Linear Modulation including Frequency Translation and Mixing

- Linear modulation: AM, DSB, SSB, including some digital amplitude modulation
- Frequency translation mixing including superheterodyne
- Pulse amplitude Modulation (PAM) and related
- Pulse code modulation (PCM) and relation

In [1]:
```
# %pylab inline
from numpy import * # better in import numpy as np
from matplotlib.pyplot import * # better to import matplotlib.pyplot a
import sk_dsp_comm.sigsys as ss
import sk_dsp_comm.multirate_helper as mrh
import sk_dsp_comm.digitalcom as dc
import scipy.signal as signal
from IPython.display import Audio, display
from IPython.display import Image, SVG

%config InlineBackend.figure_formats=['svg'] # SVG inline viewing
```

Wavefile Import/Export and Player

Beyond the typical `imports`, you now include the line below to be able to use the `Audio` control. The module `sigsys.py` aliased as `ss` contains the functions `fs, x = ss.from_wave("file.wav")` and `ss.to_wave("file_name",fs,x)` for importing and exporting wav files to the Python workspace.

`from IPython.display import Audio, display`
For example import one of three audio test vectors recorded at 8 ksps:

In [2]:
```
fs,xs = ss.from_wav('OSR_us_000_0018_8k.wav') # female US
#fs,xs = ss.from_wav('OSR_us_000_0030_8k.wav') # male US
#fs,xs = ss.from_wav('OSR_uk_000_0050_8k.wav') # male UK
```

Add some noise to the imported `*.wav` file and the then re-save the array `y` to `*.wav` to allow playing the file using the `Audio` control in the IPython notebook.

In [3]:
```
y = xs + 0.01*random.randn(len(xs))
```

```
ss.to_wav('xs.wav',fs,y)
```

In [4]:
```
ls *.wav
```

```
m0_hat.wav*              PCM_output.wav*          xr.wav*
OSR_uk_000_0050_8k.wav*  PCM_output16.wav*        xs.wav*
OSR_us_000_0018_8k.wav*  PCM_output8.wav*         ydd.wav*
OSR_us_000_0030_8k.wav*  PCM_output8d.wav*        yp2.wav*
part7_proj2.wav*         QAM_PCM.wav*
PCM_input.wav*           xm.wav*
```

In [5]:
```
# Audio('xs.wav')
Audio(xs,rate=8000)
```

Out[5]: 00:00 00:34

Multirate DSP

The needed multirate DSP support code is now brought into this notebook via the Python code module `sk_dsp_comm.multirate_helper.py` at the top cell.

Double-Sideband Modulation (DSB)

- Let $A(t) \propto m(t)$, the message signal, thus

$$x_c(t) = A_c m(t) \cos(2\pi f_c t) \tag{1}$$

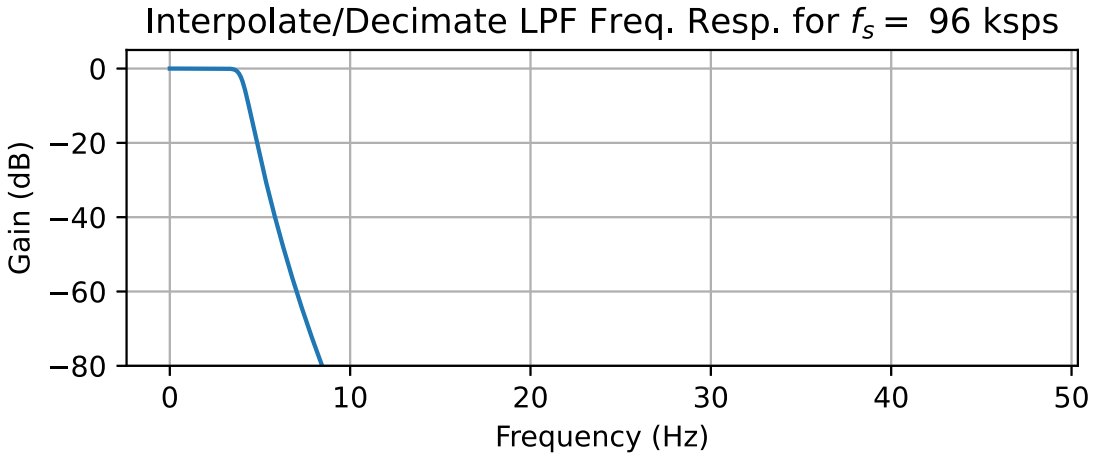
- From the modulation theorem it follows that

$$X_c(f) = \frac{1}{2} A_c M(f - f_c) + \frac{1}{2} A_c M(f + f_c) \tag{2}$$

- Now construct a simple model of the modulator in Python using an upsampling factor of 12
- With an input sampling rate of 8 ksps this makes the working sampling rate 96 ksps
- To make the resampling easier I have encapsulated the needed DSP algorithms in the *class* `rate_change(self,M_change = 12,fcutoff=1/2,N_filt_order=8)`
- This is followed by a quick check of the frequency response of the interpolation and decimation lowpass filter

In [18]:
```
by12 = mrh.rate_change(M_change=12,fcutoff=1.0,N_filt_order=12)
```

In [19]:
```
fs_up = 96 # ksps
mrh.freqz_resp(by12.b,by12.a,mode='dB',fs=fs_up,fsize=(6,2))
grid()
title(f'Interpolate/Decimate LPF Freq. Resp. for $f_s =$ {fs_up} ksps'
ylim([-80,5]);
```

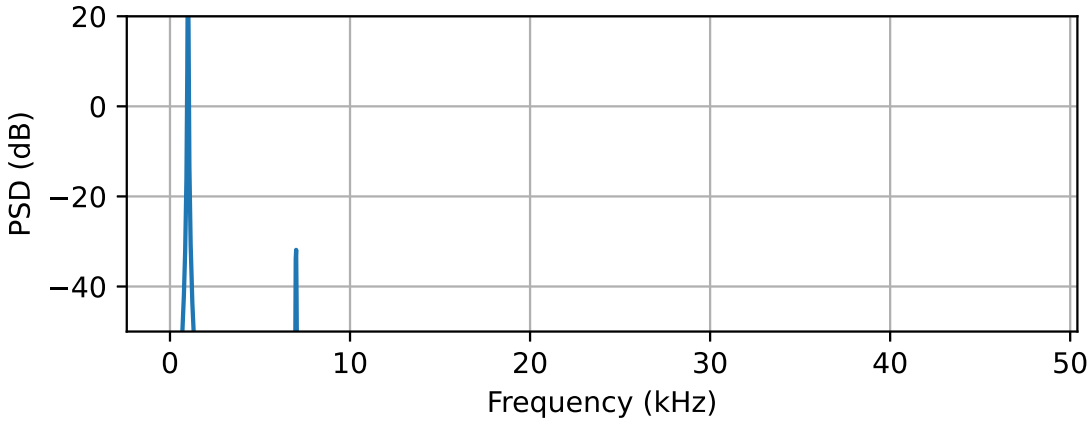


Test Signal Generation and Spectrum Verification

In [10]:
```
Tspan = 10 # Time span in seconds
```

In [11]:
```
# Create a digital data source at Rb = 1000 bps
PN,b,data = ss.nrz_bits(1000*Tspan,8000/1000)
```

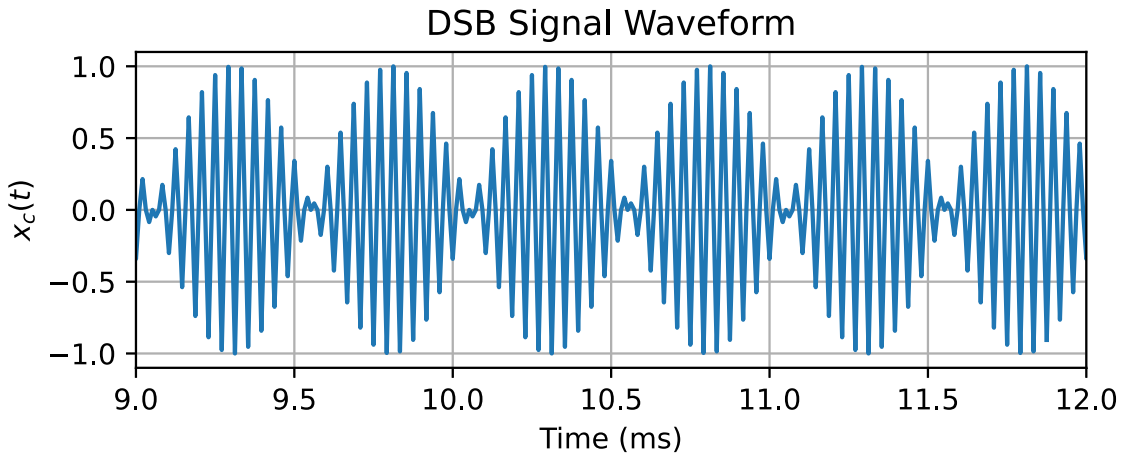
In [41]:
```
t = arange(0,Tspan,1/8000)
xm = cos(2*pi*1000*t) # fm = 1000 Hz
# xm = 3*xs[20000:len(t)+20000] # use the speech file
# xm = 1/1*PN # 100 bps digital data
xm_up = by12.up(xm)
figure(figsize=(6,2))
Pxm, f_xm = ss.psd(xm_up,n_fft=2**12,fs=fs_up,scale_noise=True); # fs
plot(f_xm,10*log10(Pxm))
ylabel(r'PSD (dB)')
ylim([-50,20])
xlabel(r'Frequency (kHz)');
grid();
```

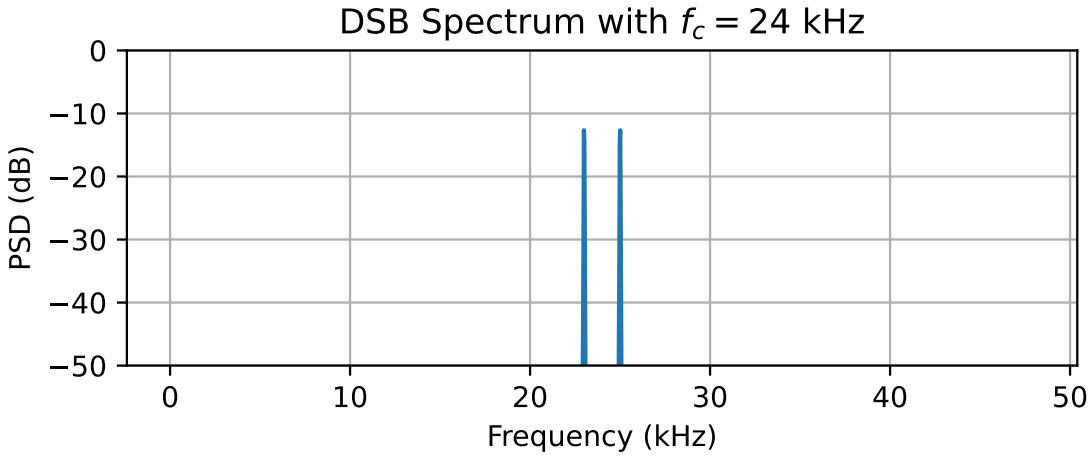


In [ ]:
```
ss.to_wav('xm.wav',fs,xm/2)
Audio('xm.wav')
```

DSB Modulator and Spectrum Verification

In [48]:
```
t_up = arange(0,Tspan,1/(8000*12))
# Multiply the message signal by cos(wc*t)
x_c = xm_up*cos(2*pi*24000*t_up)
figure(figsize=(6,2))
plot(t_up[800:1200]*1000,x_c[800:1200])
xlim([9,12])
title(r'DSB Signal Waveform')
xlabel(r'Time (ms)')
ylabel(r'$x_c(t)$')
grid();
figure(figsize=(6,2))
Pxc, f_xc = ss.psd(x_c,n_fft=2**12,fs=fs_up,scale_noise=False); # fs i
plot(f_xc,10*log10(Pxc))
ylim([-50,0])
title(r'DSB Spectrum with $f_c = 24$ kHz')
ylabel(r'PSD (dB)')
xlabel(r'Frequency (kHz)')
grid();
```





Coherent Demodulator and Spectrum Verification

- The received signal is multiplied by the signal $2\cos(2\pi f_c t)$, which is synchronous with the transmitter carrier
- For an ideal channel $x_r(t) = x_c(t)$, so

$$d(t) = \left[A_c m(t)\cos(2\pi f_c t)\right] 2\cos(2\pi f_c t) \tag{3}$$
$$= A_c m(t) + A_c m(t)\cos(2\pi(2f_c)t) \tag{4}$$

  where we have used the trig identity $2\cos^2 x = 1 + \cos 2x$ or more generally

$$\cos(x)\cos(y) = \frac{1}{2}\cos(x+y) + \frac{1}{2}\cos(x-y) \tag{5}$$

  Note: You will be using this identity a lot

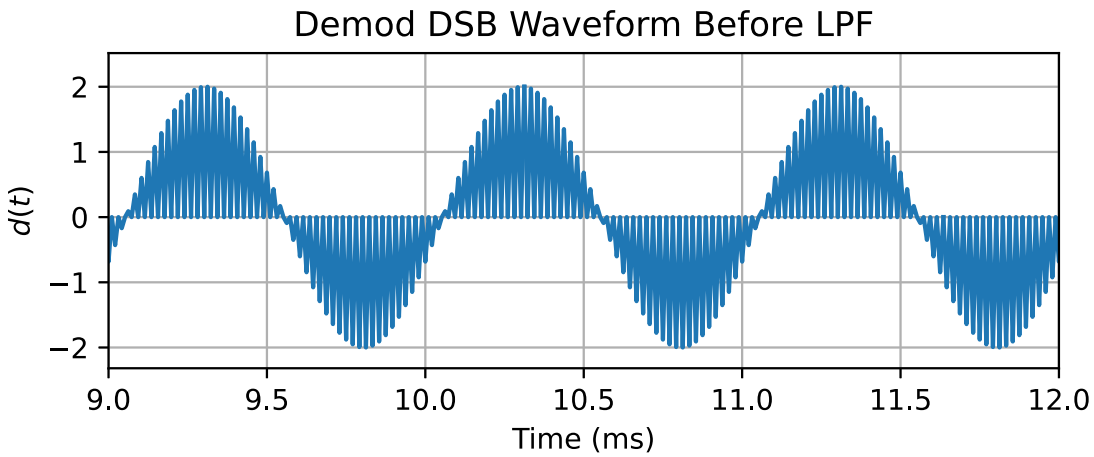
- Suppose that $\theta(t)$ is a constant or slowly varying, then the $\cos\theta(t)$ appears as a fixed or time varying attenuation factor
- Even a slowly varying attenuation can be very detrimental from a distortion standpoint
- If say $\theta(t) = \Delta f t$ and $m(t) = \cos(2\pi f_m t)$, then

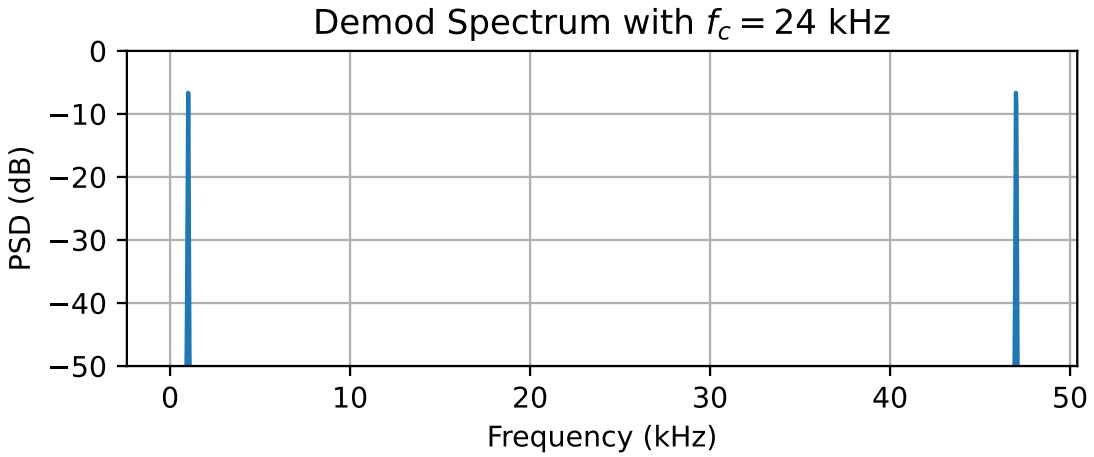
$$y_D(t) = \frac{1}{2}[\cos[2\pi(f_m - \Delta f)t] + \cos[2\pi(f_m + \Delta f)t]] \tag{6}$$

  which is the sum of two tones

- Being able to generate a coherent local reference is also a practical manner

In [49]:
```
d = 2*x_c*cos(2*pi*24000*t_up)
figure(figsize=(6,2))
#plot(t_up[800:1200]*1000,d[800:1200])
plot(t_up*1000,d)
xlim([9,12])
#xlim([0,100])
title(r'Demod DSB Waveform Before LPF')
xlabel(r'Time (ms)')
ylabel(r'$d(t)$')
grid();
figure(figsize=(6,2))
Pd, f_d = ss.psd(d,n_fft=2**12,fs=fs_up,scale_noise=False); # fs in kH
plot(f_d,10*log10(Pd))
ylim([-50,0])
title(r'Demod Spectrum with $f_c = 24$ kHz')
ylabel(r'PSD (dB)')
xlabel(r'Frequency (kHz)')
grid();
```

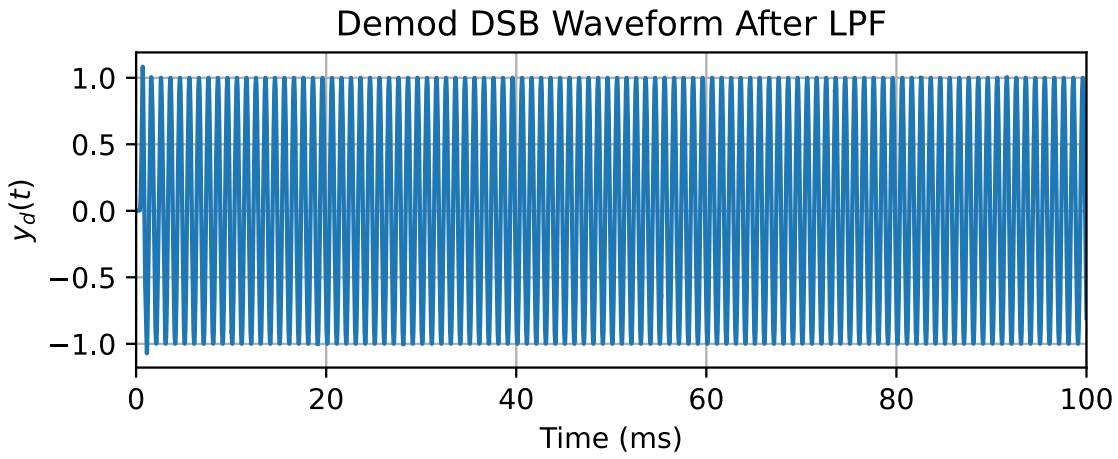


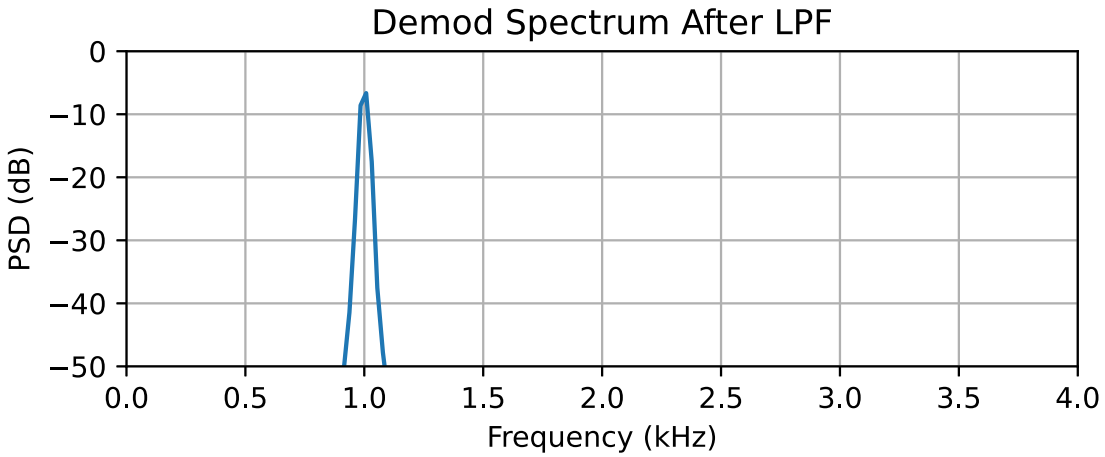


Recover the Message Using a LPF

In [51]:
```
yd = signal.lfilter(by12.b,by12.a,d)
figure(figsize=(6,2))
#plot(t_up[800:1200]*1000,yd[800:1200])
```

```
#plot(t_up[:50000]*1000,yd[:50000])
plot(t_up*1000,yd)
#xlim([0,500])
xlim([0,100])
title(r'Demod DSB Waveform After LPF')
xlabel(r'Time (ms)')
ylabel(r'$y_d(t)$')
grid();
figure(figsize=(6,2))
Pyd, f_yd = ss.psd(yd,n_fft=2**12,fs=fs_up,scale_noise=False); # fs in
plot(f_yd,10*log10(Pyd))
ylim([-50,0])
xlim([0,4])
title(r'Demod Spectrum After LPF')
ylabel(r'PSD (dB)')
xlabel(r'Frequency (kHz)')
grid();
```





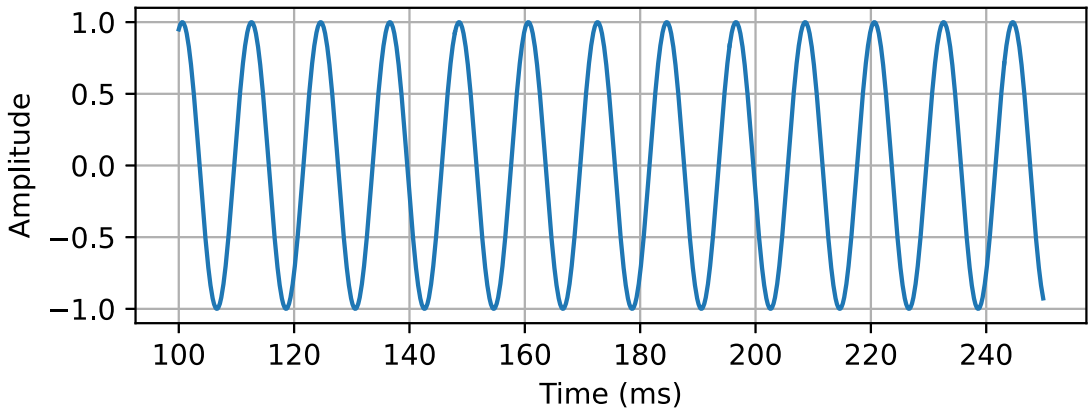
In [52]:
```
ydd = by12.dn(d)
ss.to_wav('ydd.wav',fs,ydd)
Audio('ydd.wav')
```

Out[52]: 00:00 00:10

Amplitude Modulation

In [170…

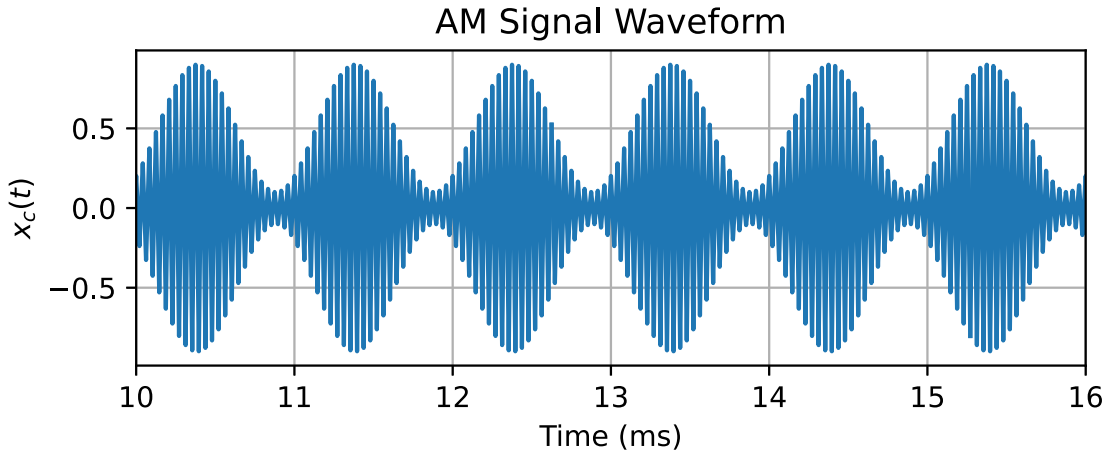
```
t = arange(0,Tspan,1/8000)
xm = cos(2*pi*1000*t)
#xm = 20*xs[20000:len(t)+20000] # use the speech file
#xm = 1/1*PN # 100 bps digital data
xm_up = by12.up(xm)
figure(figsize=(6,2))
plot(t[800:2000]*1000,xm_up[800:2000]);
grid()
#xlim([0,10])
ylabel(r'Amplitude')
xlabel(r'Time (ms)');
```

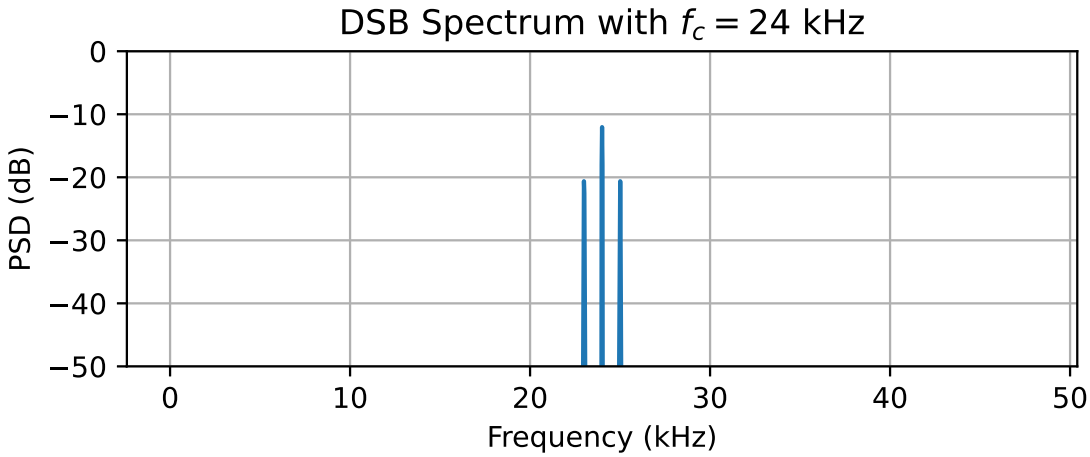


Generate AM

In [171…

```
t_up = arange(0,Tspan,1/(8000*12))
# Multiply the message signal by cos(wc*t)
a_mod = 0.8
x_c = 0.5*(1+a_mod*xm_up)*cos(2*pi*24000*t_up)
figure(figsize=(6,2))
plot(t_up[800:2000]*1000,x_c[800:2000])
xlim([10,16])
title(r'AM Signal Waveform')
xlabel(r'Time (ms)')
ylabel(r'$x_c(t)$')
grid();
figure(figsize=(6,2))
Px_c, f_xc = ss.psd(x_c,n_fft=2**12,fs=fs_up,scale_noise=False); # fs
plot(f_xc,10*log10(Px_c))
ylim([-50,0])
title(r'DSB Spectrum with $f_c = 24$ kHz')
ylabel(r'PSD (dB)')
xlabel(r'Frequency (kHz)')
grid();
```





Demodulate AM

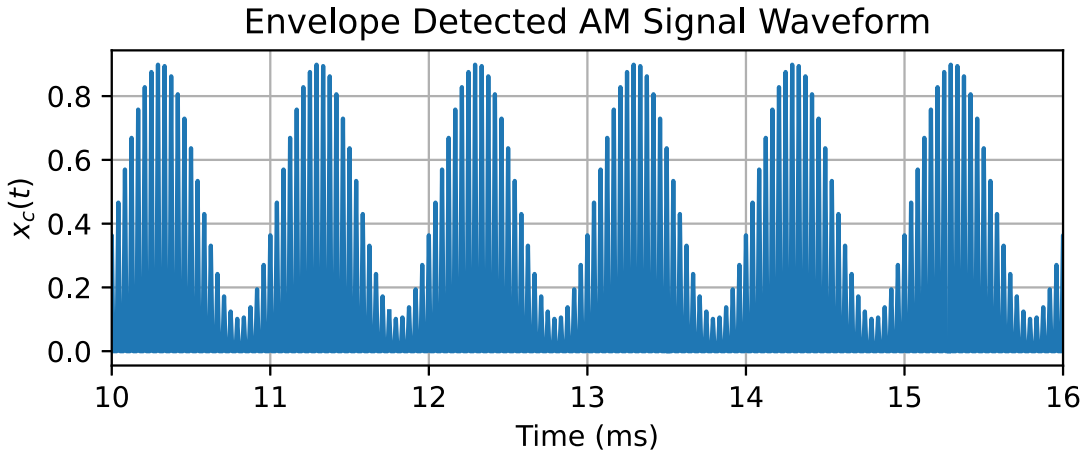
AM can be demodulated coherently, but why? Envelope detection is sufficient and easy to implement.

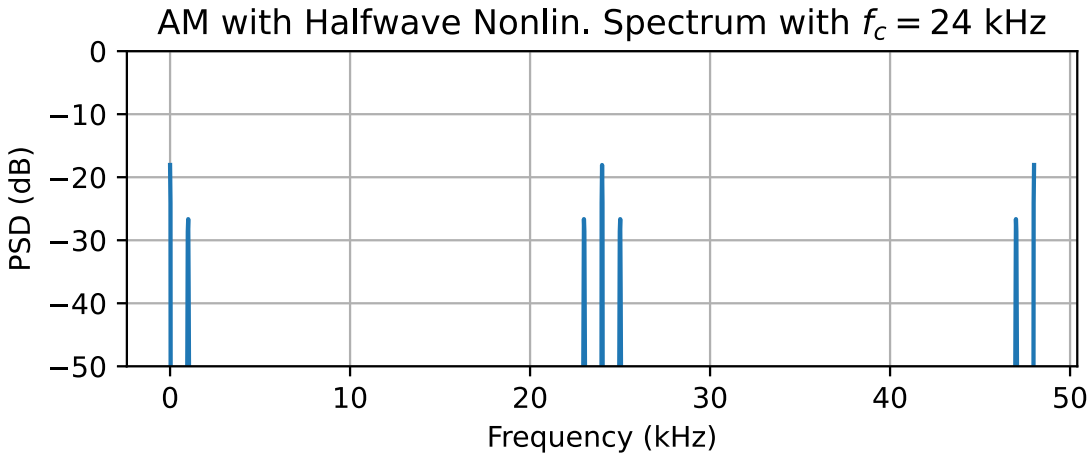
A discrete-time signal processing model actually takes a bit more work than the simple diode and RC lowpass filter found in the notes and text. There are several ways to accomplish the simple detector. As a first attempt I will start by using a biased version of the `sign()` function (also in MATLAB) to effectively multiply the signal by one when $x_c(t)$ is positive and zero when $x_c(t)$ is negative. We will be using the coresponding theoretical function $\mathrm{sgn}()$ in the discussion of the Hilbert transform a little later. Formally,

$$\mathrm{sgn}(x) = \mathrm{sign}(x) = \begin{cases} 1, & x > 0 \\ -1, & x < 0 \end{cases} \tag{7}$$

In [60]:
```
# Envelope Detector
x_env = x_c*(1+sign(x_c))/2
figure(figsize=(6,2))
plot(t_up[800:2000]*1000,x_env[800:2000])
```

```
xlim([10,16])
title(r'Envelope Detected AM Signal Waveform')
xlabel(r'Time (ms)')
ylabel(r'$x_c(t)$')
grid();
figure(figsize=(6,2))
Px_env, f_x_env = ss.psd(x_env,n_fft=2**12,fs=fs_up,scale_noise=False)
plot(f_x_env,10*log10(Px_env))
ylim([-50,0])
title(r'AM with Halfwave Nonlin. Spectrum with $f_c = 24$ kHz')
ylabel(r'PSD (dB)')
xlabel(r'Frequency (kHz)');
grid();
```





The next step is to implement a lowpass filter. In the simple analog circuit diode RC filter combination forms a nonlinear system, so the detailed analysis is not as easy as it looks. In the discrete-time you can implement a first-order discrete-time lowpass that will be fulled decoupled from the envelope detector, hence nonlinear processing follwed by linear processing with no interaction.

A first-order lowpass can be implemented via the difference equation

$$y[n] = ay[n-1] + (1-a)x[n], \tag{8}$$

where a is set to model the RC time constant of interest. Consider the impulse response

$$h_a(t) = e^{-t/RC}u(t) \tag{9}$$

If you sample this impulse response at rate f_s Hz, you obtain an *impulse invariant* discrete-time impulse response

$$h[n] = e^{-n/(fs \cdot RC)}u[n] = (e^{-2\pi f_3/f_s})^n u[n] \tag{10}$$

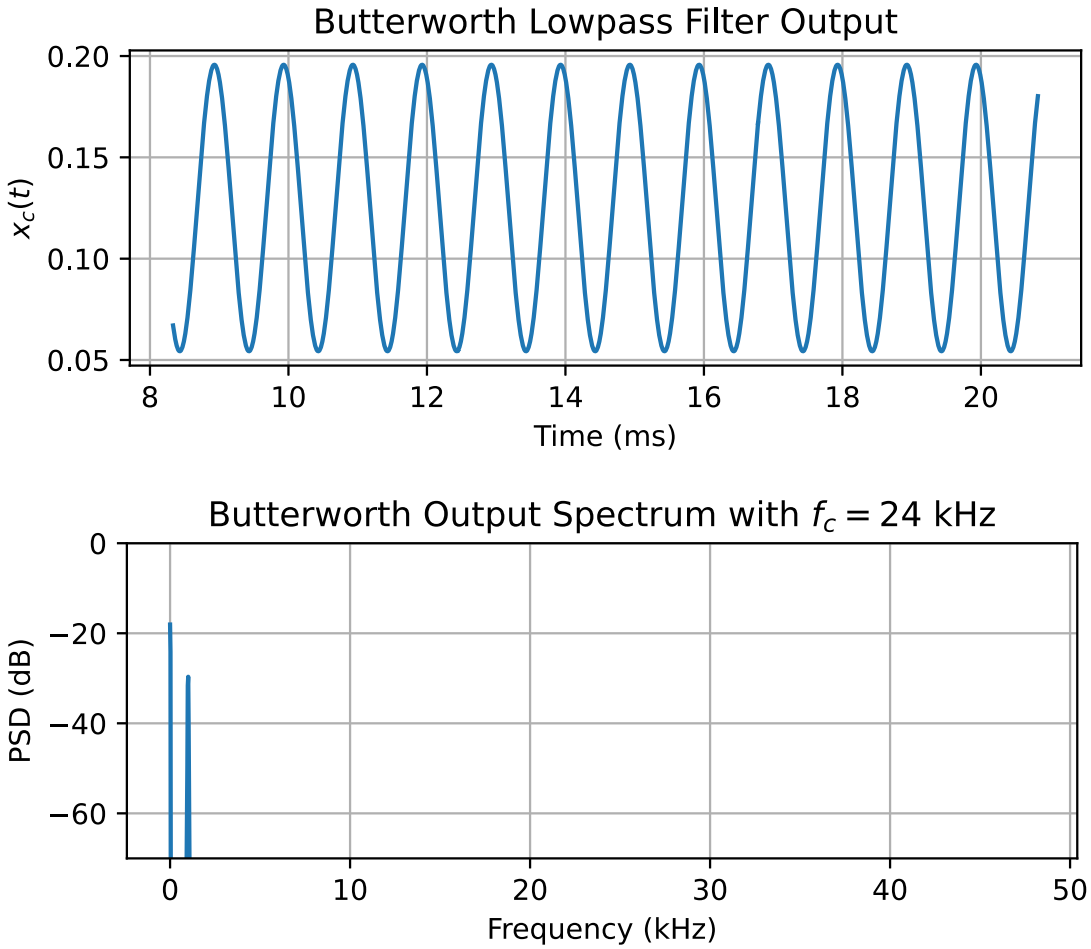
where $f_3 = 1/(2\pi RC)$ is the RC lowpass 3dB cutoff frequency. In the difference model

$$a = e^{-2\pi f_3/f_s} \tag{11}$$

In [ ]:
```
# Define a for  f3 = 1000 Hz
a = exp(-2*pi*1000/96000)
e_o = signal.lfilter([1-a],[1,-a],x_env)
figure(figsize=(6,2))
plot(t_up[800:2000]*1000,e_o[800:2000])
xlim([10,16])
title(r'Lowpass Filter Output')
xlabel(r'Time (ms)')
ylabel(r'$x_c(t)$')
grid();
```

The above plot shows that if is difficult to filter the envelope detector output to remove the carrier cycles. Reducing the cutoff frequency f_3 helps with carrier removal, but also takes the signal. Consider a higher-order filter.

In [64]:
```
# Replace 1st-order Filter with N = 5 Butterworth
b5,a5 = signal.butter(5,2*1000/96000)
e_o = signal.lfilter(b5,a5,x_env)
figure(figsize=(6,2))
plot(t_up[800:2000]*1000,e_o[800:2000])
#xlim([10,16])
title(r'Butterworth Lowpass Filter Output')
xlabel(r'Time (ms)')
ylabel(r'$x_c(t)$')
grid();
figure(figsize=(6,2))
P_e_o, f_e_o = ss.psd(e_o,n_fft=2**12,fs=fs_up,scale_noise=False); # f
plot(f_e_o,10*log10(P_e_o))
ylim([-70,0])
title(r'Butterworth Output Spectrum with $f_c = 24$ kHz')
ylabel(r'PSD (dB)')
xlabel(r'Frequency (kHz)')
grid();
```



In the above two plots you see:

- The time domain waveform is now very clean, with no obvious sign of the carrier and its harmonics
- The spectrum plot shows down to 60 dB below the 1000 Hz recovered message, there is nothing visible

In [65]:
```
ydd = by12.dn(e_o)
ss.to_wav('ydd.wav',fs,ydd)
Audio('ydd.wav')
```

Out[65]: 00:00 00:10

Example: Multiple Sinusoids and Finding min(m(t))

- Suppose that $m(t)$ is a sum of multiple sinusoids (multi-tone AM)

$$m(t) = \sum_{k=1}^{M} A_k \cos(2\pi f_k t + \phi_k) \tag{12}$$

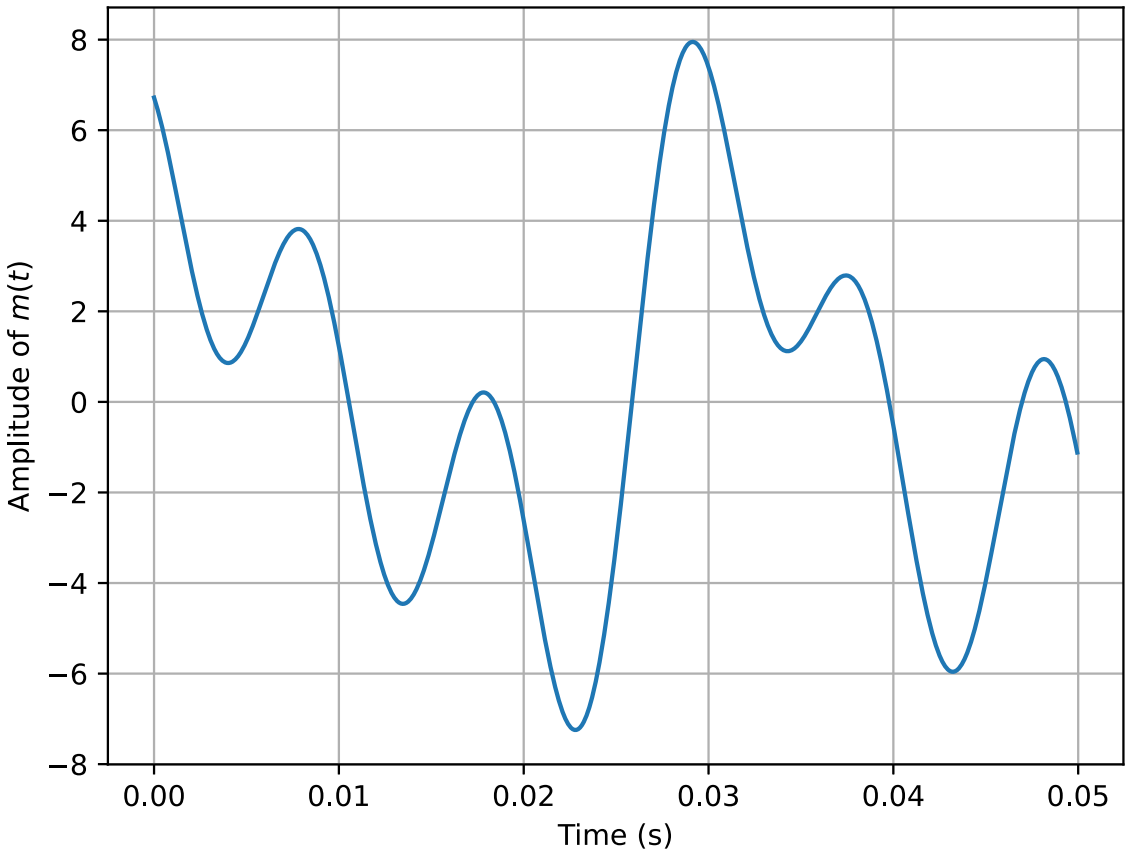
where M is the number of sinusoids, f_k values might be constrained over some band of frequencies W, e.g., $f_k \leq W$, and the phase values ϕ_k can be any value on $[0, 2\pi]$

- To find $m_n(t)$ we need to find $\min m(t)$
- A lower bound on $\min m(t)$ is $-\sum_{k=1}^{M} A_k$; why?
- The worst case value may not occur in practice depending upon the phase and frequency values, so we may have to resort to a numerical search or a plot of the waveform
- Suppose that $M = 3$ with $f_k = \{65, 100, 35\}$ Hz, $A_k = \{2, 3.5, 4.2\}$, and $\phi_k = \{0, \pi/3, -\pi/4\}$ rad
- Note the fundamental frquency here is 5 Hz (found using the gcd())

In [66]:
```
def M_sinusoids(Nsamp,f,A,phi,fsamp):
    """
    m,t = M_sinusoids(Nsamp,f,A,phi,fsamp)

    Mark Wickert, February 2015

    For a sum of sinusoids having amplitude and phase values
    stored in vectors f and phi respectively. The number of
    samples in the waveform will be Nsamp and the sampling
    rate will be fsamp
    """
    n = arange(0,Nsamp)
    t = n/fsamp
    M = len(f)
    m = zeros_like(t)

    for k in range(M):
        m += A[k]*cos(2*pi*f[k]*t + phi[k])
    return m, t
```

In [67]:
```
m,t = M_sinusoids(1000,[65, 100, 35],[2, 3.5, 4.2],[0, pi/3, -pi/4], 2
plot(t,m)
ylabel(r'Amplitude of $m(t)$')
xlabel(r'Time (s)')
grid();
print('min(m) = %6.4f and lower bound -sum([2, 3.5, 4.2] = %6.4f' \
      % ((min(m),-sum([2, 3.5, 4.2]))))
```

```
min(m) = -7.2462 and lower bound -sum([2, 3.5, 4.2] = -9.7000
```



Finding the Power in $m_n(t)$

Direct calculation makes use of the fact that the total power is just the sum of the powers of the indivdual terms (see the Chapter 2 Jupyter notebook for a proof), providing the frdquencies are unique. If any frequencies are the same, these terms must first be combined using the phasor addition formula, first seen in ECE2610.

$$\langle m_n^2(t) \rangle = \frac{1}{7.2462^2}\left[\frac{2^2}{2} + \frac{3.5^2}{2} + \frac{4.2^2}{2}\right] = 0.3227 \text{ W}$$

In [68]:
```
1/7.2462**2*(4/2 + 3.5**2/2 + 4.2**2/2)
```

Out[68]:
```
0.32271632836469183
```

- First try using the Numpy stats function `var()` on the sequence of samples.

In [74]:
```
# We need to use more samples and increase the sampling rate
m,t = M_sinusoids(10000,[65, 100, 35],[2, 3.5, 4.2],[0, pi/3, -pi/4],
print(f'<m_n^2(t)> = {var(m/7.2462):2.4f} W')
```

```
<m_n^2(t)> = 0.3227 W
```

In [70]:
```
import scipy.integrate as integrate
```

In [71]:
```
# The integrand for finding the average of m_n^2(t)
def mn_sq(t):
    """
    mn**2
    """
    m = 2*cos(2*pi*65*t + 0)
    m += 3.5*cos(2*pi*100*t + pi/2)
    m += 4.2*cos(2*pi*35*t - pi/4)
    m /= 7.2462
    return m**2
```

In [73]:
```
T_inte = 1.0
[I,err] = integrate.quad(mn_sq,-T_inte/2,T_inte/2,epsabs=1e-6, epsrel=
print(f'<m_n^2(t)> = {I/T_inte:2.4f} W')
```

```
<m_n^2(t)> = 0.3227 W
```

Plotting the Spectrum of a Multi-Tone AM Signal

Make use of the `ss.line_spectra()` function by filling arrays with the proper frequency values and the amplitude values for a two-sided spectrum. The AM signal is assumed to be of the form

$$x_c(t) = A_c \left[1 + \frac{a}{|\min m(t)|} \sum_{k=1}^{M} A_{m,k} \cos(2\pi f_{m,k} t + \theta_{m,k}) \right] \cos(2\pi f_c t) \quad (1$$

where

$$m(t) = \sum_{k=1}^{M} A_{m,k} \cos(2\pi f_{m,k} t + \theta_{m,k}) \tag{14}$$

The function `multi_tone_AM_spectra(A_c,f_c,a_index,min_m,A_k,f_mk,theta_mk)` takes all of the parameters in the above equation as scalars for the first four arguments and lists (or `ndarrays`) for the remaining three sinuusoidal modulation parameters. The function returns `ndarrays` for inserting into `ss.line_spectra()`.

In [76]:
```
def multi_tone_AM_spectra(A_c,f_c,a_index,min_m,A_k,f_mk,theta_mk):
```

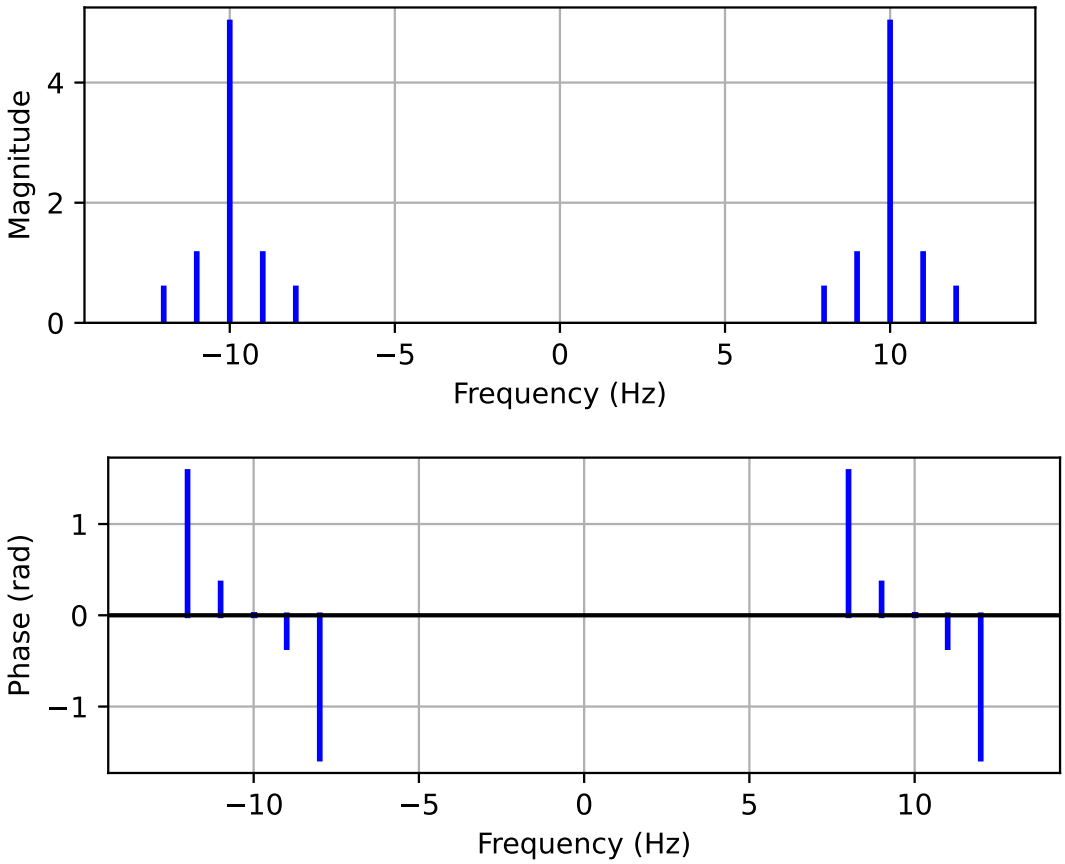
```
    """
    Organize the parameters of multi-tone AM modulation into a form
    suitable for using ssd.line_spectra for magnitude and phase plots.

    A_c, f_c, a_index, and min_m are scalar parameters
    A_k, f_mk, and theta_mk are 1D lists or ndarrays of the
    same length, e.g., M

    Mark Wickert February 2017
    """
    M = len(A_k)
    f_k = zeros(2*M + 1)
    X_k = zeros(2*M + 1,dtype=complex128)
    f_k[0] = f_c
    X_k[0] = A_c/2
    print('f_c = %4.2f, A_c = %4.2f' % (f_k[0],2*abs(X_k[0])))
    scale_factor = A_c*a_index/(4*abs(min_m))
    for k in range(M):
        X_k[2*k+1] = scale_factor*A_k[k]*exp(1j*theta_mk[k])
        f_k[2*k+1] = f_c + f_mk[k]
        X_k[2*k+2] = scale_factor*A_k[k]*exp(-1j*theta_mk[k])
        f_k[2*k+2] = f_c - f_mk[k]
        print('f_m[%d]_up = %4.2f, |A_m| = %4.2f, /_A_m = %4.2f (rad)'
              (k,f_k[2*k+1],2*abs(X_k[2*k+1]),angle(X_k[2*k+2])))
        print('f_m[%d]_low = %4.2f, |A_m| = %4.2f, /_A_m = %4.2f (rad)
              (k,f_k[2*k+2],2*abs(X_k[2*k+2]),angle(X_k[2*k+2])))
    return f_k, X_k
```

Text Example 3.1

In [77]:
```
f_k, X_k = multi_tone_AM_spectra(10,10,0.5,-4.3642,
                                 [4,2],[1,2],[-pi/9,-pi/2])
```

```
f_c = 10.00, A_c = 10.00
f_m[0]_up = 11.00, |A_m| = 2.29, /_A_m = 0.35 (rad)
f_m[0]_low = 9.00, |A_m| = 2.29, /_A_m = 0.35 (rad)
f_m[1]_up = 12.00, |A_m| = 1.15, /_A_m = 1.57 (rad)
f_m[1]_low = 8.00, |A_m| = 1.15, /_A_m = 1.57 (rad)
```

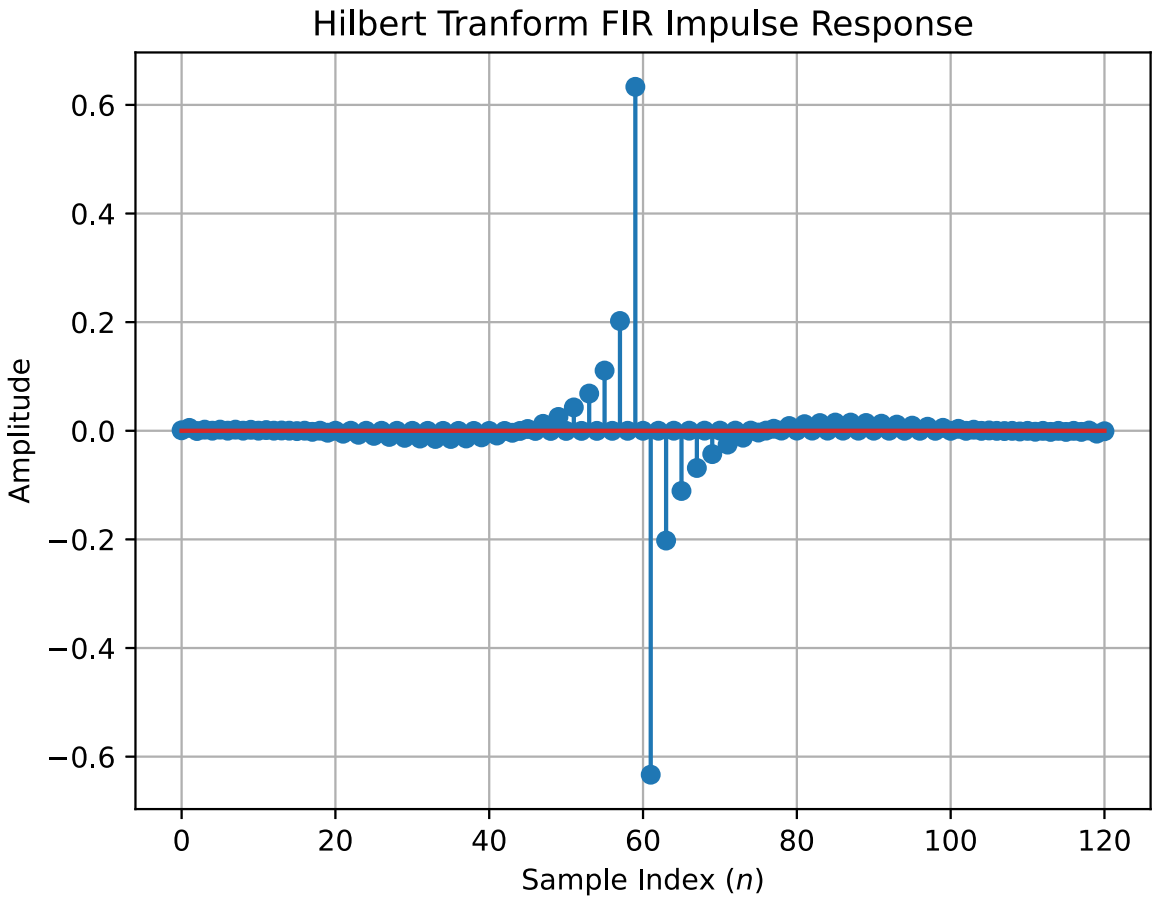
In [78]:
```
ss.line_spectra(f_k,X_k,'mag',fsize=(6,2))
ss.line_spectra(f_k,X_k,'phase',fsize=(6,2))
```



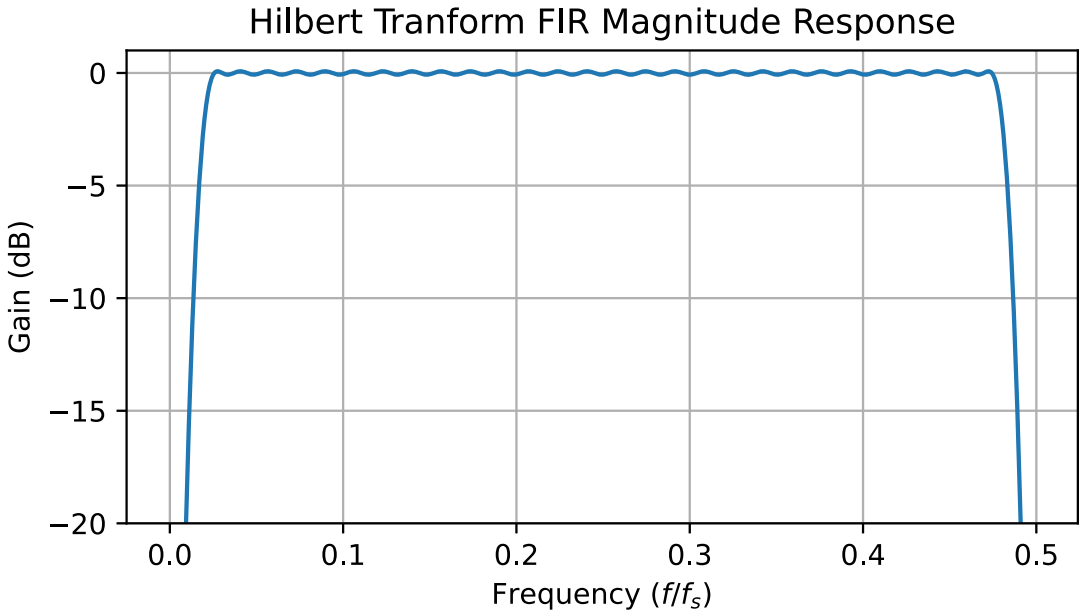
Hilbert Transform

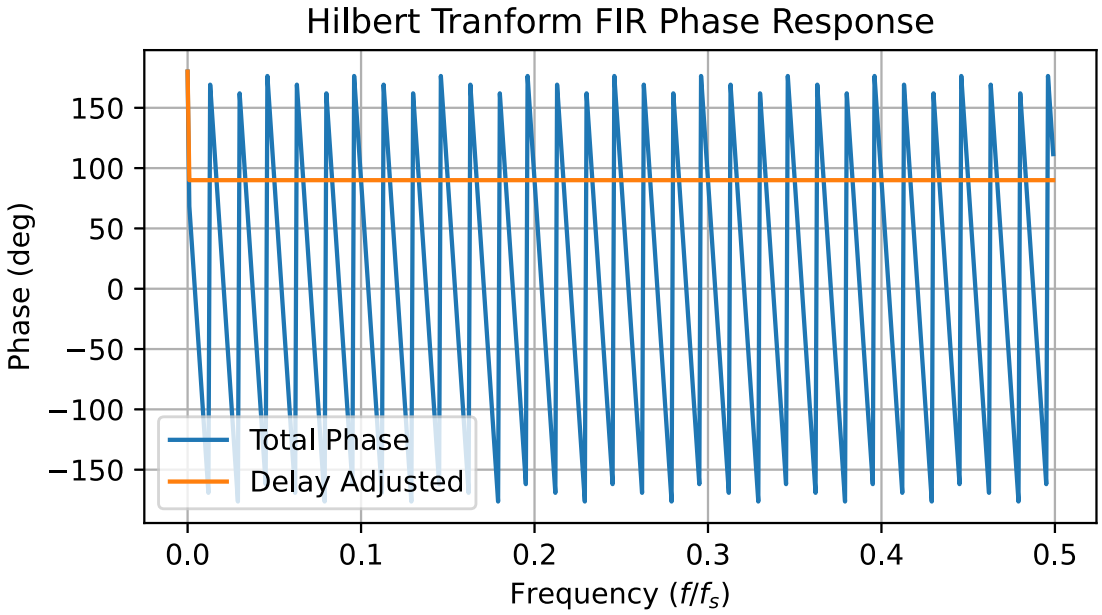
Consider the design a Hilbert transforming digital filter. The `scipy.signal` module is capable of such designs.

In [80]:
```
b31 = signal.remez(31,array([0, .05,.1,.9,0.95,1])/2,[.1,1,.1],
                   fs=1,type='hilbert')
b63 = signal.remez(63,array([0, .01,.05,.95,0.99,1])/2,[.01,1,.01],
                   fs=1,type='hilbert')
b121 = signal.remez(121,array([0, .01,.05,.95,0.99,1])/2,[.01,1,.01],
                    fs=1,type='hilbert')
stem(b121)
title(r'Hilbert Tranform FIR Impulse Response')
ylabel(r'Amplitude')
xlabel(r'Sample Index ($n$)')
grid();
```

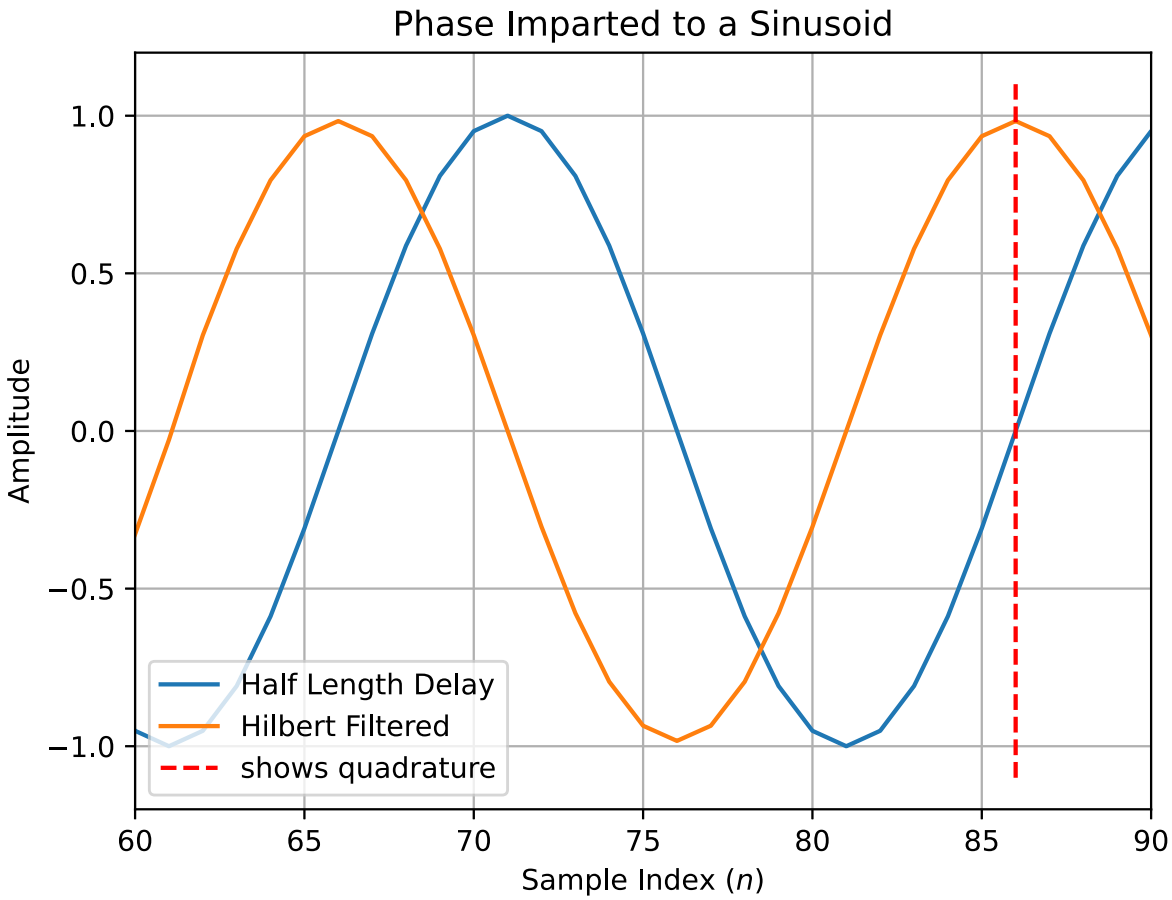


In [81]:
```
f = arange(0,.5,.001)
w,H = signal.freqz(b121,1,2*pi*f)
figure(figsize=(6,3))
plot(f,20*log10(abs(H)))
ylim([-20,1])
title(r'Hilbert Tranform FIR Magnitude Response')
ylabel(r'Gain (dB)')
xlabel(r'Frequency ($f/f_s$)')
grid();
figure(figsize=(6,3))
plot(f,angle(H)*180/pi)
plot(f,(unwrap(angle(H))+2*pi*(121-1)/2*f)*180/pi)
title(r'Hilbert Tranform FIR Phase Response')
ylabel(r'Phase (deg)')
xlabel(r'Frequency ($f/f_s$)')
legend((r'Total Phase',r'Delay Adjusted'),loc='lower left')
grid();
```





In [82]:
```
n = arange(0,100)
x = cos(2*pi*0.05*n)
b_del = zeros(32)
b_del[-1] = 1 # place a 1 at the end of the 32 tap delay
y_del = signal.lfilter(b_del,1,x)
y_hil = signal.lfilter(b63,1,x)
plot(n,y_del)
plot(n,y_hil)
plot([86,86],[-1.1,1.1],'r--')
ylim([-1.2,1.2])
xlim([60,90])
title(r'Phase Imparted to a Sinusoid')
ylabel(r'Amplitude')
xlabel(r'Sample Index ($n$)')
legend((r'Half Length Delay',r'Hilbert Filtered',r'shows quadrature'),
grid();
```



Single Sideband

Using the filter designed above we can create a single sideband signal at baseband, and then shift it up to a desired carrier frequency.

In [157…
```
def analytic_signal(m,b_hilbert,usb_lsb='upper'):
    b_del = zeros((len(b_hilbert)+1)//2)
    b_del[-1] = 1
    y_del = signal.lfilter(b_del,1,m)
    y_hil = signal.lfilter(b_hilbert,1,m)
    if usb_lsb == 'upper':
        return y_del - 1j*y_hil
    else:
        return y_del + 1j*y_hil
```

In [158…
```
#fs,xs = ss.from_wav('OSR_us_000_0018_8k.wav') # female US
fs,xs = ss.from_wav('OSR_us_000_0030_8k.wav') # male US
#fs,xs = ss.from_wav('OSR_uk_000_0050_8k.wav') # male UK
```

In [159…
```
ss.to_wav('xs.wav',8000,xs)
Audio('xs.wav')
```

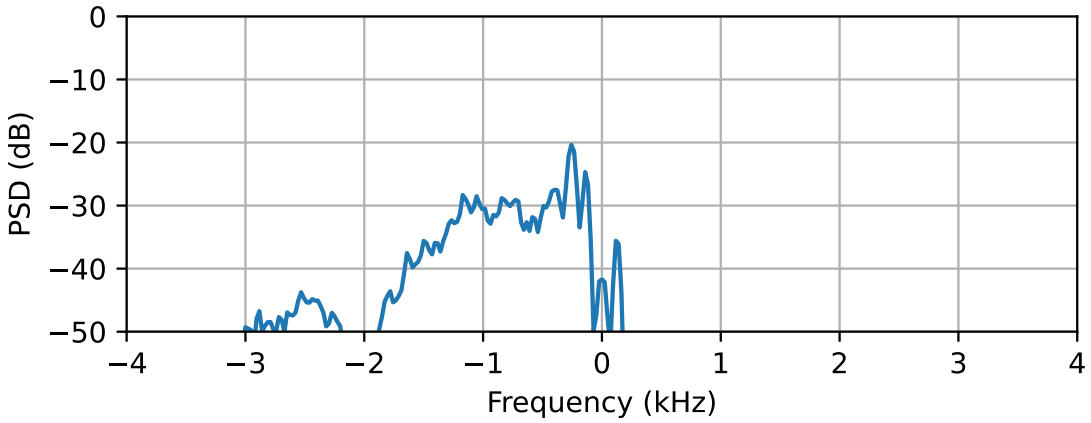
Out [159… 00:00 00:46

In [160…
```
by12 = mrh.rate_change(M_change=12,fcutoff=0.8,N_filt_order=12)
```

Transmit Signal Generation

Create Analytic Signal Version of m(t)

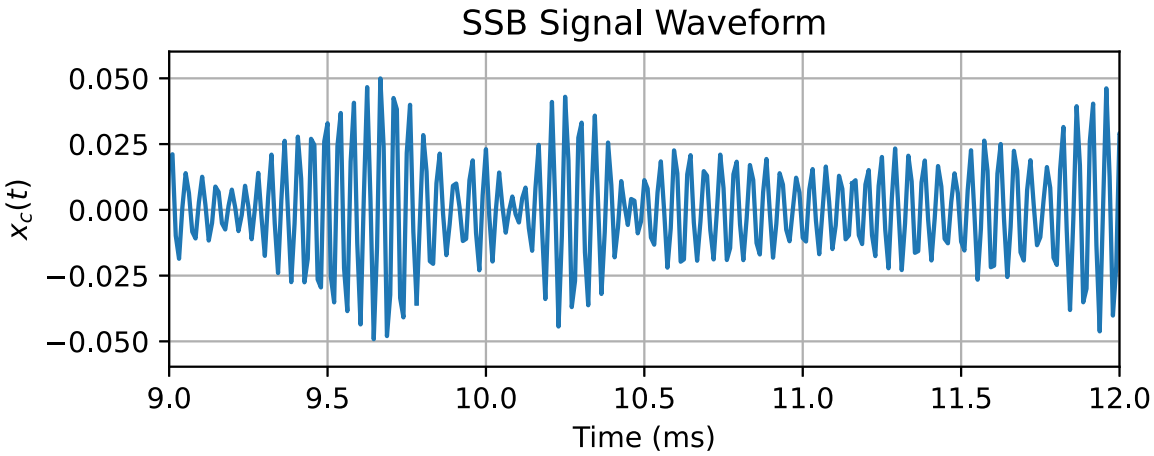
In [185…
```
Tspan = 10 # Time span in seconds
t = arange(0,Tspan,1/8000)
# xm = cos(2*pi*1000*t)
xm = 3*xs[20000:len(t)+20000] # use the speech file
#xm = 1/1*PN # 100 bps digital data
xm_SSB = analytic_signal(xm,b121,'lower')
xm_up = by12.up(xm_SSB)
figure(figsize=(6,2))
P_xm_up, f_xm_up = ss.psd(xm_up,n_fft=2**12,fs=fs_up,scale_noise=False
plot(f_xm_up,10*log10(P_xm_up))
xlim([-4,4])
ylabel(r'PSD (dB)')
ylim([-50,0])
xlabel(r'Frequency (kHz)')
grid();
```

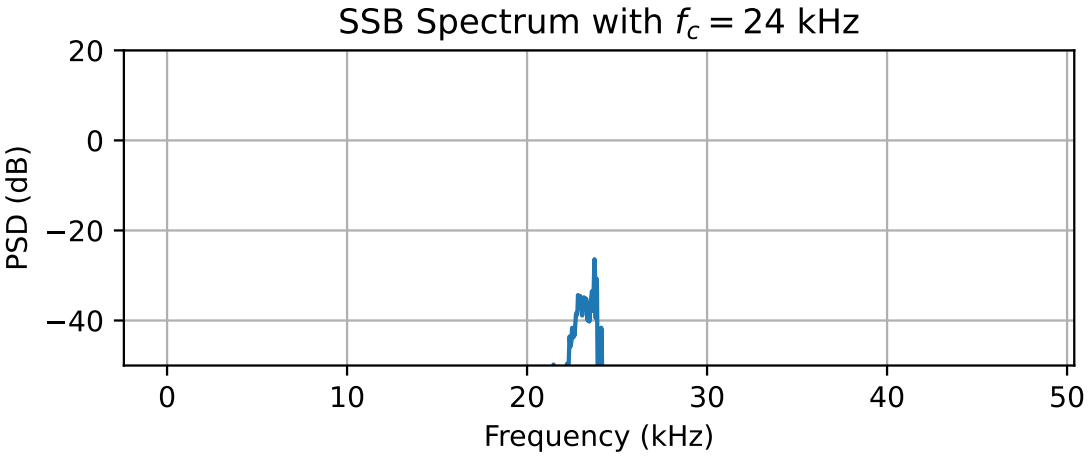


Up Convert the Analytic Signal and Take the Real Part

In [186…
```
t_up = arange(0,Tspan,1/(8000*12))
# Multiply the message signal by cos(wc*t)
x_c = real(xm_up*exp(1j*2*pi*24000*t_up))
figure(figsize=(6,2))
plot(t_up[800:1200]*1000,x_c[800:1200])
xlim([9,12])
title(r'SSB Signal Waveform')
xlabel(r'Time (ms)')
```

```
ylabel(r'$x_c(t)$')
grid();
figure(figsize=(6,2))
P_x_c_up, f_x_c_up = ss.psd(x_c,n_fft=2**12,fs=fs_up,scale_noise=False
plot(f_x_c_up,10*log10(P_x_c_up))
ylim([-50,20])
title(r'SSB Spectrum with $f_c = 24$ kHz')
ylabel(r'PSD (dB)')
xlabel(r'Frequency (kHz)')
grid();
```





Receiver Using Coherent Demodulation

Additional Analysis of SSB Demod with Imperfect Reference

In the course notes the coherent demod is analyzed for message $m(t)$ and a reference of the form

$$4\cos([2\pi f_c t + \theta(t)]$$

Let us now consider the special case of

$$m(t) = \cos(2\pi f_m t) \tag{15}$$
$$\theta(t) = 2\pi \Delta f t \tag{16}$$

We make subsitutions into the general demod result

$$y_D(t) = m(t)\cos\theta(t) \mp \hat{m}(t)\sin\theta(t)$$

where the top sign is USSB and bottom sign is LSSB.

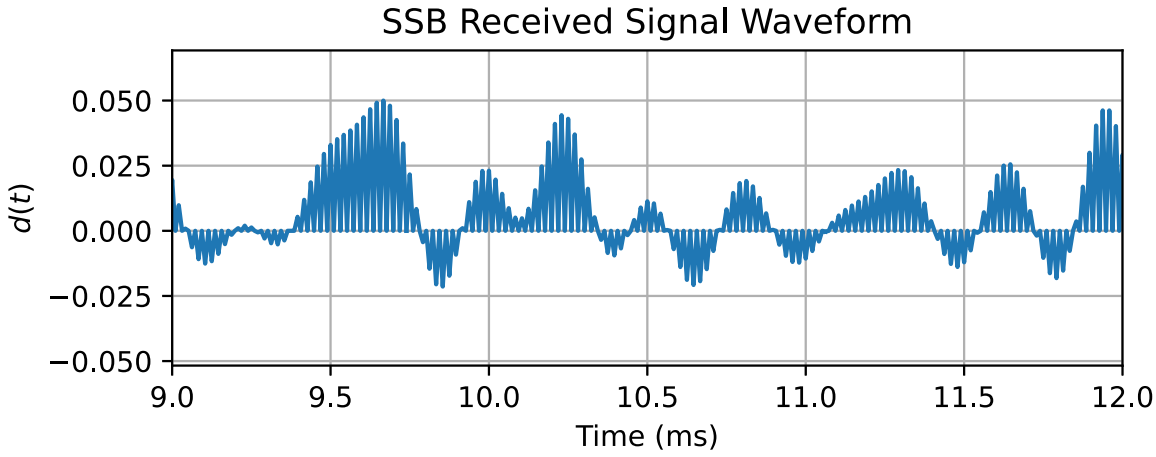
We know that $\hat{m}(t) = \sin(2\pi f_m t)$. Working with trig identities we have

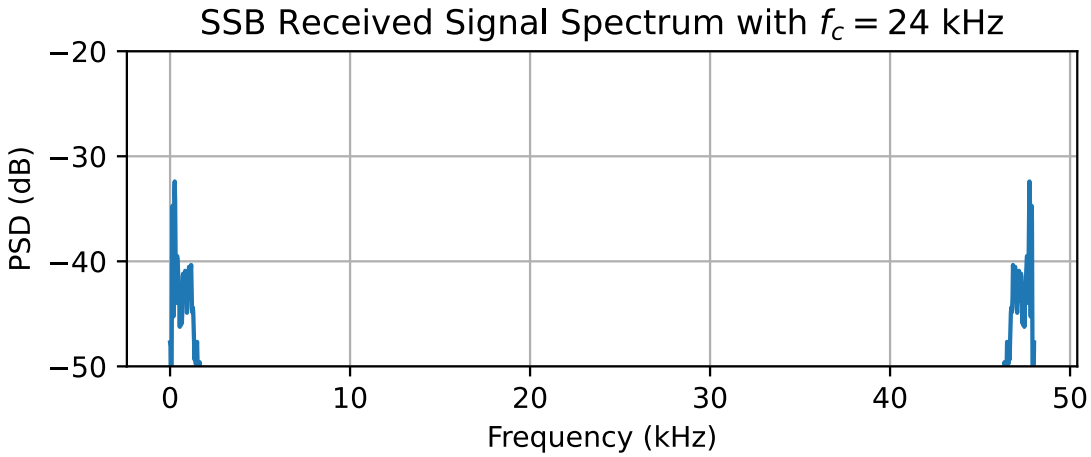
$$y_D(t) = \cos(2\pi f_m t)\cos(2\pi \Delta f t) \mp \sin(2\pi f_m t)\sin(2\pi \Delta f t) \tag{17}$$
$$= \frac{1}{2}\left[\cos[2\pi(f_m + \Delta f)t] + \cos[2\pi(f_m - \Delta f)t]\right] \tag{18}$$
$$\pm \frac{1}{2}\left[\cos[2\pi(f_m + \Delta f)t] - \cos[2\pi(f_m - \Delta f)t]\right] \tag{19}$$

Summarizing for the two cases:

- USSB: $y_D(t) = \cos[2\pi(f_m + \Delta f)t]$
- LSSB: $y_D(t) = \cos[2\pi(f_m - \Delta f)t]$
- Which says the message tone is either up-shifted **or** down-shifted, but not both simulataneously as in the case of DSB demodulation

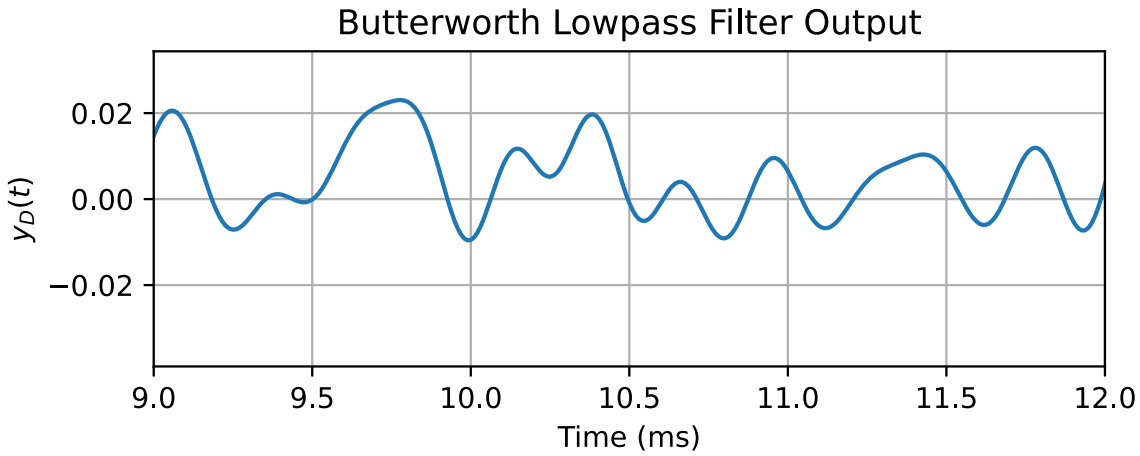
In [198…
```
# Multiply received signal by cos() at fc
d = x_c*cos(2*pi*24000*t_up)
figure(figsize=(6,2))
plot(t_up[800:1400]*1000,d[800:1400])
xlim([9,12])
title(r'SSB Received Signal Waveform')
xlabel(r'Time (ms)')
ylabel(r'$d(t)$')
grid();
figure(figsize=(6,2))
P_d, f_d = ss.psd(d, 2**12,fs=fs_up,scale_noise=False)
plot(f_d,10*log10(P_d))
ylim([-50,-20])
title(r'SSB Received Signal Spectrum with $f_c = 24$ kHz')
ylabel(r'PSD (dB)')
xlabel(r'Frequency (kHz)')
grid();
```





In [188…

```
# Lowpass Filter
b5,a5 = signal.butter(5,2*4000/96000)
yD = signal.lfilter(b5,a5,d)
figure(figsize=(6,2))
plot(t_up[800:2000]*1000,yD[800:2000])
xlim([9,12])
title(r'Butterworth Lowpass Filter Output')
xlabel(r'Time (ms)')
ylabel(r'$y_D(t)$')
grid();
```

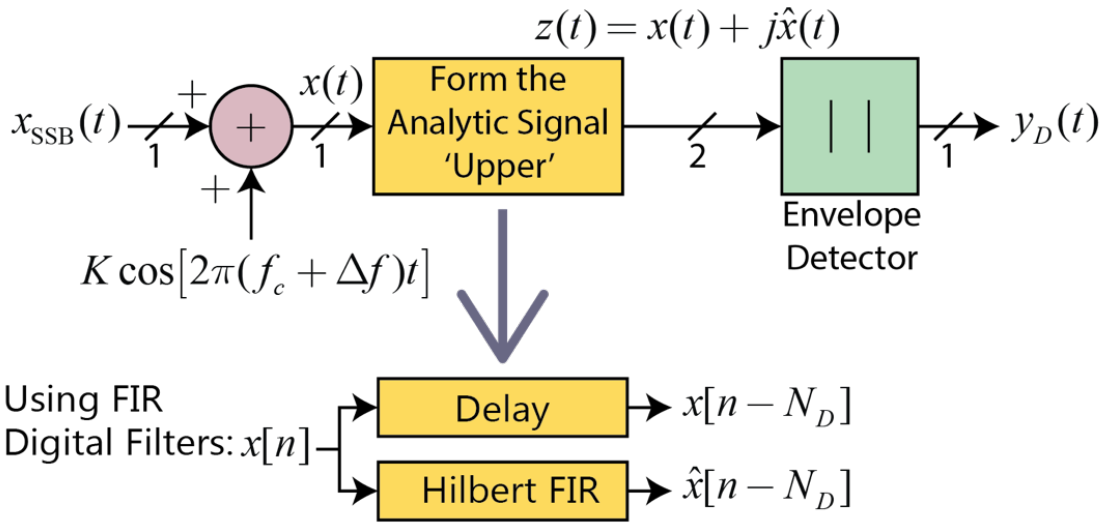


In [189…
```
ydd = by12.dn(yD - mean(yD))
ss.to_wav('ydd.wav',8000,ydd)
Audio('ydd.wav')
```

Out[189… 00:00 00:10

Receiver Using Carrier Reinsertion and Envelope Detection

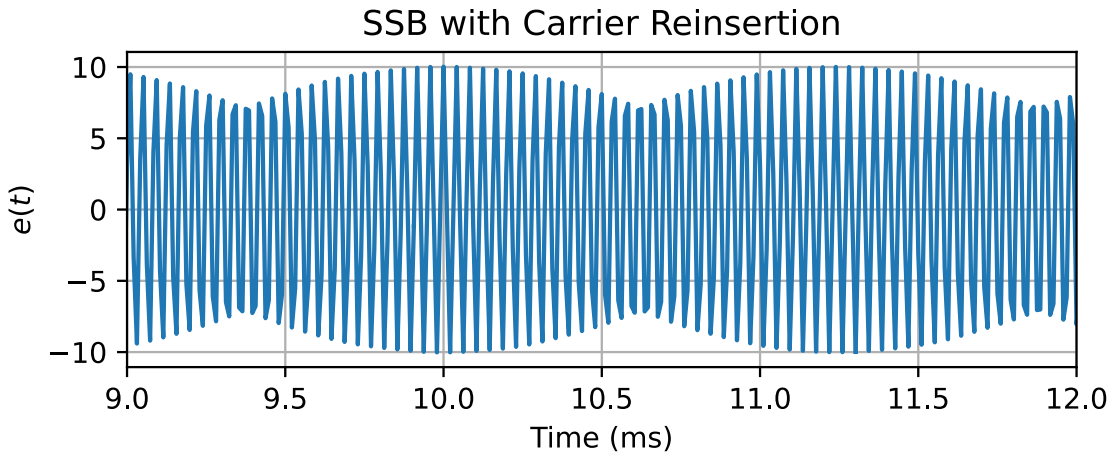
Here we implement an envelope detector by first forming the analytic signal from the received *real* signal, then obtain the envelope by simply obtaining the absolute value. This is a convenient DSP-based means to form an envelope detector in software, as opposed to the traditional use of a diode followed by a RC lowpass filter. DC bias is removed by subtracting the signal mean. The block diagram below shows the configuration.

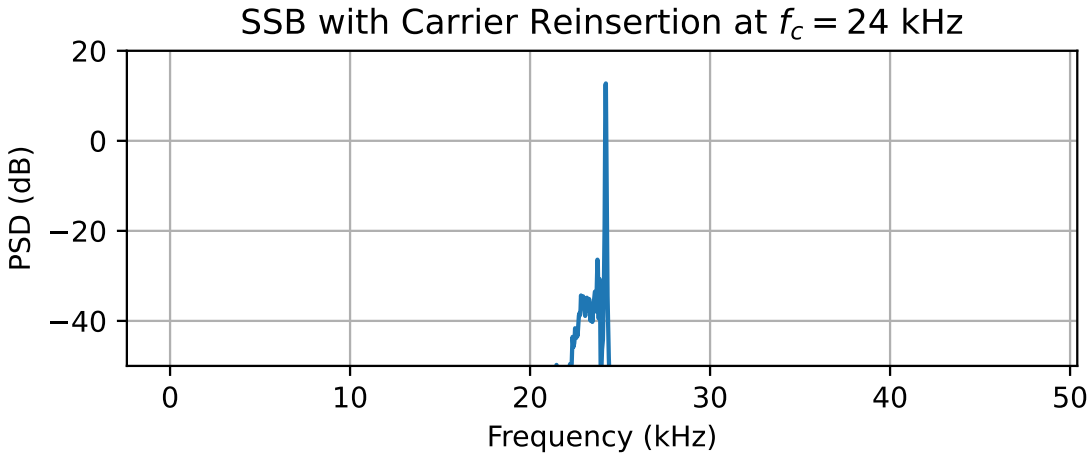


Carrier Reinsertion with Amplitude K

In [194…
```
K = 10
e = x_c+K*cos(2*pi*24200*t_up)
figure(figsize=(6,2))
plot(t_up[800:1400]*1000,e[800:1400])
xlim([9,12])
title(r'SSB with Carrier Reinsertion')
xlabel(r'Time (ms)')
ylabel(r'$e(t)$')
grid();
figure(figsize=(6,2))
P_e, f_e = ss.psd(e,2**12,fs=fs_up,scale_noise=False)
plot(f_e,10*log10(P_e))
ylim([-50,20])
```

```
title(r'SSB with Carrier Reinsertion at $f_c = 24$ kHz')
ylabel(r'PSD (dB)')
xlabel(r'Frequency (kHz)')
grid();
```

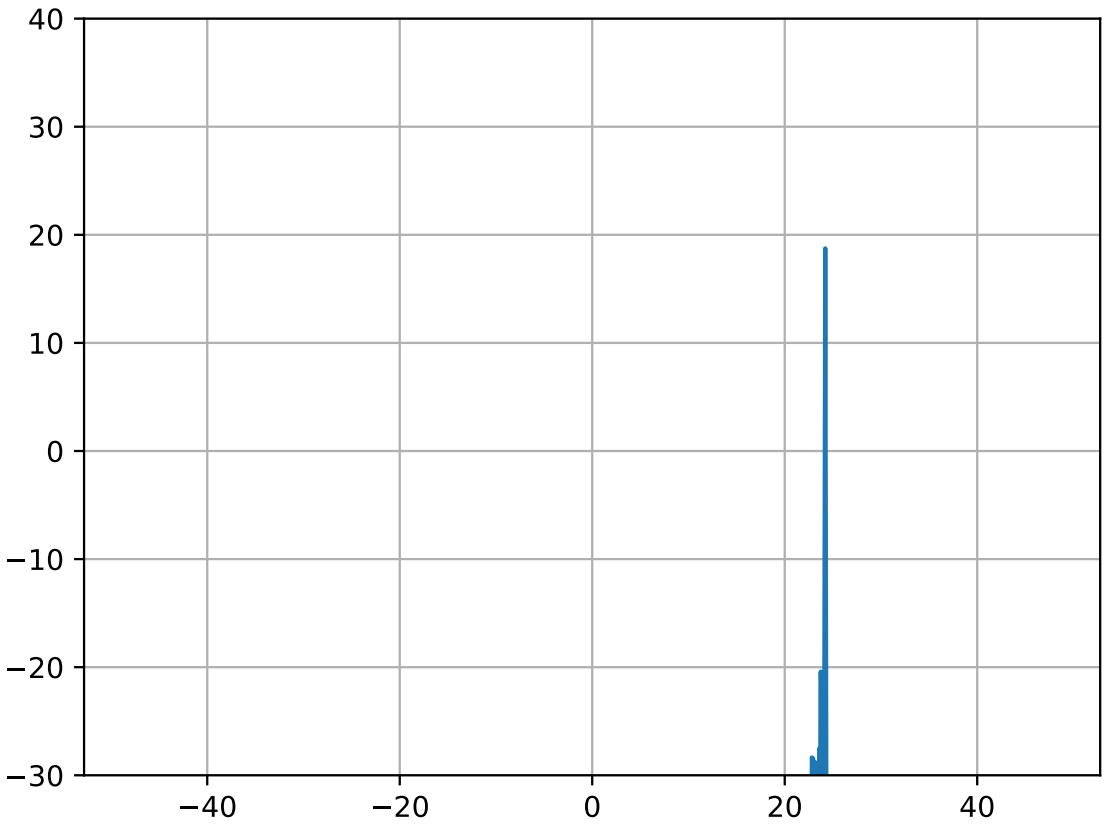




Form the analytic signal:

In [195…

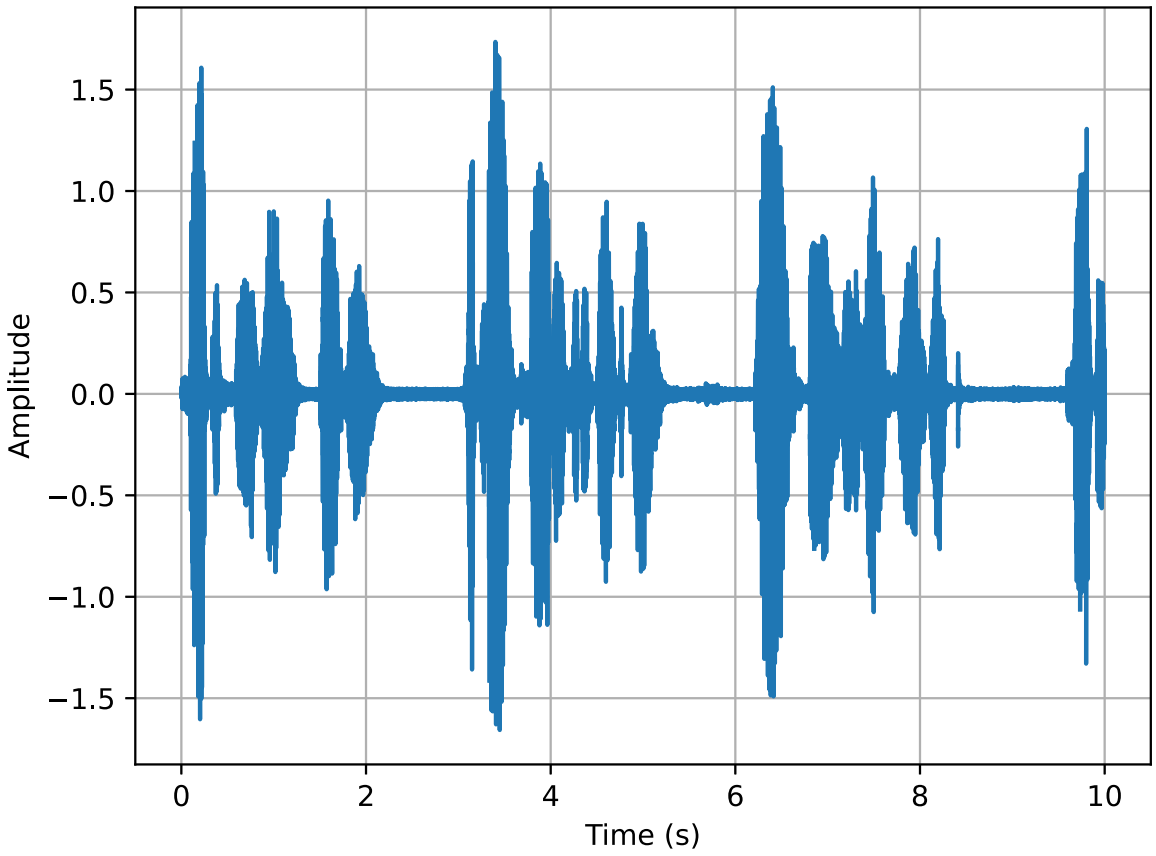
```
e_plus = analytic_signal(e,b121,'upper')
P_e_plus, f_e_plus = ss.psd(e_plus,2**12,fs=fs_up,scale_noise=False);
plot(f_e_plus,10*log10(P_e_plus))
ylim([-30,40])
grid();
```



Take the absolute value and subtract the signal mean to remove the DC bias:

In [196…

```
e_plus_env = abs(e_plus)
yD = e_plus_env[100:]
plot(t_up[100:],yD-mean(yD))
ylabel(r'Amplitude')
xlabel(r'Time (s)')
grid();
```



In [197…
```
ydd = by12.dn(yD - mean(yD))
ss.to_wav('ydd.wav',8000,ydd/max(abs(ydd)))
Audio('ydd.wav')
```

Out[197… 00:00 00:09

Frequency Translation and Mixing

In this section we consider the use spectral analysis to access frequency translation and mixing. A simple DSB carrier with multiple sinusoidal tones serving as the message signal. The sampling rate is fixed at 200 kHz.

When a carrier-based signal at f_c is multiplied by a local oscillator (LO) signal $\cos(2\pi f_{\text{LO}} t)$ the result is two signals, one centered at $f_c + f_{\text{LO}}$ and another at $|f_c - f_{\text{LO}}$. If say $x_c(t)$ is a DSB signal, $m(t)\cos(2\pi f_c t)$, then the mixer output is

$$e(t) = x_{\text{mix}}(t) = m(t)\frac{1}{2}\left\{\cos\left[2\pi(f_c + f_{\text{LO}})t\right] + \cos\left[2\pi(f_c - f_{\text{LO}})t\right]\right\} \quad ($$

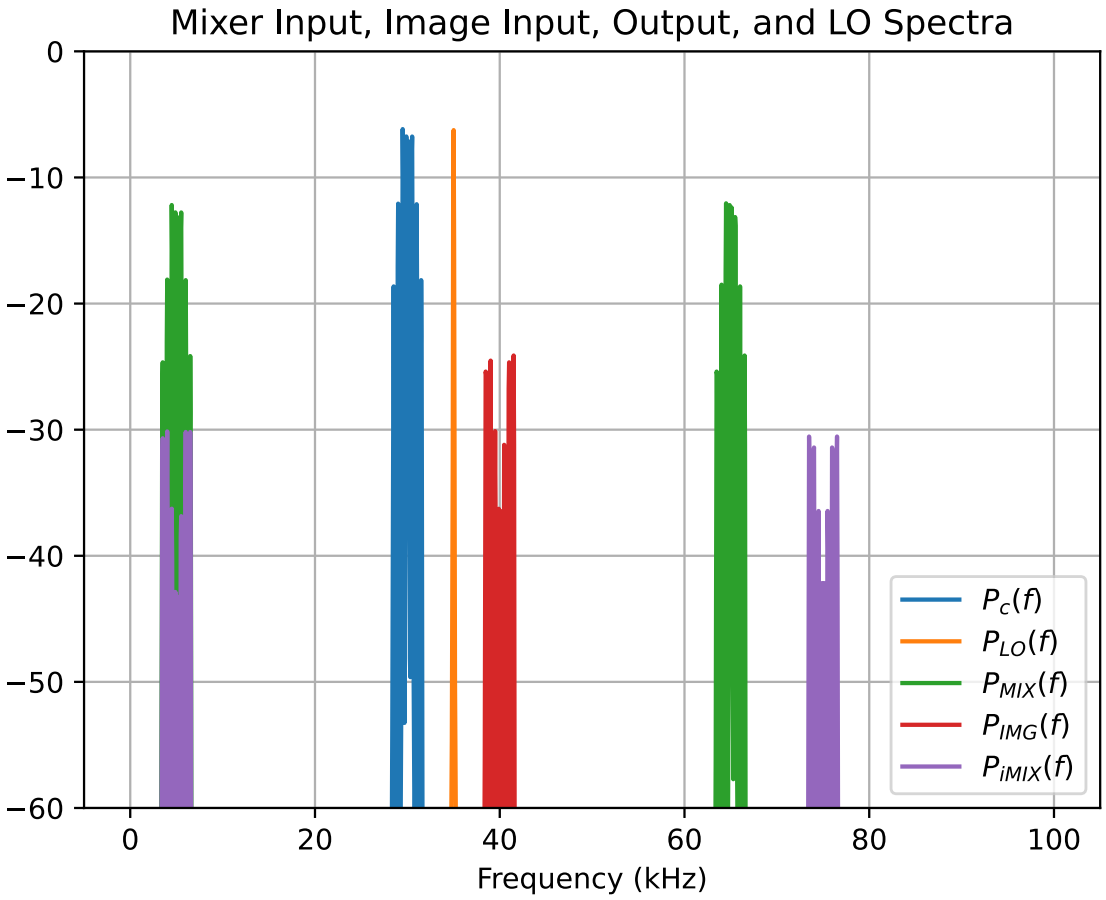
In general we may choose to make the sum frequency or the difference frequency the desired output.

- When the output frequency is lower than the input frequency, the difference frequency is of interest. In a superheterodyne receiver the output frequency is also known as the *intermediate* frequency or $f_{\rm IF}$
- With *highside* tuning we choose the LO frequency to above the input frequency, i.e., $f_{\rm LO}$ to be $f_c + f_{\rm IF}$. For $f_{\rm LO} = f_c + f_{\rm IF}$ the *image frequency*, which also down-converts to $f_{\rm IF}$, is at $f_{\rm IMAGE} = f_c + 2f_{\rm IF} = f_{\rm LO} + f_{\rm IF}$
- Why is this called the image? Well this frequency also converts to the same IF, as $f_{\rm IMAGE} - f_{\rm LO} = f_{\rm IF}$
  - In the example below $f_c = 30$ kHz and $f_{\rm IF} = 5$ kHz
  - We initially set $f_{\rm LO} = 35$ kHz. This puts the image frequency at $f_{\rm IMAGE} = 30 + 2 \times 5 = 40$ kHz.
- When the output frequency is higher than the input frequency either the sum frequency or the difference frequency may be used to obtain $f_{\rm IF}$
  - Suppose $4F_{\rm IF} = 50$ kHz, then we may choose $f_{\rm LO} = 50 - 30 = 20$ kHz or $f_{\rm LO} = 50 + 30 = 80$ kHz

In [128…
```
# Create a message signal as a collection of tones:
m,t = M_sinusoids(10000,[100,0.5e3,1.0e3,1.5e3],[2,2,1,0.5],
                  [0,0,0,0],200000)
# Create a image signal as a collection of tones:
mi,t = M_sinusoids(10000,[100,0.5e3,1.0e3,1.5e3],[0.5,1,2,2],
                  [0,0,0,0],200000)
# Choose a carrier frequency for the message and DSB modulate
fc = 30e3
xc = m*cos(2*pi*fc*t)
# Choose a carrier frequency for the image and DSB modulate
fi = (30+2*5)*1e3
xIMAGE = 0.125*mi*cos(2*pi*fi*t)
# Choose an LO and generate the LO signal
fLO = 35e3
xLO = cos(2*pi*fLO*t)
# Mix the siganl and the image with the same LO and see what happens
xMIX = xc*xLO
xiMIX = xIMAGE*xLO
```

In [140…
```
P_xc, f_mix = ss.psd(xc,2**12,200,scale_noise=False);
plot(f_mix,10*log10(P_xc))
P_xLO, f_mix = ss.psd(xLO,2**12,200,scale_noise=False);
plot(f_mix,10*log10(P_xLO))
P_xMIX, f_mix = ss.psd(xMIX,2**12,200,scale_noise=False);
plot(f_mix,10*log10(P_xMIX))
P_xIMAGE, f_mix = ss.psd(xIMAGE,2**12,200,scale_noise=False);
plot(f_mix,10*log10(P_xIMAGE))
```

```
P_xiMIX, f_mix = ss.psd(xiMIX,2**12,200,scale_noise=False);
plot(f_mix,10*log10(P_xiMIX))
title(r'Mixer Input, Image Input, Output, and LO Spectra')
legend((r'$P_c(f)$',r'$P_{LO}(f)$',r'$P_{MIX}(f)$',
        r'$P_{IMG}(f)$',r'$P_{iMIX}(f)$'),
        loc='lower right')
xlabel(r'Frequency (kHz)')
ylim([-60,0])
grid();
```



Interference with Envelope Detection

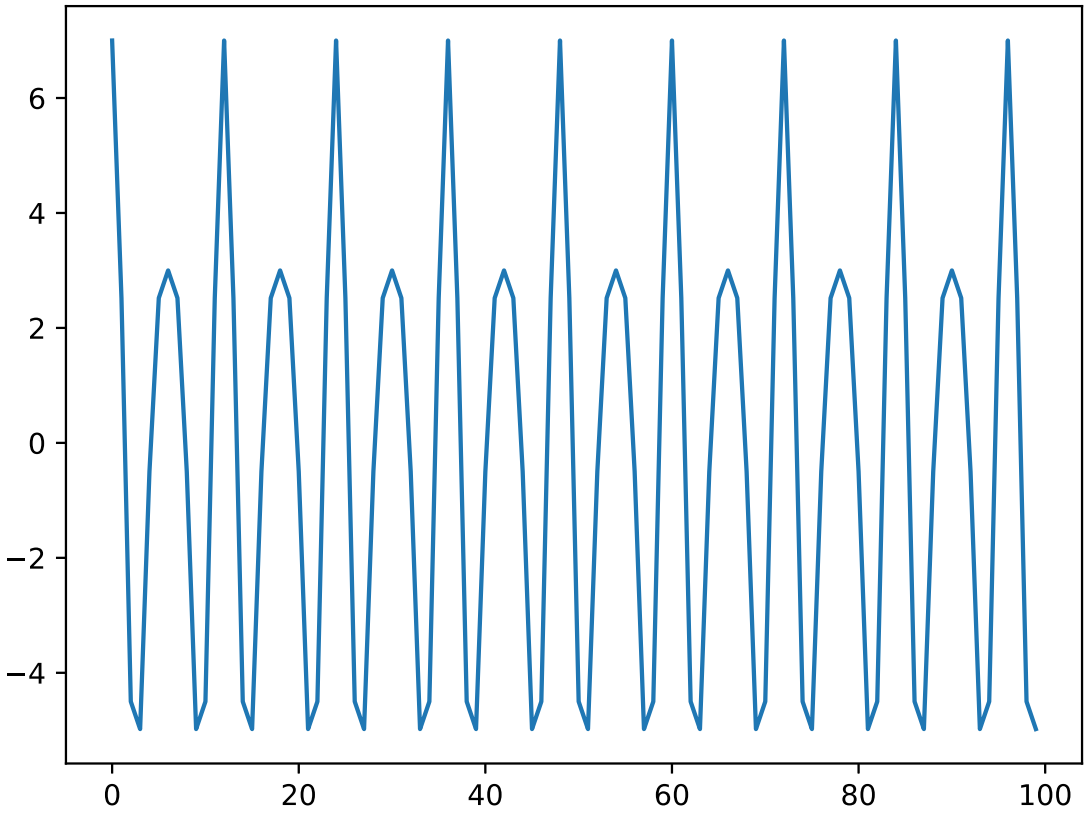
In [145…
```
t_int = arange(0,5,1/fs_up)
```

In [155…
```
A = 100
B = 2
xI = A + 5*cos(2*pi*10e3*t_int) + B*cos(2*pi*15e3*t_int)
xQ = B*sin(2*pi*15e3*t_int)
R = abs(xI + 1j*xQ)
```

In [156…
```
# P_int, f_int = ss.psd(R,2**12,200); # fs = 200
# ylim([-60,30])
plot(R[:100] - A)
```

```
# ylim([0, 1.2*A])
```

Out[156… `[<matplotlib.lines.Line2D at 0x169132210>]`



Bandpass Sampling Theory

Create a function for plotting the spectra of sampled bandpass signal. Assume the bandpass spectral shape is $X(f)$, the sampled spectrum is of the form (ignoring gain scaling by f_s) is

$$X_s(f) = \sum_{n=-\infty}^{\infty} X(f - nf_s) \tag{21}$$

In particular here $X(f)$ is a right triangle shape starting at f_1 and stopping at f_2, where $f_2 > f_1$. The tuple variable `B = (f1, f2)` holds these two values, which to $(2, 3)$.

In [122…

```
def BP_spec(f,fs,Ntranslates=10,B=(2,3)):
    """
    Sampled bandpass signal spectrum plot
              f = frequency axis created using arange
             fs = sampling frequency
    Ntranslates = the number of spectrum translates
                  to use in the calculation
```

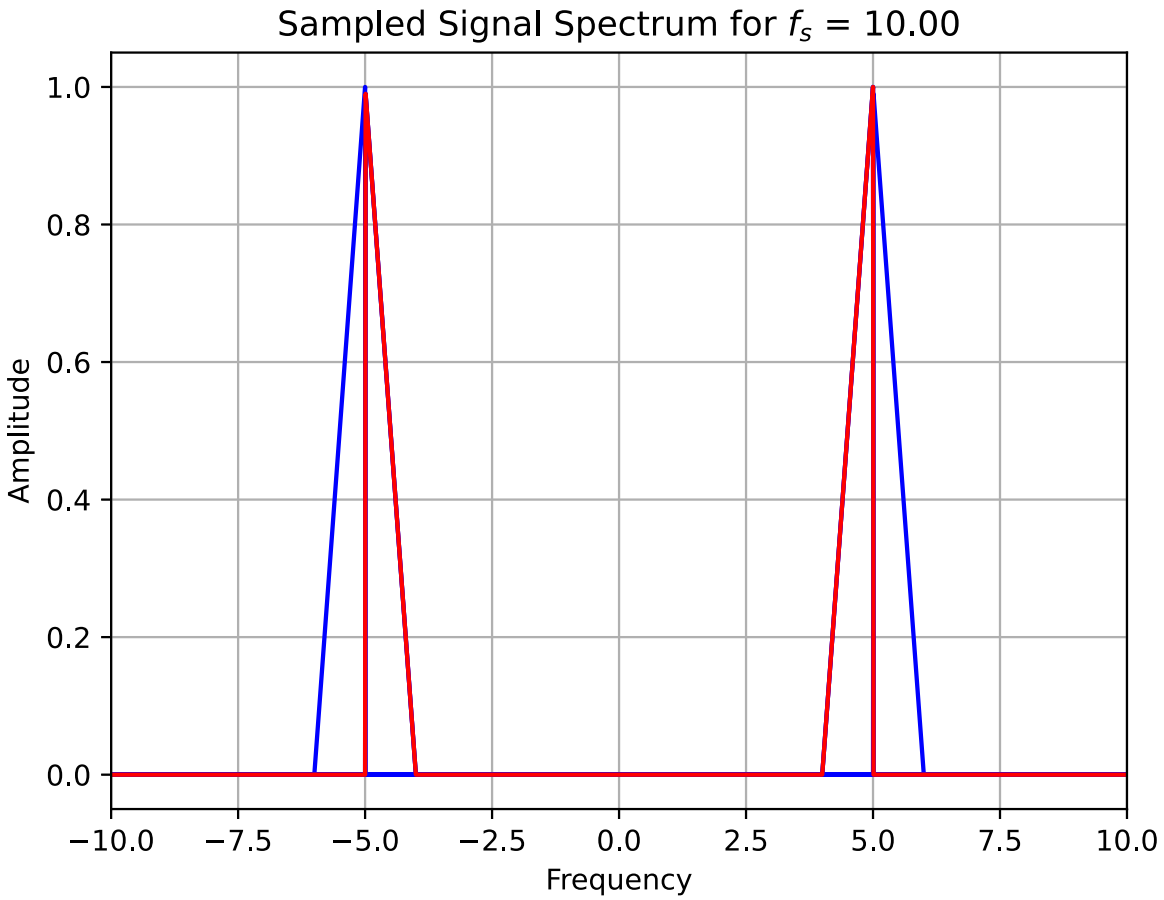
```
            B = (B_lower,B_upper)

    Mark Wickert March 2015
    """
    W = B[1]-B[0]
    X = BP_primP(f-B[0],W)+BP_primN(f+B[0],W)
    for k in range(Ntranslates):
        plot(f,BP_primP(f-B[0]-k*fs,W)+BP_primN(f+B[0]-k*fs,W),'b')
        plot(f,BP_primP(f-B[0]+k*fs,W)+BP_primN(f+B[0]+k*fs,W),'b')
    plot(f,X,'r')
    title(r'Sampled Signal Spectrum for $f_s$ = %2.2f' % fs)
    ylabel(r'Amplitude')
    xlabel(r'Frequency')
    xlim([-2*B[1],2*B[1]])
    grid();

def BP_primP(f,W=1):
    return ss.tri(f-W,W)*ss.rect(f-W/2,W)

def BP_primN(f,W=1):
    return ss.tri(f+W,W)*ss.rect(f+W/2,W)
```

To use the function above (plus two helper functions), you just enter in a frequency axis using arange, enter a sampling rate, the number of spectral translates to display, and a Python tuple containing the lower and upper bandedge of the bandpass signal spectrum being sampled.

- The value for the number of translates should be kept relatively small, so as not to require a lot of graphics plotting time

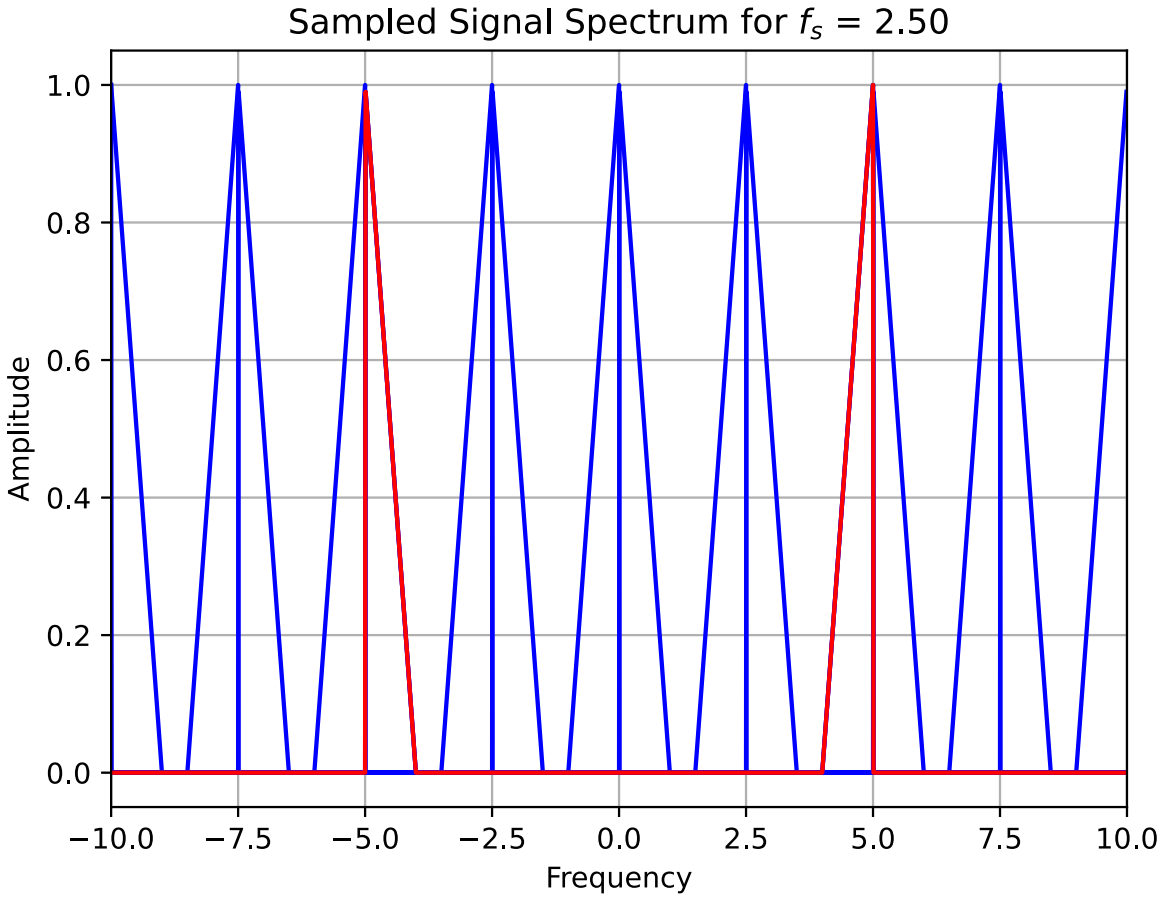
In [123…

```
f = arange(-10,10,.01)
BP_spec(f,10.0,8,(4,5))
```



In [124…

```
f = arange(-10,10,.01)
BP_spec(f,2.5,8,(4,5))
```

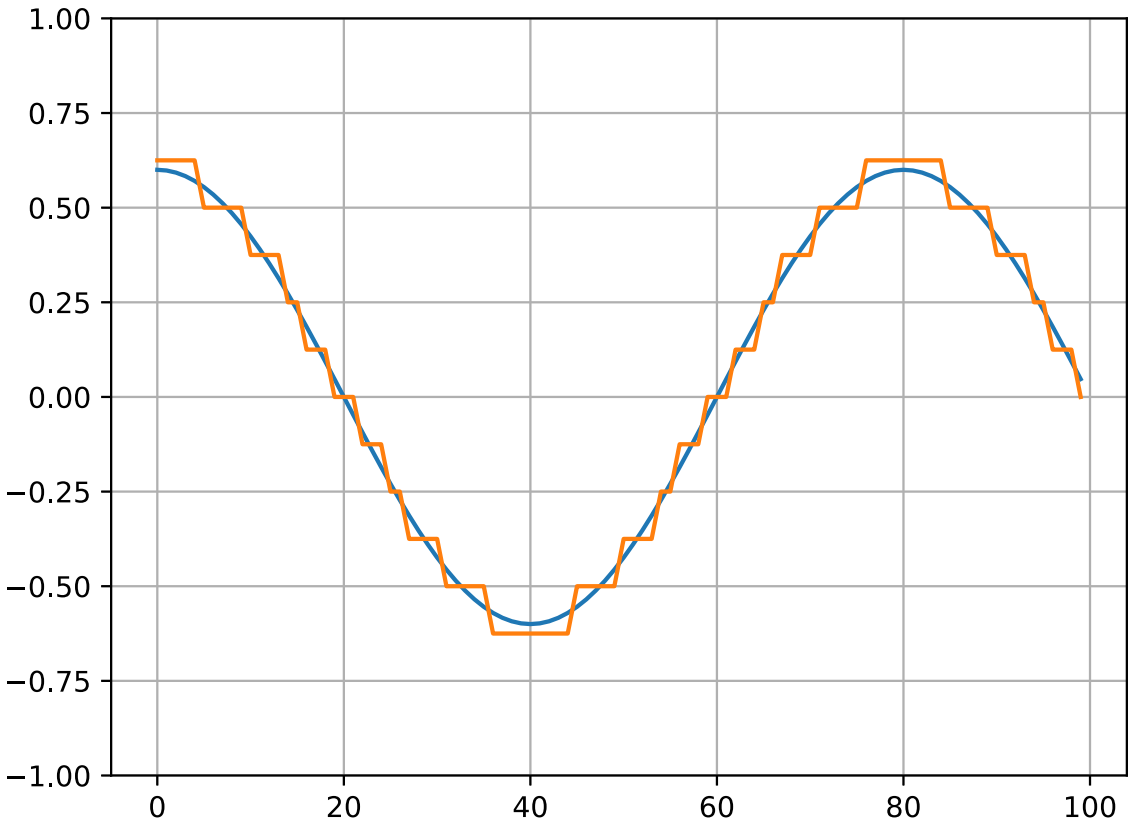


Pulse Code Modulation

Test With a Sinusoid

In [104…

```
N_bits = 4
EbN0_dB = 50
n = arange(0,100)
x = 0.6*cos(2*pi*n*100/8000)
xe = dc.pcm_encode(x,N_bits) # 8 bits per sample
xer = dc.awgn_channel(xe,EbN0_dB)
x_hat = dc.pcm_decode(xer,N_bits)
plot(n,x)
plot(n,x_hat)
ylim([-1,1])
grid();
```



Test with a Speech File

In the past Project 2 has involved PCM encoding and decoding speech with channel impairment.

In [105…
```
N = 120000; # number of speech samples to process
fs,m1 = ss.from_wav('OSR_uk_000_0050_8k.wav')
m1 = m1[20000:N]
```

In [118…
```
EbN0_dB = 20
xe = dc.pcm_encode(m1,8) # 8 bits per sample
xer = dc.awgn_channel(xe,EbN0_dB)
m1_hat = dc.pcm_decode(xer,8)
```

In [119…
```
total = len(xe)
errors = sum(abs(int16(xer[:]-xe[:])))
Pe = errors/total
print('Total Bits = %d' % total)
print('Bit Errors = %d' % errors)
print('Bit Error Probability (est) = %1.2e' % (errors/total,))
```

```
Total Bits = 800000
Bit Errors = 0
Bit Error Probability (est) = 0.00e+00
```

In [120…
```
y_demod = m1_hat
```

```
ss.to_wav('yp2.wav',8000,y_demod)
Audio('yp2.wav')
```

Out[120… 00:00 00:12

In [121…

```
plot(m1[2000:4000])
plot(m1_hat[2000:4000])
grid();
```