Real-Time Digital Signal Processing Using pyaudio_helper and the ipywidgets

Mark Wickert[‡][*]

Abstract—The focus of this paper is on teaching real-time digital signal processing to electrical and computer engineers using the Jupyter notebook and the code module `pyaudio_helper`, which is a component of the package scikit-dsp-comm. Specifically, we show how easy it is to design, prototype, and test using PC-based instrumentation, real-time DSP algorithms for processing analog signal inputs and returning analog signal outputs, all within the Jupyter notebook. A key feature is that real-time algorithm prototyping is simplified by configuring a few attributes of a `DSP_io_stream` object from the `pyaudio_helper` module, leaving the developer to focus on the real-time DSP code contained in a *callback* function, using a template notebook cell. Real-time control of running code is provided by ipywidgets. The PC-based instrumentation aspect allows measurement of the analog input/output (I/O) to be captured, stored in text files, and then read back into the notebook to compare with the original design expectations via `matplotlib` plots. In a typical application *slider* widgets are used to change variables in the callback. One and two channel audio applications as well as algorithms for complex signal (in-phase/quadrature) waveforms, as found in software-defined radio, can also be developed. The analog I/O devices that can be interfaced are both internal and via USB external sound interfaces. The sampling rate, and hence the bandwidth of the signal that can be processed, is limited by the operating system audio subsystem capabilities, but is at least 48 KHz and often 96 kHz.

Index Terms—digital signal processing, pyaudio, real-time, scikit-dsp-comm

Introduction

As the power of personal computer has increased, the dream of rapid prototyping of real-time signal processing, without the need to use dedicated DSP-microprocessors or digital signal processing (DSP) enhanced microcontrollers, such as the ARM Cortex-M4 [cortexM4], can be set aside. Students can focus on the powerful capability of `numpy`, `scipy`, and `matplotlib`, along with packages such as `scipy.signal` [Scipysignal] and `scikit-dsp-comm` [DSPComm], to explore real-time signals and systems computing.

The focus of this paper is on teaching real-time DSP to electrical and computer engineers using the Jupyter notebook and the code module `pyaudio_helper`, which is a component of the package `scikit-dsp-comm`. To be clear, `pyaudio_helper` is built upon the well known package [pyaudio], which has its roots in *Port Audio* [portaudio]. Specifically, we show how easy it is to design, prototype, and test using PC-based instrumentation, real-time DSP algorithms for processing analog signal inputs and returning analog signal outputs, all within the Jupyter notebook. Real-time algorithm prototyping is simplified by configuring a `DSP_io_stream` object from the `pyaudio_helper` module, allowing the developer to quickly focus on writing a DSP *callback* function using a template notebook cell. The developer is free to take advantage of `scipy.signal` filter functions, write custom classes, and as needed utilize global variables to allow the algorithm to maintain *state* between callbacks pushed by the underlying PyAudio framework. The PC-based instrumentation aspect allows measurement of the analog input/output (I/O) to be captured, stored in text files, and then read back into the notebook to compare with the original design expectations via `matplotlib` plots. Real-time control of running code is provided by ipywidgets. In a typical application *slider* widgets are used to change variables in the callback during I/O streaming. The analog I/O devices that can be interfaced are both internal and via USB external sound interfaces. The sampling rate, and hence the bandwidth of the signal that can be processed, is limited by the operating system audio subsystem capabilities, but is at least 48 KHz and often 96 kHz.

We will ultimately see that to set up an audio stream requires: (1) create and instance of the `DSP_io_stream` class by assigning valid input and output device ports to it, (2) define a callback function to process the input signal sample frames into output sample frames with a user defined algorithm, and (3) call the method `interactive_stream()` to start streaming.

* Corresponding author: mwickert@uccs.edu
‡ University of Colorado Colorado Springs

Analog Input/Output Using DSP Algorithms

A classic text to learn the theory of digital signal processing is [Opp2010]. This book is heavy on the underlying theoretical concepts of DSP, including the mathematical modeling of analog I/O systems as shown in Figure 1. This block diagram is a mathematical abstraction of what will be implemented using [pyaudio] and a PC audio subsystem. An analog or continuous-time signal $x(t)$ enters the system on the left and is converted to the discrete-time signal $x[n]$ by the analog to digital block. In practice this block is known as the analog-to-digital converter (ADC). The sampling rate f_s, which is the inverse of the sampling period, T, leads to $x[n] = x(nT)$. To be clear, $x[n]$, denotes a sequence of samples corresponding to the original analog input $x(t)$. The use of brackets versus parentheses differentiates the two signal types as discrete-time and continuous-time respectively. The sampling theorem [Opp2010] tells us that the sampling rate f_s must be greater than twice the highest frequency we wish to represent in the discrete-time domain. Violating this condition results in *aliasing*, which means a signal centered on frequency $f_0 > f_s/2$ will land inside the band of frequencies $[0, f_s/2]$. Fortunately, most audio ADCs limit the signal bandwidth of $x(t)$ in such a way that signals with frequency content greater than $f_s/2$ are eliminated from passing through the ADC. Also note in practice, $x[n]$ is a scaled and finite precision version of $x(t)$. In real-time DSP environments the ADC maps the analog signal samples to signed integers, most likely `int16`. As we shall see in pyaudio, this is indeed the case.

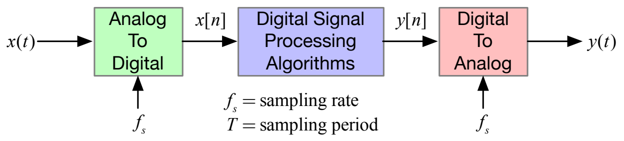


Fig. 1: *Analog signal processing implemented using real-time DSP.*

The DSP algorithms block can be any operation on samples $x[n]$ that makes sense. Ultimately, once we discuss frame-based processing in the next section, we will see how Python code fulfills this. At this beginning stage, the notion is that the samples flow through the algorithm one at a time, that is one input results in one output sample. The output samples are converted back to analog signal $y(t)$ by placing the samples into a digital-to-analog converter (DAC). The DAC does not simply set $y(nT) = y[n]$, as a continuous function time t must be output. A *reconstruction* operation takes place inside the DAC which *interpolates* the $y[n]$ signal samples over continuous time. In most DACs this is accomplished with a combination of digital and analog filters, the details of which is outside the scope of this paper. The use of

In a DSP theory class the algorithm for producing $y[n]$ from $x[n]$ is typically a *causal* linear time-invariant (LTI) system/filter, implemented via a difference equation, i.e.,

$$y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{m=0}^{M} b_m x[n-m] \qquad (1)$$

where $a_k, k = 1, 2, \ldots, N$ and $b_m, m = 0, 1, \ldots, M$ are the filter coefficients. The filter coefficients that implement a particular filter design can be obtained using design tools in [DSPComm].

Other algorithms of course are possible. We might have a two channel system and perform operations on both signals, say combining them, filtering, and locally generating time varying periodic signals to create audio special effects. When first learning about real-time DSP it is important to start with simple algorithm configurations, so that external measurements can be used to characterize the systems and verify that the intended results are realized. Developing a real-time DSP project follows along the lines of, design, implement, and test using external test equipment. The Jupyter notebook allows all of this to happen in one place, particularly if the test instrumentation is also PC-based, since PC-based instrument results can be exported as `csv` and then imported in Jupyter notebook using `loadtxt`. Here we advocate the use of PC-based instruments, so that all parties, student/instructor/tinkerer, can explore real-time DSP from most anywhere at any time. In this paper we use the Analog Discovery 2 [AD2] for signal generation (two function generator channels), signal measurement (two scope channels, with fast Fourier transform (FFT) spectrum analysis included). It is also helpful to have a signal generator cell phone app available, and of course music from a cell phone or PC. All of the cabling is done using 3.5mm stereo patch cables and small pin header adapters [3p5mm] to interface to the AD2.

Frame-based Real-Time DSP Using the `DSP_io_stream` class

The block diagram of Figure 2 illustrates the essence of this paper. Implementing the structure of this figure relies upon the class `DSP_io_stream`, which is housed in `sk_dsp_comm.pyaudio_helper.py`. To make use of this class requires the scipy stack (numpy, scipy, and matplotlib), as well as [DSPComm] and [pyaudio]. PyAudio is multi-platform, with the configuration platform dependent. The set-up is documented at [pyaudio] and SPCommTutorial. The classes and functions of `pyaudio_helper` are detailed in Figure 3. We will make reference to the classes, methods, and functions throughout the remainder of this paper.

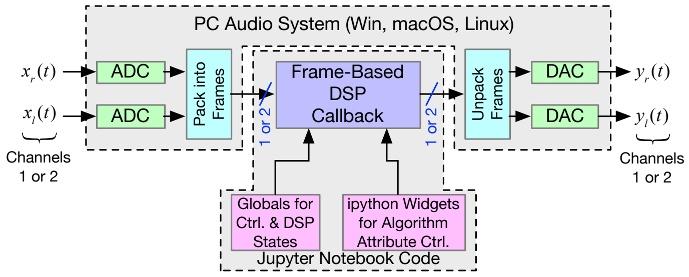


Fig. 2: *Two channel analog signal processing implemented using frame-based real-time DSP.*

With `DSP_io_stream` one or two channel streaming is possible, as shown in Figure 2. The ADCs and DACs can be internal to the PC or external, say using a USB interface. In a modern PC the audio subsystem has a microphone hardwired to the ADCs and the DACs are connected to the speakers and 3.5mm headphone jack. To provide more flexibility in doing real-time DSP, an external USB audio interface is essential. Two worthy options are the Sabrent at less than \$10 and the iMic at under \$40. You get what you pay for. The iMic is ideal for full two channel audio I/O processing and also has a line-in/mic switch setting, while the Sabrent offers a single channel input and two channel output. Both are very capable for their intended purposes. A photograph of the AD2 with the iMic interface, 3.5mm splitters and the pin header interfaces mentioned earlier, is shown in Figure 4. The 3.5mm audio splitters are optional, but allow headphones to be plugged into the output while leaving the AD2 scope connected, and the ability to input music/function generator from a cellphone while leaving the AD2 input cable connected (pins wires may need to be pulled off the AD2 to avoid interaction between the two devices in parallel).

When a `DSP_io_stream` is created (top of Figure 3) it needs to know which input and output devices to connect to. If you just want and input or just an out, you still need to supply a valid output or input device, respectively. To list the internal/external devices available on a given PC we use the function `available_devices()` from Figure 3. If you add or remove devices while the notebook kernel is running, you will need to restart the kernel to get an accurate listing of devices. The code block below was run with the iMic plugged into a USB hub:

Module: sk_dsp_comm.pyaudio_helper.py	
Class: DSP_io_stream	Inputs/Outputs
Constructor( ):	(0) Stream callback function name (1) Input device index (default 1) (2) Output device index (default 4) (3) Frame length (default 1024) (4) Sampling rate in Hz (default 44100) (5) Capture buffer length in s (default 0) (6) Sleep time (default 0.1 s from PyAudio)
interactive_stream( ): (threaded & buttons)	(0) Stream time in s (default 2, 0 for infinite) (1) Number of channels (default 1 or 2)
returns:	none, but ipywidget start/stop buttons
DSP_callback_tic( ):	None, but updates a time stamp attribute
returns:	none
DSP_callback_toc( ):	None, but updates a time stamp attribute
returns:	none
stream_stats( ):	None
returns:	Prints callback statistics
DSP_capture_add_ samples( ):	(0) Append a new frame of float signal samples to the attribute data_capture
returns:	none
cb_active_plot( ):	(0) Start time in ms (1) Stop time in ms (2) Line color (default ‘b’)
returns:	Timing plot showing time in callback
DSP_capture_add_ samples_stereo( ):	(0) Append a new frame of left float signal samples to the attribute data_capture_left (1) Append a new frame of right float signal samples to the attribute data_capture_right
returns:	none
get_LR( ):	(0) Packed float32 input frame
returns:	(0) Unpacked float32 left channel (1) Unpacked float32 right channel
pack_LR( ):	(0) Left output float32 frame (1) Right output float32 frame
returns:	(0) Packed float32 frame
Class: loop_audio	Inputs/Outputs
Constructor( ):	(0) Audio sample array to be looped (1) Offset into array (default 0)
get_samples( ):	(0) frame_length
Functions:	Inputs/Outputs
available_devices( ):	None
returns:	Prints available input and output audio devices along with their port indices

Fig. 3: *The major classes and functions of the module* `sk_dsp_comm.pyaudio_helper.py`*.*

```
import sk_dsp_comm.pyaudio_helper as pah
In[3]: pah.available_devices()
Out[3]:
Index 0 device name = Built-in Microphone,
         inputs = 2, outputs = 0
Index 1 device name = Built-in Output,
         inputs = 0, outputs = 2
Index 2 device name = iMic USB audio system,
         inputs = 2, outputs = 2
```

The output list can be viewed as a look-up table (LUT) for how to *patch* physical devices into the block diagram of Figure 2.

We now shift the focus to the interior of Figure 2 to discuss frame-based DSP and the *Frame-Based DSP Callback*. When a DSP microcontroller is configured for real-time DSP, it can focus on just this one task very well. Sample-by-sample processing is possible with low I/O latency and overall reasonable audio sample throughput. On a PC, with its multitasking OS, there is a lot going on. To get reasonable audio sample throughput the PC audio subsystem fills or *packs* an input buffer with `frame_length` samples (or two times `frame_length`), sample for a two channel stream) originating as 16-bit signed integers (i.e., `int16`), before calling the *callback* function. The details of the callback function is the subject of the next section. As the callback prepares to exit, an output buffer of 16-bit signed integers is formed, again of length `frame_length`, and the buffer is absorbed by the PC audio subsystem. In the context of *embedded systems* programming, the callback can be thought of as an *interrupt service routine*. To the PC audio community the frame or buffer just described, is also known as a *CHUNK*. In a two-channel stream the frame holds an interleaving of left and right channels, `...LRLRL...` in the buffer formed/absorbed by the PC audio system. Understand that the efficiency of frame-based processing comes with a price. The buffering either side of the callback block of Figure 2 introduces a latency or processing time delay of at least two times the `frame_length` times the sampling period.

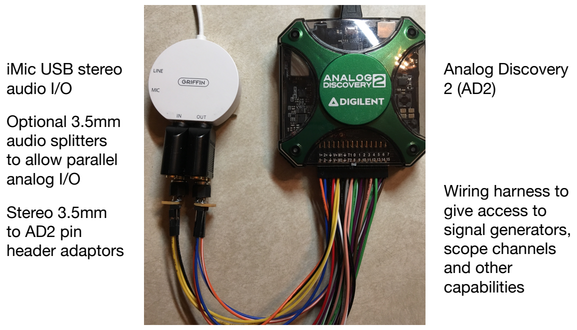

iMic USB stereo audio I/O
Optional 3.5mm audio splitters to allow parallel analog I/O
Stereo 3.5mm to AD2 pin header adaptors
Analog Discovery 2 (AD2)
Wiring harness to give access to signal generators, scope channels and other capabilities
GRIFFIN
ANALOG DISCOVERY 2
DIGILENT

Fig. 4: *The iMic stereo USB audio device and the Digilent Analog Discovery 2 (AD2), including the wiring harness.*

Moving along with this top level discussion, the central block of Figure 2 is labeled Frame-Based DSP Callback, and as we have alluded to already, is where the real-time DSP code resides. Global variables are needed inside the callback, as the input/output signature is fixed by [pyaudio]. The globals allow algorithm parameters to be available inside the callback, e.g., filter coefficients, and in the case of a digital filter, the filter state must be maintained from frame-to-frame. We will see in the examples section how `scipy.signal.lfilter()`, which implements (1), conveniently supports frame-based digital filtering. To allow interactive control of parameters of the DSP algorithm we can use `ipywidgets`. We will also see later the sliders widgets are particularly suited to this task.

Anatomy of a PyAudio Callback function

Before writing the callback we first need to instantiate a `DSP_io_stream` object, as shown in the following code block:

```
DSP_IO = pah.DSP_io_stream(callback, #callback name
                 2,2, # set I/O device indices
                 fs=48000, # sampling rate
                 Tcapture=0) # capture buffer length
```

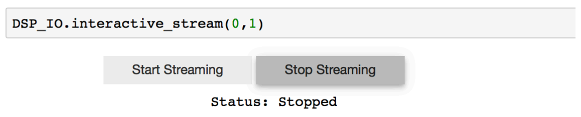
The constructor for `DSP_io_stream` of Figure 3 and the code block above confirm that most importantly we need to supply a function callback name, and most likely provide custom input/output device numbers, choose a sampling rate, and optionally choose the length of the capture buffer.

A basic single channel *loop through* callback function, where the input samples are passed to the output, is shown in the code block below:

```
# define a pass through, y = x, callback
def callback(in_data, frame_length, time_info,
             status):
    global DSP_IO, b, a, zi #no widgets yet
    DSP_IO.DSP_callback_tic() #log entering time
    # convert audio byte data to an int16 ndarray
    in_data_nda = np.frombuffer(in_data,
                                dtype=np.int16)
    #**********************************************
    # Begin DSP operations here
    # for this app cast int16 to float32
    x = in_data_nda.astype(float32)
    y = x # pass input to output
    # Typically more DSP code here
    # Optionally apply a linear filter to the input
    #y, zi = signal.lfilter(b,a,x,zi=zi)
    #**********************************************
    # Save data for later analysis
    # accumulate a new frame of samples if enabled
    # with Tcapture
    DSP_IO.DSP_capture_add_samples(y)
    #**********************************************
    # Convert from float back to int16
    y = y.astype(int16)
    DSP_IO.DSP_callback_toc() #log departure time
    # Convert ndarray back to bytes
    return y.tobytes(), pah.pyaudio.paContinue
```

The `frame_length` has been set to 1024, and of the four required inputs from [pyaudio], the first, `in_data`, is the input buffer which we first convert to a `int16 ndarray` using `np.frombuffer`, and then as a working array convert to `float32`. Note to fill the full dynamic range of the fixed-point signal samples, means that the $x[n]$ sample values can range over $[-2^{15}, 2^{15}-1]$. Passing over the comments we set `y=x`, and finally convert the output array `y` back to `int16` and then in the `return` line back to a byte-string buffer using `.tobytes()`. In general when `y` is converted from `float` back to `int16`, clipping/overflow will occur unless the dynamic range mentioned above is observed. Along the way code instrumentation methods from Figure 3 are included to record time spent in the callback (`DSP_callback_tic()` and `DSP_callback_toc()`) and store samples for later analysis in the attribute `capture_buffer` (`DSP_capture_add_samples`). These features will be examined in an upcoming example.

To start streaming we need to call the method `interactive_stream()`, which runs the stream in a thread and displays `ipywidgets` start/stop buttons below the code cell as shown in Figure 5.



```
DSP_IO.interactive_stream(0,1)
```

```
Status: Stopped
```

Fig. 5: *Setting up an interactive stream for the simple $y = x$ loop through, using a run time of 0, which implies run forever.*

Performance Measurements

The loop through example is good place to explore some performance metrics of 2, and take a look at some of the instrumentation that is part of the `DSP_io_stream` class. The methods `DSP_callback_tic()` and `DSP_callback_toc()` store time stamps in attributes of the class. Another attribute stores samples in the attribute `data_capture`. For the instrumentation to collect operating data we need to set `Tcapture` greater than zero. We will also set the total run time to 2s:

```
DSP_IO = pah.DSP_io_stream(callback,2,2,fs=48000,
                           Tcapture=2)
DSP_IO.interactive_stream(2,1)
```

Running the above in Jupyter notebook cell will capture 2s of data. The method `stream_stats()` displays the following:

```
Ideal Callback period = 21.33 (ms)
Average Callback Period = 21.33 (ms)
Average Callback process time = 0.40 (ms)
```

which tells us that as expected for a sampling rate of 48 kHz, and a frame length of 1024 is simply

$$T_{\text{callback period}} = 1024 \times \frac{1}{48000} = 21.33 \text{ ms} \tag{2}$$

The time spent in the callback should be very small, as very little processing is being done. We can also examine the callback latency by first having the AD2 input a low duty cycle pulse train at a 2 Hz rate, thus having 500 ms between pules. We then use the scope to measure the time difference between the input (scope channel C2) and output (scope channel C1) waveforms. The resulting plot is shown in Figure 6. We see that PyAudio and and the PC audio subsystem introduces about 70.7ms of latency. A hybrid iMic ADC and builtin DAC results in 138 ms on macOS. Moving to Win 10 latency increases to 142 ms, using default USB drivers.

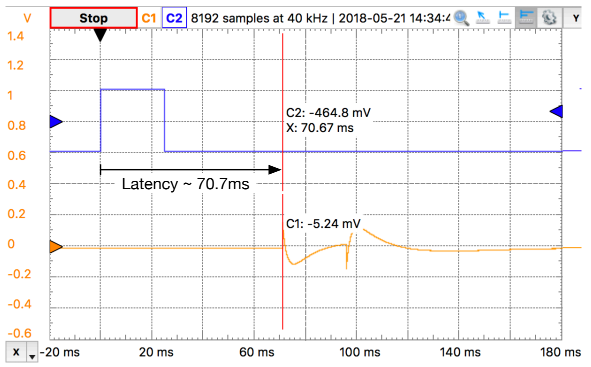


Fig. 6: *Callback latency measurement using the AD2 where C2 is the input and C1 is the output, of a 2 Hz pulse train in the loop through app.*

The frequency response magnitude of an LTI system can be measured using the fact that [Opp2010] at the output of a system driven by white noise, the measured power output spectrum is a scaled version of the underlying system frequency response magnitude squared, i.e.,

$$S_{y,\text{measured}}(f) = \sigma_x^2 |H_{\text{LTI system}}(f)|^2 \tag{3}$$

where σ_x^2 is the variance of the input white noise signal. Here we use this technique to first estimate the frequency response magnitude of the input path (ADC only) using the attribute `DSP_IO.capture_buffer`, and secondly take end-to-end (ADC-DAC) measurements using the AD2 spectrum analyzer in dB average mode (500 records). In both cases the white noise input is provided by the AD2 function generator. Finally, the AD2 measurement is saved to a CSV file and imported into the Jupyter notebook, as shown in the code block below. This allows an overlay of the ADC and ADC-DAC measurements, entirely in the Jupyter notebook.

```python
import sk_dsp_comm.sigsys as ss
f_AD,Mag_AD = loadtxt('Loop_through_noise_SA.csv',
                      delimiter=',',skiprows=6,
                      unpack=True)
Pxx, F = ss.my_psd(DSP_IO.data_capture,2**11,48000);
plot(F,10*log10(Pxx/Pxx[20]))
plot(f_AD,Mag_AD-Mag_AD[100])
ylim([-10,5])
xlim([0,20e3])
ylabel(r'ADC Gain Flatness (dB)')
xlabel(r'Frequency (Hz)')
legend((r'ADC only from DSP_IO.capture_buffer',r
       'ADC-DAC from AD2 SA dB Avg'))
title(r'Loop Through Gain Flatness using iMic at
      $f_s = 48$ kHz')
grid();
savefig('Loop_through_iMic_gain_flatness.pdf')
```

The results are compared in Figure 7, where we see a roll-off of about 3 dB at about 14 kHz in both the ADC path and the composite ADC-DAC path. The composite ADC-DAC begins to rise above 17 kHz and flattens to 2 dB down from 18-20 kHz. As a practical matter, humans do not hear sound much above 16 kHz, so the peaking is not much of an issue. Testing of the Sabrent device the composite ADC-DAC 3 dB roll-off occurs at about 17 kHz. The native PC audio output can for example be tested in combination with the iMic or Sabrent ADCs.

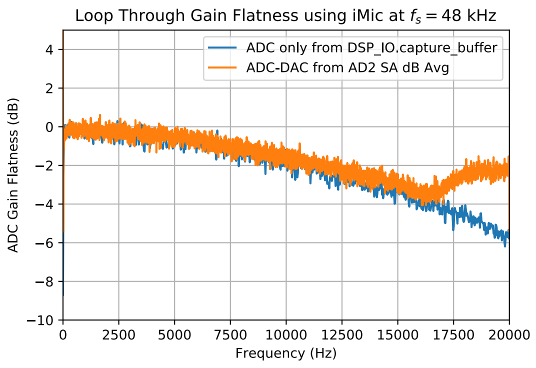


Fig. 7: *Gain flatness of the loop through app of just the ADC path via the `DSP_IO.capture_buffer` and then the ADC-DAC path using the AD2 spectrum analyzer to average the noise spectrum.*

Examples

In this section we consider a collection of applications examples. This first is a simple two channel loop-through with addition of left and right gain sliders. The second is again two channel, but now cross left-right panning is developed. In of these examples the DSP is memoryless, so there is no need to maintain state using Python globals. The third example is an equal-ripple bandpass filter, which utilizes `sk_dsp_comm.fir_design_helper` to design the filter. The final example develops a three-band audio equalizer using *peaking filters* to raise and lower the gain over a narrow band of frequencies.

Left and Right Gain Sliders

In this first example the signal processing is again minimal, but now two-channel (stereo) processing is utilized, and left and right channel gain slider using `ipywidgets` are introduced. Since the audio stream is running in a thread, the `ipywidgets` can freely run and interactively control parameters inside the callback function. The two slider widgets are created below, followed by the callback, and finally calling the `interactive_stream` method to run without limit in two channel mode. A 1 kHz sinusoid test signal is input to the left channel and a 5 kHz sinusoid is input to the right channel. While viewing the AD2 scope output in real-time, the gain sliders are adjusted and the signal levels move up and down. A screenshot taken from the Jupyter notebook is combined with a screenshot of the scope output to verify the correlation between the observed signal amplitudes and the slider positions is given in Figure 8. The callback listing, including the set-up of the ipywidgets gain sliders, is given below:

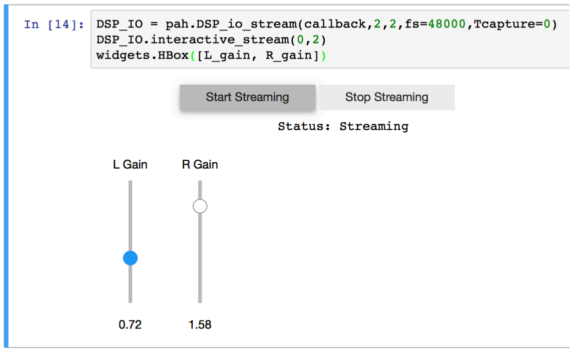
```python
# Set up two sliders
L_gain = widgets.FloatSlider(description = 'L Gain',
             continuous_update = True,
             value = 1.0, min = 0.0,
             max = 2.0, step = 0.01,
             orientation = 'vertical')
R_gain = widgets.FloatSlider(description = 'R Gain',
             continuous_update = True,
             value = 1.0, min = 0.0,
             max = 2.0, step = 0.01,
             orientation = 'vertical')

# L and Right Gain Sliders callback
def callback(in_data, frame_count, time_info,
             status):
    global DSP_IO, L_gain, R_gain
    DSP_IO.DSP_callback_tic()
    # convert byte data to ndarray
    in_data_nda = np.frombuffer(in_data,
                                dtype=np.int16)
    # separate left and right data
    x_left,x_right = DSP_IO.get_LR(in_data_nda.\
                                   astype(float32))
    #*********************************************
    # DSP operations here
    y_left = x_left*L_gain.value
    y_right = x_right*R_gain.value

    #*********************************************
    # Pack left and right data together
    y = DSP_IO.pack_LR(y_left,y_right)
    # Typically more DSP code here
    #*********************************************
    # 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()
    # Convert ndarray back to bytes
    return y.tobytes(), pah.pyaudio.paContinue
```

Note for this two channel stream, the audio subsystem interleaves left and right samples, so now the class methods `get_LR` and `pack_LR` of Figure 3 are utilized to unpack the left and right samples and then repack them, respectively. A screenshot of the gain sliders app, including an AD2 scope capture, with C1 on the left channel and C2 on the right channel, is given in Figure 8.

The ability to control the left and right audio level are as expected, especially when listening.

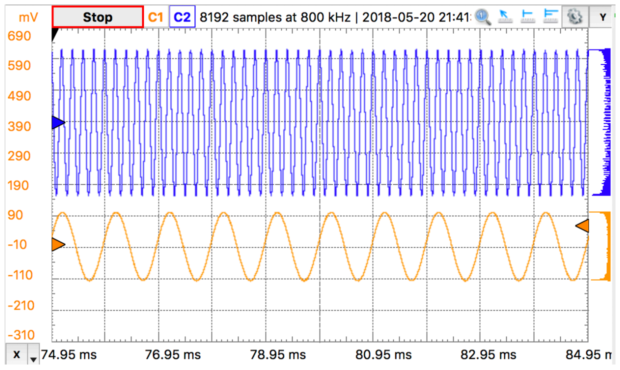
Cross Left-Right Channel Panning

This example again works with a two channel signal flow. The application is to implement a cross channel panning system. Ordinarily panning moves a single channel of audio from 100%



```
In [14]: DSP_IO = pah.DSP_io_stream(callback,2,2,fs=48000,Tcapture=0)
         DSP_IO.interactive_stream(0,2)
         widgets.HBox([L_gain, R_gain])
```

(a) Jupyter notebook start/stop stream controls and left/right gain sliders



(b) Audio outputs for a 1 kHz left input and 5 kHz right input

Fig. 8: A simple stereo gain slider app: (a) Jupyter notebook interface and (b) testing using the AD2 with generators and scope channel C1 (orange) on left and C2 (blue) on right.

left to 100% right as a slider moves from 0% to 100% of its range. At 50% the single channel should have equal amplitude in both channels. In cross channel panning two input channels are super imposed, but such that at 0% the left and right channels are fully in their own channel. At 50% the left and right outputs are equally mixed. At 100% the input channels are now swapped. Assuming that a represents the panning values on the interval $[0,100]$, a mathematical model of the cross panning app is

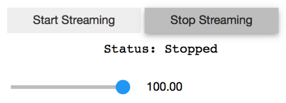
$$L_{\text{out}} = (100-a)/100 \times L_{\text{in}} + a/100 \times R_{\text{in}} \quad (4)$$

$$R_{\text{out}} = a/100 \times L_{\text{in}} + (100-a)/100 \times R_{\text{in}} \quad (5)$$

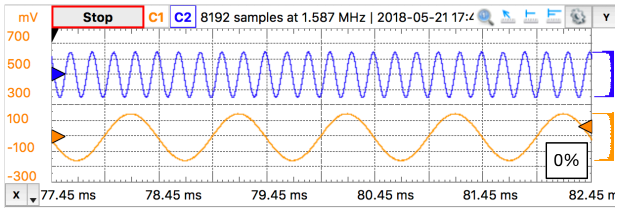
where L_{in} and L_{out} are the left channel inputs and outputs respectively, and similarly R_{in} and R_{out} for the right channel. In code we have:

```
# Cross Panning
def callback(in_data, frame_length, time_info,
             status):
    global DSP_IO, panning
    DSP_IO.DSP_callback_tic()
    # convert byte data to ndarray
    in_data_nda = np.frombuffer(in_data,
                                dtype=np.int16)
    # separate left and right data
    x_left,x_right = DSP_IO.get_LR(in_data_nda.\
                                   astype(float32))
    #***********************************************
    # DSP operations here
    y_left = (100-panning.value)/100*x_left \
             + panning.value/100*x_right
    y_right = panning.value/100*x_left \
              + (100-panning.value)/100*x_right

    #***********************************************
    # Pack left and right data together
    y = DSP_IO.pack_LR(y_left,y_right)
    # Typically more DSP code here
    #***********************************************
    # 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()
    # Convert ndarray back to bytes
    return y.tobytes(), pah.pyaudio.paContinue
```

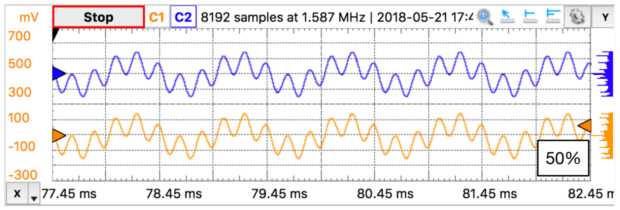
This app is best experienced by listening, but visually Figure 9 shows a series of scope captures, parts (b)-(d), to explain how the sounds sources swap from side-to-side as the panning value changes.



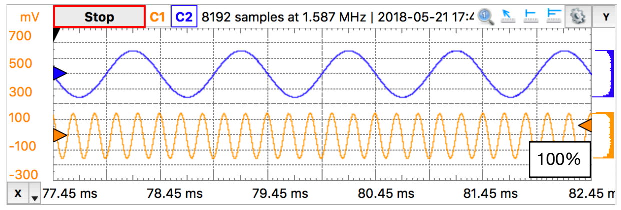
(a) Jupyter notebook start/stop stream controls and the panning slider



(b) Audio outputs with panning control at 0%



(c) Audio outputs with panning control at 50%



(d) Audio outputs with panning control at 100%

Fig. 9: Cross left/right panning control: (a) launching the app in the Jupyter notebook and (b)-(d) a sequence of scope screenshots as the panning slider is moved from 0% to 50%, and then to 100%.

For dissimilar left and right audio channels, the action of the slider creates a spinning effect when listening. It is possible to extend this app with an automation, so that a low frequency sinusoid or other waveform changes the panning value at a rate controlled by a slider.

FIR Bandpass Filter

In this example we design a high-order FIR bandpass filter using `sk_dsp_comm.fir_design_helper` and then implement the design to operate at $f_s = 48$ kHz. Here we choose the bandpass critical frequencies to be 2700, 3200, 4800, and 5300 Hz, with a passband ripple of 0.5 dB and stopband attenuation of 50 dB (see fir_d). Theory is compared with AD2 measurements using, again using noise excitation. When implementing a digital filter using frame-based processing, `scipy.signal.lfilter` works nicely. The key is to first create a zero initial condition array `zi` and hold this in a global variable. Each time `lfilter` is used in the callback the old initial condition `zi` is passed in, then the returned `zi` is held until the next time through the callback.

```
import sk_dsp_comm.fir_design_helper as fir_d
import scipy.signal as signal
b = fir_d.fir_remez_bpf(2700,3200,4800,5300,
                        .5,50,48000,18)
a = [1]
# Set up a zero initial condition to start
zi = signal.lfiltic(b,a,[0])

# define callback (#2)
def callback2(in_data, frame_length, time_info,
              status):
    global DSP_IO, b, a, zi
    DSP_IO.DSP_callback_tic()
    # convert byte data to ndarray
    in_data_nda = np.frombuffer(in_data,
                                dtype=np.int16)
    #***********************************************
    # DSP operations here
    # Here we apply a linear filter to the input
    x = 5*in_data_nda.astype(float32)
    #y = x
    # The filter state/(memory), zi,
    # must be maintained from frame-to-frame,
    # so hold it in a global
    # for FIR or simple IIR use:
    y, zi = signal.lfilter(b, a, x, zi=zi)
    # for IIR use second-order sections:
    #y, zi = signal.sosfilt(sos, x, zi=zi)
    #***********************************************
    # Save data for later analysis
    # accumulate a new frame of samples
    DSP_IO.DSP_capture_add_samples(y)
    #***********************************************
    # Convert from float back to int16
    y = y.astype(int16)
    DSP_IO.DSP_callback_toc()
    return y.tobytes(), pah.pyaudio.paContinue

DSP_IO = pah.DSP_io_stream(callback2,2,2,
                           fs=48000,Tcapture=0)
DSP_IO.interactive_stream(Tsec=0,numChan=1)
```

Following the call to `DSP_io.interactive_stream()` the *start* button is clicked and the AD2 spectrum analyzer estimates the power spectrum. The estimate is saved as a CSV file and brought into the Jupyter notebook to overlay the theoretical design. The comparison results are given in Figure 10.

The theory and measured magnitude response plots are in very close agreement, making the end-to-end design, implement, test very satisfying.

Three Band Equalizer

Here we consider the second-order peaking filter, which has infinite impulse response, and place three of them in cascade with a `ipywidgets` slider used to control the gain of each filter. The peaking filter is used in the design of audio equalizer, where perhaps each filter is centered on octave frequency spacings running from from 10 Hz up to 16 kHz, or so. Each peaking filter can be implemented as a 2nd-order difference equation, i.e., $N = 2$ in equation (1). The design equations for a single peaking filter are given below using z-transform [Opp2010] notation:

$$H_{pk}(z) = C_{\text{pk}} \frac{1 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \tag{6}$$

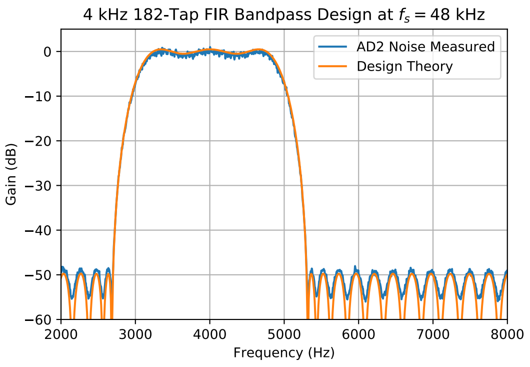


Fig. 10: *An overlay plot of the theoretical frequency response with the measured using an AD2 noise spectrum capture import to the Jupyter notebook.*

which has coefficients

$$C_{\text{pk}} = \frac{1 + k_q \mu}{1 + k_q} \tag{7}$$

$$k_q = \frac{4}{1 + \mu} \tan\left(\frac{2\pi f_c / f_s}{2Q}\right) \tag{8}$$

$$b_1 = \frac{-2\cos(2\pi f_c / f_s)}{1 + k_q \mu} \tag{9}$$

$$b_2 = \frac{1 - k_q \mu}{1 + k_q \mu} \tag{10}$$

$$a_1 = \frac{-2\cos(2\pi f_c / f_s)}{1 + k_q} \tag{11}$$

$$a_2 = \frac{1 - k_q}{1 + k_q} \tag{12}$$

where

$$\mu = 10^{G_{\text{dB}}/20}, \quad Q \in [2, 10] \tag{13}$$

and f_c is the center frequency in Hz relative to sampling rate f_s in Hz, and G_{dB} is the peaking filter gain in dB. Conveniently, the function `peaking` is available in the module `sk_dsp_comm.sigsys`. The app code is given below starting with the slider creation:

```
band1 = widgets.FloatSlider(description \
                  = '100 Hz',
                  continuous_update = True,
                  value = 2.0, min = -20.0,
                  max = 20.0, step = 1,
                  orientation = 'vertical')
band2 = widgets.FloatSlider(description \
                  = '1000 Hz',
                  continuous_update = True,
                  value = 10.0, min = -20.0,
                  max = 20.0, step = 1,
                  orientation = 'vertical')
band3 = widgets.FloatSlider(description \
                  = '8000 Hz',
                  continuous_update = True,
                  value = -1.0, min = -20.0,
                  max = 20.0, step = 1,
                  orientation = 'vertical')

import sk_dsp_comm.sigsys as ss
import scipy.signal as signal
b_b1,a_b1 = ss.peaking(band1.value,100,Q=3.5,
                       fs=48000)
zi_b1 = signal.lfiltic(b_b1,a_b1,[0])
b_b2,a_b2 = ss.peaking(band2.value,1000,Q=3.5,
                       fs=48000)
```

```
zi_b2 = signal.lfiltic(b_b2,a_b2,[0])
b_b3,a_b3 = ss.peaking(band3.value,8000,Q=3.5,
                       fs=48000)
zi_b3 = signal.lfiltic(b_b3,a_b3,[0])
b_12,a_12 = ss.cascade_filters(b_b1,a_b1,b_b2,a_b2)
b_123,a_123 = ss.cascade_filters(b_12,a_12,b_b3,a_b3)
f = logspace(log10(50),log10(10000),100)
w,H_123 = signal.freqz(b_123,a_123,2*pi*f/48000)
semilogx(f,20*log10(abs(H_123)))
grid();

# define a pass through, y = x, callback
def callback(in_data, frame_length, time_info,
             status):
    global DSP_IO, zi_b1, zi_b2, zi_b3
    global band1, band2, band3
    DSP_IO.DSP_callback_tic()
    # convert byte data to ndarray
    in_data_nda = np.frombuffer(in_data,
                                dtype=np.int16)
    #***********************************************
    # DSP operations here
    # Here we apply a linear filter to the input
    x = in_data_nda.astype(float32)
    #y = x
    # Design the peaking filters on-the-fly
    # and then cascade them
    b_b1,a_b1 = ss.peaking(band1.value,100,
                           Q=3.5,fs=48000)
    z1, zi_b1 = signal.lfilter(b_b1,a_b1,x,
                               zi=zi_b1)
    b_b2,a_b2 = ss.peaking(band2.value,1000,
                           Q=3.5,fs=48000)
    z2, zi_b2 = signal.lfilter(b_b2,a_b2,z1,
                               zi=zi_b2)
    b_b3,a_b3 = ss.peaking(band3.value,8000,
                           Q=3.5,fs=48000)
    y, zi_b3 = signal.lfilter(b_b3,a_b3,z2,
                              zi=zi_b3)
    #***********************************************
    # Save data for later analysis
    # accumulate a new frame of samples
    DSP_IO.DSP_capture_add_samples(y)
    #***********************************************
    # Convert from float back to int16
    y = y.astype(int16)
    DSP_IO.DSP_callback_toc()
    # Convert ndarray back to bytes
    return y.tobytes(), pah.pyaudio.paContinue
```

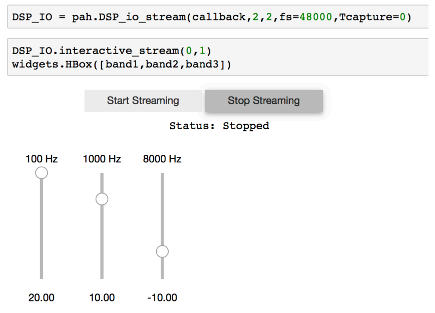
Following the call to `DSP_io.interactive_stream()` the *start* button is clicked and the FFT spectrum analyzer estimates the power spectrum. The estimate is saved as a CSV file and brought into the Jupyter notebook to overlay the theoretical design. The comparison results are given in Figure 11.

Reasonable agreement is achieved, but listening to music is a more effective way of evaluating the end result. To complete the design more peaking filters should be added.

Conclusions and Future Work

In this paper we have described an approach to implement real-time DSP in the Jupyter notebook. This real-time capability rests on top of PyAudio and the wrapper class `DSP_io_stream` contained in `sk_dsp_comm.pyaudio_helper`. The `ipywidgets` allow for interactivity while real-time DSP code is running. The *callback* function does the work using frame-based algorithms, which takes some getting used to. By working through examples we have shown that much can be accomplished with little coding.

A limitation of using PyAudio is the input-to-output latency. At a 48 kHz sampling rate a simple loop though app has around 70 ms of delay. For the application discussed in the paper latency is not a show stopper.

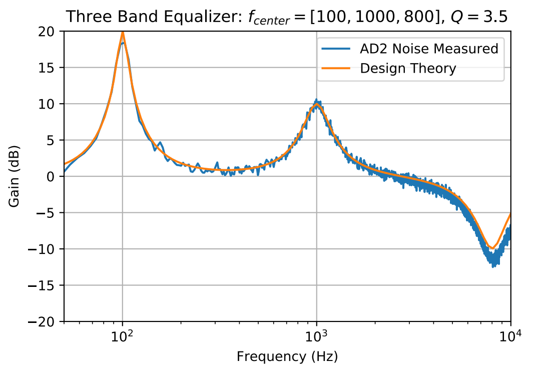


```
DSP_IO = pah.DSP_io_stream(callback,2,2,fs=48000,Tcapture=0)
```

```
DSP_IO.interactive_stream(0,1)
widgets.HBox([band1,band2,band3])
```

Status: Stopped

(a) Jupyter notebook start/stop stream controls and peaking gain sliders



(b) Composite three band frequency response; theory and noise spectrum

Fig. 11: *Three band equalizer: (a) launching the app in the Jupyter notebook and (b) an overlay plot of the theoretical log-frequency response with the measured using an AD2 noise spectrum capture import to the Jupyter notebook.*

In the future we hope to easily develop algorithms that can demodulate software-defined radio (SDR) streams and send the recovered modulation signal out the computer's audio interface via PyAudio. Environments such as GNURadio companion already support this, but being able to do this right in the Jupyter notebook is our desire.

References

[cortexM4] *The DSP capabilities of ARM® Cortex®-M4 and Cortex-M7 Processors*. (2016, November). Retrieved June 25, 2018, from https://community.arm.com/processors/b/blog/posts/white-paper-dsp-capabilities-of-cortex-m4-and-cortex-m7.

[Scipysignal] *Signal Processing*. (2018, May 5). Retrieved June 25, 2018 from https://docs.scipy.org/doc/scipy/reference/signal.html.

[DSPComm] *scikit-dsp-comm*. (2018, June 22). Retrieved June 25, 2018 from https://github.com/mwickert/scikit-dsp-comm.

[pyaudio] *PyAudio*, (2017, March). Retrieved June 25, 2018, from https://people.csail.mit.edu/hubert/pyaudio/.

[portaudio] *Port Audio*. (2012, January 25). Retrieved June 25, 2018 from http://www.portaudio.com/.

[ipywidgets] *ipywidgets*. (2018, June 11). Retrieved June 25, 2018, from https://github.com/jupyter-widgets/ipywidgets.

[Opp2010] Oppenheim, A and Schafer, R (2010). *Discrete-Time Signal Processing* (3rd ed.), New Jersey: Prentice Hall.

[AD2] *Analog Discovery 2*. (2018, June). Retrieved June 25, 2918 from https://store.digilentinc.com/analog-discovery-2-100msps-usb-oscilloscope-logic-analyzer-and-variable-power-supply/.

[3p5mm] *3.5mm Analog Discovery Adaptor Design*. (2018, January 30). Retrieved June 25, 2018 from http://www.eas.uccs.edu/~mwickert/ece5655/.