***Take-Home* Exam Honor Code**

This being a take-home exam a strict honor code is assumed. Each person is to do his/her own work with no consultation with others regarding these problems. Bring any questions you have about the exam to me. Please be clear and concise in your answers. You may use Python, MATLAB, or Mathematica where appropriate. The simulation problems, 3 and 4, are best done in Python as full code library support and a sample Jupyter notebook are available. The exam is due at the end of the day Wednesday December 15, 2021. The exam has 100 points total. This is the complete final exam in one take-home package.

40 pts. 1.) **CROSS-8QAM**: A variation on 16QAM, known as *CROSS-8QAM*, can be obtained by retaining four inner signal points and four corner signal points, and then deleting the eight edge signal points. Figure 1 shows the signal constellation.

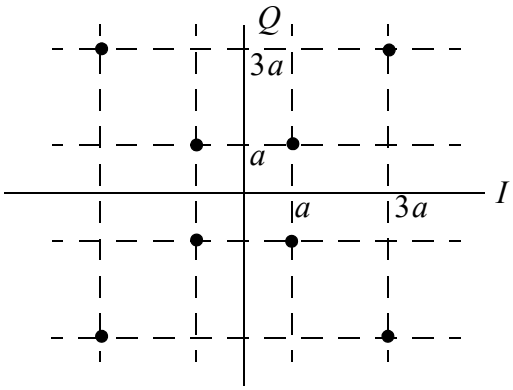


Figure 1: CROSS-8QAM signal constellation.

a.) Relate a to the average signal energy E_s. This should be similar to Z&T equation (10.58).

b.) Identify the nearest neighbors to the inner and outer signal points for proper Gray coding of the symbols.

c.) Assuming coherent demodulation, derive the probability of symbol error in terms of the average signal energy to noise power spectral density ratio, E_s/N_0.

d.) Find the approximate probability of bit error, assuming Gray coding of the signal points and rectangular decision regions similar 16QAM (suboptimal, see hints), in terms of the average energy per bit to noise power spectral density ratio, E_b/N_0.

e.) Implement a simulation of the *true optimum* receiver which uses the minimum Euclidean distance to make the symbol decisions. Represent the transmitted signal using just one sample per symbol as shown below (no waveforms). You can write the receiver code,

I-Q symbol amplitude look-up-table

```
# Cross-8QAM Modulator
Nsymb = 10000
a = 1
d_symb = array([a + 1j*a,3*a+1j*3*a,-a+1j*a,-3*a+1j*3*a,
                -a-1j*a,-3*a-1j*3*a,a-1j*a,3*a-1j*3*a])
data = randint(0,8,Nsymb)
x = d_symb[data]
y = dc.cpx_awgn(x,20,1)
```

including error detection, using one `for` loop and less than 10 lines of Python code.

f.) Plot the analytical bit error probability of coherent 8PSK and the CROSS-8QAM analysis of part (d), both versus E_b/N_0. Also overlay three or more simulation points from the optimum receiver of part (e). You should be able to get Pe values down to at least 10^{-5}.

20 pts. 2.) **IEEE802.11b**: In the wireless local area network protocol IEEE802.11b the lowest rate modulation is a binary scheme employing the two waveforms shown below:

$\tau_1 = \frac{1}{11}\mu s$

$\tau_2 = \frac{2}{11}\mu s$

$\tau_3 = \frac{3}{11}\mu s$

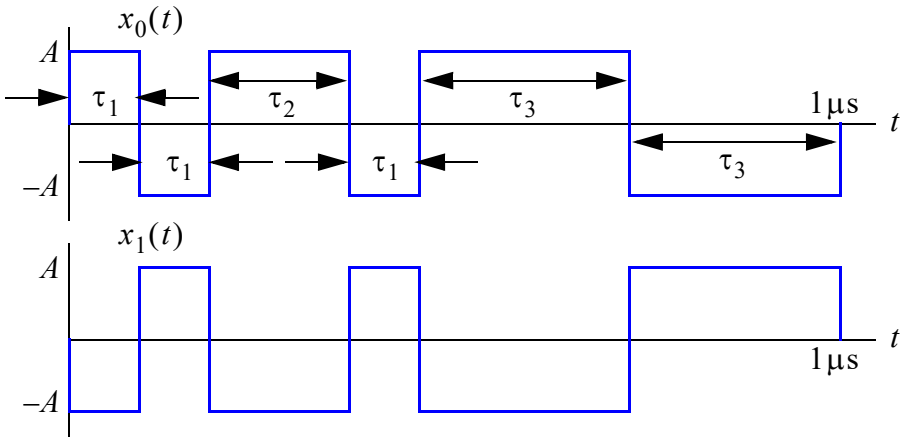


Figure 2: 802.11b low rate waveform.

a.) Choose A such that the energy per bit is E_b.

b.) What is the bit rate?

c.) What is the approximate null-to-null RF bandwidth occupied by this signal?

d.) Draw a block diagram of an optimum coherent demodulator for an AWGN channel. Assume that bits are equally likely.

20 pts. 3.) **QAM for Single Carrier and OFDM Systems**: In this problem you will perform simulation experiments with 64-QAM modulation in both a single carrier scenario and with OFDM. Of special interest is how single carrier QAM and OFDM with QAM compare over a delay spread channel.

a.) To get started you will get familiar with the QAM capability that has been added to the module `digitalcom.py`. Complex baseband 64-QAM ($M = 64$) can be generated using the function:

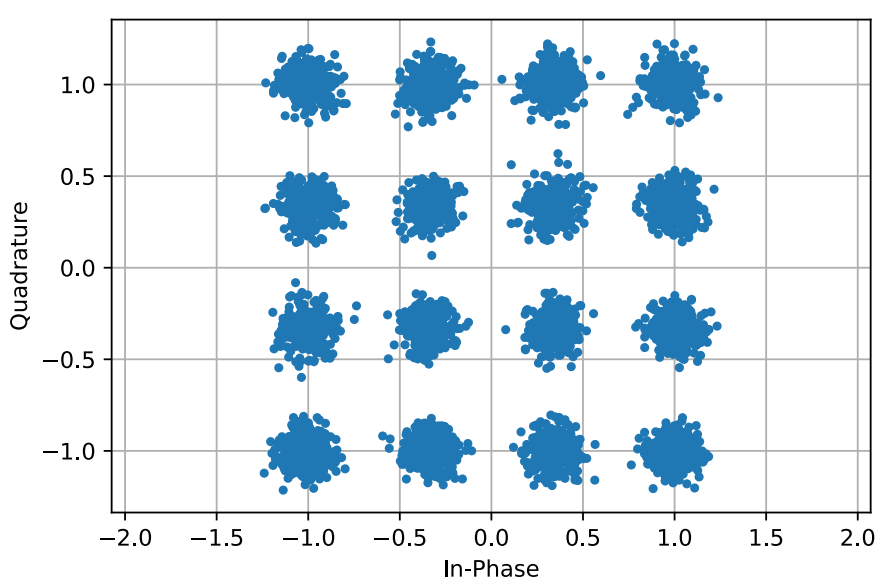
```
x,b,tx_data = dc.qam_bb(Nsymb,Ns,mod_type,pulse)
```

where `mod_type = qpsk, 16qam, 64qam,` or `256qam`, and pulse = '`rect`' or '`src`'. Symbol error detection, which allows you to estimate SEP, can be performed using the function:

```
Nsymb,Nerr,SEP = dc.qam_sep(tx_data,
                 rx_signal_samples,
                 mod_type,Ntransient=0)
```

The `tx_data` input assumes a one sample per symbol signal with real and imaginary parts taking on values of ± 1, ± 3, ± 5, etc. In contrast the `rx_signal_samples` input assumes one sample per symbol, but with the real and imaginary values scaled to fit on and inside the unit square. This interface definition works perfectly when testing a single carrier QAM modem as shown in the following example. When working with the OFDM modem you will need to change how you use this interface. As a simple example QAM with matched filter example, consider the following test that shows a scatter plot of the matched filter output:

```
x,b,tx_data = dc.qam_bb(5000,10,'16qam','src')
# Channel noise
x = dc.cpx_awgn(x,20,10)
# 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')
# Note below SEP_disp enables INFO logging through the logging module
Nsymb,Nerr,SEP = dc.qam_sep(tx_data,y[10+10*12::10],'16qam',n_transient=0)
```



- **Note**: The `x` output from `dc.QAM_BB()` is normalized so that after matched filtering the constellation points fit on and inside a unit square
- **Note**: The `tx_data` output only contains one sample per symbol and the real and imaginary values follow the modulation definition of ± 1 , ± 3 , etc. (given $a = 1$)

Figure 3: Matched filter output following the QAM modulator with $E_s/N_0 = 20$ dB.

Moving on to perform symbol error counting (here at a lower E_s/N_0), consider:

```
x,b,tx_data = dc.qam_bb(20000,1,'16qam','rect')
# Channel noise
x = dc.cpx_awgn(x,16,1) # Es/N0 = 16 dB
# Matched filter
y = signal.lfilter(b,1,x)
Nsymb,Nerr,SEP = dc.qam_sep(tx_data,y[1*12::1],'16qam',n_transient=0)
```

```
INFO:sk_dsp_comm.digitalcom:Phase ambiquity = (1j)**0, taumax = -12
INFO:sk_dsp_comm.digitalcom:Symbols = 19976, Errors 157, SEP = 7.86e-03
```

Note: The function QAM_SEP automatically adjusts the delay between `tx_data` with the sampled matched filter output. This is done by finding the location of the crosscorrelation peak between two complex baseband symbol sample sequences.

Your task in part (a) is to create a SEP plot similar to notes page 6-37 for $M = 64$. The simulation points of this plot should match the theoretical SEP given by

$$P_{E,s} \cong 4\left(1 - \frac{1}{\sqrt{M}}\right) Q\left(\sqrt{\frac{3E_s}{(M-1)N_0}}\right), \frac{E_s}{N_0} \gg 1$$

In summary, overlay you experimental SEP points, at minimum five, spread between P_E values ranging over 10^{-1} down to about 10^{-5} . The idea is to baseline the QAM simulation environment. The number of samples per symbol does not need to be 10 as shown here. You can reduce it down to `Ns=8` or even `Ns=4` at the lowest.Remember to get at least 100 error events per point. Depending upon your system RAM, you may it convenient to run smaller batches of symbols through and then keep a running count using the variables `Nsymb` and `Nerr`. You can actually use a `while` loop to break out of your simulation once 100 error events is reached. This approach is particularly useful for the smallest SEP values.

b.) In this part you will investigate the complex baseband OFDM transmitter and receiver functions found in the supplied Jupyter notebook `5630_FinalProject_Sample.ipynb.zip` and the code module `ofdm.py`. The intent of the OFDM transmitter is to create a complex baseband signal of form similar to Sesia[1].

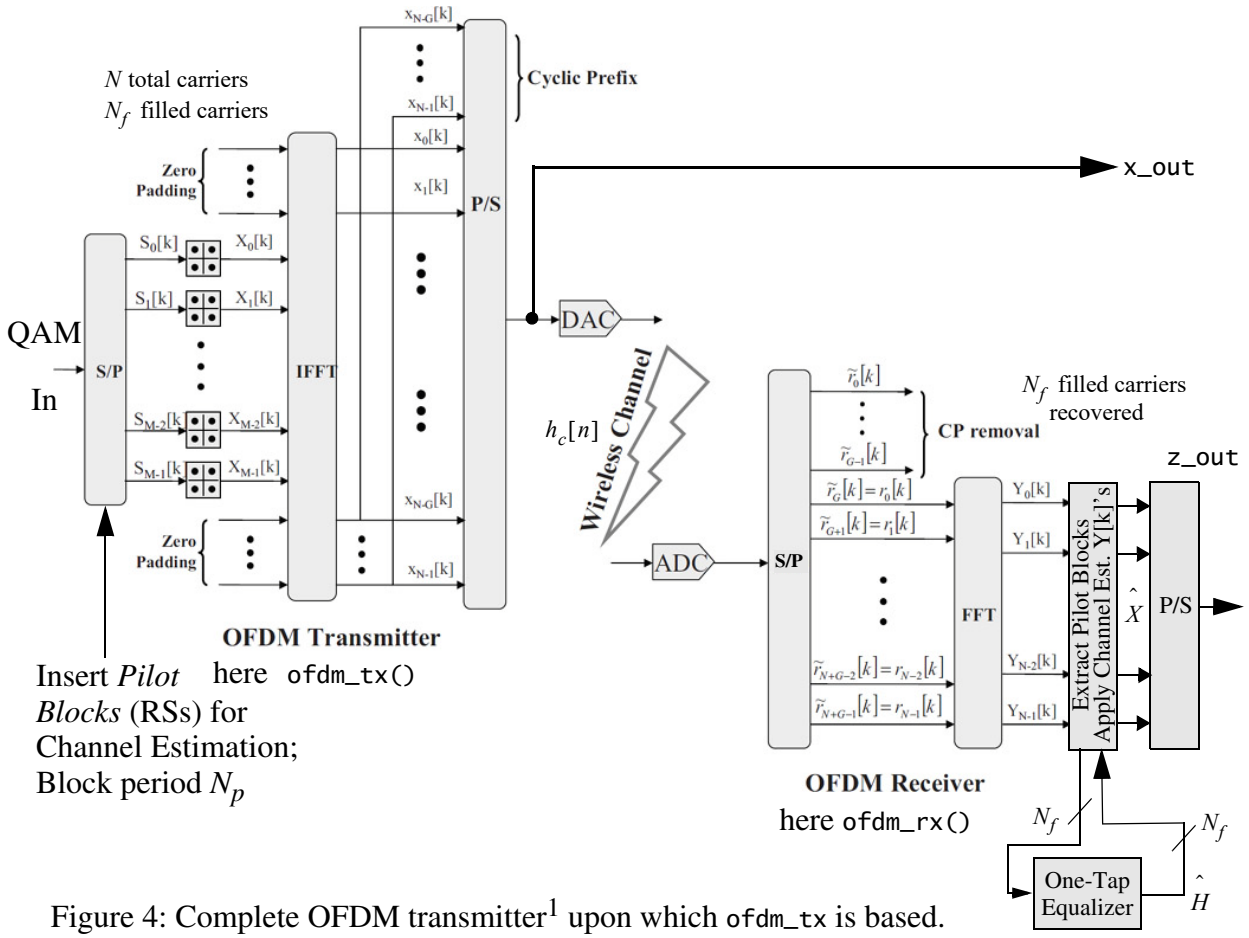


Figure 4: Complete OFDM transmitter[1] upon which `ofdm_tx` is based.

The original Sesia figure shown here has been enhanced to include the insertion of pilot carrier blocks at the transmitter and channel estimation at the receiver. Cyclic insertion and extraction is as shown, but here is software selectable to length N_{cp}. More details on channel estimation using pilots and working with the cyclic prefix is included under part (c).

The code listing of the OFDM functions of interest `ofdm_tx` and `ofdm_rx`, along with two helper functions, `mux_pilot_blocks` and `chan_est_equalize`, are also shown below. The module `digitalcom.py` now contains these functions but they are also listed below.

```
def ofdm_tx(IQ_data,Nf,N,Np=0,cp=False,Ncp=0):
    """
    x_out = ofdm_tx(IQ_data,Nf,N,Np=0,cp=False,Ncp=0)
    ================================================================================
    IQ_data = +/-1, +/-3, etc complex QAM symbol sample inputs
         Nf = number of filled carriers, must be even and Nf < N
          N = total number of carriers; generally a power 2, e.g., 64, 1024, etc
         Np = Period of pilot code blocks; 0 <=> no pilots
         cp = False/True <=> bypass cp insertion entirely if False
        Ncp = the length of the cyclic prefix
    ================================================================================
```

1. Stefania Sesia (Author), Issam Toufik, LTE – The UMTS Long Term Evolution: From Theory to Practice, Wiley, August 29, 2011.

```
     x_out = complex baseband OFDM waveform output after P/S and CP insertion
    ========================================================================

    Mark Wickert November-December 2014, Updated December 2016
    """
    N_symb = len(IQ_data)
    N_OFDM = N_symb//Nf
    IQ_data = IQ_data[:N_OFDM*Nf]
    IQ_s2p = np.reshape(IQ_data,(N_OFDM,Nf)) #carrier symbols by column
    print(IQ_s2p.shape)
    if Np > 0:
        IQ_s2p = mux_pilot_blocks(IQ_s2p,Np)
        N_OFDM = IQ_s2p.shape[0]
        print(IQ_s2p.shape)
    if cp:
        x_out = np.zeros(N_OFDM*(N+Ncp),dtype=np.complex128)
    else:
        x_out = np.zeros(N_OFDM*N,dtype=np.complex128)
    for k in xrange(N_OFDM):
        buff = np.zeros(N,dtype=np.complex128)
        for n in range(-Nf//2,Nf//2+1):
            if n == 0:  # Modulate carrier f = 0
                buff[0] = 0 # This can be a pilot carrier
            elif n > 0: # Modulate carriers f = 1:Nf/2
                buff[n] = IQ_s2p[k,n-1]
            else:       # Modulate carriers f = -Nf/2:-1
                buff[N+n] = IQ_s2p[k,Nf+n]
        if cp:
            #With cyclic prefix
            x_out_buff = fft.ifft(buff)
            x_out[k*(N+Ncp):(k+1)*(N+Ncp)] = np.concatenate((x_out_buff[N-Ncp:],\
                                                             x_out_buff))
        else:
            # No cyclic prefix included
            x_out[k*N:(k+1)*N] = fft.ifft(buff)
    return x_out

def mux_pilot_blocks(IQ_data,Np):
    """
    IQ_datap = mux_pilot_blocks(IQ_data,Np)

    A helper function called by OFDM_tx that inserts pilot block for use
    in channel estimation when a delay spread channel is present.
    ========================================================================
     IQ_data = a 2D array of input QAM symbols with the columns
               representing the NF carrier frequencies and each
               row the QAM symbols used to form an OFDM symbol
          Np = the period of the pilot blocks; e.g., a pilot block is
               inserted every Np OFDM symbols (Np-1 OFDM data symbols
               of width Nf are inserted in between the pilot blocks.
    ========================================================================
    IQ_datap = IQ_data with pilot blocks inserted
    ========================================================================

    Mark Wickert December 2014, Updated December 2016
    """
    N_OFDM = IQ_data.shape[0]
    Npb = N_OFDM//(Np-1)
    N_OFDM_rem = N_OFDM - Npb*(Np-1)
    Nf = IQ_data.shape[1]
    IQ_datap = np.zeros((N_OFDM + Npb + 1,Nf),dtype=np.complex128)
    pilots = np.ones(Nf) # The pilot symbol is simply 1 + j0
    for k in xrange(Npb):
        IQ_datap[Np*k:Np*(k+1),:] = np.vstack((pilots,\
                                    IQ_data[(Np-1)*k:(Np-1)*(k+1),:]))
    IQ_datap[Np*Npb:Np*(Npb+N_OFDM_rem),:] = np.vstack((pilots,\
                                    IQ_data[(Np-1)*Npb:,:]))
    return IQ_datap
```

```
def ofdm_rx(x,Nf,N,Np=0,cp=False,Ncp=0,alpha = 0.95,ht=None):
    """
    z_out, H = OFDM_rx(x,Nf,N,Np=0,cp=False,Ncp=0,alpha = 0.95,ht=None)
    ================================================================================
          x = received complex baseband OFDM signal
         Nf = number of filled carriers, must be even and Nf < N
          N = total number of carriers; generally a power 2, e.g., 64, 1024, etc
         Np = Period of pilot code blocks; 0 <=> no pilots; -1 <=> use the ht
              impulse response input to equalize the OFDM symbols; note
              equalization still requires Ncp > 0 to work on a delay spread
              channel.
         cp = False/True <=> if False assume no CP is present
        Ncp = the length of the cyclic prefix
      alpha = the filter forgetting factor in the channel estimator
              Typically alpha is 0.9 to 0.99.
         nt = input the known theortical channel impulse response
    ================================================================================
     z_out = recovered complex baseband QAM symbols as a serial stream;
             as appropriate channel estimation has been applied.
         H = channel estimate (in the frequency domain at each subcarrier)
    ================================================================================

    Mark Wickert November 2014, Updated December 2016
    """
    N_symb = len(x)//(N+Ncp)
    y_out = np.zeros(N_symb*N,dtype=np.complex128)
    for k in xrange(N_symb):
        if cp:
            # Remove the cyclic prefix
            buff = x[k*(N+Ncp)+Ncp:(k+1)*(N+Ncp)]
        else:
            buff = x[k*N:(k+1)*N]
        y_out[k*N:(k+1)*N] = fft.fft(buff)
    # Demultiplex into Nf parallel streams from N total, including
    # the pilot blocks which contain channel information
    z_out = np.reshape(y_out,(N_symb,N))
    z_out = np.hstack((z_out[:,1:Nf//2+1],z_out[:,N-Nf//2:N]))
    if Np > 0:
        if ht == None:
            z_out,H = chan_est_equalize(z_out,Np,alpha)
        else:
            Ht = fft.fft(ht,N)
            Hht = np.hstack((Ht[1:Nf//2+1],Ht[N-Nf//2:]))
            z_out,H = chan_est_equalize(z_out,Np,alpha,Hht)
    elif Np == -1: # Ideal equalization using hc
        Ht = fft.fft(ht,N)
        H = np.hstack((Ht[1:Nf//2+1],Ht[N-Nf//2:]))
        for k in xrange(N_symb):
            z_out[k,:] /= H
    else:
        H = np.ones(Nf)
    # Multiplex into original serial symbol stream
    return z_out.flatten(),H

def chan_est_equalize(z,Np,alpha,Ht=None):
    """
    zz_out,H = chan_est_eq(z,Nf,Np,alpha,Ht=None)

    This is a helper function for OFDM_rx to unpack pilot blocks from
    from the entire set of received OFDM symbols (the Nf of N filled
    carriers only); then estimate the channel array H recursively,
    and finally apply H_hat to Y, i.e., X_hat = Y/H_hat
    carrier-by-carrier. Note if Np = -1, then H_hat = H, the true
    channel.
    ================================================================
        z = Input N_OFDM x Nf 2D array containing pilot blocks and
            OFDM data symbols.
```

```
   Np = the pilot block period; if -1 use the known channel
        impulse response input to ht.
alpha = The forgetting factor used to recursively estimate H_hat
   Ht = the theoretical channel frequency response to allow ideal
        equalization provided Ncp is adequate.
=================================================================
zz_out= The input z with the pilot blocks removed and one-tap
        equalization applied to each of the Nf carriers.
    H = The channel estimate in the frequency domain; an array
        of length Nf; will return Ht if provided as an input.
=================================================================

Mark Wickert December 2014, Updated December 2016
"""
N_OFDM = z.shape[0]
Nf = z.shape[1]
Npb = N_OFDM//Np
N_part = N_OFDM - Npb*Np - 1
zz_out = np.zeros_like(z)
Hmatrix = np.zeros((N_OFDM,Nf),dtype=np.complex128)
k_fill = 0
k_pilot = 0
for k in xrange(N_OFDM):
    if np.mod(k,Np) == 0: # Process pilot blocks
        if k == 0:
            H = z[k,:]
        else:
            H = alpha*H + (1-alpha)*z[k,:]
        Hmatrix[k_pilot,:] = H
        k_pilot += 1
    else:               # process data blocks
        if Ht == None:
            zz_out[k_fill,:] = z[k,:]/H # apply equalizer
        else:
            zz_out[k_fill,:] = z[k,:]/Ht # apply ideal equalizer
        k_fill += 1
zz_out = zz_out[:k_fill,:] # Trim to # of OFDM data symbols
Hmatrix = Hmatrix[:k_pilot,:] # Trim to # of OFDM pilot symbols
if k_pilot > 0: #Plot a few magnitude and phase channel estimates
    chan_idx = np.arange(0,Nf//2,4)
    plt.subplot(211)
    for i in chan_idx:
        plt.plot(np.abs(Hmatrix[:,i]))
    plt.title('Channel Estimates H[k] Over Selected Carrier Indices')
    plt.ylabel('|H[k]|')
    plt.grid();
    plt.subplot(212)
    for i in chan_idx:
        plt.plot(np.angle(Hmatrix[:,i]))
    plt.xlabel('Channel Estimate Update Index')
    plt.ylabel('angle[H[k] (rad)')
    plt.grid();
return zz_out,H
```

The `ofdm_tx()` function takes a serial input stream of data symbols, real or complex. In this case we are interested in QAM, so the symbols are complex valued. Note no oversampling is employed, so the input symbols are represented at one sample per symbol (effectively `Ns=1`). As a simple example consider:

```
x1,b1,IQ_data1 = dc.qam_bb(50000,1,'16qam')
# dc.OFDM_tx(IQ_data, Nf, N, Np=0, cp=False, Ncp=0)
x_out = dc.ofdm_tx(IQ_data1,32,64)
psd(x_out,2**10,1);
xlabel(r'Normalized Frequency ($\omega/(2\pi)=f/f_s$)')
ylim([-40,0]);
xlim([-.5,.5]);
```

```
(1562, 32)
```

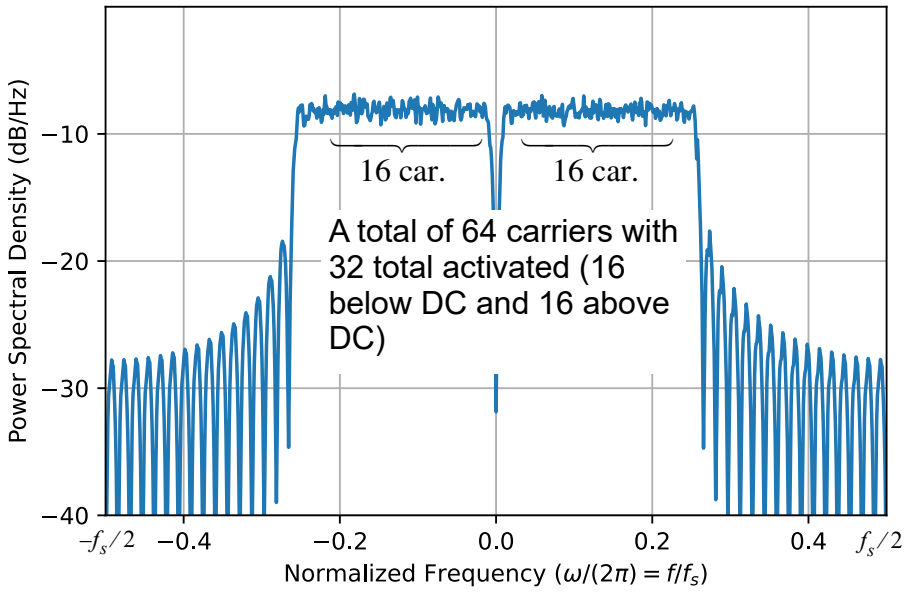
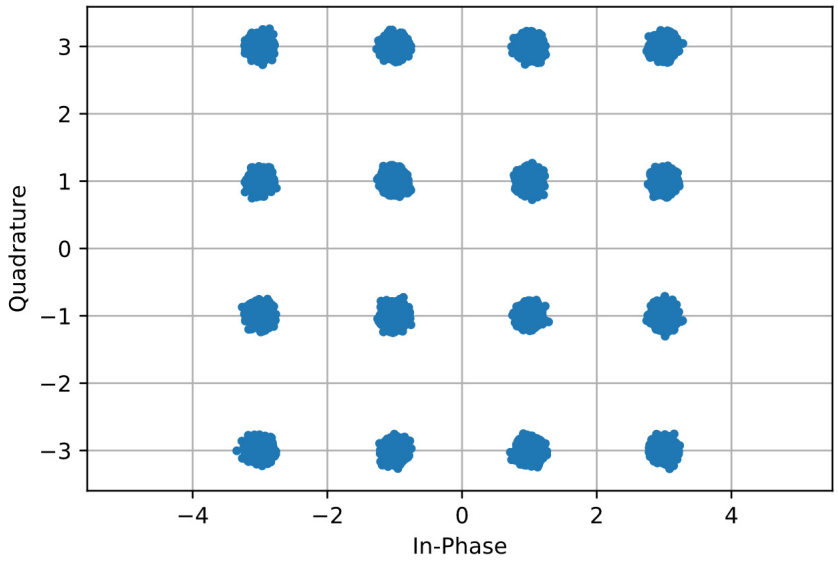


Figure 5: `OFDM_tx()` output in terms of the normalized frequency f/f_s.

Exercising the receiver, and adding noise scaled to calibrate the noise level to account for the ratio of total carrier count to active carriers, we have:

```
r_out = dc.cpx_awgn(x_out,30,64/32) # Es/N0 = 20 dB
#z_out,H = OFDM_rx(x,Nf,N,Np=0,cp=False,Ncp=0,alpha = 0.95,ht=None)
z_out,H = dc.ofdm_rx(r_out,32,64)
plot(z_out[200:].real,z_out[200:].imag,'.')
xlabel('In-Phase')
ylabel('Quadrature')
axis('equal')
grid();
results = dc.qam_sep(z_out,x1,'16qam') # note results = (Nsymb,Nerr,SEP)
```

```
INFO:sk_dsp_comm.digitalcom:Phase ambiquity = (1j)**0, taumax = 0
INFO:sk_dsp_comm.digitalcom:Symbols = 49984, Errors 0, SEP = 0.00e+00
```



- **Note**: The OFDM receiver output is not normalized to fit inside a unit square; it actually follows the amplitude swings of the input at the transmitter
- To use `QAM_SEP` for error detection reverse the roles of the inputs: signal `z_out` is input as the `tx` reference signal and the normalized `OFDM_tx` output, with `Ns=1` is input as the `rx` signal.

Figure 5: `OFDM_rx()` `z_out` with $E_s/N_0 = 20$ dB.

The task here is to baseline the SEP for the OFDM system, similar to part (a), except here you cannot say for sure that the theoretical QAM SEP should match that of the experiment. Following the instructions of part (a), overlay the anticipated experimental SEP points against the known QAM theory for $M = 64$. You are mostly running the code I have provided.

c.) In this final part you will experiment with `OFDM_tx()` and `OFDM_rx()` to study the significance of the cyclic prefix (CP) as shown in Figure 3 and the need for one-tap equalization, including obtaining channel state information from known *reference signals* (RSs). The length of the prefix needs to be determined based on expected channel delay spread conditions. A simulated delay spread channel is implemented using a simple FIR filter. In general the coefficients are complex. Here we will assume real coefficients. Since no up sampling has been employed in the transmitter, the samples are spaced at the symbol rate. Consider a channel impulse response of the form:

$$h_c[n] = \{1.0, 0.1, -0.05, 0.15, 0.2, 0.05\}$$

Cyclic prefix insertion/extraction, channel estimation, and one-tap frequency domain equalizer capability are written into the `OFDM_tx/rx` function pair.

The CP and Guard Time: With delay spread present intercarrier interference (ICI), i.e., crosstalk between the carriers arises, as they are no longer orthogonal. By including a *guard time* or CP you can insure that there is effectively an integer number of carrier cycles on the FFT interval[1]. With the proper guard time and delay spread present (channel $h_c[n]$ or $H(e^{j\omega})$ or $H[k]$ at the FFT points) a sum of sinusoids is now present resulting in an amplitude and phase shift at each of the carrier frequencies, i.e., $H[k]$. The receiver recovers $Y[k] = X[k]H[k]$ in the FFT bins. Knowing the channel, or having a good estimate, means that to remove the amplitude and phase shift you just apply a *one-tap equalizer* of the form

1. R. Van Nee and R. Prasad, *OFDM for Wireless Multimedia Communications*, Artech House, Boston, 2000.

$$\hat{X}[k] = \frac{Y[k]}{H[k]}$$

RSs, Channel Estimation, and One Tap Equalization: In LTE the RSs are spread in both time and frequency as shown below:

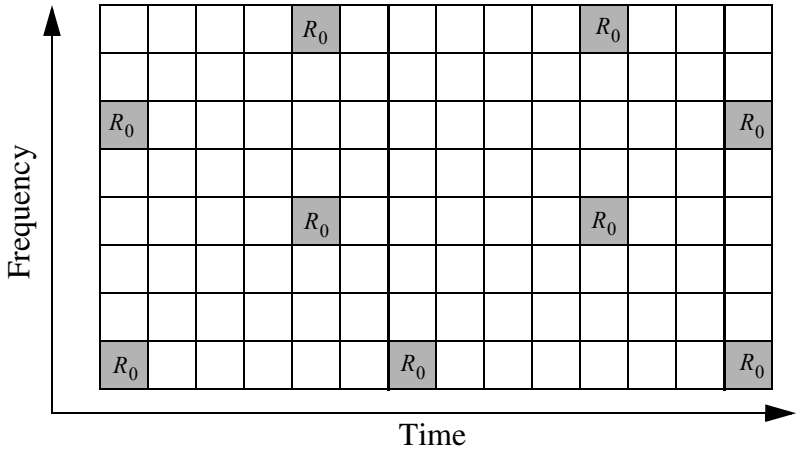


Figure 6: Reference signal distribution in LTE across both time and frequency.

It is also possible to just place RSs (pilot symbols) in a *comb* arrangement at specific frequencies over all time. The classical approach is to place the pilots as *blocks* over all the carrier frequencies but spaced in time. In `OFDM_tx`/`OFDM_rx` I have chosen to use pilot blocks as shown in Figure 7.

With known RSs at the receiver it is now a matter of estimating the channel state across all time and frequency cells. Realize that with noise present the ability to estimate the channel is still imperfect. When the channel is time-varying the challenge is even greater. Since the simulation model assumes a fixed channel with pilot blocks, noise is the main consideration. Upon the reception of a pilot block the receiver has an instantaneous, but noisy, estimate of the channel $H_n[k]$, where n denotes the time index. A simple IIR recursive estimator that can operate across the array of channel values is

$$\hat{H}_n[k] = \alpha \hat{H}_{n-1}[k] + (1-\alpha) H_n^{\text{pilot } n}[k],\ k \in \{N_f \text{ filled carrier frequencies}\}$$

where $0 < \alpha < 1$ is the *forgetting factor*. Typically α is close to one to provide good averaging and hence variance reduction due to noise.

Here you will test the transceiver system with the channel in place between the transmitter and an additive noise injection point. You will ultimately obtain SEP points for 64QAM under various considerations of CP length and channel estimation. Note, the CP adds overhead to the system, so making `N_CP` too large reduces payload data throughput. Including RSs, specifically pilot blocks also adds overhead.

(i) Working from the parameter values used to generate Figure 5, set `Np=0` in `OFDM_tx` and set `Np=-1` in `OFDM_rx`. Also set the $E_s/N_0 = 100$ dB. Make sure you input `hc` as the last argument into `OFDM_rx`. Turn on the CP (set `cp=True`) and starting with Ncp = 0, increase the CP length until you get a tight/clear scatter plot. Comment on your results. To be clear the `ODFD_tx` and `OFDM_rx` function interfaces are given below:

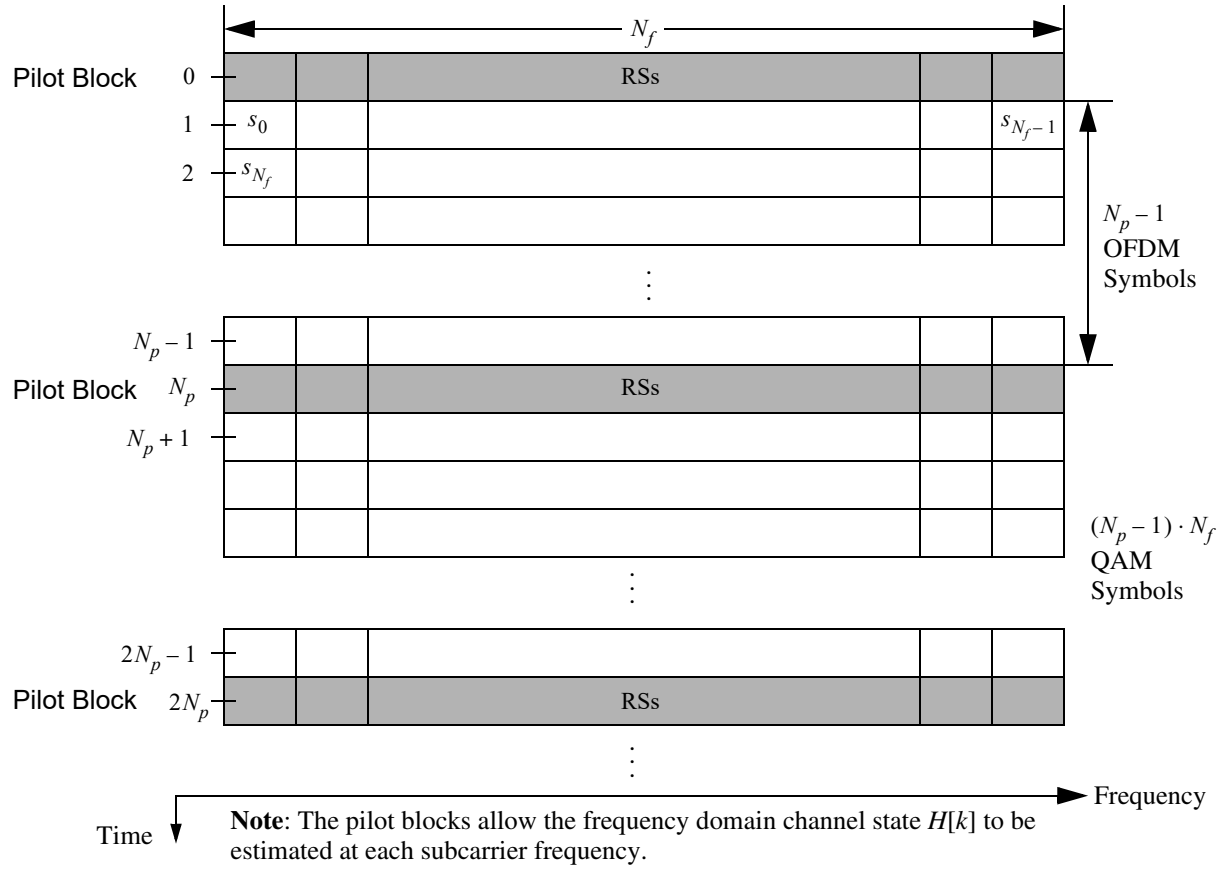


Figure 7: The layout of the pilot blocks as used in `OFDM_tx`/`OFDM_rx`.

```
x_out = ofdm_tx(IQ_data,Nf,N,Np=0,cp=False,Ncp=0)
    ================================================================================
    IQ_data = +/-1, +/-3, etc complex QAM symbol sample inputs
         Nf = number of filled carriers, must be even and Nf < N
          N = total number of carriers; generally a power 2, e.g., 64, 1024, etc
         Np = Period of pilot code blocks; 0 <=> no pilots
         cp = False/True <=> bypass cp insertion entirely if False
        Ncp = the length of the cyclic prefix
    ================================================================================
     x_out = complex baseband OFDM waveform output after P/S and CP insertion
    ================================================================================

z_out, H = ofdm_rx(x,Nf,N,Np=0,cp=False,Ncp=0,alpha = 0.95,ht=None)
    ================================================================================
          x = received complex baseband OFDM signal
         Nf = number of filled carriers, must be even and Nf < N
          N = total number of carriers; generally a power 2, e.g., 64, 1024, etc
         Np = Period of pilot code blocks; 0 <=> no pilots; -1 <=> use the ht
              impulse response input to equalize the OFDM symbols; note
              equalization still requires Ncp > 0 to work on a delay spread
              channel.
         cp = False/True <=> if False assume no CP is present
        Ncp = the length of the cyclic prefix
      alpha = the filter forgetting factor in the channel estimator
              Typically alpha is 0.9 to 0.99.
         nt = input the known theoretical channel impulse response
    ================================================================================
     z_out = recovered complex baseband QAM symbols as a serial stream;
             as appropriate channel estimation has been applied.
         H = channel estimate (in the frequency domain at each subcarrier)
           ================================================================================
```

(ii) With `Ncp=10` prove to yourself the need for channel estimation in combination with the CP. Turn of the equalizer by setting `Np=0`. Look at the scatter plot and comment.

(iii) Finally obtain a SEP plot containing 5 experimental SEP points. Your SEP plot should contain: (1) Ideal 64QAM theory, (2) experimental SEP points with ideal channel estimation/equalization and (3) experimental SEP points using the included channel estimation/equalization algorithm. Pick a reasonable value for N_p. Note: Without the CP and equalization the given channel is too difficult to send information through.

20 pts. 4.) **Convolutional Coding and Decoding Simulation**: In this problem you will experiment with rate 1/2 ($R = 1/2$) convolutional codes and use a soft decision Viterbi algorithm for decoding. The Python module `fec_conv.py` includes all of the necessary code for building simulations. You will also consider the use of *puncturing* to raise the code rate from 1/2 to 3/4.

The communications link model for both coder/decoder types is shown below in Figure 6. In

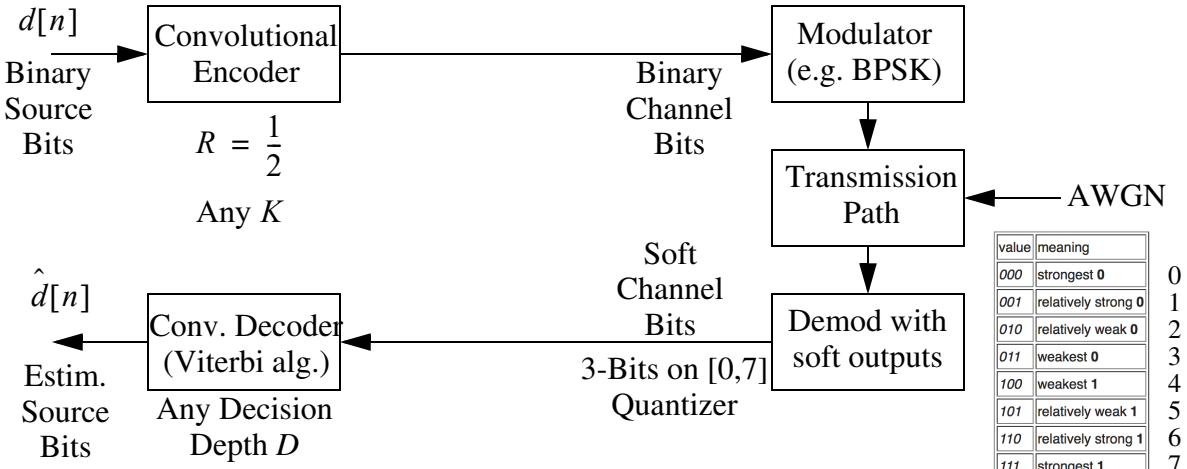


Figure 6a: Rate 1/2 forward error correction (FEC).

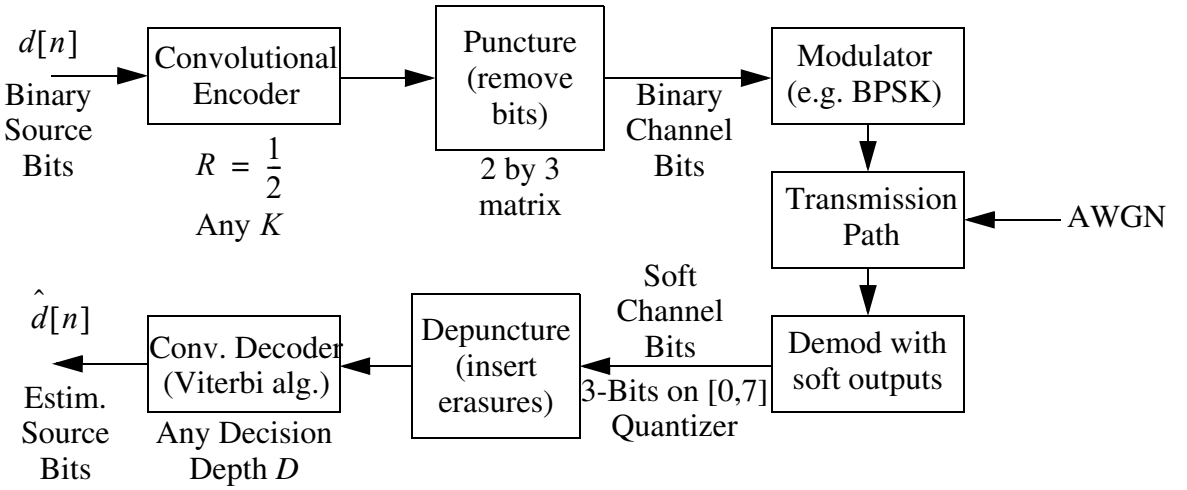


Figure 6b: Rate 3/4 forward error correction (FEC).

Figure 6a you see a $R = 1/2$, encodes each source bit into two channel bits. The convolutional encoder itself is described by two polynomials G_1 and G_2. The channel bits pass through the channel and become corrupted by AWGN. The demodulator provides soft decision values at its output on the interval [0, 7][1]. A value of 0 means the strongest '0' while 7 means the strongest '1'. Relatively strong, relatively weak, and weakest are terms used to describe the intervening values of '0's and '1's. In the current decoder implementation the precision on [0, 7] is actually doubles, but can be modified to be a true three bit value. The Viterbi algorithm (VA) is used for decoding and associated with it is an attribute know as the *decision depth*. The decision depth, D, refers to how long the VA waits before a bit decision is made regarding the actual source bit that was sent. The estimate of the binary source bits is thus

1. http://en.wikipedia.org/wiki/Viterbi_decoder

delayed by D bits.

Figure 6b shows the rate 1/2 FEC with the addition of puncturing/depuncturing to increase the rate to 3/4. The puncturing matrix is of the form

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

During puncturing row one of **A** is applied to the outputs from the G_1 polynomial and row two is applied to the outputs of the G_2 polynomial. To arrive at $R = 3/4$ note that one bit input to the encoder produces two code bits, but the puncturing matrix outputs four channel bits for every six code bits input. The ratio of channel bits to input bits is thus $2/1 \times 4/6 = 4/3$, so the effective rate, input bits to output bits is just the inverse, or 3/4. At the demodulator output the depuncturing operation inserts *erasure bits* at the location where code bits were originally located. An erasure bit relative to the 3-bit soft values on [0,7] is a value of 3.5 in real numbers.

The software allows any rate 1/2 code to be implemented. The attribute of interest here is the encoder constraint length, K. The number of memory states in the encoder is $K-1$, so the number of states needed to describe the operation of the VA is 2^{K-1}. A $K = 3$ code thus requires 4 states and $K = 7$ requires $2^6 = 64$ states. Popular $R = 1/2$ codes are shown in Table 1.

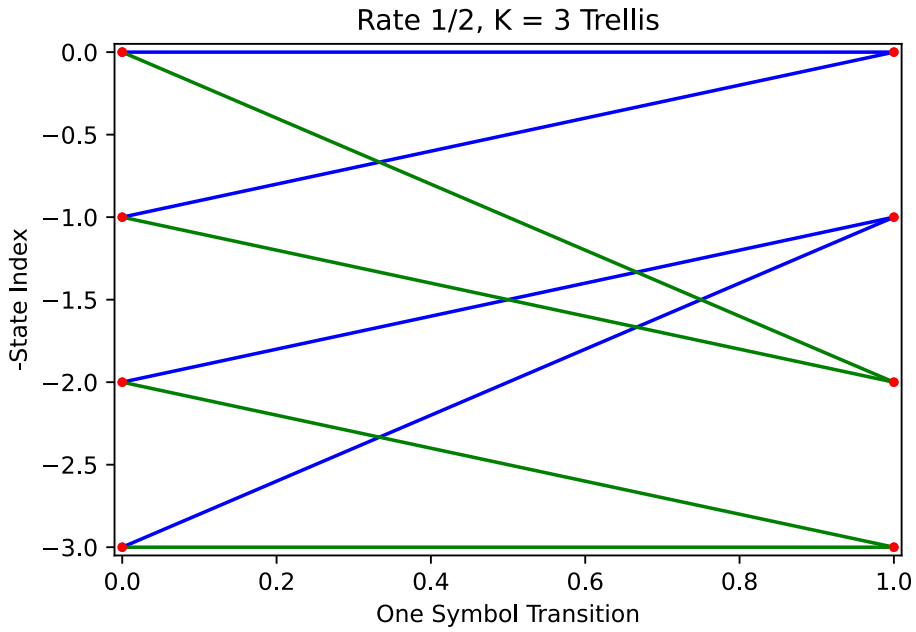
Table 1: Popular rate 1/2 convolutional codes[a]

K	G_1 and G_2 (octal and binary)	D_{free}	Weight spectra c_k for bounding the coded BEP							
			d_f	+1	+2	+3	+4	+5	+6	+7
3	(o5,o7) ('101','111')	5	1	4	12	32	80	192	448	1024
4	(o15,o17) ('1101', '1111')	6	2	7	18	49	130	333	836	2069
5	(o23,o35) ('10011', '11101')	7	4	12	20	72	225	500	1324	3680
6	(o53,o75) ('101011', '111101')	8	2	36	32	62	332	701	2342	5503
7	(o133,o171) ('1011011', '1111001')	10	36	0	211	0	1404	0	11633	0

a. Ziemer and Peterson, *Introduction to Digital Communications*, second edition, Prentice Hall, 2001.

Example: Consider the first code in Table 1 and use the Python class `fec_conv` defined inside the module `fec_conv.py`.

```
# Instantiate a fec_conv coder/decoder object
cc1 = fec.FECConv(('101','111'),Depth=10) # decision depth is 10
cc1.trellis_plot();
```




```
# Create 100 random 0/1 bits
x = randint(0,2,10000)
```

```
# Encode with shift register starting state of '00'
state = '00'
y,state = cc1.conv_encoder(x,state)
```

```
# Add channel noise to bits translated to +1/-1
yn = dc.cpx_awgn(2*y-1,2,1) # SNR = 2 dB
# Translate noisy +1/-1 bits to soft values on [0,7]
yn = (yn.real+1)/2*7
z = cc1.viterbi_decoder(yn)
```

```
print('Xor of input/output bits:')
np.int32(x[:75])^np.int32(z[:75])
```

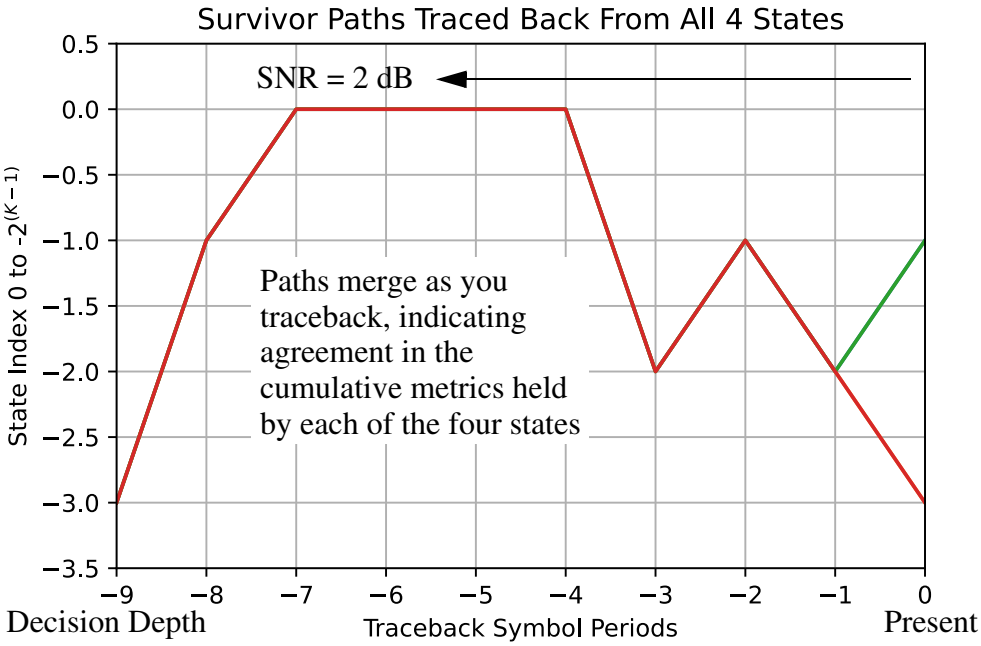
```
Xor of input/output bits:
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int32)
```

```
# Count bit errors
bit_count, bit_errors = dc.bit_errors(x,z)
print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %\
      (bit_count, bit_errors,bit_errors/bit_count))
```

```
INFO:sk_dsp_comm.digitalcom:kmax =  0, taumax = 0
Bits Received = 9991, Bit errors = 3, BEP = 3.00e-04
```

It is possible to see the traceback paths of the VA too:

```
# Look at the traceback in the VA trellis
cc1.traceback_plot()
```



a.) In this first part you will create a `fec_conv` object for the $K = 5$ code of Table 1. You will create a BEP plot similar to that found on page 7-41 of the Chapter 7 (text Chapter 12) notes. **Note**: You will need to increase D to about $5 \times 5 = 25$. Unlike the notes example, your results will be for soft-decision decoding. Functions for computing soft decision decoding upper bounds are contained in the module `fec_conv.py`. In addition to the BEP plot, also provide the trellis plot and a traceback plot under low and high SNR values.

The soft decision bounding formulas are similar to the hard decision formulas found in the Chapter 7 notes, i.e.,

$$P_E < \sum_{k=d_{\text{free}}}^{\infty} c_k P_k$$

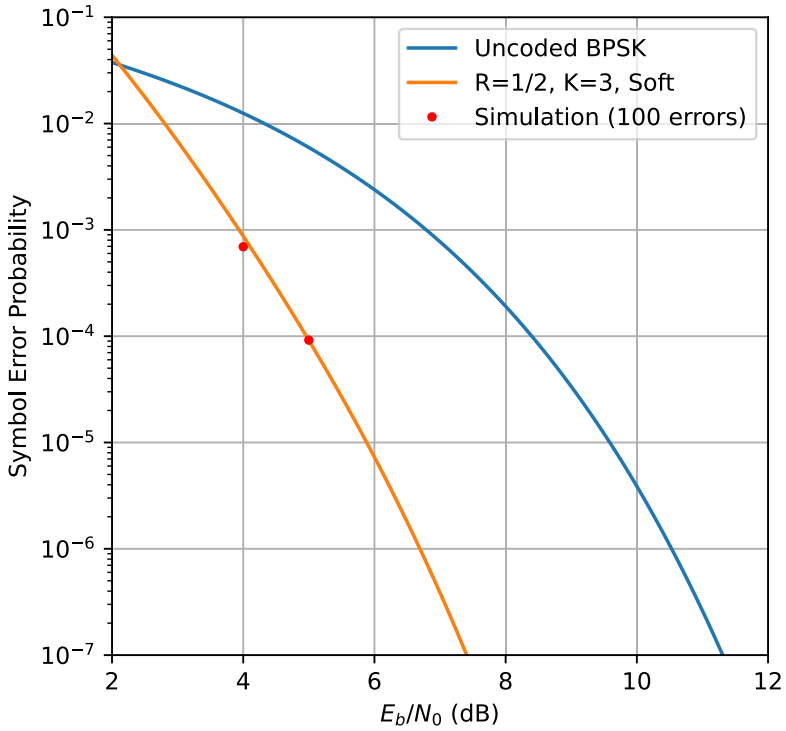
where

$$P_k = Q\left(\sqrt{\frac{2kRE_b}{N_0}}\right)$$

and $R = 1/2$, is the code rate. The Python function `conv_Pb_bound()` takes as inputs the code rate, the free distance, and the weight spectra as a list from Table 1. Continuing the $K = 5$ example from above

```
SNRdB = arange(2,12,.1)
Pb_uc = fec.conv_Pb_bound(1/2,5,[1,4,12,32,80,192,448,1024],SNRdB,2)
Pb_s = fec.conv_Pb_bound(1/2,5,[1,4,12,32,80,192,448,1024],SNRdB,1)
```

```
SNRdB = arange(2,12,.1)
# fec.conv_Pb_bound(R, dfree, Ck, SNRdB, hard_soft = 0, 1, or 2)
# 0 <==> Hard decoding bound, 1 <==> softdecoding bound, 2 <==> uncoded BPSK
Pb_uc = fec.conv_Pb_bound(1/2,5,[1,4,12,32,80,192,448,1024],SNRdB,2)
Pb_s = fec.conv_Pb_bound(1/2,5,[1,4,12,32,80,192,448,1024],SNRdB,1)
figure(figsize=(5,5))
semilogy(SNRdB,Pb_uc)
semilogy(SNRdB,Pb_s)
semilogy([4,5],[6.94e-4,9.18e-5],'r.') # expand as more sim points are collected
axis([2,12,1e-7,1e-1])
xlabel(r'$E_b/N_0$ (dB)')
ylabel(r'Symbol Error Probability')
legend((r'Uncoded BPSK',r'R=1/2, K=3, Soft',r'Simulation (100 errors)'),loc='bes
grid();
```



Simulation results are what is missing from the above plot. In your working with the $K = 5$ code you need produce the two curves shown above, plus at least three simulation overlay points for $P_E \in [10^{-2}, 10^{-5}]$.

b.) Revisit (a), but now reduce the decision depth, d, from the value you used in part (a). The objective is see how sensitive BEP is to d. The recommended approach is to fix SNR at some value where the experimental BEP is relatively easy to obtain, and then reduce d in steps of five or so to see the impact.

c.) Repeat part (a) using the 2×3 puncturing matrix defined earlier. Upper bounds on BEP can be produced again using the weight spectra for punctured codes[1]. Table 2 provides the needed values for bounding BEP with puncturing and $R = 3/4$.

1. L. Lee, "New Rate-Compatible Punctured Convolutional Codes for Viterbi Decoding," *IEEE Transactions on Communications*, Vol 42, N0. 12, pp. 3073–3079, December 1994.

Table 2: Weight spectra for selected rate 1/2 punctured convolutional codes [Lee].

K	G_1 and G_2	**A**	***R***	d_{free}	Weight spectra c_k for bounding the coded BEP						
					d_f	+1	+2	+3	+4	+5	+6
3	(o5,o7)	110, 101	3/4	3	15	104	540	2536	11302	48638	NA
5	(o23, o35)	101, 101	3/4	4	164	0	520 0	0	151211	0	3988108
7	(o133, o171	110, 101	3/4	5	42	201	149 2	10469	62935	37964 4	NA

To utilize the puncturing and depuncturing capability of the `fec_conv` class consider the following example worked for the $K = 3$ code:

```
# Create 99 random 0/1 bits (should be a multible of 3)
x = randint(0,2,99)
state = '00'
y,state = cc1.conv_encoder(x,state)
yp = cc1.puncture(y,('110','101'))
yp[:20]
```

Matrix **A** is defined by this tuple of strings

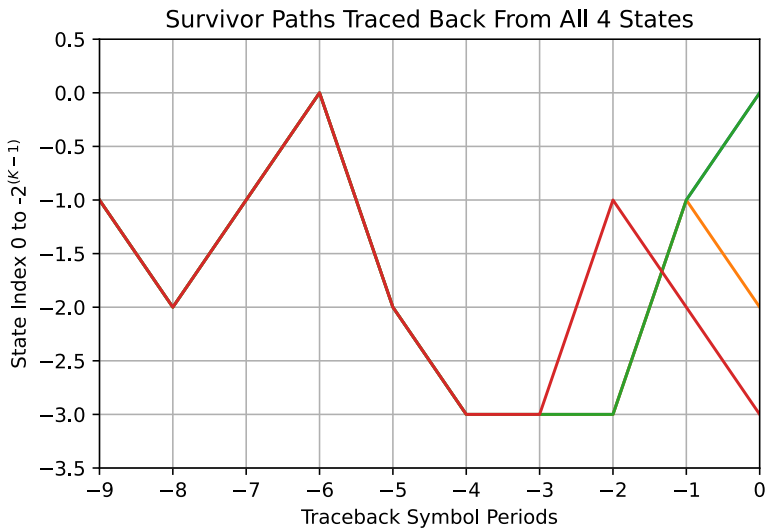
```
array([1., 1., 1., 0., 1., 1., 1., 0., 0., 1., 0., 0., 1., 1., 1., 1., 1.,
       1., 0., 1.])
```

```
ypn = dc.cpx_awgn(2*yp-1,8,1)

# Translate to soft values on [0,7]
ypn = (ypn.real+1)/2*7
# Signature: cc1.depuncture(soft_bits, puncture_pattern=('110', '101'), erase_va
zdpn = cc1.depuncture(ypn,('110','101'),3.5) # set erase threshold to 7/2
# Signature: cc1.viterbi_decoder(x, metric_type='soft', quant_level=3)
z = cc1.viterbi_decoder(zdpn)
z[:15]
```

Force the erasure value to the halfway point (since doubles are used at present

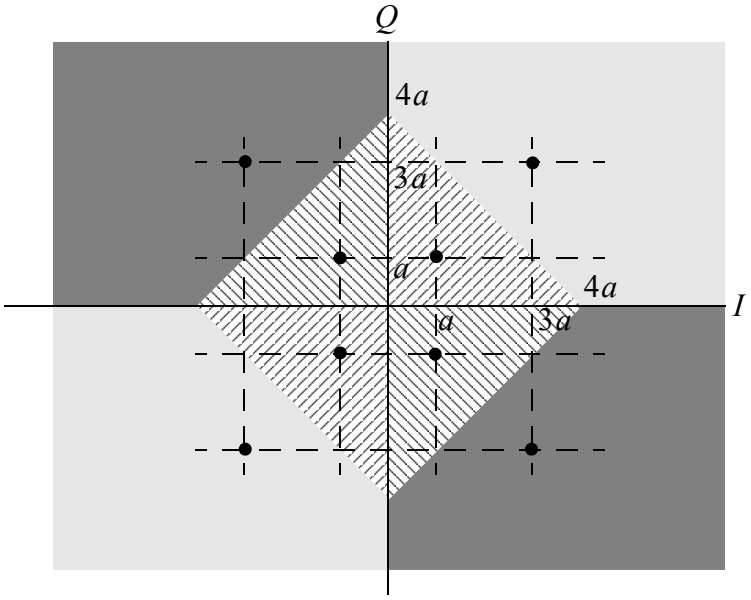
```
array([1., 1., 0., 0., 1., 1., 1., 1., 0., 0., 1., 0., 0., 0., 1.])
```



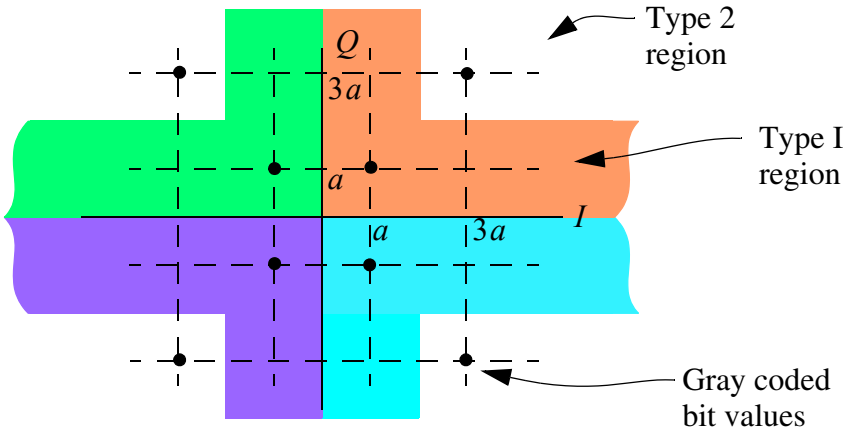
When you look at the traceback under noisy conditions you will see that more branches are required for paths to merge. This is expected as the code is weaker with the puncturing.

Hints

1.) For CROSS-8QAM the optimal decision boundaries between the 8 symbols are of the form:



This leads to a more complex P_e calculation and is more complex to detect. For this problem the intent is to keep it simple. Simplify the above with rectangular decision regions like those of 16-QAM, but *modified a bit* due to the absence of the other 8 points, e.g.,



Then, the P_e calculation is actually easier than the full 16-QAM. There are just two cases to consider. Your focus should just be on the two points in the first quadrant due to the symmetry of the constellation.

In general parts (c) and (d) are both based on the rectangular decision regions in the second figure above. In part (c) you just fund the symbol error probability (SEP) as a function of E_s/N_0. In part (d) you move on to finding the approximate bit error probability (BEP) using the Gray coding result as a function of E_b/N_0. In the simulation of part (e) you are effectively implementing the optimum decision regions of the first figure above.

Regarding part (a) note that the signal space has the nice property that the length of any signal point to the origin is the square root of the energy for that signal point/symbol. In MPSK for example the radius is just $\sqrt{E_s}$. In 16QAM the point in quadrant 1 closest to the origin has length squared of $(a^2 + a^2)^{1/2} = a\sqrt{2}$, so the energy of this signal point is $E_s = 2a^2$. The average of all the signal points is what part (a) is looking for.

2.) In part (a) you are to find A as a function of E_b and bit period T. When viewed as direct sequence spread spectrum we have an 11 bit spreading code.

3.) In part (a) expect very good agreement between theory and experiment. In part (b) expect the SEP with the added OFDM layer to be slightly degraded relative to ideal 64QAM. In part (c) the SEP may actually be better than part (b), depending on how closely you space the pilots. Note that `Np` of 10 means 10% overhead for sending pilots. Trying `Np` of 100 is more efficient as only 1% of the total throughput is allocated to pilots. When `Np` is small it appears that the SEP can actually be better than with the ideal channel estimates, apparently there is some correlation between the channel estimates and the channel noise that makes things better.

Part (c) with its three parts (i), (ii), and (iii) is the most challenging. In (i) you configure ideal channel estimation by sending no pilot. The CP is enabled and you increase the CP length starting from `Ncp = 0` in steps of one. At some point the scatter plot should be free of ISI. In (ii) you set `Ncp = 10` (higher than really needed) and turn the equalizer on, but by setting `Np = 0` you do not insert and pilots, so channel estimation is effectively turned off. Expect artistic looking 16QAM and 64QAM scatter plots. Comment on the engineering significance, however. In (iii) you obtain SEP results and have some freedom in how you do it. Choosing `Ncp` between the value you discover in (i) and 10 is a good choice. The period of the pilots, set by `Np`, should be as large to make the system efficient, but getting good channel estimates will be compromised. Selected channel amplitude and phase estimates will be plotted automatically, and the observed quality of the estimates will help you set a values for the forgetting factor α. The SEP with channel estimation will likely be better than with the ideal channel. What do you think is going on?

4.) Consider a batch mode processing where each P_e is obtained by processing say 10,000 or 100,000 bits at a time inside a `for` (or better still a `while`) loop. You keep track of the total number of bits processed and the total number of bit errors. You can set a stopping criteria when you reach a certain number of bit errors. The stopping number should be greater than 100 as with Viterbi decoding bit errors tend to come in bursts.

An important point that was not covered in class, but is briefly mentioned in the Chapter 7 notes, Example 7.5, is how you calibrate the link SNR in dB relative the operating E_b/N_0. With rate 1/2 coding for example the channel SNR needs to take into account that two code bits are transmitted for each source bit. This means at the channel the SNR in dB needs to be of the form

$$\text{SNR}_{\text{dB}} = \left(\frac{E_b}{N_0}\right)_{\text{dB}} - 10\log_{10}(2)$$

or 3 dB lower. With this understanding errors will occur quite readily. For the case of the punctured code four channel bits are sent for every three source bits, hence rate 3/4. The channel SNR calibration factor needs to changed accordingly, e.g.,

$$\text{SNR}_{\text{dB}} = \left(\frac{E_b}{N_0}\right)_{\text{dB}} - 10\log_{10}\left(\frac{4}{3}\right)$$

For the $K = 3$ code example in the notebook the rate 1/2 case results in:

```
⋮
Bits Received = 1079028, Bit errors = 98, BEP = 9.08e-05
kmax =  0, taumax = 0
Bits Received = 1089019, Bit errors = 100, BEP = 9.18e-05
*****************************************************
Bits Received = 1089019, Bit errors = 100, BEP = 9.18e-05
```

Tail end of the simulation

5 dB point

```
SNRdB = arange(2,12,.1)
# fec.conv_Pb_bound(R, dfree, Ck, SNRdB, hard_soft = 0, 1, or 2)
# 0 <==> Hard decoding bound, 1 <==> softdecoding bound, 2 <==> uncoded BPSK
Pb_uc = fec.conv_Pb_bound(1/2,5,[1,4,12,32,80,192,448,1024],SNRdB,2)
Pb_s = fec.conv_Pb_bound(1/2,5,[1,4,12,32,80,192,448,1024],SNRdB,1)
figure(figsize=(5,5))
semilogy(SNRdB,Pb_uc)
semilogy(SNRdB,Pb_s)
semilogy([4,5],[6.94e-4,9.18e-5],'r.') # expand as more sim points are collected
axis([2,12,1e-7,1e-1])
xlabel(r'$E_b/N_0$ (dB)')
ylabel(r'Symbol Error Probability')
legend((r'Uncoded BPSK',r'R=1/2, K=3, Soft',r'Simulation (100 errors)'),loc='bes
grid();
```