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 23, 2020 (Thanksgiving week). Getting it completed earlier is recommended. To work the project you will need access to Jupyter Lab and the pyaudio_helper or sounddevice_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. 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,* which imports the `numpy` and `matplotlib`. 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
- Transform domain filtering in C++ via vs code & GNU compilers, e.g., MSYS2 or similar
- Multi-rate sampling theory
- Studying an IIR notch filter
- PortAudio in Python via PyAudio and pyaudio_helper or Sounddevice and sounddevice_helper for 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 an 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 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

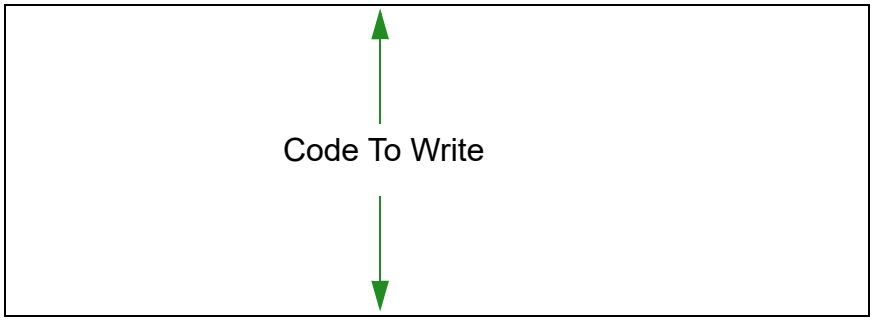
$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$
$$y[n] = \frac{1}{a_0}\left[\sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k]\right] \quad (1)$$

   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`.

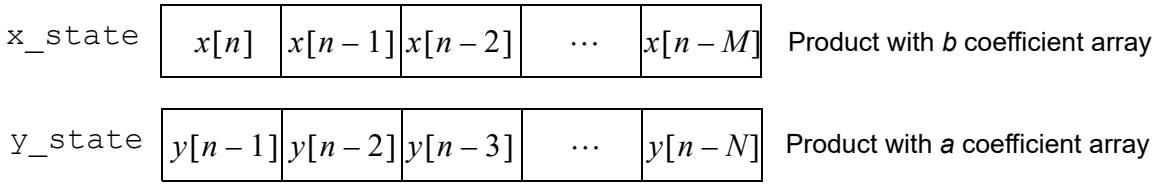
```
def LCCDE(b,a,x,yi=[0],xi=[0]):
    """
    y = LCCDE(b,a,x,yi,xi)
    Causal difference equation solver which
    includes initial conditions/states on
    x[n] and y[n] for n < 0, but consistent
    with the length of the b and a coefficient arrays.

     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
```




```
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)
```

Frequency Domain FIR Filtering in C++

2. In this task you explore transform domain filtering as first described in Chapter 7 p. 7–53 of the course notes. The primary object is to bench mark a frames-based time-domain FIR, pro-

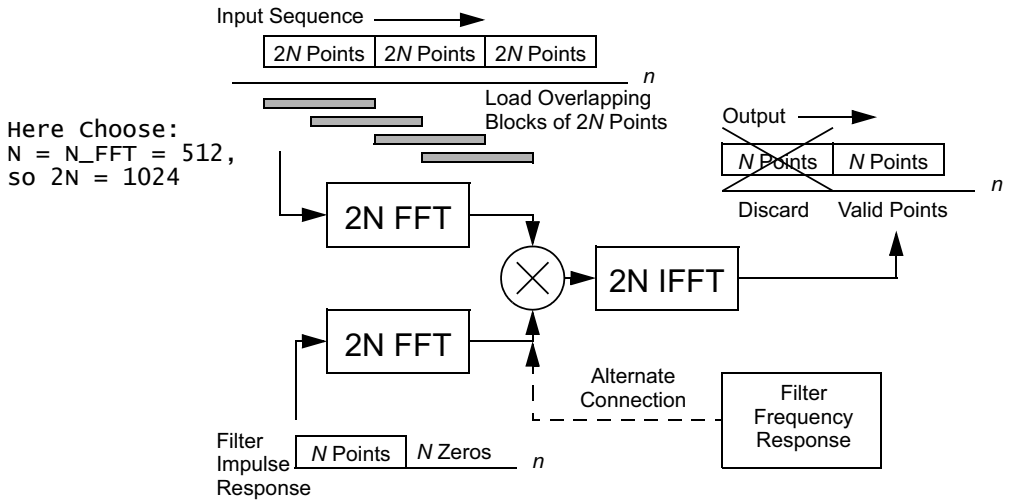


Figure 1: Transform domain FIR filtering using the overlap-and-save method.

vided, with your implementation of a transform domain implementation, using the same frame length. A Python implementation of overlap-and-save (OS) filtering is given over Chapter 7 notes pages 7–55—7-56. The function listing is repeated here in Listing 2.

Listing 2: Python implementation of OS.

```
def fft_os_fir(x,N_fft,b):
    """
    Overlap and Save FIR filtering

    y = fft_os_fir(x,N_fft,b)

    x = real input signal ndarray
    N_fft = FFT half length, i.e. overlap length N
    b = real FIR  filter coefficients of length <= N_fft

    y = filtered output (real as input is real)

    Mark Wickert October 2021
    """
    b_zp = zeros(2*N_fft) # zero pad coeff. array
    b_zp[:len(b)] = b # place coeff. at the start
```

```
    B_zp = fft.fft(b_zp) # transform filter coeff.
    N_out_buf = len(x)//(N_fft) # Number of output buffers
    y = zeros(N_fft*N_out_buf) # define the output array
    x_state = zeros(N_fft) # state array to hold past inputs
    for k in range(N_out_buf):
        # Fill 2*N_fft array with old and new subarrays
        x_in = hstack((x_state,x[k*N_fft:(k+1)*N_fft]))
        Y_all = B_zp*fft.fft(x_in) # freq. domain filter
        y_all = fft.ifft(Y_all) # back to time domain
        # save the upper index points real-part
        y[k*N_fft:(k+1)*N_fft] = y_all[N_fft:].real
        x_state = x[k*N_fft:(k+1)*N_fft]
    return y
```

This function, `fft_os_fir(x,N_fft,b)`, will serve as a reference design for your C/C++ implementation developed in this problem. A complete C++ code base is provided in the Project 1 ZIP package denoted `Proj1_Prob2.zip`. Figure 2 depicts what is provided in the Problem 2 directory tree. For starters there is a complete time-domain FIR filtering class

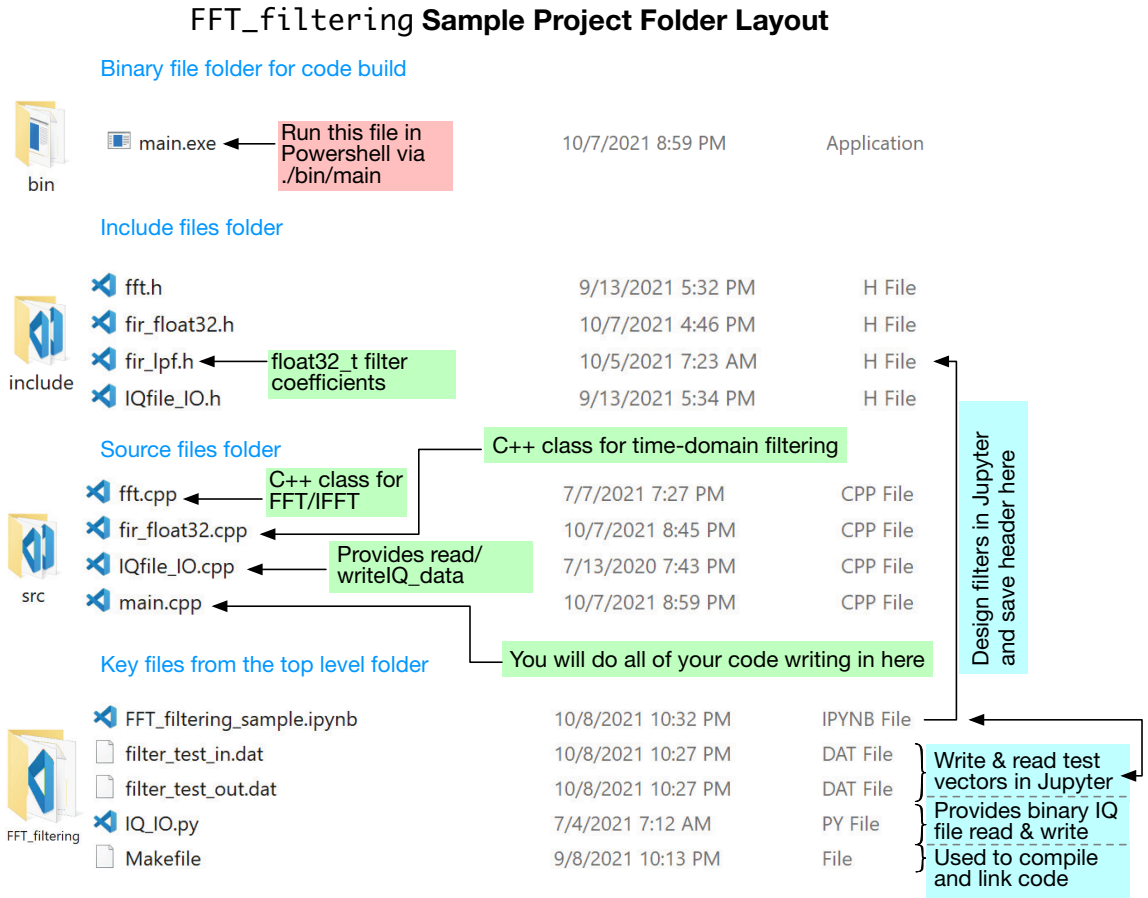


Figure 2: Project folder layout of `FFT_filtering` which enables time-domain and frequency domain filtering.

reference design (`fir_float32.cpp`/`fir_float32.h`) that you will use for code timing comparisons to your transform-based design. There is an FFT class (`fft.cpp`/`fft.h`) which provides methods for forward and inverse FFTs using a radix2 (power of two length) algorithm,

a module that contains binary file read and write functions (`IQfile_IO.cpp/IQfile_IO.h`) for complex waveform signals, a makefile (`MAKE`) for compiling and linking using the GNU compilers, and finally a main module (`main.cpp`). The main module contains completed frames-based FIR filtering code and example code on the use of the FFT class. Binary file input/output examples and a code timing example. There is also a Python module (`IQ_IO.py`) which complements the C++ file input/output to allow test vectors generated in Python to be exported to the C++ environment and then brought back into Python and used in Jupyter Lab.

The system block diagram showing signal flows for the time-domain and frequency domain filter implementations is shown in Figure 3. The left side of this is complete and imple-

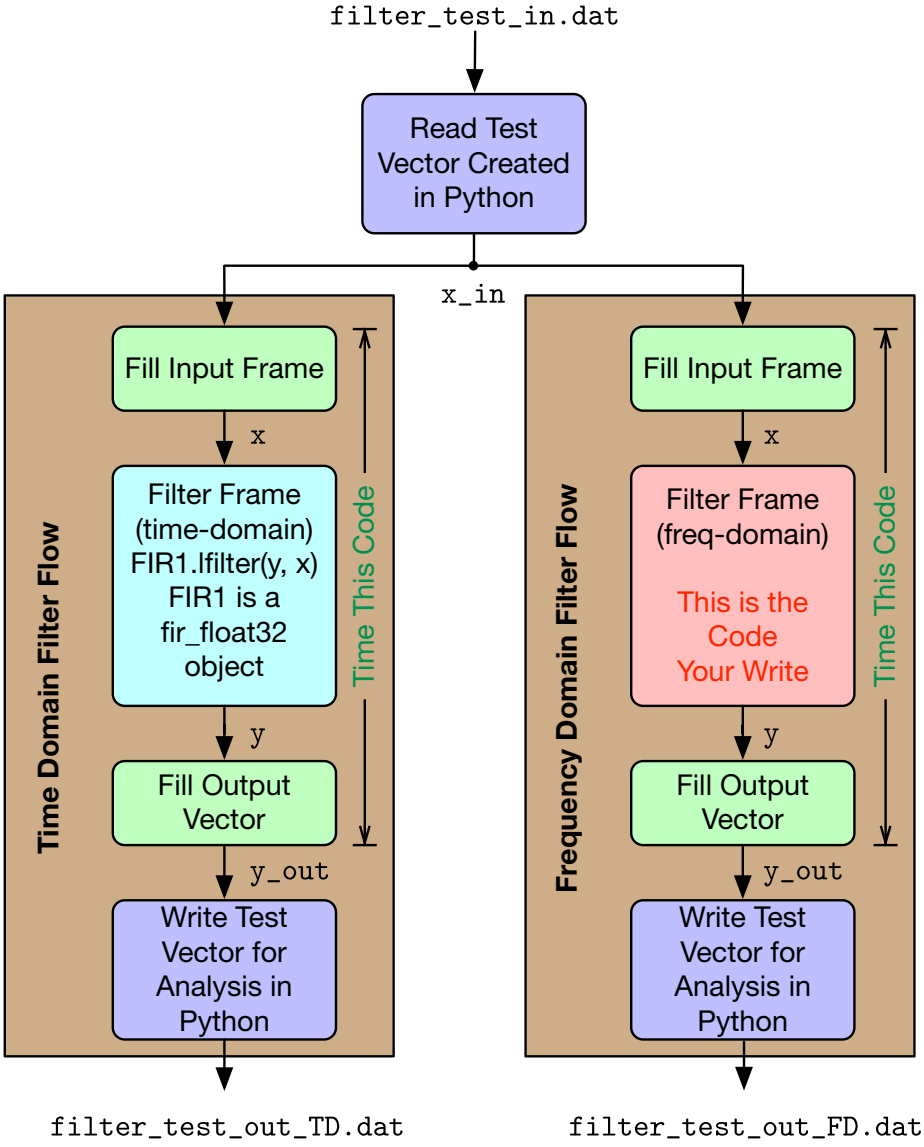


Figure 3: The signal flow block for parallel simulation of time-domain and frequency-domain filtering schemes.

mented in the main module. A place holder can be found in `main.cpp` for the right side of the figure. The next subsection get into coding details and how you will make the transition from C to helpful features of C++.

C and C++ Coding Details

You will definitely utilize your C programming skills and augment them with some new understanding of what C++ adds to C. The likely new concepts are working with objects, such as the `vector` class of the standard template library (STL) and the `fft` and `fir_float32` classes you see in Figure 2, and how to write output to the console using `std::cout`. Understand that C++ is a superset to C. The sample code found in `main.cpp`is provided in Listing 3 to point out some C++ coding styles that are different from C.

Listing 3: The main module, `main.cpp`, sample code with line numbers added.

```
1  #include <cstdint>
2  #include <cmath>
3  #include <vector>
4  #include <complex>
5  #include <chrono>
6  #include <iomanip>
7  #include "IQfile_IO.h"
8  #include "fft.h"
9  #include "fir_float32.h"
10 #include "fir_lpf.h" // bring in filter coefficients via header file
11
12 const int N_frame = 512;
13 // Define custom data types
14 using float32_t = std::float_t;
15 using complex64_t = std::complex<float>;
16 using namespace std::literals::complex_literals;
17 // double: i, float: if, long double: il, e.g., 1.0f + 2.0if
18 // makes a complex64_t containing 1.0 + j*2.0. The use of 'f'
19 // insures that 1.0f is a float32_t value rather than a double
20 // (64 bit float)
21
22 int main() {
23   // Instantiate and fir_float32 filter object for time domain
24   // filtering using coefficients input via header file which
25   // defines variable h_FIR and M_FIR. the header file is generated
26   // using the Python module sk_dsp_comm.coeff2header, following
27   // filter design using function in sk_dsp_comm.fir_design_helper.
28   fir_float32 FIR1(h_FIR, M_FIR);
29   // Create some working signal arrays using the C++ vector
30   // container templated class. To interface with binary file I/O
31   // provided in Python we a provided complex IQ array interface.
32   // This means the working arrays will be both complex64_t
33   // = float32_t + j*float32_t and just float32_t. The C++
34   // templates (i.e., decaring using <data_type>)
35   std::vector<complex64_t> x_in; //will hold possibly complex values, from Python
36   std::vector<float32_t> x(N_frame); // input working vector
37   std::vector<float32_t> y(N_frame); // output working vector
38   // Read test data created in Python and saved in a binary file
39   // into the comple vector x_in
40   readIQ_data("filter_test_in.dat",x_in); // x_in is resized in the function call
41   int k_max = x_in.size()/N_frame;
```

```
42   std::vector<complex64_t> y_out(k_max*N_frame); // Returned vector matches input
length
43
44   // ------------- Begin Time-Domain Signal Flow Frames Loop -------------
45   // Process the samples of x_in in frames/buffers of length N_frame.
46   // The outer loop in the frame processing loop
47   auto start1 = std::chrono::high_resolution_clock::now();
48   for (int k=0; k < k_max; k++) {
49     // Fill the input working vector
50     for (int n=0; n < N_frame; n++) {
51       x[n] = x_in[n+k*N_frame].real();
52     }
53     // Filter one frame at a time
54     FIR1.lfilter(y, x);
55     // FIR1.reset_state(); // see transient if we reset the state after each
56     // Transfer the frame output to the complex array y_out that will be written
57     // to a binary file that will be read back into Python.
58     // We set the imaginary part to zero, i.e., y_out.imag = 0.0f.
59     for (int n=0; n < N_frame; n++) {
60       y_out[n+k*N_frame] = y[n] + 0.0if;
61     }
62   }
63   auto stop1 = std::chrono::high_resolution_clock::now();
64   // ------------- End Time-Domain Signal Flow Frames Loop -------------
65
66   // Write the test data in y_out to a binary file
67   // that can be read into Python for analysis
68   writeIQ_data("filter_test_out_TD.dat",y_out);
69   // ------------- End Time-Domain Signal Flow File Write --------------
70
71   // ------------- Begin Freq-Domain Signal Flow Frames Loop -------------
72
73
74   // Write your code for overlap and save here
75
76
77   // ------------- Begin Freq-Domain Signal Flow Frames Loop -------------
78
79   // ---------------------------------------------------------------------
80   // Some FFT Testing code for the class radix2fft
81   // An fft object, fft1 is instantiated for an 8 point
82   // transform as shown below:
83   radix2fft fft1(8);
84   // The FFT class operates on std::vector<complex64_t> vectors
85   // to perform an "in-place" FFT/IFFT, i.e. fft1.fft(x2); or fft1.ifft(x2);
86   // In-place means the input vector is over written by the output values.
87   std::vector<complex64_t> x1{1,0,0,1,0,0,0,0};
88   std::vector<complex64_t> x1_copy;
89   std::vector<complex64_t> X1;
90   for (int k=0; k < x1.size(); k++) {
91     x1_copy.push_back(x1[k]);
92     X1.push_back(x1[k]);
93   }
94
95   auto start2 = std::chrono::high_resolution_clock::now();
96   fft1.fft(x1_copy);
97   fft1.fft(X1);
98   auto stop2a = std::chrono::high_resolution_clock::now();
99   fft1.ifft(x1_copy);
```

```
100  auto stop2b = std::chrono::high_resolution_clock::now();
101
102  std::cout << "        x1         X1=fft(x1)        ifft(fft(x1))"
103            << std::endl;
104  std::cout << std::fixed << std::setprecision(4) << std::showpos;
105  for (int m=0; m<x1.size(); m++) {
106    std::cout << x1[m] << " " << X1[m] << " " << x1_copy[m] << std::endl;
107  }
108  // ------------- End FFT Processing test code ------------------
109
110  // Display timing results for the FIR time domain filter
111  auto duration1 = std::chrono::duration_cast<std::chrono::microseconds>(stop1-
start1);
112  std::cout << std::endl;
113  std::cout << "////////////////////////////////////////////////"
114            << std::endl;
115  std::cout << "FIR (time domain) execution time = " << duration1.count()
116            << "us" << std::endl;
117  std::cout << "Done!" << std::endl;
118
119  return 0;
120}
```

From the code listing notice that the time-domain and frequency-domain signal flows corresponds to line numbers 44–69 and 71–77, respectively. The frequency domain code is just a skeleton at present, as this is the code you must write. Significant observations from the code:

- Lines 14 and 15 declare custom data types `float32_t` and `complex64_t`, which correspond single precision floating point real valued numbers and single precision floating point complex valued numbers; both of these these types are at the heart of the code you write in this task
- Working with arrays you need to take advantage of the STL `vector` class, e.g. in lines 35, 36, 37 and 42; Using the template capability arrays of different data types can easily created and access to the arrays using standard C notation (i.e. `x[n]`) is possible; Read the comments in this area of the code to learn more
- Lines 28 and 83 show you how to create an instance of a class, known as an object of type `fir_float32` and `fft`, respectively; note object creation typically involves the use of one or more parameters, e.g., a pointer to an array of filter coefficients, the number for filter coefficients, or the number points in the FFT
- Functions associated with a class (known as class *methods*) are similar to functions, except the instance variables of the class are readily available in defining what the functions do, so no need to pass this information through the function call
  - In the case of the fft class the methods are simply `fft()` and `ifft()` for in-place calculation of the transform; STL vectors of type `complex64_t` are passed in and returned by reference
  - Note: The vector lengths supplied to the object `fft1` must match the length input when the `fft` object is created, i.e., `fft1.fft(x1_copy)` in line and `fft1.ifft(x1_copy)` in line

must be of the same length used to create the object, i.e., `radix2fft fft1(8)`

– Since you are filtering a real signal, when you write values from a complex array to the output array you want to keep just the real part, i.e. `x1_copy.real()`

a) Develop and implement the overlap and save algorithm in C++ patterned after the Python function `fft_os_fir()`. The frame length should be 512. Test the function using an qual-ripple FIR design having passband ripple of 0.1 dB, stopband attenuation of 70 dB with $f_{\text{pass}} = 5\,\text{kHz}$ and $f_{\text{stop}} = 6\,\text{kHz}$. Assume a sampling rate of $f_s = 48\,\text{kHz}$. As an initial test waveform input a 1 kHz sinusoid of 1024 samples (two frames) and compare the outputs from the time-domain FIR code base and your code base. Should they be the same?

b) Compare the timing of the two schemes as depicted in Figure 3. The test vector length run though the filters should have length $4 \times 512 = 2048$ or four frames. Also obtain timing with an without compiler optimization turned on. Optimization is turned using a line in the makefile by changing the line

```
CXX_FLAGS    : -std=c++17 -ggdb
```

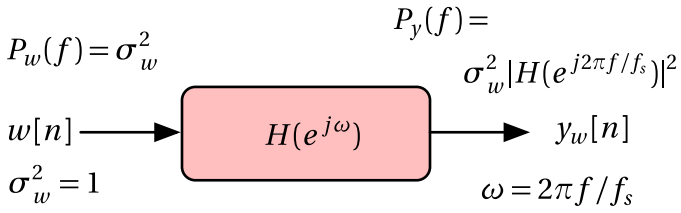
to

```
CXX_FLAGS    : -std=c++17 -ggdb -O3
```

c) Use a noise test to estimate the filter frequency response magnitude squared, $|H(e^{j\omega})|^2$ and plot it in dB ($10\log_{10}(\ )$ as P_y has power units). We use the fact that the power spectrum at the output of an LTI system driven by white noise is

$$P_y(f) = \sigma_w^2 |H(e^{j2\pi(f/f_s)})|^2 \tag{2}$$

where σ_w^2 is the variance of the white noise at the filter input. See Figure 4 for details.




```
# Create a white noise test vector
var_w = 1
w = sqrt(var_w)*randn(1000000)
# Save to an IQ binary file
IQ_IO.writefile_IQ_frame(w,'filter_noise_test_in.dat')

# As a test, filter the test vector in Python
wf = signal.lfilter(b,1,w)
# In reality wf is obtained from the C++ code
# Estimate the power spectral density
Py,fy = ss.my_psd(wf,2**10,10)
```

Figure 4: Estimating frequency response by driving the a filter with white noise.

Multirate Systems with Python Using PyLab

3. 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.

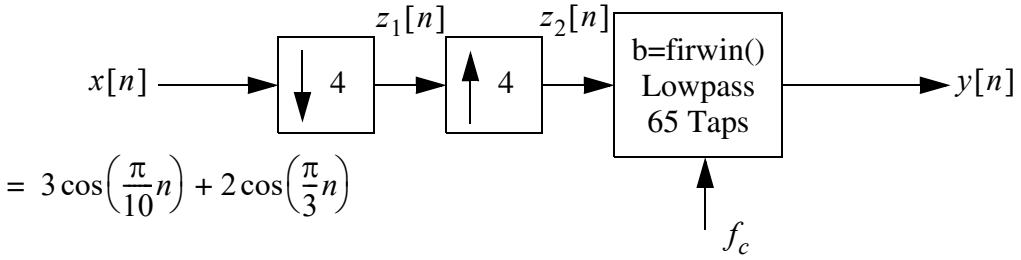


Figure 5: Downsampler/upsampler with lowpass interpolation filter block diagram.

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

```
f,Sx = ss.simple_sa(x, NS, NFFT, fs, window='hanning')
```

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 4.

Listing 4: Code snippet for listing the peaks found by `simple.SA`.

```
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='hanning')
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 5.

Listing 5: 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.

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

$$x[n] = A\cos(\omega_0 n + \phi), -\infty < n < \infty \tag{3}$$

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

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

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 ω_0 from the waveform. The filter of interest is

$$H(e^{j\omega}) = \frac{1 - 2\cos(\omega_0)e^{-j\omega} + e^{-j2\omega}}{1 - 2r\cos(\omega_0)e^{-j\omega} + r^2 e^{-j2\omega}} \tag{5}$$

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 6, below. To make effective use of true

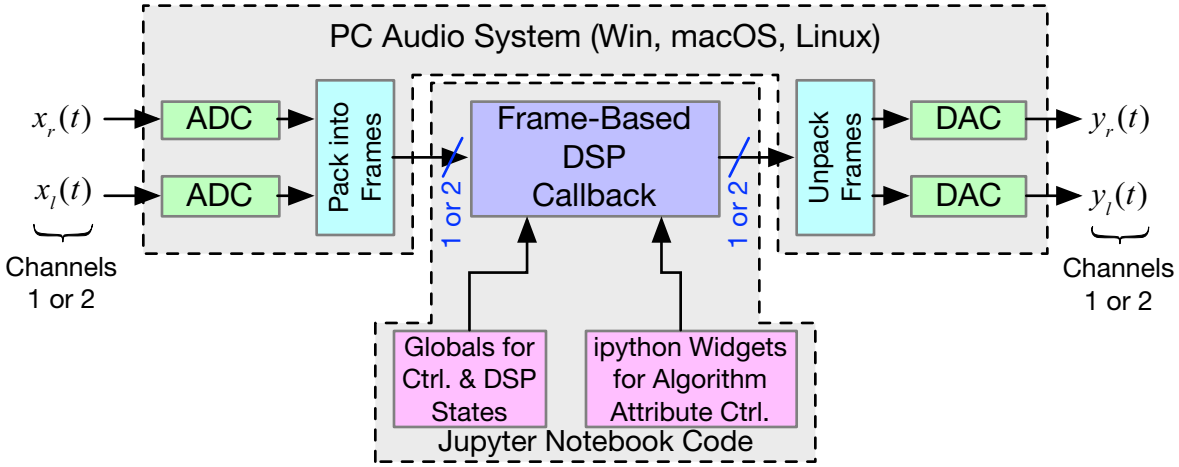


Figure 6: Real-time DSP-I/O as seen through the eyes of `pyaudio_helper`.

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 6 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 6: 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 7.

Listing 7: 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 8.

Listing 8: 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 of 44.1 kHz. Figure 7 shows the code cell and a screen shot of the widgets as rendered in

```
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])
```

Usually 0 unless filling a capture buffer; avoids memory management issues with RAM

`Tsec` = 0 for infinite capture,
`numChan` = 2 for two channels

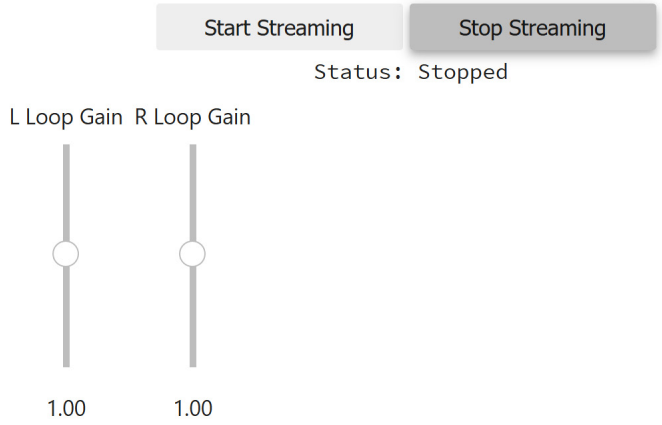


Figure 7: A code cell example for Step 3 also include a screen shot of the resulting widgets that are created when the cell is executed. Note when you first open a notebook the widgets are not displayed and you likely will see:

Error creating widget: could not find model

Error creating widget: could not find model

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 8.

```
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()
```

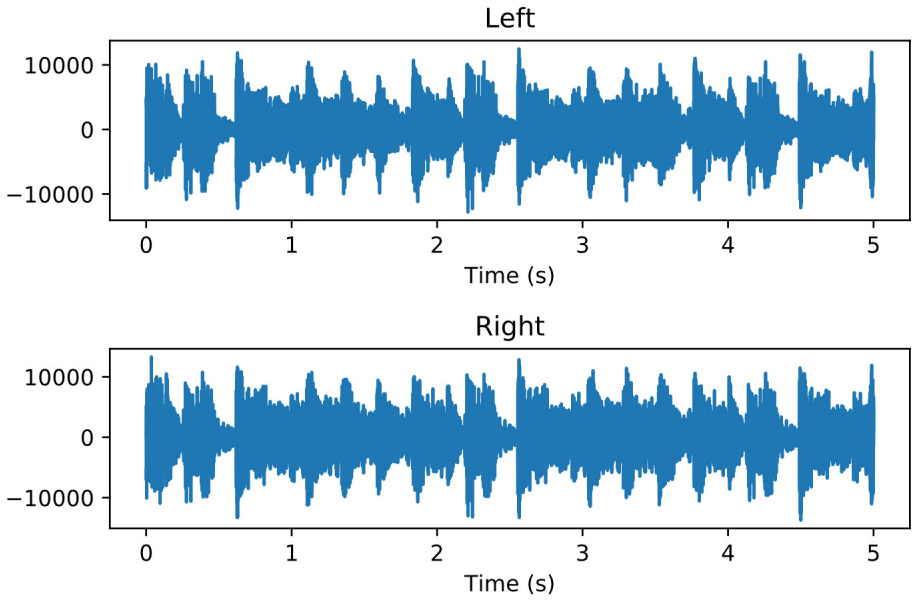


Figure 8: Working with the capture `DSP_io_stream.data_capture` buffer when `numChan = 2` and displaying the left and right channel waveforms.

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.,

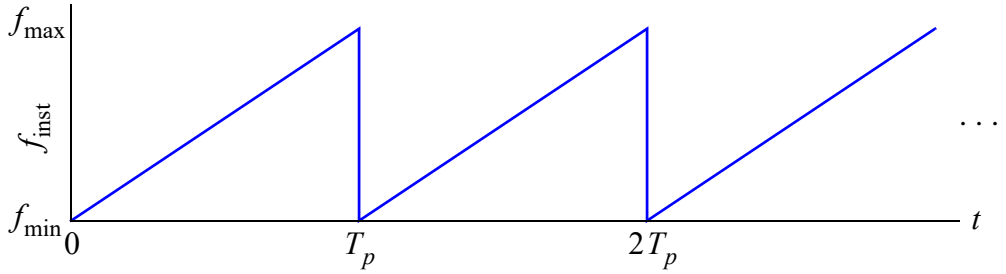


Figure 9: The instantaneous frequency of a linear chirp sinusoid.

In continuous time the signal takes the form

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

which has instantaneous frequency

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

A discrete-time implementation of $x(t)$ is found in the class `LinearChirp`, Listing 9 below.

Listing 9: 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 10: 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 10 shows the result spectrogram is an indeed a linear chirp running from 2 kHz to 3 KHz with a period of 400 ms.

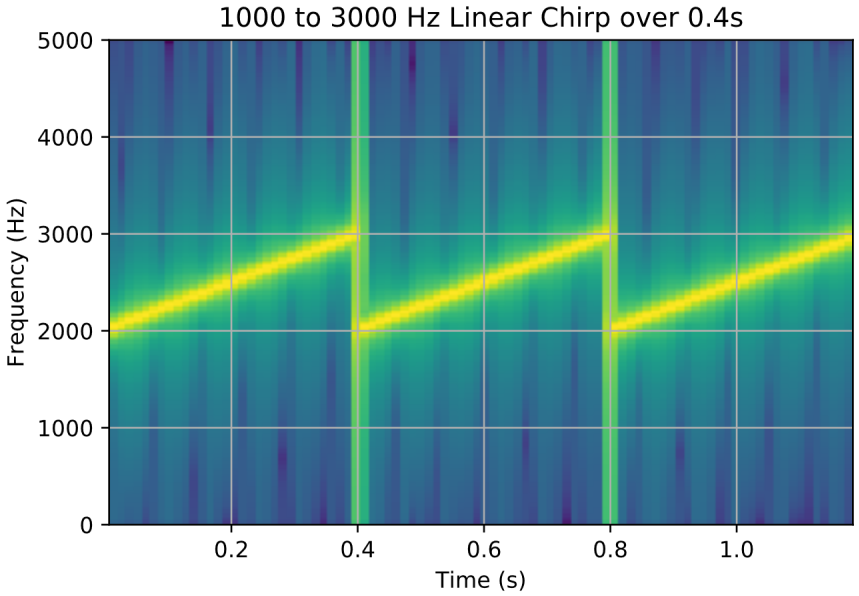
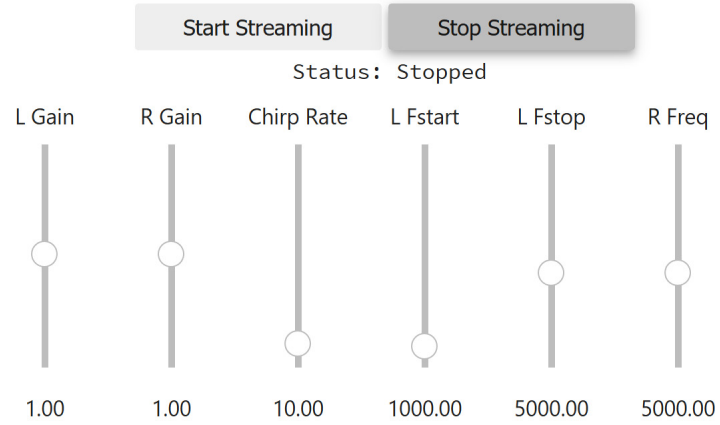


Figure 10: Spectrogram of a statically created linear chirp created using a 12000 sample frame.

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 11 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 11: 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 12 by configuring the capture buffer with `Tcapture = 5` and `Tsec = 10` and then

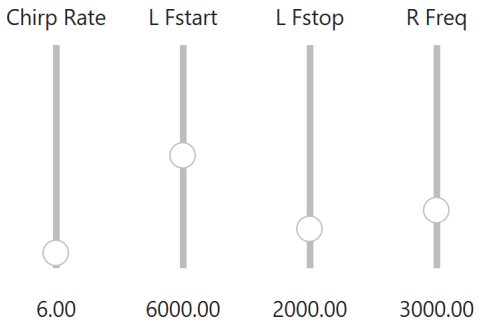


Figure 12: Linear chirp slider settings to be used in the capture buffer experiment.

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 13 shows how white noise

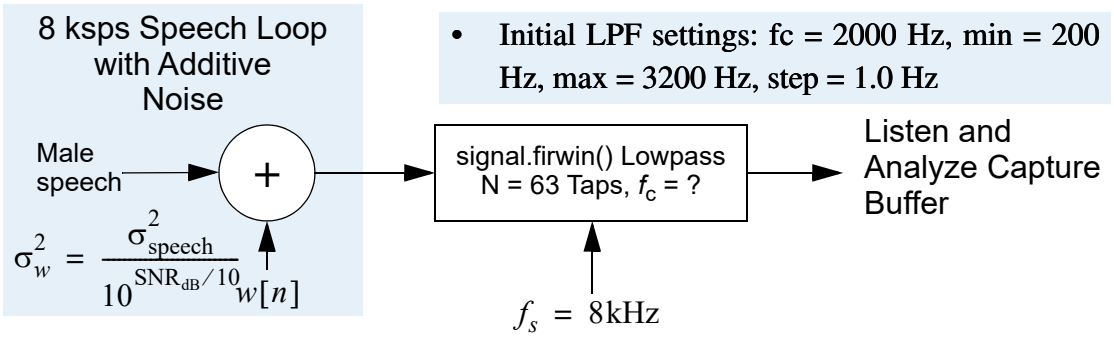


Figure 13: Improving the intelligibility of noisy speech using a 63-tap FIR lowpass inside the callback of a `pyaudio_helper` app.

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 11. Step 2a also obtains the speech loop standard deviation, σ_{speech}, so the signal-to-noise ratio (SNR), defined as $SNR = \sigma_{speech}^2 / \sigma_w^2$, is properly calibrated on-the-fly in the callback. The details can be found in Listing 11 and in `Project1_pyaudio_helper_sample`.

Listing 11: 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 12.

Listing 12: Code cell highlights for Step 2b, the callback, for the tunable bandpass 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) * 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 14 shows the GUI that follows the Step 3 code cell. The attributes of the `fc` slider can be Found in Figure 13.

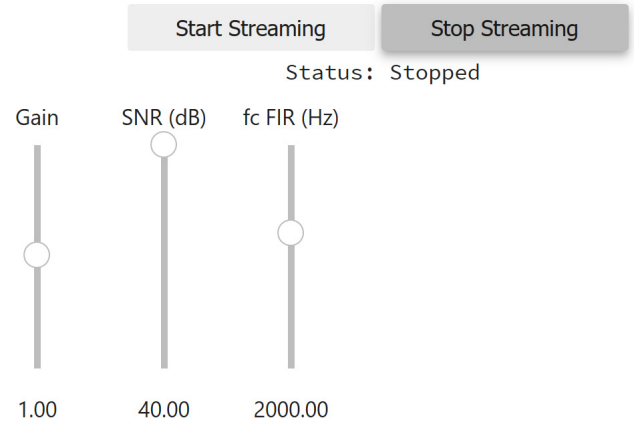


Figure 14: A screen shot of the GUI following the Step 3 code cell, which most importantly has the `fc` slider for changing the lowpass filter cutoff frequency.

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 15. The task is to implement the

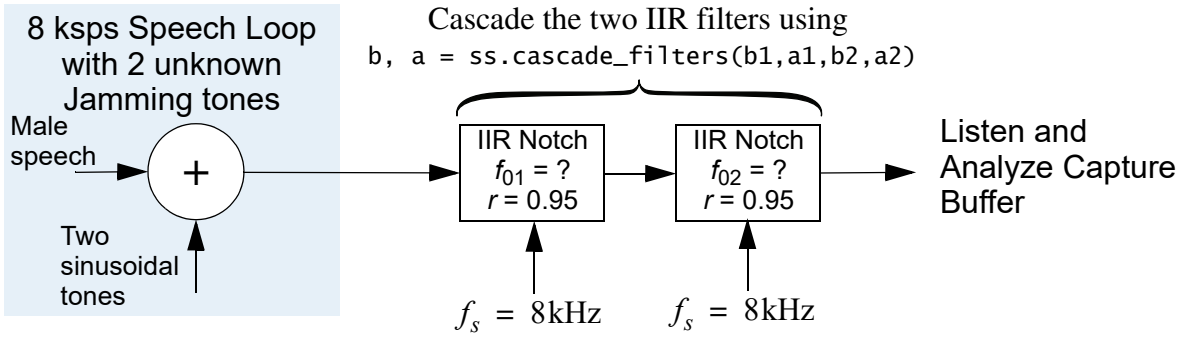


Figure 15: System block diagram for notch filtering tone jammed speech in a `pyaudio_helper` app.

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 16.

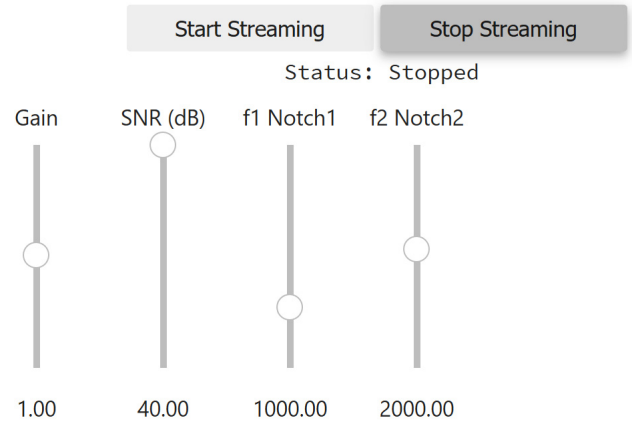


Figure 16: A screen shot of the GUI following the Step 3 code cell, which most importantly has the two sliders for changing the notch center frequencies.

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.

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. C++ overlap and save transform domain filtering 35pts	(a) 20 Develop and implement the algorithm including the filter coefficients and a 1kHz waveform test (b) 10 Compare the timing of the two schemes (c) 10 Noise test to estimate the frequency response
3. Multirate systems simulation using Pylab	(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
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 + 40 + 95 + 40 + 90 = 290