ECE 5625 Spring 2019 Project 1
Multicarrier SSB Transceiver

1 Introduction

In this team project you will be implementing the Weaver SSB modulator as part of a multicarrier transmission scheme. Coherent demodulation will be implemented for a single carrier that lies between two adjacent carriers. Test message signals consisting of bandlimited noise will be used to check crosstalk levels. Recorded speech waveforms will be used to provide a final listening test of the complete system. The entire simulation will be constructed in Python, preferably using the Jupyter notebook with Pylab imported (`%pylab inline`). A code framework is provided as part of the project's Jupyter notebook (`'5625_Project_1_2020.ipynb'`). The project can be worked right in this notebook and ultimately used for the project submission. Code submitted with the project report can be easily tested using the speech waveform message signals.

Honor Code: The project teams will be limited to at most two members, except for graduate students who must work as a team of one. Teams are to work independent of one another. Bring questions about the project to me. For ECE 4625 students I encourage you to work in teams. Since each team member receives the same project grade, a group of three should attempt to give each team member equal responsibility. For ECE 5625 students you must work as *solo* teams. The due date for the completed project will be on or before 12:00 pm, Friday, April 3, 2020. **Note:** This is the first Friday after we return from spring break.

2 System Description

A block diagram of the complete multicarrier single sideband system (SSB) is shown in Figure 1. This is more of a conceptual block diagram, as the actual system that will be implemented will use discrete-time signal processing. Specifically the input message signals will be either bandlimited white noise or speech waveforms, both at a sampling rate of 8 ksps (ksamples per second). Using SSB we are able to pack three message signals very close to each other. This provides bandwidth efficiency for sending many message signals over a narrow band of frequencies. The frequency plan shows that three $W = 4$ kHz message signals when SSB modulated can easily fit a channel spanning 16 to 32 kHz. Packing the signals right next to each other may not be the most desirable since adjacent channel interference may occur. Placing a small guard band, say Δf, between each channel may give improved performance.

Each of the SSB modulator blocks will be implemented as a reusable function block in Python. The message signal input to each modulator is a signal sampled at 8 ksps. From basic sampling theory, when we sample at f_s samples per second, the usable lowpass bandwidth runs from 0 – $f_s/2$ Hz. If a continuous-time signal has bandwidth exceeding $f_s/2$ Hz, aliasing will occur, effectively folding spectral energy at frequencies greater than $f_s/2$ back down into the $0 - f_s/2$ band. In particular for $f_s = 8$ ksps the usable signal bandwidth must be less than 4 kHz. Signals with bandwidth exceeding 4 kHz will be *aliased* when sampled at 8 ksps. Telephony grade speech is sampled at 8 ksps. Band limiting of a speech waveform must be done prior to sampling at 8 ksps to avoid aliasing. The passband of telephone grade speech is about 300 – 3300 Hz, thus an

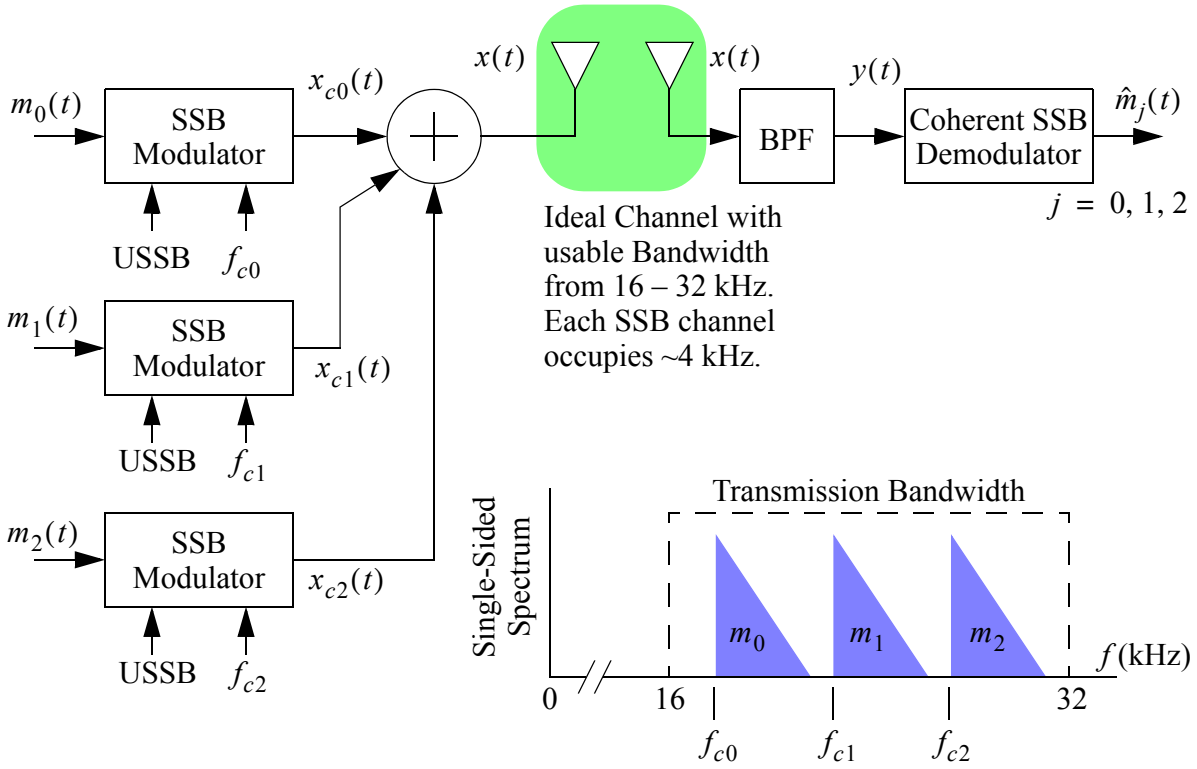

Figure 1: Multicarrier SSB transmission system top level block diagram.

antialiasing filter can be designed to avoid aliasing when sampled at 8 kHz, yet avoid disturbing the 300 – 3300 Hz signal spectrum.

In the SSB modulators of Figure 1 the message signal is assumed to be sampled at $f_{s1} = 8$ ksps. In discrete-time form the message signal is denoted by the sequences $m_j[n] = m_j(n/f_{s1})$, $j = 0, 1, 2$. These discrete-time signals enter a bank of three SSB modulators, each of which processes its input signal with filters and multiplications by reference sinusoids to produce nominally three upper SSB (USSB) signals. The signal carrier frequencies are denoted f_{c0}, f_{c1}, and f_{c2}. In order to accommodate a useful range of carrier frequencies, yet still remain in the discrete-time domain, the sampling rate is increased from 8 ksps up to 96 ksps, i.e., $f_{s1} = 8 \rightarrow f_{s2} = 96$, which corresponds to an upsampling factor of $L = 12$. This factor was chosen to be convenient for use in this project. A real system would likely consider a different value for L.

With the sampling rate increased to 96 ksps, the usable bandwidth, from sampling theory, is now $0 - 96/2 = 48$ kHz. In Figure 1 we further see that for the purposes of this project, the transmission bandwidth or channel is defined to be 16 to 32 kHz. In practice this band of frequencies could be located at any assignable frequency band. Note that the discrete-time signals might first be digital-to-analog converted and then frequency translated using analog mixing.

The received signal $x(t)$, which is composed of three adjacent USSB signals, is ready for demodulation of just one of the message signals $m_j[n]$, $j = 1, 2, 3$. In this project a coherent demodulator is proposed as the means to recover an estimate of the transmitted message signal (sequence). A coherent demodulator needs a coherent reference, but we will assume for this project

that such a reference is available. It might be that one or more pilot signals are sent such that through frequency mixing operations, following a nonlinearity, all three coherent carriers, f_{c1}, f_{c2}, and f_{c3}, can be obtained from the received signal.

The various filters, frequency mixing, and digital signal processing , will result in some crosstalk between the three adjacent channels, as well as undesired band limiting of the desired signal. The Project Tasks section of this project description document, will detail measurements to assess some of these distortions.

2.0.1 Top Level Transmit Function

Below is a top level function, `ssb_transmitter`, that creates the composite transmit signal $x(t)$ as shown in Figure 1. A channel bandpass filter is also included to represent the fact that you are only allowed to transmit signals in the frequency band from 16 kHz to 32 kHz. You will be experimenting with how close you can space the three carrier signals, f_{c0}, f_{c1}, and f_{c2}.

The function `ssb_transmitter()` creates the $x(t)$ waveform at a sampling rate of $12 \times 8000 = 96000$ Hz. **Note**: This function also calls `weaver()` which each team must write.

```
# Top level transmit and channel function
def SSB_transmitter(fc,mtype='speech',mcombo='111',demo=False):
    """
    m_recovered = SSB_transceiver(fc,mtype='noise',mcombo='111')

    =============== Inputs ======================
        fc = Carrier frequencies in Hz of signals to be
             generated as a list; [fc1,fc2,fc3]
     mtype = String value either 'noise' or 'speech';
             the speech input requires that the .wav
             files be present.
    mcombo = combination of channels produced, such as
             '111' <=> all three, '101' <=> the outer two

    =============== Outputs =====================
        m_combined = The combined transmit signal with channel filter
                m0 = message signal 0
                m1 = message signal 1
                m2 = message signal 2

     Mark Wickert, March 2015
    """
    L = 12 # Upsampling factor
    fs = 8000 # sampling rate = 8000 Hz

    #/////////////// BEGIN TRANSMITTER ////////////////////////////
    #Create transmitter message signals from either bandlimited white noise or
    #from available sampled speech test vectors

    if mtype.lower() == 'noise':
        # Design a 9th-order bandpass noise shaping filter with 3 dB
        # cutoffs at 300 & 3300 Hz relative to a 8000 Hz sampling rate.
        bn,an = signal.butter(9,2*array([300, 3300])/8000,btype='bandpass')
        # Create 3 zero mean noise vectors
        m0 = randn(100000)
```

```
        m1 = randn(100000)
        m2 = randn(100000)
        # Filter the noise vectors.
        m0 = signal.lfilter(bn,an,m0)
        m1 = signal.lfilter(bn,an,m1)
        m2 = signal.lfilter(bn,an,m2)
    elif mtype.lower() == 'speech':
        N = 100000; # number of speech samples to process
        fs,m0 = ssd.from_wav('OSR_uk_000_0050_8k.wav')
        m0 = m0[:N]
        fs,m1 = ssd.from_wav('OSR_us_000_0018_8k.wav')
        m1 = m1[:N]
        fs,m2 = ssd.from_wav('OSR_us_000_0030_8k.wav')
        m2 = m2[:N]
    else:
        print('Valid message type must be ''noise'' or ''speech''')
        return 0

    # Input message vectors into SSB modulators
    if demo:
        # external weaver function
        xc0,fs2 = solved.weaver(m0,fs/2,fc[0],fs,L,'upper')
        xc1,fs2 = solved.weaver(m1,fs/2,fc[1],fs,L,'upper')
        xc2,fs2 = solved.weaver(m2,fs/2,fc[2],fs,L,'upper')
    else:
        # notebook local weaver function
        xc0,fs2 = weaver(m0,fs/2,fc[0],fs,L,'upper')
        xc1,fs2 = weaver(m1,fs/2,fc[1],fs,L,'upper')
        xc2,fs2 = weaver(m2,fs/2,fc[2],fs,L,'upper')

    # Form the composite signal based on mcombo selection
    x = zeros_like(xc1)
    if mcombo[0] == '1':
        x += xc0
    if mcombo[1] == '1':
        x += xc1
    if mcombo[2] == '1':
        x += xc2
    if mcombo[0] != '1' and mcombo[1] != '1' and mcombo[2] != '1':
        print('mcombo not of the form "xyz" where x,y,z are 0 or 1')
        return 0
    #////////////////// END Transmitter ////////////////////////////

    #////////////// BEGIN Channel ////////////////////////////////
    # Channel bandlimits signal to 16 - 32 kHz bandwidth
    # The sampling rate is L*fs = 48 kHz
    # It is the responsibility of the receiver to downsample
    # the signal back to the fs rate
    # Create the channel bandpass filter over 16kHz to 32kHz:
    b_chan,a_chan = signal.butter(7,2*array([16000, 32000])/(L*fs),btype='bandpass')
    x = signal.lfilter(b_chan,a_chan,x)
    return x,m0,m1,m2
```

Note the above listing, and all remaining listing found in this document, assume the following

collection of `imports` are run in the fist *code* cell of your Jupyter notebook (the notebook sample has this included)

```
%pylab inline
#%pylab notebook # for plots editable in the notebook
#%matplotlib qt # for popout plots
import sk_dsp_comm.sigsys as ss
import sk_dsp_comm.fir_design_helper as fir_d
import sk_dsp_comm.iir_design_helper as iir_d
import sk_dsp_comm.multirate_helper as mrh
import scipy.signal as signal
from IPython.display import Audio, display
from IPython.display import Image, SVG
```

Since ECE 4650/5650, *Modern Digital Signal Processing*, is not a prerequisite for this course, some details on how to implement the needed SSB modulator (`weaver`) and demodulator (`SSB_demod`) is now provided.

2.1 SSB Modulator Design

A continuous-time version of Weaver's SSB modulator is shown in Figure 2 [1]. For this project

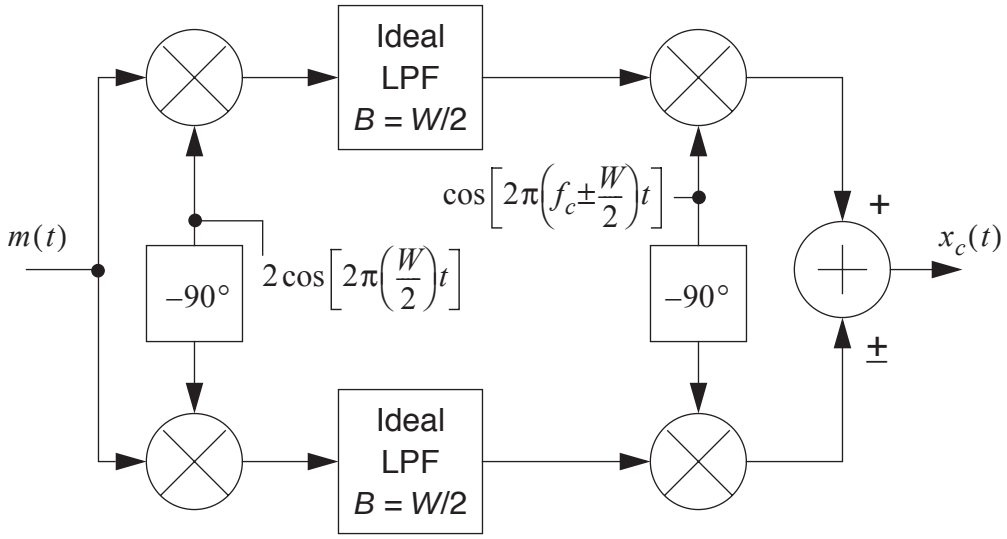

Figure 2: Weaver SSB modulator as an analog signal processing function.

you will be implementing the modulator in the discrete-time domain , i.e., using digital signal processing (DSP) techniques. A hardware implementation of Weaver's modulator, that employs multirate DSP techniques, can be found in [2]. The Palmero implementation serves as the motivation for the discrete-time version of Figure 2 shown in Figure 3. Note that all signals have been replaced by discrete-time signals or sequences. If you are unfamiliar with DSP notation and math, guidance will be provided as the design approach unfolds. First of all notice that the first pair of cos/sin signal generators have the continuous variable t replaced by the *time index* variable over the sampling rate n/f_{s1}. This is a natural substitution, as periodic sampling of the time axis (consider

analog-to-digital conversion) means that we sample the analog signal $m(t)$ at $T_s = 1/f_s$ second intervals, that is $m_j[n] = m_j(nT_s) = m_j(n/f_s)$, etc., with time index n being the discrete-time time axis equivalent.

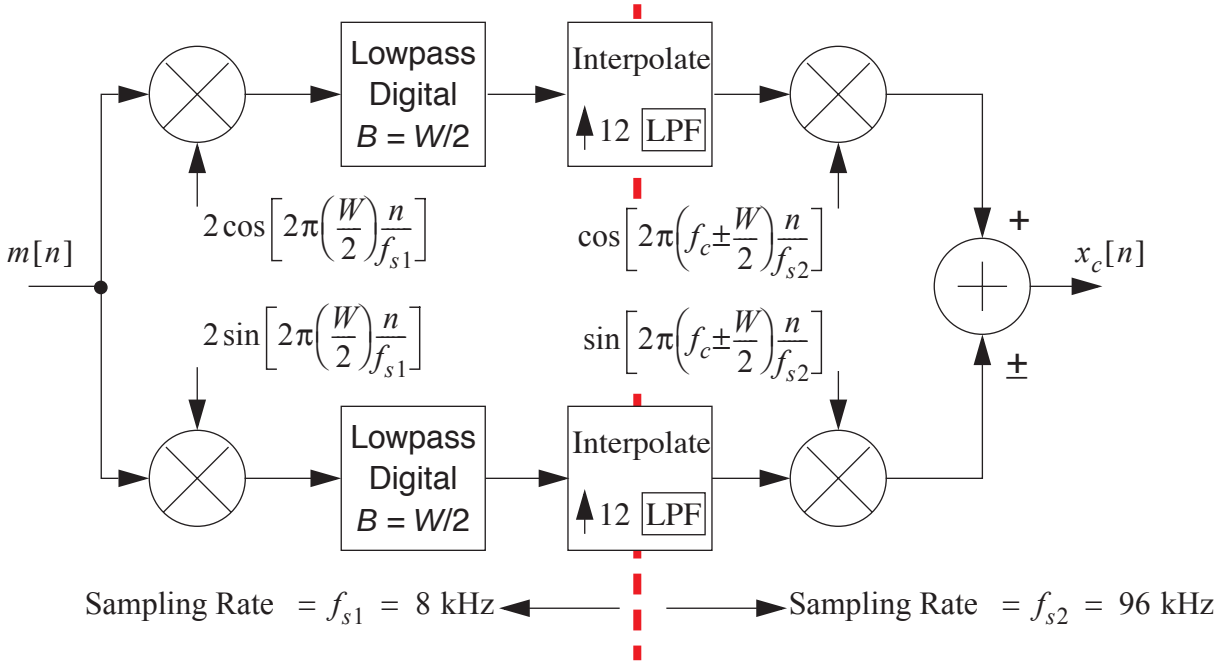

Figure 3: Weaver SSB modulator discrete-time implementation.

The implementation of discrete-time (digital) filters can be kept at a rather abstract level by just thinking of the filters as having a transfer function that is a ratio of polynomials in the z-domain. The z-domain is the discrete-time equivalent of the familiar s-domain. Consider an analog lowpass or bandpass filter has transfer function

$$H(s) = \frac{c_0 + c_1 s + c_2 s^2 + \cdots + c_M s^M}{d_0 + d_1 s + d_2 s^2 + \cdots + d_N s^N}. \tag{1}$$

A study of digital signal processing, in particular digital filter design, reveals that analog filter prototypes, e.g., Butterworth, Chebyshev, Elliptic, etc., can be converted to digital filter equivalents with knowledge of the sampling rate, f_s, and the analog filter coefficients $\{c_k, d_k\}$. The corresponding digital filter in the discrete-time domain will have a transfer function in the z-domain given by

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_M z^{-M}}{a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_N z^{-N}} \tag{2}$$

A z-domain expression results by taking the z-transform of a discrete-time sequence, much like an s-domain expression results by taking the Laplace transform of a continuous-time signal. A course in DSP is needed to fully grasp the details. The course ECE 2610 serves as the starting point for this study, with more knowledge coming from ECE3205. The significance of having z-domain

filter coefficients is that we can implement the filter as an algorithm of the form

$$y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k], \tag{3}$$

assuming we have normalized the denominator coefficients so that $a_0 = 1$. Note that (3) is known as a linear constant coefficient difference equation (LCCDE).

The `scipy.signal` sub package has a collection of functions for designing digital filters (not as extensive as MATLAB). The modules `fir_design_helper.py` and `iir_design_helper.py`, part of `scikit-dsp-comm`, were written to provide a more useful interface to the `scipy.signal` functions. See scikit-dsp-comm FIR/IIR Helper Jupyter notebook examples for details, as well as the project Jupyter notebook. In any case the output of a particular digital filter design is simply a pair of coefficient ndarrays `(b,a)`, or a cascade of second-order sections 2D matrix `sos` for IIR filters and a single ndarray `b` for FIR filters. For the purposes of this project it is sufficient to obtain all of your needed filter designs, both lowpass and bandpass, from the Butterworth family, but with the design helper modules you can now explore other options. To get started make sure your Jupyter notebook or console session has imported `scipy.signal` and the helper modules, e.g.,

```
# In the Project 1 notebook this is already done for you
import scipy.signal as signal
import sk_dsp_comm.fir_design_helper as fir_d
import sk_dsp_comm.iir_design_helper as iir_d
```

2.1.1 Upsampling and Interpolation

The Weaver modulator requires that you implement a sampling rate change from $f_{s1} = 8$ kHz to $f_{s2} = 96$ kHz. Without getting into the DSP math details, this requires the cascade of an *upsampler* with $L = 12$ followed by a lowpass *interpolation* filter as shown in Figure 4. The upsample function inserts $L-1$ zero samples between every input sample, and the lowpass filter interpolates signal sample values to replace the zero sample values inserted by the upsampler. The bandwidth of the lowpass interpolation filter should be the bandwidth of the signal being upsampled, in this case about 4 kHz.

To implement this in Python the *classes*, `mrh.multirate_FIR` and `mrh.multirate_IIR` were written. Method/functions associated with these classes utilize `ss.upsample(x,L)` and `ss.downsample(x,M)` (see Python basics for more information on classes). The class definitions can be found in the helper module `sk_dsp_comm.multirate_helper.py`. As mentioned previously this module needs to be imported into your Jupyter notebook using

```
# In the Project 1 notebook this is near the top
import sk_dsp_comm.multirate_helper as mrh
```

Sample Jupyter notebooks part of the `scikit-dsp-comm` package provide details on using the multirate classes and designing the filters that go into them. See scikit-dsp-comm FIR/IIR Design Helper Jupyter notebook examples and scikit-dsp-comm Multirate Helper Jupyter notebook examples. To create or *instantiate* a multirate object, FIR or IIR, requires first designing an appropriate filter, then using the filter in the creation of the object. A simple example using the multirate IIR class is given in the following code snippet:

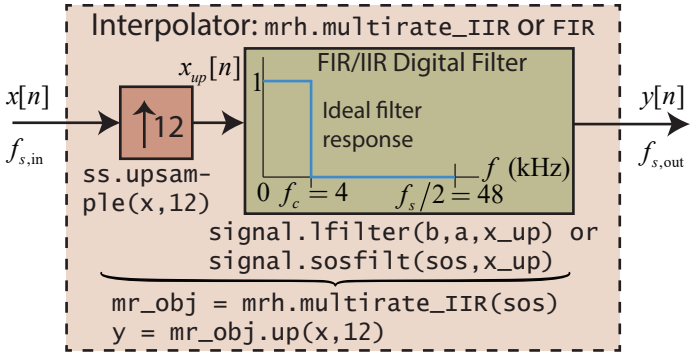

Figure 4: Top level view of interpolation by $L = 12$, underlying primitives, and the use of `sk_dsp_comm.multirate_helper` (`mrh`) objects.

```
# Design the filter core for an interpolator used in changing the sampling rate
# from 80000 Hz to 96000 Hz
b_up, a_up, sos_up = iir_d.IIR_lpf(3300,4300,0.5,60,96000,'cheby1')
# Create the multirate object
mr1 = mrh.multirate_IIR(sos_up)
# Interpolate the signal x to y by the factor L = 12
y = mr1.up(x,12)
```

Once the object is created you can then use one of the three *methods* or functions implemented by the class to filter, interpolate, or decimate. For the case of a `mrh.multirate_IIR` this looks like: (1) `y = mr1.filter(x)` to filter, (2) `y = mr1.up(x,12)` to upsample and interpolate, or (3) `y = mr1.dn(x,12)` to antialias filter and downsample. See scikit-dsp-comm Multirate helper Jupyter notebook examples. **Note**: Decimation is discussed in the next section, as it is required in the demodulator.

2.2 SSB Coherent Demodulator with Decimation

Coherent demodulation of SSB is described in [3]. A DSP implementation follows this exact approach. The transmitter carrier frequency is generated from knowledge of the sampling frequency, so a coherent demodulation reference can be generated in like fashion. A real receiver would not have this knowledge, but as mentioned earlier, might be able to take advantage of a pilot signal from which to derive a demodulation carrier reference.

One question you need to consider is whether or not you need to place a bandpass filter in front of the coherent demodulator? The answer is yes. You need to design a bandpass filter with center frequency dependent upon the carrier frequency you want to receive. See scikit-dsp-comm Multirate helper Jupyter notebook examples for a design example.

Secondly, once the signal is demodulated, the sampling rate is reduced back to $f_{s1} = 8$ kHz. The DSP function used for sampling rate reduction is known as a *decimator*. A decimator is composed of a lowpass filter in cascade with a *downsampler* (`ss.downsample(x,M)`) as shown in Figure 5. Downsampling by the factor M keeps every Mth sample of the input sequence. The $M-1$ samples in between are discarded. Since downsampling reduces the effective sampling rate,

and we know that the usable frequency band is always 0–$f_s/2$, the signal input to a downsampler must be strictly bandlimited to $(f_s/2)/M$ Hz to insure that aliasing does not occur. Note: In the multirate classes when you use the method `dn(x,M)` you need to enter the decimation factor $M = 12$. The input signal to the decimator must be lowpass filtered down to a nominal bandwidth of no more than $W = (96000/2/12) = 4000$ Hz. Recall that telephone grade audio only needs 300–3300 Hz. Note that this is the same bandwidth as required in the coherent demodulator, so can the coherent demodulator and decimator share the same lowpass filter?

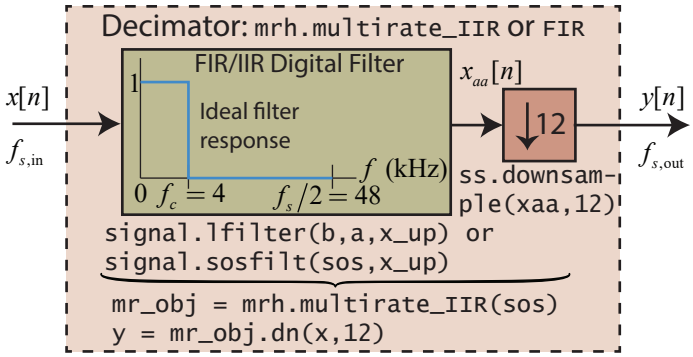

Figure 5: Top level view of decimation by $M = 12$, underlying primitives, and the use of `sk_dsp_comm.multirate_helper` (`mrh`) objects.

2.3 System Performance via Spectral Analysis

As you are building up your system it will be helpful to view signals in the frequency domain. Since random signals are involved in the testing process, both the noise input the speech input, we need to use the power spectral density (PSD) as opposed to just the Fourier transform of the signal. Recall that Chapter 2 of [3] introduced the PSD as a tool for spectral analysis of deterministic and random signals. The Python function `psd()` (brought into the workspace from matplotlib) estimates the power spectrum of a random signal (sequence).

```
psd(x,NFFT,fsamp); # here the ; is needed to suppress a listing
```

To demonstrate this function in action we will execute the full SSB transmitter system using filtered noise as the message signals. The carrier frequencies have been chosen to be 20, 24, and 28 kHz. There are no guard bands between the channels with this configuration. An in-class demonstration will demonstrate this configuration using the sample speech files available in the project zip package.

```
# Run with demo=True means use my solution
# Turn on only the first carrier
In [109]:
fcar = [20000,24000,28000]
In [110]:
x,m1,m2,m3 = SSB_transmitter(fcar,mtype='noise',mcombo='100',demo=True)
In [11]:
figure(figsize=(6,3))
```

```
psd(m1,2**10,8000);
ylim([-90,-30])
title(r'PSD␣of␣Bandlimited␣White␣Noise␣Input␣to␣SSB␣Modulator')
xlabel(r'Frequency␣(Hz)')
```

Figure 6 shows the PSD of bandlimited white noise entering the SSB modulators. The sampling rate at this point is 8 ksps. Note that the bandpass shaping filter has given the spectrum a passband running from 300–3300 Hz as expected.

Now we move on to plotting the composite transmit spectrum, first with `mcombo='100'` which gives only one active carrier, then with `mcombo='111'` to activate all three carriers.

```
In [112]:
figure(figsize=(6,3))
psd(x,2**12,96000);
ylim([-120,-30])
xlim([10000,40000])
title(r'PSD␣of␣$f_{c0}=20$␣kHz,␣$f_{c1}=24$␣kHz,␣$f_{c2}=28$␣kHz')
xlabel(r'Frequency␣(Hz)')
# Turn on all three carriers
In [113]:
x,m1,m2,m3 = SSB_transmitter(fcar,mtype='noise',mcombo='111',demo=True)
```

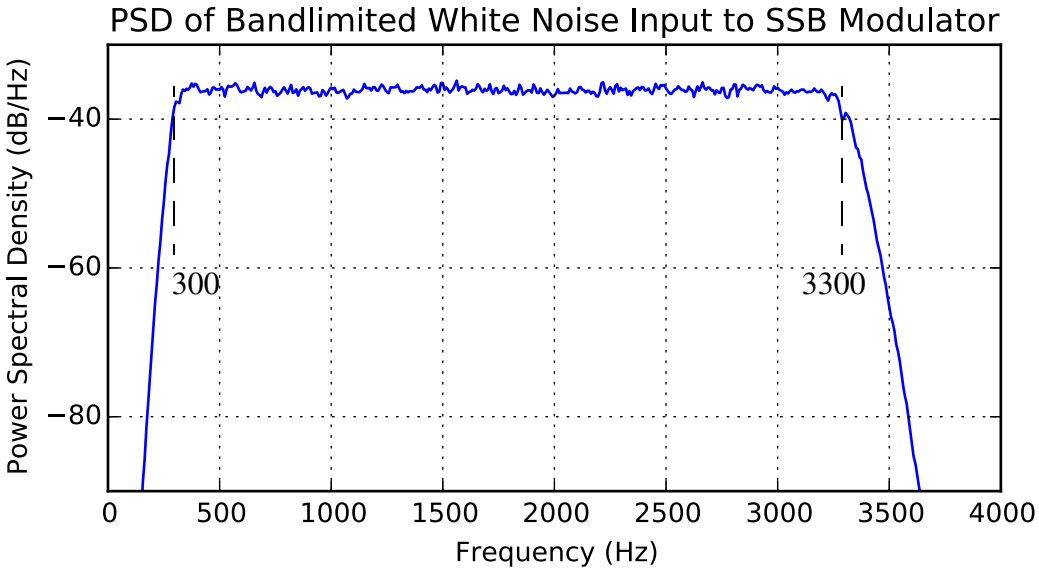

Figure 6: Spectra of filtered white noise representing a message signal spectrum.

In Figure 7 we see the spectrum of the SSB signal at $f_{c1} = 20$ kHz. Notice that the spectrum is far from perfect. The filters in the Weaver modulator, especially the upsampling operation from 8 ksps to 96 ksps, has introduced spectral images which are not completely removed by the default interpolation filter of the function `interp`. The composite spectrum, as sent through the channel is shown in Figure 8.

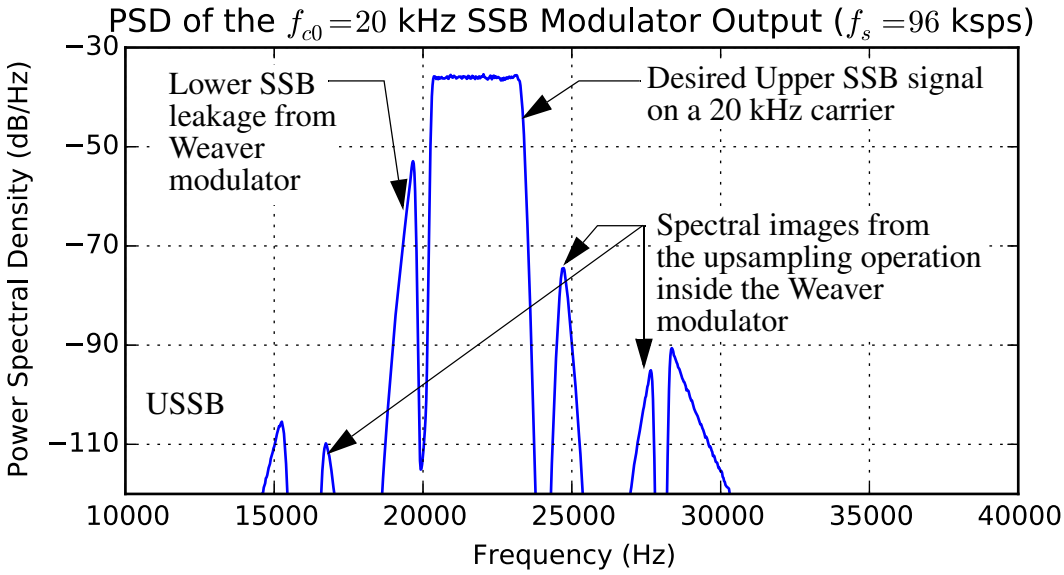

Figure 7: Spectra of $f_{c1} = 20$ kHz SSB noise modulated carrier.

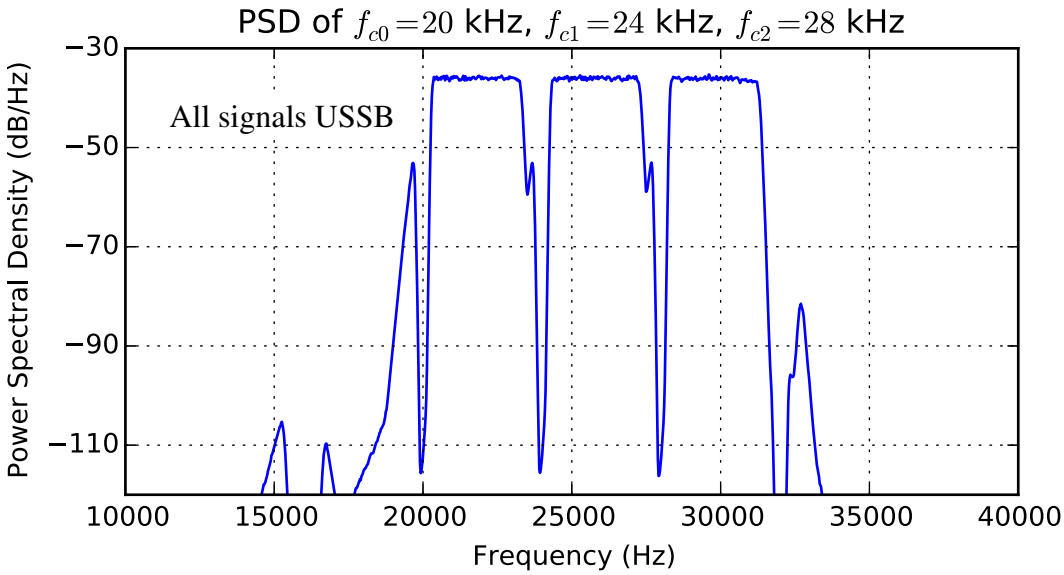

Figure 8: Spectra of the composite SSB noise modulated carrier.

3 Project Tasks

1. Implement and test the SSB modulator function `weaver` using the function interface described below. During normal operation, i.e., as exercised by `SSB_transceiver`, the input sampling rate will be 8 ksps and the output sampling rate will be 96 ksps, i.e., $L = 12$.

```
def weaver(x,W,fc,fs=8000,L=12,mode='upper'):
    """
    y,fs2 = weaver(x,W,fc,fs=8000,L=12,mode='upper')
```

```
    ================== Inputs ======================
        x = Input message signal
        W = Bandwidth of first lowpass in Weaver mod;
            nominally 4000 Hz
       fc = Output carrier frequency in Hz
       fs = Input sampling rate (nominally 8000 Hz)
        L = Upsampling factor (nominally 12)
     mode = Select upper of lower SSB via 'upper' or 'lower'
    ================== outputs =====================
       y = modulator output at sample rate L*fs
     fs2 = output sample rate in Hz

    """
    #Write your code starting here to implement the
    #Weaver SSB modulator

    #Use the rate_change object inside this function to perform
    #the required upsampling by L

    return y, fs2
```

(a) Plot the PSD of the modulator output for a band limited noise input, with $f_c = 20$ kHz. The plot window should be scaled as in Figure 7.

(b) Comment on the performance of your modulator function.

(c) Provide a fully commented source code listing of this function in your report.

2. Implement a coherent SSB demodulator using the function interface described below.

```
def SSB_demod(x,fc,fs_in,fs_out):
    """
    y_demod = SSB_demod(x,fc,fs_in,fs_out)

    =============== Inputs ======================
          x = Received signal input at sampling rate fs_in
         fc = Carrier frequency of signal to be demodulated
      fs_in = Sampling rate in samples per second of input signal
     fs_out = Sampling rate of output demodulated signal; it is assumed that
              fs_in/fs_out is an integer

    =============== Outputs =====================
     y_demod = Recovered modulation from the carrier based signal at fc;
               The sampling rate is fs_out (nominally 8 ksps)

    """
    #Write your code starting here to implement the
    #Coherent SSB demodulator

    #Use the rate_change object inside this function to perform
    #the required downsampling and lowpass filtering by M = L

    return y_demod
```

(a) Test your modulator using just a single SSB carrier at 24 kHz (turn off the other carriers in `SSB_transceiver` by letting `mcombo='010'`), plot the PSD of the recovered message signal. Assume again that band limited noise is used for the input message. The plot window should scaled similarly to Figure 6.

(b) Comment on the performance of your coherent demodulator.

(c) Provide a fully commented source code listing of this function in your report.

3. In this task all three carriers will be turned on. Choose any spacing you wish for the carriers so long as they fit within the channel bandwidth (the next task will allow you to experiment with a guard band between the carriers). Band limited noise messages sources will again be employed for testing.

   (a) Obtain a composite signal spectrum plot similar to Figure 8 for your specific modulator bank implementation. Make your axis scaling match that of Figure 8.

   (b) Moving to the demodulator output measure the signal-to-interference ratio (SIR) in dB via superposition of demodulated output signals. Set the demodulator to recover the center carrier (this should be carrier number 1). With just the center carrier being transmitted, i.e. set `m`, find the power in the demodulator output as `Psig = var(y_demod)`. Next find the interference power by setting the transmiiter output to be `x = xc0 + xc2` (`mcombo='101'`) and find the interference power to be `Pinterfere = var(y_demod)`. Now form the ratio

$$\mathrm{SIR}_{\mathrm{dB}} = 10\log_{10}\left(\frac{P_{\mathrm{sig}}}{P_{\mathrm{interfere}}}\right) \tag{4}$$

   (c) Measure the SIR on one of the outside signals, e.g., f_{c1} or f_{c3} to see if there is less interference present than when *surrounded* by two signals.

   (d) Comment on your SIR measurement results.

4. Repeat Task 3, except now you are free to move the carrier frequencies to allow guard bands. Note that the channel bandpass filter must be left intact, that is you may not change it. Comment on any observed performance improvements obtained by including guard bands between the carriers.

5. Returning once again to carrier frequencies at 20, 24, and 28 kHz, switch the modulator inputs to the sample speech files provided in the zip package. With all three signals present at the transmitter output (`mcombo='111'`), demodulate message signal $m_1[n]$. Listen to this signal using the Audio control available on the Jupyter notebook via

```
ssd.to_wav('ydd.wav',8000,y_demod)
Audio('ydd.wav')
```

   (a) Listen carefully for any interference heard in the background of the desired message signal. You may want to listen to all three speech files straight from the ZIP package so that you know what they are supposed to sound like. Comment on what you hear.

   (b) Calculate the SIR for demodulation of the center signal using the superposition technique of Task 3b.

6. Comment on your overall experience with this team project.

7. In addition to a project report, please package you completed simulation software into a ZIP package and Email this package to me. I will be verifying your projects by listening to the demodulated output of the speech files. Best yet, send me your Jupyter notebook with embedded solution code for `weaver` and `ssb_demod`.

References

[1] Donald K. Weaver, "A Third Method of Generation and Detection of Single-Sideband Signals," *IRE Proceedings*, 1956, pp. 1703-1705.

[2] Nico Palermo, "A 9 MHz Digital SSB Modulator," available on the internet at www.microtelecom.it/ssbdex/ssbdex-e.htm.

[3] R. Ziemer and W. Tranter, *Principles of Communications*, seventh edition, John Wiley, 2015.