# Chapter 4 Jupyter Notebook

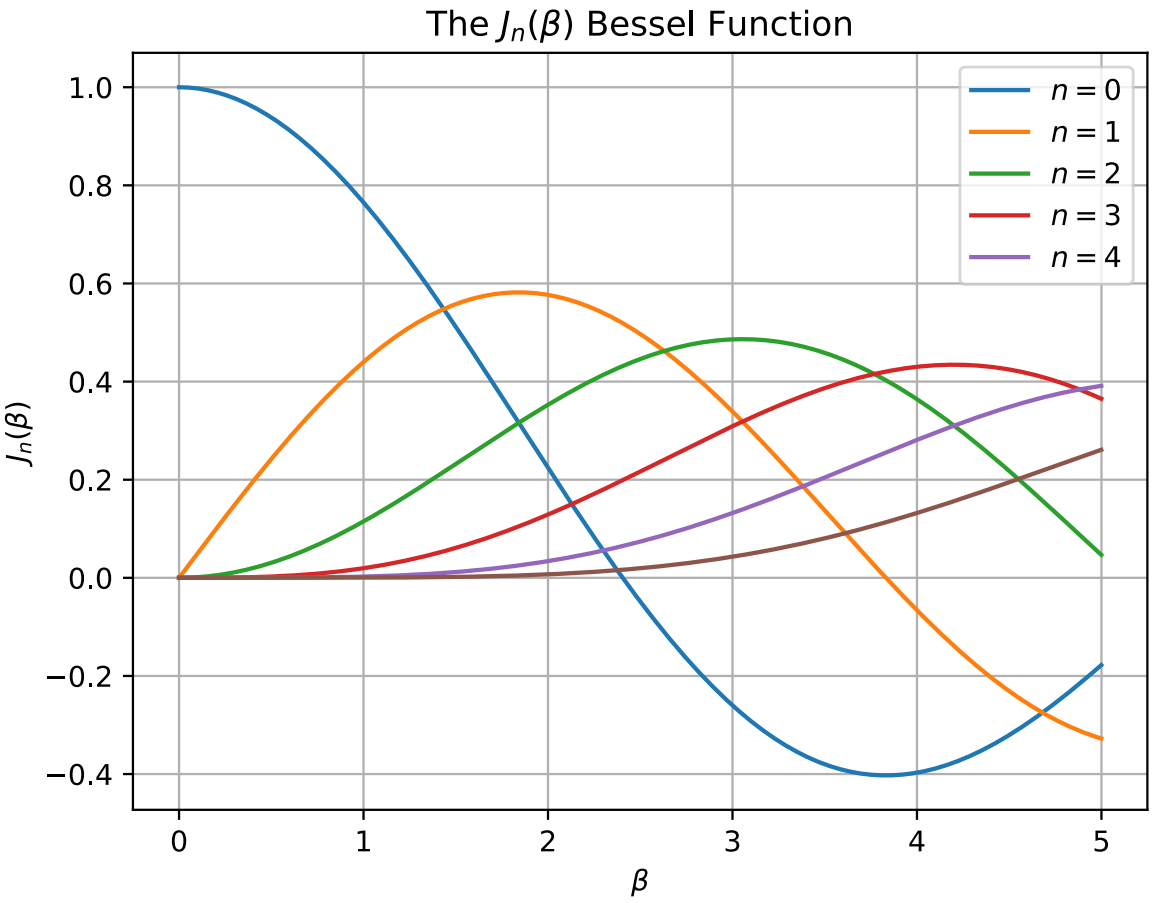
In [1]:
```
# %pylab inline
from numpy import * # better in import numpy as np
from matplotlib.pyplot import * # better to import matplotlib.pyplot a
import sk_dsp_comm.sigsys as ss
import sk_dsp_comm.synchronization as pll
import sk_dsp_comm.digitalcom as dc
import scipy.signal as signal
from IPython.display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

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

# Sinusoidal Angle Modulation Spectra

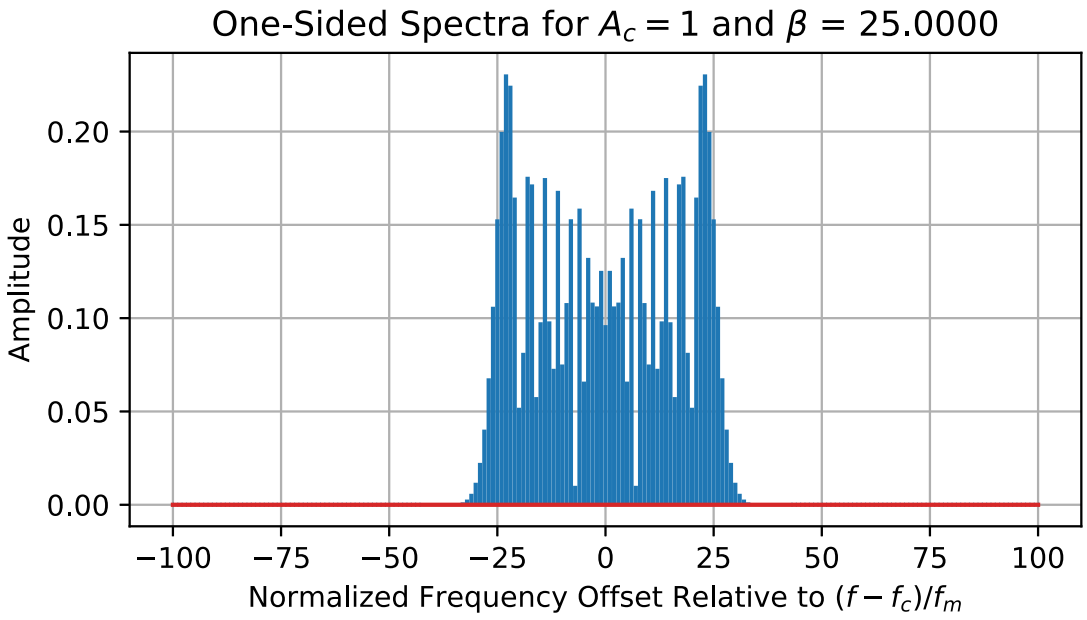
In [2]:
```
import scipy.special as special
```

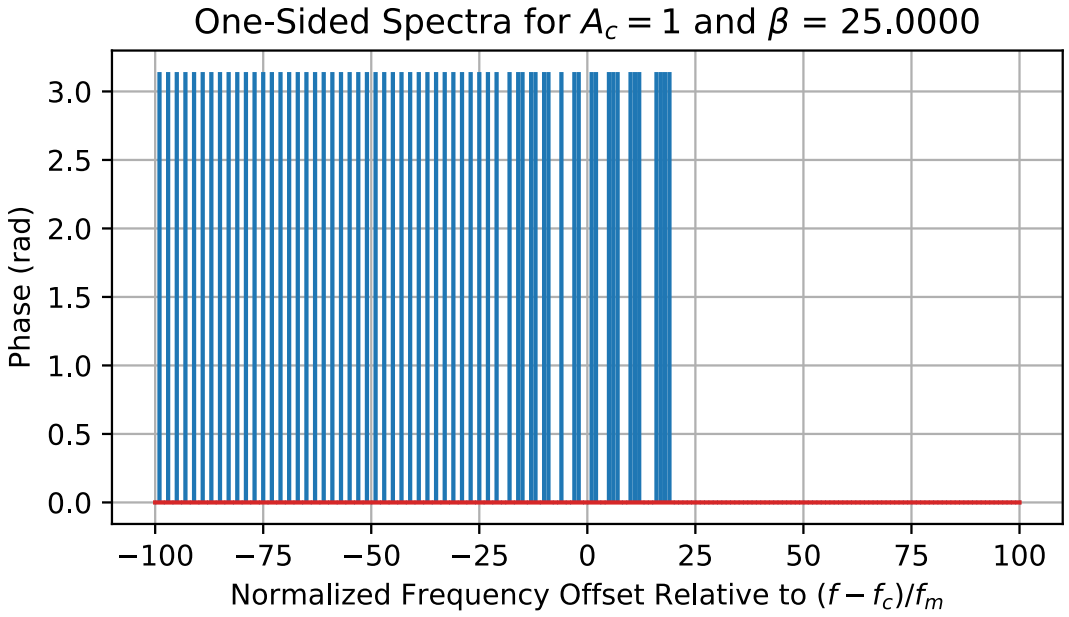
In [3]:
```
xj = arange(0,5,.001)
for n in range(6):
    plot(xj,special.jn(n,xj))
title(r'The $J_n(\beta)$ Bessel Function')
ylabel(r'$J_n(\beta)$')
xlabel(r'$\beta$')
legend((r'$n=0$',r'$n=1$',r'$n=2$',r'$n=3$',r'$n=4$'),loc='best')
grid();
```




# Plot Spectra

In [4]:
```
#beta = 2.4048
beta = 25
Nterms = 100
idx = arange(-Nterms,Nterms+1)
Yn = special.jn(idx,beta)
figure(figsize=(6,3))
stem(idx,abs(Yn),markerfmt=" ")
grid();
ylabel(r'Amplitude')
xlabel(r'Normalized Frequency Offset Relative to $(f-f_c)/f_m$')
title(r'One-Sided Spectra for $A_c=1$ and $\beta$ = %2.4f' % beta);
figure(figsize=(6,3))
stem(idx,angle(Yn),markerfmt=" ")
grid();
ylabel(r'Phase (rad)')
xlabel(r'Normalized Frequency Offset Relative to $(f-f_c)/f_m$')
title(r'One-Sided Spectra for $A_c=1$ and $\beta$ = %2.4f' % beta);
```




# Bandwidth Calculation

In [6]:
```
beta = 75
n = arange(0,100)
J_beta15=special.jn(n,beta)
```

Find the power in 11 sidebands total centered onthe carrier. The given signal power is $A_c^2/2 = 5000$ W. We must find:

$$P_{\text{out}} = \frac{A_c^2}{2}\left[J_0^2(\beta) + 2\sum_{n=1}^{5} J_n^2(\beta)\right] \tag{1}$$

for $A_c^2/2 = 5000$W and $\beta = 75$.

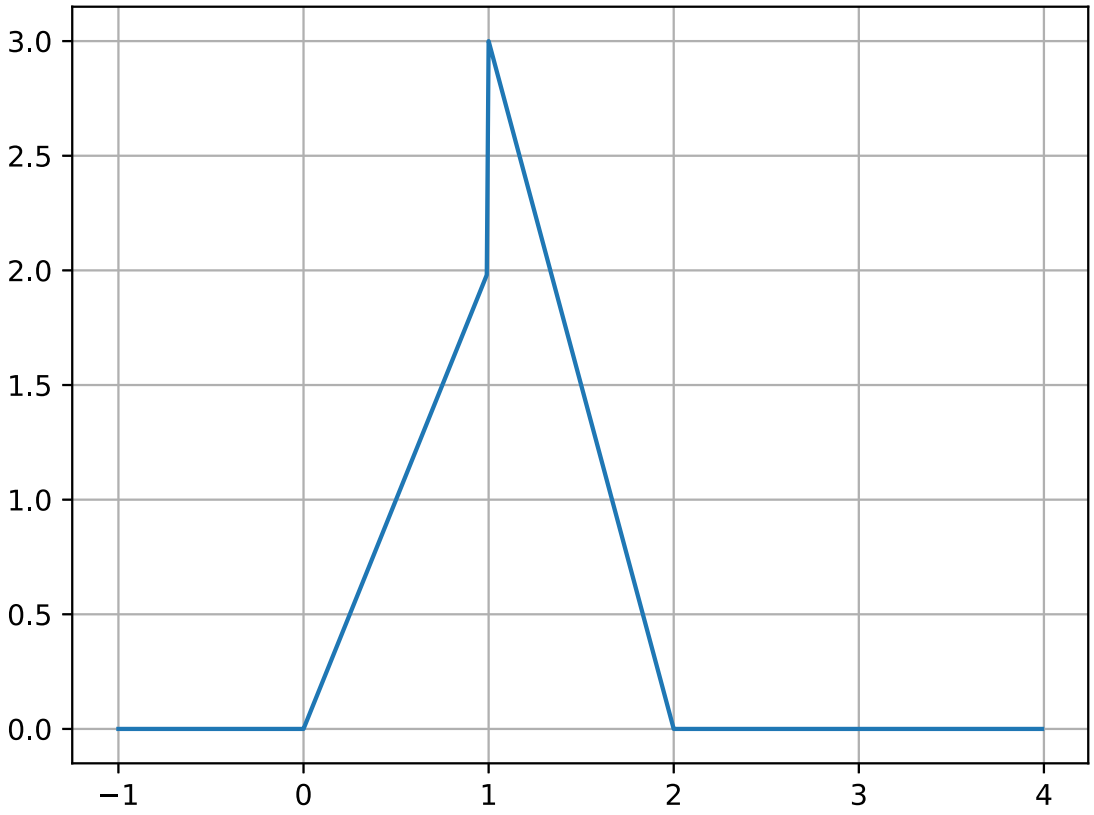
In [7]:
```
Pout = 5000*(J_beta15[0]**2 + 2*sum(J_beta15[1:6]**2))
print('Power contained in 11 Hz bandwidth = %4.4f W.' % Pout)
```

```
Power contained in 11 Hz bandwidth = 241.9323 W.
```

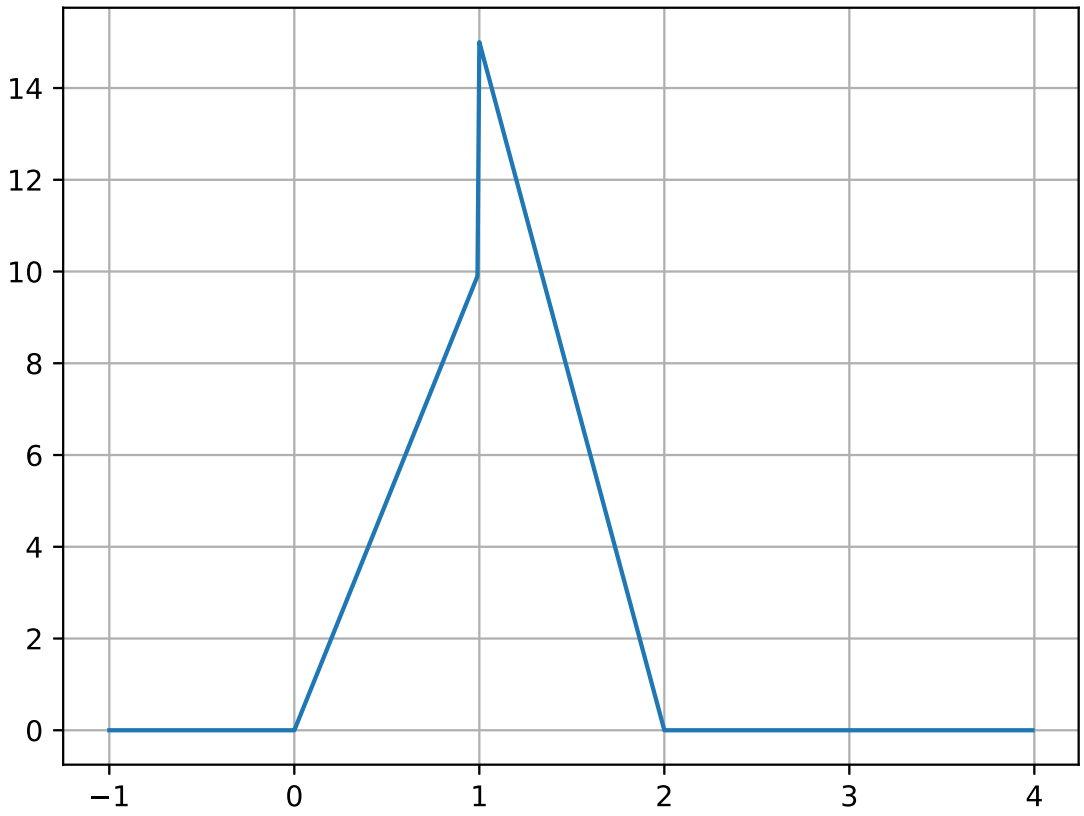
# Text Problem 13 Comments

Axis labels, a grid, and titles are missing, but expected.

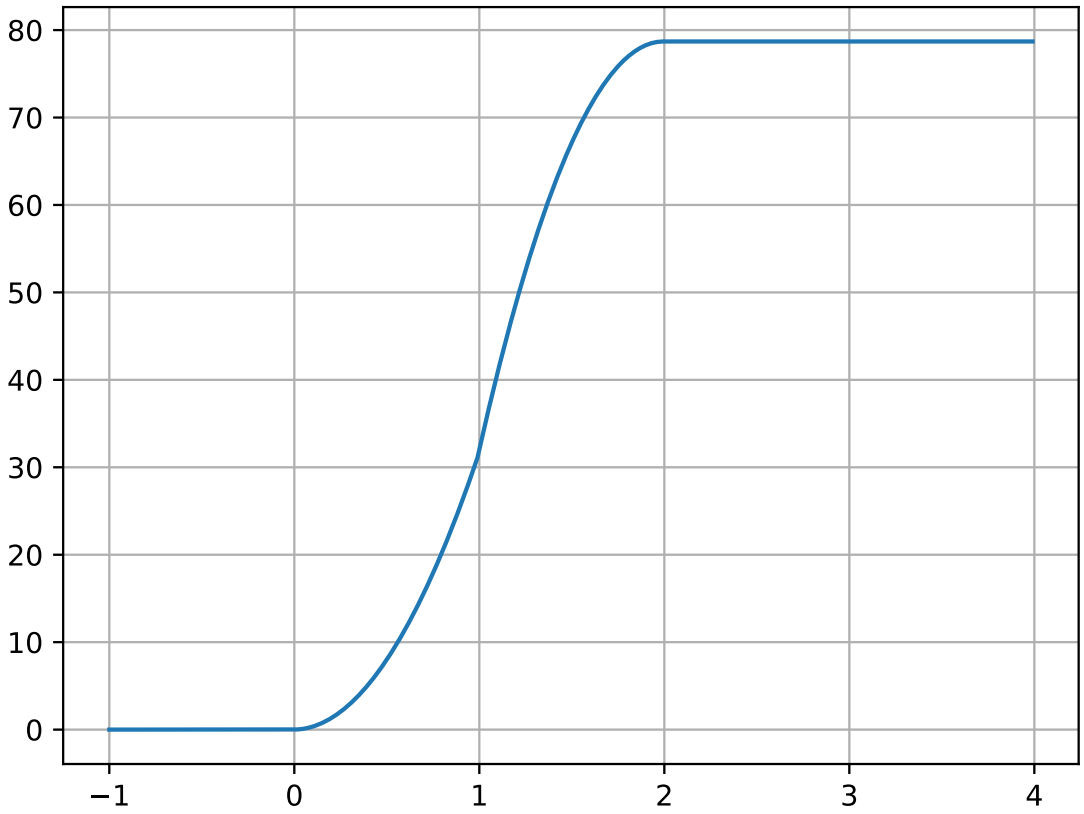
In [10]:
```
dt = 0.01
t = arange(-1,4,dt)
m = 2*t*(ss.step(t)-ss.step(t-1)) + (6-3*t)*(ss.step(t-1)-ss.step(t-2))
plot(t,m)
grid();
```




In [12]:
```
plot(t,5*m)
grid();
```




In [13]:
```
plot(t,2*pi*5*cumsum(m)*dt)
grid();
```



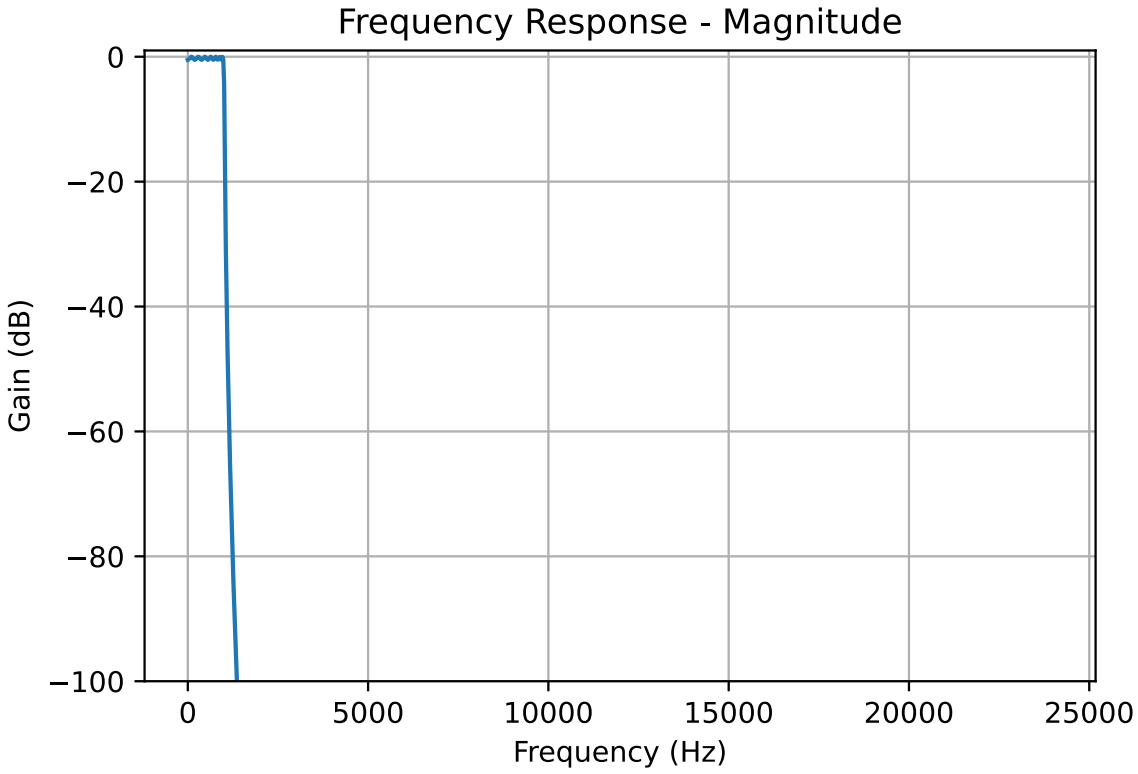
# Bandlimited Noise PM and FM

Following notes Example 4.6 we set up the following simulation:

In [14]:
```
import sk_dsp_comm.iir_design_helper as iir_h
```

In [15]:
```
fs = 48000
b, a, sos = iir_h.IIR_lpf(1000,1200,0.5,70,fs,'cheby1')
```

In [16]:
```
iir_h.freqz_resp_cas_list([sos],'dB',fs)
ylim([-100,1])
grid();
```




## Model Details

Assuming a phase state of zero at $t = 0$, an FM signal for $t \geq 0$ takes the form

$$x_c(t) = A_c \cos\left[2\pi f_c t + 2\pi f_d \int_0^t m(\alpha)\, d\alpha\right] \tag{2}$$

$$= A_c \mathrm{Re}\left\{\exp\left[j2\pi f_c t + j2\pi f_d \int_0^t m(\alpha)\, d\alpha\right]\right\} \tag{3}$$

$$= A_c \mathrm{Re}\left\{\exp[j2\pi f_c t] \cdot \exp\left[j2\pi f_d \int_0^t m(\alpha)\, d\alpha\right]\right\} \tag{4}$$

Consider just the complex envelope

$$\tilde{x_c}(t) = A_c \exp\left[j2\pi f_d \int_0^t m(\alpha)\, d\alpha\right]$$

In the discrete-time domain we have

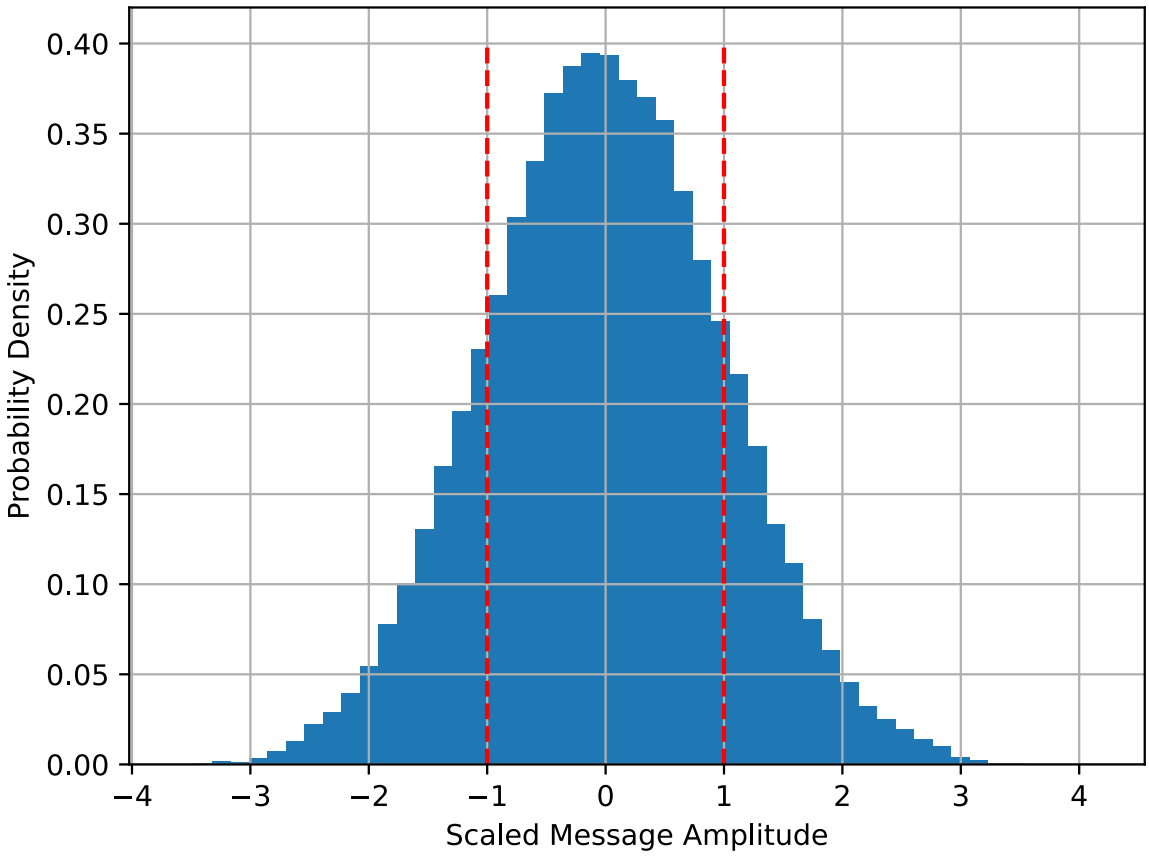
$$\tilde{x_c}[n] = A_c \exp\left[j2\pi f_d \sum_{k=0}^{n} m[k]\, \frac{1}{f_s}\right]$$

In Python implement the modulator as follows:

```
Ac = 1
fs = some value in Hz
fd = some value in Hz
m = message signal array
xc_bb = Ac*exp(1j*2*pi*fs*cumsum(m)*1/fs) # complex
envelope/complex baseband
```

In [17]:
```
# Filter white noise
w = random.randn(100000)
m = signal.sosfilt(sos,w)
# Scale the noise to establish a peak deviation for the message signal
#xs = x/max(abs(x)) # too conservative
ms = m/std(m) # seems reasonable
fd = 1000 #Hz; Modulator deviation constant
#xc = exp(1j*kp*ms) # For PM
xc = exp(2*pi*1j*fd*cumsum(ms)*1/fs) # For FM
```

In [18]:
```
hist(ms,50,density=True);
plot([-1,-1],[0,.4],'r--')
plot([1,1],[0,.4],'r--')
xlabel(r'Scaled Message Amplitude')
ylabel(r'Probability Density');
grid();
```



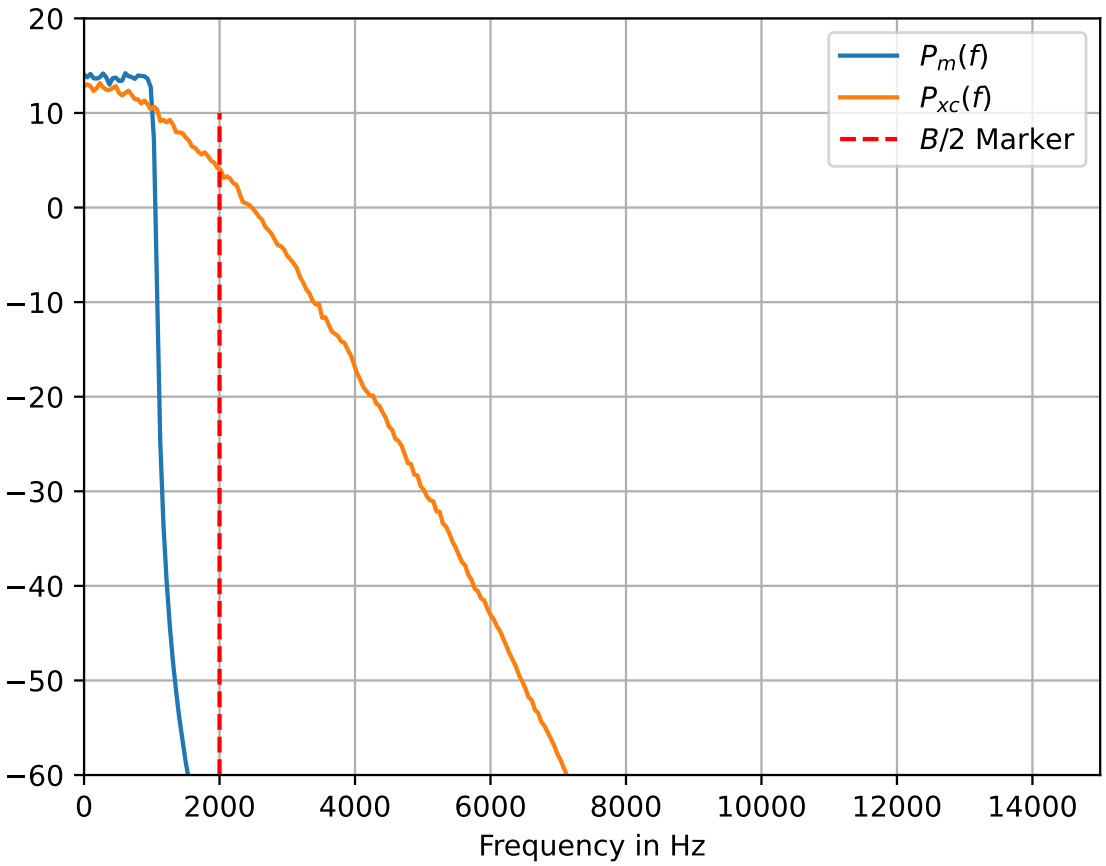

## Using the Given $f_d$ Approximate $\Delta f$ and Find the Bandwidth

In [20]:
```
# Approximate single-sided bandwidth using Carson's Rule
W = 1000
Df = fd*1 # Model peak deviation assuming 1 sigma for message peak
B = 2*(W + Df)
print('B/2 = %7.2f Hz' % (B/2,))
```

```
B/2 = 2000.00 Hz
```

## Plot the Message PSD and the FM Modulation PSD (1-sided)

In [21]:
```
Pms,fms = ss.psd(ms,2**10,fs,scale_noise=True);
plot(fms,10*log10(Pms))
ylim([-60,20])
Pxc,fxc = ss.psd(xc,2**10,fs,scale_noise=True)
plot(fxc,10*log10(Pxc))
plot([B/2,B/2],[-60,10],'r--')
xlabel(r'Frequency in Hz')
xlim([0,fs/2]);
legend((r'$P_m(f)$',r'$P_{xc}(f)$',r'$B/2$ Marker'),loc='best')
xlim(0,15000)
grid();
```




# Sinusoidal FM Using the FM Modulator Model Developed for Noise

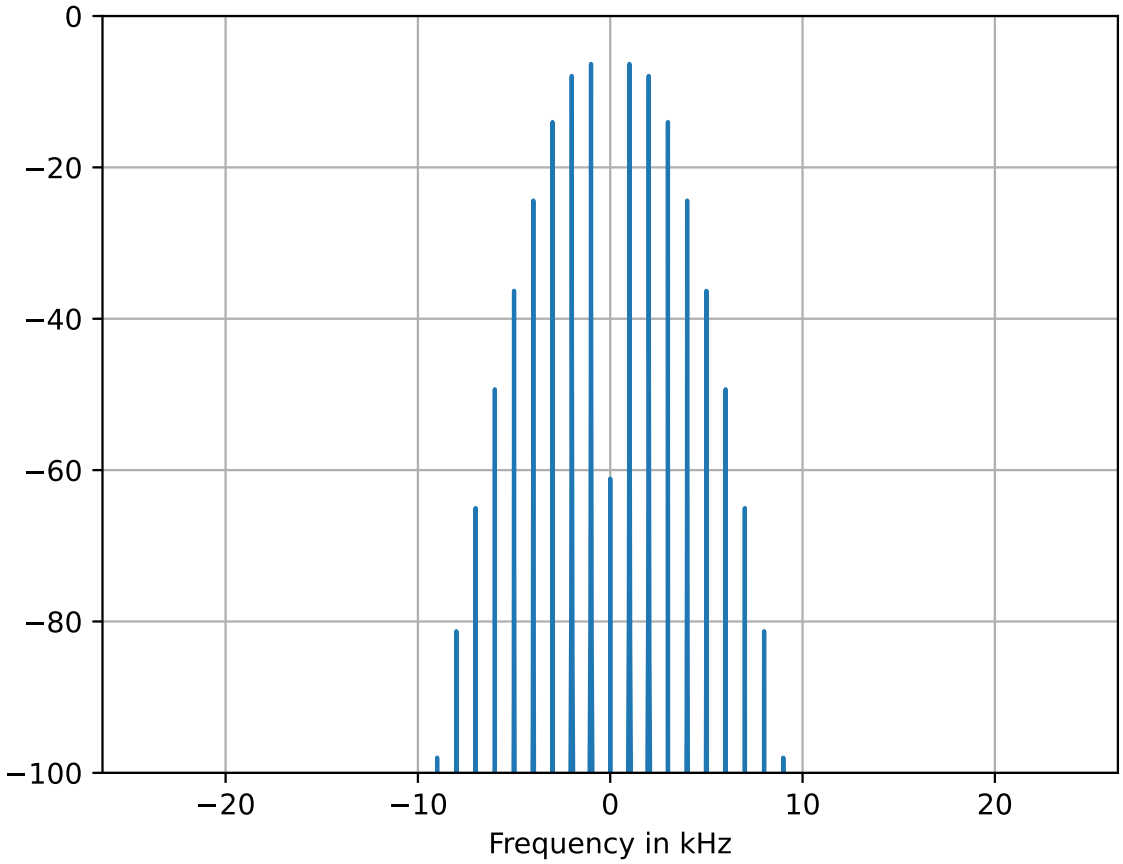
In [22]:
```
# A sinusoidal message (one or more tones)
fs = 48000
n = arange(100000)
fd = 1000 #Hz; Modulator deviation constant
fm1 = 1000; fm2 = 100
m = 2.4048*cos(2*pi*fm1/fs*n) # + 2*cos(2*pi*fm2/fs*n)
#xc = exp(1j*kp*ms) # For PM
xc = exp(2*pi*1j*fd*cumsum(m)*1/fs) # For FM this owrks for any signal
# xc = exp(1j*m) # For sinusoidal angle mod this is an exact modulator
```

- Recall that for sinusoidal FM

$$\beta = A\frac{f_d}{f_m}$$

In [23]:
```
Pxc, fm = ss.psd(xc,2**14,fs,scale_noise=False)
plot(fm/1e3,10*log10(Pxc))
xlabel(r'Frequency in kHz')
ylim([-100, 0])
# xlim([-5,5])
```

```
grid();
```




# Demodulation of Angle Modulation

## Complex Baseband Discriminator

In [24]:
```
# This function can be found in
# sk_dsp_comm.rtlsdr_helper
def discrim(x):
    """
    function disdata = discrim(x)
    where x is an angle modulated signal in complex baseband form.

    Mark Wickert
    """
    X=np.real(x)        # X is the real part of the received signal
    Y=np.imag(x)        # Y is the imaginary part of the received sign
    b=np.array([1, -1]) # filter coefficients for discrete derivative
    a=np.array([1, 0])  # filter coefficients for discrete derivative
    derY=signal.lfilter(b,a,Y)  # derivative of Y,
    derX=signal.lfilter(b,a,X)  #    "          X,
    # normalize by the squared envelope acts as a limiter
    disdata=(X*derY-Y*derX)/(X**2+Y**2)
    return disdata
```

- To test the `discrim()` function I generate a complex baseband angle modulated signal of the form

$$x(t) = e^{j\beta\cos(2\pi f_m t)} \tag{5}$$

which is related to the real angle modulated signal $x_c(t)$ via

$$x_c(t) = \cos\left[2\pi f_c t + \beta\cos(2\pi f_m t)\right] \tag{6}$$

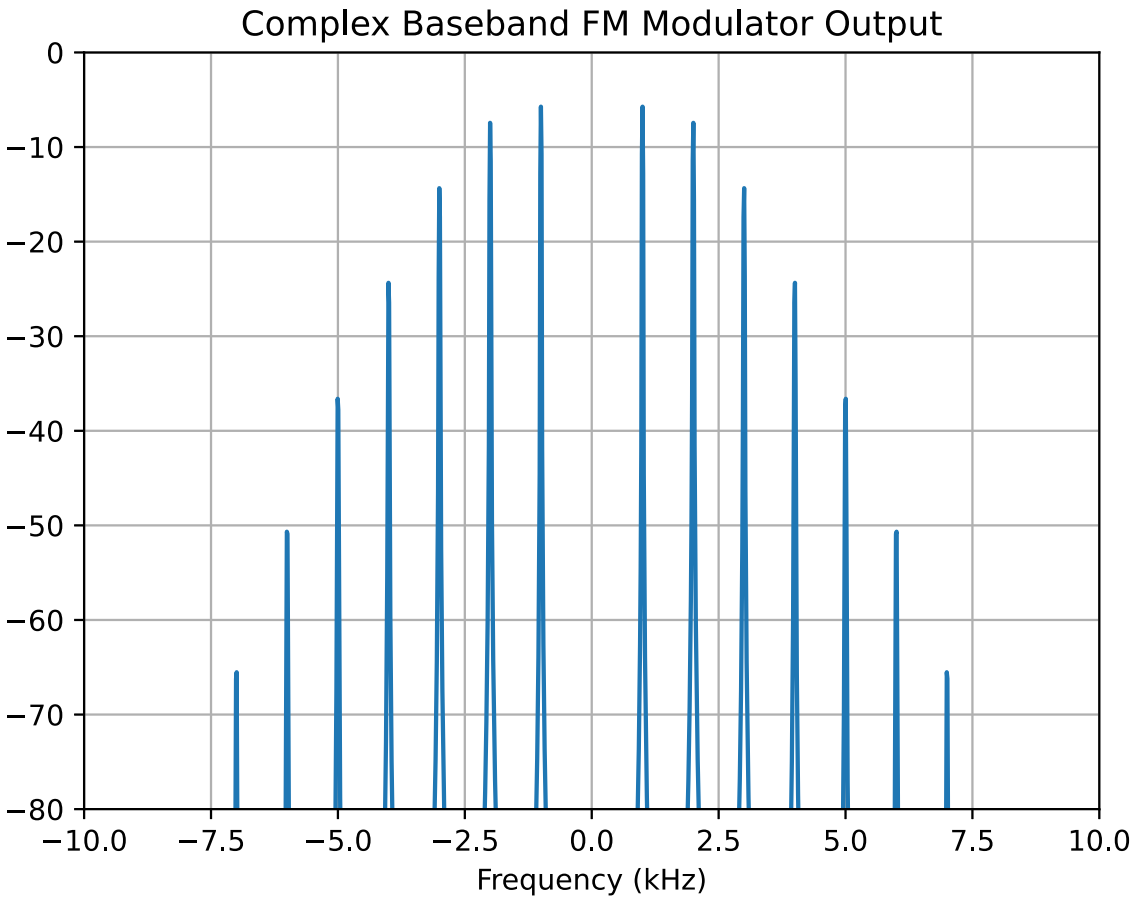
$$= \mathrm{Re}\{e^{j2\pi f_c t}\cdot \underbrace{e^{j\beta\cos(2\pi f_m t)}}_{x(t)}\}. \tag{7}$$

- The complex baseband signal is passed through the discriminator function to recover the angle modulation.

In [25]:
```
fs = 50000
fc = 0
n = arange(0,10000)
m = cos(2*pi*n*1000/fs)
xc = exp(1j*2.4048*m)*exp(1j*2*pi*fc/fs*n)
yD = discrim(xc)
```

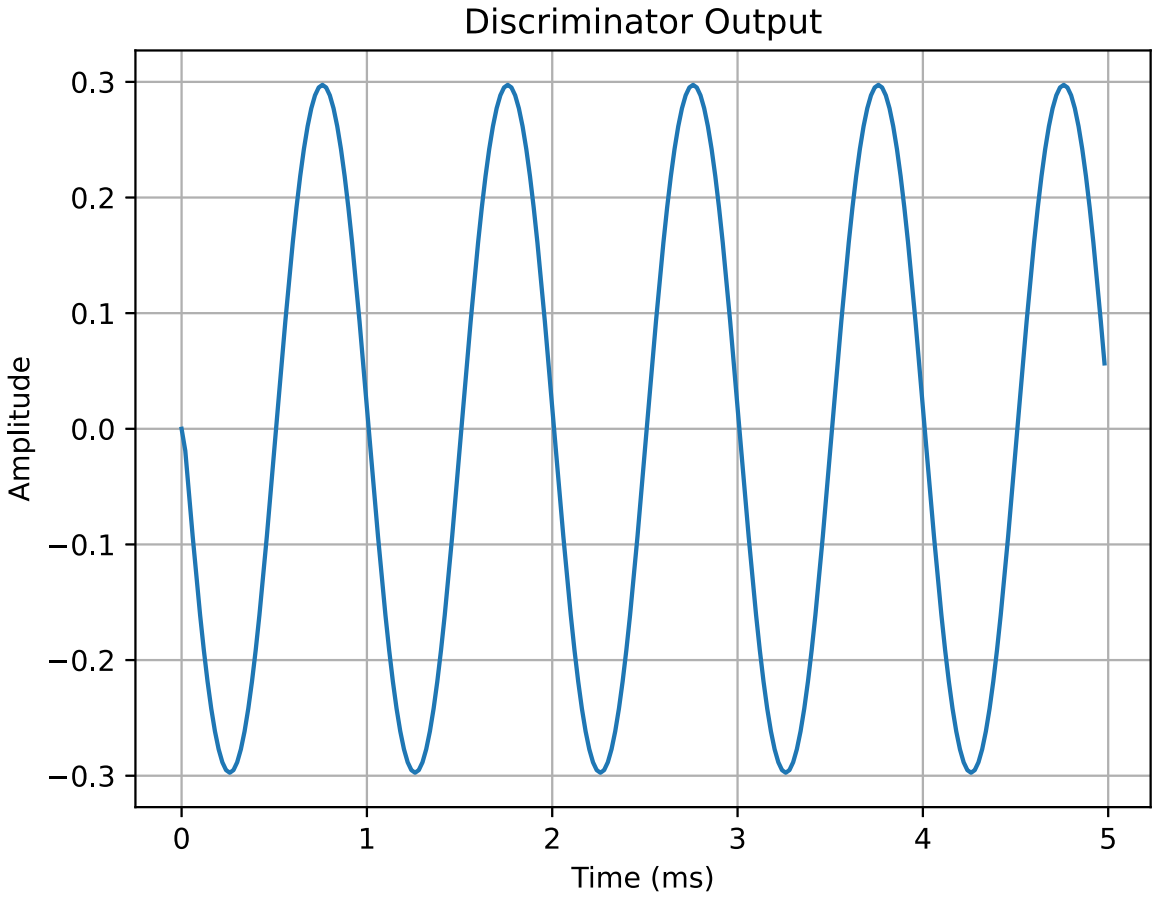
- Form and estmate of the power spectral density (not simply the spectrum or Fourier transform) of `xc`
- You see from the plot below that the spectrum appears as a line spectrum just the power spectrum for any other periodic signal dealt with in Chapter 2
- The signal is indeed complex as it is formed from a complex sinusoid.

In [26]:
```
Pxc,fxc = ss.psd(xc,2**12,fs,scale_noise=False);
plot(fxc/1000,10*log10(Pxc))
title(r'Complex Baseband FM Modulator Output')
xlim([-10,10])
xlabel(r'Frequency (kHz)')
ylim([-80,0])
grid()
# savefig('fm_spec_for_fig47.pdf')
```




- Below you see the sinsusoidal is message is recovered:

In [27]:
```
t = n/fs*1e3 # units of ms
Nmax = 250
plot(t[:Nmax],yD[:Nmax])
title(r'Discriminator Output')
ylabel(r'Amplitude')
xlabel(r'Time (ms)')
grid();
# savefig('fm_wavefrm_for_fig47.pdf')
```

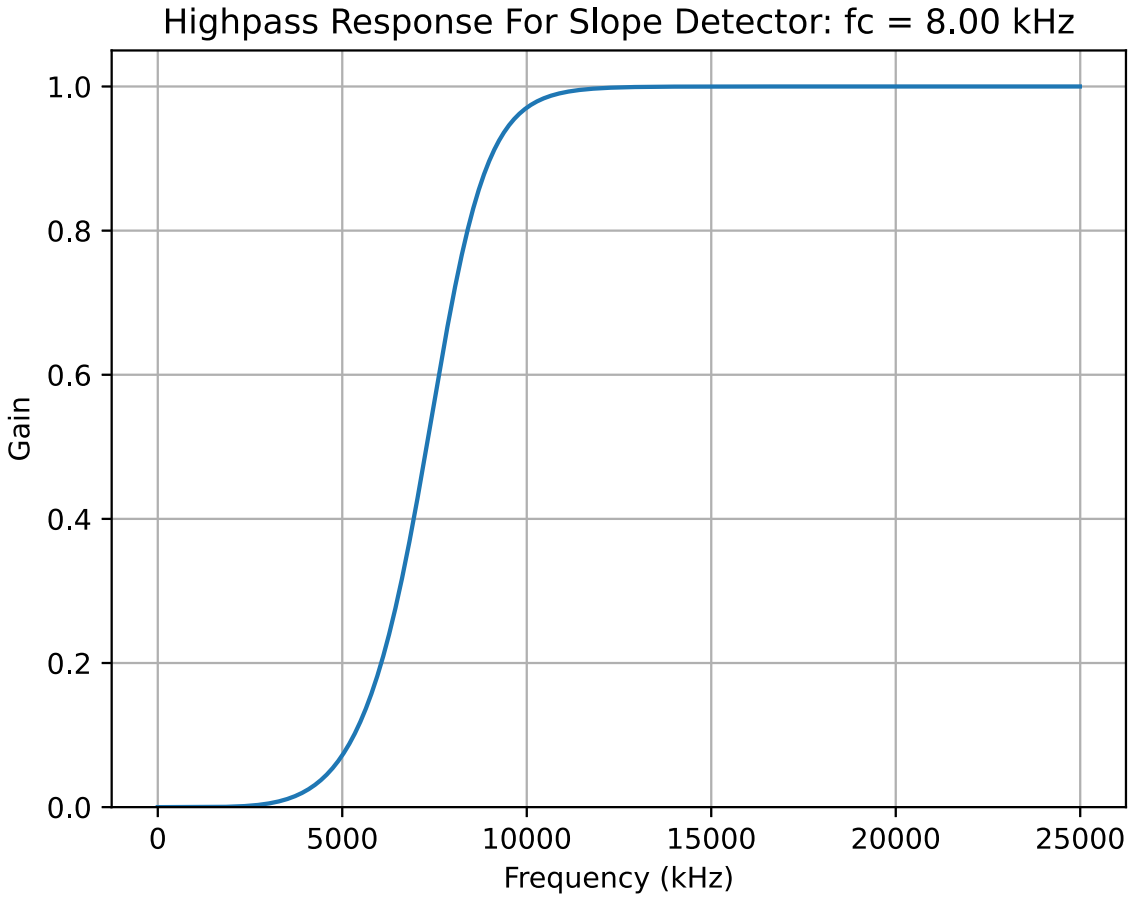



# Slope Detector

The slope detector form of FM demodulator uses the gain slope of a filter, say highpass or bandpass, to convert frequency deviation to amplitude fluctuations. An envelope detector can then recover the FM modulation converted to AM modulation.

Here we consider a Butterworth highpass filter, implemented as a digital filter for further use in simulation. In particular use a 5th-order design.

In [28]:
```
fs = 50000
fc = 8000
b,a = signal.butter(5,2*fc/fs,'high') # could use iir_design_helper to
f = arange(0,25000,10)
w,H = signal.freqz(b,a,2*pi*f/fs)
plot(f,abs(H))
title(r'Highpass Response For Slope Detector: fc = %1.2f kHz' % (fc/10
ylabel(r'Gain')
xlabel(r'Frequency (kHz)')
ylim([0,1.05])
grid();
```

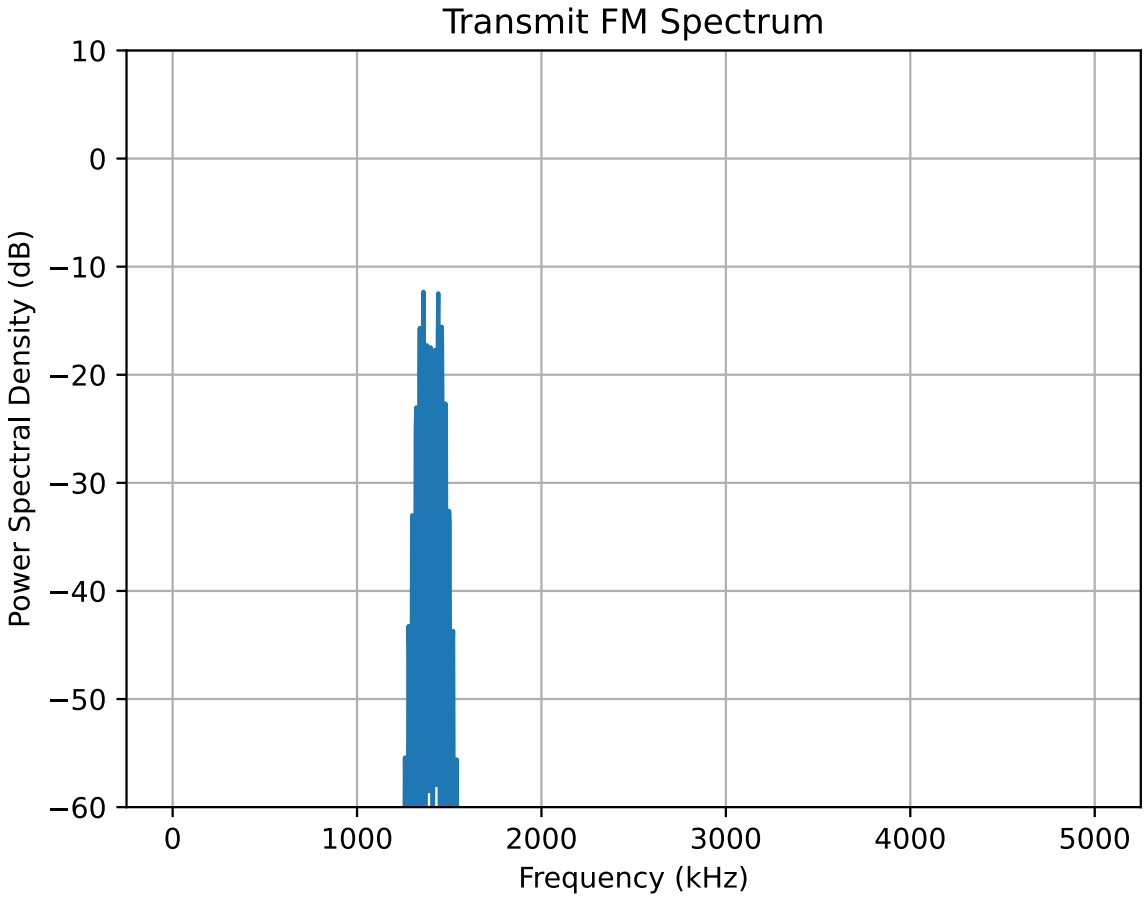



# FM Modulator Using a Non-Zero Carrier Frequency

First generate an FM signal on a carrier frequency of 10 kHz. Use single tone FM with $f_m = 100$ Hz and $f_D = 100$ as starting points.

In [29]:
```
n = arange(0,10000)
fm = 200
m = cos(2*pi*100/fs*n)
f0 = 7000
fD = 100
xc = cos(2*pi*f0/fs*n + 2*pi*fD/fm*m)
```

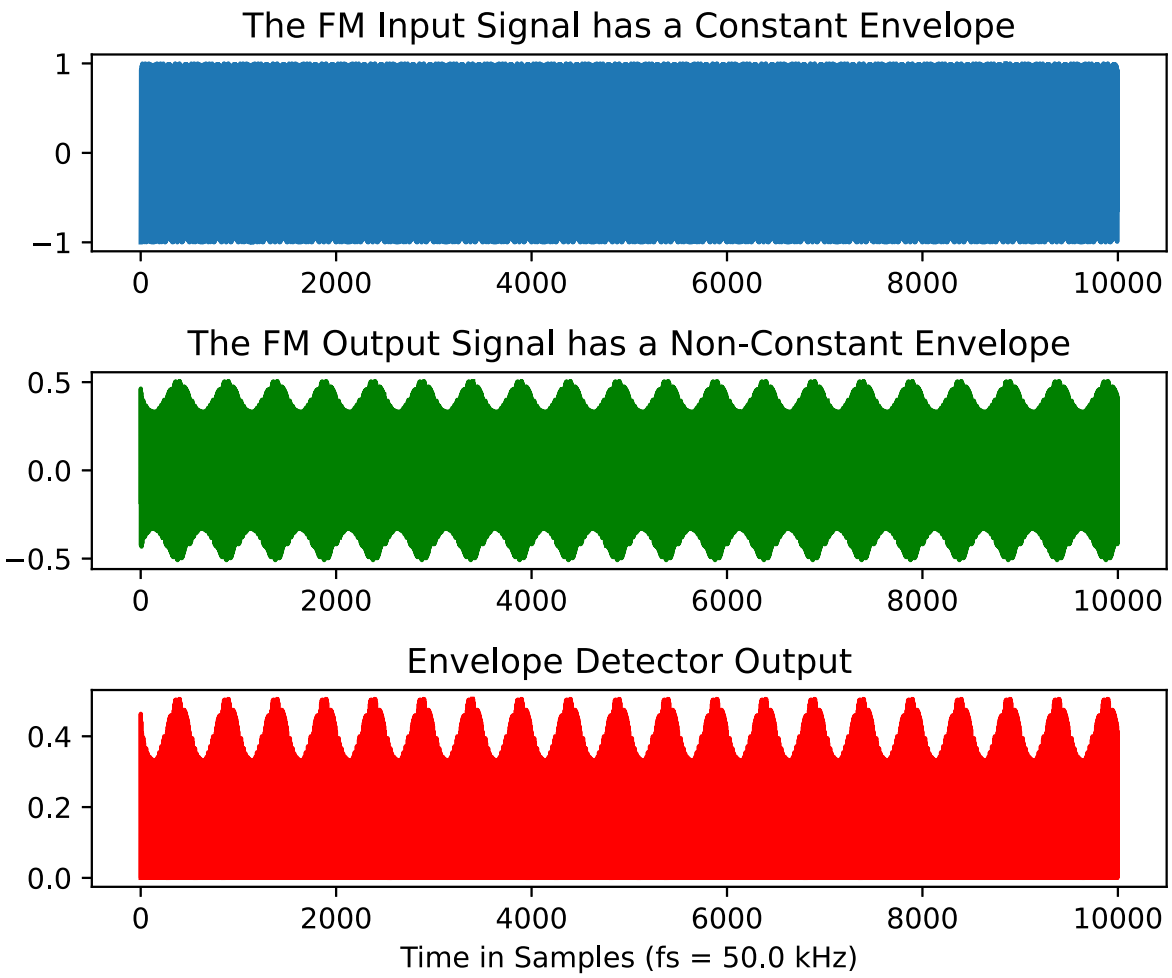
In [46]:
```
Pxc,fxc = ss.psd(xc,2**12,fs,scale_noise=False);
plot(fxc,10*log10(Pxc))
ylim([-60,10])
ylabel(r'Power Spectral Density (dB)')
xlabel(r'Frequency (kHz)')
title(r'Transmit FM Spectrum')
grid();
```




## Pass the FM Signal Through the Highpass FIlter

Next we highpass filter the FM signal which will introduce envelope fluctuations in the carrier envelope, as the filter gain slope is passing right through $f_0 = 7$ kHz. Following that we envelope detect to recover the FM that has been converted to AM.

In [31]:
```
yc = signal.lfilter(b,a,xc)
figure(figsize=(6,5))
subplot(311)
plot(n,xc)
ylim([-1.1,1.1])
title(r'The FM Input Signal has a Constant Envelope')
subplot(312)
plot(n,yc,'g')
title(r'The FM Output Signal has a Non-Constant Envelope')
# ylim([-.025,.025])
subplot(313)
plot(n,yc*(1+sign(yc))/2,'r')
# ylim([-.005,.025])
title(r'Envelope Detector Output')
xlabel(r'Time in Samples (fs = %1.1f kHz)' % (fs/1000,))
tight_layout()
```




# Phase Locked Loops

In [ ]:
```
pll.cbb_pll(
```

The module `synchronization.py`, imported at the top of t his notebook, contains PLLs in addition to digital communications synchronization algorithms. One of the functions inside this module is `PLL1`, listed below

```
def PLL1(theta,fs,loop_type,Kv,fn,zeta,non_lin):
    """
    theta_hat, ev, phi =
PLL1(theta,fs,loop_type,Kv,fn,zeta,non_lin)
    Baseband Analog PLL Simulation Model

================================================================
        theta = input phase deviation in radians
           fs = sampling rate in sample per second or Hz
    loop_type = 1, first-order loop filter F(s)=K_LF; 2,
integrator
                with lead compensation F(s) = (1 + s tau2)/(s
```

```
tau1),
                i.e., a type II, or 3, lowpass with lead
compensation
                F(s) = (1 + s tau2)/(1 + s tau1)
           Kv = VCO gain in Hz/v; note presently assume Kp =
1v/rad
                and K_LF = 1; the user can easily change this
            fn = Loop natural frequency (loops 2 & 3) or cutoff
                frquency (loop 1)
          zeta = Damping factor for loops 2 & 3
      non_lin = 0, linear phase detector; 1, sinusoidal phase
detector

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    theta_hat = Output phase estimate of the input theta in
radians
           ev = VCO control voltage
          phi = phase error = theta - theta_hat

================================================================
    Alternate input in place of natural frequency, fn, in Hz
is
    the noise equivalent bandwidth Bn in Hz.

================================================================
    Mark Wickert, April 2007 for ECE 5625/4625
    Modified February 2008 and July 2014 for ECE 5675/4675
    Python version August 2014
    """
    T = 1/float(fs)
    Kv = 2*np.pi*Kv # convert Kv in Hz/v to rad/s/v

    if loop_type == 1:
        # First-order loop parameters
        # Note Bn = K/4 Hz but K has units of rad/s
        #fn = 4*Bn/(2*pi);
        K = 2*np.pi*fn # loop natural frequency in rad/s
    elif loop_type == 2:
        # Second-order loop parameters
        #fn = 1/(2*pi) * 2*Bn/(zeta + 1/(4*zeta));
        K = 4 *np.pi*zeta*fn # loop natural frequency in rad/s
        tau2 = zeta/(np.pi*fn)
    elif loop_type == 3:
        # Second-order loop parameters for one-pole lowpass
with
        # phase lead correction.
        #fn = 1/(2*pi) * 2*Bn/(zeta + 1/(4*zeta));
        K = Kv # Essentially the VCO gain sets the single-
sided
                # hold-in range in Hz, as it is assumed that
```

```
Kp = 1
                # and KLF = 1.
        tau1 = K/((2*np.pi*fn)^2);
        tau2 = 2*zeta/(2*np.pi*fn)*(1 - 2*pi*fn/K*1/(2*zeta))
    else:
        print('Loop type must be 1, 2, or 3')

    # Initialize integration approximation filters
    filt_in_last = 0; filt_out_last = 0;
    vco_in_last = 0; vco_out = 0; vco_out_last = 0;

    # Initialize working and final output vectors
    n = np.arange(len(theta))
    theta_hat = np.zeros_like(theta)
    ev = np.zeros_like(theta)
    phi = np.zeros_like(theta)

    # Begin the simulation loop
    for k in  xrange(len(n)):
        phi[k] = theta[k] - vco_out
        if non_lin == 1:
            # sinusoidal phase detector
            pd_out = np.sin(phi[k])
        else:
            # Linear phase detector
            pd_out = phi[k]
        # Loop gain
        gain_out = K/Kv*pd_out # apply VCO gain at VCO
        # Loop filter
        if loop_type == 2:
            filt_in = (1/tau2)*gain_out
            filt_out = filt_out_last + T/2*(filt_in +
filt_in_last)
            filt_in_last = filt_in
            filt_out_last = filt_out
            filt_out = filt_out + gain_out
        elif loop_type == 3:
            filt_in = (tau2/tau1)*gain_out -
(1/tau1)*filt_out_last
            u3 = filt_in + (1/tau2)*filt_out_last
            filt_out = filt_out_last + T/2*(filt_in +
filt_in_last)
            filt_in_last = filt_in
            filt_out_last = filt_out
        else:
            filt_out = gain_out;
        # VCO
        vco_in = filt_out
        if loop_type == 3:
            vco_in = u3
```

```
        vco_out = vco_out_last + T/2*(vco_in + vco_in_last)
        vco_in_last = vco_in
        vco_out_last = vco_out
        vco_out = Kv*vco_out # apply Kv
        # Measured loop signals
        ev[k] = vco_in
        theta_hat[k] = vco_out
    return theta_hat, ev, phi
```

# 1st-Order Loop Simulation

Set up a first-order PLL with loop gain $K_t/(2\pi) = 10$ Hz, which means the lock range is $\pm 10$ Hz.

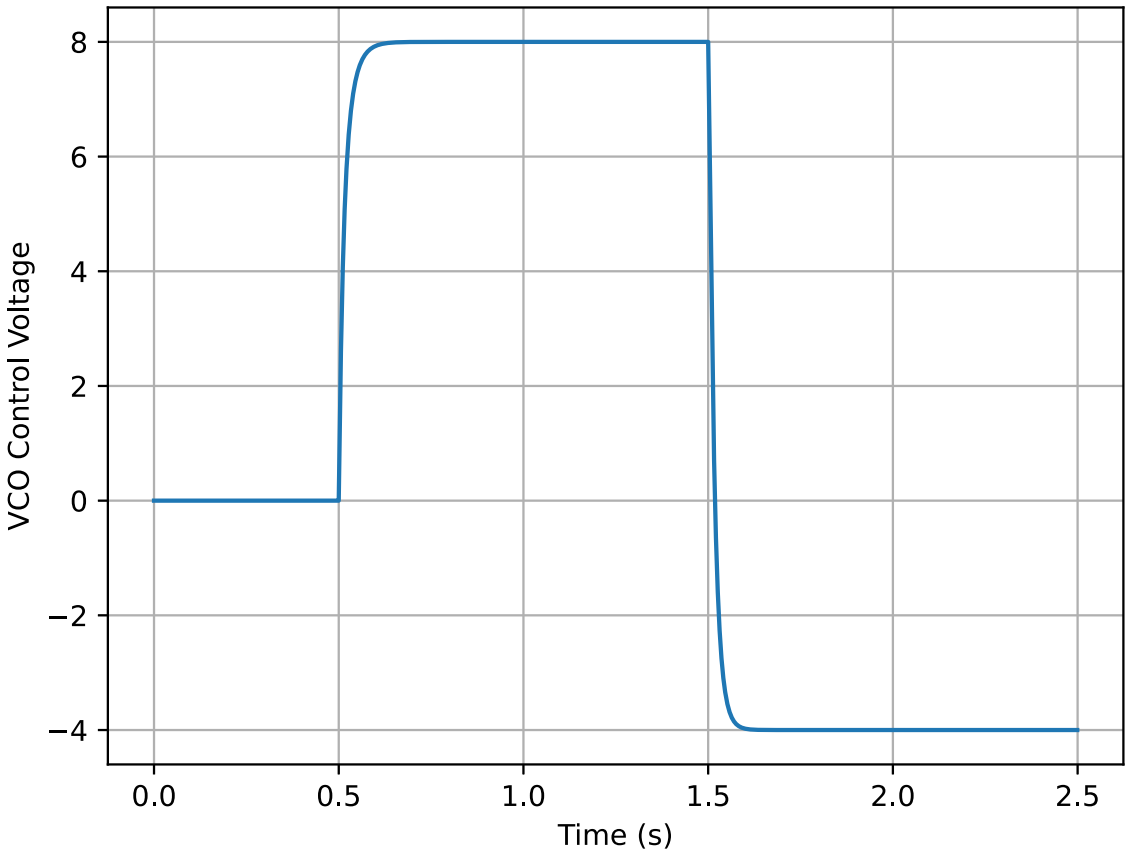
- We input $\phi(t)$ as a phase ramp (frequency step) involving the difference of two step functions
- The initial input is

$$\phi(t) = 2\pi \left\{ 8\left(t - \frac{1}{2}\right) u(t - 0.5) - 12\left(t - \frac{3}{2}\right) u(t - 1.5) \right\} \quad (8)$$

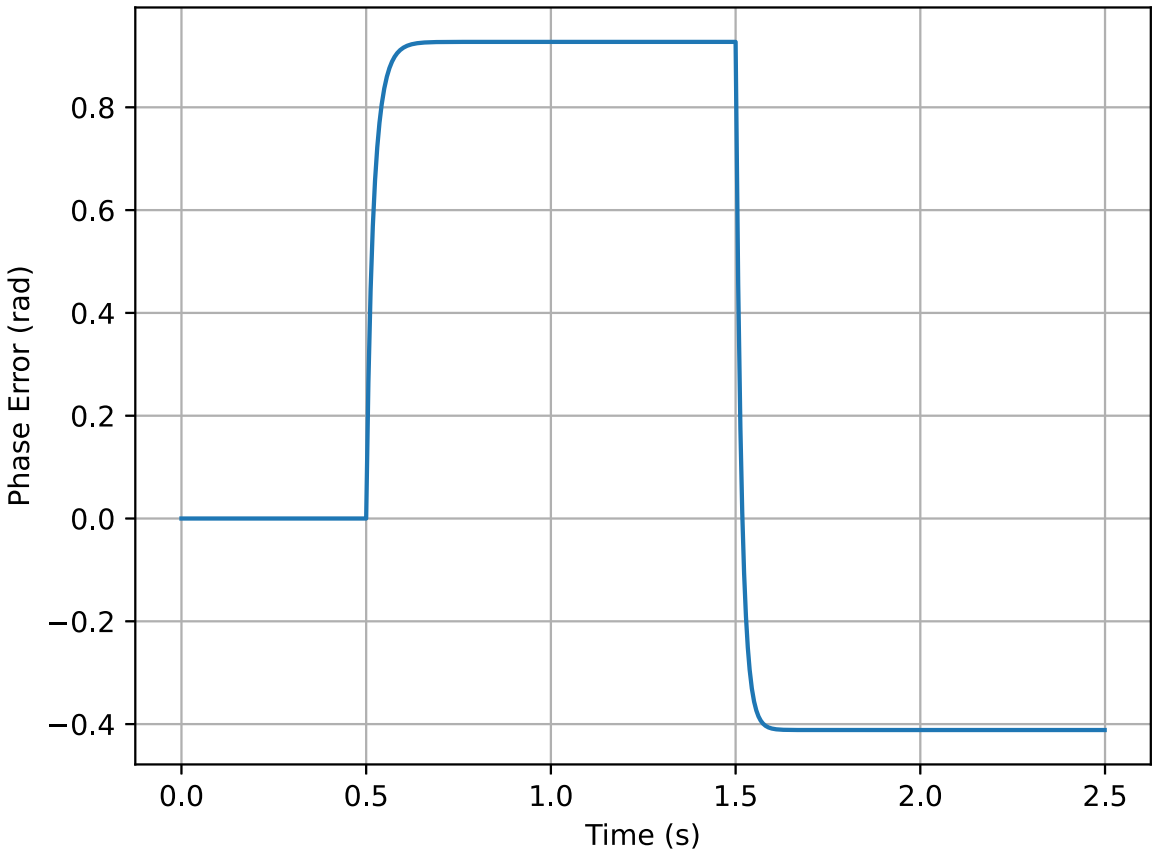
- The sampling frequency used for the simulation is 1000 Hz

In [32]:
```
fs = 1000 # dt = 1/fs
t = arange(0,2.5,1/fs)
phi = 2*pi*8*((t-0.5)*ss.step(t-0.5)) - 2*pi*12*((t - 1.5)*ss.step(t-1
theta_hat, ev, psi = pll.PLL1(phi,1000,1,1,10,0.707,1)
```

In [33]:
```
plot(t,ev)
#plot(t,psi)
xlabel(r'Time (s)')
ylabel(r'VCO Control Voltage')
grid();
# savefig('pll1_ev.pdf')
```



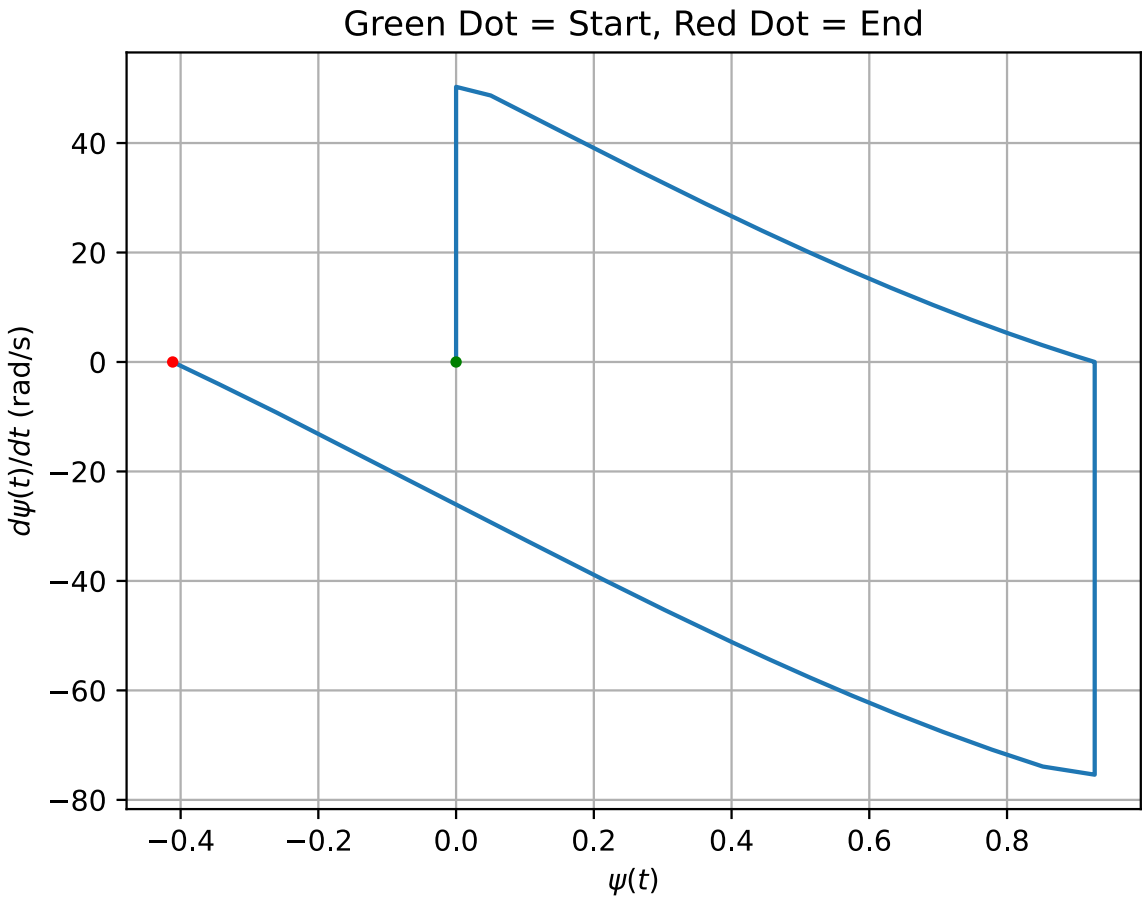

In [34]:
```
#plot(t,ev)
plot(t,psi)
xlabel(r'Time (s)')
ylabel(r'Phase Error (rad)')
grid();
# savefig('pll1_psi.pdf')
```




## Making Sense of the Phase Plane

The phase plane plot is $d\psi(t)/dt$ versus $\psi(t)$. We can form this plot by taking the outputs of the PLL simulation and doing some further signal processing.

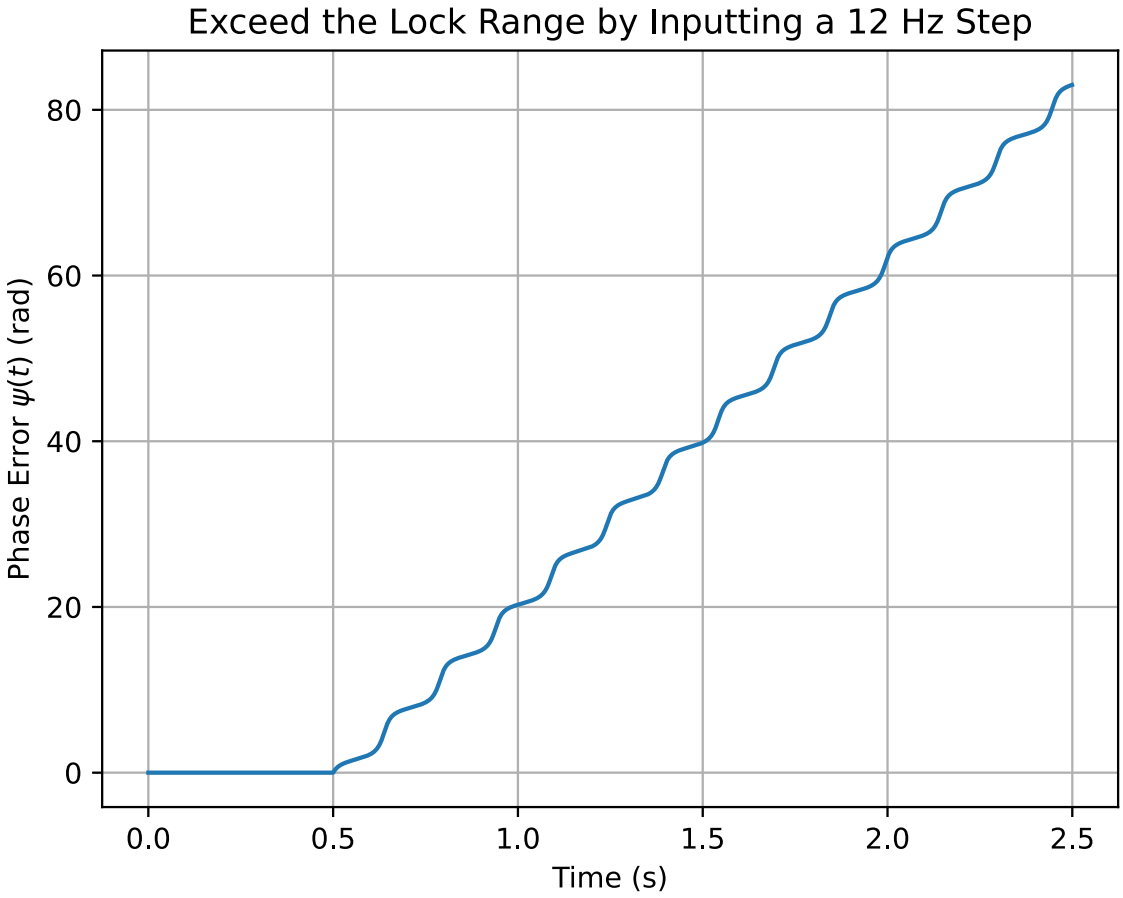
In [35]:
```
psi_dot = diff(psi)*fs # units rad/s
plot(psi[:-1],psi_dot)
plot(psi[0],psi_dot[0],'g.')
plot(psi[-2],psi_dot[-1],'r.')
title(r'Green Dot = Start, Red Dot = End')
xlabel(r'$\psi(t)$')
ylabel(r'$d\psi(t)/dt$ (rad/s)')
grid();
# savefig('pll1_phase_plane.pdf')
```




# Drive the 1st-Order Loop Out of Lock

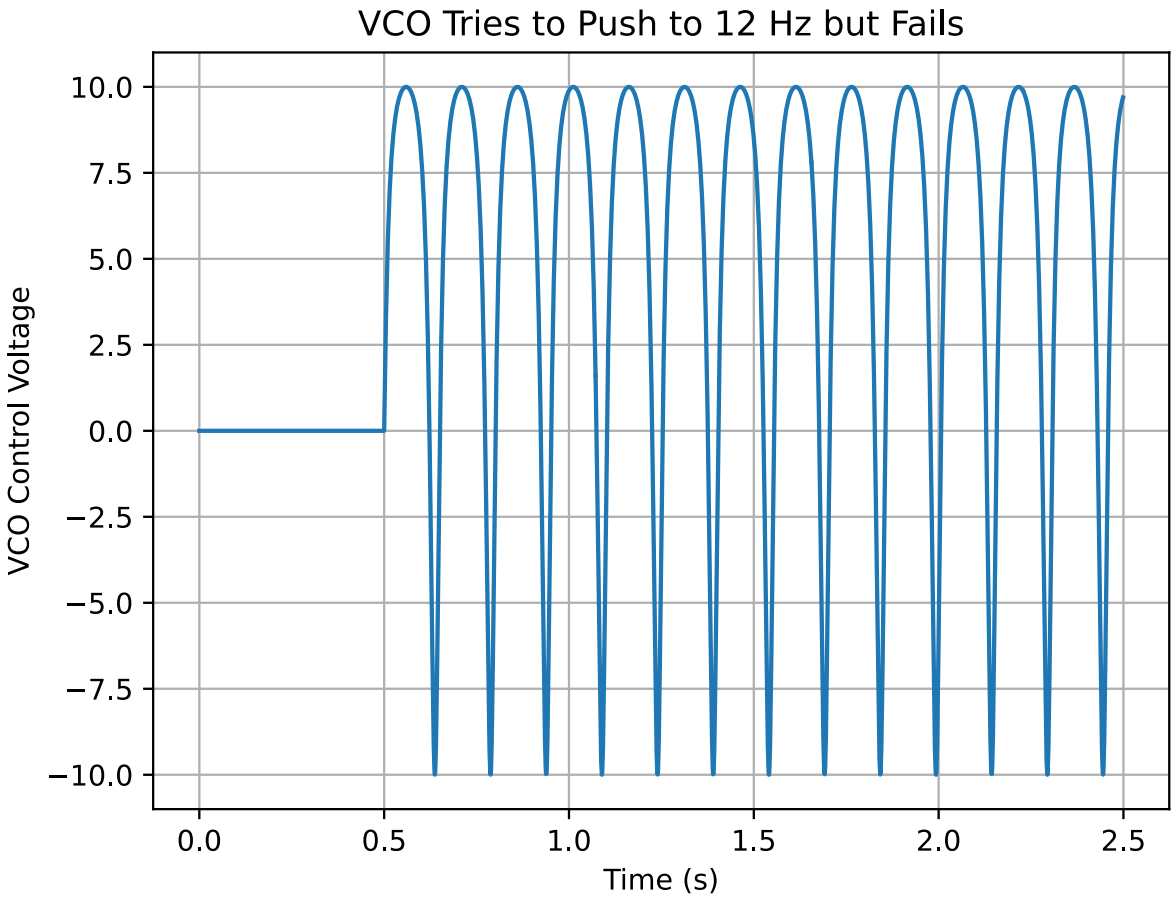
In [36]:
```
t = arange(0,2.5,1/1000)
phi = 2*pi*12*((t-0.5)*ss.step(t-0.5))
theta_hat, ev, psi = pll.PLL1(phi,1000,1,1,10,0.707,1)
```

In [37]:
```
#plot(t,ev)
plot(t,psi)
title(r'Exceed the Lock Range by Inputting a 12 Hz Step')
xlabel(r'Time (s)')
ylabel(r'Phase Error $\psi(t)$ (rad)')
grid();
# savefig('pll1_psi_nolock.pdf')
```

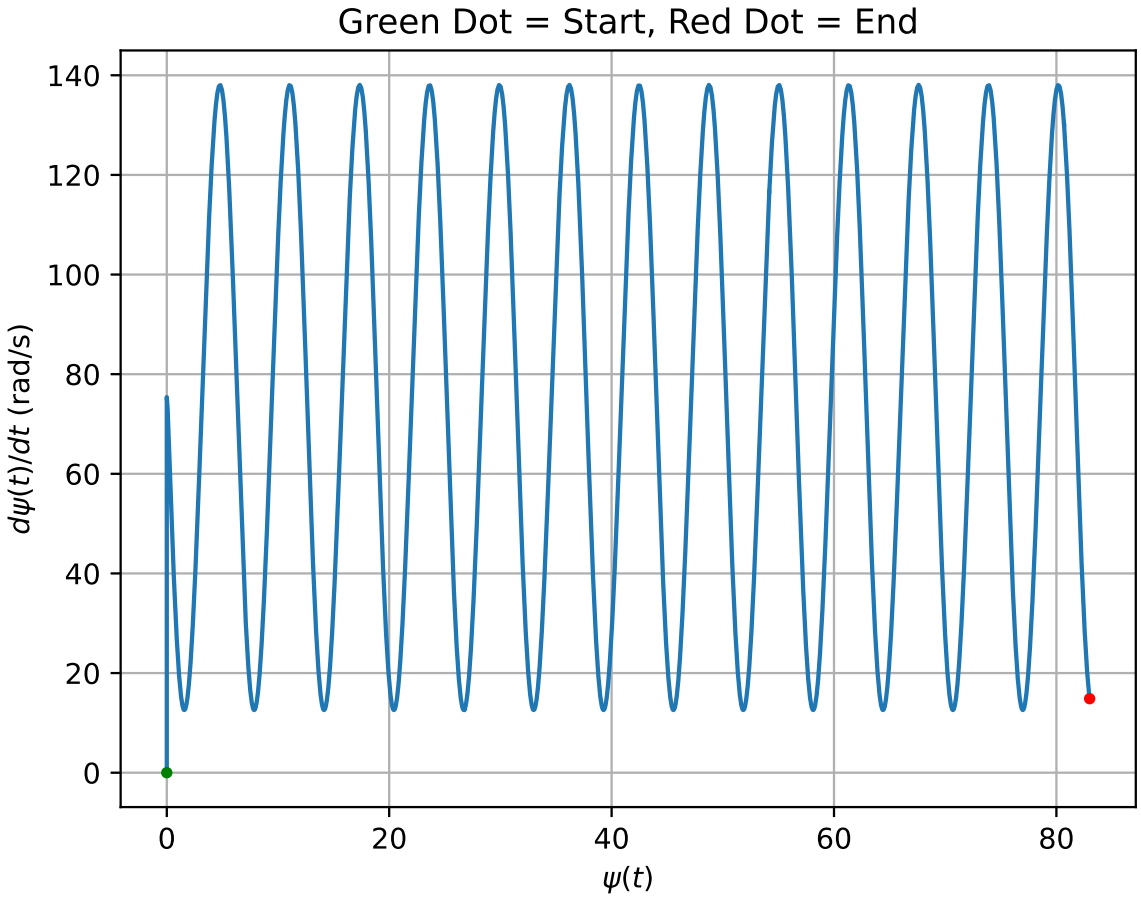



In [141…

```
plot(t,ev)
#plot(t,psi)
title(r'VCO Tries to Push to 12 Hz but Fails')
xlabel(r'Time (s)')
ylabel(r'VCO Control Voltage')
grid();
savefig('pll1_ev_nolock.pdf')
```



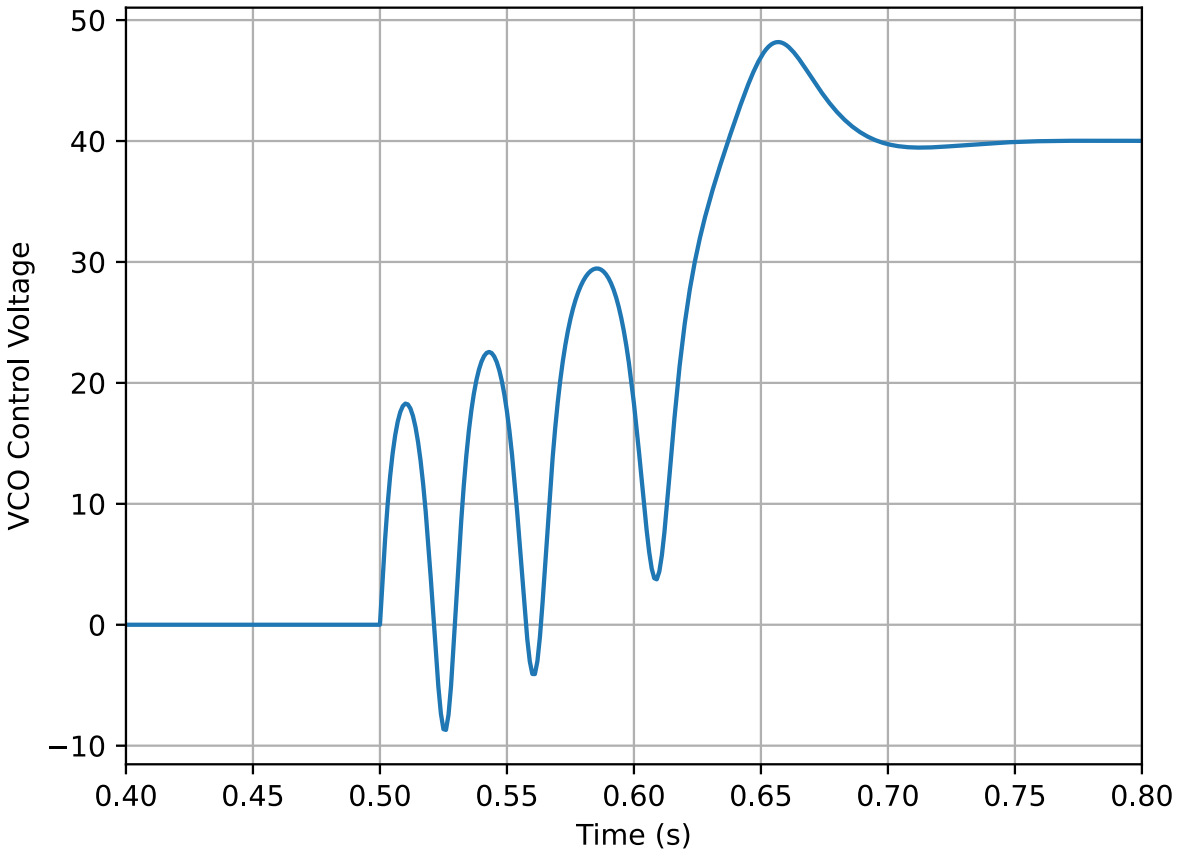

In [38]:
```
psi_dot = diff(psi)*fs # units rad/s
plot(psi[:-1],psi_dot)
plot(psi[0],psi_dot[0],'g.')
plot(psi[-2],psi_dot[-1],'r.')
title(r'Green Dot = Start, Red Dot = End')
xlabel(r'$\psi(t)$')
ylabel(r'$d\psi(t)/dt$ (rad/s)')
grid();
# savefig('pll1_phase_plane_nolock.pdf')
```




# Second-Order Loop Simulation

Drive a second-order PLL with a frequency step $\Delta f$. Intially set: $\Delta f = 40$ Hz, $f_s = 1000$ Hz, $K_v = 1$Hz/V, $\zeta = 0.707$, and $\sin(()$ nonlinearity present.

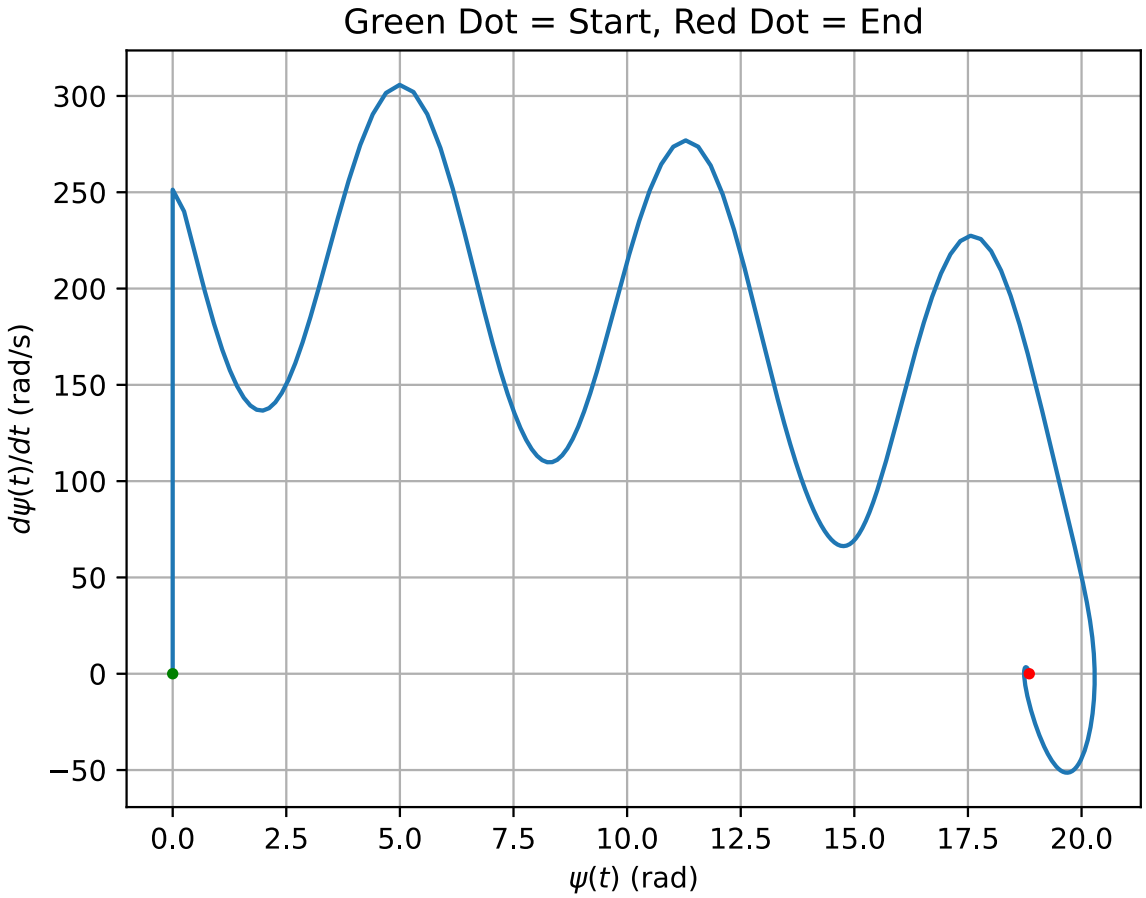
In [39]:
```
t = arange(0,2.5,1/1000)
DeltaF = 40
#phi = 2*pi*8*((t-0.5)*ssd.step(t-0.5)) - 2*pi*12*((t - 1.5)*ssd.step(
phi = 2*pi*DeltaF*((t-0.5)*ss.step(t-0.5))
#                        phi(t),fs,type,Kv(Hz/V),fn,zeta,non_lin
theta_hat, ev, psi = pll.PLL1(phi,1000,2,1,10,0.707,1)
plot(t,ev)
#plot(t,psi)
xlim([0.4,0.8])
xlabel(r'Time (s)')
ylabel(r'VCO Control Voltage')
grid();
# savefig('pll2_ev40.pdf')
```




## Phase Plane Plot

With the 40 Hz frequency step the phase-plane plot shows the loop initially slipping cycles ($d\psi(t)/dt > 0$), until finally the trajectory drops below $d\psi(t)/dt = 0$ and finally comes back up to settle at $d\psi(t)/dt = 0$.

In [40]:
```
psi_dot = diff(psi)*fs
plot(psi[:-1],psi_dot)
plot(psi[0],psi_dot[0],'g.')
plot(psi[-2],psi_dot[-1],'r.')
title(r'Green Dot = Start, Red Dot = End')
xlabel(r'$\psi(t)$ (rad)')
ylabel(r'$d\psi(t)/dt$ (rad/s)')
grid();
# savefig('pll2_phase_plane40.pdf')
```




# Complex Baseband PLL

Here we consider a PLL that operates with a complex signal input, simply the phase of the input. The function `pll.PLL_cbb()` is a rework of `PLL1()` to allow a complex signal input using a sinusoidal phase detector.

No description has been provided for this image

**Note** The block diagram above was created for the ECE 5675 PLL course so the variables are defined differently. In particular in the block diagram $\theta(t) \rightarrow \phi(t)$, $\hat{\theta}(t) \rightarrow \theta(t)$, and $e_d(t) \rightarrow A_c \sin(\psi(t))$.

With the simulation code below you can model a PLL at the full waveform level, not just phase in/phase out, as is done in `PLL1`. A sinusoidal phase detector is employed which forms as output

$$e_d(t) = \text{Im}\left[x_r(t) \cdot e_o^*(t)\right] \tag{9}$$

where the now complex signals $x_r(t)$ and $e_o(t)$ are of the form

$$x_r(t) = A_c e^{j[2\pi f_c t + \phi(t)]} \tag{10}$$

$$e_o(t) = e^{j\theta(t)} \tag{11}$$

In advanced applications $x_r(t)$ may also include a noise signal. In what follows $A_c = 1$ to insure that the internal gain scaling for the loop parameters remains calibrated.

For the special homework problem the example allows you to study the sinusoidal FM response of the PLL. The FM modulation input is setup in the lines:

```
# Sinusoidal modulation at fm Hz and fD Hz peak deviation
Am = 1
fD = 1
fm = 10
phi = fD/fm*Am*cos(2*pi*fm*t)
```

Under linear operation of the loop the VCO control signal $e_v(t)$ is the sinusoidal response to the input message $m(t) = A_m \cos(2\pi f_m t + \phi_m)$, where here $A_m = 1$ and $\phi_m = 0$. From the loop linear theory the amplitude of $e_v(t)$, in steady-state is

$$\left|E_v(j2\pi f_m)\right| = \frac{k_f}{K_v} A_m |e^{j\phi_m}| \left|H(j2\pi f_m)\right| = \frac{f_D}{K_v'} A_m |e^{j\phi_m}| \left|H(j2\pi f_m)\right| \quad (12$$

where $H(j2\pi f) = H(s)\big|_{s=j2\pi f}$ is the frequency response of the closed-loop transfer function. Recall also that for an FM modulator $k_f = 2\pi f_D$ and $K_v$, the VCO gain constant includes a $2\pi$ when one considers the VCO gain in Hz/V. This means that $k_f/K_v$ simplifies to $f_D/K_v'$, where the $K_v'$ has units of Hz/V, i.e., $K_v = 2\pi K_v'$. In the simulation below $K_v' = 1$, $A_m = 1$, and $f_D = 1$, so the expected peak amplitude of $e_v(t)$ is $|H(j2\pi f_m)|$.

To evaluate $|H(j2\pi f_m)|$ in Python consider the first-order loop:

In [143…

```
f = arange(0,20,1)
# H(f) = K_t/(s + K_t) with s = 2*pi*f and Kt = 2*pi*fn, so
# H(f) = f/(1j*f + fn)
fn = 10
magH = abs(fn/(1j*f + fn))
(f,magH)
```

Out[143… (array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19]),
array([1. , 0.99503719, 0.98058068, 0.95782629, 0.92847669,
0.89442719, 0.85749293, 0.81923192, 0.78086881, 0.74329415,
0.70710678, 0.67267279, 0.6401844 , 0.60971076, 0.58123819,
0.5547002 , 0.52999894, 0.50702013, 0.48564293, 0.46574643]))

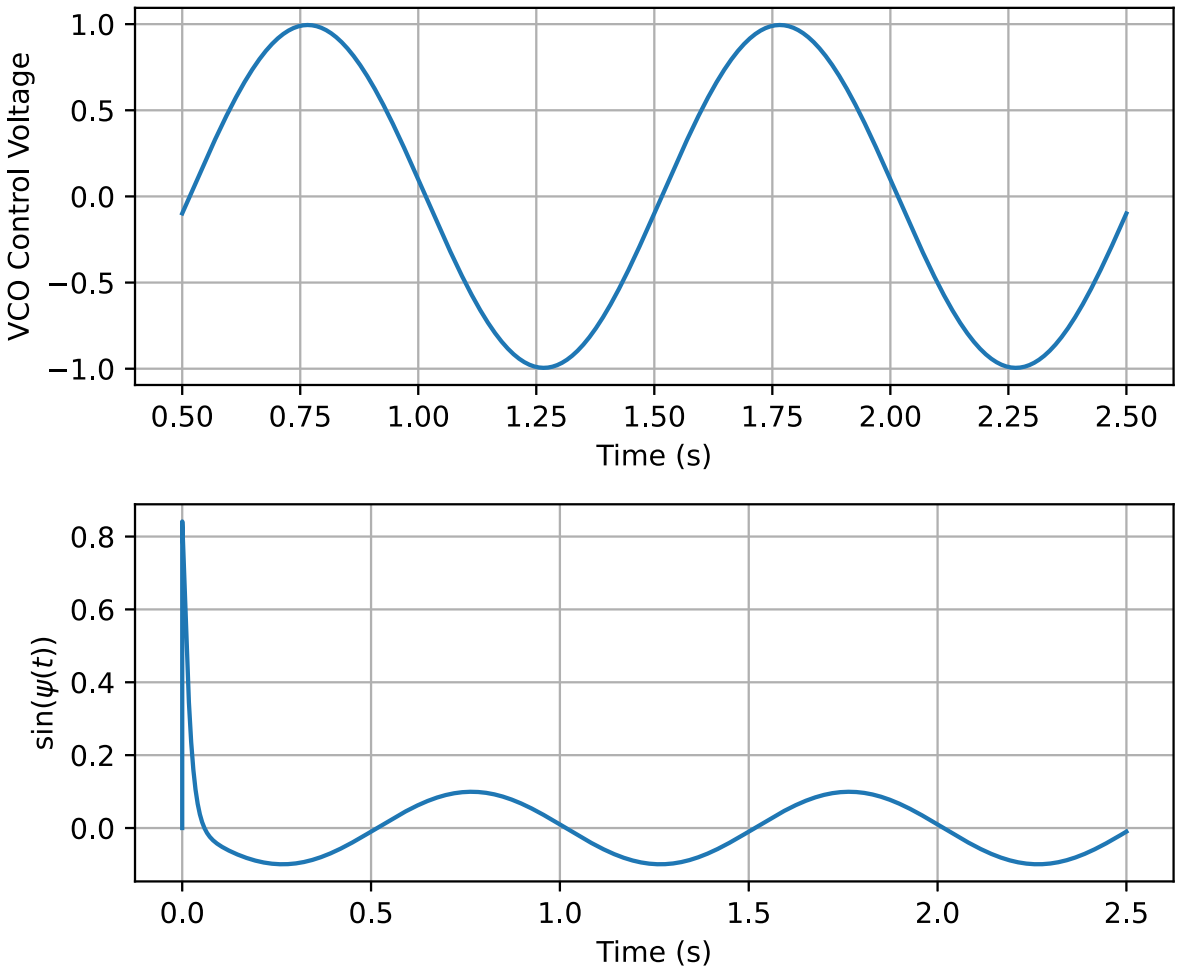
- Note at `f = 10` , `magH = 0.707` as expected!
- For the second-order loop things are a bit more complicated because $H(f)$ from notes page 4-42 & 4-43 is of the form

$$H(f) = H(s)\big|_{s=j2\pi f} \tag{13}$$

$$\text{where } H(s) = \frac{K_t(s+a)}{s^2 + K_t a s + K_t a} = \frac{K_t(s+a)}{s^2 + 2\zeta\omega_n s + \omega_n^2}, \tag{14}$$

$\omega_n = 2\pi f_n$, and $a$ and $K_t$ can be written in terms of $\omega_n$ and the damping $\zeta$.

In [41]:
```
fs = 10000
t = arange(0,2.5,1/fs)
# Optional Carrier frequency step
Df = 0 # Frequency step (Delta_f) applied directly to the complex sinu
f0 = Df*ss.step(t-0.5)
# Sinusoidal modulation at fm Hz and fD Hz peak deviation
Am = 1
fD = 1
fm = 1 # You vary this in Problem 2 of Set #11
phi = fD/fm*Am*cos(2*pi*fm*t) # note beta = fD/fm*Am for sinusoidal FM
x = exp(1j*(2*pi*f0*t+phi)) # modulation is phi
# cpx input, fs, loop_type (1, 2, or 3), Kv (V/Hz), fn (Hz), zeta
theta_hat, ev, psi = pll.PLL_cbb(x,fs,1,1,10,0.707)
figure(figsize=(6,5))
subplot(211)
# plot(t,ev)
plot(t[-20000:],ev[-20000:]) # last 20000
xlabel(r'Time (s)')
ylabel(r'VCO Control Voltage')
grid();
subplot(212)
plot(t,psi)
xlabel(r'Time (s)')
ylabel(r'$\sin(\psi(t))$')
grid();
tight_layout()
```




In [42]:
```
max(ev[-20000:])
```

Out[42]:
```
np.float64(0.9950528517663114)
```

In [43]:
```
Ev_mag_sim = 20*log10(sqrt(2)*std(ev[-20000:])) # use last 20000 point
print('fm = %1.2f Hz and Ev_mag_sim = %1.2f dB' % (fm,Ev_mag_sim))
```

```
fm = 1.00 Hz and Ev_mag_sim = -0.04 dB
```

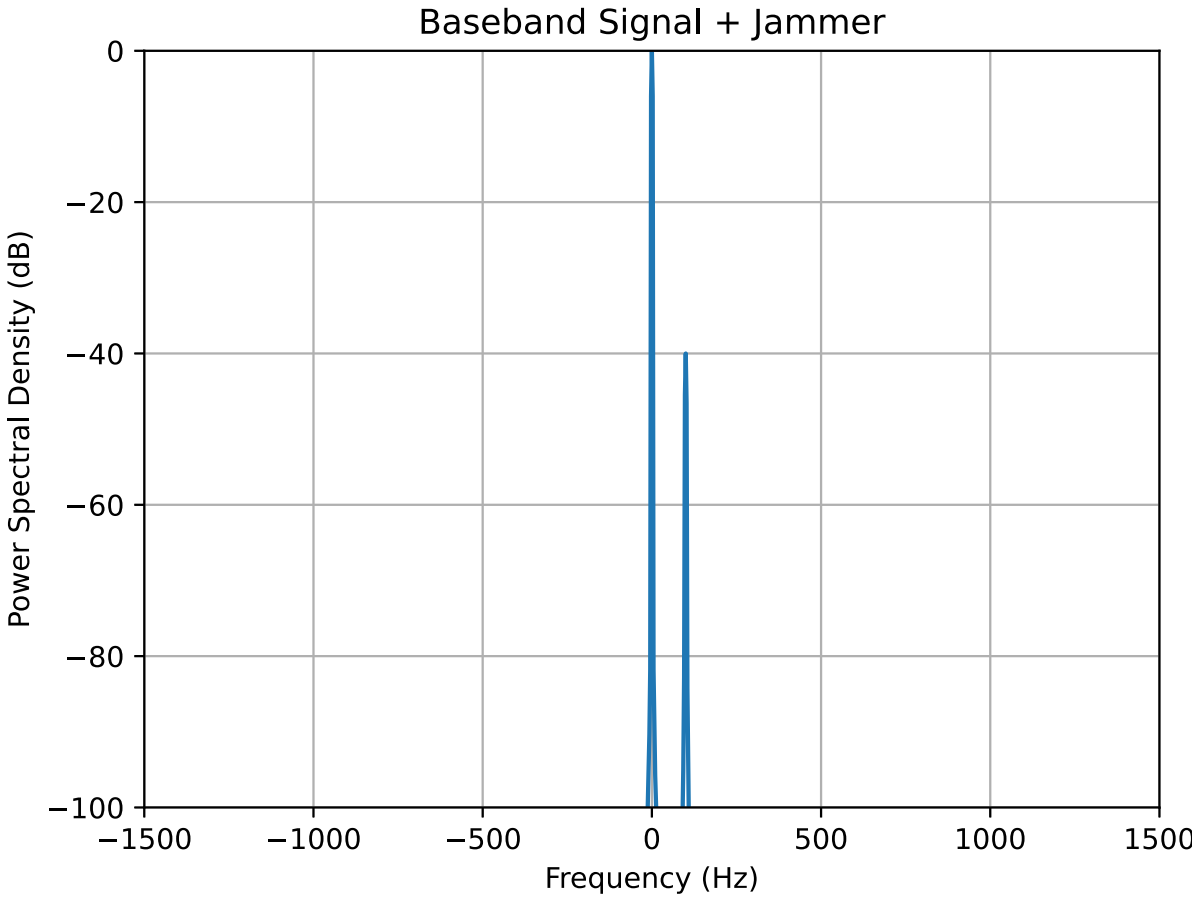
- As expected the peak value of $e_v(t)$ ( `ev` ) in steady-state is about 1 when $f_m \ll f_n$
- The factor of $\sqrt{2}$ scales the `std()` calculation so the RMS value of the unit amplitude sinusoid is normalized from $1/\sqrt{2} = 0.70$ to $1.0 \leftrightarrow 0$ dB

# Discriminator with Tone Interference

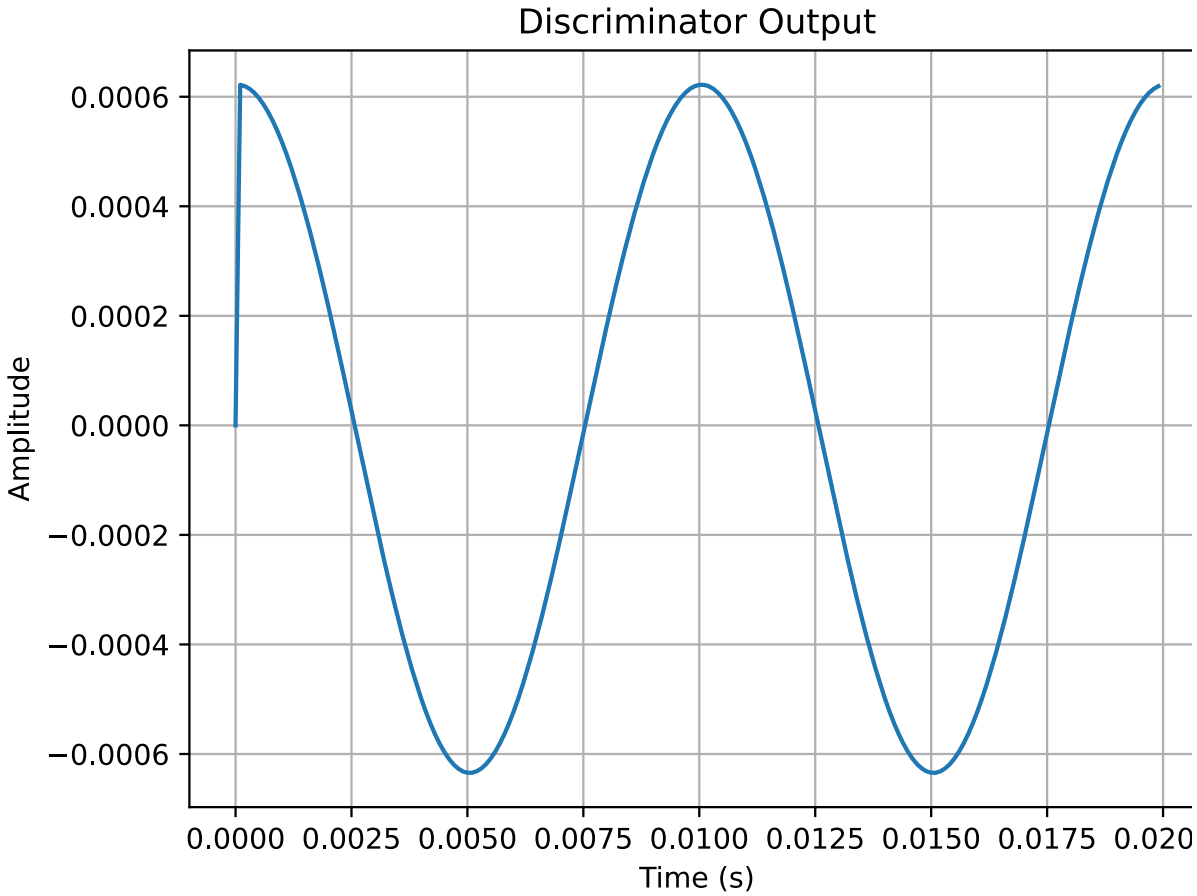
In [51]:
```
fs = 10000
t = arange(0,1,1/fs)
Ac = 1.0
```

```
Ai = 0.01
fi = 100 # in Hz
xr = 1 + Ai*exp(1j*2*pi*fi*t)
```

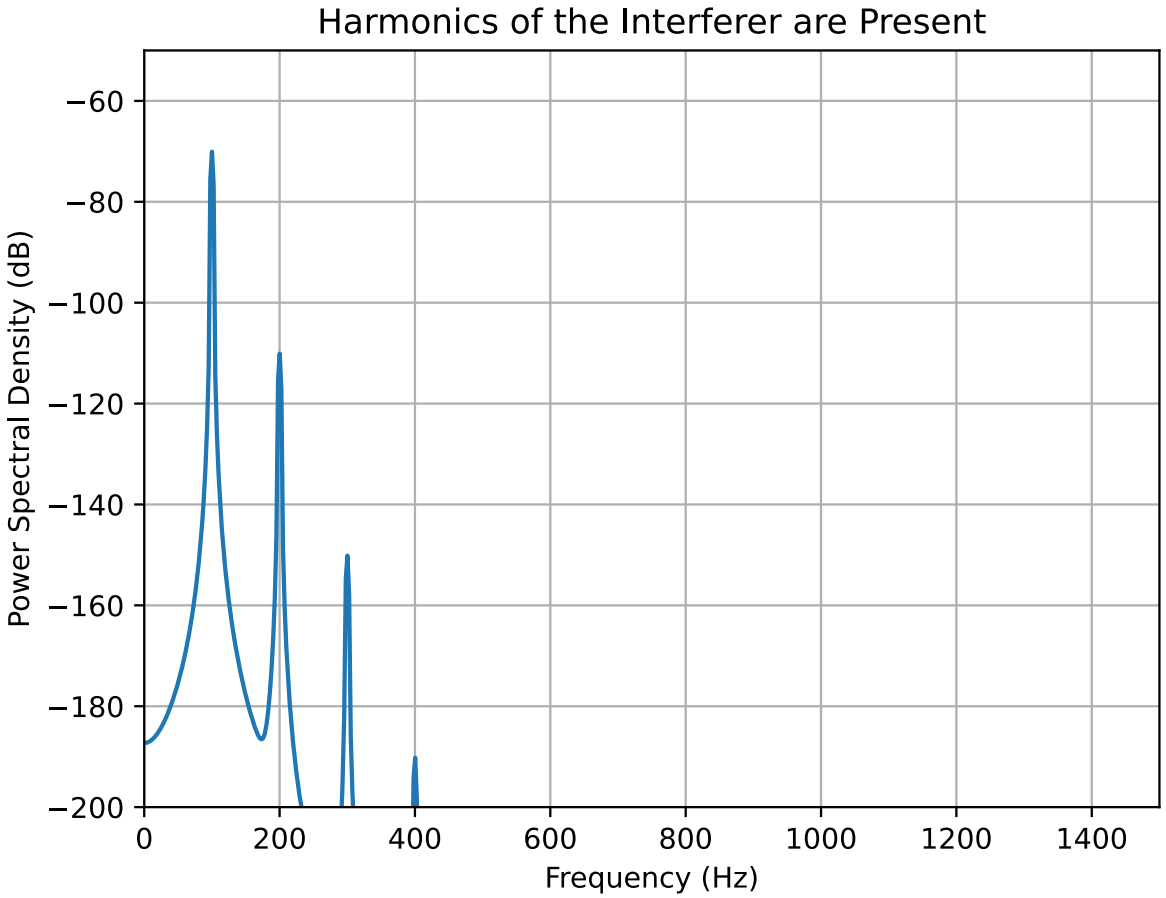
In [47]:
```
Pxr, fxr = ss.psd(xr,2**12,fs,scale_noise=False);
plot(fxr,10*log10(Pxr))
xlim([-1500,1500])
ylim([-100,0]);
ylabel(r'Power Spectral Density (dB)')
xlabel(r'Frequency (Hz)')
title(r'Baseband Signal + Jammer')
grid();
```




In [48]:
```
yD = discrim(xr)
plot(t[:200],yD[:200])
title(r'Discriminator Output')
ylabel(r'Amplitude')
xlabel(r'Time (s)')
grid();
```




In [59]:
```
yD = discrim(xr)
Pyd,fyd = ss.psd(yD,2**12,fs,scale_noise=False);
plot(fyd,10*log10(Pyd))
xlim([0,1500])
ylim([-200,-50]);
ylabel(r'Power Spectral Density (dB)')
xlabel(r'Frequency (Hz)')
title(r'Harmonics of the Interferer are Present')
grid();
```

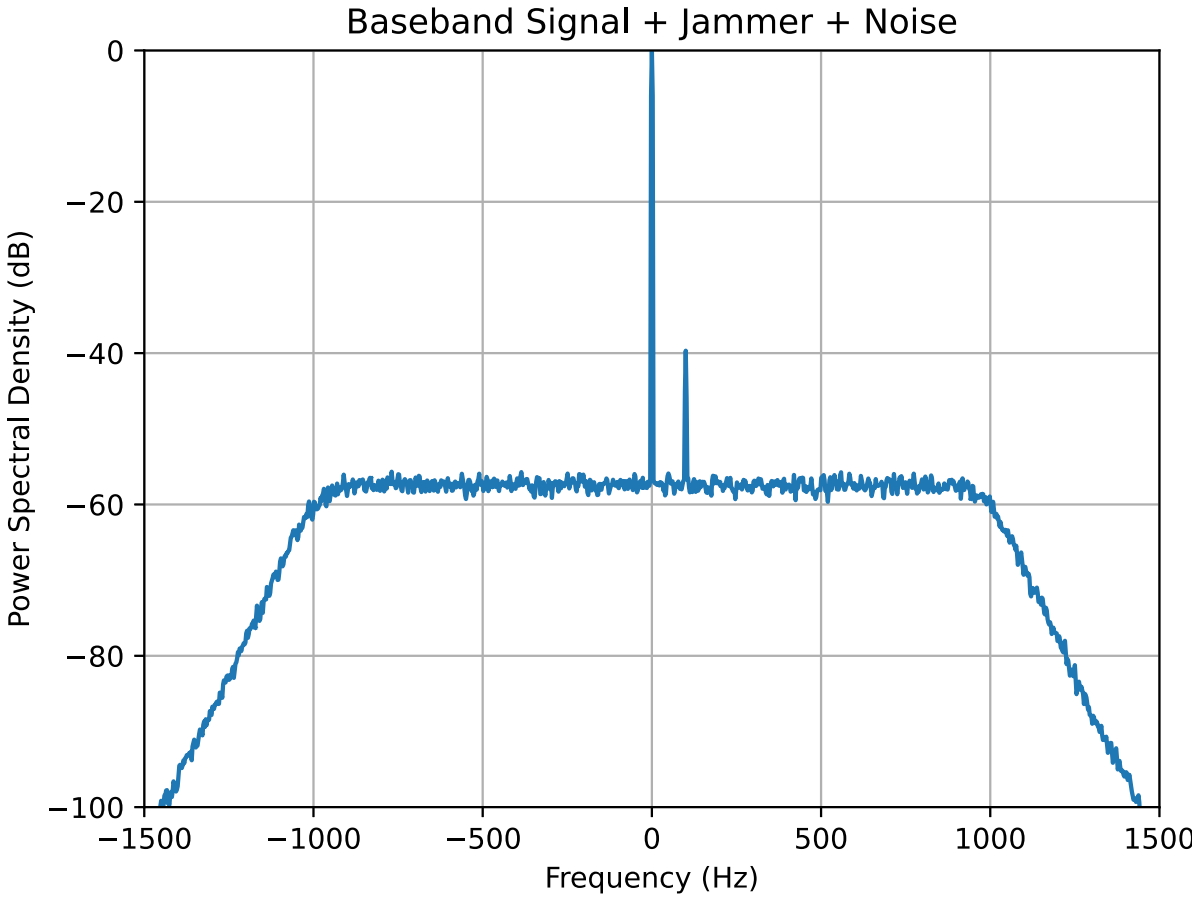



# Tone Plus Bandlimited Noise Interference

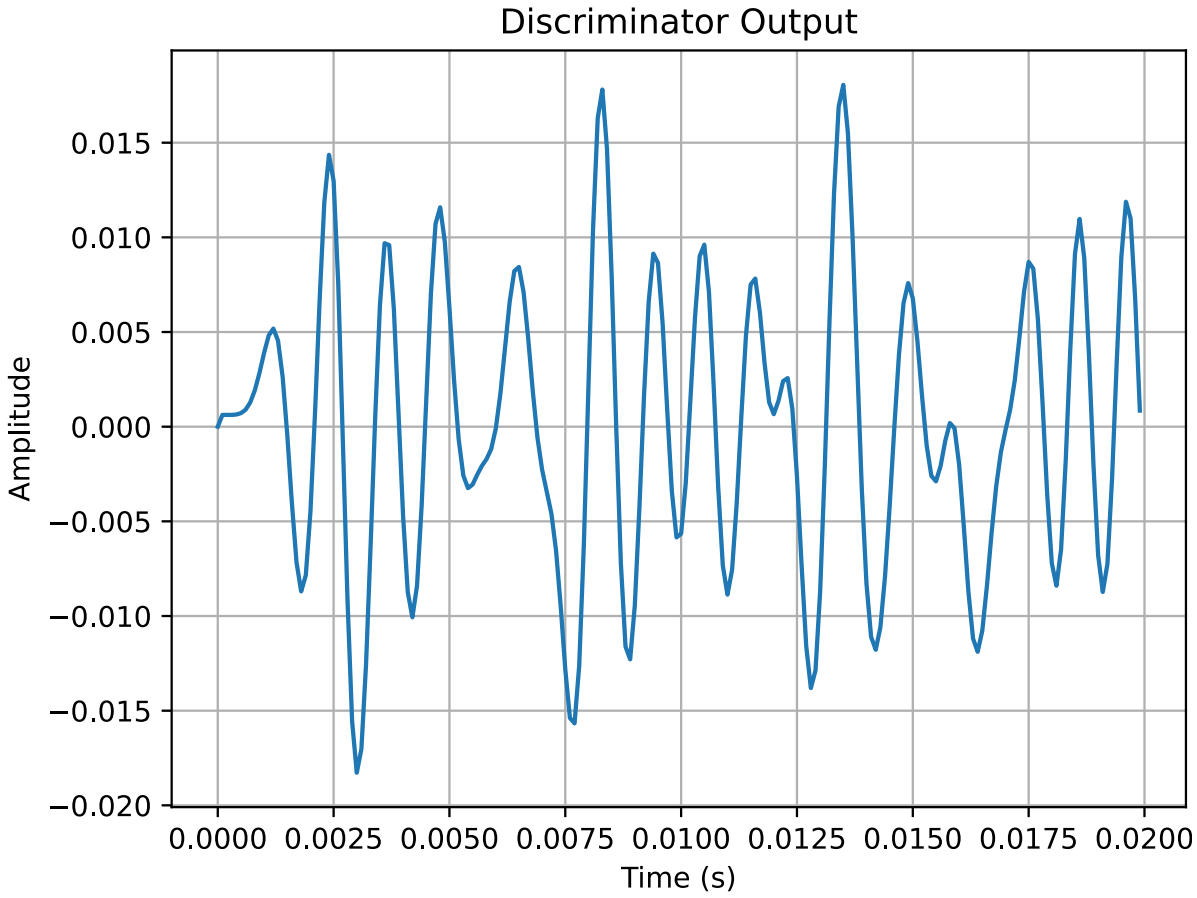
In [77]:
```
fs = 10000
t = arange(0,10,1/fs)
Ac = 1.0
Ai = 0.01
fi = 100 # in Hz
xr = 1 + Ai*exp(1j*2*pi*fi*t)
```

In [90]:
```
W = 1000
b,a = signal.butter(12,2*W/fs)
w = (random.randn(len(xr)) + 1j*random.randn(len(xr)))*.05
wf = signal.lfilter(b,a,w)
```

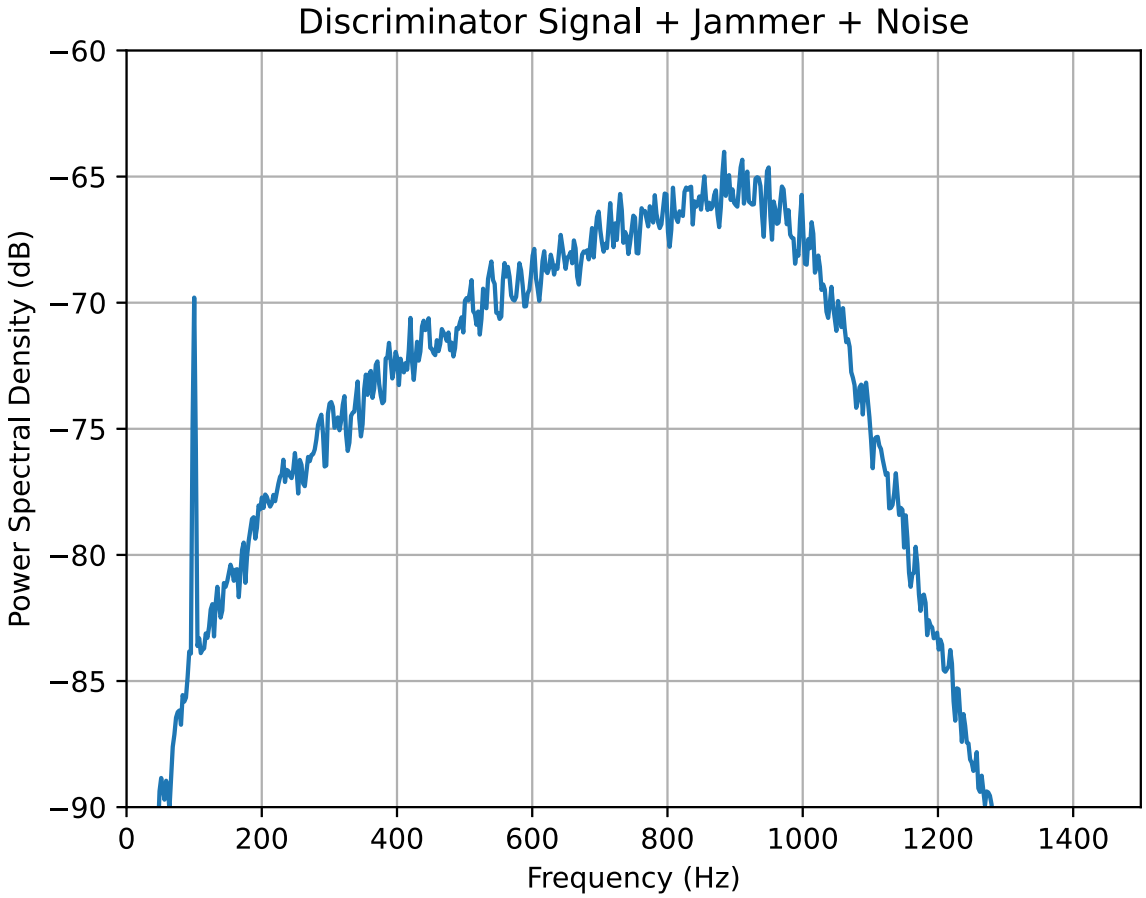
In [91]:
```
Pxr,fxr = ss.psd(xr+wf,2**12,fs,scale_noise=False);
plot(fxr,10*log10(Pxr))
xlim([-1500,1500])
ylim([-100,0]);
ylabel(r'Power Spectral Density (dB)')
xlabel(r'Frequency (Hz)')
title(r'Baseband Signal + Jammer + Noise')
grid();
```




In [92]:
```
yD = discrim(xr+wf)
plot(t[:200],yD[:200]);
title(r'Discriminator Output')
ylabel(r'Amplitude')
xlabel(r'Time (s)')
grid();
```




In [93]:
```
Pxr,fxr = ss.psd(discrim(xr+wf),2**12,fs,scale_noise=False);
plot(fxr,10*log10(Pxr))
title(r'Discriminator Output Spectrum')
xlim([0,1500])
ylim([-90,-60])
ylabel(r'Power Spectral Density (dB)')
xlabel(r'Frequency (Hz)')
title(r'Discriminator Signal + Jammer + Noise')
grid();
```


Discriminator Signal + Jammer + Noise
Power Spectral Density (dB)
−60
−65
−70
−75
−80
−85
−90
0
200
400
600
800
1000
1200
1400
Frequency (Hz)