Lab5_notebook_sample

April 13, 2022

1 Lab 5 Sample Notebook

[1]:
```
%pylab inline
#%pylab notebook
#%matplotlib qt
import sk_dsp_comm.sigsys as ss
import sk_dsp_comm.fir_design_helper as fir_d
import sk_dsp_comm.digitalcom as dc
import scipy.signal as signal
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

1.1.1 Complex Baseband Discriminator

[3]:
```
# This function can be found in
# sk_dsp_comm.rtlsdr_helper
def discrim(x):
    """
    function disdata = discrim(x)
    where x is an angle modulated signal in complex baseband form.

    Mark Wickert
    """
    X=np.real(x)        # X is the real part of the received signal
    Y=np.imag(x)        # Y is the imaginary part of the received signal
    b=np.array([1, -1]) # filter coefficients for discrete derivative
```

```
    a=np.array([1, 0])  # filter coefficients for discrete derivative
    derY=signal.lfilter(b,a,Y)  # derivative of Y,
    derX=signal.lfilter(b,a,X)  #    "          X,
    disdata=(X*derY-Y*derX)/(X**2+Y**2)
    return disdata
```

1.2 Collecting VCO Data

Use the Jupyter notebook to record VCO tuning data, plot it, fit a strainght line, and experimentally determining the deviation constant f_d.

1.2.1 Plotting VCO Data with a Linear Fit

The relevant functions in `numpy` are `polyfit()` and `polyval()`. An example using simulated data is given below.

[4]:
```
v_sim = arange(-2,2+.25,.25) # in Volts
f_sim = 50+v_sim*3 + 0.1*randn(len(v_sim)) # in MHz
```

[5]:
```
v_sim
```

[5]:
```
array([-2.  , -1.75, -1.5 , -1.25, -1.  , -0.75, -0.5 , -0.25,  0.  ,
        0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
```

[6]:
```
f_sim
```

[6]:
```
array([44.08761993, 44.77776941, 45.71176776, 46.06475678, 47.05802433,
       47.73678742, 48.62426294, 49.21112504, 50.00798982, 50.76265162,
       51.58891542, 52.13240571, 53.04184459, 53.84725518, 54.43262151,
       55.21999881, 55.98214   ])
```

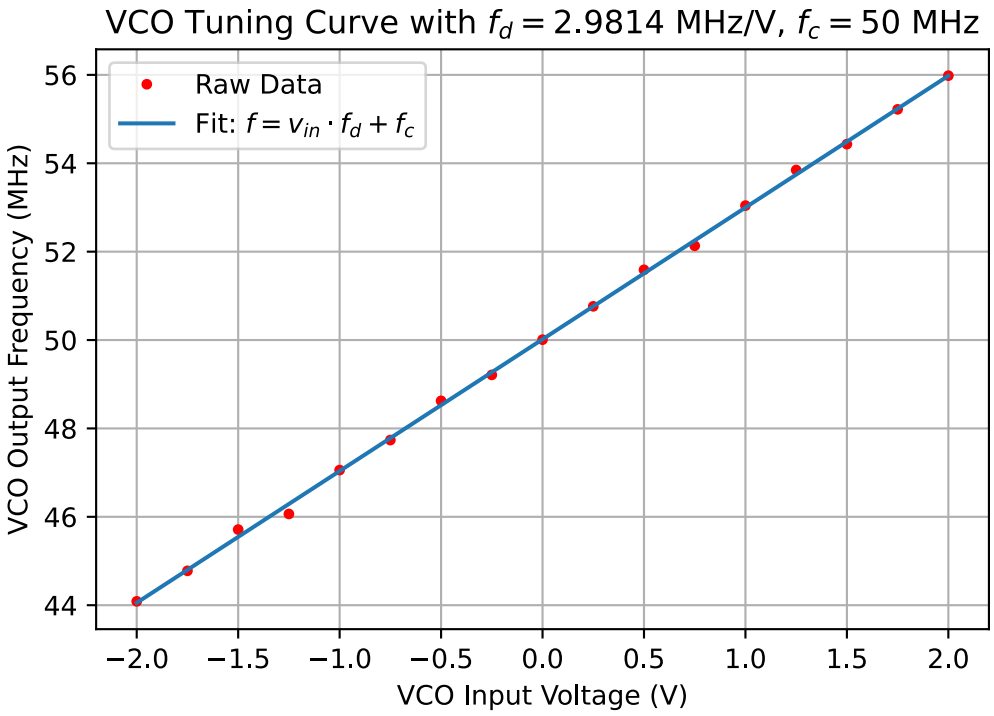
[7]:
```
# Save data in the Python lists:
#v_data = [-2.0, -1.75, -1.5, -1.25, ...] # voltage data in Volts
#f_data = [44.072, 44.315, 44.545, ...] # Frequency data in MHz
# Here we load the simulated data from above as a placeholder
v_data = v_sim
f_data = f_sim
p = polyfit(v_data,f_data,1)
f_data_fit = polyval(p,v_data)
print('Slope = %4.2f MHz/V, Offset = %4.2f MHz' % (p[0],p[1]))
```

```
Slope = 2.98 MHz/V, Offset = 50.02 MHz
```

The polynomial fit to the data is held in the array `p`, which is a first degree polynomial relating to the model

$$\begin{aligned}\hat{f}_{\text{VCO}} &= f_d \cdot v_{\text{in}} + f_c \\ &= p[0]v_{\text{in}} + p[1]\end{aligned}$$

[8]:
```
plot(v_data,f_data,'r.')
plot(v_data,f_data_fit)
xlabel(r'VCO Input Voltage (V)')
ylabel(r'VCO Output Frequency (MHz)')
title(r'VCO Tuning Curve with $f_d = 2.9814$ MHz/V, $f_c = 50$ MHz')
legend((r'Raw Data',r'Fit: $f = v_{in}\cdot f_d + f_c$'))
grid();
```



Note: When the Keysight 33600A is configured as a VCO, it will have essentially a *perfect* or linear tuning curve. The theoretical values of f_d and f_c are formed from the generator parameters `Freq Dev`, `Mod In` value 1V or 5V, and the center frequency `fc`. Can you deduce the theoretical expression?

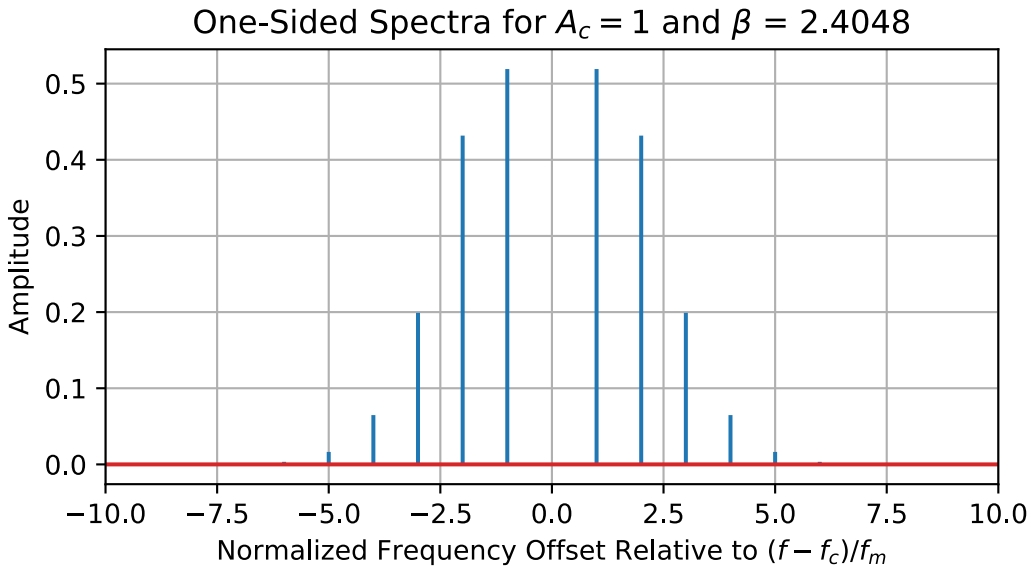
1.3 Sinusoidal Frequency Modulation Theory

[9]:
```
import scipy.special as special
```

1.3.1 Plot Magnitude Spectra

The spectrum plot below is plotting the single-sided line spectra as a linear plot. You should be able to convert this to a dBm plot.

[10]:
```
beta = 2.4048
#beta = 25
Nterms = 100
idx = arange(-Nterms,Nterms+1)
Yn = special.jn(idx,beta)
figure(figsize=(6,3))
stem(idx,abs(Yn),markerfmt=" ")
ylabel(r'Amplitude')
xlabel(r'Normalized Frequency Offset Relative to $(f-f_c)/f_m$')
title(r'One-Sided Spectra for $A_c=1$ and $\beta$ = %2.4f' % beta);
xlim([-10,10])
grid();
```



1.4 Frequency Modulation Simulation

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

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

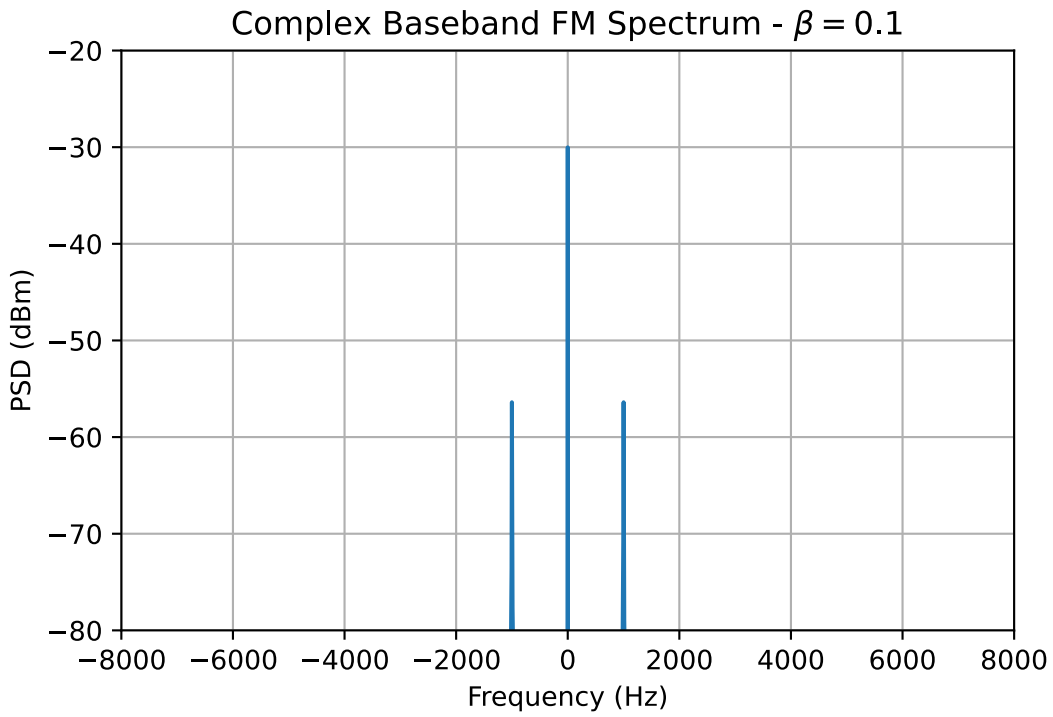
where the integration is replaced by the running sum, `cumsum` in Python's numpy. In the plot below uses a log scale and is calibrated in dBm. Since the signal is a complex sinusoid you can

view the spectrum as a single-sided spectrum for the case of a real sinusoid with the plot being made with $f \rightarrow (f - f_c)$.

1.4.1 Single Sinusoid

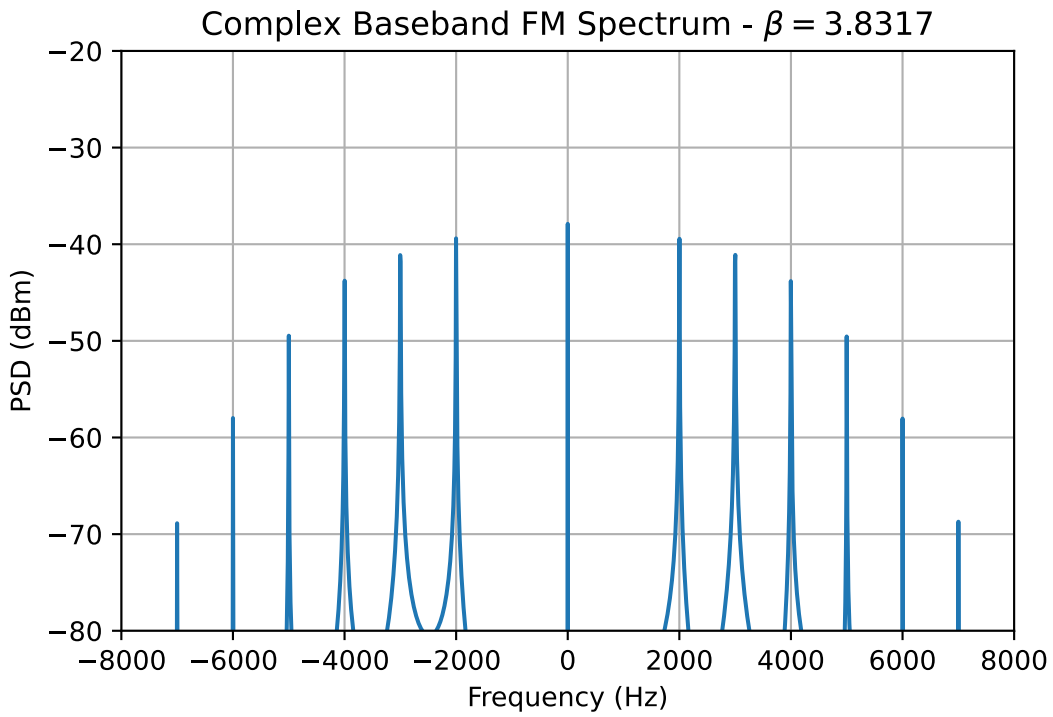
Narrow Band β Small

[12]:
```
fs = 100000
fc = 0
N_samp = 100000
t = arange(N_samp)/fs
Pc_dBm = -30
fd = 1000 #Hz/v
Am = 0.1
fm = 1000
m = Am*cos(2*pi*fm*t) # sinusoid
#m = Am*sign(cos(2*pi*fm*t)) # square wave
xc = sqrt(10**((Pc_dBm-30)/10))*exp(1j*2*pi*fd*cumsum(m)/fs)*exp(1j*2*pi*fc*t)
f, Sx = ss.simple_sa(xc,N_samp,2**14,fs)
plot(f,10*log10(Sx)+30)
#psd(xc,2**12,48000);
title(r'Complex Baseband FM Spectrum - $\beta = 0.1$')
ylabel(r'PSD (dBm)')
xlabel(r'Frequency (Hz)')
xlim([-8000,8000])
ylim([-80,-20]);
grid();
```



Narrow Band $\beta = 3.8317$ To Eliminate the First Sideband Pair

[13]:
```
fs = 100000
fc = 0
N_samp = 100000
t = arange(N_samp)/fs
Pc_dBm = -30
fd = 1000 #Hz/v
Am = 3.8317
fm = 1000
m = Am*cos(2*pi*fm*t) # sinusoid
#m = Am*sign(cos(2*pi*fm*t)) # square wave
xc = sqrt(10**((Pc_dBm-30)/10))*exp(1j*2*pi*fd*cumsum(m)/fs)*exp(1j*2*pi*fc*t)
f, Sx = ss.simple_sa(xc,N_samp,2**14,fs)
plot(f,10*log10(Sx)+30)
#psd(xc,2**12,48000);
title(r'Complex Baseband FM Spectrum - $\beta = 3.8317$')
ylabel(r'PSD (dBm)')
xlabel(r'Frequency (Hz)')
xlim([-8000,8000])
ylim([-80,-20]);
grid();
```



1.4.2 Post Processing FieldFox Spectrum `.csv` Files

A simple Python class was written to import spectrum `.csv` files and perform calculations and plot the measured power spectrum.

Raw Input `csv` from FieldFox

[8]:
```
# Skip the first 32 rows, then skip the last row that contains 'END'
f_SA, Sx_SQ_FM_10K20K_m15dBm = loadtxt('SQ_FM_10K20K.
 ↪csv',delimiter=',',skiprows=32,
                usecols=(0,1),comments='END',unpack=True)
```

Automating `.csv` File Processing with the class `FieldFox_capture` Fell free to modify the class to better meet your needs. This is a first pass at making a helper tool.

[14]:
```
class FieldFox_capture(object):
    """
    A class for processing FieldFox spectrum capture .csv files

    Mark Wickert April 2019
    """

    def __init__ (self, capture_file_name):
        """
        Initialize the object by reading the csv file into class attributes
        """
        # Skip the first 32 rows, then skip the last row that contains 'END'
        f_SA, Sx_dBm = loadtxt(capture_file_name,delimiter=',',skiprows=32,
                usecols=(0,1),comments='END',unpack=True)
        self.f_SA = f_SA # in Hz
        self.Sx_dBm = Sx_dBm # in dBm
        self.Sx_mW = 10**(Sx_dBm/10)
        self.p_total_mW = sum(self.Sx_mW)
        self.p_total_dBm = 10*log10(sum(self.Sx_mW))
        self.Npts = len(f_SA)
        self.Df = f_SA[1] - f_SA[0]

    def spectrum_plot(self,fc = 50e6, f_units = 'MHz'):
        """
        Plot the spectrum in MHz units for now
        """
        if f_units == 'MHz':
            plot(self.f_SA/1e6,self.Sx_dBm)

            xlabel(r'Frequency (MHz)')
            ylabel(r'Power Spectrum (dBm)')
```

```
            title(r'FM Spectrum Centered on %4.1f MHz' % (fc/1e6,))
            grid()
        else:
            raise ValueError("Units must be 'MHz'")


    def total_power_dBm(self):
        """
        Calculate the total captured power in dBm
        """
        print('Total power in capture = %4.2f dBm' % self.p_total_dBm)


    def bandwidth(self,fc = 50e6):
        """
        Calculate the 98% containment bandwidth
        using the measurement data. Units hard-coded to MHz
        for now
        """
        ctr_idx = ravel(self.f_SA == fc).nonzero()
        ctr_idx = ctr_idx[0][0]
        frac_power = self.Sx_mW[ctr_idx]/self.p_total_mW
        k = 1
        while frac_power < 0.98:
            frac_power += self.Sx_mW[ctr_idx + k]/self.p_total_mW
            frac_power += self.Sx_mW[ctr_idx - k]/self.p_total_mW
            k += 1

        print('Measured 98 percent BW = %5.3f MHz' % (2*k*self.Df/1e6,))
```

Example Import See the Lab 5 Jupyter notebook sample for details on the `FieldFox_capture` class. Here the capture is at $f_c = 30$ MHz, -10 dBm, $f_m = 10$ kHz and *Freq Dev* = 50 kHz.

[15]:
```
sin_FM = FieldFox_capture('SIN_10K50K30Mfc.csv')
```

[16]:
```
sin_FM.total_power_dBm()
```

```
Total power in capture = -10.17 dBm
```

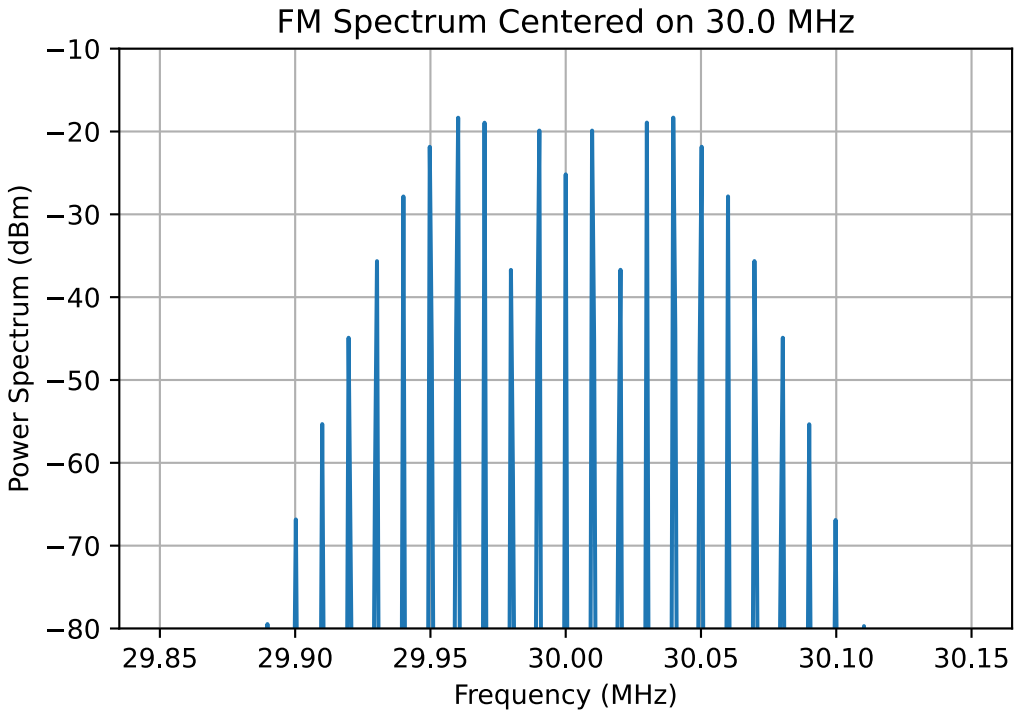
[17]:
```
sin_FM.bandwidth(fc=30e6)
```

```
Measured 98 percent BW = 0.121 MHz
```

[18]:
```
# Carson's rule check
print('Theory 98 percent BW = %4.3f MHz' % (2*(.05 + 0.01),))
```

```
Theory 98 percent BW = 0.120 MHz
```

[19]:
```
sin_FM.spectrum_plot(30e6)
ylim([-80,-10]);
```



1.4.3 Square Wave Analytical Spectrum Model (work in progress)

Working through the model in the Lab 5 reader we have

$$|X_n| = \frac{1}{T_m}|P(nf_m)| \tag{1}$$

$$= \frac{A_c}{2}\left|\text{sinc}\Big((n-\frac{\Delta f}{f_m})\frac{1}{2}\Big)e^{-j2\pi\left(n-\frac{\Delta f}{f_m}\right)\frac{1}{4}} + \text{sinc}\Big((n+\frac{\Delta f}{f_m})\frac{1}{2}\Big)e^{-j2\pi\left(n+\frac{\Delta f}{f_m}\right)\frac{3}{4}}\right| \tag{2}$$

The PSD is given by

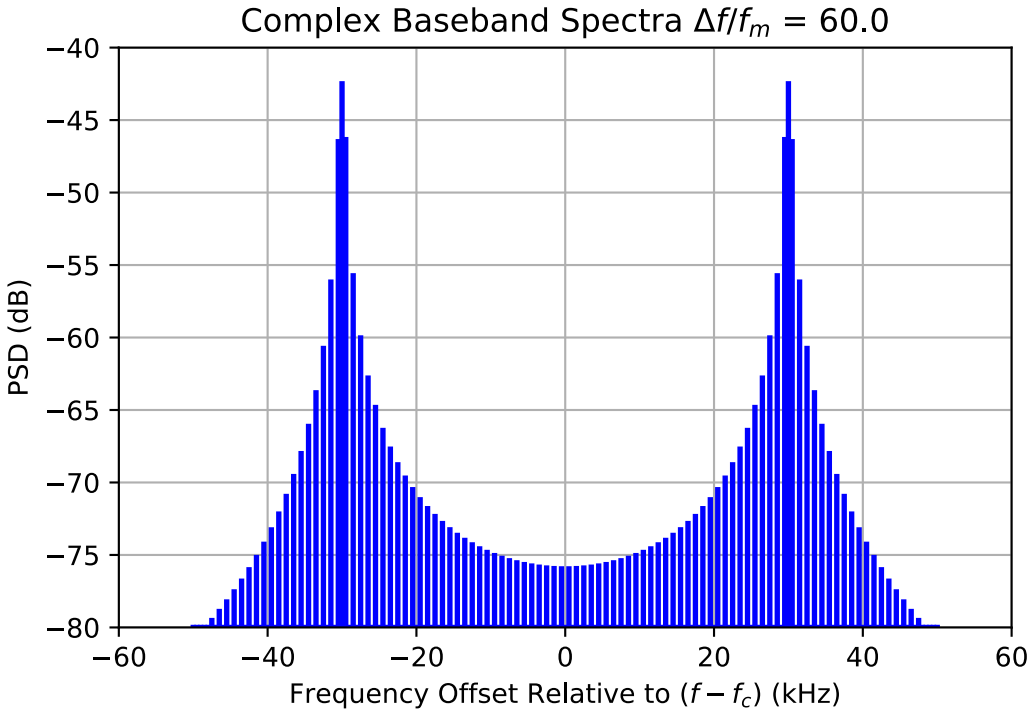
$$S_x(f) = \sum_{n=-\infty}^{\infty} |X_n|^2\,\delta(f - nf_m)$$

[20]:
```
def sqwave_Xn(n,Df2fm):
    """
    Fourier coefficient for square wave FM

    Mark Wickert April 2019
    """
```

```
    Xn = abs(sinc((n - Df2fm)/2)*exp(-1j*2*pi*(n - Df2fm)/4) + sinc((n + Df2fm)/
↪2) \
             *exp(-1j*2*pi*(n + Df2fm)*3/4))/2
    return Xn
```

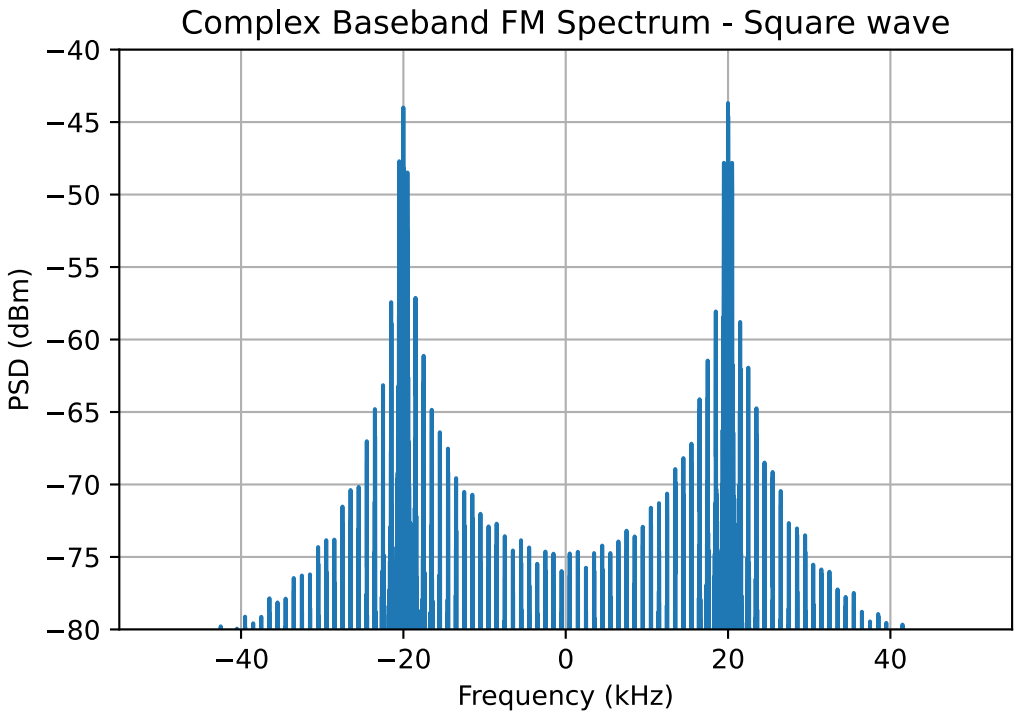
[21]:
```
Df2fm = 60
Nterms = 100
#idx = arange(-Nterms,Nterms+1)
idx = arange(0,Nterms+1)
Xn = sqwave_Xn(idx,Df2fm)
figure(figsize=(6,3))
ss.line_spectra(idx*500/1000,Xn*.015,'magdB',2,floor_dB=-80)
ylabel(r'PSD (dB)')
xlabel(r'Frequency Offset Relative to $(f-f_c)$ (kHz)')
title(r'Complex Baseband Spectra $\Delta f/f_m$ = %4.1f' % Df2fm);
#xlim([-10,10])
#grid();
```

```
<Figure size 432x216 with 0 Axes>
```



1.4.4 Square wave: Simulation

[22]:
```
fs = 100000
fc = 0
N_samp = 500000
Nfft = 2**15
t = arange(N_samp)/fs
Pc_dBm = -30
fd = 1000 #Hz/v
Am = 20
fm = 500
#m = Am*cos(2*pi*fm*t) # sinusoid
m = Am*sign(cos(2*pi*fm*t)) # square wave
xc = sqrt(10**((Pc_dBm-30)/10))*exp(1j*2*pi*fd*cumsum(m)/fs)*exp(1j*2*pi*fc*t)
f, Sx = ss.simple_sa(xc,N_samp,Nfft,fs)
plot(f/1e3,10*log10(Sx)+30)
#psd(xc,2**12,48000);
title(r'Complex Baseband FM Spectrum - Square wave')
ylabel(r'PSD (dBm)')
xlabel(r'Frequency (kHz)')
#xlim([-8000,8000])
ylim([-80,-40]);
grid();
```



1.4.5 Square wave FieldFox Capture Analysis

[ ]:

1.5 Slope Detection

Model the slope detector using a digital filter representation of the RF board 5th-order 0.05 dB passband ripple Chebyshev lowpass filter.

[ ]:

1.6 PLL Detection

PLL simulation models in Python using `sk_dsp_comm.synchronization.pll1`. The idea is to model the system shown in Figure 16 of Lab 5, as repeated below.

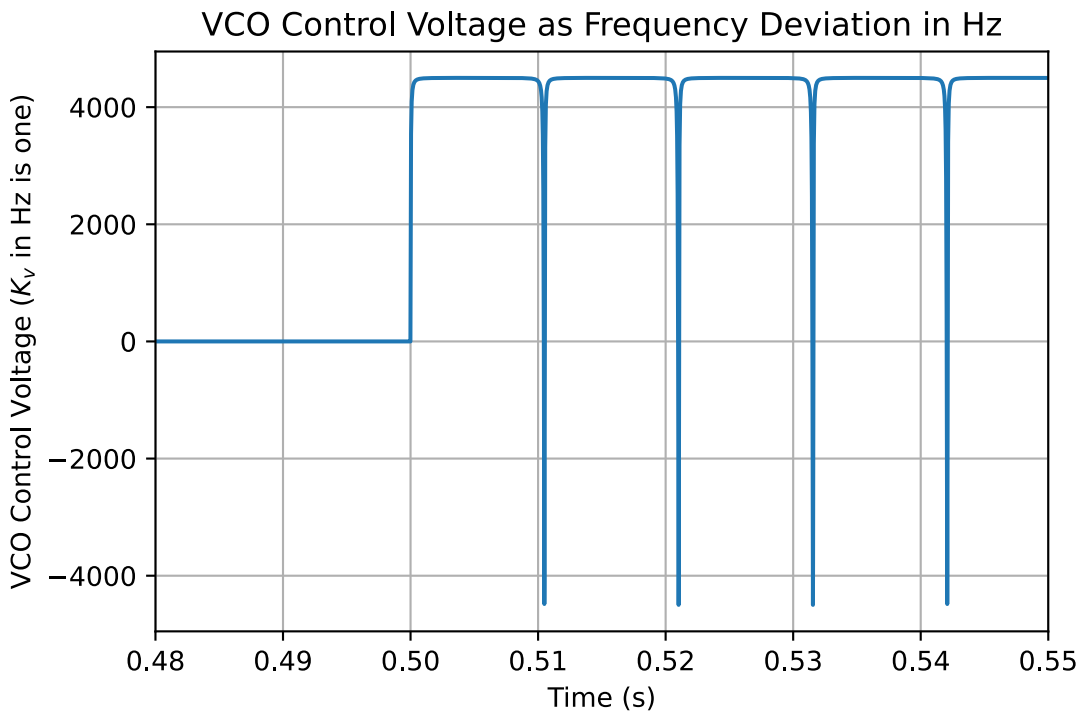
[24]:
```
import sk_dsp_comm.synchronization as pll
```

1st-Order PLL Nonlinear Simulation Three modulation types can inserted by commenting and uncommenting code: frequency step, sinusoidal FM and sqaure wave FM.

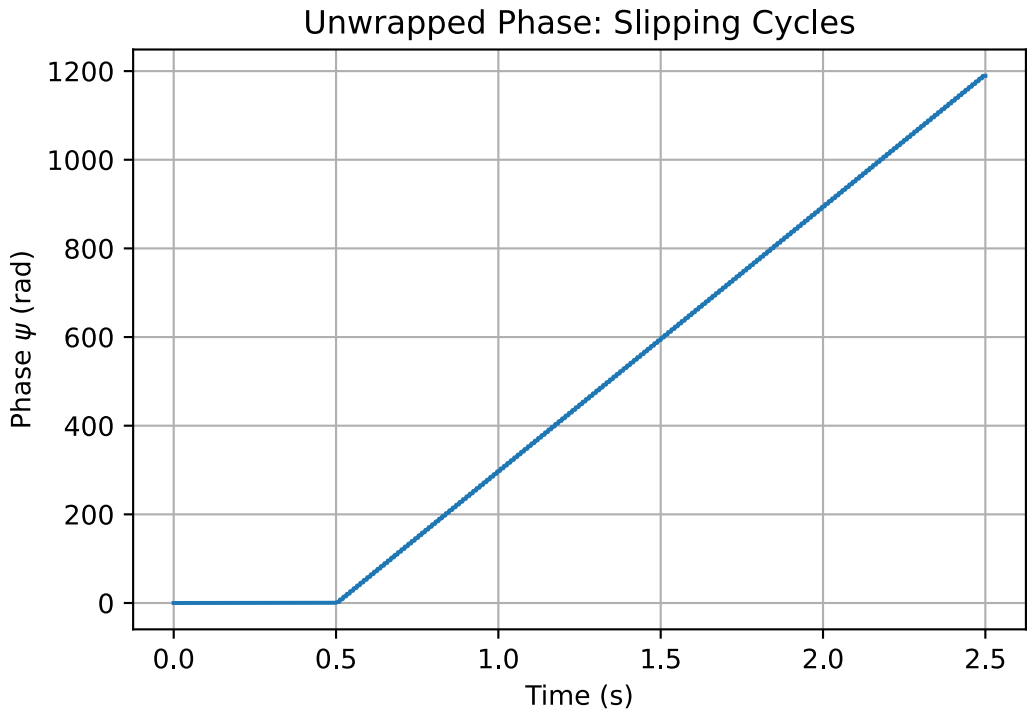
Case Study 1: Frequency Step Exceeds Lock Range

[25]:
```
# Sampling rate is well above the expected loop bandwidth K_t/(2pi) of ~4500 Hz.
fs = 100000
Kt_Hz = 4500
t = arange(0,2.5,1/fs)
# FM modulation m(t) with fD = 1 Hz/volt so peak deviation is max{m(t)}
Df = 4501 # peak deviation in Hz
# Freq Step
m = Df*ss.step(t-.5)
# Sinusoidal FM
fm = 100
#m = Df*cos(2*pi*fm*t)*ss.step(t-.5)
# Squarewave FM
#m = Df*sign(cos(2*pi*fm*t))*ss.step(t-.5)
# Accumulate freq to phase to form input to baseband PLL
phi = 2*pi*cumsum(m)/fs
theta_hat, ev, psi = pll.PLL1(phi,fs,1,1,Kt_Hz,0.707,1)
```

[26]:
```
plot(t,ev)
#plot(t,psi)
xlabel(r'Time (s)')
ylabel(r'VCO Control Voltage ($K_v$ in Hz is one)')
xlim([0.48,0.55])
title(r'VCO Control Voltage as Frequency Deviation in Hz')
grid();
```



[27]:
```
#plot(t,ev)
plot(t,psi)
xlabel(r'Time (s)')
ylabel(r'Phase $\psi$ (rad)')
#xlim([0.48,0.55])
title(r'Unwrapped Phase: Slipping Cycles')
grid();
```

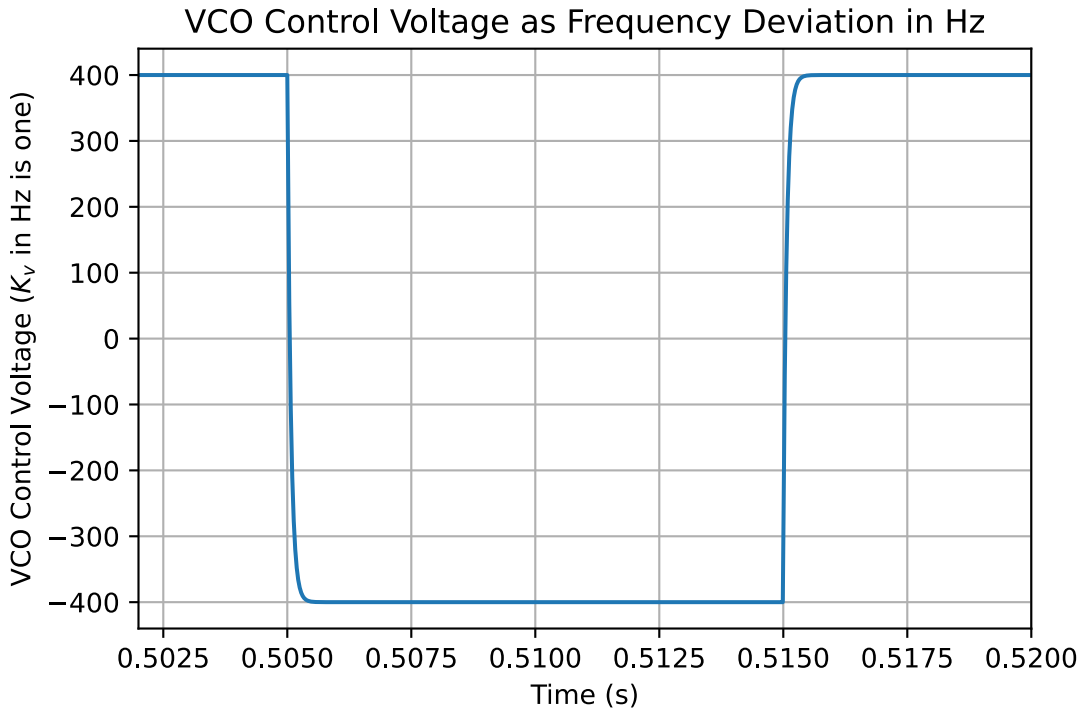


Case Study 2: Square wave FM

[28]:
```
# Sampling rate is well above the expected loop bandwidth K_t/(2pi) of ~4500 Hz.
fs = 100000
Kt_Hz = 2000
t = arange(0,2.5,1/fs)
# FM modulation m(t) with fD = 1 Hz/volt so peak deviation is max{m(t)}
Df = 400 # peak deviation in Hz
# Freq Step
#m = Df*ss.step(t-.5)
# Sinusoidal FM
fm = 50
#m = Df*cos(2*pi*fm*t)*ss.step(t-.5)
# Squarewave FM
m = Df*sign(cos(2*pi*fm*t))*ss.step(t-.5)
# Accumulate freq to phase to form input to baseband PLL
phi = 2*pi*cumsum(m)/fs
theta_hat, ev, psi = pll.PLL1(phi,fs,1,1,Kt_Hz,0.707,1)
```

[29]:
```
plot(t,ev)
#plot(t,psi)
xlabel(r'Time (s)')
ylabel(r'VCO Control Voltage ($K_v$ in Hz is one)')
xlim([0.502,0.52])
```

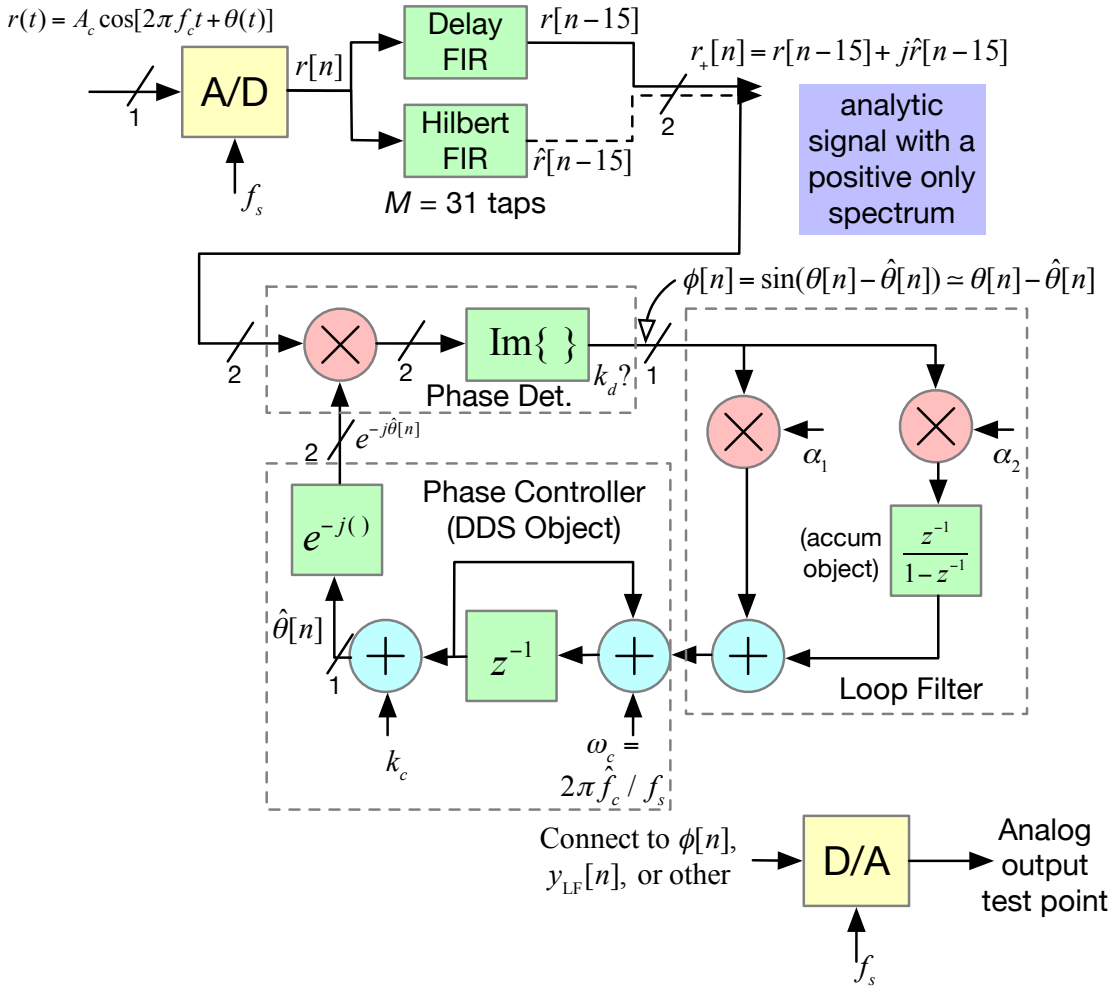
```
title(r'VCO Control Voltage as Frequency Deviation in Hz')
grid();
```



[ ]:

1.7 Digital PLL Using `pyaudio_helper`

Under development, but the code does run using the Sabrent USB audio device. More later. This is not formally part of Lab 5.



DPLL Block diagram as implemented in `pyaudio_helper`

[3]:
```
# import sk_dsp_comm.pyaudio_helper as pah
import pyaudio_helper_old as pah
import ipywidgets as widgets
```

[10]:
```
pah.available_devices()
```

[10]:
```
{0: {'name': 'iMic USB audio system', 'inputs': 2, 'outputs': 2},
 1: {'name': 'MacBook Pro Microphone', 'inputs': 1, 'outputs': 0},
 2: {'name': 'MacBook Pro Speakers', 'inputs': 0, 'outputs': 2}}
```

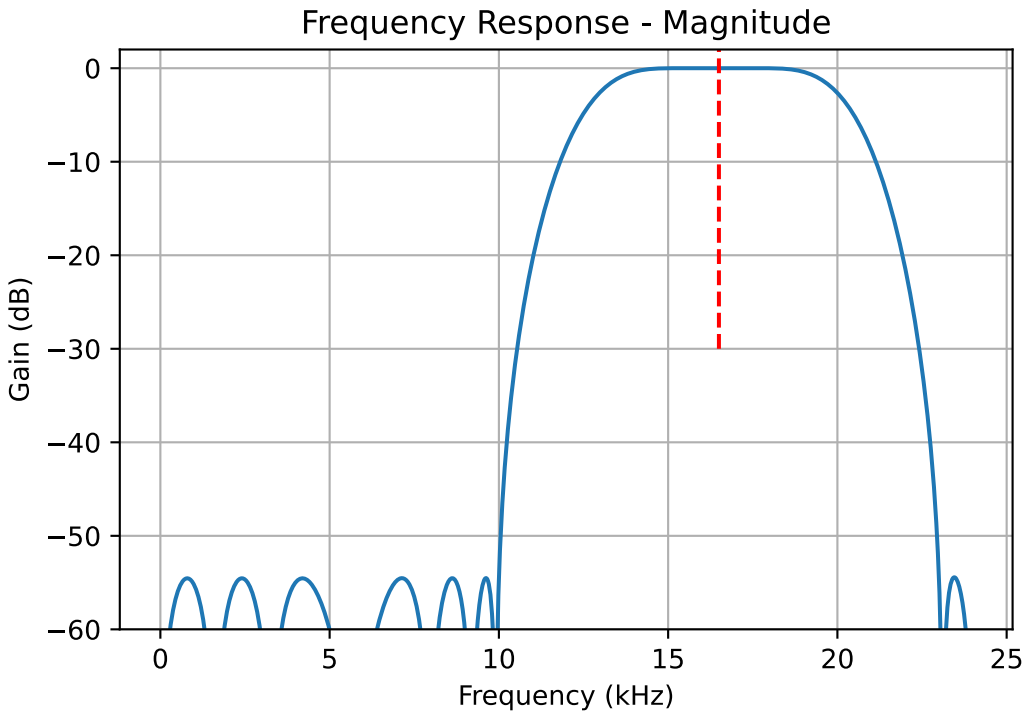
[5]:
```
class DDS(object):
    """
    Implementation of a DDS that outputs exp(-1j*theta_hat)

    Mark Wickert October 2017
    """
```

```
    def __init__ (self,fcenter,fs,kc = 1,state_init = 0):
        """
        Initialize the DDS with a center frequency in Hz, the
        sampling rate, the gain kc, and initial theta_hat state
        in radians.
        """
        self.fcenter = fcenter
        self.fs = fs
        self.delta_w = 2*pi*self.fcenter/self.fs
        self.kc = kc
        self.theta_hat = state_init


    def update(self,e_in):
        """
        Update the DDS phase accumulator
        """
        self.theta_hat += self.delta_w + self.kc*e_in
        if self.theta_hat >= 2*pi:
            self.theta_hat -= 2*pi
        elif self.theta_hat < 0:
            self.theta_hat += 2*pi


    def output_exp(self):
        """
        Output exp(-1j*theta_hat)
        """
        return exp(-1j*self.theta_hat)


    def set_fcenter(self,fcenter_new):
        """
        Set a new center frequency in Hz
        """
        self.fcenter = fcenter_new
        self.delta_w = 2*pi*self.fcenter/self.fs


class accum(object):
    """
    An accumulator class for use in DPLL loop filters

    Mark Wickert October 2017
    """

    def __init__ (self,state=0):
```

```
        """
        Initialize the accumulator object by setting the state
        """
        self.state = state


    def update(self,x_in):
        """
        Update the accumulator
        """
        self.state += x_in
```

[6]:
```
# Filter coefficients for creating an analytic signal
b_hilbert31 = array([0.003509471301,-0.006308706055, 0.002331697153,
                     0.002743280552, 0.005233294088,-0.007862655531,
                    -0.018707992912, 0.043635854730,-0.022090995276,
                    -0.003554769529,-0.040705227792, 0.081732429463,
                     0.036883198159,-0.252049421960, 0.272988493623,
                     0.000000000000,-0.272988493623, 0.252049421960,
                    -0.036883198159,-0.081732429463, 0.040705227792,
                     0.003554769529, 0.022090995276,-0.043635854730,
                     0.018707992912, 0.007862655531,-0.005233294088,
                    -0.002743280552,-0.002331697153, 0.006308706055,
                    -0.003509471301])
b_delay31 = hstack((zeros(15),[1]))
zi_hilbert31 = signal.lfiltic(b_hilbert31,1,[0])
zi_delay31 = signal.lfiltic(b_delay31,1,[0])
```

[7]:
```
fir_d.freqz_resp_list([b_hilbert31],[1],'dB',48)
plot([16.5,16.5],[-30,2],'r--')
xlabel(r'Frequency (kHz)')
ylim([-60,2])
grid();
```



1.7.1 Set Up Globals

- Get variables used globally in place for the remaining lab questions
- Some items such as the phase detector gain will be back filled in with the measured value once it is measured

[8]:
```
# Variables used globally
Nframe = 1024
fs = 48000 # may try 96000 Hz
f0 = 15000
DDS1 = DDS(f0,fs,kc=1.0)
acc1 = accum()
# The below parameters configure the loop parameters
# that define a second-order digital PLL (DPLL)
kd = 20000 #kd_meas #20000.0
kc = 1.0
wn = 2*pi*100
zeta = 0.707
alpha1 = (2*zeta*wn/fs + (wn/fs)**2/2)/kc/kd
alpha2 = (wn/fs)**2/kc/kd
(alpha1,alpha2)
```

[8]:
```
(9.297445183357438e-07, 8.567364931501179e-09)
```

[9]:
```
DDS_f0 = widgets.FloatSlider(description = 'Center Freq',
                 continuous_update = True,
                 value = f0,
                 min = 12000,
                 max = 18000,
                 step = 1,
                 orientation = 'horizontal')
# widgets.HBox([DDS_f0])
```

1.7.2 Define a Callback for the DPLL

- This callback will be used for all of the remaining experiments, but will initially be configured without feedback.

[19]:
```
# define a DPLL callback
def callback_DPLL(in_data, frame_count, time_info, status):
    global Nframe, DDS1, acc1, DDS_f0
    global b_delay31, b_hilbert31, zi_delay31, zi_hilbert31
    DSP_IO.DSP_callback_tic()
    # convert byte data to ndarray
    in_data_nda = np.frombuffer(in_data, dtype=np.int16)
    #***********************************************
    # Convert signed integer data to float
    x = in_data_nda.astype(float32)
    #y = x # pass input to output
    # Begin DSP code here
    #***********************************************
    # Hilbert transform the real signal input into an
    # analytic signal (complex) having only a positive
    # frequency spectrum.
    # Each filter state/(memory), zi_delay31 and zi_hilbert32, must be
    # maintained from frame-to-frame.
    x_re, zi_delay31 = signal.lfilter(b_delay31,1,x,zi=zi_delay31)
    x_im, zi_hilbert31 = signal.lfilter(b_hilbert31,1,x,zi=zi_hilbert31)
    z = x_re + 1j*x_im
    y = zeros_like(x)
    # Sample-by-sample processing to implement
    # DPLL feedback
    # ///////////////////////////////////////////
    for k in range(Nframe):
        # Phase detector
        phi = imag(z[k]*DDS1.output_exp())
#         phi = 5000*imag(DDS1.output_exp()) # used for open-loop testing
        # Form loop filter output and update accum
        y_LF = alpha1*phi + alpha2*acc1.state
        acc1.update(phi)
        # Update DDS/phase controller
```

```
        DDS1.update(y_LF)
        # Collect variable(s) to pass to the output
        # Options include: phi, y_LF, or others you may derive
        y[k] = phi
#         y[k] = 5000*DDS1.output_exp().real
#         y[k] = x_re[k]
#         y[k] = x_im[k]
#         y[k] = y_LF*32000
#         y[k] = x[k]
    # ////////////////////////////////////////////

    # 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
```

[20]:
```
DSP_IO = pah.DSP_io_stream(callback_DPLL,0,0,Nframe,fs,Tcapture=0)
DSP_IO.interactive_stream(0)
# DDS1.set_fcenter(DDS_f0.value)
widgets.HBox([DDS_f0])
```

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

```
HBox(children=(FloatSlider(value=15000.0, description='Center Freq', max=18000.0,␣
 ↪min=12000.0, step=1.0),))
```

[ ]:

1.8 References

1. Rodger Ziemer and William Tranter, Principles of Communications, 8th edition, Wiley, 2014.
2.

[ ]: