# Advanced Data Comm

July 1, 2025

## 1 Advanced Data Communications

[2]:
```
%pylab inline
#%matplotlib qt
import sk_dsp_comm.sigsys as ss
import sk_dsp_comm.digitalcom as dc
import sk_dsp_comm.synchronization as synch
import scipy.signal as signal
from IPython.display import Image, SVG

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

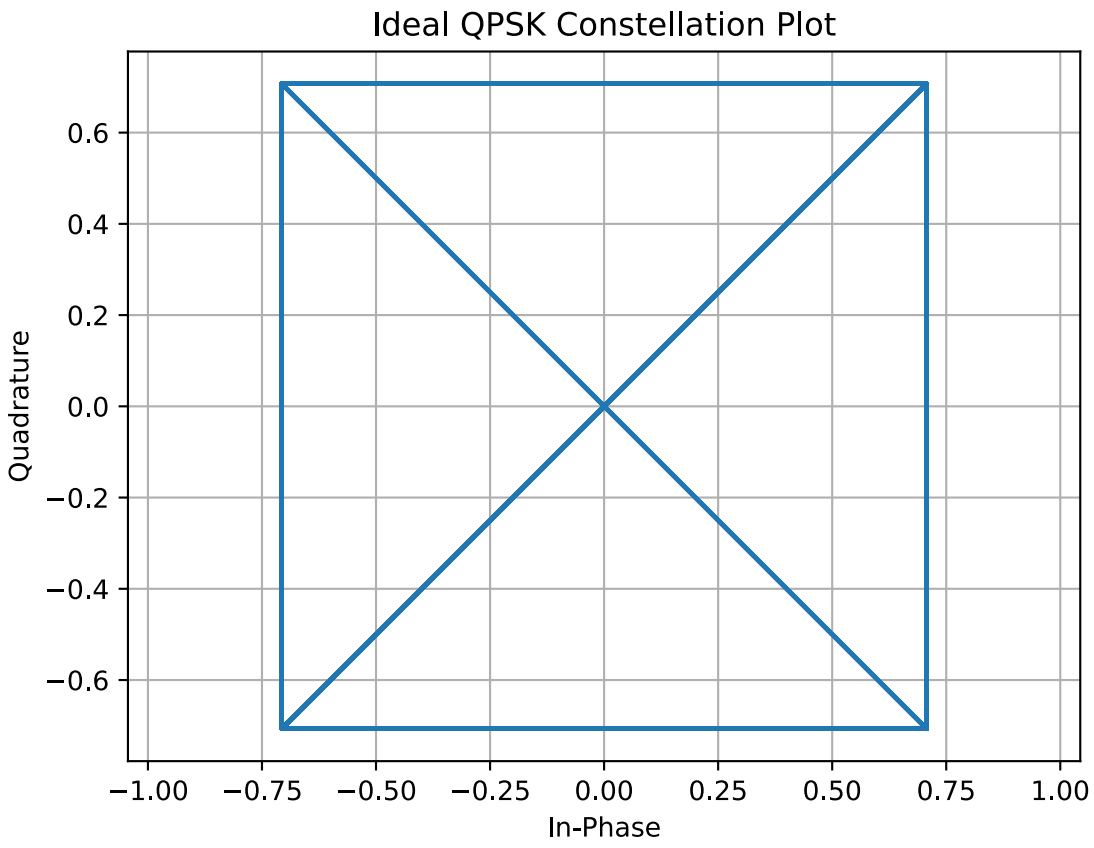
```
%pylab is deprecated, use %matplotlib inline and import the required libraries.
Populating the interactive namespace from numpy and matplotlib
```

[2]:
```
pylab.rcParams['savefig.dpi'] = 100 # default 72
#pylab.rcParams['figure.figsize'] = (6.0, 4.0) # default (6,4)
#%config InlineBackend.figure_formats=['png'] # default for inline viewing
%config InlineBackend.figure_formats=['svg'] # SVG inline viewing
#%config InlineBackend.figure_formats=['pdf'] # render pdf figs for LaTeX
```

### 1.1 QPSK with Filtering

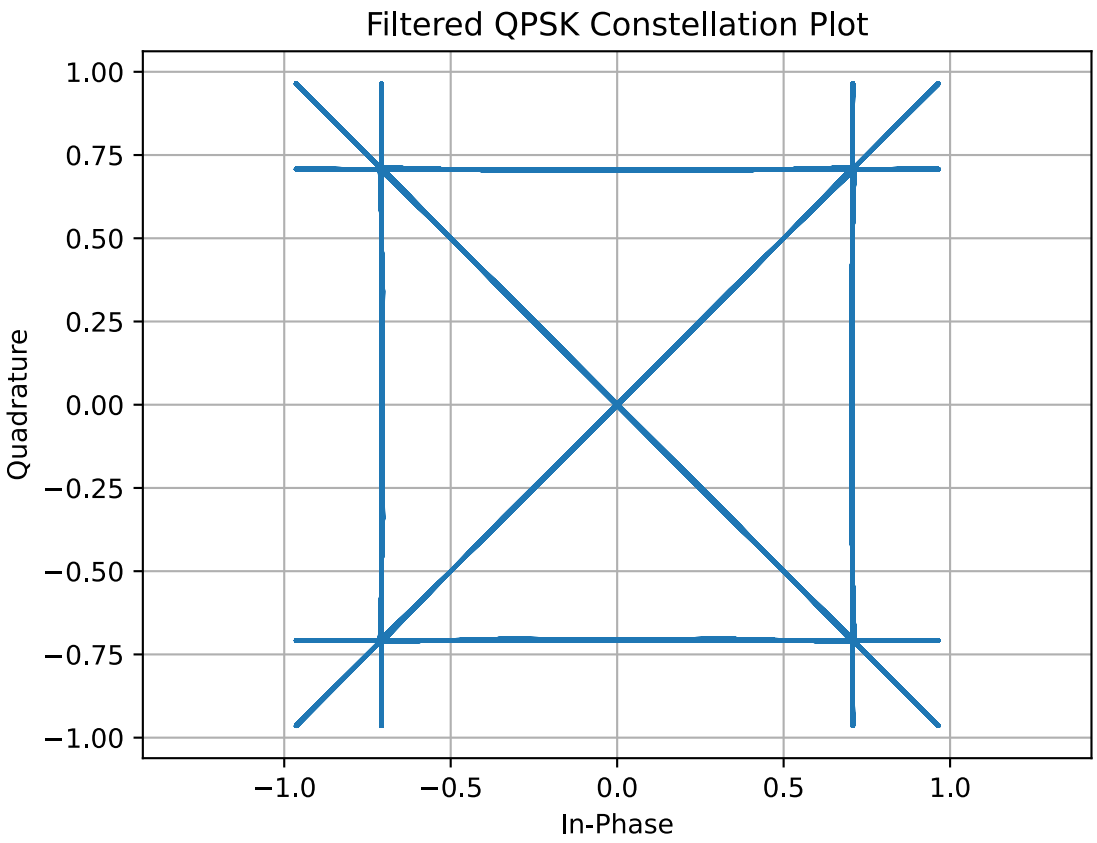
With QPSK the carrier phase takes four values, say $\theta_k \in \{\pm\pi/4, \pm 3\pi/4\}$. With traditional rectangular pulse shaping make abrumpt jumps evert $T_s$ seconds. * The QPSK *constellation plot* is shown below where the phase trajectories between the signal is obvious

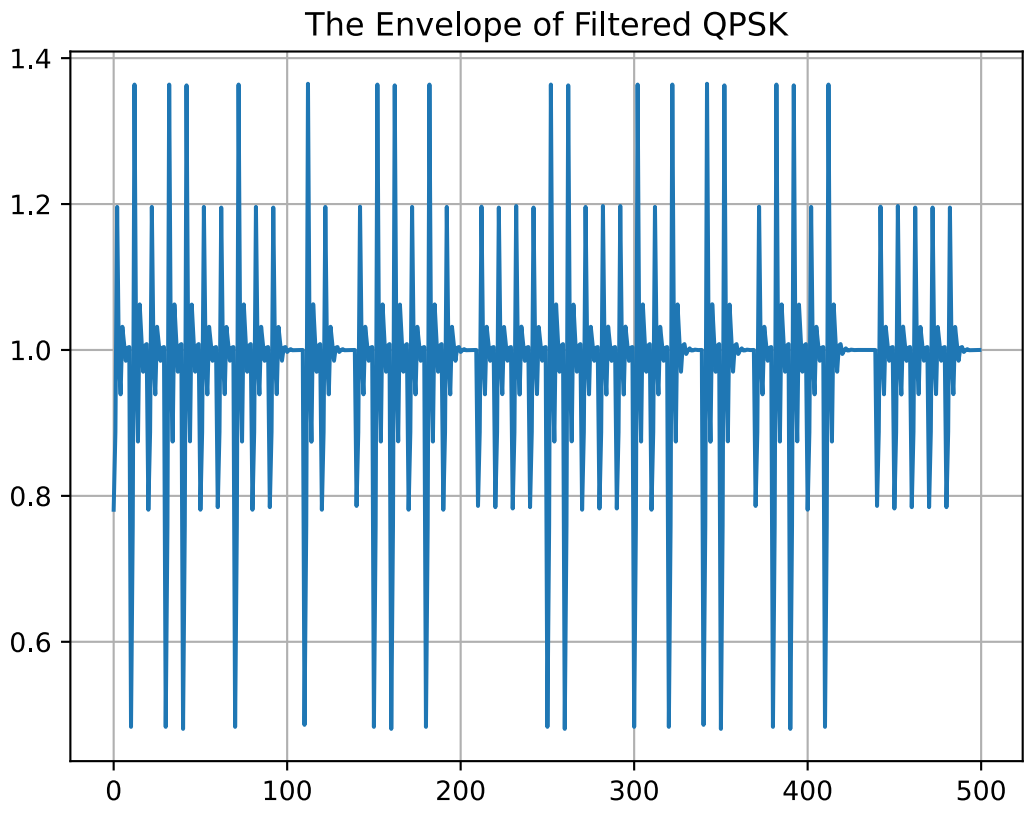
[6]:
```
x,b,data  = dc.mpsk_bb(1000,10,4,pulse='rect')
y = dc.cpx_awgn(x,20,10)
z = signal.lfilter(b,1,y)
#dc.eye_plot(y.real,2*10,12*10)
plot(x.real,x.imag)
grid()
title(r'Ideal QPSK Constellation Plot')
xlabel('In-Phase')
ylabel('Quadrature')
axis('equal');
```




- Idealy the waveform has a constant RF envelope
- If even a small amount of lowpass filtering is included the spectral occupancy gains made by the filtering can be easily lost if non-linear high power amplifiers (HPAs) are employed downstream
- The constellation plot shows that signal envelope is non-constant following the filtering

[8]:
```
b,a = signal.butter(3,2*1./10*3)
xf = signal.lfilter(b,a,x)
plot(xf.real,xf.imag)
title(r'Filtered QPSK Constellation Plot')
grid()
xlabel('In-Phase')
ylabel('Quadrature')
axis('equal');
figure()
plot(abs(xf[1000:1500]))
title(r'The Envelope of Filtered QPSK')
grid();
```

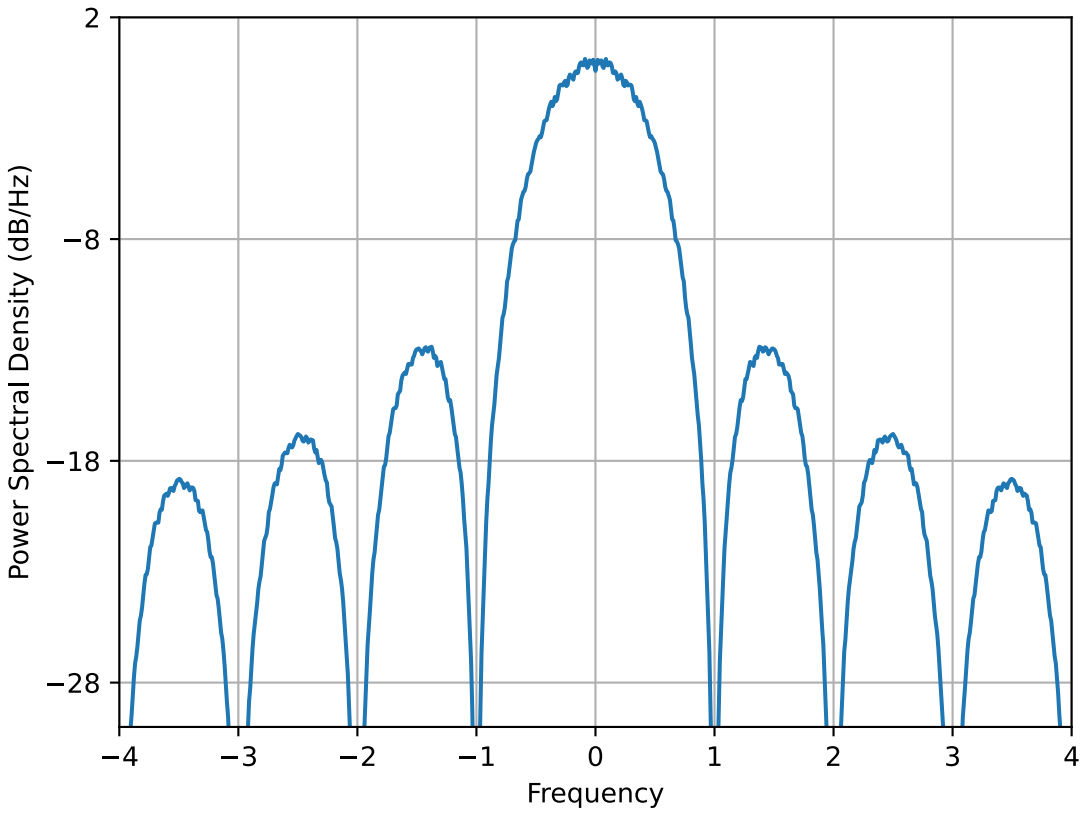

Filtered QPSK Constellation Plot
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Quadrature
−1.0
−0.5
0.0
0.5
1.0
In-Phase




- The non-constant envelope now seen make the signal vulnerable to spectral regrowth if non-linear amplification is employed

[16]:
```
x,b,data  = dc.mpsk_bb(100000,10,2,pulse='rect')
y = dc.cpx_awgn(x,20,10)
z = signal.lfilter(b,1,y)
psd(x,NFFT=2**10,Fs=10);
ylim([-30,2])
xlim([-4,4])
```

[16]: (-4.0, 4.0)

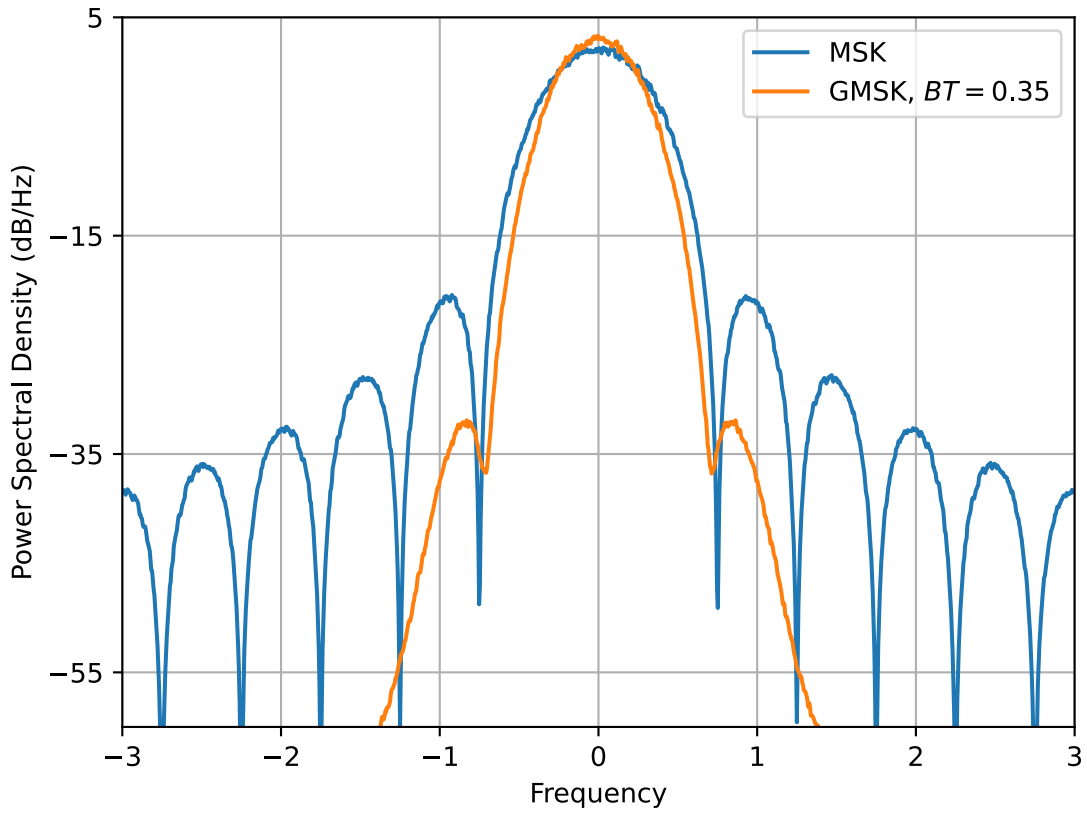



## 2 MSK/GMSK Complex Baseband Modulator

Implement a complex baseband MSK/GMSK modulator. The features of this modulator are similar to the MATLAB version found in the course notes. The function is housed in the module `digitalcom.py`.

[17]:
```
x_MSK, data = dc.gmsk_bb(100000,10,0,.35)
x_GMSK, data = dc.gmsk_bb(100000,10,1,.35)
```

[19]:
```
psd(x_MSK,NFFT=2**11,Fs=10)
psd(x_GMSK,NFFT=2**11,Fs=10);
ylim([-60,5])
legend(((r'MSK',r'GMSK, $BT = 0.35$')),
       loc='best').get_frame().set_alpha(0.8)
xlim([-3,3]);
```



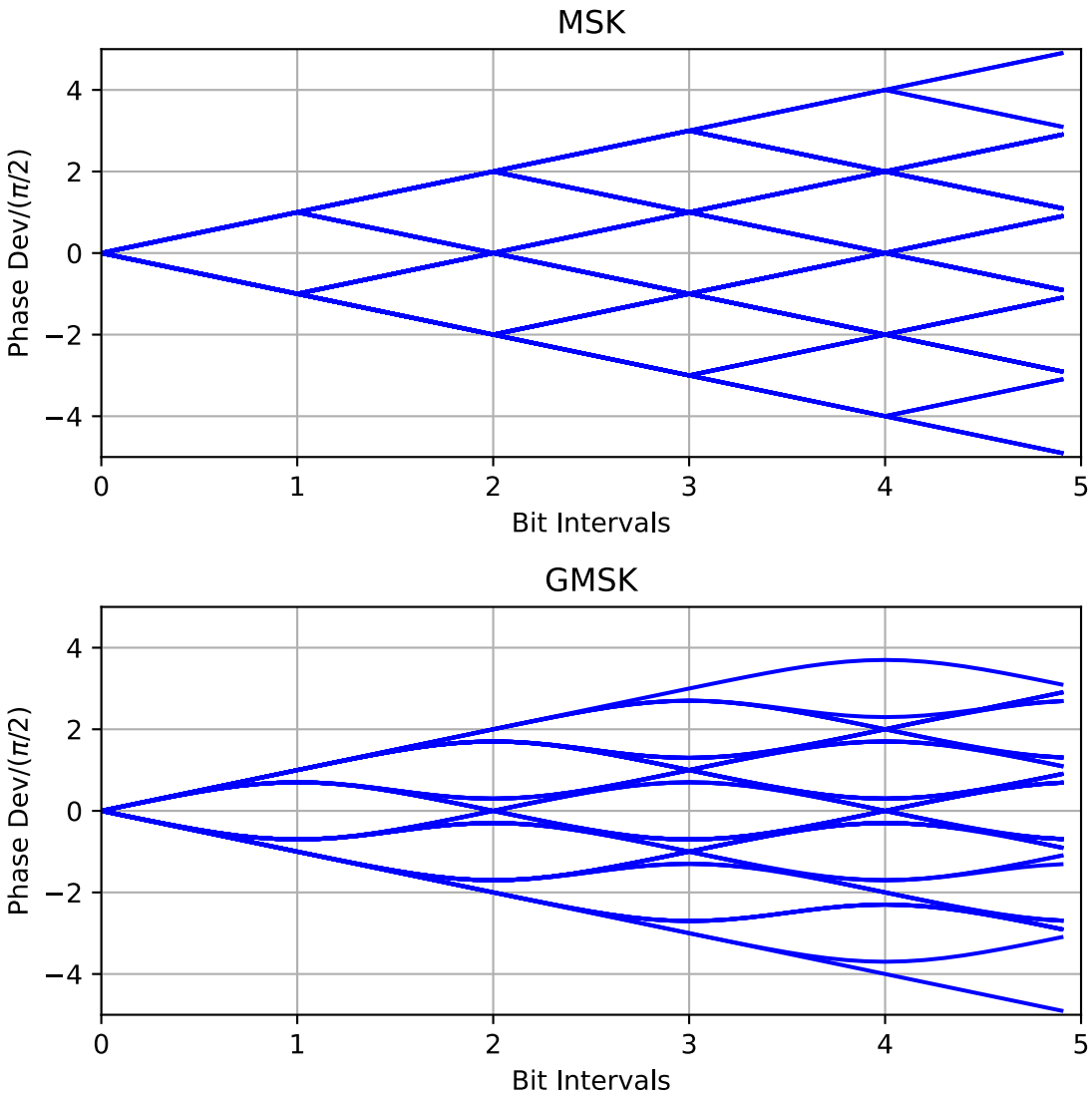

### 2.1 Phase Trellis Plot

[20]:
```
def trellis_plot(x,Ns):
    """
    trellis_plot(x,Ns)


    Mark Wickert June 2007
    coded to Python November 2016
    """
    # Trim to remove filter delay
    tau = Ns
    x = x[4*Ns:]
    K = len(x)/Ns
    M = 5
    NK = int(K//M)
    phase = unwrap(angle(x))
    #t = arange(M*Ns)/Ns
    flag = 0
    for i in range(NK):
        if i < NK-1:
```

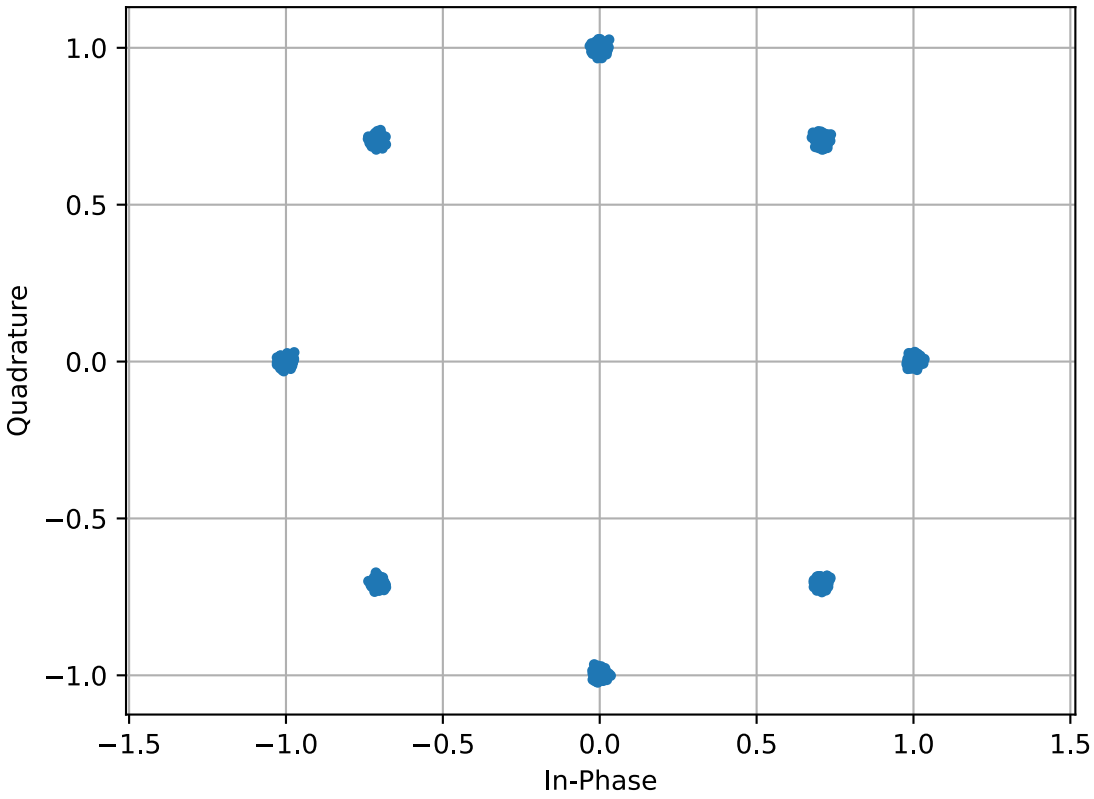
```
            phase1 = phase[i*M*Ns+(tau-1):(i+1)*M*Ns+(tau-1)]
        else:
            phase1 = phase[i*M*Ns+(tau-1):(i+1)*M*Ns+(tau-1)]
        t = arange(len(phase1))/Ns
        phase2 = (phase1-round(phase1[0]/(pi/2))*(pi/2))/(pi/2)
        if abs(phase2[0]) <= 1/10:
            plot(t,phase2,'b')
    axis([0, M, -M, M])
    xlabel(r'Bit Intervals')
    ylabel(r'Phase Dev/$(\pi/2)$')
    grid()
```

[21]:
```
figure(figsize=(6,6))
subplot(211)
trellis_plot(x_MSK[:6000],10)
title(r'MSK')
subplot(212)
trellis_plot(x_GMSK[:6000],10)
title(r'GMSK')
tight_layout()
```




## 3 M-ary MPSK

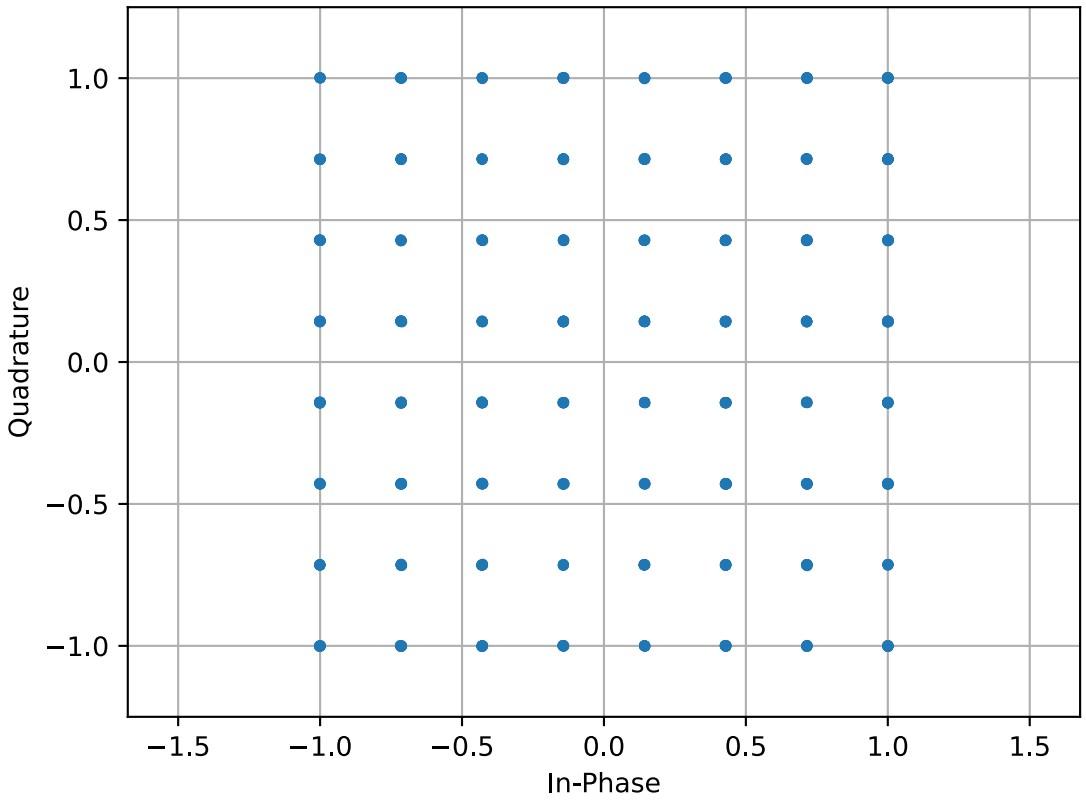
[22]:
```
x,b,data  = dc.mpsk_bb(1000,10,8,pulse='src') # 3rd param is M
y = dc.cpx_awgn(x,40,10)
z = signal.lfilter(b,1,y)
#dc.eye_plot(y.real,2*10,12*10)
plot(z[10*12::10].real,z[10*12::10].imag,'.')
grid()
xlabel('In-Phase')
ylabel('Quadrature')
axis('equal');
```




## 4 QAM Modulator

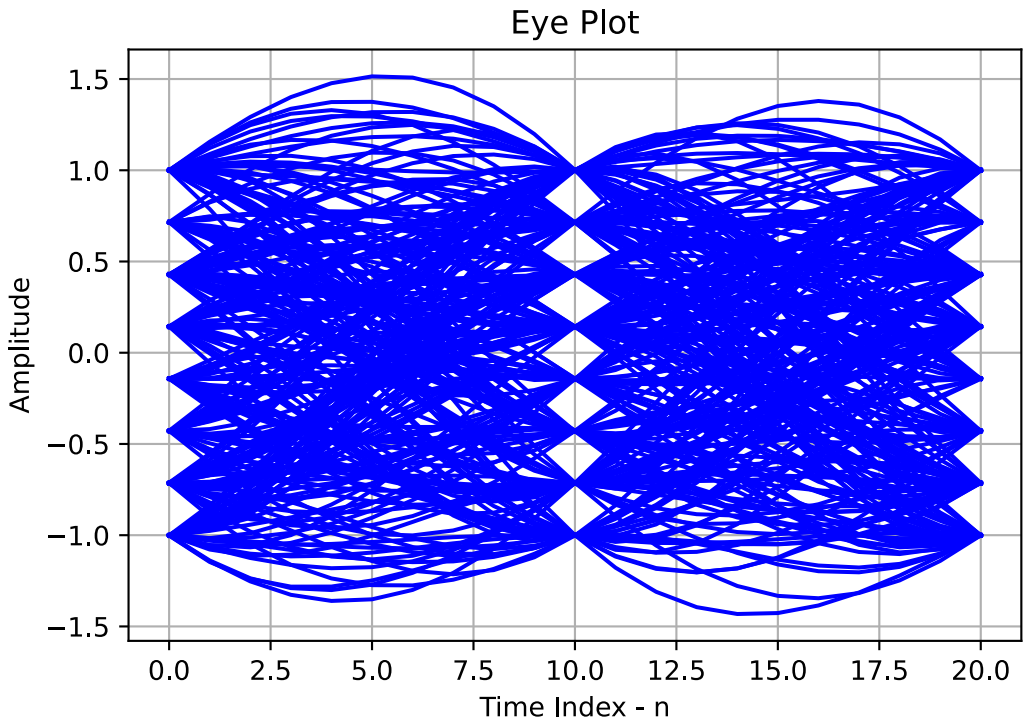
[23]:
```
x,b,tx_data = dc.qam_bb(1000,10,'64qam','src')
# Channel noise
x = dc.cpx_awgn(x,100,10)
# Matched filter
y = signal.lfilter(b,1,x)
plot(y[10+12*10::10].real,y[10+12*10::10].imag,'.')
grid()
xlabel('In-Phase')
ylabel('Quadrature')
axis('equal')
axis([-1.25,1.25,-1.25,1.25]);
```

```
Ignoring fixed x limits to fulfill fixed data aspect with adjustable data
limits.
```




[24]:
```
dc.eye_plot(y[:5000].real,20,12*10)
```

[24]: 0




## 5 Fractional Out-of-Band Power via Simulation

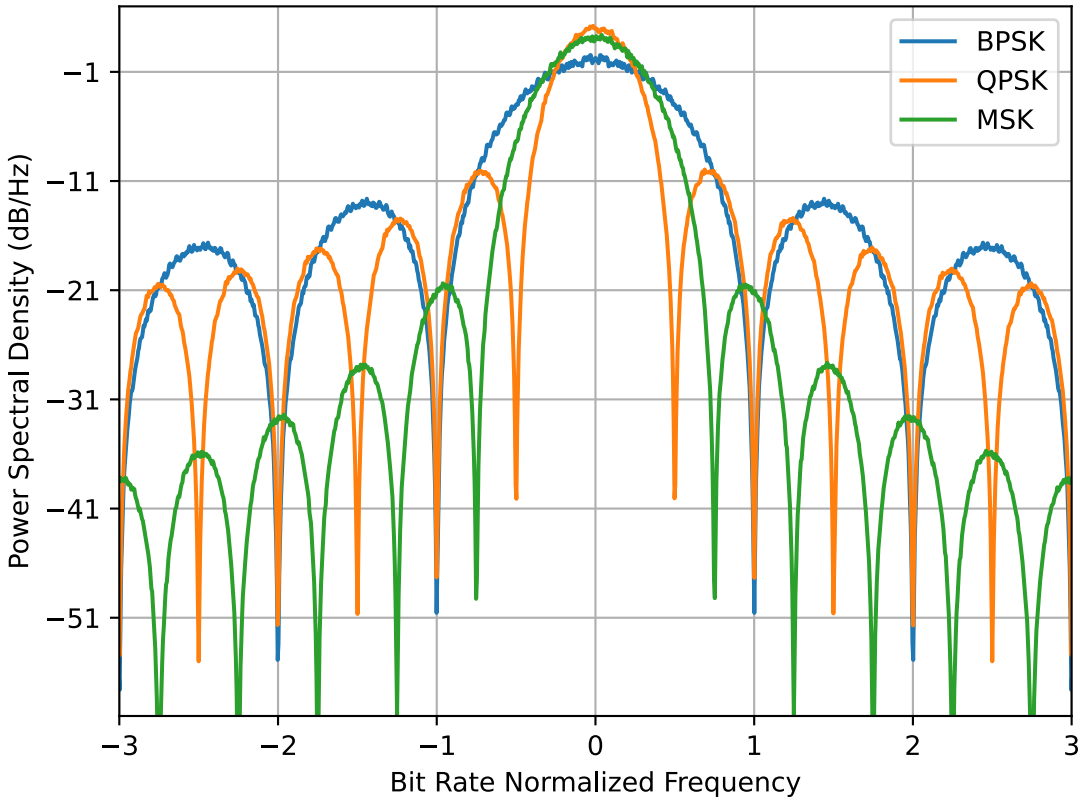
The fractional out-of-band power, $\Delta P_{OB}$, can be numerically calculated using either an analytic for a particular modulation scheme or complex baseband simulation results. Here I will use simulation results. I will compare BPSK, QPSK, and MSK.

To get started you need to generate about 100000 symbols at say 10 samples per symbol for a particular modulation scheme. You then need to estimate the power spectral density (PSD) using the PyLab function `Pxx, f = psd(, return_line=False)` with the option return_line set to false, so yo can get your hands the PSD value array and frequency axis value array.

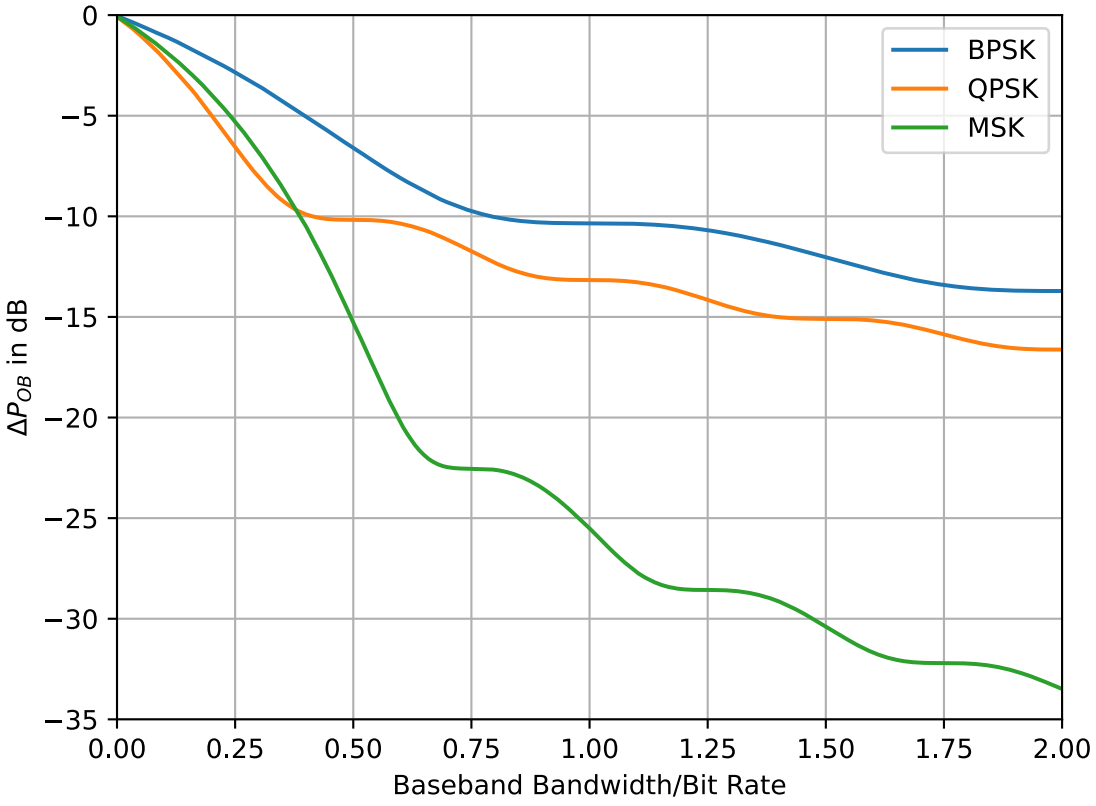
[25]:
```
# Ns = 10 samples per bit since M = 2
x_BPSK,b,data = dc.mpsk_bb(100000,10,2,pulse='rect')
# Ns = 10 samples per bit since M = 4
x_QPSK,b,data = dc.mpsk_bb(100000,20,4,pulse='rect')
# Ns = 10 samples per bit by design of modulator
x_MSK,data = dc.gmsk_bb(100000,10,0,.35) # Ns is samples per bit
```

[27]:
```
Pxx_BPSK, fb = psd(x_BPSK,NFFT=2**11,Fs=10,return_line=False);
Pxx_QPSK, fq = psd(x_QPSK,NFFT=2**11,Fs=10,return_line=False);
Pxx_MSK, fm = psd(x_MSK,NFFT=2**11,Fs=10,return_line=False);
xlim([-3,3])
ylim([-60,5])
```

```
legend(('BPSK','QPSK','MSK')),loc='best')
xlabel('Bit Rate Normalized Frequency');
```




[28]:
```
plot(fb[len(fb)//2:-1],10*log10(1-cumsum(Pxx_BPSK[len(fb)//2:-1])\
                                /sum(Pxx_BPSK[len(fb)//2:])))
plot(fq[len(fq)//2:-1],10*log10(1-cumsum(Pxx_QPSK[len(fq)//2:-1])\
                                /sum(Pxx_QPSK[len(fq)//2:])))
plot(fm[len(fm)//2:-1],10*log10(1-cumsum(Pxx_MSK[len(fq)//2:-1])\
                                /sum(Pxx_MSK[len(fq)//2:])))
grid()
legend(('BPSK','QPSK','MSK')),loc='best')
ylabel(r'$\Delta P_{OB}$ in dB')
xlabel('Baseband Bandwidth/Bit Rate')
xlim([0,2])
ylim([-35,0]);
```




To get a detailed numerical value, you may want to switch the notebook mode from

`%pylab inline`

to

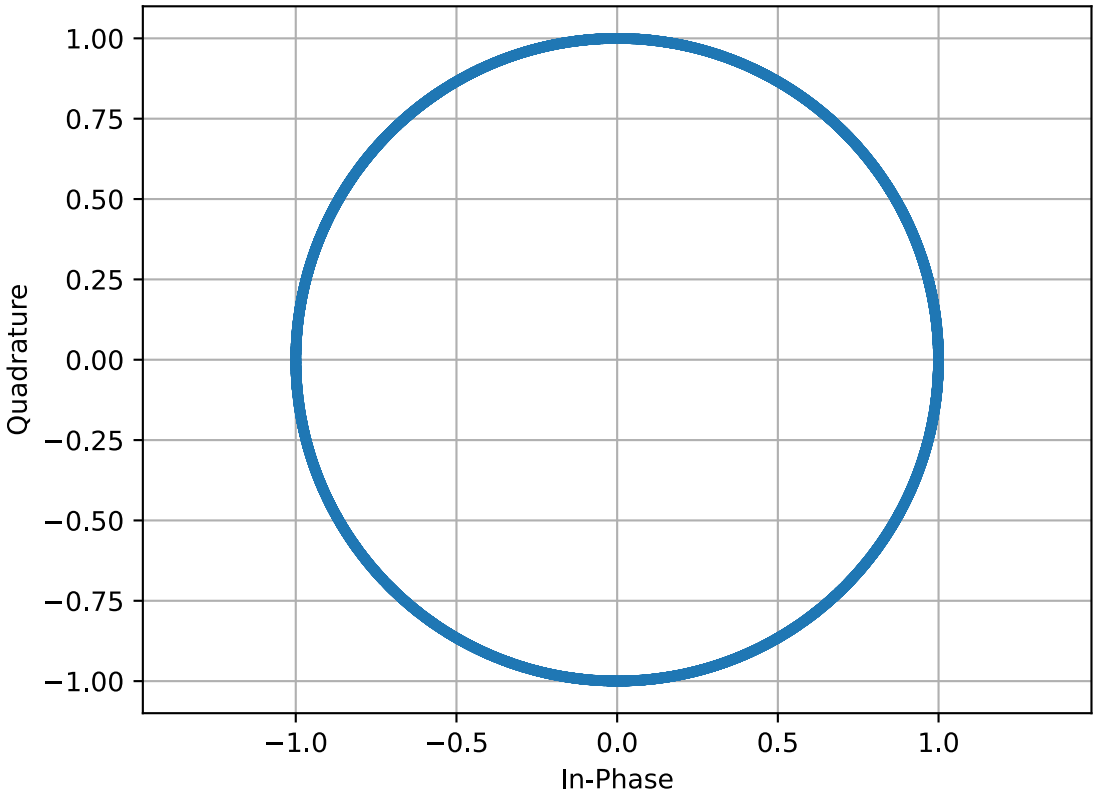
`%pylab qt`

This way the plot window will appear as a pop-up window and you can use the x-y value display to mark a point.

## 6 Synchronization

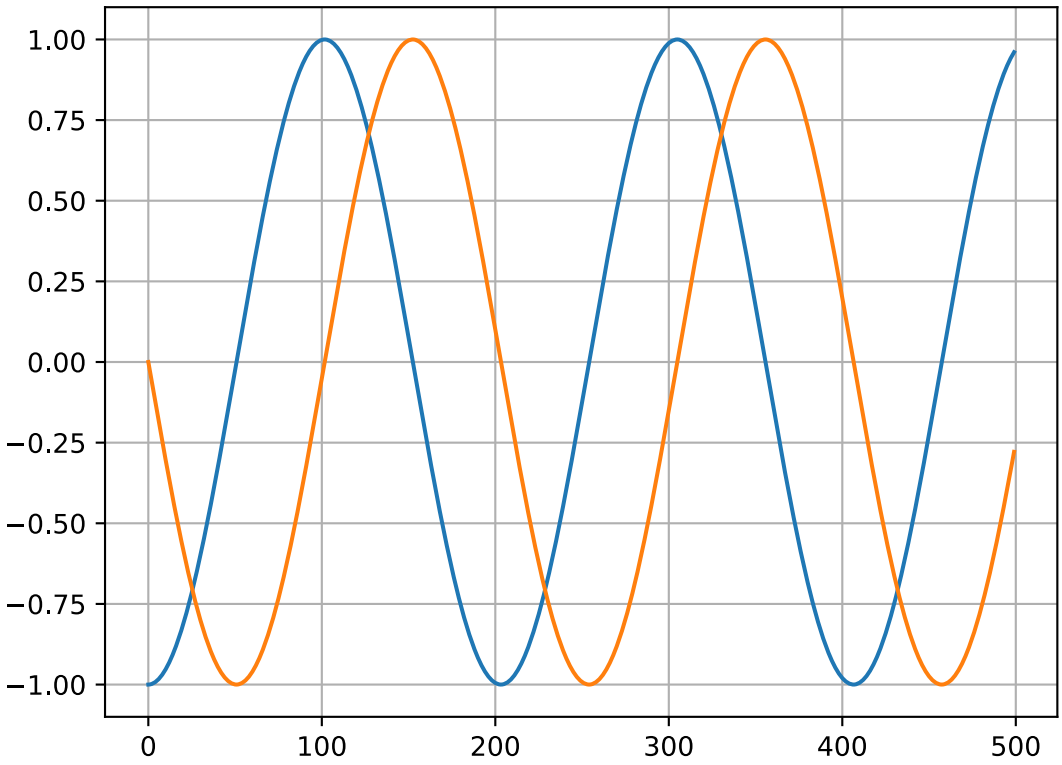
[31]:
```
x_qpsk,b,data = dc.mpsk_bb(10000,10,4)
n = arange(0,len(x_qpsk))
y_qpsk = x_qpsk*exp(1j*2*pi*0.00123*n)
#psd(y_qpsk,2**10,10);
z_qpsk = signal.lfilter(b,1,y_qpsk)
plot(z_qpsk[9::10].real,z_qpsk[9::10].imag,'.')
grid()
xlabel('In-Phase')
ylabel('Quadrature')
```

```
axis('equal');
# axis([-1.25,1.25,-1.25,1.25]);
```



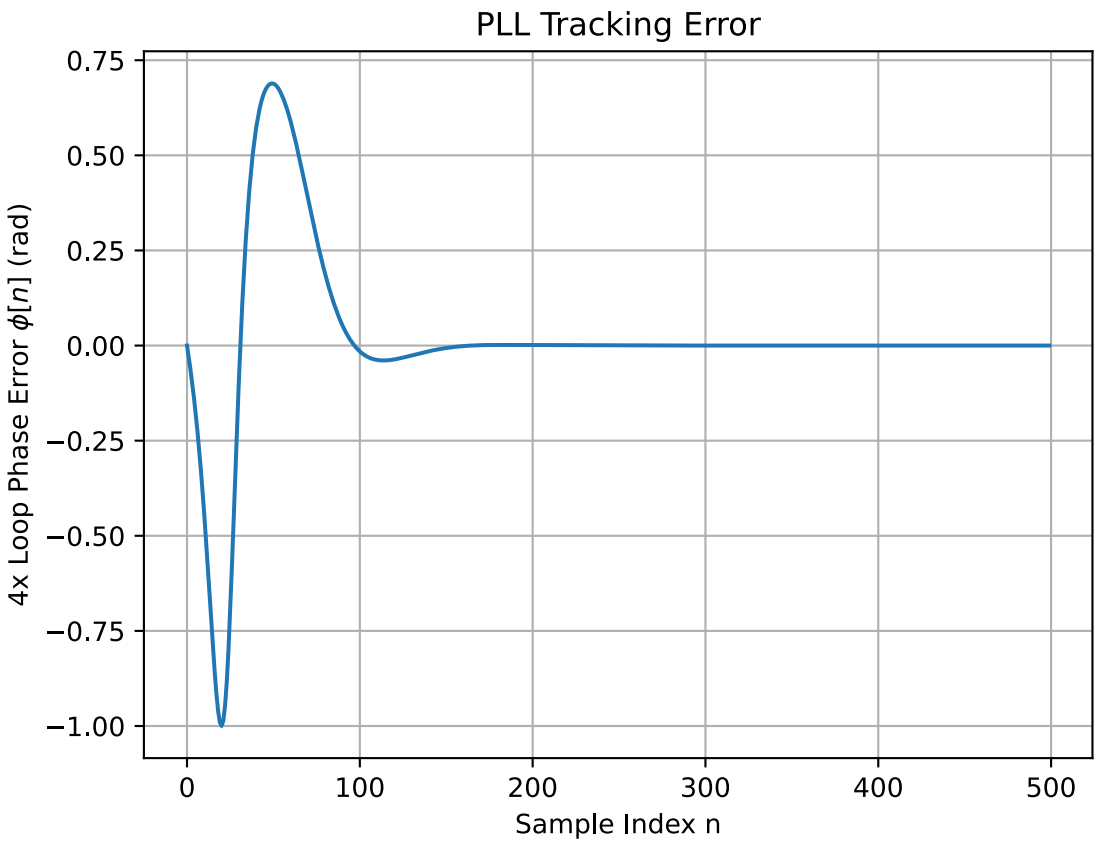

[32]:
```
z_sync = y_qpsk**4
plot(z_sync[:500].real)
plot(z_sync[:500].imag)

grid();
```




### 6.1 Implement a Times Four PLL

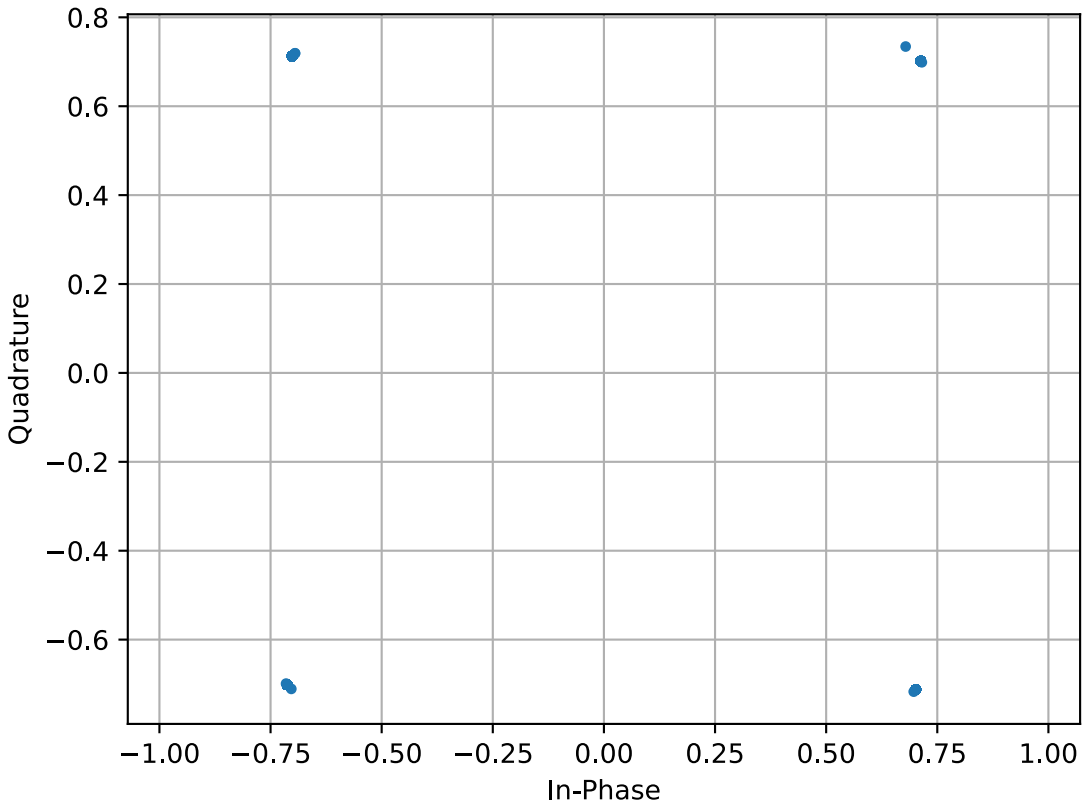
[34]:
```
z_sync = y_qpsk**4
# Initally test the loop with just a pure complex sinusoid
#theta_hat, ev, phi  = synch.PLL_cbb(exp(1j*2*pi*.05*n),10,2,1,.05,0.707)
# Test with the 4th power QPSK signal
theta_hat, ev, phi  = synch.PLL_cbb(z_sync,10,2,1,.1,0.707)
plot(phi[:500])
title('PLL Tracking Error')
ylabel(r'4x Loop Phase Error $\phi[n]$ (rad)')
xlabel('Sample Index n')
grid();
```




### 6.2 Demodulate Using the Tracked Phase/4

[36]:
```
z_recover = y_qpsk*exp(-1j*theta_hat/4)*exp(1j*pi/4)
#z_recover = x_qpsk
# Wait for the transient to decay before plotting scatter points
plot(z_recover[70+9::10].real,z_recover[70+9::10].imag,'.')
grid()
xlabel('In-Phase')
ylabel('Quadrature')
axis('equal')
# axis([-1.25,1.25,-1.25,1.25]);
```

[36]:
```
(np.float64(-0.7866495608999127),
 np.float64(0.7867758597555037),
 np.float64(-0.7896589473730499),
 np.float64(0.8068596885841082))
```

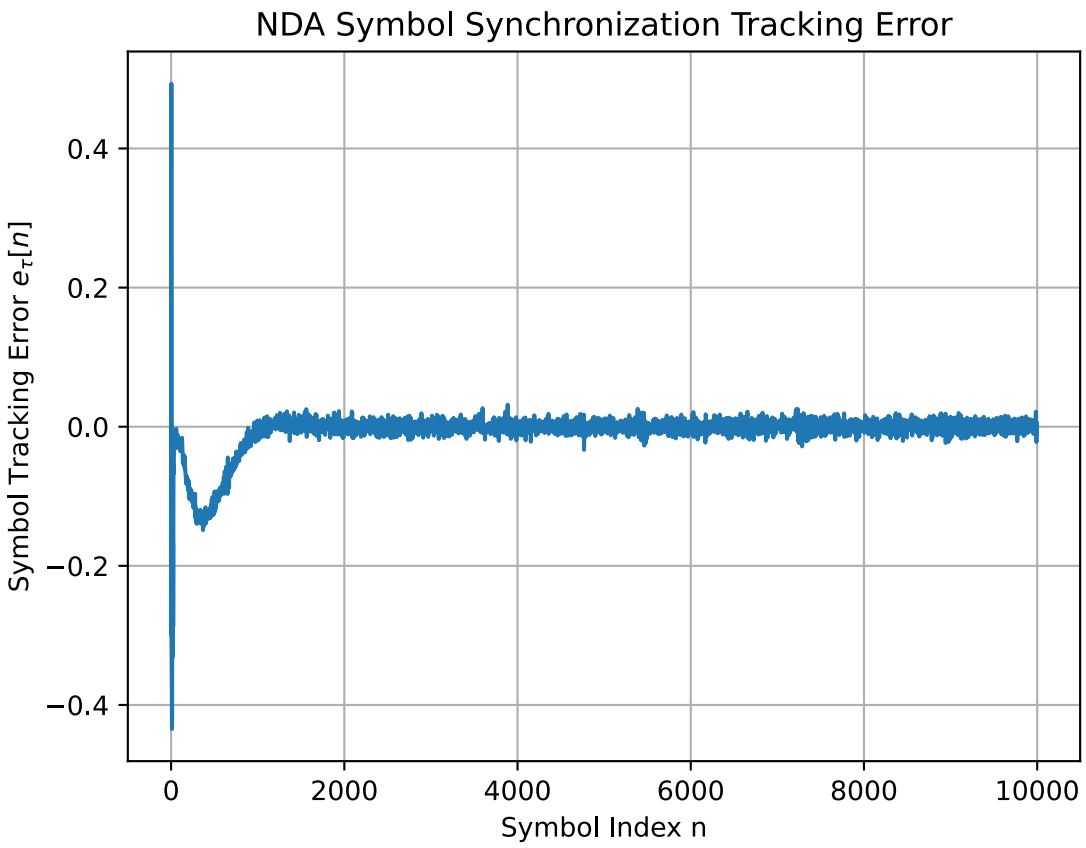



### 6.3 Symbol Synchronization (no Phase Error)

Consider QPSK with SRC pulse shaping and timing error. A second-order tracking loop is used with the loop bandwidth $T_s \cdot B_n = 0.002$.

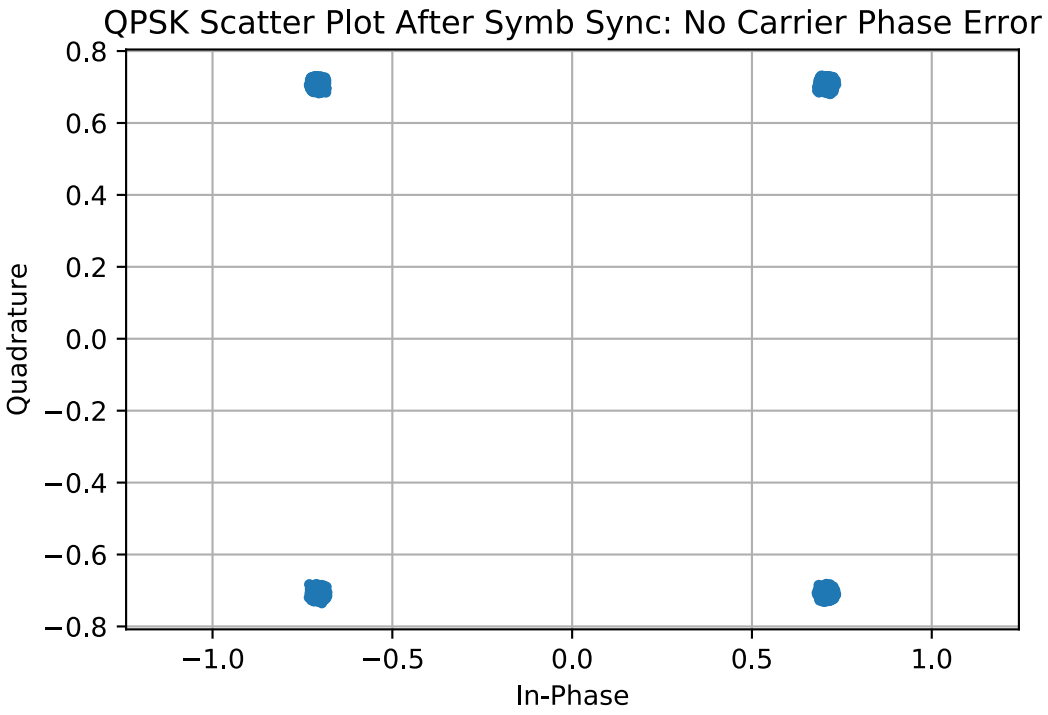
[38]:
```
x_b,b_b,data = dc.mpsk_bb(10000,4,4,'src')
y_b = dc.farrow_resample(x_b,1.001,1)
z_b = signal.lfilter(b_b,1,y_b)
zz, e_tau = synch.NDA_symb_sync(z_b,4,50,0.002,zeta=0.707,I_ord=3)
```

[39]:
```
plot(e_tau)
title('NDA Symbol Synchronization Tracking Error')
ylabel(r'Symbol Tracking Error $e_\tau[n]$')
xlabel('Symbol Index n')
grid();
```




As expected the loop locks and settles in about 1000 symbols.

[17]:
```
plot(zz[2000:].real,zz[2000:].imag,'.')
title('QPSK Scatter Plot After Symb Sync: No Carrier Phase Error')
xlabel('In-Phase')
ylabel('Quadrature')
axis('equal')
grid();
```

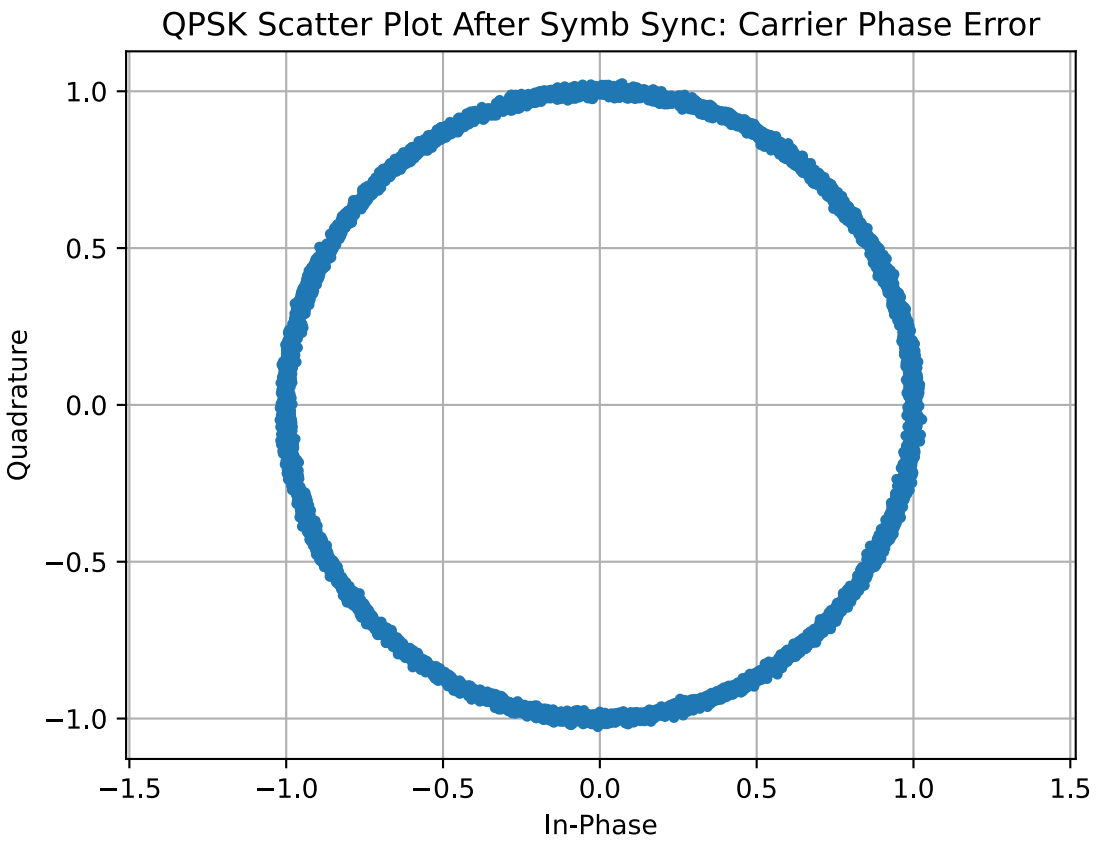



### 6.4 Timing Error Plus Phase/Frequency Error

Now we consider sampling clock error as well as carrier phase/frequency error.

[40]:
```
x_b,b_b,data = dc.mpsk_bb(10000,4,4,'src')
n = arange(0,len(x_b))
y_b = dc.farrow_resample(x_b*exp(1j*2*pi*0.0001*n),1.001,1)
z_b = signal.lfilter(b_b,1,y_b)
zz, e_tau = synch.NDA_symb_sync(z_b,4,50,0.002,zeta=0.707,I_ord=3)
```

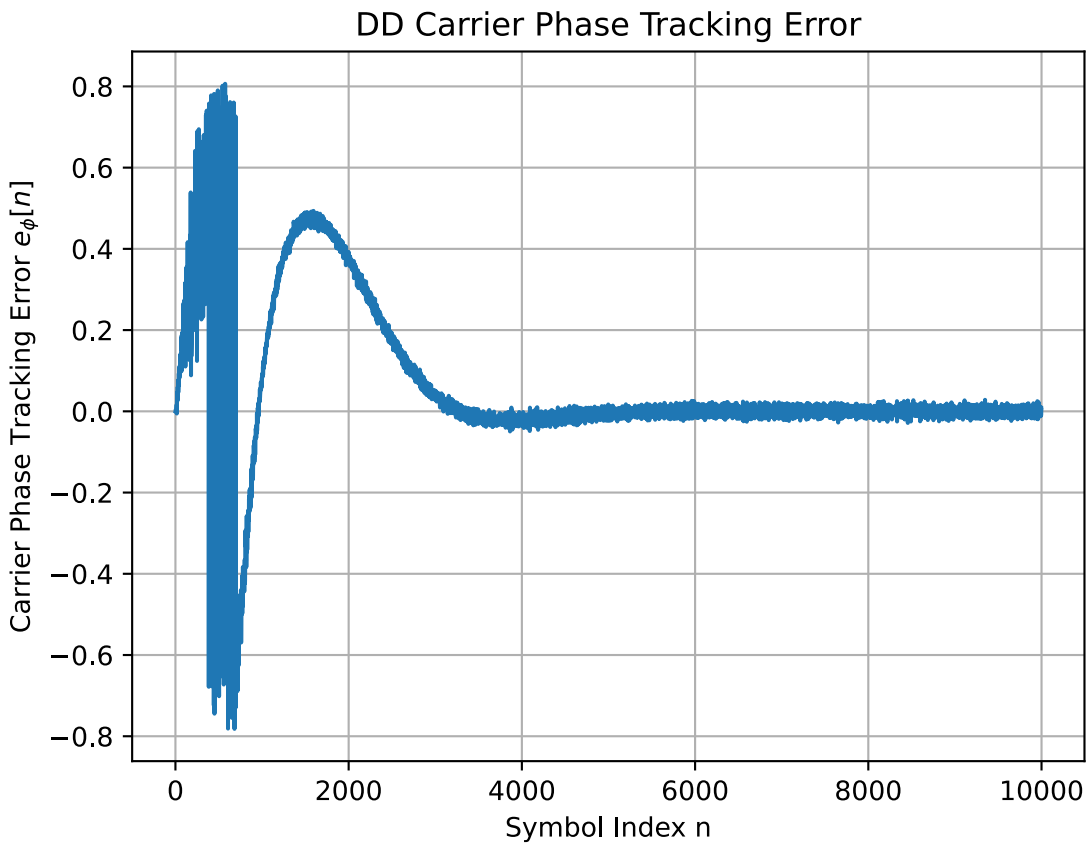
[41]:
```
plot(zz[2000:].real,zz[2000:].imag,'.')
title('QPSK Scatter Plot After Symb Sync: Carrier Phase Error')
xlabel('In-Phase')
ylabel('Quadrature')
axis('equal')
grid();
```



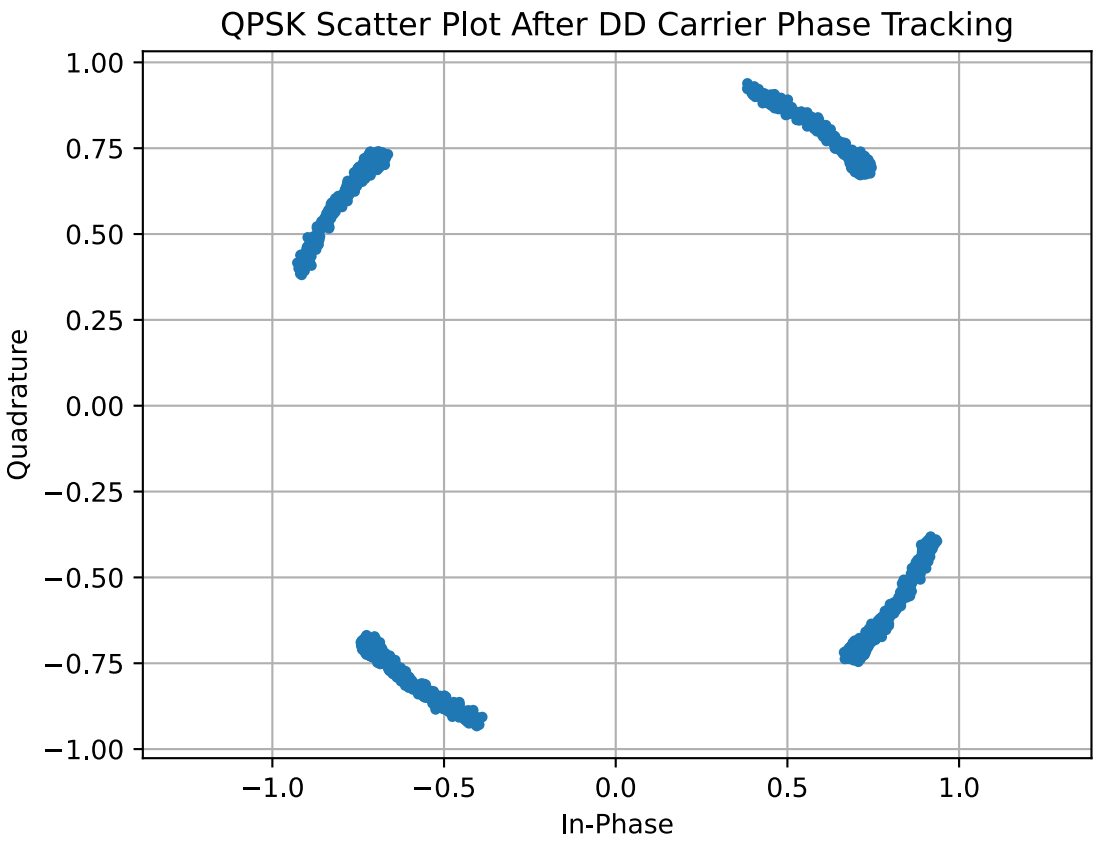

The symbol timing is working, but the phase/frequency error remains by virtue of the circular spread of the scatter plot. We send the one sample per symbol output of the symbol timing loop on to a decision-directed phase tracking algorithm

[42]:
```
z_prime, a_hat, e_phi, theta_h = \
            synch.DD_carrier_sync(zz,4,0.001,zeta=0.707,type=0)
```

[43]:
```
plot(e_phi)
title('DD Carrier Phase Tracking Error')
ylabel(r'Carrier Phase Tracking Error $e_\phi[n]$')
xlabel('Symbol Index n')
grid();
```




[44]:
```
plot(z_prime[2000:].real,z_prime[2000:].imag,'.')
title('QPSK Scatter Plot After DD Carrier Phase Tracking')
xlabel('In-Phase')
ylabel('Quadrature')
axis('equal')
grid();
```

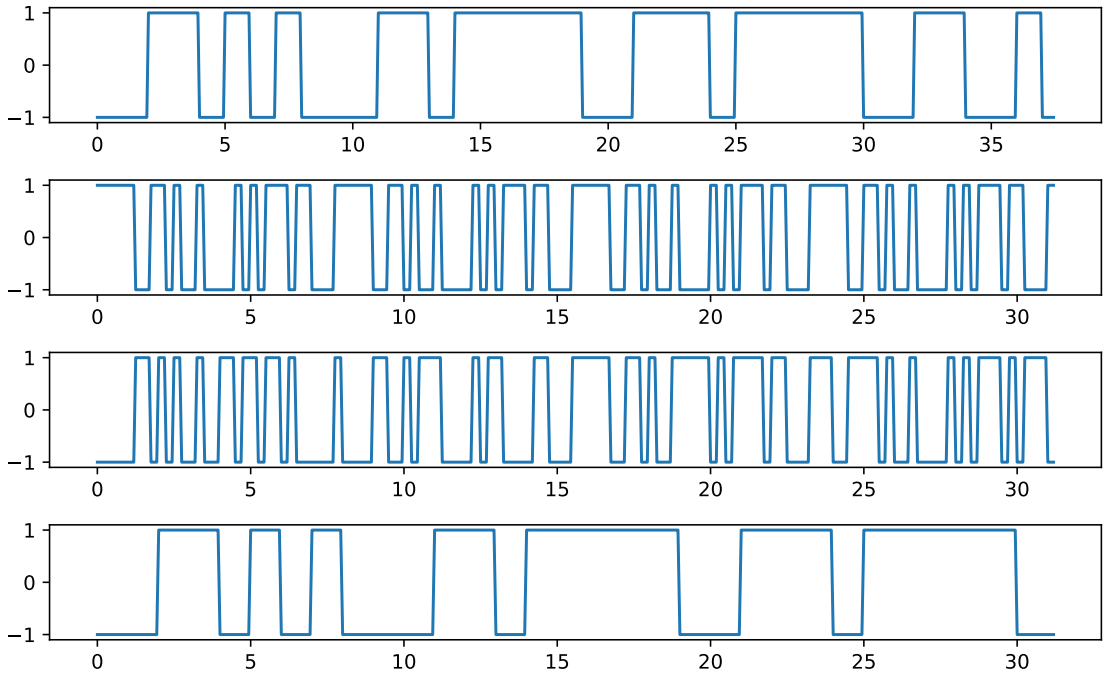



## 7 Spread Spectrum

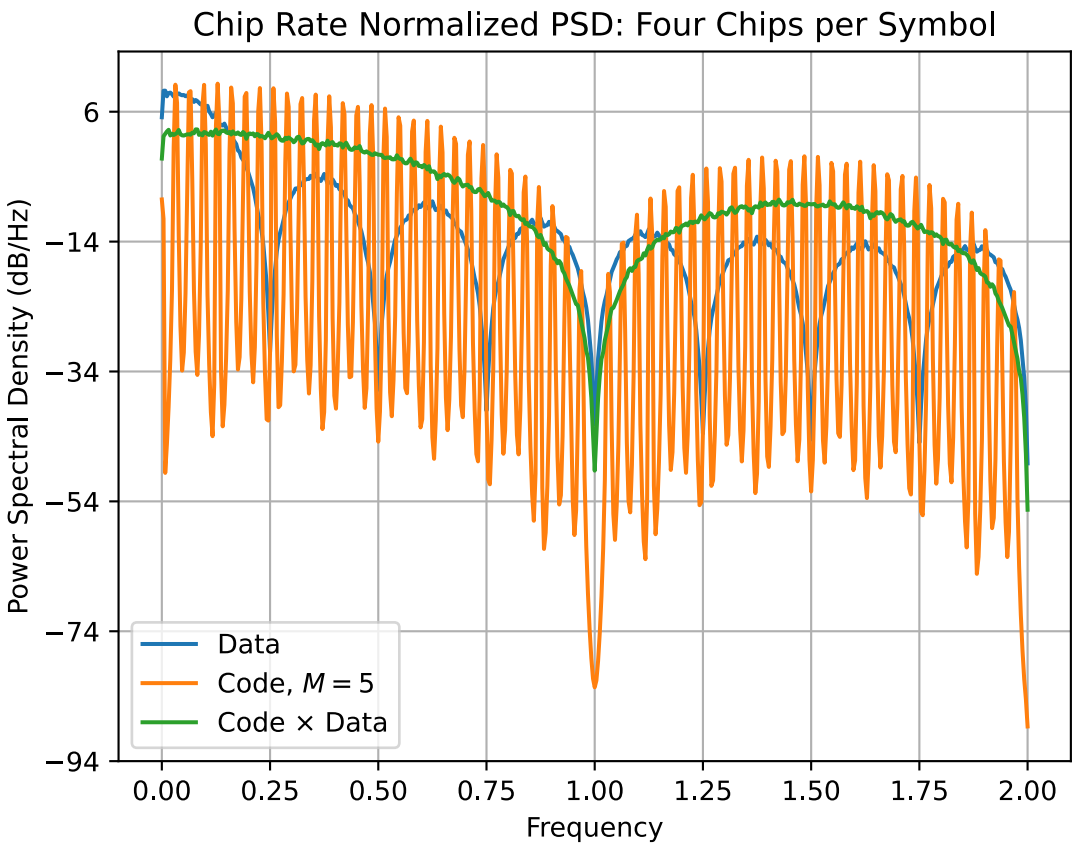
Model a simple direct sequence system usiing a recctangular pulse shape. The numer of chips per bit is four.

[47]:
```
figure(figsize=(8,5))
c,b = dc.nrz_bits2(dc.pn_gen(4*10000,5),4)
d,b,data = dc.nrz_bits(10000,16)
N_disp = 600
n = arange(0,len(c))/16
subplot(411)
plot(n[:N_disp],d[:N_disp])
subplot(412)
plot(n[:500],c[:500])
subplot(413)
cd = c*d
plot(n[:500],cd[:500])
subplot(414)
ccd = c*cd
```

```
plot(n[:500],ccd[:500]);
tight_layout()
```




[48]:
```
psd(d,NFFT=2**10,Fs=4);
psd(c,NFFT=2**10,Fs=4);
psd(cd,NFFT=2**10,Fs=4);
title('Chip Rate Normalized PSD: Four Chips per Symbol')
legend(('Data','Code, $M=5$',r'Code $\times$ Data'),loc='best');
```

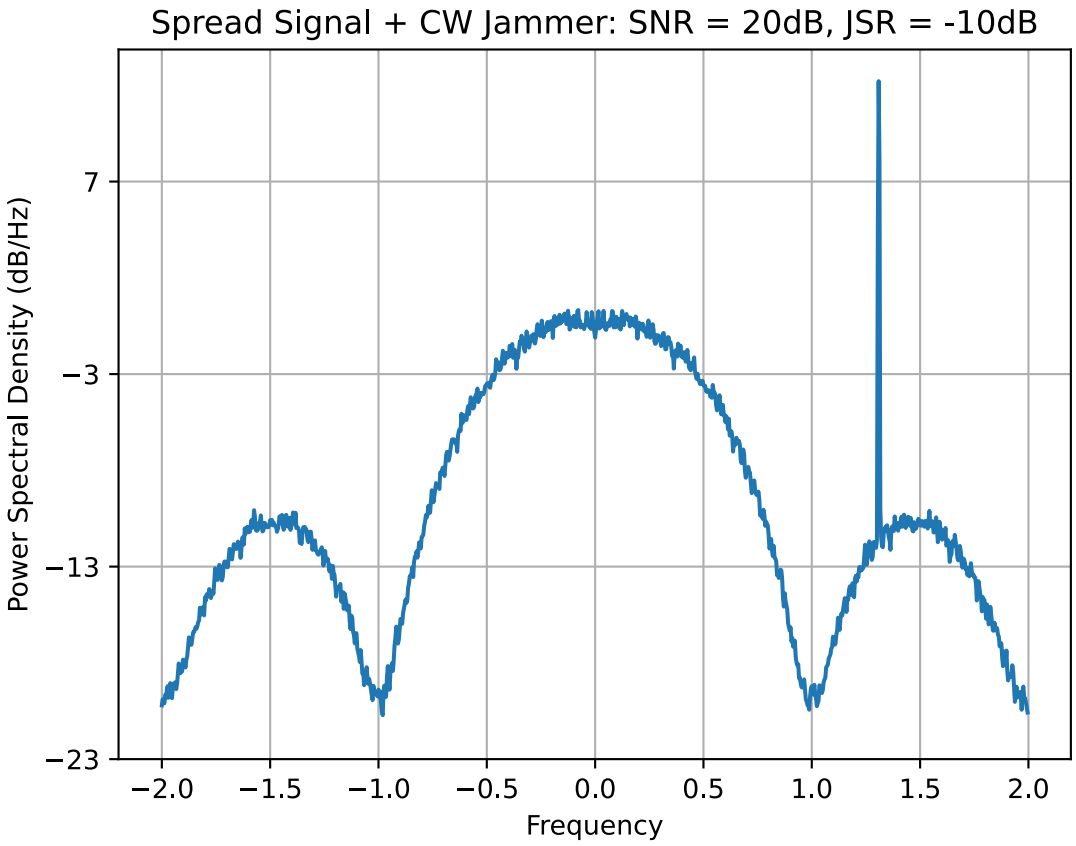


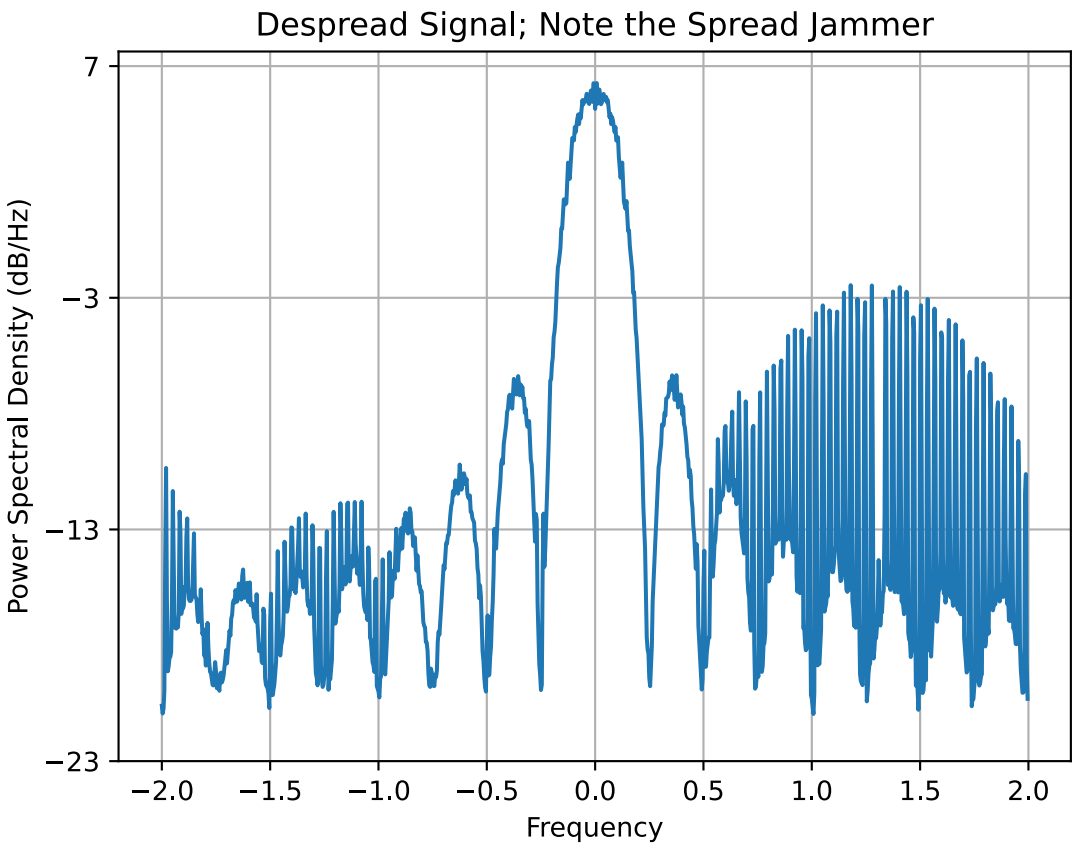

### 7.1 DSSS with Jamming

Continue the above example by adding a CW jammer and noise.

[65]:
```
J = exp(1j*2*pi*2.3273*n)
cd_n = dc.cpx_awgn(cd,20,4)
psd(cd_n + J[:len(cd_n)]*10**(-10/20),NFFT=2**10,Fs=4);
title(r'Spread Signal + CW Jammer: SNR = 20dB, JSR = -10dB');
figure()
psd((cd_n + J[:len(cd_n)]*10**(-10/20))*c,NFFT=2**10,Fs=4);
title(r'Despread Signal; Note the Spread Jammer')
```

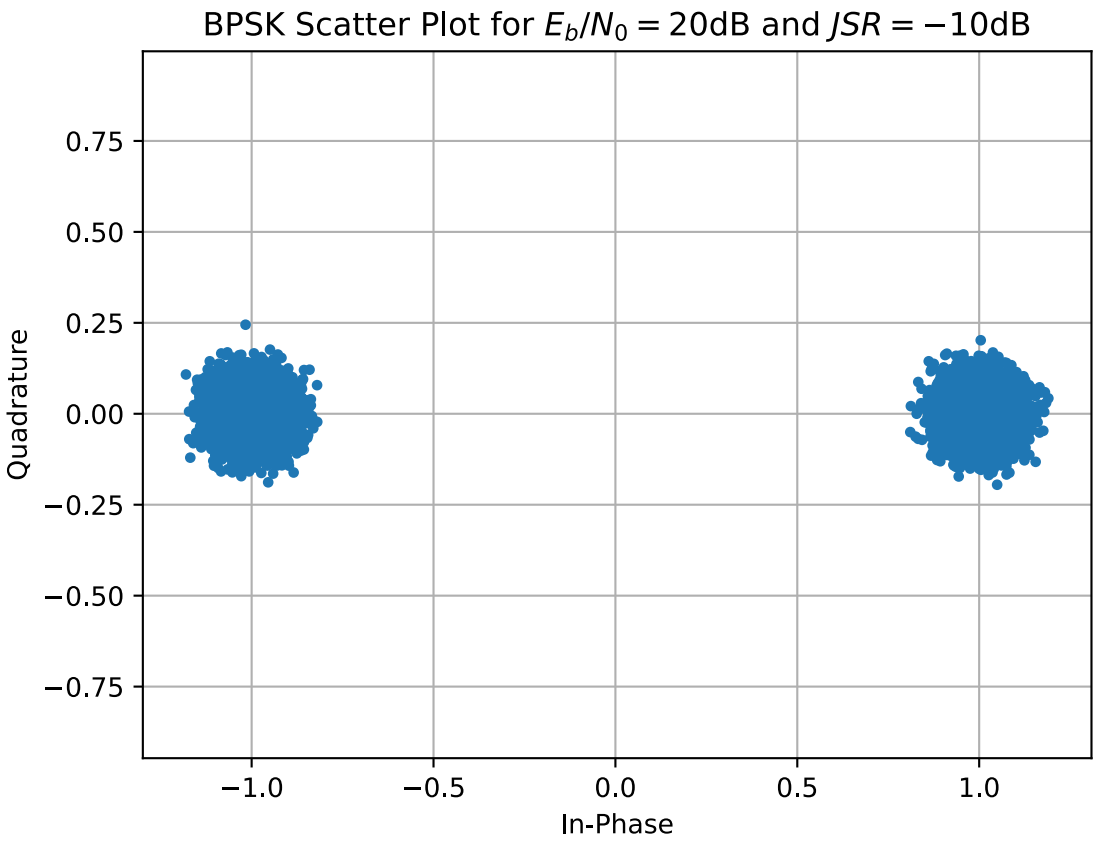
[65]:
```
Text(0.5, 1.0, 'Despread Signal; Note the Spread Jammer')
```


Spread Signal + CW Jammer: SNR = 20dB, JSR = -10dB
Power Spectral Density (dB/Hz)
7
−3
−13
−23
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
Frequency




### 7.2 Apply the Matched Filter and Look at the Scatter Plot

[50]:
```
z = signal.lfilter(b/sum(b),1,(cd_n + J*20**(-10/20))*c)
# psd(z,NFFT=2**10,Fs=4);
plot(z[10*16-1::16].real,z[10*16-1::16].imag,'.')
#plot(z[:200].real)
grid()
xlabel('In-Phase')
ylabel('Quadrature')
title(r'BPSK Scatter Plot for $E_b/N_0 = 20$dB and $JSR = -10$dB')
axis('equal');
```

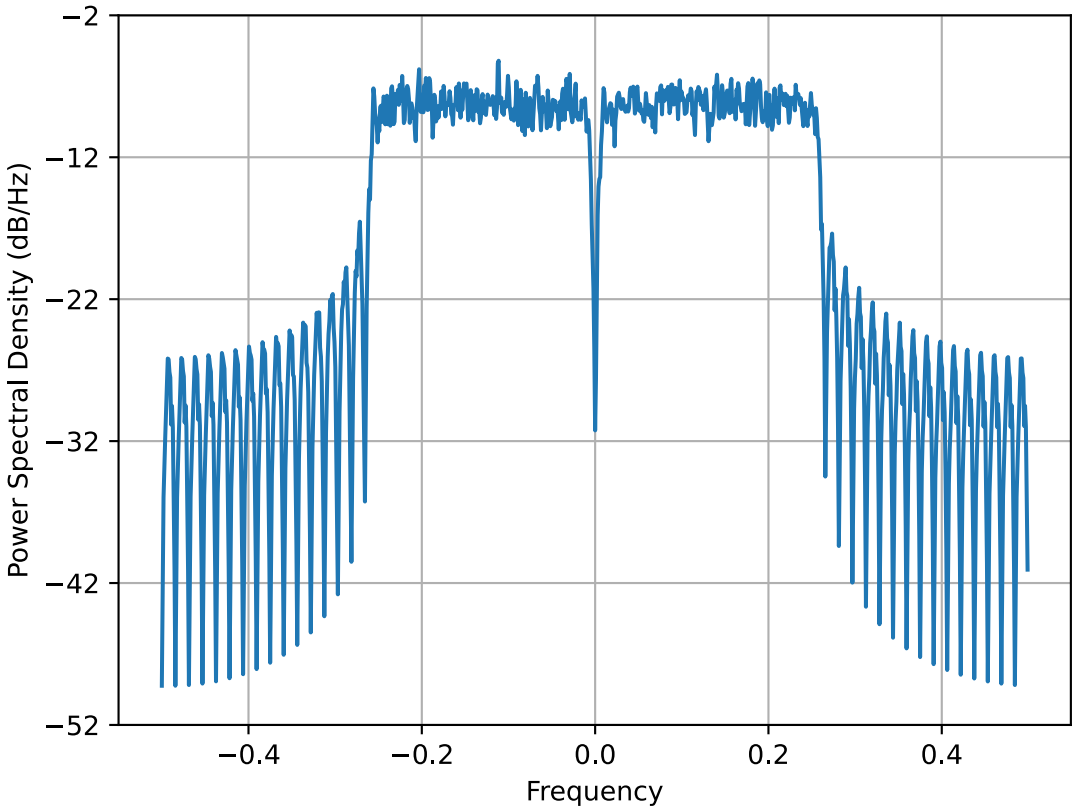



# 8 OFDM

### 8.0.1 Transmit with One Sample per Symbol

[58]:
```
x1,b1,IQ_data1 = dc.qam_bb(10000,1,'16qam')
x_out = dc.ofdm_tx(IQ_data1,32,64)
psd(x_out,NFFT=2**10,Fs=1);
```

```
(312, 32)
```

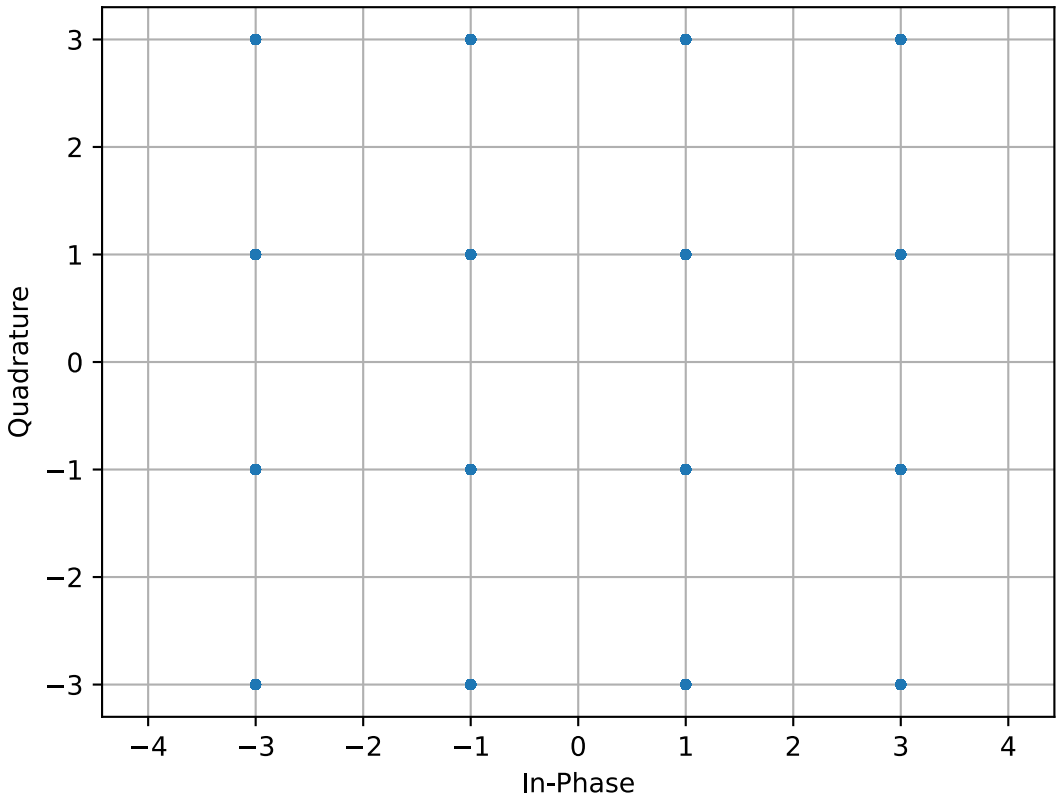



### 8.0.2 Receive with One Sample per Symbol

[7]:
```
len(z_out)
```

[7]:
```
9984
```

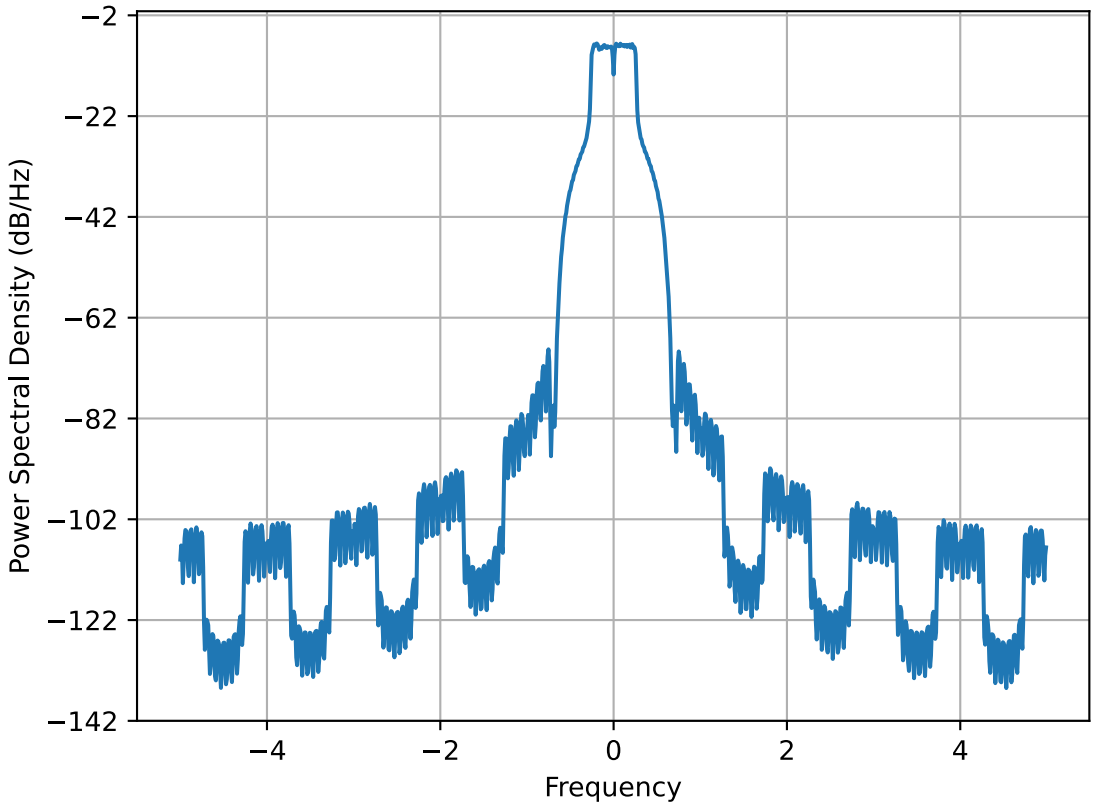
[52]:
```
r_out = dc.cpx_awgn(x_out,100,64/32) # Es/N0 = 100 dB
n = arange(len(r_out))
z_out,H = dc.ofdm_rx(r_out,32,64,-1,True,0,alpha=0.95,ht=[1])
plt.plot(z_out[10:].real,z_out[10:].imag,'.')
plt.xlabel('In-Phase')
plt.ylabel('Quadrature')
plt.axis('equal');
grid();
```




### 8.0.3 Create a Waveform Using Upsampling and Interpolation

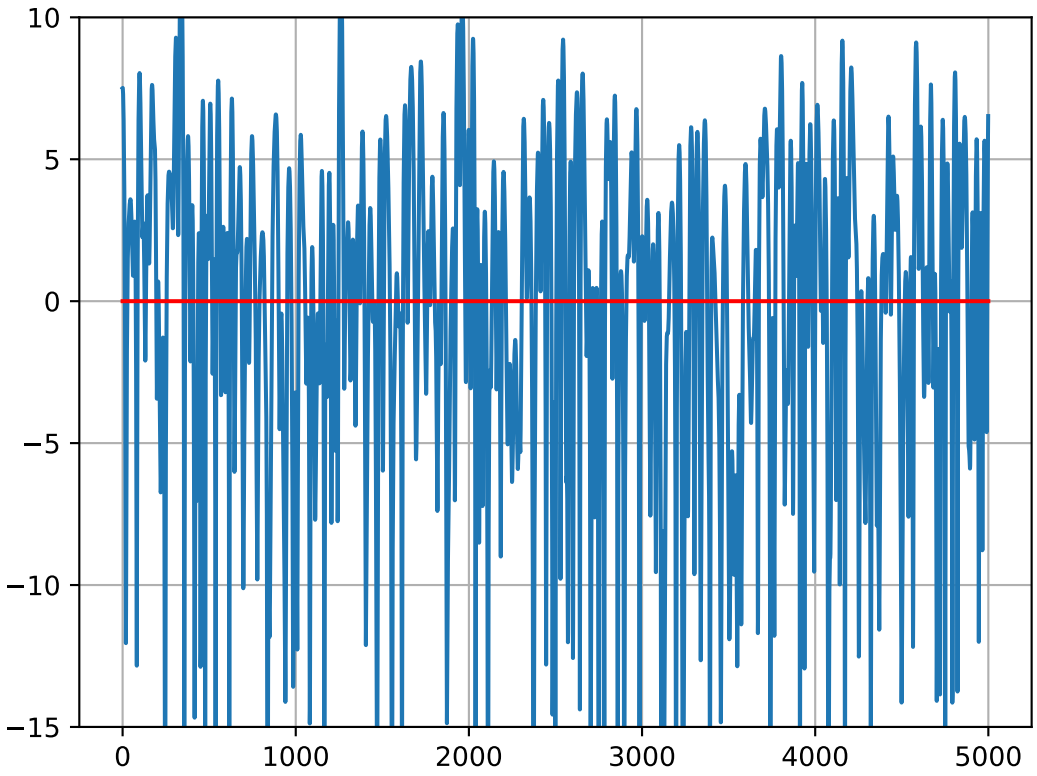
[59]:
```
x_out_up = ss.upsample(x_out,10)
# Use an RC filter for interpolation
x_out_upf = signal.lfilter(dc.rc_imp(10,.35,8),1,x_out_up)
psd(x_out_upf,NFFT=2**10,Fs=10);
len(dc.rc_imp(10,.35,8))
```

[59]: 161



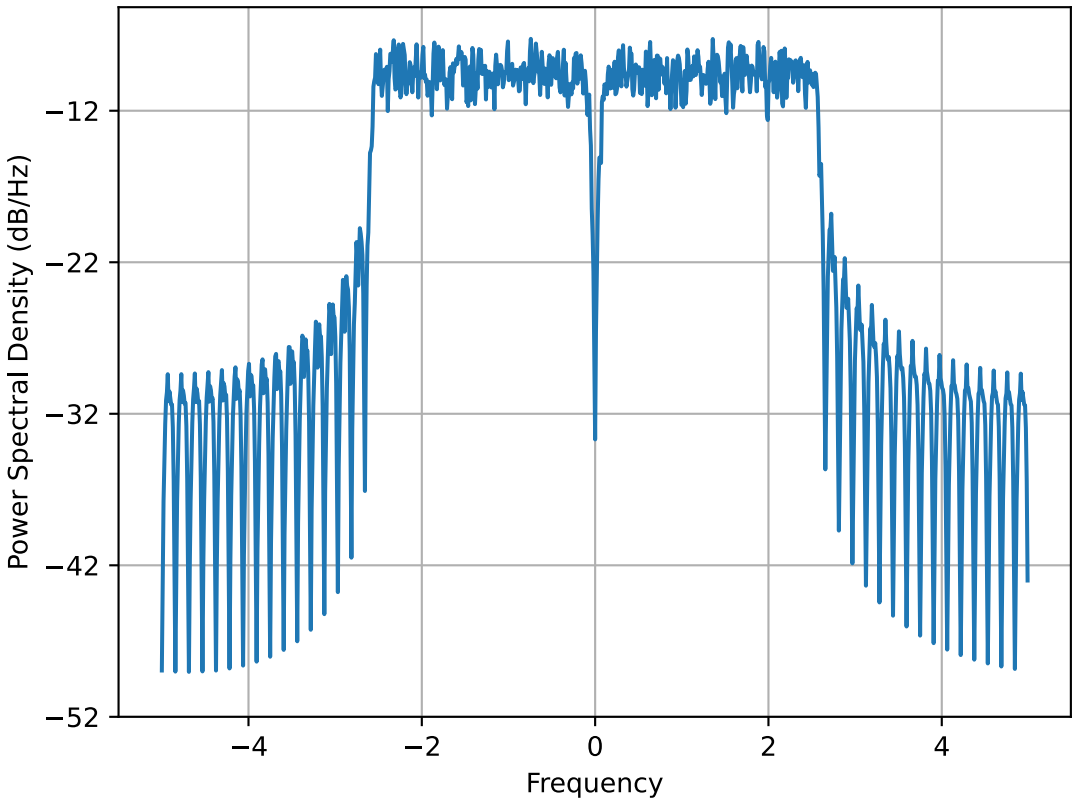

### 8.0.4 Observe the Signal Envelope

[54]:
```
# Find the envelope mean in dB
dB_mean = mean(20*log10(abs(x_out_upf[5000:10000])))
plot(20*log10(abs(x_out_upf[5000:10000]))-dB_mean)
plot([0,5000],[0,0],'r')
ylim([-15,10])
grid();
```

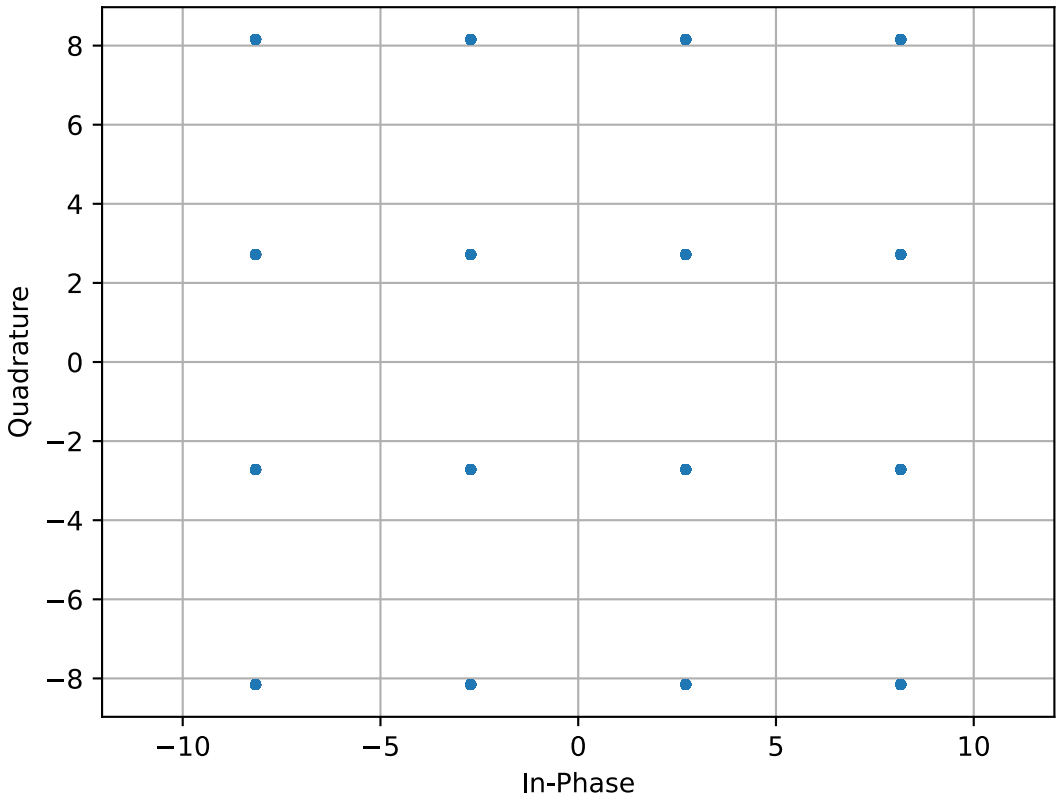



### 8.0.5 Downsample and View the PSD

[55]:
```
# Add Channel Noise
r_out = dc.cpx_awgn(x_out_upf,100,64/32*10) # Es/N0 = 100 dB
# Introduce Frequency Error with signal Df
n = arange(len(r_out))
Df = exp(1j*2*pi*0*n) # try 1e-5
r_out_Df = r_out*exp(Df)
psd(r_out_Df[0::10],NFFT=2**10,Fs=10);
```




[56]:
```
# Feed signal into OFDM receiver and consider frequency and timing errors
z_out,H = dc.ofdm_rx(r_out_Df[80+640::10],32,64,-1,True,0,alpha=0.95,ht=[1])
plt.plot(z_out[10:].real,z_out[10:].imag,'.')
plt.xlabel('In-Phase')
plt.ylabel('Quadrature')
plt.axis('equal');
grid();
```




### 8.1 LTE and OFDM

[ ]:

### 8.2 Mobile Radio Channel Simulation

A doubly-spread (delay and Doppler) channel simulator can be created rather easily. Such a simulator was developed in `m-code` and can be ported to Python or Julia.

more ...

[ ]: