(np.rint(fcenter* 2**48/fs))
        self.kc = kc
        self.k_hat = np.uint64(state_init_k)
        self.k_old = np.uint64(0)

    def NCO48_update(self,e_in):
        """
        NCO48_update(e_in)
        Update the NCO 48-bit phase accumulator
        """
        self.k_old = self.k_hat
        self.k_hat = np.uint64(np.mod(np.rint(float(self.k_hat) + float(self.delta_k)
                               + float(self.kc*2**48*e_in)),2**48))

    def NCO48_out_sin(self):
        """
        e_out = NC48_out_sin()
        Output sin(k_hat * fs/2**48)
        """
        e_out = np.sin(2*np.pi * self.k_hat/2**48)
        return e_out
```

- Plus three additional methods:

```
e_out = NC48_out_sin()          # outputs a sinusoid sample
e_out = NCO48_out_exp()         # outputs a complex sinusoid sample
e_out = NCO48_out_square()      # outputs a squarewave sample
edge_bool = NCO48pos-edge()     # used as event trigger as in HDL programming
NCO48_set_fcenter(fcenter_new) # allows a new center frequency to be set
```

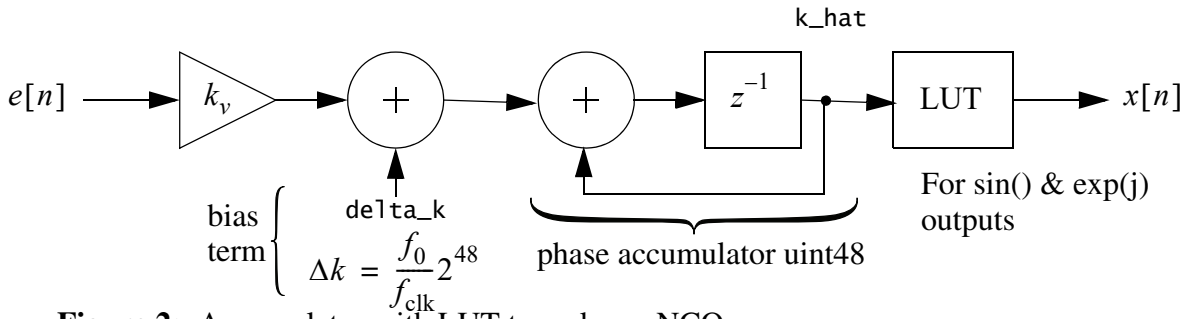


Figure 2: Accumulator with LUT to make an NCO.

The system block diagram is shown in Figure 2. Assume signal index n corresponds to clock periods or ticks.

The problems to solve: **Omitted**

a) TBD

b) TBD

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

$$x[n] = A\cos(\omega_0 n + \phi), -\infty < n < \infty \tag{2}$$

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

$$y[n] = A\left|H(e^{j\omega_0})\right|\cos[(\omega_0 n + \phi) + \angle H(e^{j\omega_0})], -\infty < n < \infty \tag{3}$$

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 ω_0 from the waveform. The filter of interest is

$$H(e^{j\omega}) = \frac{1 - 2\cos(\omega_0)e^{-j\omega} + e^{-j2\omega}}{1 - 2r\cos(\omega_0)e^{-j\omega} + r^2 e^{-j2\omega}} \tag{4}$$

where ω_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, $\left|H(e^{j\omega})\right|$, and in dB, i.e., $20\log_{10}(\left|H(e^{j\omega})\right|)$.

b) Using `signal.lfilter()`, input $\cos(\pi/3 \cdot n)$ for $0 \le n \le 50$. 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-$H(e^{j\omega})$-D/A system. An interfering signal is present at 120 Hz and the system sampling rate is set at 2000 Hz. Determine ω_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 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 3, below. To make effective use of true

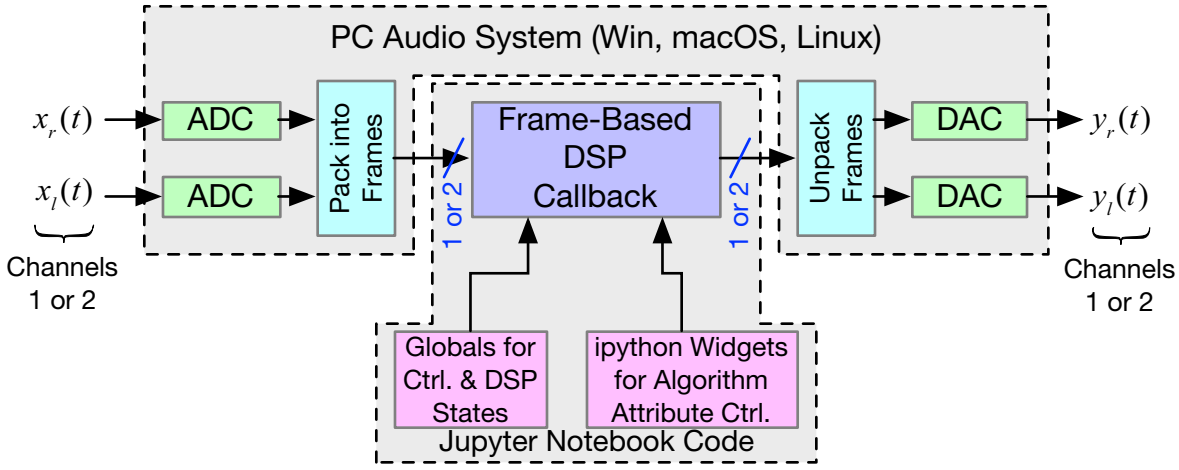


Figure 3: Real-time DSP-I/O as seen through the eyes of `pyaudio_helper`.

DSP I/O a USB audio dongle is needed, such as the $7.49 Sabarent (mono mic input stereo headphone 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 4 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 4: A code cell showing how to list the available audio devices.

```
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 plugin 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 many other widgets and widget containers are available. Syntax for creating two vertically stacked sliders is shown in Listing 5.

Listing 5: A code cell showing how to create vertical `float` sliders.

```
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])
```

Label at top of slider

Use `L_gain.value` to get slider value in code

Remove comment if you want to see the vertical widgets stacked in an HBox immediately below

- **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 exampling is given in Listing 6.

Listing 6: A sample PyAudio callback function that happens to be named `callback`

```
# L and Right Gain Sliders for looped stereo audio clip
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
    # convert byte data to ndarray
    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_mic,x_right_mic = 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)
```

globals are needed for objects, arrays, and variables that are needed to maintain state between callbacks

Scale float arrays

Minimal 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(), pah.pyaudio.paContinue
```

- **Step 3**: Finally you create a `DSP_io_stream` object (details in [4]) configured to use as a callback the function `callback`, input device 1 and output device 3, and a sampling rate

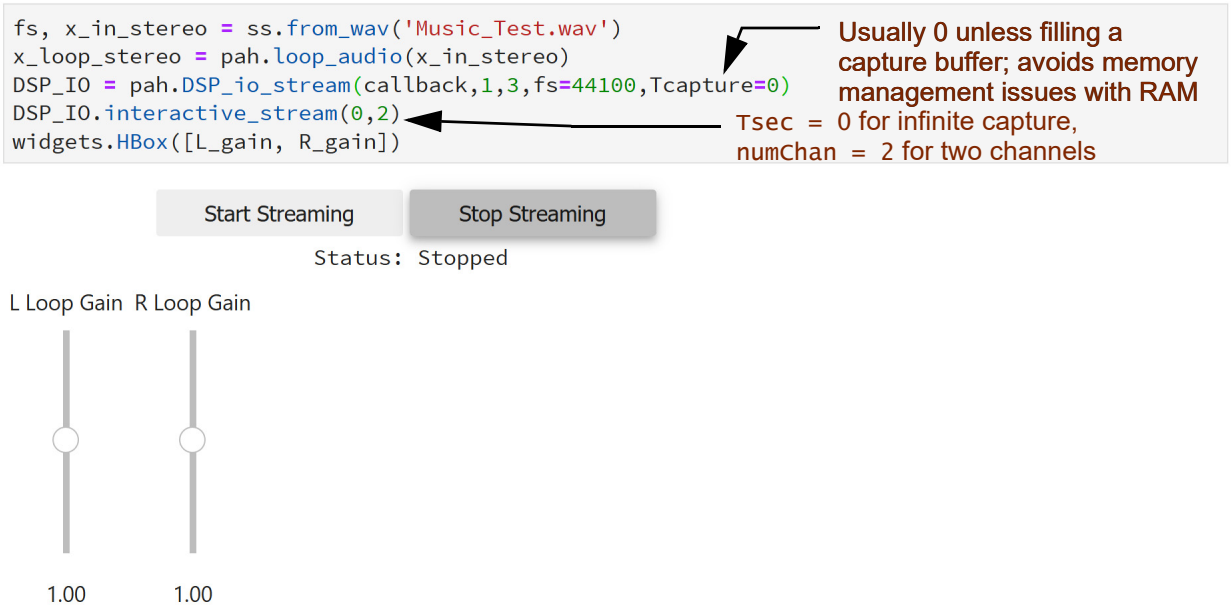


Figure 4: A code cell example for Step 3 also include a screen shot of the resulting widgets that are created when the cell is executed. Note when you first open a notebook the widgets are not displayed and you likely will see:

Error creating widget: could not find model

Error creating widget: could not find model

of 44.1 kHz. Figure 4 shows the code cell and a screen shot of the widgets as rendered 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 used in combination with 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 shots of the code cells and resulting graphics of the waveforms are given in Figure 5.

```
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])
```

```
# create a time axis
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()
```

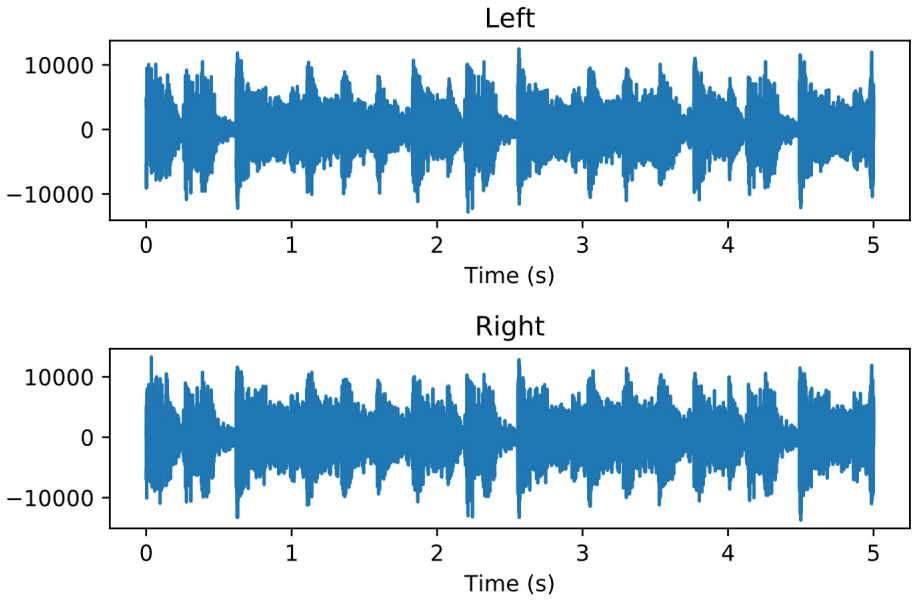


Figure 5: Working with the capture `DSP_io_stream.data_capture` buffer when `numChan = 2` and displaying the left and right channel waveforms.

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_{start}, to an ending frequency, f_{stop}, over a time interval T_p. The process repeats at rate $R_p = 1/T_p$ with a sawtooth shaped waveform, i.e.,

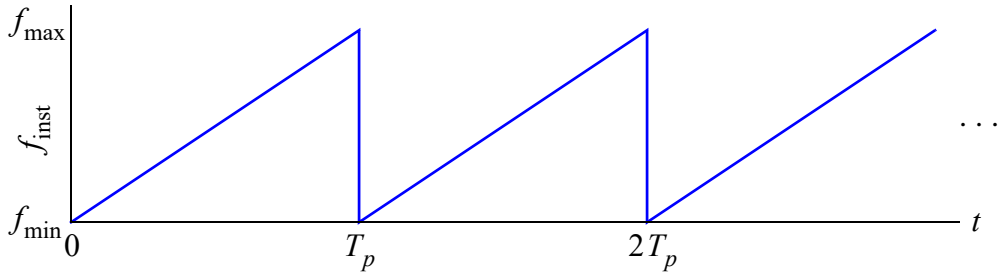


Figure 6: The instantaneous frequency of a linear chirp sinusoid.

In continuous time the signal takes the form

$$x(t) = A\cos[2\pi f_{\text{start}}t + 2\pi\mu t^2 + \theta] \tag{5}$$

which has instantaneous frequency

$$f_i(t) = \frac{1}{2\pi}\cdot\frac{d}{dt}[2\pi f_{\text{start}}t + 2\pi\mu t^2 + \theta] = f_{\text{start}} + 2\mu t \text{ Hz} \tag{6}$$

A discrete-time implementation of $x(t)$ is found in the class `LinearChirp`, Listing 7 below.

Listing 7: 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 8: 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()
specgram(x,256,10000);
title(r'1000 to 3000 Hz Linear Chirp over 0.4s')
ylabel(r'Frequency (Hz)')
xlabel(r'Time (s)')
grid()
```

Figure 7 shows the result spectrogram is an indeed a linear chirp running from 2 kHz to 3 KHz with a period of 400 ms.

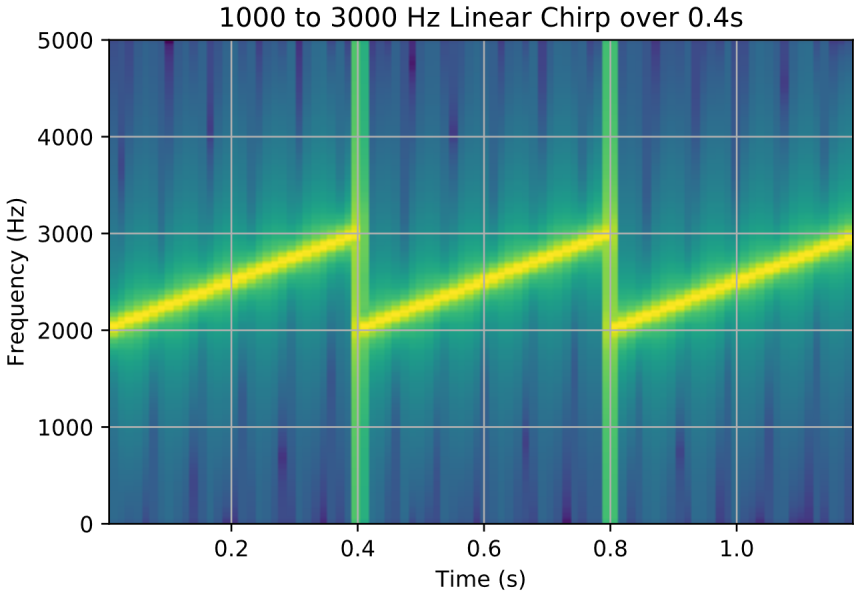
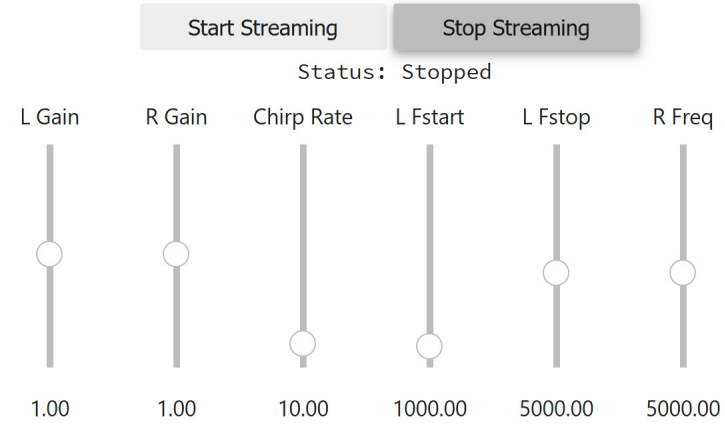


Figure 7: Spectrogram of a statically created linear chirp created using a 12000 sample frame.

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 8 shows the Step 3 code cell and widgets.

```
fs = 44100
FL = 1024
LFM1 = LinearChirp(1000,5000,0.1,FL,fs)
LFM2 = LinearChirp(5000,1000,0,FL,fs)
DSP_IO = pah.DSP_io_stream(callback,1,3,frame_length=FL,fs=fs,Tcapture=0)
DSP_IO.interactive_stream(Tsec=0,numChan=2)
widgets.HBox([L_gain, R_gain, ChRate, L_fstart, L_fstop, R_freq])
```



- 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

Figure 8: The Step 3 code cell and a screen shot of the widgets required for Problem 5a.

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 9 by configuring the capture buffer with `Tcapture = 5` and `Tsec = 10` and then

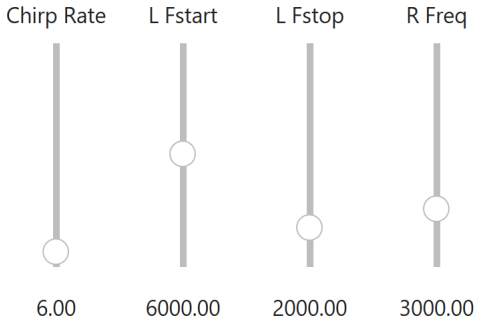


Figure 9: Linear chirp slider settings to be used in the capture buffer experiment.

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.

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 10 shows how white noise

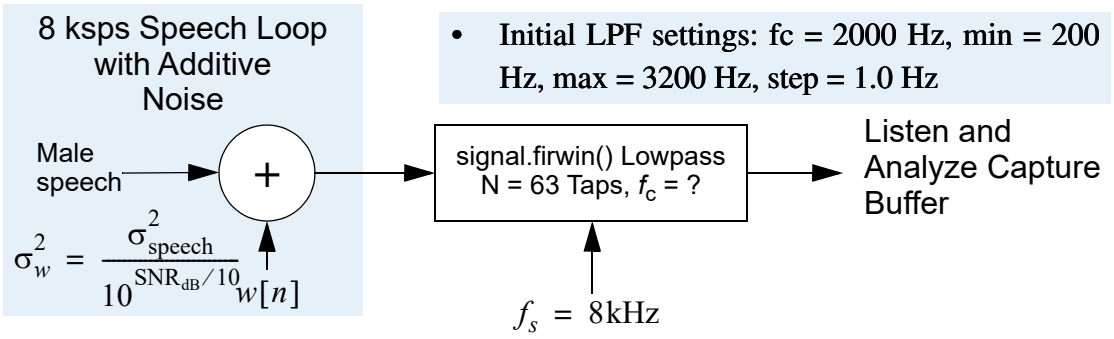


Figure 10: Improving the intelligibility of noisy speech using a 63-tap FIR lowpass inside the callback of a `pyaudio_helper` app.

of variance σ_w^2 is summed in with the 8 ksps 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 `zi_BPF` as shown in Listing 9. Step 2a also obtains the speech loop standard deviation, σ_{speech}, so the signal-to-noise ratio (SNR), defined as

SNR $= \sigma^2_{\text{speech}} / \sigma^2_w$, is properly calibrated on-the-fly in the callback. The details can be found in Listing 9 and in `Project1_pyaudio_helper_sample`.

Listing 9: 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.

```
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 10.

Listing 10: Code cell highlights for Step 2b, the callback, for the tunable band-pass filter.

```
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) * random.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 `fc` via a GUI slider. To start with, the filter design is implemented using the scipy.signal function `b = 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 11 shows the GUI that follows the Step 3 code cell. The attributes of the `fc` slider can be Found in Figure 10.

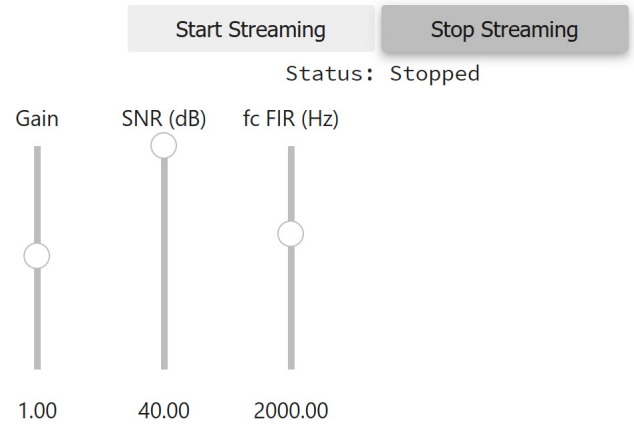


Figure 11: A screen shot of the GUI following the Step 3 code cell, which most importantly has the `fc` slider for changing the lowpass filter cutoff frequency.

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 `fc` is varied. Stop the streaming and document your filter cutoff frequency and plot $\left|H(e^{j2\pi f/f_s})\right|$ in dB. Consider using the function freqz_resp_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 12. The task is to implement the

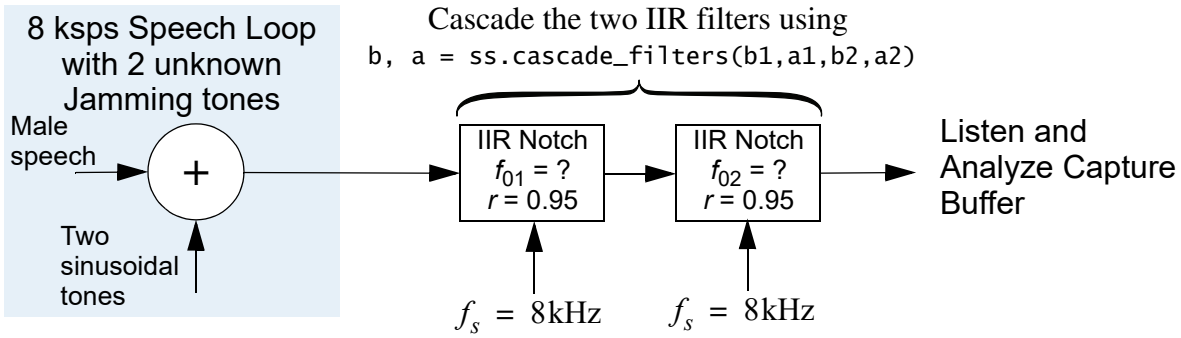


Figure 12: System block diagram for notch filtering tone jammed speech in a `pyaudio_helper` app.

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

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 13.

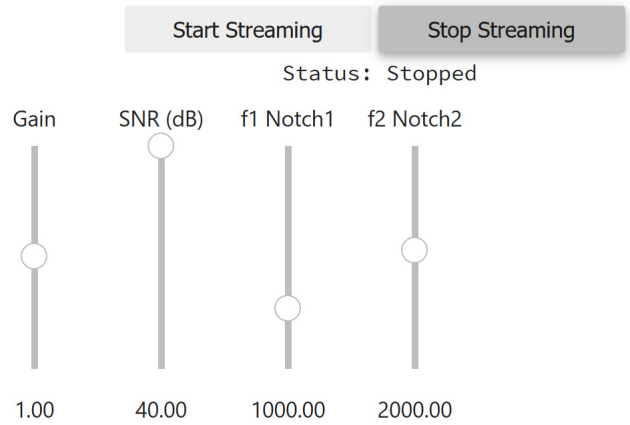


Figure 13: A screen shot of the GUI following the Step 3 code cell, which most importantly has the two sliders for changing the notch center frequencies.

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_resp_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

[1] J. H. McClellan, C.S. Burrus, A.V. Oppenheim, T.W. Parks, R.W. Schafer, and H.W. Schuessler, *Computer-Based Exercises for Signal Processing Using MATLAB 5*, Prentice Hall, New Jersey, 1998.

[2] S.K. Mitra, *Digital Signal Processing: A Computer Based Approach*, third edition, Mc Graw Hill, New York, 2006.

[3] S.K. Mitra, *Digital Signal Processing Laboratory Using MATLAB*, McGraw Hill, New York, 1999. [4] M.J. Smith and R.M. Mersereau, *Introduction to Digital Signal Processing: A Computer Laboratory Textbook*, John Wiley, New York, 1992.

[4] Mark Wickert, "Real-Time Digital Signal Processing Using pyaudio_helper and the ipywidgets," *Proceedings of the 17th Python in Science Conference*, pp. 91–98, 2018, doi: 10.25080/Majora-4af1f417-00e.

C++ Reference Guides

[5] Mikael Olsson, *C++17 Quick Syntax Reference*, third edition, Apress, New York, 2018. Current edition, C++ 20, on Amazon.

[6] Peter Van Weert and Marc Gregoire, *C++17 Standard Library Quick Reference*, second edition, Apress, New York, 2019. Current edition on Amazon.

Comments

1. The sample code sets you up with the two state arrays initially filled for the first sample output, $y[0]$. In the `for` loop created using enumerate you step through all values of the input array $x[n]$. The main calculation that takes place in the for loop is to form the *sum of products* (SOP) between each of the state arrays and the corresponding coefficient arrays. Additional considerations inside the for loop are (i) manage the state arrays using Python `roll()` after the respective SOP, (ii) make sure to always load `x_state[0]` with the present input $x[n]$ at the start of the for loop and (iii) load `y_state[0]` with most recent output $y[n]$ at the end of the for loop, (iv) manage the SOP calculations for the special cases of `len(a) > 1` and `len(a) == 1` using `if` and `else`. Crank the LCCDE by hand to obtain a few outputs to see if they agree with what your code produces. Create other examples if they are helpful in testing your code.

2. Viewing the signal in the time and frequency domains along the signal processing chain should be straight forward. Interpreting the power spectral density results theory versus measured, part (d) at each test point, is an important part of this problem. You use the `find_peaks()` function at each test point in the block diagram to find in a more exact way the location of the relevant spectral peaks on the normalized frequency axis $0 \le \omega/(2\pi) \le 0.5$.

   Obtaining the theoretical results requires you knowledge of sampling theory and multirate processing via the downsampler and upsampler processing blocks. I found it useful to create a two element frequency array `w_in = array([pi/10,pi/3])`. This array can be manipulated, such as scaling by 2π and say using `ss.prin_alias()`. At $x[n]$ the analysis needed is obvious. At $z_1[n]$ the analysis is simple as well except you have to consider the impact of aliasing. Recall the spectrum unrolls four time slower, i.e., $X(e^{j\omega}) \Rightarrow X(e^{j(\omega/4)})$, so frequencies stretch by the factor 4.

```
# The frequencies at x[n]
w_in/2/pi
array([0.05      , 0.16666667])
# The frequencies at z1[n]
ss.prin_alias(const? * w_in/2/pi,1)
array([???  , ???])
# The frequencies at z2[n]
# 4 frequency pairs; see the figure below
```

   The analysis at $z_2[n]$ can use `w_in` to form sum and difference frequency pairs to account for the 4 upsampling frequency image pairs. Consider a triangular spectral shape with some embedded spectral lines in Figure 14. Recall that relative $Z_1(e^{j\omega})$ at $z_2[n]$ you have a spec-

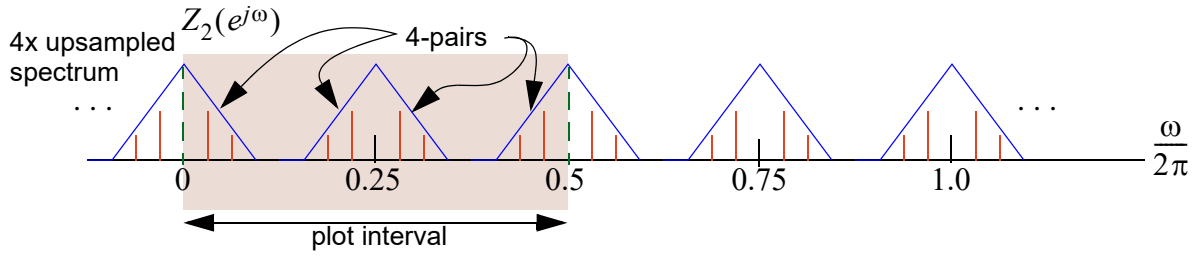


Figure 14: Upsampled by four triangular spectrum shape.

trum of the form $Z_1(e^{j4\omega})$ which makes spectrum unroll four times faster and of course is still periodic. This compresses the frequencies by a factor of 4.

With the lowpass interpolation filter finally included you will want to set the dB threshold value in `find_peaks()` lower so you can see the impact of the stopband attenuation on the image frequencies. Also, the cutoff frequency should be $\omega_c = \pi/L = \pi/4$.

3. ~~In this problem part (b) is the hardest as you have to write a polyphase algorithm from scratch in Python (Julia if you wish). I suggest *function* over *form*, meaning make it work in spite of not being fast, efficient, and elegant. Following the block diagram is best unless you have a minds eye view is clearer to you. It is best to use the supplied `shift()` function over using a modified version of `roll()` that you used in Problem 1. Over parts (a), (b), and (c) you should verify that the same output waveform is obtained independent of how the filtering and down sampling is performed. Once (b) is working the rest goes smoothly, except you will have to familiarize yourself a Julia Jupyter notebook a bit in order run the polyphase code. The sample Julia notebook should be very very helpful here.~~

4. This problem centers on the IIR notch filter discussed early in the semester and further utilized in Problem 5c of this project. The notch (b, a) filter coefficients can be obtained from the given system function $H(z)$. Alternatively you can use the function `ss.fir_iir_notch()`. In part (b) note that the notch center frequency and input frequency are different, thus the output signal $y[n]$ is not zero. In finding the transient response the total response is obtained from the simulation or driving the notch with cosine input. The steady-state is found using the development on 2-47 of the lecture notes. You will need $H(e^{j\omega_0})$ in magnitude and phase. The total response minus the steady-state response will lean just the transient response. In part (c) the problem changes to driving the input with a sinusoid at the notch center frequency. In this case the total response decays to zero.

5. Of the three parts to this problem only part (a) requires two channel in the callback. Parts (b) and (c) are similar in that they both use additive noise. Earlier in this document I eplain that there are three steps to creating a `pyaudio_helper` real-time DSP app.

   a) Getting started with the chirp generator was discussed in class with a demo of a partial solution in Jupyter Lab. Instances of the `LinearChirp` class, `LFM1` and `LFM2` are created in Step 3. By making these objects `global` in the callback function of Step 2 you can use the method `generate()` to create `frame_count` arrays of a linear chirp inside the callback. You can use the setter methods `set_f_start`, `set_f_stop`, and `set_period` to make on-the-fly attribute changes using the various `ipywidget` slider controls.

   b) This part follows logically from the tunable bandpass filter example at the start of part (b).

   c) This part follows logically from part (b) with the change up being the use of two IIR notch filters is cascade.

Set #1p Point Assignments

Problem	Points per part
1. Causal Difference Equations Solver with Non-Zero Initial Conditions	(a) 10 Complete the LCCDE() function (b) 10 Test the code using the two examples given in the DOCstring (c) 5 Compare the execution speed of LCCDE() with signal.lfilter()
2. Multirate systems simulation using Python	(a) 5 Generate a 10,000 point test signal (b) 5 Plot $x[n]$ (c) 5 Plot the PSD of $x[n]$ (d) 5 Verify the spectral peaks (e) 5,5,5,5 Downsampler repeat (b)–(d) + new code (f) 5,5,5,5 Upsampler repeat (b)–(d) + new code (g) 5,5,5,5 Filter, Mag, Phase, Pole-Zero (h) 5,5,5 Filtering, Plots(?), comments
3. Polyphase Decimator Investigation using both Python and Julia	Filter class enhancements (a)–(d) (a) 10 Complete reference design (b) 20 Complete the polyphase design for $M = 5$ compare the (a)–(b) outputs to see that they perfectly match using the two-tone input (c) 5 Julia polyphase implementation using f = DSP.FIRFilter(b1,1//5) and z_DSP = DSP.filt(f,x) to verify correct output waveform (d) 5 Execution time compare of the (a), (b), and (c) implementations
4. Difference equations, frequency response, and filtering	(a) 5,5,5 Mag. in dB and phase of notch filter for $\omega_0 = \pi/2$, $r = 0.9$ (b) 10 Filter gain and phase at $\omega_0 = \pi/2$; trans. resp. $y[n] - y_{ss}[n]$ (c) 5,5,5 Design for $f_c = 120$ Hz with $f_s = 2000$ Hz; remove steady-state; find settling time
5. pyaudio _helper problem	(a) 10,5,10,10 Slider config., Callback code, Chirp spectrogram, 3k constant frequency spectrogram (b) 5,5,5,5 Slider config., Callback code, Best f_c, Plot of $\left\|H(e^{j2\pi f/f_s})\right\|$ (c) 10,5,10,10 Slider config., Callback code, f_1 and f_2 with notch plot, Spectrum estimation with peak search for peaks.
Total	25 + 95 + 40 + 40 + 90 = 290