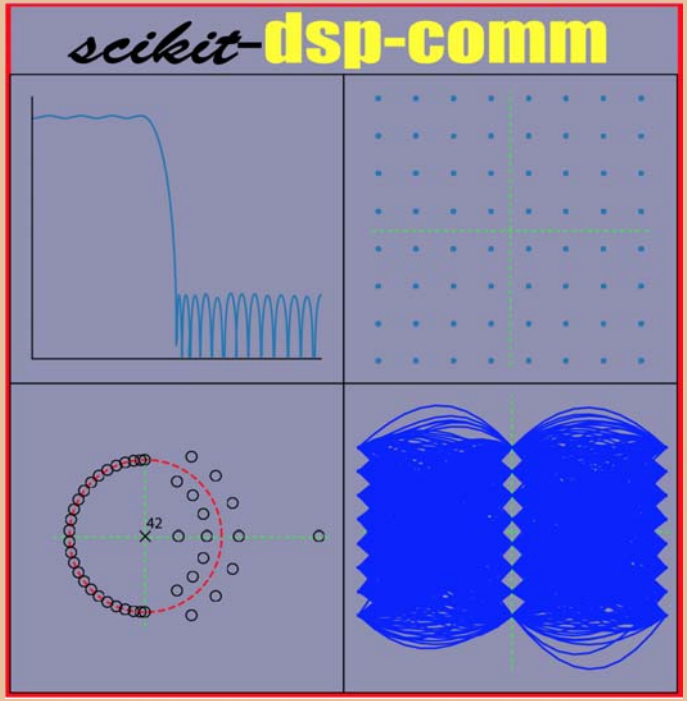
Scikit-DSP-Comm featuring PyAudio for Real-Time Digital Signal Processing

Mark Wickert, PhD. and Professor (mwickert@uccs.edu)
Andrew Smit, Research Assistant and Master's Student (asmit@uccs.edu)
College of Engineering and Applied Science,
University of Colorado Colorado Springs, CO 80933-7150



Introduction

- Open-source Python package available on GitHub. https://github.com/mwickert/scikit-dsp-comm
- Includes a vast array of digital signal processing and communications solutions
- Supports real-time digital signal processing using a low cost (< $7.00) USB audio interface.
- Hands-on interactive demos allow you to hear and see signals being processed from audio sources such as a music players and microphones

Scikit-dsp-comm Package Overview

This package is a collection of functions and classes to support signal processing and communications theory teaching and research. The foundation for this package is `scipy.signal`. The code base currently runs under Python `2.7x` and recently has made great strides to run under Python `3.6`. As of Scipy 2017, the Python `3.6` compatibility is very good.

There are presently ten modules that make up scikit-dsp-comm:

1. `sigsys.py` for basic signals and systems functions both continuous-time and discrete-time, including graphical display tools such as pole-zero plots, up-sampling and down-sampling.
2. `digitalcomm.py` for digital modulation theory components, including asynchronous resampling and variable time delay functions, both useful in advanced modem testing.
3. `synchronization.py` which contains phase-locked loop simulation functions and functions for carrier and phase synchronization of digital communications waveforms.
4. `fec_conv.py` for the generation rate one-half convolutional codes and soft decision Viterbi algorithm decoding, including trellis and trellis-traceback display functions.
5. `fir_design_helper.py` which for easy design of lowpass, highpass, bandpass, and bandstop filters using the Kaiser window and equal-ripple designs, also includes a list plotting function for easily comparing magnitude, phase, and group delay frequency responses.
6. `iir_design_helper.py` which for easy design of lowpass, highpass, bandpass, and bandstop filters using scipy.signal Butterworth, Chebyshev I and II, and elliptical designs, including the use of the cascade of second-order sections (SOS) topology from scipy.signal, also includes a list plotting function for easily comparing of magnitude, phase, and group delay frequency responses.
7. `multirate.py` that encapsulate digital filters into objects for filtering, interpolation by an integer factor, and decimation by an integer factor.
8. `coeff2header.py` write `C/C++` header files for FIR and IIR filters implemented in `C/C++`, using the cascade of second-order section representation for the IIR case. This last module find use in real-time signal processing on embedded systems, but can be used for simulation models in `C/C++`.

Presently the collection of modules contains about 125 functions and classes. The authors/maintainers are working to get more detailed documentation in place.

Extras

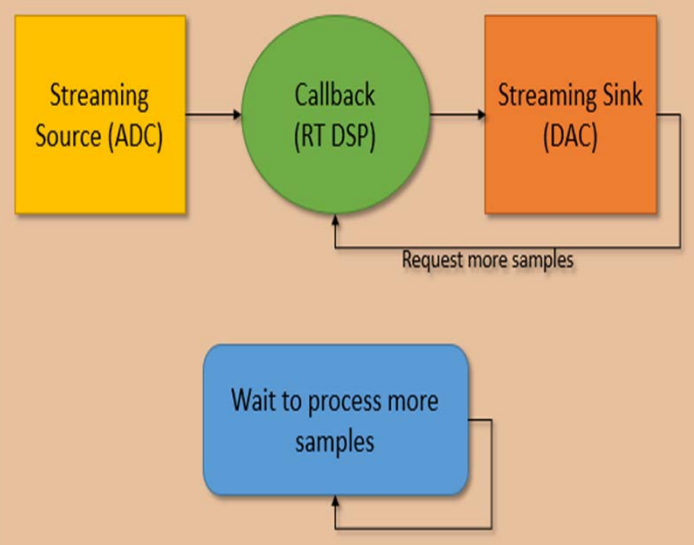
This package contains the helper modules `rtlsdr_helper`, and `pyaudio_helper` which require the packages pyrtlsdr and PyAudio. To use the full functionality of these helpers, install these package with the extras as follows:

```
pip install scikit-dsp-comm[helpers]
```

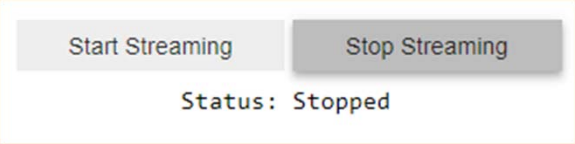
1. `pyaudio_help.py` wraps a class around the code required in `PyAudio` (wraps the C++ library `PortAudio`) to set up a non-blocking audio input/output stream. The user only has to write the callback function to implement real-time DSP processing using any of the input/output devices available on the platform. This resulting object also contains a capture buffer for use in post processing and a timing markers for assessing the processing time utilized by the callback function.
2. `rtlsdr_helper.py` interfaces with `pyrtldsr` to provide a simple captures means for complex baseband software defined radio (SDR) samples from the low-cost (~$20) RTL-SDR USB hardware dongle. The remaining functions in this module support the implementation of demodulators for FM modulation and examples of complete receivers for FM mono, FM stereo, and tools for FSK demodulation, including bit synchronization.

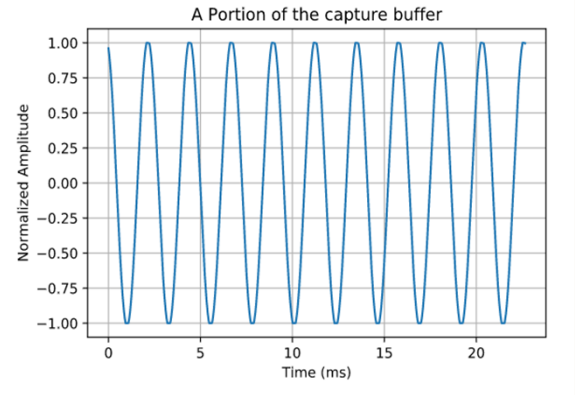
Pyaudio_helper Module Overview

- Wraps a class around the code required in PyAudio (wraps the C++ library PortAudio)
- Non-blocking audio input/output stream
- User only has to write a callback function (green block) to implement DSP algorithms
- Use any of the input/output devices available on the computer
- Capture buffer included for use in post processing




```python
# define a pass through, y = x, callback
def callback(in_data, frame_count, time_info, status):
    DSP_IO.DSP_callback_tic()
    # convert byte data to ndarray
    in_data_nda = np.fromstring(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
    # Typically more DSP code here
    #**********************************************
    # 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 (in_data_nda.tobytes(), pyaudio.paContinue)
    return y.tobytes(), pah.pyaudio.paContinue
```





Audio Interfacing

USB audio interface

- Sabrent USB External Stereo Sound Adapter
- Other USB audio interface

Interfacing cables

- 3.5 mm mono mic input
- 3.5 mm stereo headphone output

Signal Source

- Signal generator phone app
- Software Defined Radio Antenna
- Microphone

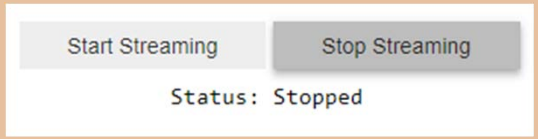

```
pah.available_devices()

Index 0 device name = Microsoft Sound Mapper - Input, inputs = 2, outputs = 0
Index 1 device name = Microphone (2- USB Audio Device, inputs = 2, outputs = 0
Index 2 device name = Microphone (High Definition Aud, inputs = 2, outputs = 0
Index 3 device name = Microphone (NoMachine Microphon, inputs = 2, outputs = 0
Index 4 device name = Microsoft Sound Mapper - Output, inputs = 0, outputs = 2
Index 5 device name = Speakers (2- USB Audio Device), inputs = 0, outputs = 2
Index 6 device name = Speakers (NoMachine Microphone , inputs = 0, outputs = 2
Index 7 device name = Speakers (High Definition Audio, inputs = 0, outputs = 2
```

Call pah.available_devices() to see all available I/O ports

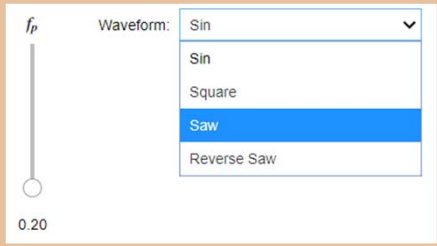
Ipywidgets for Real-Time Parameter Control

- Control parameters in real-time using Ipywidgets package
- Sliders, toggle buttons, radio buttons, dropdown tabs, integer and float text boxes, and more included
- User-defined to meet specific parameter needs
- Application examples include volume sliders, bandpass filter center frequency control, waveform selection, and more
- Threading is used in the pyaudio_helper so that parameters can be changed in real-time during a callback routine

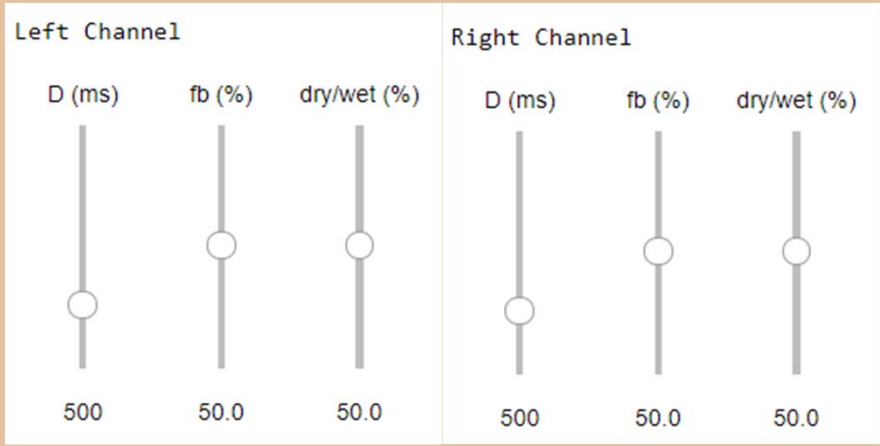
```
# Clear buffer and cook variables
DDL_S.prepareForPlay()

# Create streaming object
DSP_IO = pah.DSP_io_stream(DDL_S.callbackLoop,1,5,fs=FS,frame_length = 512)

# use thread stream so widget can be used
DSP_IO.interactive_stream(Tsec = 0, numChan = 2) # infinite stereo stream

# Display our sliders
DDL_S.showSliders()
```




```
# auto panning frequency slider from 0.01 Hz to 2 Hz
panning_frequency = widgets.FloatSlider(description = r'$f_p$',
                                        continuous_update = True,       # Non-Continuous updates
                                        value = 0.2,
                                        min = 0.01,
                                        max = 4,
                                        step = 0.01,
                                        orientation = 'vertical')
mode = widgets.Dropdown(
    options=['Sin', 'Square', 'Saw', 'Reverse Saw'],
    value='Sin',
    description='Waveform:',
    disabled=False)

widgets.HBox([panning_frequency,mode])
```



Auto-Panning Control

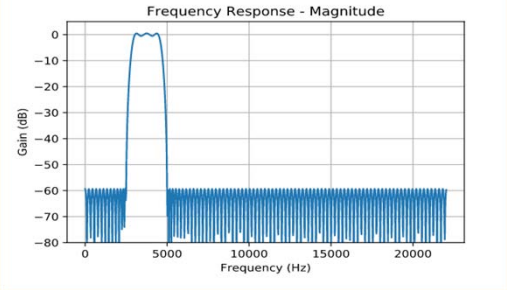


Stereo Delay Control

Real-time Filtering

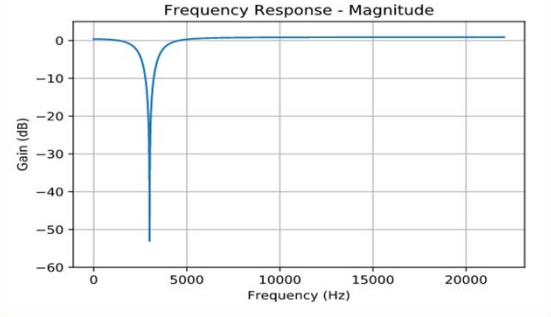
- Use fir_design_helper.py and iir_design_helper.py to design filters
- Show theoretical frequency response
- Implement the filter in a callback
- Input white noise into the system
- Use the filtered white noise to measure the frequency response
- Show spectrogram of the output

```
import sk_dsp_comm.fir_design_helper as fir_d

b = fir_d.fir_remez_bpf(2500,3000,4500,5000,.5,60,44100,18)
fir_d.freqz_resp_list([b],[1],'dB',44100)
ylim([-80,5])
grid();

Remez filter taps = 193.
```

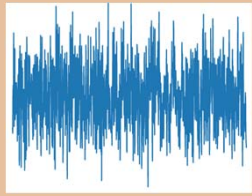


FIR Bandpass Filter

```
# Design an IIR Notch
b, a = ss.fir_iir_notch(3000,44100,r= 0.9)
fir_d.freqz_resp_list([b],[a],'dB',44100,4096)
ylim([-60,5])
grid();
```



IIR Notch Filter

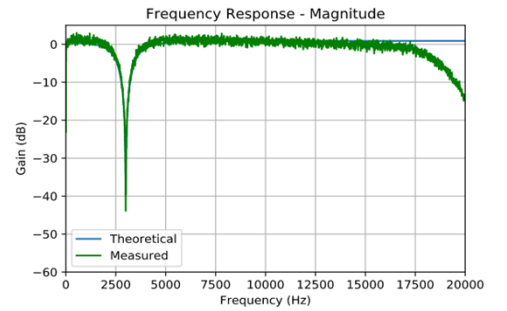


White Noise

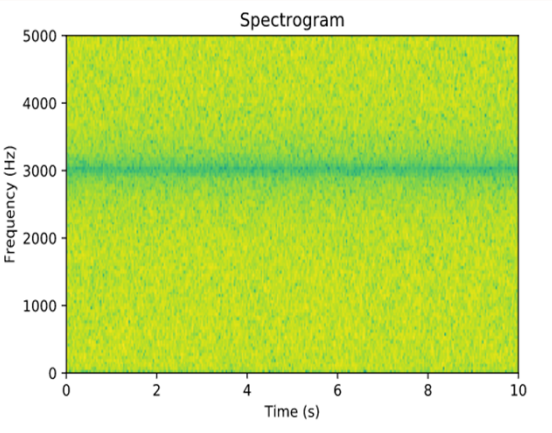
```
# define callback (#2)
def callback2(in_data, frame_count, time_info, status):
    global b, a, zi
    DSP_IO.DSP_callback_tic()
    # convert byte data to ndarray
    in_data_nda = np.fromstring(in_data, dtype=np.int16)
    #***********************************************
    # DSP operations here
    # Here we apply a linear filter to the input
    x = in_data_nda.astype(float32)
    #y = x
    # The filter state/(memory), zi, must be maintained from frame-to-frame
    y, zi = signal.lfilter(b,a,x,zi=zi) # for FIR or simple IIR
    #y, zi = signal.sosfilt(sos,x,zi=zi) # for IIR use second-order sections
    #***********************************************
    # 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
```

Filter Callback

```
Pxx, F = ss.my_psd(DSP_IO.data_capture,2**13,44100);
fir_d.freqz_resp_list([b],[a],'dB',44100)
plot(F,10*log10(Pxx/max(Pxx))+3,'g') # Normalize by the max PSD
ylim([-60,5])
xlim([0,20e3])
legend(('Theoretical','Measured'),loc='best')
grid();
```



Measured and Theoretical Response



Spectrogram

Jupyter Notebooks

The Jupyter Notebook is the ideal environment for the Pyaudio_helper. There will soon be several Jupyter Notebooks available from GitHub with interactive pyaudio_helper examples. We have them available now for demos. Some of these examples include:

- Real-Time DSP
- DSP Playback
- Parameter Control
- Volume Control
- Filtering
- Stereo Processing
- Panning Effects
- Digital Delay
- Object-Oriented Processing
- Modulation Effects