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 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. 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. The Python functions are documented in Appendix A and the source code itself is on the course web site as a ZIP file package.

Honor Code: The project teams will be limited to at most three 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, March 24, 2017. Note: This is the Friday before 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 ksp/s (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 $\Delta f$, 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 ksp/s. 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 $f_s = 8$ ksp/s the usable signal bandwidth must be less than 4 kHz. Signals with bandwidth exceeding 4 kHz will be aliased when sampled at 8 ksp/s. Telephony grade speech is sampled at 8 ksp/s. Band limiting of a speech waveform must be done prior to sampling at 8...
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 $m_j[n] = m_j(n/f_{s1})$, $j = 1, 2, 3$. 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 $f_{c1}$, $f_{c2}$, and $f_{c3}$. 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.
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, \texttt{ssb\textunderscore 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 \texttt{ssb\textunderscore transmitter()} creates the \( x(t) \) waveform at a sampling rate of \( 12 \times 8000 = 96000 \) Hz. \textbf{Note:} This function also calls \texttt{weaver()} which each team must write.

```python
# 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')
    """
    m_combined = SSB_transceiver(fc, mtype='noise', mcombo='111')

    #////////////// BEGIN TRANSMITTER ////////////////////////////
    #Create transmitter message signals from either bandlimited white noise or
    #from available sampled speech test vectors
    if mtype.lower() == 'speech':
        # 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 * np.array([300, 3300]) / 8000, btype='bandpass')
```

Mark Wickert, March 2015

L = 12 # Upsampling factor
fs = 8000 # sampling rate = 8000 Hz

```
# 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

# 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
Since *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 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 $n = f_{s_1}$. 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.

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

$$H(s) = \frac{c_0 + c_1s + c_2s^2 + \cdots + c_Ms^M}{d_0 + d_1s + d_2s^2 + \cdots + d_Ns^N}.$$  

![Figure 2: Weaver SSB modulator as an analog signal processing function.](image-url)
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

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

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 obtain the details. The course ECE 2610 course 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

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

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 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 were written Fall 2016 to provide a more useful interface to the scipy.signal functions. See Appendix A 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 \((b, a)\). 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.,

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

### 2.1.1 Upsampling and Interpolation

The Weaver modulator requires that we implement a sampling rate change from \( f_{s1} = 8 \text{ kHz} \) to \( f_{s2} = 96 \text{ kHz} \). Without getting into the DSP math details, this requires the cascade of an upsample with \( L = 12 \) followed by a lowpass interpolation filter. 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 upsample. 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 `multirate_FIR` and `multirate_IIR` were written. Method/functions associated with these classes utilize `ssd.upsample(x,L)` and `ssd.downsample(x,M)` (see Python basics for more information on classes). The class definitions can be found in the support module `comm1_support.py`. As mentioned previously this module needs to be imported into your Jupyter notebook using

```python
# In the Project 1 notebook this is near the top
import comm1_support as cs1
```

Appendix A and B of this document is dedicated using the multirate classes and designing the filters that go into them. The legacy `rate_change` class is still available in the module `comm1_support.py`, but should not be used for this project. 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

```python
# 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 = cs1.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 `multirate_IIR` this looks like: (1) \( y = \text{mr1.filter}(x) \) to filter, (2) \( y = \text{mr1.up}(x,12) \) to upsample and interpolate, or (3) \( y = \text{mr1.dn}(x,12) \) to antialias filter and downsample. See Appendix B for more details and 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 Appendix B 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 (ssd.downsample(x,M)). Downsampling by the factor $M$ keeps every $M$th 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?

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).

```python
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.

```python
# 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)
```
2.3 System Performance via Spectral Analysis

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 4 shows the PSD of bandlimited white noise entering the SSB modulators. The sampling rate at this point is 8 ksp/s. 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)

![PSD of Bandlimited White Noise Input to SSB Modulator](image)

Figure 4: Spectra of filtered white noise representing a message signal spectrum.

In Figure 5 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 ksp/s to 96 ksp/s, 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 6.


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., \( L = 12 \). The

\[
\text{def weaver}(x,W,fc,fs=8000,L=12,mode='upper'):
\]

"""
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 \( fc = 20 \) kHz. The plot window should be scaled as in Figure 5.

(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 ksp)

    #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 4.

(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 6 for your specific modulator bank implementation. Make your axis scaling match that of Figure 6.

(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 \( P_{sig} = \text{var}(y_{demod}) \). Next find the interference power by setting the transmitter output to be \( x = xc0 + xc2 \) (mcombo='101') and find the interference power to be \( P_{interfere} = \text{var}(y_{demod}) \). Now form the ratio

\[
\text{SIR}_{DB} = 10 \log_{10} \left( \frac{P_{sig}}{P_{interfere}} \right)
\]

(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

```python
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


A Filter Design using the Helper Modules

At the top of this notebook two filter design modules are imported so that you can experiment with filter design in terms of amplitude response specifications. Take a look and verify that the lines are present

```python
import fir_design_helper as fir_d
import iir_design_helper as iir_d
```

are present. The functions in these modules provide an easier and more consistent interface for both FIR (linear phase) and IIR classical designs. Functions inside these modules *wrap* `scipy.signal` functions and also incorporate new functions.

Two approaches two filter design this project are:

1. Enter into `scipy.signal` functions the desired filter cutoff frequency (lowpass) or pair of cutoff frequencies (bandpass), choose the passband ripple, and finally choose the filter order. The filter coefficients, ‘b’ (numerator) and ‘a’ (denominator) arrays, are then returned.

2. Enter into `fir_design_helper` or `iir_design_helper` functions the full amplitude response requirements of filter passband critical frequencies, stopband critical frequencies, passband ripple, and stopband attenuation. The number of taps/coefficients (FIR case) or the filter order (IIR case) needed to meet these requirements is then determined and the filter coefficients are returned as `b` for FIR, `b`, `a`, and an `sos` 2D array with the rows containing second-order sections for IIR.
A FILTER DESIGN USING THE HELPER MODULES

The functions contained in `fir_design_helper.py` support linear phase FIR filter design using windowing and equal-ripple response via the Remez algorithm. The design from amplitude response masks are given in Figure A.1.

The interface to the design functions are contained in Table A.1.

The functions in `iir_design_helper.py` support Butterworth (‘butter’), Chebyshev type 1 (‘cheby1’) and type 2 (‘cheby2’), elliptical (‘ellip’) responses. The design from amplitude response masks are given in Figure A.2.

The interface to the design functions are contained in Table A.2.

To be clear, in this project you only need to implement lowpass and bandpass designs. Example of lowpass and bandpass designs from amplitude response specifications are found in the next two example subsections.

A.1 Amplitude Response Lowpass Design

In this subsection we consider a lowpass designs suitable for the interpolator and the decimator, both of which require a a lowpass cutoff frequency at 4 kHz with sampling rate at 96 kHz. A lowpass filter with cutoff at \(W/2 = 4000/2 = 2000\) Hz is also needed in the Weaver modulator.

A.1.1 Equal-Ripple FIR Lowpass at 4 kHz \((W)\) with \(f_s = 96\) kHz

The amplitude response requirement is chosen to have the passband run from 0 to 3300 Hz and the stopband runs from 4300 Hz to \(f_s/2 = 48,000\) Hz. The stopband ripple is 0.5 dB and stopband attenuation is 60 dB. The filter will have linear phase, which is good, but being an FIR filter we expect a large number of taps as the transition band normalized by \(f_s\) is very small. i.e., 0.0105 (1.05%). Looking at the figures extracted from the Jupyter notebook, we see that 196 taps are required!
Fig. A.1: General amplitude response requirements for the lowpass, highpass, bandpass, and bandstop filter types.
A FILTER DESIGN USING THE HELPER MODULES

Introduction

There are 10 filter design functions and one plotting function available in fir_design_helper.py. Four functions for designing Kaiser window based FIR filters and four functions for designing equiripple based FIR filters. Of the eight just described, they all take in amplitude response requirements and return a coefficients array. Two filter functions are simply wrappers around the scipy.signal function signal.firwin() for designing filters of a specific order with one (lowpass) or two (bandpass) critical frequencies are given. The wrapper functions fix the window type to the firwin default of hann (hanning). The plotting function provides an easy means to compare the resulting frequency response of one or more designs on a single plot. Display modes allow gain in dB, phase in radians, group delay in samples, and group delay in seconds for a given sampling rate. This function, freq_resp_list(), works for both FIR and IIR designs. Table 1 provides the interface details to the eight design functions where d_stop and d_pass are positive dB values and the critical frequencies have the same unit as the sampling frequency f_s. These functions do not create perfect results so some tuning of the design parameters may be needed, in addition to bumping the filter order up or down via N_bump.

Table A.1: FIR filter design functions in fir_design_helper.py.

<table>
<thead>
<tr>
<th>Type</th>
<th>FIR Filter Design Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kaiser Window</strong></td>
<td></td>
</tr>
<tr>
<td>Lowpass</td>
<td>h_FIR = firwin_kaiser_lpf(f_pass, f_stop, d_stop, fs = 1.0, N_bump=0)</td>
</tr>
<tr>
<td>Highpass</td>
<td>h_FIR = firwin_kaiser_hpf(f_stop, f_pass, d_stop, fs = 1.0, N_bump=0)</td>
</tr>
<tr>
<td>Bandpass</td>
<td>h_FIR = firwin_kaiser_bpf(f_stop1, f_pass1, f_pass2, f_stop2, d_stop, fs = 1.0, N_bump=0)</td>
</tr>
<tr>
<td>Bandstop</td>
<td>h_FIR = firwin_kaiser_bsf(f_stop1, f_pass1, f_pass2, f_stop2, d_stop, fs = 1.0, N_bump=0)</td>
</tr>
<tr>
<td><strong>Equiripple Approximation</strong></td>
<td></td>
</tr>
<tr>
<td>Lowpass</td>
<td>h_FIR = fir_remez_lpf(f_pass, f_stop, d_pass, d_stop, fs = 1.0, N_bump=5)</td>
</tr>
<tr>
<td>Highpass</td>
<td>h_FIR = fir_remez_hpf(f_stop, f_pass, d_pass, d_stop, fs = 1.0, N_bump=5)</td>
</tr>
<tr>
<td>Bandpass</td>
<td>h_FIR = fir_remez_bpf(f_stop1, f_pass1, f_pass2, f_stop2, d_pass, d_stop, fs = 1.0, N_bump=5)</td>
</tr>
<tr>
<td>Bandstop</td>
<td>h_FIR = fir_remez_bsf(f_pass1, f_stop1, f_stop2, f_pass2, d_pass, d_stop, fs = 1.0, N_bump=5)</td>
</tr>
</tbody>
</table>

The optional N_bump argument allows the filter order to be bumped up or down by an integer value in order to fine tune the design. Making changes to the stopband gain main also be helpful in fine tuning. Note also that the Kaiser bandstop filter order is constrained to be even (an odd number of taps).
A.1 Amplitude Response Lowpass Design

Fig. A.2: General amplitude response requirements for the lowpass, highpass, bandpass, and bandstop IIR filter types.
A FILTER DESIGN USING THE HELPER MODULES

Bilinear Filter Design in Python

Within `iir_design_helper.py`, there are four filter design functions and a collection of supporting functions available. The four functions are used for designing lowpass, highpass, bandpass, and bandstop filters according to Butterworth, Chebyshev type 1, Chebyshev type 2, elliptical, and Bessel analog filter prototypes. Table 1 details these functions and their outputs.

### Table A.2: IIR filter design functions in `iir_design_helper.py` and support functions.

<table>
<thead>
<tr>
<th>Type</th>
<th>IIR Filter Design Functions*</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Transfer Function (b,a) and SOS</strong></td>
<td></td>
</tr>
<tr>
<td>Lowpass (bilinear)</td>
<td><code>b, a, sos = IIR_lpf(f_pass, f_stop, Ripple_pass, Atten_stop, fs = 1.00, ftype = 'butter')</code></td>
</tr>
<tr>
<td></td>
<td>ftype may be 'butter', 'cheby1', 'cheby2', 'elliptic' or 'bessel'</td>
</tr>
<tr>
<td>Highpass (bilinear)</td>
<td><code>b, a, sos = IIR_hpf(f_stop, f_pass, Ripple_pass, Atten_stop, fs = 1.00, ftype = 'butter')</code></td>
</tr>
<tr>
<td></td>
<td>ftype may be 'butter', 'cheby1', 'cheby2', 'elliptic' or 'bessel'</td>
</tr>
<tr>
<td>Bandpass (bilinear)</td>
<td><code>b, a, sos = IIR_bpf(f_stop1, f_pass1, f_pass2, f_stop2, Ripple_pass, Atten_stop, fs = 1.00, ftype = 'butter')</code></td>
</tr>
<tr>
<td></td>
<td>ftype may be 'butter', 'cheby1', 'cheby2', 'elliptic' or 'bessel'</td>
</tr>
<tr>
<td>Bandstop (bilinear)</td>
<td><code>b, a, sos = IIR_bsf(f_pass1, f_stop1, f_stop2, f_pass2, Ripple_pass, Atten_stop, fs = 1.00, ftype = 'butter')</code></td>
</tr>
<tr>
<td></td>
<td>ftype may be 'butter', 'cheby1', 'cheby2', 'elliptic' or 'bessel'</td>
</tr>
<tr>
<td><strong>Support Functions</strong></td>
<td></td>
</tr>
<tr>
<td>SOS list plot</td>
<td><code>freqz_resp_cas_list(sos,mode = 'dB',fs=1.0,Npts = 1024,fsize=(6,4))</code></td>
</tr>
<tr>
<td>SOS freqz</td>
<td><code>w, Hcas = freqz_cas(sos,w)</code></td>
</tr>
<tr>
<td>SOS plot pole-zero</td>
<td><code>sos_zplane(sos,auto_scale=True,size=2,tol = 0.001)</code></td>
</tr>
<tr>
<td></td>
<td>More accurate root factoring results in more accurate pole-zero plot.</td>
</tr>
<tr>
<td>Cascade SOS</td>
<td><code>sos = sos_cascade(sos1,sos2)</code></td>
</tr>
</tbody>
</table>

*These functions wrap `scipy.signal.iirdesign()` to provide an interface more consistent with the FIR design functions found in the module `fir_design_helper.py`. The function `unique_cpx_roots()` is used to mark repeated poles and zeros in `sos_zplane`. Note: All critical frequencies given in increasing order.
A.1 Amplitude Response Lowpass Design

Design of a lowpass suitable $L = 12$ interpolation and decimation
A FILTER DESIGN USING THE HELPER MODULES

# Verify the passband and stopband gains are as expected
mr_up.freq_resp('db',96000)

Frequency Response - Magnitude

Amplitude response requirements met, but filter order high

# Verify that the FIR design has constant group delay (N_taps - 1)/2 samples
mr_up.freq_resp('groupdelay_s',96000,[0,100])

Frequency Response - Group Delay

For a linear phase filter we expect the group delay to be a constant
A.1.2 Elliptic IIR Lowpass at 2 kHz \((W/2)\) with \(f_s = 8\) kHz

This \(W/2\) filter design is relatively easy when using an elliptical

A.2 Amplitude Response Bandpass Design

Here we consider FIR and IIR bandpass designs for use in the SSB demodulator to remove an adjacent channel signal. In the design of the SSB receiver this filter needs to be tunable, as it must follow the carrier frequency of the desired channel.

A.2.1 Equal-Ripple FIR with Bandwidth 4 KHz \(W\)

Set out to design a 4 kHz wide Bandpass filter using an equal-ripple FIR. In an attempt to ease the filter order, the stopbands, above and below the passband, are each set 1 kHz away.
A FILTER DESIGN USING THE HELPER MODULES

Equal-ripple FIR BPF centered on 26 kHz, a very high order design

- This filter design exceeds the threshold of 200 taps maximum for FIR designs
- In general to filter a signal using this design the code is:
  \[ y = \text{signal} \cdot \text{lfilter}(b\_rec\_bpf1,[1],x) \]

A.2.2 Equal-Ripple Elliptic with Bandwidth 4 KHz \((W)\)

Set out again to design a 4 kHz wide Bandpass filter, this time using an elliptic IIR. The same amplitude requirements used in the FIR BPF design are used here.

• The order 14 design is reasonable
• More stopband attenuation may be desireable, or it may be better to go with a chebyshev type 1 as the stopbands asymptotically approach infinite attenuation
• In any case going forward to use this filter it is best to use the sos (second-order sections) form as follows:
  \[ y = \text{signal} \cdot \text{sosfilt}(sos\_rec\_bpf2,x) \]
Elliptical IIR BPF centered on 26 kHz and of reasonable order

**B Using the Classes multirate\_FIR and multirate\_IIR**

In this appendix the classes multirate\_FIR and multirate\_IIR found in the module are discussed, with the aim being how they can be used to filter, interpolate (upsample and filter), and decimate (filter and downsample) discrete time signals. If you choose to design the filters needed in this project using full amplitude response requirements, then these classes replace the legacy class rate\_change described earlier in this project description document.

In Appendix A there are examples of lowpass and bandpass filters. Once a design is complete you will need to use it in one of three operational modes: (1) lowpass or bandpass filtering, (2) as a prefilter to a downsampler, (3) As a postfilter to an upsampler. For FIR filter designs you create an object that holds the $b$ array filter coefficients and supplies a number of supporting methods (like functions). Of particular interest are methods that provide:

- Filtering: `filter(x)`
- Interpolation, i.e., upsampling followed by filtering with the $b$ coefficients: `up(x, L\_change = 12)`
- Decimation, i.e., filtering with the $b$ coefficients followed by downsampling: `dn(x, M\_change = 12)`

For IIR filter designs you create an object that holds the $sos$ 2D array of filter coefficients and supplies a number of supporting methods. Of particular interest are methods that provide:

- Filtering: `filter(x)`
- Interpolation, i.e., upsampling followed by filtering with the $b$ coefficients: `up(x, L\_change = 12)`
B USING THE CLASSES MULTIRATE_FIR AND MULTIRATE_IIR

- Decimation, i.e., filtering with the b coefficients filter followed by downsampling: \( d_{n}(x, M_{\text{change}} = 12) \)

Both classes also provide a means to obtain frequency response plots and pole-zero plots directly from the instantiated multirate objects:

- Frequency response: \(
\text{freq\_resp}(\text{mode} = \text{dB}, f_{s} = 8000, y\text{lim} = [-100, 2])
\). Mode can be 'dB' magnitude, 'phase' in radians, or 'groupdelay_s' in samples and 'groupdelay_t' in sec, all versus frequency in Hz

- Pole-zero Plot: \(
\text{zplane}(\text{auto\_scale=True, size=2, detect\_mult=True, tol=0.001})
\)

A summary of the filtering capabilities of the two multirate classes is shown below in Figure B.1.

\[
\begin{align*}
\text{Given: } & mr = \text{multirate\_FIR}(b) \text{ or } mr = \text{multirate\_IIR}(sos) \\
\text{Filter: } & x[n] \rightarrow \text{Filter b or sos} \rightarrow y[n] \quad y = mr.\text{filter}(x) \\
\text{Interpolate: } & x[n] \rightarrow L \rightarrow \text{Filter b or sos} \rightarrow y[n] \quad y = mr.\text{up}(x, L) \\
\text{Decimate: } & x[n] \rightarrow \text{Filter b or sos} \rightarrow M \rightarrow y[n] \quad y = mr.\text{dn}(x, M)
\end{align*}
\]

Filtering capabilities of the multirate\_FIR and multirate\_IIR classes

B.1 Interpolator Design Example

Here we take the earlier lowpass filter designed to interpolate a signal being upsampled from \( f_{s1} = 8000 \) kHz to \( f_{s2} = 96 \) kHz. The upsampling factor is \( L = f_{s2}/f_{s1} = 12 \). The ideal interpolation filter should cutoff at \( f_{s1}/2 = f_{s2}/(2 \cdot 12) = 8000/2 = 4000 \) Hz.

The upsampler (\( y = \text{ssd.upsampler}(x, L) \)) inserts \( L - 1 \) samples between each input sample. In the frequency domain the zero insertion replicates the input spectrum on \([0, f_{s1}/2] \) \( L \) times over the interval \([0, f_{s2}] \) (equivalently \( L/2 \) times on the interval \([0, f_{s2}/2] \)). The lowpass interpolation filter serves to removes the images above \( f_{s2}/(2L) \) in the frequency domain and in so doing filling in the zeros samples with waveform interpolants in the time domain.

- Start by designing a Remez equal ripple FIR with the intent of it serving as an interpolation filter for a rate change by a factor of 12

- Of necessity, the filter cutoff frequency needs to be at the output sampling rate, \( f_{s2}/(2 \cdot 12) \)
The stopband attenuation is set to just 50 dB, yet with a narrow transition band relative to the sampling rate, i.e., \((4300 - 3300)/96000 = 1/96 = 0.0104\) (about 1% of the sampling rate), the filter design is not trivial.

```matlab
# Design the filter core for an interpolator used in changing the sampling rate from 8000 Hz
# to 96000 Hz
b_up = fir_d.fir_remez_lpf(3300,4300,0.5,60,96000)
# Create the multirate object
mr_up = cs1.multirate_FIR(b_up)

# Remez filter taps = 196.
# FIR filter taps = 196

# Sinusoidal test signal
n = arange(10000)
x = cos(2*pi*1000/8000*n)
# Interpolate by 12 (upsample by 12 followed by lowpass filter)
y = mr_up.up(x,12)

# Plot the results
subplot(211)
psd(x,2**12,8000);
title('1 kHz Sinusoid Input to L=12$ Interpolator')
ylabel('PSD (dB)')
ylim([-100,0])
subplot(212)
psd(y,2**12,12*8000)
title('1 kHz Sinusoid Output from L=12$ Interpolator')
ylabel('PSD (dB)')
ylim([-100,0])
tight_layout()
```

The filter order, and hence the number of taps, is very large at 196.

To test the design input a sinusoid at 1 kHz and observe the interpolator output spectrum compared with the input spectrum.

$L = 12$ interpolator input and output spectrum
B. Using the Classes Multirate FIR and Multirate IIR

- In the above spectrum plots notice that images of the input 1 kHz sinusoid are down \( \approx 60 \) dB, which is precisely the stopband attenuation provided by the interpolation filter.

- The variation is due to the stopband ripple.

- To push the spectral images down further with an FIR design is a serious challenge.

- IIR is likely a better choice for the speech application of the SSB project.

B.2 Decimator Design Example

When a signal is decimated the signal is first lowpass filtered then downsampled. The low-pass filter serves to prevent aliasing as the sampling rate is reduced. Downsampling by \( M \) (\( y = \text{ssd}.\text{downsample}(x, M) \)) removes \( M - 1 \) sampling for every \( M \) sampling input or equivalently retains one sample out of \( M \). The lowpass prefilter has cutoff frequency equal to the folding frequency of the output sampling rate, i.e., \( f_c = f_{s2}/2 \). Note avoid confusion with the project requirements, where the decimator is needed to take a rate \( f_{s2} \) signal back to \( f_{s1} \), let the input sampling rate be \( f_{s2} = 96000 \) Hz and the output sampling rate be \( f_{s1} = 8000 \) Hz. The input sampling rate is \( M \) times the output rate, i.e., \( f_{s2} = M f_{s1} \), so you design the lowpass filter to have cutoff \( f_c = f_{s2}/(2 \cdot L) \).

**Important Observation:** In the coherent SSB demodulator of this project, the decimator can be conveniently integrated with the lowpass filter that serves to remove the double frequency term.

In the example that follows a Chebyshev type 1 lowpass filter is designed to have cutoff around 4000 Hz. A sinusoid is used as a test input signal at sampling rate 96000 Hz.

- Note the Chebyshev lowpass filter design above is very efficient compared with the 196-tap FIR lowpass designed for use in the interpolator. It is perhaps a better overall choice. The FIR has linear phase and the IIR filter does not, but for the project this is not really an issue.
The Chebyshev type 1 design provides an asymptotic stopband

As an input consider a sinusoid at 1 kHz and observe the interpolator output spectrum compared with the input spectrum.

```matlab
# Sinusoidal test signal
n = arange(100000)
x = cos(2*pi*1000/96000*n)
# Decimate by 12 (lowpass filter followed by downsample by 12)
y = mr_dn.dn(x, 12)

# Plot the results
subplot(211)
psd(x, 2**12, 96000);
title('1 kHz Sinusoid Input to $\text{SN}=12$ Decimator')
ylabel('PSD (dB)')
ylim([-100, 0])
subplot(212)
psd(y, 2**12, 8000)
title('1 kHz Sinusoid Output from $\text{SN}=12$ Decimator')
ylabel('PSD (dB)')
ylim([-100, 0])
tight_layout()
```
Decimate by $M = 12$ input and output spectrum.

The spectrum is clean. As a simple experiment revisit the interpolator example and try out the cheby1 lowpass in place of the equal-ripple FIR design. The object `mr_dn` can be used as an interpolator with instantiating a new object, i.e., $y = \text{mr}_\text{dn}.\text{up}(x, 12)$ will work. Why? The filter coefficients held in `mr_dn` are design to have the proper cutoff frequency as $f_{s2} = 96000$ Hz in both applications.