Lab4_notebook_sample

April 14, 2022

1 Lab 4 Sample Notebook

[1]:
```
%pylab inline
#%pylab notebook
#%matplotlib qt
import sk_dsp_comm.sigsys as ss
# import sk_dsp_comm.pyaudio_helper as pah
import pyaudio_helper_old as pah
import sk_dsp_comm.fir_design_helper as fir_d
import sk_dsp_comm.digitalcom as dc
import scipy.signal as signal
import ipywidgets as widgets
from IPython.display import Audio, display
from IPython.display import Image, SVG
```

```
Populating the interactive namespace from numpy and matplotlib
```

[2]:
```
pylab.rcParams['savefig.dpi'] = 100 # default 72
#pylab.['figure.figsize'] = (6.0, 4.0) # default (6,4)
#%config InlineBackend.figure_formats=['png'] # default for inline viewing
%config InlineBackend.figure_formats=['svg'] # SVG inline viewing
#%config InlineBackend.figure_formats=['pdf'] # render pdf figs for LaTeX
#<div style="page-break-after: always;"></div> #page breaks after in Typora
```

1.1 Some Helper Functions

[3]:
```
class loop_audio_contig(object):
    """
    Loop a signal ndarray contiguousy during playback.
    Optionally start_offset samples into the array.
    Array may be 1D (one channel) or 2D (two channel, Nsamps by 2)

    Mark Wickert March 2019
    """
    def __init__(self,x,start_offset = 0):
        """
        Create a 1D or 2D array for audio looping
```

```
        """
        self.n_chan = x.ndim
        if self.n_chan == 2:
            # Transpose if data is in rows
            if x.shape[1] != 2:
                x = x.T
        self.x = x
        self.x_len = x.shape[0]
        self.loop_pointer = start_offset


    def get_samples(self,frame_count):
        """

        """
        n_mod = mod(arange(frame_count)+self.loop_pointer,self.x_len)
        if self.n_chan == 1:
            buffer = self.x[n_mod]
        else:
            buffer = self.x[n_mod,:]
        self.loop_pointer = n_mod[-1] + 1
        return buffer
```

[4]:

```
def sccs_bit_sync(y,Ns):
    """
    rx_symb_d,clk,track = sccs_bit_sync(y,Ns)

    //////////////////////////////////////////////////////
     Symbol synchronization algorithm using SCCS
    //////////////////////////////////////////////////////

    Inputs
    ======
    y: baseband NRZ data waveform
    Ns: nominal number of samples per symbol

    Returns
    =======
    rx_symb_d: The recovered binary 0/1 symbols
    clk: The clock signal
    track: The sampling clock edge relative to [0,Ns-1] possible timing values

    Reference
    =========
    K. Chen and J. Lee, "A Family of Pure Digital Signal Processing Bit
    Synchronizers," IEEE Trans. on Commun., Vol. 45, No. 3, March 1997,
    pp. 289-292.
```

```python
Mark Wickert April 2014, Updated March 2019
"""
# decimated symbol sequence for SEP
rx_symb_d = np.zeros(int(np.fix(len(y)/Ns))+1)
track = np.zeros(int(np.fix(len(y)/Ns))+1)
bit_count = -1
y_abs = np.zeros(len(y))
clk = np.zeros(len(y))
k = Ns #initial 1-of-Ns symbol synch clock phase
# Sample-by-sample processing required
for i in range(len(y)):
    #y_abs(i) = abs(round(real(y(i))))
    if i >= Ns-1: # do not process first Ns samples
        # Collect timing decision unit (TDU) samples
        y_abs[i] = np.abs(np.sum(y[i-Ns+1:i+1]))
        # Update sampling instant and take a sample
        # For causality reason the early sample is 'i',
        # the on-time or prompt sample is 'i-1', and
        # the late sample is 'i-2'.
        if (k == 0):
            # Load the samples into the 3x1 TDU register w_hat.
            # w_hat[1] = late, w_hat[2] = on-time; w_hat[3] = early.
            w_hat = y_abs[i-2:i+1]
            bit_count += 1
            if w_hat[1] != 0:
                if w_hat[0] < w_hat[2]:
                    k = Ns-1
                    clk[i-2] = 1
                    rx_symb_d[bit_count] = y[i-2-int(np.round(Ns/2))-1]
                elif w_hat[0] > w_hat[2]:
                    k = Ns+1
                    clk[i] = 1
                    rx_symb_d[bit_count] = y[i-int(np.round(Ns/2))-1]
                else:
                    k = Ns
                    clk[i-1] = 1
                    rx_symb_d[bit_count] = y[i-1-int(np.round(Ns/2))-1]
            else:
                k = Ns
                clk[i-1] = 1
                rx_symb_d[bit_count] = y[i-1-int(np.round(Ns/2))]
            track[bit_count] = np.mod(i,Ns)
    k -= 1
# Trim the final outputs to bit_count
rx_symb_d = rx_symb_d[:bit_count]
track = track[:bit_count]
```

```
    return rx_symb_d, clk, track
```

2 Introduction

A simplified block diagram of PyAudio *streaming-based* (nonblocking) signal processing when using `pyaudio_helper` and `ipython` widgets.

[5]:
```
pah.available_devices()
```

[5]:
```
{0: {'name': 'USB Audio Device', 'inputs': 1, 'outputs': 2},
 1: {'name': 'MacBook Pro Microphone', 'inputs': 1, 'outputs': 0},
 2: {'name': 'MacBook Pro Speakers', 'inputs': 0, 'outputs': 2}}
```

2.1 Stream NRZ Data Bits over the Link as an M-Sequence

[30]:
```
M = 5
N_bits = 2**M - 1
data = ss.pn_gen(N_bits,M)
x_NRZ, b_pulse = ss.nrz_bits2(data,20,pulse='rect')
```

Provide gain sliders on the tx and rx streams

[7]:
```
tx_gain = widgets.FloatSlider(description = 'Tx Gain',
                continuous_update = True,
                value = 1.0,
                min = 0.0,
                max = 2.0,
                step = 0.01,
                orientation = 'horizontal')

rx_gain = widgets.FloatSlider(description = 'Rx Gain',
                continuous_update = True,
                value = 1.0,
                min = 0.0,
                max = 2.0,
                step = 0.01,
                orientation = 'horizontal')

# widgets.HBox([tx_gain, rx_gain])
```

[13]:
```
def callback(in_data, frame_count, time_info, status):
    global DSP_IO, tx_gain, rx_gain, x_loop_mono, rx_buf
    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
```

```
    # The input samples will contain the input from the FSK demod output
    x_rx = in_data_nda.astype(float32)
    # Use a loop object as a source of mono NRZ waveform contiguous samples
    # Note since wave files are scaled to [-1,1] we are scaling to
    # rescale to the dynamic range of int16
    #new_frame = x_loop_stereo.get_samples(frame_count)
    new_frame = x_loop_mono.get_samples(frame_count)
    x_tx = 20000*new_frame
    #************************************************
    # DSP operations here
    y = x_tx*tx_gain.value # The transmitted NRZ bit stream
    y_rx = x_rx*rx_gain.value # The received NRZ bit stream

    # Save data for later analysis
    # accumulate a new frame of samples
    # Note: We are doing a 1 channel capture, but the attributes of
    # DSP_IO contain thge means to capture left (tx) and right (rx) channels
    DSP_IO.DSP_capture_add_samples_stereo(y,y_rx)
    #************************************************
    # 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
```

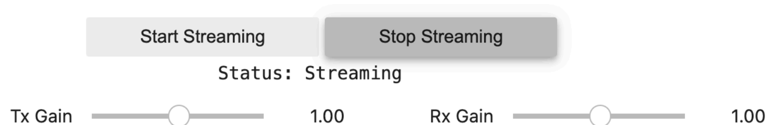
[31]:
```
T_record = 5 # in s; 0 <==> infinite, but no capture, typical 5 to 30s
x_loop_mono = loop_audio_contig(x_NRZ)
DSP_IO = pah.DSP_io_stream(callback,0,0,fs=48000,Tcapture=2*T_record)
DSP_IO.interactive_stream(2*T_record,1)
widgets.HBox([tx_gain,rx_gain])
```

```
interactive(children=(ToggleButtons(description=' ', index=1, options=('Start␣
↪Streaming', 'Stop Streaming'), v…
```

```
HBox(children=(FloatSlider(value=1.0, description='Tx Gain', max=2.0, step=0.
↪01), FloatSlider(value=1.0, descr…
```

[45]:
```
# Insert a placeholder image of pyaudio_helper GUI the above code creates
Image('pyaudio_helper_GUI_screenshot.png',width='1%')
```

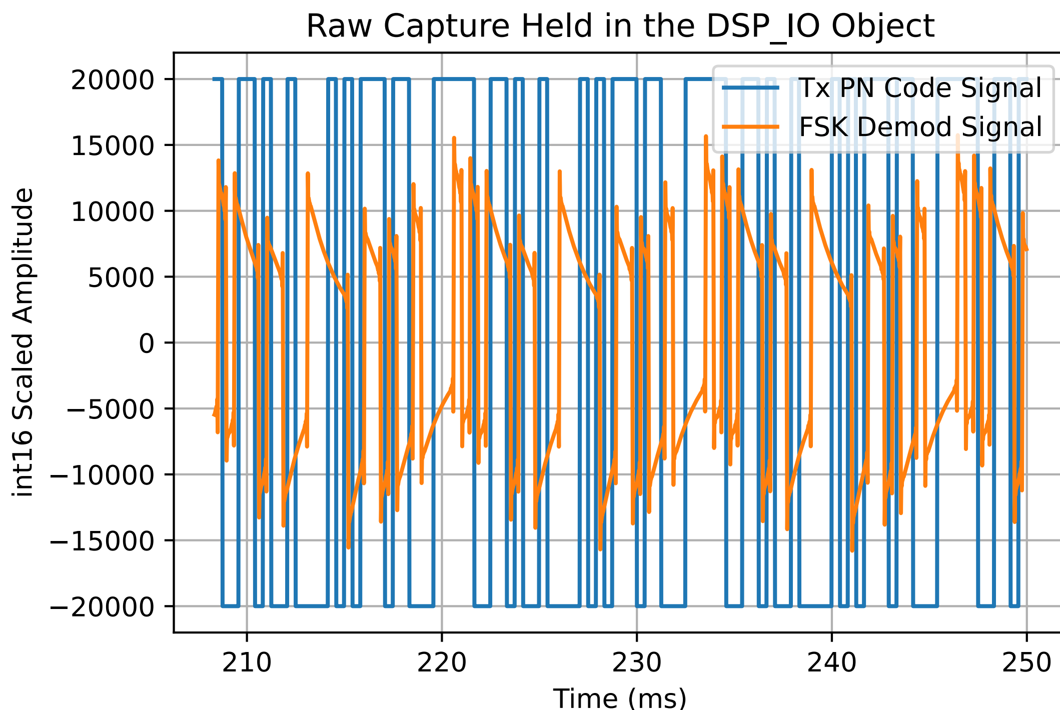
[45]:



The *L Gain* slider adjusts the transmit digital message level into the **External** modulation back panel input of the Keysight 33600A generator. The *R Gain* slider adjusts the signal level coming from the USB audio card mic input into the `DSP_IO.data_capture` buffer.

Take a Quick Look at the Capture Move down the capture as PyAudio I/O latency will likely make the demodulated bit stream brought through the mic input of the sound card lag by as much as 165 ms .

[35]:
```
fs = 48000
Nstart = 10000 # Wait about 163 ms for the Rx signal to arrive in the␣
↪data_capture buffer
Nspan = 2000
t_capture = arange(0,len(DSP_IO.data_capture_left))/fs*1000 # ms
plot(t_capture[Nstart:Nstart+Nspan],DSP_IO.data_capture_left[Nstart:
↪Nstart+Nspan])
plot(t_capture[Nstart:Nstart+Nspan],1.5*DSP_IO.data_capture_right[Nstart:
↪Nstart+Nspan])
title(r'Raw Capture Held in the DSP_IO Object')
ylabel(r'int16 Scaled Amplitude')
xlabel(r'Time (ms)')
legend((r'Tx PN Code Signal',r'FSK Demod Signal'),loc='upper right')
grid();
```



Gain Level the Captures In preparation for saving the capture to a `.wav` file for archiving, we scale both the Tx and Rx waveforms to an amplitude that lies on the interval $(-1, 1)$.

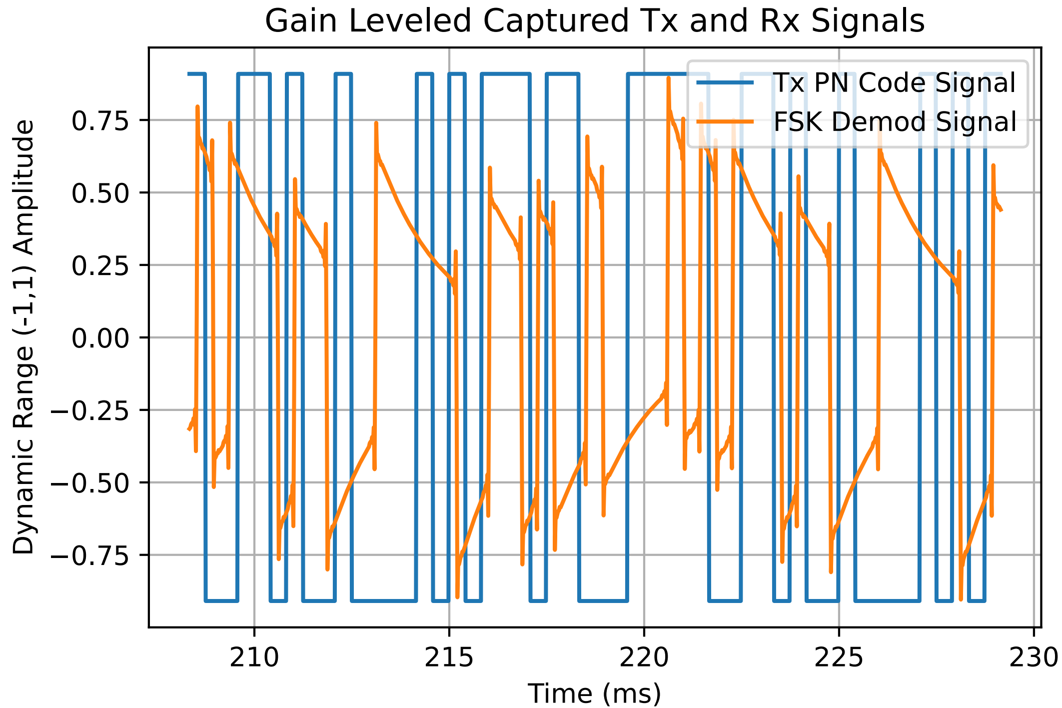
[36]:
```
left_right_2400bps = hstack((array([DSP_IO.data_capture_left]).T/(1.
↪1*max(DSP_IO.data_capture_left)),
                             array([DSP_IO.data_capture_right]).T/(1.
↪1*max(DSP_IO.data_capture_right))))
# ss.to_wav('left_right_2400bps_lcap_m15dBm.wav',48000,left_right_2400bps) #␣
↪Need to scale to (-1,1) for wave
ss.to_wav('mono_mac_test.wav',48000,left_right_2400bps) # Need to scale to␣
↪(-1,1) for wave
```

Load Wave File Archive (if needed)

[37]:
```
# fs, left_right_2400bps = ss.from_wav('left_right_2400bps5.wav') #files below␣
↪not in sample ZIP
fs, left_right_2400bps = ss.from_wav('mono_mac_test.wav') #files below not in␣
↪sample ZIP
# fs, left_right_2400bps = ss.from_wav('left_right_2400bps_m15dBm.wav')
#fs, left_right_2400bps = ss.from_wav('left_right_2400bps_lcap_m15dBm.wav')
#fs, left_right_2400bps = ss.from_wav('left_right_2400bps_lcap_m17dBm.wav')
#fs, left_right_2400bps = ss.from_wav('left_right_2400bps_lcap_LB.wav')
print('Capture period from sample count = %4.2f s' % (left_right_2400bps.
↪shape[0]/48000,))
```

```
Capture period from sample count = 5.01 s
```

[38]:
```
fs = 48000
Nstart = 10000
Nspan = 1000
t_capture = arange(0,left_right_2400bps.shape[0])/48000*1000 # ms
plot(t_capture[Nstart:Nstart+Nspan],left_right_2400bps[Nstart:Nstart+Nspan,0])
plot(t_capture[Nstart:Nstart+Nspan],left_right_2400bps[Nstart:Nstart+Nspan,1])
title(r'Gain Leveled Captured Tx and Rx Signals')
ylabel(r'Dynamic Range (-1,1) Amplitude')
xlabel(r'Time (ms)')
legend((r'Tx PN Code Signal',r'FSK Demod Signal'),loc='upper right')
grid();
```



Note: There is serious *baseline wander* (BW) due to the coupling capacitor at the USB sound card mic input. A correction algorithm will be introduced shortly.

2.2 Baseline Wander Correction

See Baseline wander - EECS: www-inst.eecs.berkeley.edu for more detail.

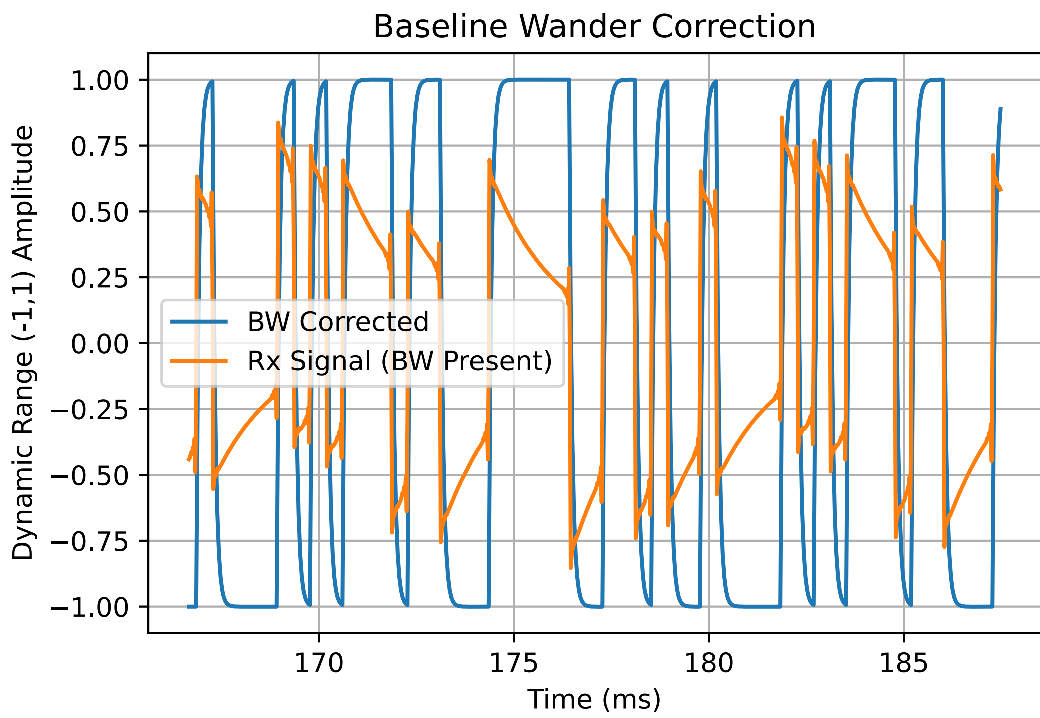
A simple baseline wander correction is implemented on the scaled mic input signal. The algorithm applied a small amount of positive feedback using a 1st-order filter with feedback gain `G_BW` and filter coefficient `alpha`. The filter is a first-order lowpass applied to a hard-limited version of the input, i.e., $x_{\text{sgn}}[n] = \text{sgn}\{x[n]\}$ is filtered by

$$H(z) = \frac{Y_{\text{sgn}}(z)}{X_{\text{sgn}}(z)} = \frac{1-\alpha}{1-\alpha z^{-1}} \tag{1}$$

$$y_{\text{sgn}}[n] = \alpha y_{\text{sgn}}[n-1] + (1-\alpha)x_{\text{sgn}}[n] \tag{2}$$

[39]:
```
Nsamps = left_right_2400bps.shape[0]
G_BW = 1000
alpha = 0.25
y_old = 0
y_BW_cor = zeros(Nsamps) # Baseline wander corrected signal
for k in range(Nsamps):
    x_sign = sign(left_right_2400bps[k,1]) #BW correct column 1
    y_sign = (1-alpha)*y_old + alpha*x_sign
    y_old = y_sign
```

```
    y_BW_cor[k] = (left_right_2400bps[k,1] + G_BW*y_sign)/G_BW

Nstart = 8000 #10000
Nspan = 1000
t_capture = arange(0,left_right_2400bps.shape[0])/48000*1000 # ms
plot(t_capture[Nstart:Nstart+Nspan],y_BW_cor[Nstart:Nstart + Nspan])
plot(t_capture[Nstart:Nstart+Nspan],left_right_2400bps[Nstart:Nstart + Nspan,1])
title(r'Baseline Wander Correction')
ylabel(r'Dynamic Range (-1,1) Amplitude')
xlabel(r'Time (ms)')
legend((r'BW Corrected',r'Rx Signal (BW Present)'))
grid();
```



Note: It takes about 163 ms before the received signal shows up in the buffer. This is a `PyAudio`/PC sound system property.

2.3 Bit Synchronization and Bit Error Probability Estimation

To characterize the FSK Tx to Rx performance we first need to manage clock drift and then compare the transmitted bit pattern with the received bit pattern, forming the ratio of bit errors to total bits processed.

2.3.1 Bit Synchronization

The received bit stream recovered by demodulating the FSK signal on the narrow band radio board and then capturing back to digital form via `pyaudio_helper`, will have have *clock drift*. This makes the original 20 bits per sample Tx signal and the Rx have slowly slide past one another. To fix this problem we use an algorithm found in the **Helper Functions** at the top of this notebook. The doc string is given below:

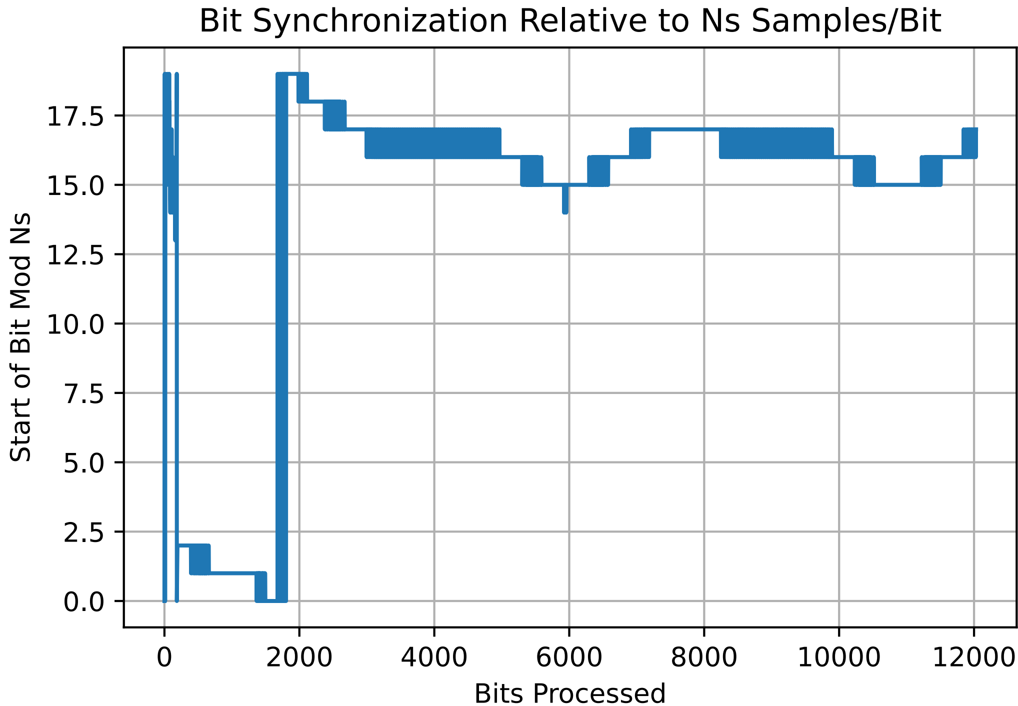
```
def sccs_bit_sync(y,Ns):
    """
    rx_symb_d,clk,track = sccs_bit_sync(y,Ns)

    //////////////////////////////////////////////////////
     Symbol synchronization algorithm using SCCS
    //////////////////////////////////////////////////////

    Inputs
    ======
    y: baseband NRZ data waveform
    Ns: nominal number of samples per symbol

    Returns
    =======
    rx_symb_d: The recovered binary 0/1 symbols
    clk: The clock signal
    track: The sampling clock edge relative to [0,Ns-1] possible timing values

    Mark Wickert April 2014
    """
```

The clock drift is not that large, but still must be dealt with. The nominal number of samples per bit is 20, as the serial bit rate is 2400 bits/sec sampled at $f_s = 48000$ Hz.

The input to the SCCS bit synch is converted to ± 1 values before processing:

[40]:
```
rx_symb_d, clk, trac = sccs_bit_sync(sign(y_BW_cor),20)
```

[41]:
```
plot(trac)
title(r'Bit Synchronization Relative to Ns Samples/Bit')
ylabel(r'Start of Bit Mod Ns')
xlabel(r'Bits Processed')
grid();
```



Note: Tracking is not *locked* until a signal is actually present. Once the SCCS [3] is tracking it tends to *hunt* around the optimal timing instant modulo `Ns`. It generally varies over three values, say 2,3,4, which for the case of `Ns=20` may straddle the wrapping point, i.e., 18,19,0 in a modulo `Ns` sense.

2.3.2 Bit Error Probability (BEP)

The transmitted NRZ bits at `Ns` samples per bit are first down sampled to just one sample per bit. The bits returned from `sccs_bit_sync()` in `rx_symb_d` then compared with `tx_bits`. The time delay due to PyAudio and other analog processing the Tx-Rx link means that the two bit patterns need to first be brought into alignment. The function `sk_dsp_comm.digitalcom.bit_errors()` takes care of the alignment problem using crosscorrelation. This assumes that the bit errors are not so numerous so as to make a correlation peak pop up. Since the transmit bit stream is repeating *m*-sequence, the bit streams may not be that far out of alignment, modulo the sequence period.

[42]:
```
tx_bits = int64(ss.downsample(sign(left_right_2400bps[:,0]),20)+1)//2
len(tx_bits)
```

[42]: 12032

Here we trim away the 500 bits which corresponds to the delayed arrival of the received signal (just noise). The `bit_errors` function from `sk_dsp_comm` automatically aligns the transmit waveform with the received signal to allow bit error counting to take place. Because of the relatively high signal-to-noise ratio (SNR) of the received signal no errors occur.

[43]:
```
N_bits, N_errors = dc.bit_errors(tx_bits,int16((rx_symb_d[500:]+1)/2))
print('N_bits = %d, N_errors = %d, BEP = %1.2e' % (N_bits,N_errors, N_errors/
↪N_bits))
```

```
N_bits = 11519, N_errors = 0, BEP = 0.00e+00
```

2.4 FSK Modulation Theory

In Chapter 4 of [1] you learn that an frequency modulated carrier takes the form

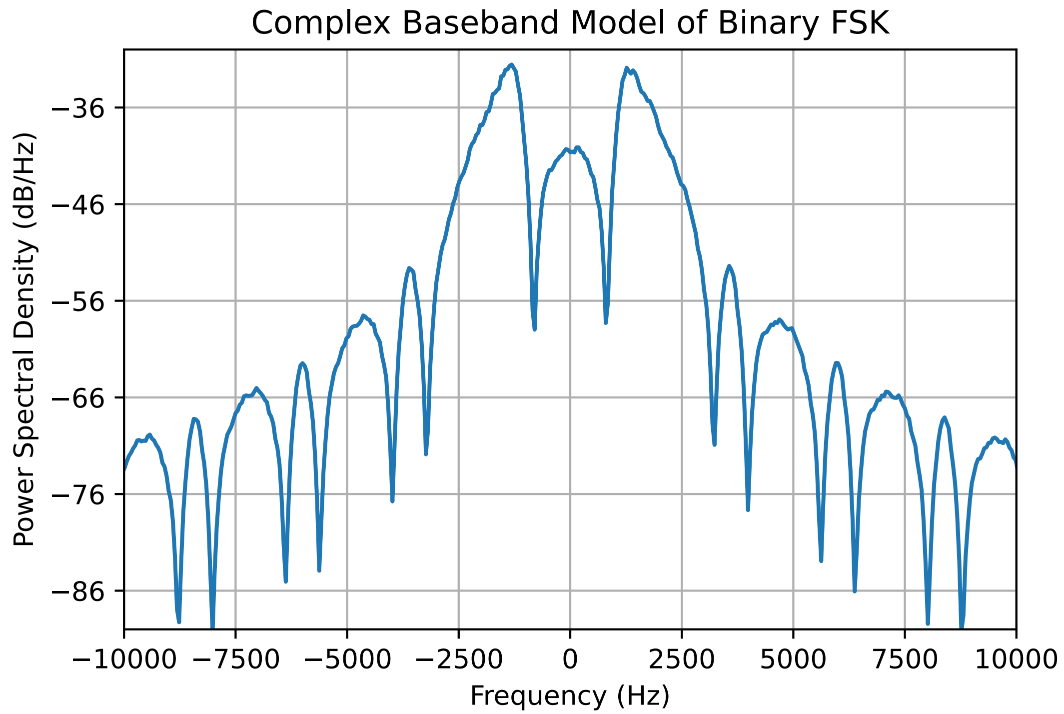
$$x_c(t) = A_c \cos\left[2\pi f_c t + 2\pi f_d \int^t m(\lambda)\, d\lambda\right]$$

where A_c is the carrier amplitude, $m(t)$ the message signal, here a NRZ data stream, and f_d is the modulator deviation contant having units of Hz per unit of $m(t)$. In a discrete-time implementation and with the carrier at f_0, *complex baseband* FM takes the form

$$x_c[n] = A_c \exp\left[2\pi f_c \sum_{k=0}^{n} m[k] \frac{1}{f_s}\right]$$

where the integration is replaced by the running sum, `cumsum` in Python's numpy.

[15]:
```
fs = 48000
fc = 0
M = 5 # try 10 to see a striking difference between long and short patterns
N_bits = 10000
n = arange(0,20*N_bits)
fd = 2000 #Hz/v
data = ss.pn_gen(N_bits,M)
# x_NRZ, b_pulse = ss.nrz_bits2(data,20,pulse='rect') # repeating M-seq bits
x_NRZ, b_pulse, data = ss.nrz_bits(N_bits,20,pulse='rect') # random bits
m = fd*.79/2*x_NRZ
xc = exp(1j*2*pi*2*cumsum(m)/fs)*exp(1j*2*pi*fc/fs*n) # 2 next to cumsum for␣
↪doubler
psd(xc,2**10,48000);
title(r'Complex Baseband Model of Binary FSK')
# title(r'Complex Baseband Model of Binary FSK: M = %d' % M)
xlabel(r'Frequency (Hz)')
xlim([-10000,10000])
ylim([-90,-30]);
```



2.5 References

1. Rodger Ziemer and William Tranter, Principles of Communications, 8th edition, Wiley, 2014.
2. Baseline wander - EECS: www-inst.eecs.berkeley.edu
3. K. Chen and J. Lee, "A Family of Pure Digital Signal Processing Bit Synchronizers," IEEE Trans. on Commun., Vol. 45, No. 3, March 1997, pp. 289–292.

[ ]: