ECE 5625/4625 Computer Project 2
Digital Modulation and Demodulation

Introduction

In this team project you will be studying digital modulation over an additive white Gaussian noise channel. Recorded speech waveforms will be used to provide test messages to be encoded into a serial bit stream of ones and zeros. The sync pattern of scheme of Project 1 will be employed in one scenario as well as random bit streams. The entire simulation will be constructed in a Jupyter notebook. A sample Jupyter notebook containing examples and template content, plus one wave file is available in the ZIP package, `Project2_s2026.zip`, on the course Web Site. *Please do your work in Jupyter, then export as HTML, from the browser print to PDF as 8.5 x 11 pages I can grade on.*

Honor Code: The project teams will be limited to at most *two* members. Teams are to work independent of one another. Bring questions about the project to me via Slack. I encourage you to work in teams of two at the very least. Since each team member receives the same project grade, a group of two should attempt to give each team member equal responsibility. The due date for the completed project will be on or before Wednesday May 13, at 12:00 pm, 2026. Note the final exam is Tuesday, May 12, 2020 from 8:00–10:00 pm.

Overview: A general digital communication system block diagram is shown in Figure 1. The dia-

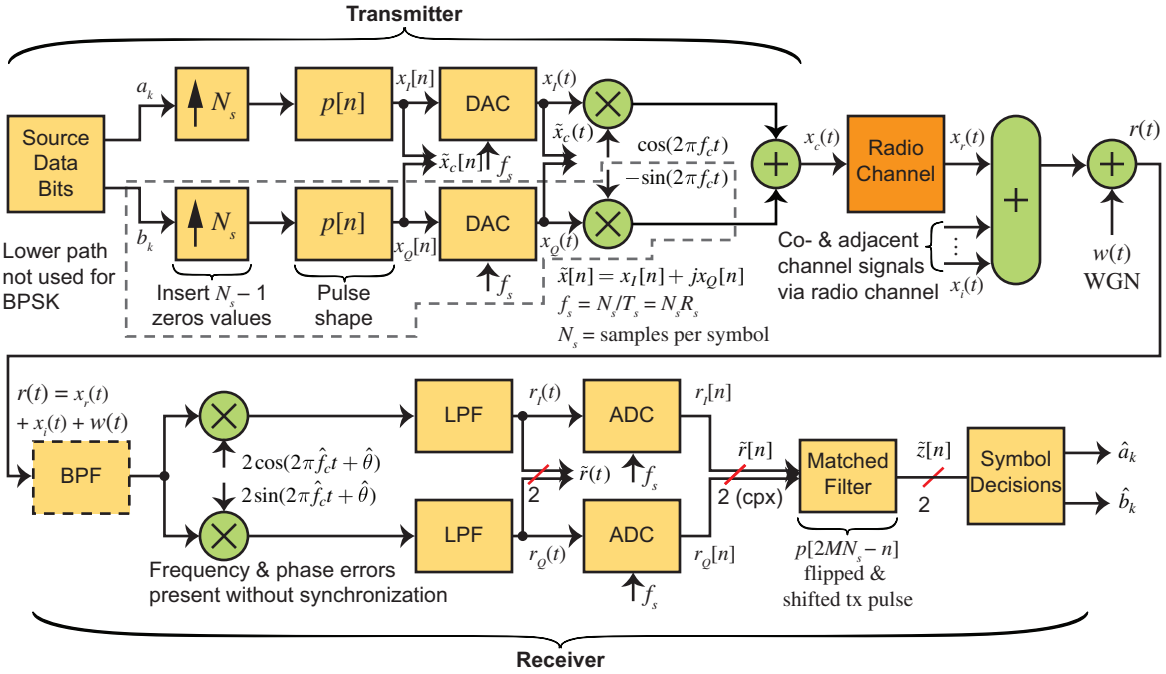

Figure 1: A top level digital communication system block diagram for both transmitting and receiving.

gram shows a digital data source consisting of bit streams a_k and b_k. Recall the discussion of quadrature modulation at the end of notes/text Chapter 4. The a_k stream ultimately modulates a cos() carrier while the b_k stream modulates a sin() carrier. The function $p[n]$ is a causal discrete-time pulse shaping function. The block with the up arrow is an upsampler, meaning $N_s - 1$ zero samples are stuffed between each (a_k, b_k) sample pair. This creates what is known as an *impulse train modulator* when combined with $p[n]$. The constant N_s is the number samples used to represent each symbol and corresponds to the actual symbol period in seconds when the sampling rate f_s is considered, i.e., $T_s = N_s/f_s$. The pulse shape, $p(t)$, as the continuous-time equivalent although non-causal as shown in Figure 2, can be as simple as a rectangle pulse of duration T_s, or a perhaps a square-root raised cosine pulse shape, extending over $\pm 6T_s$ intervals, and having a very compact spectrum, i.e., $1.25R_s$, where $R_s = 1/T_s$ is the symbol rate.

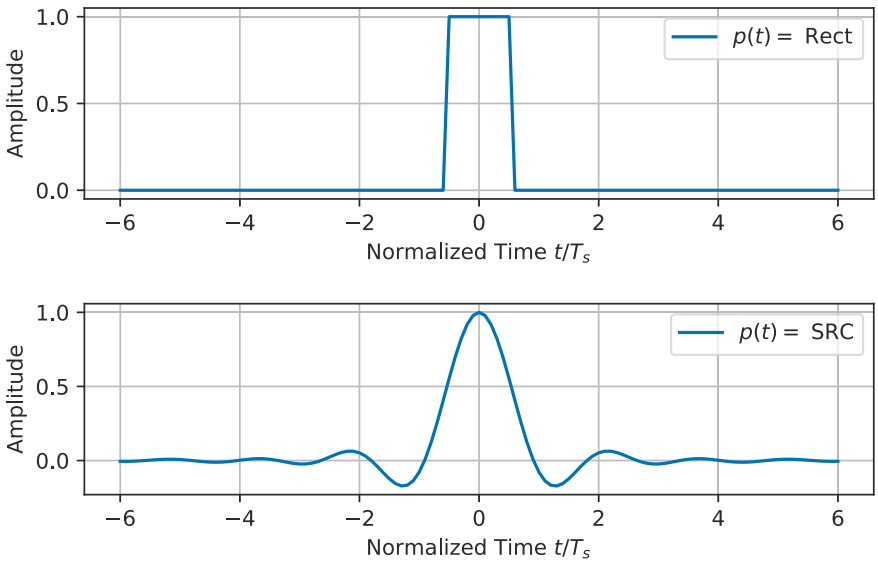

Figure 2: The popular rectangle and square-root raised cosine pulse shapes in the continuous-time domain, both non-causal just in this plot.

In general the streams a_k and b_k may each take on $\sqrt{M}$ (an integer) different amplitude levels and hence be termed *symbols* that carry more than one bit of information. This scheme is known as quadrature amplitude modulation (QAM). The values that a_k and b_k take on are traditionally

$$a_k, b_k = \pm 1, \pm 3, \ldots, \pm(\sqrt{M} - 1) \tag{1}$$

where M is the number of unique symbols. To explain further, the QAM waveform can be expressed as quadrature modulation

$$\begin{aligned} x_c(t) &= A\left[\sum_{k=-\infty}^{\infty} a_k p(t - kT_b)\cos(2\pi f_c t) - \sum_{k=-\infty}^{\infty} b_k p(t - kT_b)\sin(2\pi f_c t)\right] \\ &= A\mathrm{Re}\left\{(a_k + jb_k)e^{j2\pi f_c t}\right\} \end{aligned} \tag{2}$$

where we can view the symbols as complex valued, $c_k = a_k + jb_k$ and collectively forming a *constellation* of possible symbol locations in the complex plane. Note for BPSK we have simply $c_k = a_k$. The constellation plots of BPSK (special case $M = 2$ QAM) and $M = 16$ QAM (16QAM) are both shown in Figure 3. The transmitted signal $x_c(t)$ has carrier frequency f_c.

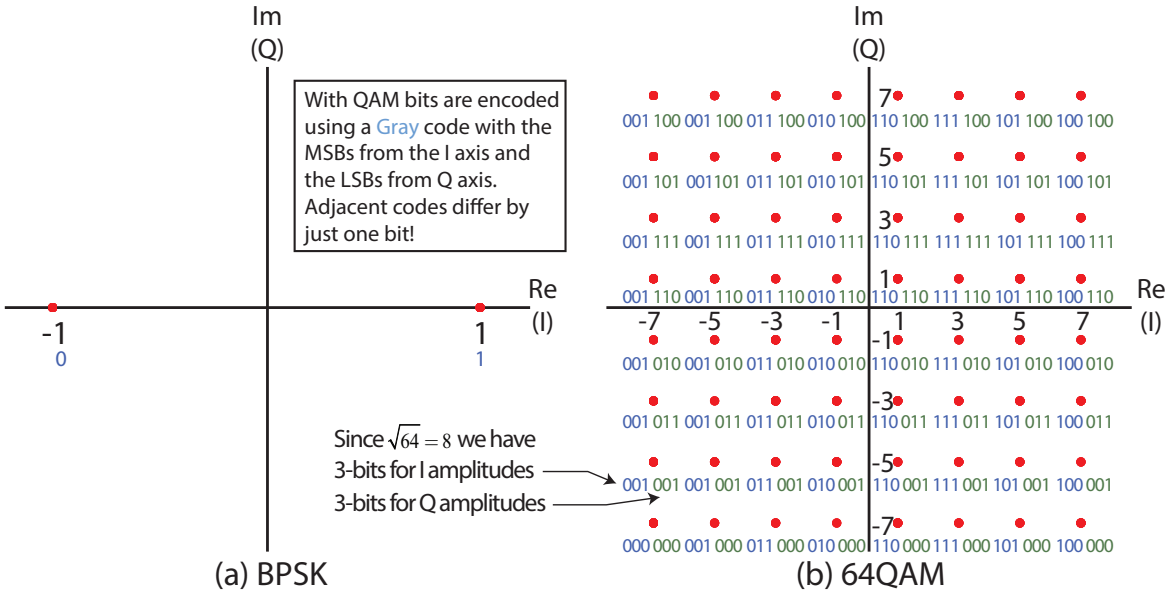

Figure 3: Constellation plots for (a) BPSK and (b) 64QAM, including the serial bit stream assignments.

QAM is used widely in cable modems[1] (DOCSIS), Wi-Fi[2], and 5G[3] cellular services. It's popularity is due to the fact that many bits per symbol are transmitted. In 256QAM you pack 8-bits per symbol, which is eight times more throughput than simple BPSK, for the same spectral occupancy! In Part II you will see that high-order QAM is more sensitive to signal impairments, so there is a price to be paid in the end.

For both BPSK and QAM a coherent receiver, as with double-sideband of Chapter 3, is required so that with the signal processing of Figure 1 the signal constellation at baseband is returned to the original orientation. The *Symbol Decisions* block, reduces to simple thresholding/slicing to form symbol estimates. Finally, symbol-to-bit mapping, returns estimates of the transmitted serial bit stream as repacked bits `[Ibits,Qbits,Ibits,Qbits,...]`. As shown in Figure 3 each symbol corresponds to $\log_2 M = 6$ bits with 3 `Ibits` and 3 bits the `Qbits`. For BPSK the decision process can be viewed as a simple 1-bit quantizer or comparator, with threshold at zero. For QAM classification rectangles surround each signal point as shown in Figure 4. Notice that bits are

1. Data over cable standard (DOCSIS). https://en.wikipedia.org/wiki/DOCSIS
2. Wi-Fi at 2.5 and 5 GHz. https://en.wikipedia.org/wiki/Wi-Fi
3. 5G. https://www.telecomtrainer.com/what-modulation-schemes-are-used-in-5g-communication/

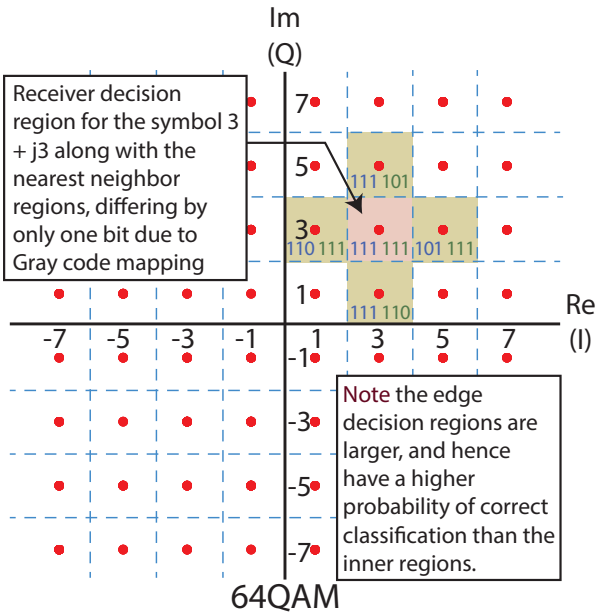

Figure 4: Rectangular decision regions used to classify the received symbol points to one of M expected values, and hence allow mapping back to serial bits.

assigned to each of the M points using Gray coding[1] which insures that the nearest neighbor decision rectangles differ by only one bit. With channel noise and other impairments present on the received signal, the Gray code mapping makes the most probable error event have only one bit error as opposed to all $\log_2(M)$ bits.

White Gaussian Noise Channel

In Project 2, unlike Project 1, a full waveform channel is being used. In a discrete-time Tx/Rx model with synchronous sampling we will typically choose `N_S` samples per symbol which corresponds to a symbol duration of T_s seconds. From the earlier discussion of QAM I want to be clear that when bits are encoded into symbols the effective bit period is a fraction of the symbol period via the relationship $T_s = T_b \cdot \log_2 M$ where M is the modulation order. The symbol period is inversely proportional to the channel symbol rate R_s. A channel is termed *additive white Gaussian noise* (AWGN) if the waveform passing through the channel is subjected to Gaussian noise having a flat (white) power spectral density over the signal's spectral bandwidth. If $x_c[n]$ is the transmitted serial symbol stream (in discrete-time form), the AWGN channel produces at the receiver

$$r[n] = x_c[n] + w[n] \tag{3}$$

where $w[n]$ is a white Gaussian noise process. The fact that $w[n]$ is Gaussian means that at any time instant n_0, we have $w[n_0]$ a Gaussian or normal random variable with zero mean and variance $N_0/2$. The fact that $w[n]$ is white also means that random variables $w[n_0]$ and $w[n_1]$,

1. Gray code mapping. https://en.wikipedia.org/wiki/Gray_code

$n_0 \neq n_1$, are uncorrelated, and the power spectrum of the process $w[n]$ has a constant spectral density, $S_w(f) = N_0/2$ W/Hz. Note that for Gaussian random variables uncorrelated implies independence, so the random variates employed are actually independent. For complex baseband digital comm signals we use the function `sigsys.cpx_awgn()` of Listing 1 to add symmetrical random variates to the complex baseband signal samples.

Listing 1: Function for introducing complex Gaussian noise for the block diagram of Figure 1.

```
def cpx_awgn(x, es_n0, ns):
    """
    Apply white Gaussian noise to a digital communications signal.

    This function represents a complex baseband white Gaussian noise
    digital communications channel. The input signal array may be real
    or complex.

    Parameters
    ----------
    x : ndarray noise free complex baseband input signal.
    EsNO : set the channel Es/N0 (Eb/N0 for binary) level in dB
    ns : number of samples per symbol (bit)

    Returns
    -------
    y : ndarray x with additive noise added.

    See Also
    --------
    cpx_awgn2

    Notes
    -----
    Set the channel energy per symbol-to-noise power spectral
    density ratio (Es/N0) in dB.

    Examples
    --------
    >>> import sk_dsp_comm.sigsys as ss
    >>> x,b, data = ss.nrz_bits(1000,10)
    >>> # set Eb/N0 = 10 dB
    >>> y = ss.cpx_awgn(x,10,10)
    """
    w = np.sqrt(ns * np.var(x) * 10 ** (-es_n0 / 10.) / 2.) * \
        (np.random.randn(len(x)) + 1j * np.random.randn(len(x)))
    return x+w
```

At the receive end of the channel the noisy waveform values $r[n]$ need to be converted back to hard decision values of $\pm 1, \pm 3, \ldots, \pm(\sqrt{M} - 1)$. For the special case of BPSK ($M = 2$) the values are just ± 1. A matched filter is used to optimally filter the signal plus noise. The fact that noise is present means that some bit errors will occur. This constitutes channel bit errors. From a performance standpoint, we are interested in the ratio of energy per received bit, E_b, to the received noise power spectral density ratio, N_0, or E_b/N_0. For the one sample per bit discrete-time channel model, $E_b = (\pm 1)^2 = 1$. Since $w[n]$ Gaussian, it can be shown that the probabil-

ity of a bit error is

$$P_{e,\,\text{thy}} = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) = Q\left(\sqrt{2 \cdot 10^{\text{SNR}_{\text{dB}}/10}}\right) \tag{4}$$

where $Q(\ )$ is the tail area of a zero mean unit variance normal/Gaussian PDF

$$Q(x) = \frac{1}{\sqrt{2\pi}}\int_u^\infty e^{-v^2/2}dv \tag{5}$$

Note that it is common practice to refer to E_b/N_0 as the channel signal-to-noise ratio (SNR), and to use dB values for this ratio, i.e.,

$$\text{SNR}_{\text{dB}} = \left(\frac{E_b}{N_0}\right)_{\text{dB}} = 10\log_{10}\left(\frac{E_b}{N_0}\right) \tag{6}$$

In Python you can use `sk_dsp_comm.digitalcom.Q_fctn`, as shown in Listing 2 for this binary bit error probability (BEP) calculation. You can also use functions in `scipy.stats.norm`.

Listing 2: Simple binary ($M = 2$) BEP calculation using the Gaussian Q function or the survival function in `scipy.stats`.

```
# Use the Q() function in digitalcom to find Pe_thy
EbN0_dB = 5
Pe_thy = dc.Q_fctn(sqrt(2*10**(EbN0_dB/10)))
print('Pe_thy = %1.3e' % Pe_thy)
```

```
Pe_thy = 5.954e-03
```

```
# Use the survival function sf() = 1 - cdf() = Q() in scipy.stats.norm to find Pe_thy
from scipy.stats import norm
EbN0_dB = 5
Pe_thy = norm.sf(sqrt(2*10**(EbN0_dB/10)))
print('Pe_thy = %1.3e' % Pe_thy)
```

```
Pe_thy = 5.954e-03
```

QAM Signal Sets and Waveform Modeling

We now drill into QAM modulation and demodulation at the waveform level. Working at the waveform level is where things get interesting. A baseband QAM signal (including BPSK when $M = 2$) is easily formed by *impulse train modulating* a serial bit stream (perhaps from PCM encoding). As mentioned earlier the impulse train modulator stuffs $N_s - 1$ zero samples in between each input. This is also known as upsampling by N_s. The upsampled signal is then sent through an FIR pulse shaping filter as shown in Figure 5. Baseband QAM exits the pulse shape filter and is frequency translated to a carrier frequency f_k relative to a sampling rate of f_s Hz. Note also that the symbol period is $T_s = N_s/f_s$ s so the symbol rate is $R_s = 1/T_s = f_s/N_s$ sps and due to the encoding, the serial bit rate $R_b = \log_2(M)R_s$. Hence high-order QAM (M large) is very bandwidth efficients as $\log_2(M)$ bits are sent per QAM symbol. In a physical system $x_c[n]$, now a complex signal, can be sent to a pair of DAC's and further up converted to an actual RF/ microwave carrier. **Note**: The scheme of Figure 1 uses a DAC and puts the baseband pulse shaped signal directly on the analog carrier signal. The desire here is to keep signals in the discrete-time domain for further simulation analysis.

The purpose of the pulse shape filter is to control the spectrum of the signal. A simple option is to produce rectangle pulses of duration T_s to represent each symbol. The transmitted signal spectrum will have the form $T_s\text{sinc}^2(fT_s)$, which has very large sidelobes and hence have a large spectral *footprint*. If you try to filter the signal to reduce the bandwidth the signal will now contain *intersymbol interference* (ISI), which smears energy from adjacent bits together. The ISI impairs the overall link performance by increasing the probability of making a bit error. A better approach is to use a *Nyquist* pulse shape (see p. 227 of the text [1]). A Nyquist pulse shape insures zero ISI and also allows for a compact spectrum, allowing more QAM signals to be packed close together over a designated band of frequencies. A popular end-to-end pulse shape is the *raised cosine* (RC) and the related *square root raised cosine* (SRC or RRC). The SRC pulse is the best choice as the shaping filter can be spread equally between the transmitter and the receiver to not only insure zero ISI, but also gain AWGN immunity in an optimal way. Figure 6 below shows this Tx/Rx con-

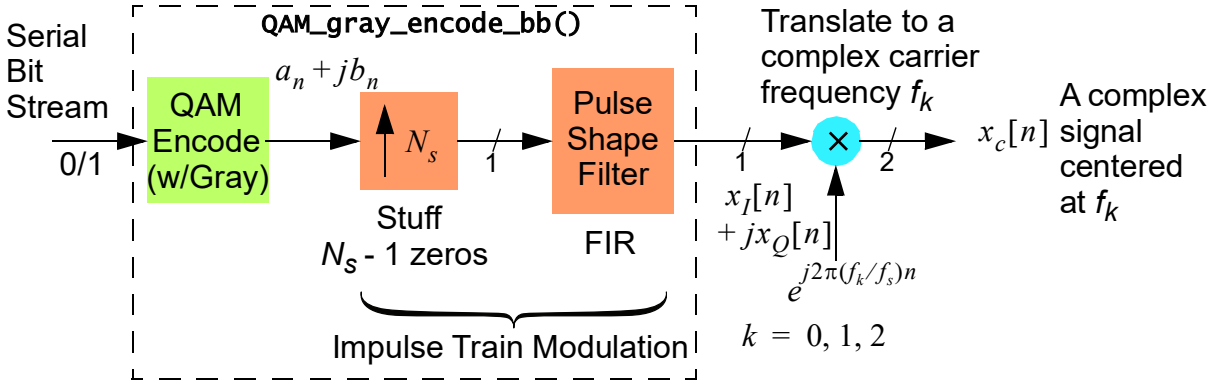

Figure 5: Impulse train modulator implemented in `QAM_gray_encode()` and complex frequency translator for QAM as well as the special case of BPSK ($M = 2$).

figuration [1]. The second SRC filter serves as the matched filter in this receiver. In Python all of

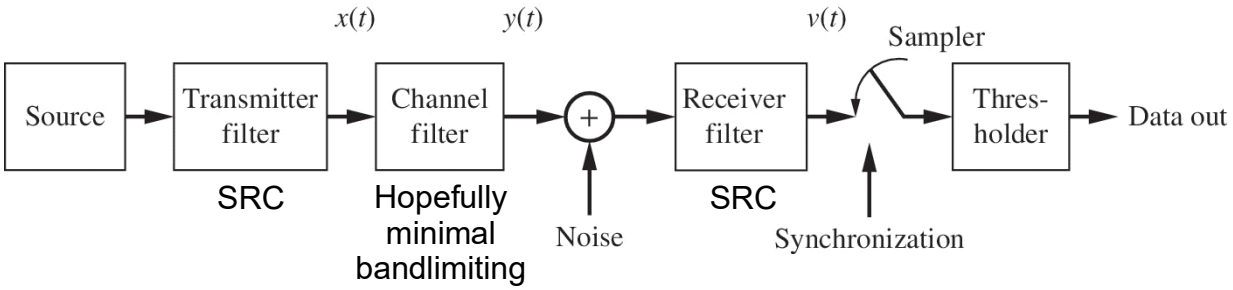

Figure 6: A simplified transceiver block diagram showing how pulse shaping is spread between the transmitter and receiver [1].

this is pre-build using functions available in `sigsys.py` and `digitalcom.py`. When synchronization is needed functions are available in `synchronization.py`.

Example: QAM Transmitter Waveform Modeling and the Power Spectrum

As an example the code of Listing 3 generates baseband 64-QAM with SRC pulse shaping and then translates the signal to $f_c = 4R_s$ for the case of $N_s = 16$ samples per bit. In Figure 7 the power spectrum is plotted.

Listing 3: Creating a 64-QAM complex baseband waveform with and without complex frequency translation and then plotting its power spectrum.

```
#qam_gray_encode_bb(N_symb,Ns,M=4,pulse='rect',alpha=0.35,ext_data=None)
Ns = 16; M = 64
xbb1,b,data = dc.qam_gray_encode_bb(10000,Ns,M,'src')
n = arange(0,len(xbb1))
# Translate baseband to fc1/fs = 4.0/16
# Relative to Ns = 16 samps/bit & Rb = 1 bps
xc1 = xbb1*exp(1j*2*pi*4.0/Ns*n)

figure(figsize=(6,2.5))
psd(xbb1,2**10,Ns);
psd(xc1,2**10,Ns);
ylim([-60,5])
xlabel(r'Normalized Frequency $f/R_s$ (note: $M=64 \Rightarrow R_b = 6R_s$)')
legend((r'Baseband',r'At $f_c = f_s/4$'))
title(r'Spectrum SRC Pulse Shaped 64-QAM at $f_{c1} \
      = 4R_s$ and $N_s = %d$' % Ns);
tight_layout()
savefig('p2_fig7.pdf')
```

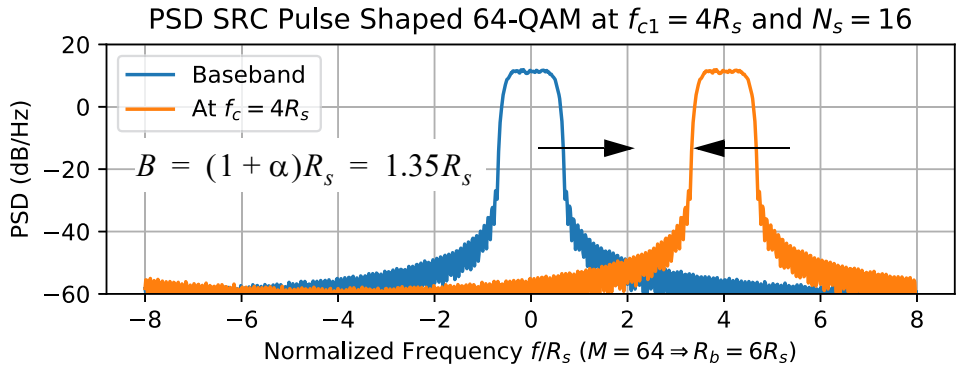

Figure 7: Sample SRC shaped 64QAM spectrum shifted from 0 Hz to $f_{c1} = 4R_s$.

As Figure 1 shows, the receiver undoes the action of the transmitter, but in practice must also deal with synchronization issues. In general the receiver clock is asynchronous with respect to the transmitter clock and there is also carrier phase and frequency uncertainty. In this project we assume that both carrier frequency and phase is perfect and symbol timing is perfect, that is the Tx and Rx clocks are synchronous and coherent carrier reduces the frequency rotation back to zero Hz. A simplified receiver block diagram is shown in Figure 8. The only remnant of carrier recovery/coherent demodulation is the complex multiply by $e^{j\theta[n]}$ so that carrier tracking loop phase noise can be modeled in a later simulation task.

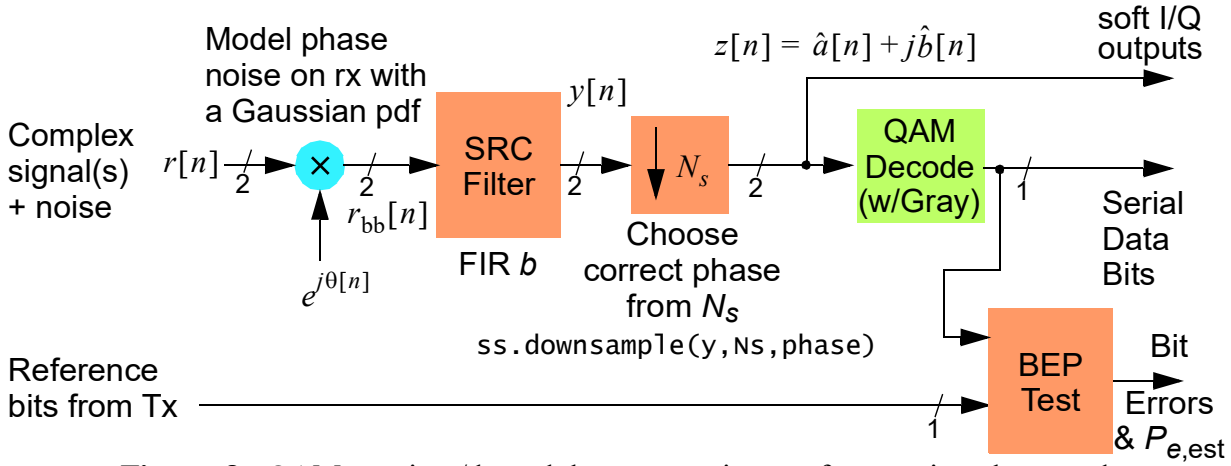

Figure 8: QAM receiver/demodulator assuming perfect carrier phase and symbol timing, but possible phase noise on the coherent carrier tracking.

Example: A One Sample Per Symbol Bit Error Probability Test

With the software building blocks configured as shown in Figure 8 there are many options for digital communications studies. A scatter plot of $\hat{a}[n] + j\hat{b}[n]$ gives you some idea of how noise, sampling error, and other impairments will introce symbol/bit errors. The serial output from the QAM Decode block, `qam_gray_decode()` can drive the PCM decoder to complete a speech data link using QAM. Probability of bit error (BEP) studies can be conducted by sending the data bits

into the BEP testing function `digitalcom.BPSK_BEP()`, which is useful comparing any 0/1 binary date sequences, not just BPSK:

```
N_bits, Nbit_errors = dc.BPSK_BEP(tx_data,rx_data,Ncorr = 1024,
                                  Ntransient = 0)
```

where `tx_data` and `rx_data` are the 0/1 data bits transmitted and received respectively. Since the processing chain involves delays transmit and receive filters, error detection requires a time alignment of the two bit streams before error detection and error counting can actually occur. The function `dc.BPSK_BEP()` automates this by *cross-correlating* the input arrays to find the correct code alignment. The optional argument `Ncorr` sets the search interval, and the fourth argument can be used to skip over initial bits in the receive array where transients may exist.

A simple 64-QAM example is shown in Listing 4 with a scatter plot of the 64-QAM constellation in noise shown in Figure 9.

Listing 4: A one sample per symbol (`Ns=1`) 64-QAM simulation with additive white Gaussian channel noise (AWGN) via `cpx_AWGN` followed by plotting of the noisy constellation and finally BEP estimation using `BPSK_BEP`.

```
M = 64
Nsymb = 100000
Ns = 1 # With Ns=1 there is no need for the matched filter
EbN0_dB = 15
EsN0_dB = 10*log10(log2(M))+ EbN0_dB
print('Eb/N0 = %4.2f dB and Es/N0 = %4.2f dB' % (EbN0_dB,EsN0_dB))
xbb,b,data = dc.qam_gray_encode_bb(Nsymb,Ns,M,'src')
# Enter the comm channel by adding noise
rbb = dc.cpx_AWGN(xbb,EsN0_dB,Ns)
```

```
Eb/N0 = 15.00 dB and Es/N0 = 22.78 dB
```

```
# Plot the 64-QAM constellation points as a scatter plot
Npts = 2000
scat_data = rbb
plot(scat_data[:Npts].real,scat_data[:Npts].imag,'r.')
axis('equal')
title('IQ Scatter Plot')
ylabel(r'Quadrature')
xlabel(r'In-Phase')
grid();

# Decode the QAM symbols back to a serial bit stream
data_hat = dc.qam_gray_decode(rbb,M)
Nbits,Nerrors = dc.BPSK_BEP(data,data_hat)
print('BEP: Nbits = %d, Nerror = %d, Pe_est = %1.3e' % \
      (Nbits, Nerrors, Nerrors/Nbits))
```

```
kmax =  0, taumax = 0BEP: Nbits = 600000,
Nerror = 498, Pe_est = 8.300e-04
```

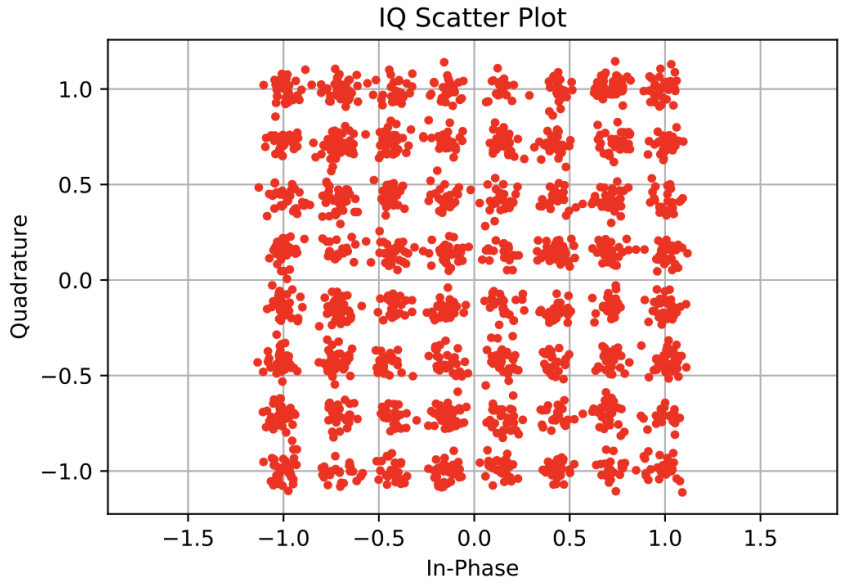

Figure 9: A one-sample per symbol 64QAM simulation using 100,000 bits with additive noise making the received $E_b/N_0 = 15$ dB.

The above example shows how a simulation using one sample per QAM symbol is run to pass serial data bits from input to output, and finally estimate the bit error probability (BEP), via the ratio

$$\hat{P}_{e,\,\text{est}} = \frac{N_{\text{bit errors}}}{N_{\text{bits total}}} \tag{7}$$

by using `BPSK_BEP()` to count errors between the transmitted noise free bit stream and the received bit stream. In (7) N_{errors} is formed from the decoded serial bit stream. Recall from Figure 4 that decision regions are set up around each nominal constellation point to decode the received point back to its 6-bit (in general $\log_2(M)$-bit) representation, ordered as I-bits followed by Q-bits. The clustering of received points around the nominal constellation point is often referred to as a *Gaussian cloud*. As E_b/N_0 gets smaller the cloud grows larger and the chance of making a symbol classification/detection errors increases rapidly.

Earlier you saw a theoretical expression, equation (4), for the bit error probability for antipodal/BPSK signaling in additive Gaussian noise. For QAM, as simulated above, a related theoretical formula exists, but the formula is more complex than (4). A simple approximate QAM BEP formula, valid when the SNR is well above 0 dB, is

$$P_{e,\,\text{thy}} \cong \frac{4}{\log_2(M)}\left(1 - \frac{1}{\sqrt{M}}\right) Q\left(\sqrt{\frac{3}{M-1}} \cdot \frac{E_b}{N_0} \cdot \log_2(M)\right) \tag{8}$$

This formula, along with the original binary formula ($M = 2$) is coded in Python for $M = 2, 4, 16, 64$, and 256, in Listing 5.

Listing 5: A function for calculating the theoretical BEP for QAM having

$M = 2$ (binary/BPSK) and $M = 4$, 16, 64 and, 256.

```
def qam_bep_thy(SNR_dB,M,EbN0_Mode = True):
    """
    Approximate the bit error probability of QAM assuming Gray encoding

    Mark Wickert November 2018
    """
    if EbN0_Mode:
        EsN0_dB = SNR_dB + 10*np.log10(np.log2(M))
    else:
        EsN0_dB = SNR_dB
    if M == 2:
        BEP = Q_fctn(np.sqrt(2*10**(EsN0_dB/10)))
    elif M > 2:
        SEP = 4*(1 - 1/np.sqrt(M))*Q_fctn(np.sqrt(3/(M-1)*10**(EsN0_dB/10)))
        BEP = SEP/np.log2(M)
    return BEP
```

Compare the simulation results from in Listing 4 with theory via `dc.QAM_BEP_thy` in Listing 6.

Listing 6: Theoretical BEP for $M = 64$ with $E_b/N_0 = 15$ dB.

```
EbN0_dB = 15
P_e_thy = dc.qam_bep_thy(EbN0_dB,64)
print('P_e_bit_thy = %4.2e' % P_e_thy)
```

```
P_e_bit_thy = 7.72e-04
```

The results are close, considering the statistical confidence associated with observing 498 error events.

64-QAM Waveforms and the Coherent Receiver Model

The objective here is to build up a coherent receiver model that follows Figure 8. The first step is generating the transmit waveform and the channel which includes additive noise. In Listing 3 we had `Ns` greater than one to produce an actual pulse-shaped waveform from `QAM_gray_encode_bb()`. To explore this further rework the Listing 4 code with `Ns = 10` as shown in Listing 7 below.

Listing 7: Create a 64-QAM waveform with `rect` pulse shaping and the waveform passed through an AWGN channel having $E_b/N_0 = 100$ dB.

```
M = 64
Nsymb = 500
Ns = 10 # With Ns=10 we have a waveform
EbN0_dB = 100 # Make the addive noise negligible for now
EsN0_dB = 10*log10(log2(M)) + EbN0_dB
print('Eb/N0 = %4.2f dB and Es/N0 = %4.2f dB' % (EbN0_dB,EsN0_dB))
xbb,b,data = dc.qam_gray_encode_bb(Nsymb,Ns,M,'rect')
# Enter the comm channel by adding noise
rbb = dc.cpx_AWGN(xbb,EsN0_dB,Ns)
```

```
Eb/N0 = 100.00 dB and Es/N0 = 107.78 dB
```

In this code the pulse shaping is set to `rect` so that the waveforms are easy to look at. Li sets up the plotting of the inphase (I) and quadrature (Q) waveforms, rescaled back to the original ampli-

tude values first seen in Listing 8. The plot (two subplots) are shown in Figure 10.

Listing 8: I and Q waveforms for 64-QAM with pulse shaping, rescaled to ± 1, $\pm 3 \ldots$, $\pm(\sqrt{M}-1)$ amplitude levels as found in .

```
Nplot = 100
subplot(211)
plot(rbb[:Nplot*Ns].real*7) # Remove the unity amplitude scaling
                            # to see the +/-1, +/-3, +/-5, +/-7
Nplot = 100
subplot(211)
plot(rbb[:Nplot*Ns].real*7) # Remove the unity amplitude scaling to see the +/-1, +/-3,
...
ylabel(r'Amplitude')
xlabel(r'Sample Index $n$')
title(r'Scaled Inphase or $I[n]$ Signal Transmitted at Baseband')
subplot(212)
plot(rbb[:Nplot*Ns].imag*7)
ylabel(r'Amplitude')
xlabel(r'Sample Index $n$')
title(r'Scaled Quadrature or $Q[n]$ Signal Transmitted at Baseband')
tight_layout()
```

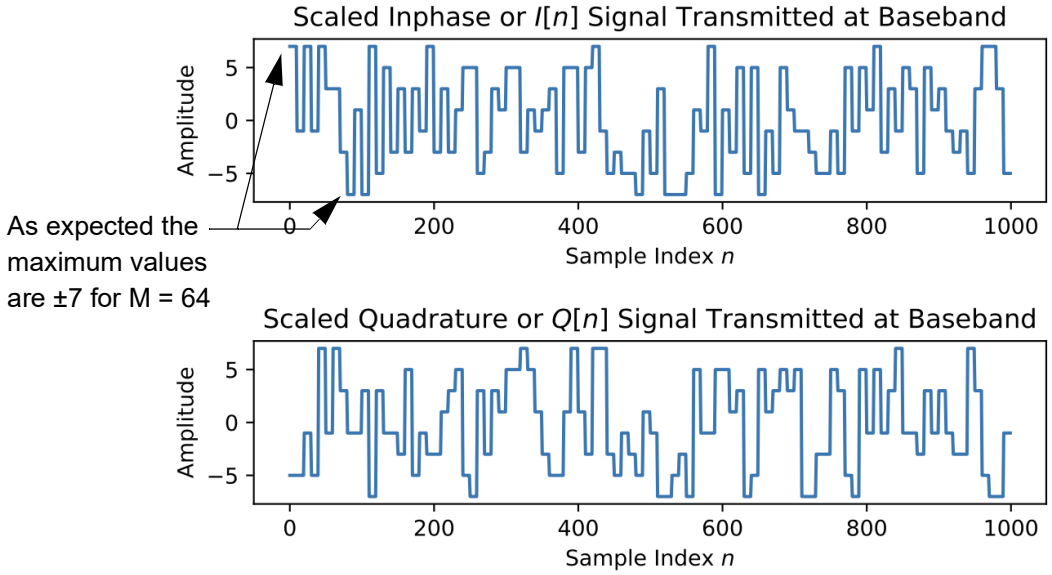

Figure 10: Transmitted IQ waveforms of 64-QAM using `rect` pulse shaping and $N_s = 10$ samples per symbol.

Including the Matched Filter to Mitigate Noise

Inside the receiver of Figure 8 we see the first step, following the phase noise model, is the *matched filter*, which optimally filters additive noise while keeping signal maximized at some sample point along the $[0, N_s]$ symbol time interval `mod(Ns)`. Here the matched filter takes the same form as the transmitter pulse shaping filter. The filter, in the form of FIR filter coefficients, is conveniently returned by `QAM_gray_encode_bb()` as the array `b`. Recall the `rect` and `src` pulse shapes are shown in Figure 2. Matched filtering is carried out in Listing 9.

Listing 9: Matched filter implementation using signal.lfilter().

```
# Enter the receiver by passing through the matched filter
# Note carrier frequency ofset and phase tracking would likely be needed, but here
# both are zero to keep things simple.
y = signal.lfilter(b,1,rbb)
```

Choosing the Proper Once Per Symbol Sampling Instant

The symbol synchronization operation requires sampling the matched filter output once per symbol, such that the maximum SNR is achieved, and hence the minimum BEP is obtained. Here we use `ss.downsample(y,Ns,phase)` to do this. The optional third argument, `phase`, is chosen from $[0, N_s - 1]$, which is exactly what we need. In practice a PLL symbol synchronizer performs this automatically.

To best visualize the optimum sample instant we use the *eye plot* as discussed in Appendix A. The code for producing an eye plot is given in List.

Listing 10: Using the function `dc.eye_plot()` to produce an eye plot of the real (shown here) or imaginary part waveform, shows the resulting eye plot.

```
dc.eye_plot(y.real,2*Ns,12*Ns) # The 2*Ns spans two symbol period. The 12*Ns removes
                               # the pulse shaping filter transients
plot([9,9],[-1.2,1.2],'g--')
title(r'Eyeplot of Inphase with Rect Pulse Shaping')
```

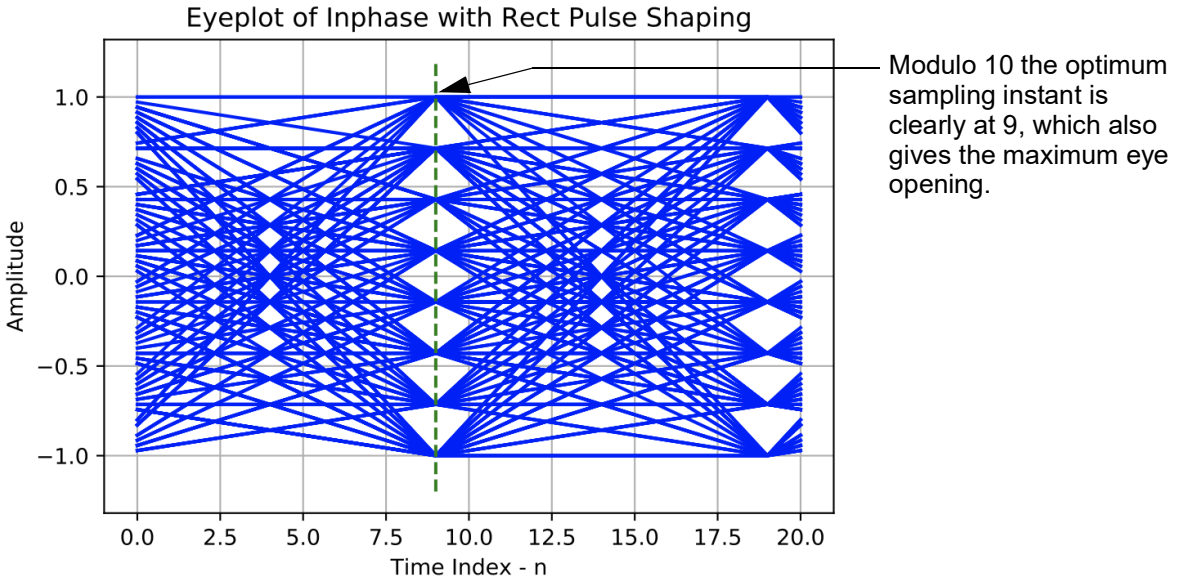

Figure 11: Eye plot of the matched filter output for `rect` pulse shaped 64-QAM with `Ns = 10` (here the waveform amplitude is scaled to a peak value of ±1.

Knowing the optimum sampling instant we continue with receiver down sampling processing in Listing 11. The scaled samples are plotted using `'r.'` to clearly see the values as expected.

Listing 11: The receiver once per symbol sampling and plotting of the sampled values, to ultimately allow symbol decoding.

```
# Symbol synchronize manually
```

```
z = ss.downsample(y,Ns,9)
figure(figsize=(6,3))
plot(z[:2*Nplot].real*7,'r.') # Again scale to see that the sample values are aligning
                              # with the expected unscaled tx values.
title(r'Scaled Optimal Symbol Spaced Samples')
ylabel(r'Inphase Symbol Samples')
xlabel(r'Sample Index')
grid();
```

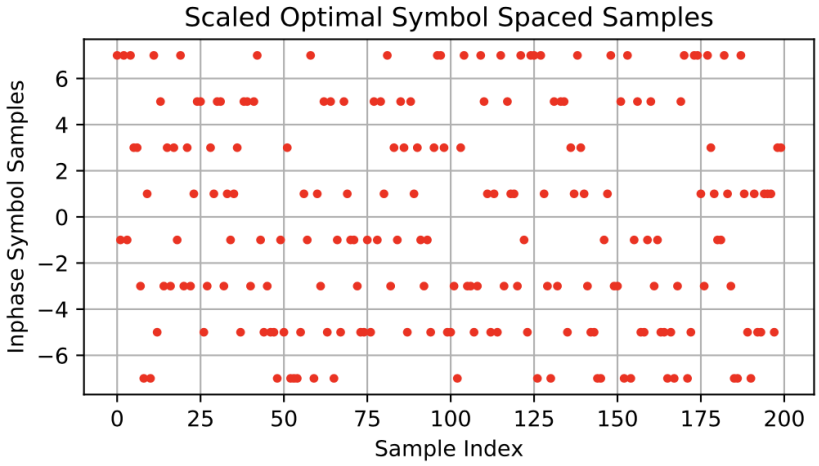

Figure 12: Scaled down sampler outputs showing for the inphase channel, now ready for the symbol decoding back into a serial bit stream.

Putting the Entire Receiver Together for the SRC Pulse Shaping Case

Now put the entire receiver together in one code block for the less easy to visualize square-root raised cosine (`src`) shape. Remember the SRC shape has a very compact spectrum, and hence is preferred over the `rect` to allow tight spectrum packing. The complete receiver code listing, including generation of the eye plot and the sampler outputs versus time is given in Listing 12. The corresponding plots are given in Figure 13.

Listing 12: Complete receiver list with plotting function of the eye plot and the sampler output plot.

```
M = 64
Nsymb = 500
Ns = 10 # With Ns=10 we have a waveform
EbN0_dB = 50 # Make the addive noise negligible for now
EsN0_dB = 10*log10(log2(M)) + EbN0_dB
print('Eb/N0 = %4.2f dB and Es/N0 = %4.2f dB' % (EbN0_dB,EsN0_dB))
xbb,b,data = dc.qam_gray_encode_bb(Nsymb,Ns,M,'src')
# Enter the comm channel by adding noise
rbb = dc.cpx_awgn(xbb,EsN0_dB,Ns)
# Enter the receiver by passing through the matched filter
# Note carrier frequency offset and phase tracking would likely be needed, but here
# both are zero to keep things simple.
y = signal.lfilter(b,1,rbb)
# Timeout for an eye plot to check timing
figure(figsize=(6,2))
dc.eye_plot(y[:10000].real,2*Ns,12*Ns) # 12*Ns removes the filter transients
plot([0,0],[-1.2,1.2],'g--')
```

```
plot([10,10],[-1.2,1.2],'g--')
title(r'Eyeplot of Inphase with SRC Pulse Shaping')
# Symbol synchronize manually
z = ss.downsample(y,Ns,1)
figure(figsize=(6,3))
# Scale so the peak is +/- (sqrt(M) - 1).
plot(z[:2*Nplot].real/(std(z) * sqrt(3/(2*(M-1)))),'r.')
title(r'Scaled Optimal Symbol Spaced Samples')
ylabel(r'Inphase Symbol Samples')
xlabel(r'Sample Index')
ylim([-8,8])
grid();
tight_layout()
```

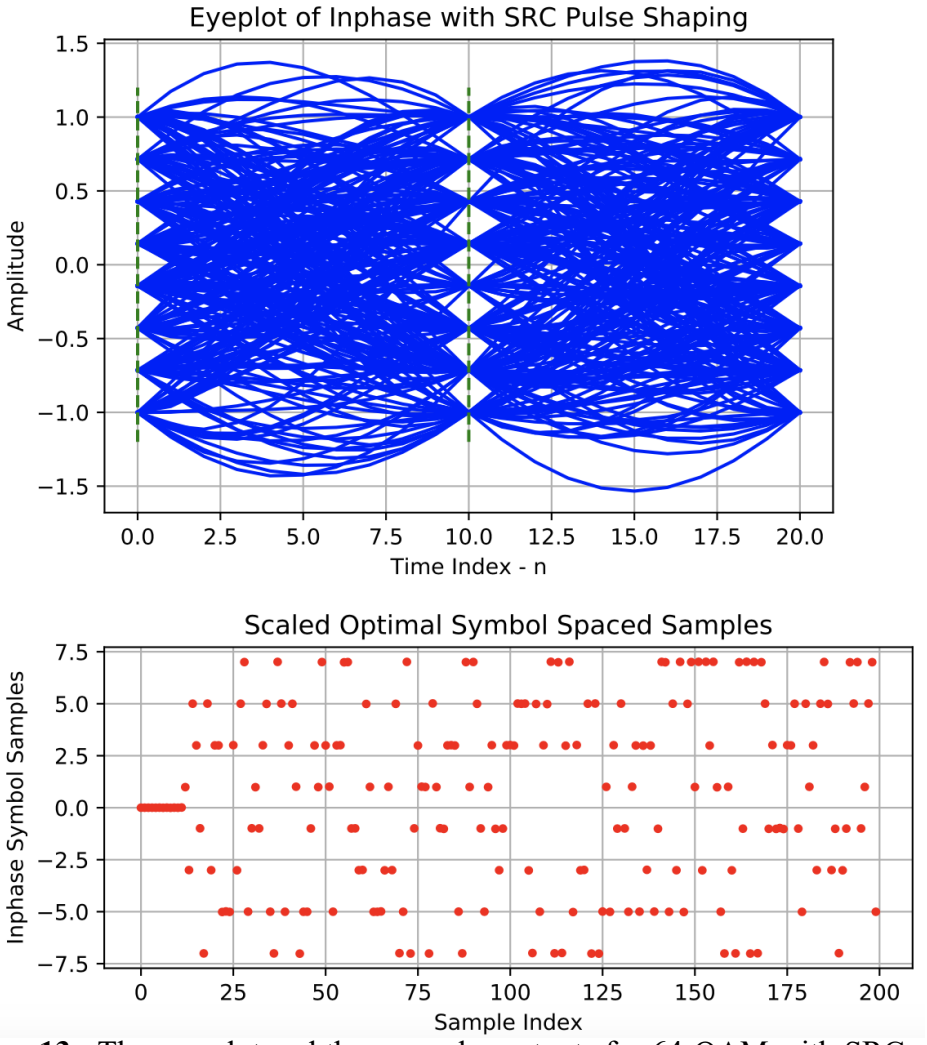

Figure 13: The eye plot and then sampler outputs for 64-QAM with SRC pulse shaping and `Ns = 10`.

Here in particular the optimum sampling instant is now `phase = mod(10,Ns) = 0`. The eye plot makes it clear that the `src` pulse shaped waveform is much smoother, and has a more compact spectrum (recall Figure 7). Note a constellation scatter plot similar to Figure 9 can be created at this point, which is denoted as the soft IQ outputs in Figure 8.

Symbol Decoding and BEP Estimation with Waveforms

Here we feed the reference bit stream output, `data`, from `QAM_gray_encode_bb` along with the decoded bit stream `QAM_gray_decode` into `BPSK_BEP`. For the SRC pulse shaped waveform the net filter delay from the encoder and the matched filter is 12 symbols. A cross-correlation is performed inside `BPSK_BEP` to align the two input bit streams for error detection. This of course assumes there are not too many errors.

Listing 13: QAM decoding using the soft IQ values input to `QAM_gray_decode()` and then BEP estimation using `BPSK_BEP()`.

```
data_hat = dc.qam_gray_decode(z,M)
Nbits,Nerrors = dc.bpsk_bep(data,data_hat)
print('BEP: Nbits = %d, Nerror = %d, Pe_est = %1.3e' % \
      (Nbits, Nerrors, Nerrors/Nbits))
```

```
kmax =  0, taumax = 72
BEP: Nbits = 2928, Nerror = 0, Pe_est = 0.000e+00
```

Essentially zero bit errors at very high E_b/N_0 is the expected result.

Example: 64-QAM Waveform with PCM Audio Loop Through

To send audio through the 64QAM link using the PCM encoder/decoder pair, you will use sync pattern (SP) insertion at the transmitter and SP detection and removal at the receiver. The functions from Project 1 have been enhanced slightly. The three functions are `to_sp_pcm_stream()` at the transmitter and find_sync_idx() and return_pcm_stream() at the receiver. These functions are included in the Project 2 Jupyter notebook and also given in Listing 14.

Listing 14: SP related transmit and receiver functions, `to_pcm_stream()`, `find_sync_idx()`, and `return_pcm_stream()`, updated from Project 1.

```
def to_sp_pcm_stream(pcm_raw_bits,sp,pcm_frm_len=500*8,rand_shift=True):
    """
    Generate PCM frames with a sync pattern, sp, at the start of every
    group of pcm_frm_len n_bits PCM words. PCM encoding is included.
    A random time delay in integer samples is pre-pended to PCM with SP
    frames output when rand_shift is True (default)

   Parameters:
   pcm_raw_bits = raw PCM bits from encoding
             sp = sync pattern
    pcm_frm_len = number PCM bits per frame, defaul is 500*8
     rand_shift = apply a time to the output between [0,500] samples

    Returns:
    out_sp_pcm_stream as 0/1 bits

    Mark Wickert April 2026
    """
    # Random time shift on interval [0,500] samples
    R_time_shift = random.randint(0,501,1)[0]
    if rand_shift:
        print(f'R_time_shift sample = {R_time_shift}')
    else:
```

```
        print(f'R_time_shift sample = 0')
    n_pcm_raw_bits = len(pcm_raw_bits)
    # Round down to make all frames full, discard partial frame pcm bits
    n_frm = int(floor(n_pcm_raw_bits/(pcm_frm_len)))
    pcm_raw_bits = pcm_raw_bits[:n_frm*pcm_frm_len]
    # total_pcm_bits including the sp bits
    n_total_pcm_bits = (len(sp) + pcm_frm_len) * n_frm
    out_sp_pcm_stream = zeros(n_total_pcm_bits)
    total_frm_len = len(sp) + pcm_frm_len
    for k in range(n_frm):
        # fill output bit stream with sp + pcm raw bits
        out_sp_pcm_stream[k*total_frm_len:(k+1)*total_frm_len] = \
                  hstack((sp,pcm_raw_bits[k*pcm_frm_len:(k+1)*pcm_frm_len]))
    if rand_shift:
        # Prepend random time delay samples
        out_sp_pcm_stream = hstack((zeros(R_time_shift),out_sp_pcm_stream))
    return out_sp_pcm_stream.astype('int16')

def find_sync_idx(pcm_sp_buff,sp,pcm_frm_len=500*8,cplot=False):
    """
    Extract the out_pcm_bit_stream PCM start index

    Parameters:
    pcm_sp_buffer = frames containing SP + PCM words
               sp = the SP pattern as [0.1] bits, nominal 68 bits
      pcm_frm_len = number of PCM words per frame, default 500*8
            cplot = True to plot SP cross correlation, default is False

    Returns:
    idx_find_sp[0] + 1 of the first SP found in the input
    """
    # Cross-correlation with SP as FIR coefficients in a linear filter,
    # (sequence reversed for correlation via convolution)
    # Level shifter from [0,1] to [-1,1]
    pcm_sp_buff_corr = signal.lfilter(2*sp[::-1]-1,1,2*pcm_sp_buff.astype('float64')-
1)
    if cplot:
        plot(pcm_sp_buff_corr)
        xlabel(r'Correlation Index')
        ylabel(r'Cross Corr')
        title(f'Correlation peaks of {len(sp)} expected')
        grid()
    idx_find_sp = np.nonzero(np.ravel(pcm_sp_buff_corr > len(sp)-1))[0]
    print(f'Found {len(idx_find_sp)} sp correlation peaks.')
    if (len(idx_find_sp) >= 2) and (idx_find_sp[1] - idx_find_sp[0] ==
len(sp)+pcm_frm_len):
        print(f'Found two correlations spaced by pcm_frm_len + sp_len,')
        print(f'i.e., {idx_find_sp[1] - idx_find_sp[0]} bits.')
        # Add 1 to make the returned index the first payload bit
    return idx_find_sp[0] + 1

def return_pcm_stream(out_sp_pcm_stream,idx_find_sp,sp,pcm_frm_len=500*8):
    """
    Generate PCM frames with the sync pattern removed following

    Parameters:
    out_sp_pcm_stream = PCM stream with sync pattern embedded
                   sp = sync pattern
         pcm_frm_len = number of PCM bits per frame
```

```
    Returns:
       out_pcm_stream = 0/1 logic values

    Mark Wickert April 2026
    """
    n_frm = int(floor(len(out_sp_pcm_stream[idx_find_sp:])/(len(sp) + pcm_frm_len)))
    out_sp_pcm_stream_trim = out_sp_pcm_stream[idx_find_sp:]
    out_pcm_stream = zeros(n_frm*pcm_frm_len)
    total_frm_len = len(sp) + pcm_frm_len
    for k in range(n_frm):
        out_pcm_stream[k*pcm_frm_len:(k+1)*pcm_frm_len] = \
                  out_sp_pcm_stream_trim[k*total_frm_len:(k+1)*total_frm_len - len(sp)]
    return out_pcm_stream
```

We will also see how qam_gray_encode_bb can work with an externally supplied bit stream, as opposed to its internally generated random bit stream. The complete simulation is given in Listing 15.

Listing 15: End-to-end audio loop through test using 64-QAM with SRC pulse shaping, 10 bits per symbol, and very high E_b/N_0.

```
Nstart = 20000
N = 60000; # number of speech samples to process
fs,m1 = ss.from_wav('OSR_uk_000_0050_8k.wav')
m1 = 2*m1[Nstart:Nstart+N]
N_PCM = 8 # PCM encode with 8 bits per audio sample
data_ext = dc.pcm_encode(m1,N_PCM) # Obtain external data bits for QAM_gray_encode
sp_data_ext = to_sp_pcm_stream(data_ext,sp_68,rand_shift=False)
M = 64 # QAM modulate using a 64 point constellation (8x8 points)
Ns = 10 # With Ns=10 we have a waveform
EbN0_dB = 100 # Make the addive noise negligible for now
EsN0_dB = 10*log10(log2(M)) + EbN0_dB
print('Eb/N0 = %4.2f dB and Es/N0 = %4.2f dB' % (EbN0_dB,EsN0_dB))
xbb,b,data_ext_trim = dc.qam_gray_encode_bb(None,Ns,M,'src',ext_data=data_ext)
# Enter the comm channel by adding noise
rbb = dc.cpx_awgn(xbb,EsN0_dB,Ns)
# Enter the receiver by passing through the matched filter
# Note carrier frequency offset and phase tracking would likely be needed, but here
# both are zero to keep things simple.
y = signal.lfilter(b,1,rbb)
# Symbol synchronize manually
z = ss.downsample(y,Ns,0)
data_ext_hat = dc.qam_gray_decode(z,M)
Nbits,Nerrors = dc.bpsk_bep(data_ext_trim,data_ext_hat)
print('BEP: Nbits = %d, Nerror = %d, Pe_est = %1.3e' % \
      (Nbits, Nerrors, Nerrors/Nbits))
```

```
Eb/N0 = 100.00 dB and Es/N0 = 107.78 dB
kmax =  0, taumax = 72
BEP: Nbits = 479928, Nerror = 0, Pe_est = 0.000e+00
```

A quick look at the sampler output, z, reveals via the plot of Figure 14, that all is well with the

sampling again being 0.

```
plot(z[2000:2100].real*7,'r.')
ylim([-8,8])
title(r'Scaled Optimal Symbol Spaced Samples')
ylabel(r'Inphase Symbol Samples')
xlabel(r'Sample Index')
grid();
```

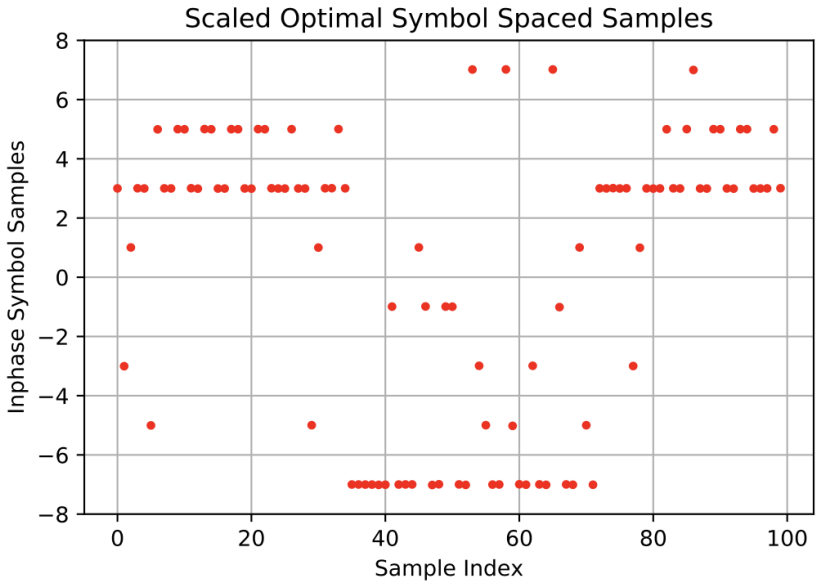

Figure 14: The scaled sampler soft IQ outputs, z, real part to verify that sampling is correct.

As we get ready to decode the recovered/demodulated bit stream data_ext_hat we have to bear in mind the signal imposed by the two SRC pulse shaping filters. Why? Bits that enter the PCM decoder must be properly framed, that is N_PCM is the word length, in this case 8 bits, so the word boundaries have to align for the decoder the do its job. This is where forming PCM frames with SP helps out. The default pulse shaping filter, b, introduces a delay of 6 symbols as does the matched filter in the receiver. Hence a total delay of 12 symbols is introduced. For $M = 64$ QAM we transmit 6 bits per symbol. The total delay in bits is thus $6 \times 12 = 72$. The concern is this delay an integer multiple of NPCM=8? Yes, as $72/8 = 9$. This will not always be the case when NPCM takes on different values, or in a practical we do not know the exact arrival time of the waveform.

Listing 16: Resolving PCM frame alignment using find_sync_idx() and return_pcm_stream(), the decoded PCM bits now return the transmit audio waveform as shown in Figure 15.

```
sync_idx = find_sync_idx(data_ext_hat[:500*8*2],sp_68,cplot=False)
out_pcm_stream = return_pcm_stream(data_ext_hat,sync_idx,sp_68)
m1_hat = dc.pcm_decode(out_pcm_stream,N_PCM)

plot(m1_hat)
ylabel(r'Speech Amplitude')
```

```
xlabel(r'Sample Index at 8 ksps')
title(r'A Portion of OSR_uk_000_0050_8k.wav')
```

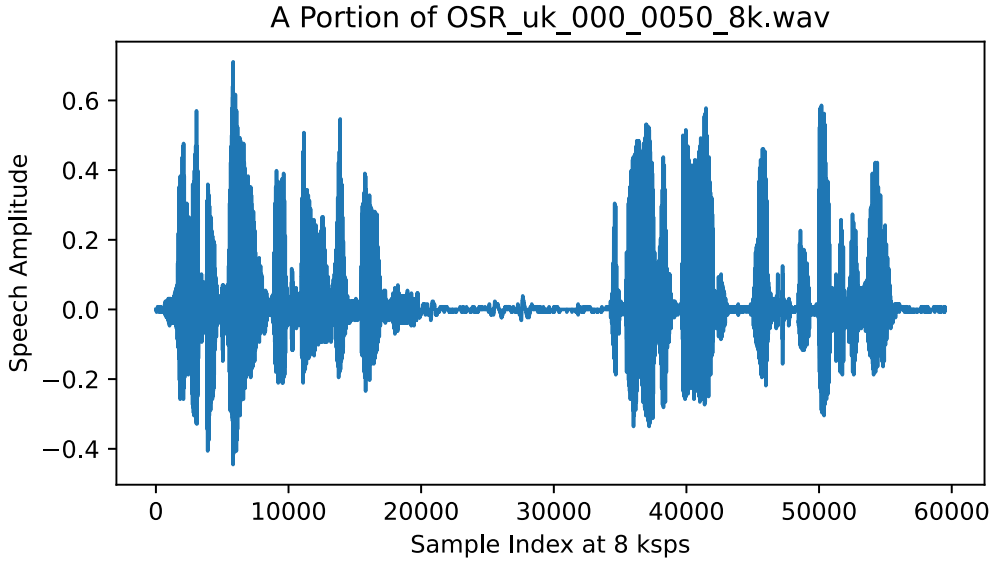

Figure 15: Recovered serial bits PCM decoded back to audio at 8 ksps.

A few more details on the audio loop through are included in the Jupyter notebook version of this example.

Project Tasks

When needed, the speech message source used in all parts is `OSR_uk_000_0050_8k.wav`.

1. **Scatter Plots & BEP Waterfall Curves**: In this task you compare the BEP performance of QAM with $M = 2, 4, 16, 64,$ and 256 versus the received E_b/N_0.

   a) To get started create a 3 row by 2 column subplot array of scatter plots similar to Figure 9 as M is stepped over the indicated values. Use the code of Listing 4 as your template, but for all of the subplots fix `EbNo_dB = 20`. Note the noise variance is not constant as number of bits per symbol changes according to $10\log_2(M)$. This does however keep the cost in transmitter energy required to send one bit constant across all M values. Does it make sense that as M increases the cost in E_b/N_0 (higher value) to send a bit goes up in order to maintain the same BEP? What do you get in return for this increased cost?

   b) For the same M values plot a family of theoretical BEP curves on the same plot. Appendix C Listing 19 and Figure 22 provides details. When $\log_{10}(P_e)$ versus E_b/N_0 in dB is plotted the curve shape resembles a *waterfall*. A sample plot with the required plot limits is given in Listing 17. Be sure to extend the plot legend accordingly.

Listing 17: Sample BEP plot with axis limits appropriate for Task 4b.

```
# Sample BEP Plot
figure(figsize=(6,6))
EbNO_dB_plot = arange(0,25.6,0.5)
```

```
Pe_thy_plotM2 = dc.qam_bep_thy(EbN0_dB_plot,2)
semilogy(EbN0_dB_plot,Pe_thy_plotM2)
ylim([1e-6,1e-1])
xlim([0,25])
ylabel(r'Probability of Bit Error')
xlabel(r'Received $E_b/N_0$ (dB)')
title(r'BPSK (M=2) BEP')
legend((r'M = 2',),loc='best')
grid();
```

c) Using your limited understanding of digital comm try to explain why the $M = 2$ and $M = 4$ BEP curves overlap. **Hint**: Refer the first two subplots of 4a to help in your explanation.

2. **Phase Noise Sensitivity**: Configure an `Ns = 1` (one sample per bit) 64-QAM simulation according to Listing 4.

   a) Make baseline BEP measurements for `EbN0_dB = 15`, `16`, and `17`. Overlay your measured BEP values on the theoretical curve similar to Task4b, except now you need to plot the $M = 64$ theory curve with your measured BEP points. Measure at least 100 bit errors.

   b) Repeat (a) except now set $\sigma_\phi = 1°$ of phase noise using the code of Listing 18, also given in the sample notebook. The experimental BEP points with phase noise should begin to flare out as `EbN0_dB` increases (significantly with $2°$ in part (c)).

   c) Repeat (a) except now set $\sigma_\phi = 2°$, of phase noise using the code of Listing 18, also given in the sample notebook.

Listing 18: Phase noise injection code for an `Ns = 1` simulation.

```
...
EbN0_dB = 15
EsN0_dB = 10*log10(log2(M))+ EbN0_dB
print('Es/N0 = %4.2f (dB)' % EsN0_dB)
rbb = dc.cpx_AWGN(x_IQ_scaled,EsN0_dB,1)
std_pn_deg = 1.0
rbb_pn = rbb*exp(1j*2*pi*std_pn_deg*randn(len(rbb))/360)
...
```

3. **Symbol Timing Sensitivity**: In this task you will introduce timing error by deliberately skewing the sample time from the optimum or maximum eye opening value. To get started configure a waveform simulation with `Ns = 50` that uses SRC pulse shaping. A good starting point is to combine Listing 12 and Listing 13, less the two plots at the end of Listing 12.

   a) With $M = 64$ collect two BEP points of at least 100 errors, such that the error probability values sit above and below 10^{-4} for a sample time one over the optimum. If the optimum is 0 then set `phase = 1` in `ss.downsample()`. With 50 samples per symbol the timing error is skewed by 1/50 symbol or 2%. Find the approximate dB degradation at $P_e = 10^{-4}$ resulting from the timing error. The sketch in Figure 16 shows how to graphically form the measurement. The theory curve should help in setting E_b/N_0 values.

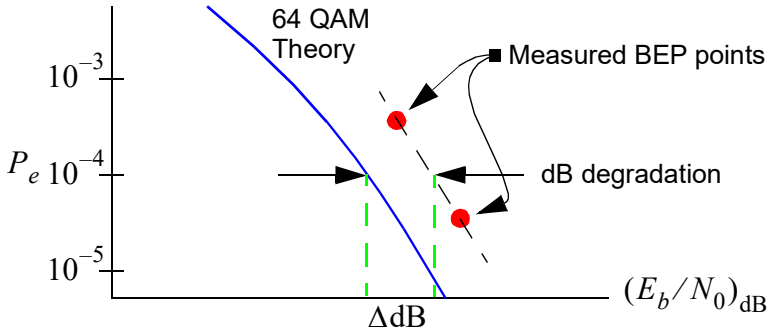

Figure 16: Measuring BEP degradation at 10^{-4}.

b) Repeat (a) with the timing skew increased to two samples or 4% timing error.

4. Rework the PCM Audio Loop Through example of Listing 15 for the case of $M = 256$ and N_PCM doubled to 16 bits per sample. Use the capabilities of the added sync pattern (SP) to insure that PCM word alignment is correct for this 16-bit per symbol QAM scheme. Leave $E_b/N_0 = 100$ dB to insure that the channel is clean and the end-to-end BEP is essentially zero. Include a figure equivalent to Figure 15 for the code of Listing 16, modified accordingly. There is no need to include a wave file, as the code and the figure which matches the known speech pattern will be the proof. Make comments as appropriate.

Bibliography/References

[1] R.E. Ziemer and W.H. Tranter, *Principles of Communications*, 7th edition, Wiley, 2015.

[2] M. Rice, *Digital Communications A Discrete-Time Approach*, Prentice Hall, 2009.

Appendix A: Eye and Scatter Plots in Digital Comm

The eye plot and scatter plot are two very useful characterization tools when working with digital modulation waveforms. The power spectrum is also very useful, but you already have an understanding of that.

Eye Plot

The eye plot operates at the waveform level, typically observing the output of the matched filter, by overlaying integer multiples of the signaling interval. As long as you have synchronously waveform sampled, here that means an integer number of sample per bit period, the eye plot is very easy constructed as shown in Figure 17.

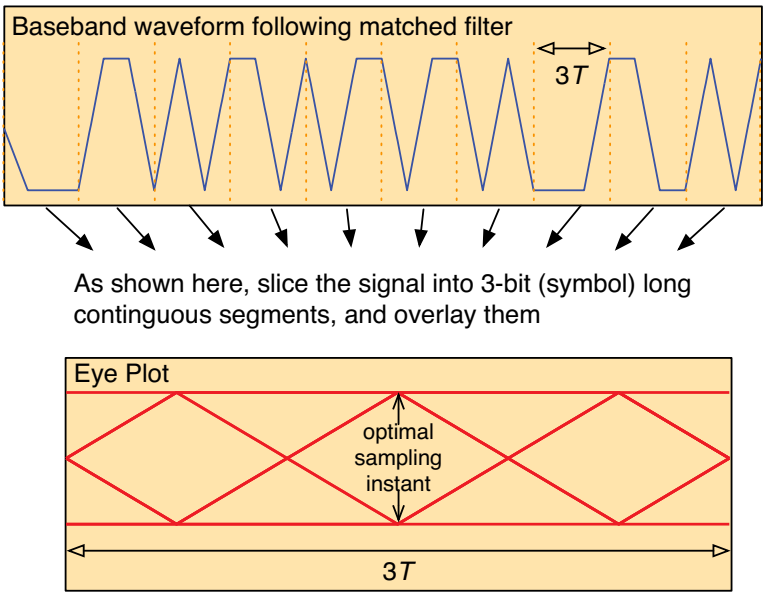

Figure 17: The construction of the eye plot.

The modules ssd.py and digitalcom.py both contain an eye plot function, i.e.,

```
ssd.eyeplot(x,L,S) or dc.eyeplot(x,L,S)
Parameters
==========
x = ndarray of the real input data vector/array
L = display length in samples (usually two symbols)
S = start index
```

The signal array x must be real. You usually don't want the array length to be too long as plotting all of the overlays is time consuming. A simple example for the case of baseband BPSK is in Figure 18, below.

```
x,b,d = dc.NRZ_bits(500,16,'src')
# Matched filter
z = signal.lfilter(b,1,x)
# Delay start of plot by SRC filter length = 2*M*Ns = 12*16
dc.eye_plot(z,2*16,12*16)
```

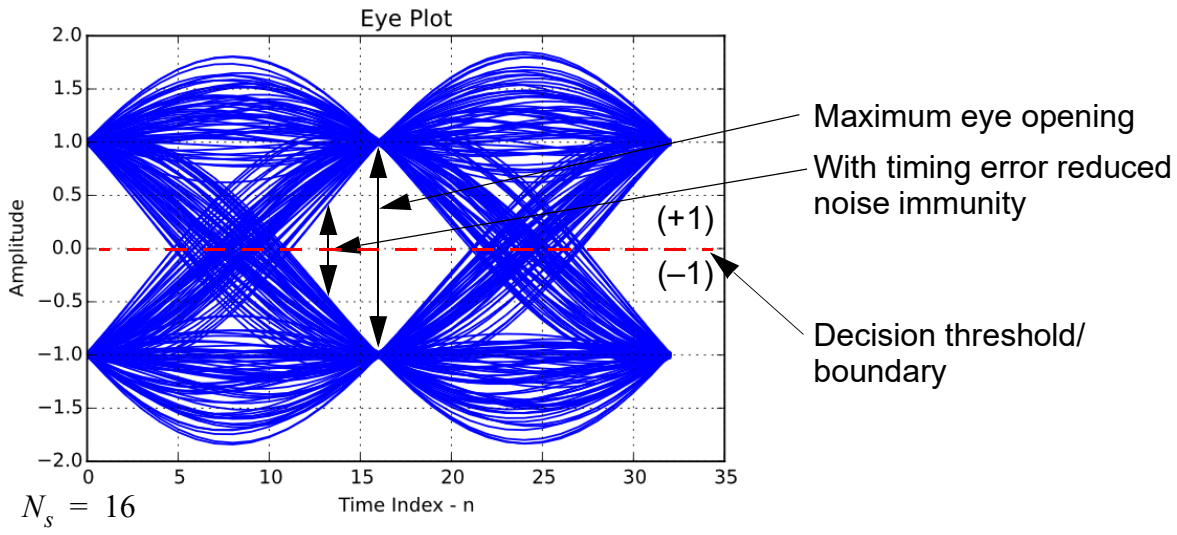

Figure 18: Example eye plot taken at very high E_b/N_0.

The eye plot reveals the optimal sampling instant over the interval $[0, N_s - 1]$, which is where the *eye* is most open. In the plot above you see that 16 is the answer, but this is modulo N_s, so it is really 0.

Scatter Plot

The scatter plot typically observes the complex baseband signal at the matched filter output, following the sampler (every T_s seconds or N_s samples if in the discrete-time domain as we have N_s samples per bit). Under ideal conditions, the scatter plot points lie at the values seen at the maximum eye opening in the eye plot. For the case of BPSK at baseband the nominal value is ± 1. The ideal sample point locations constitute what is known as the signal constellation. If say, a phase error is present, the signal point set will be rotated about the origin by angle θ. If AWGN is present there will be a cloud of points centered at the ideal location. If the channel distorts the signal, then even under noise free conditions there will be a cloud of points, rather than a single point. In general we desire the scatter plot to form a tight array of sample points, with the clusters easily discernible from each other, so that bit or symbol errors can be kept to a minimum. With increasing AWGN the clusters eventually cross the decision boundary (the imaginary axis as shown in Figure 19), resulting in bit errors.

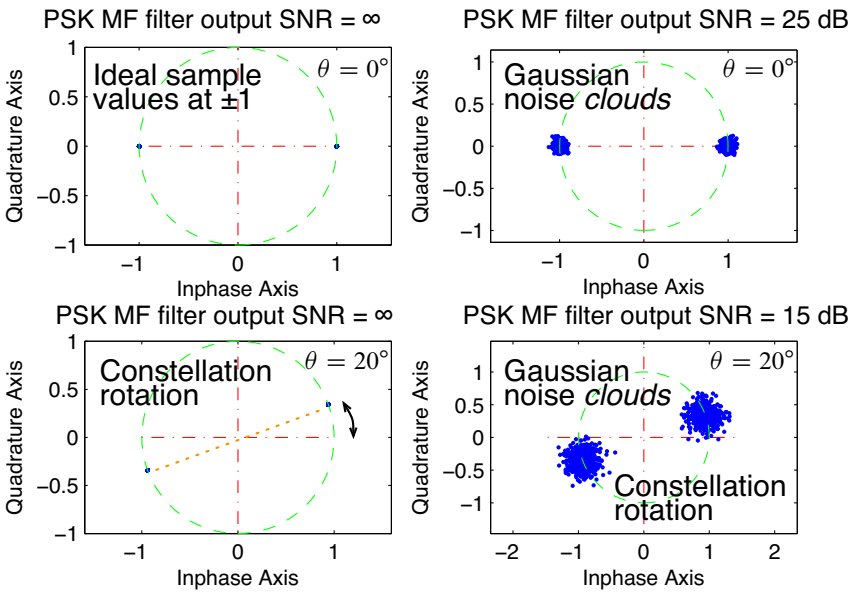

Figure 19: A collection of scatter plots for BPSK under various operating conditions.

Consider now a specific example in Python. Figure 20 shows you can use the function `dc.scat-`

```
x,b,d = dc.NRZ_bits(1000,16,'src')
y = dc.cpx_AWGN(x,20,16)
# Matched filter
z = signal.lfilter(b,1,y)
# Delay start of plot by SRC filter length = 2*M*Ns = 12*16
zI,zQ = dc.scatter(z,16,12*16)
plot(zI,zQ,'.')
z = signal.lfilter(b,1,x)
zI,zQ = dc.scatter(z,16,12*16)
plot(zI,zQ,'r.')
axis('equal')
title(r'BPSK Scatter Plot for $E_b/N_0 = 20$ dB')
xlabel(r'In-Phase')
ylabel(r'Quadrature')
grid();
```

Overlay noise free case

`ter()` or just use plot and break out the real and imaginary parts on your own. You will also have choose the sampling instant and the stride interval, e.g,

```
plot(z[start::Ns].real,z[start::Ns].imag,'.')
```

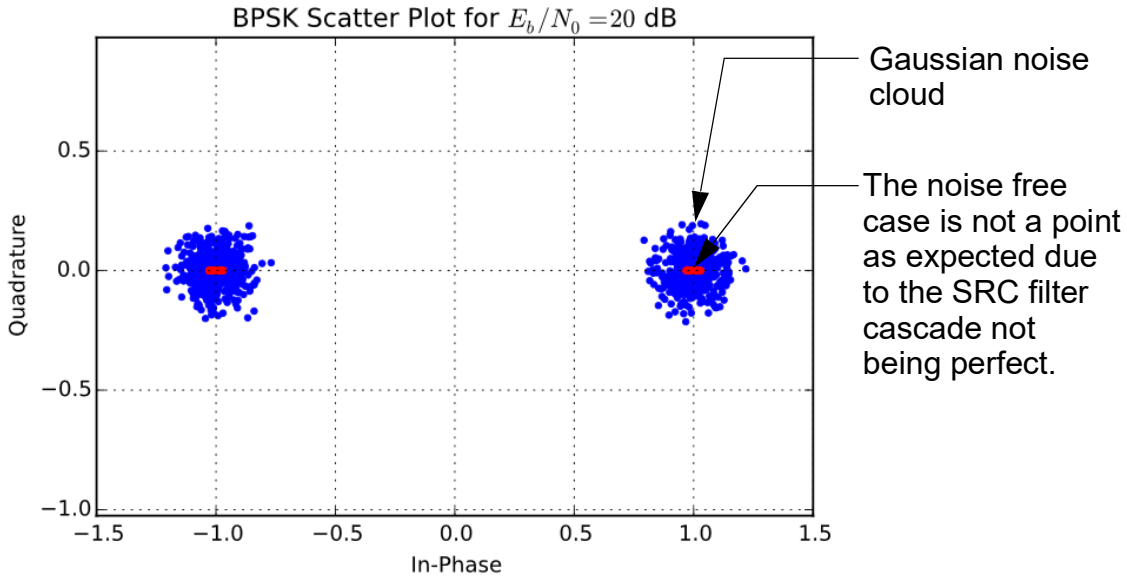

Figure 20: An over plot of two scatter plots, one at 20 dB and one at very high SNR.

Scatter Plot with Timing Error

You know that the eye plot helps you find the optimum sampling instant modulo N_s. Consider now the BPSK scatter plot of where the sampling instant is purposefully skewed by $\Delta T/T_b = 3/16$, which is equivalent to a three sample offset in a simulation where $N_s = 16$.

```
x,b,d = dc.NRZ_bits(1000,16,'src')
y = dc.cpx_AWGN(x,100,16)
# Matched filter
z = signal.lfilter(b,1,y)
# Delay start of plot by SRC filter length = 2*M*Ns = 12*16
# Include also a 3 sample timing offset
zz = ssd.downsample(z,16,3)
plot(zz[12:].real,zz[12:].imag,'.')
xlim([-2,2])
axis('equal')
title(r'BPSK Scatter Plot with Timing Error \
$\Delta t/T_b = 3/16$')
xlabel(r'In-Phase')
ylabel(r'Quadrature')
grid();
```

High SNR

Use downsample by 16 and phase of 3

Start display with 12 bit period delay for filter transient

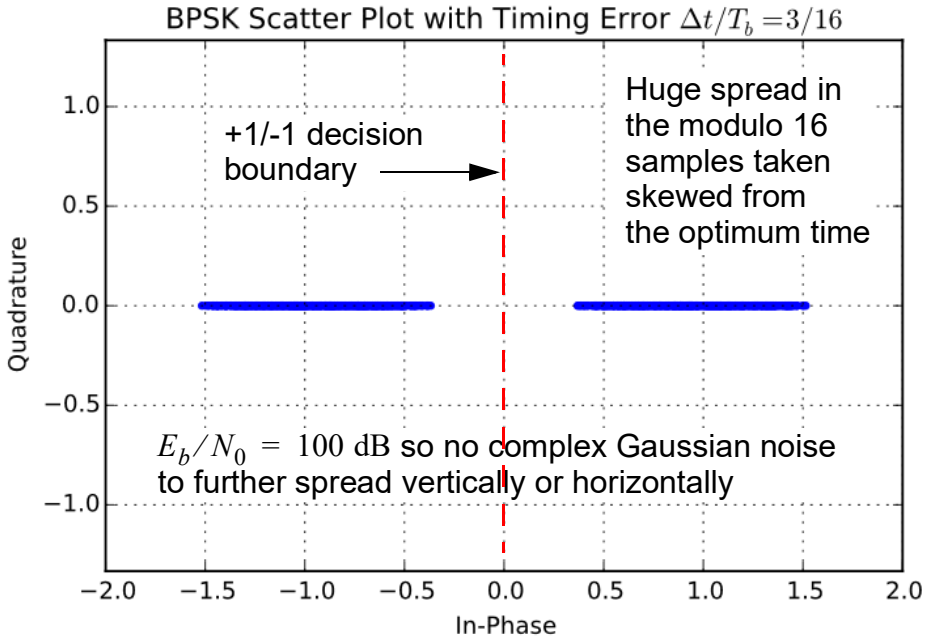

Figure 21: Scatter plot with timing error of $\Delta T/T_b = 3/16$.

Appendix C: Waterfall Curves

In digital communications the ultimate performance of a link is determined by the bit error probability or BEP versus the received E_b/N_0 in dB. Note industry often uses bit error rate or BER in place of the term BEP. BEP expressions have been given for binary antipodal/BPSK and QAM in (4) and (8) respectively. The code for producing a sample theoretical curve with the overlay of a couple of simulation points is given in Listing 19 and the resulting plot is in Figure 22.

Listing 19: Sample BPSK BEP plot with experimental points overlaid.

```
data = randint(0,2,1000000)
EbN0_dB = 6
data_hat = dc.AWGN_chan(data,EbN0_dB)
Nbits,Nerrors = dc.BPSK_BEP(data,data_hat)
print('BEP: Nbits = %d, Nerror = %d, Pe_est = %1.3e' % \
      (Nbits, Nerrors, Nerrors/Nbits))
EbN0_dB = 8
data_hat = dc.AWGN_chan(data,EbN0_dB)
Nbits,Nerrors = dc.BPSK_BEP(data,data_hat)
print('BEP: Nbits = %d, Nerror = %d, Pe_est = %1.3e' % \
      (Nbits, Nerrors, Nerrors/Nbits))
```

```
kmax =  0, taumax = 0
BEP: Nbits = 1000000, Nerror = 2354, Pe_est = 2.354e-03
kmax =  0, taumax = 0
BEP: Nbits = 1000000, Nerror = 188, Pe_est = 1.880e-04
```

```
EbN0_dB_r = arange(0,14,.1)
M = 2
P_eb_thy = dc.QAM_BEP_thy(EbN0_dB_r,M)
semilogy(EbN0_dB_r,P_eb_thy)
```

```
ylim([1e-6,1e-1])
xlim([0, 14])
ylabel(r'Bit Error Probability')
xlabel(r'$E_b/N_0$ (dB)')
semilogy([6,8],[2.354e-3,1.880e-4],'r.')
legend((r'$M=2$',r'Measured'))
grid();
```

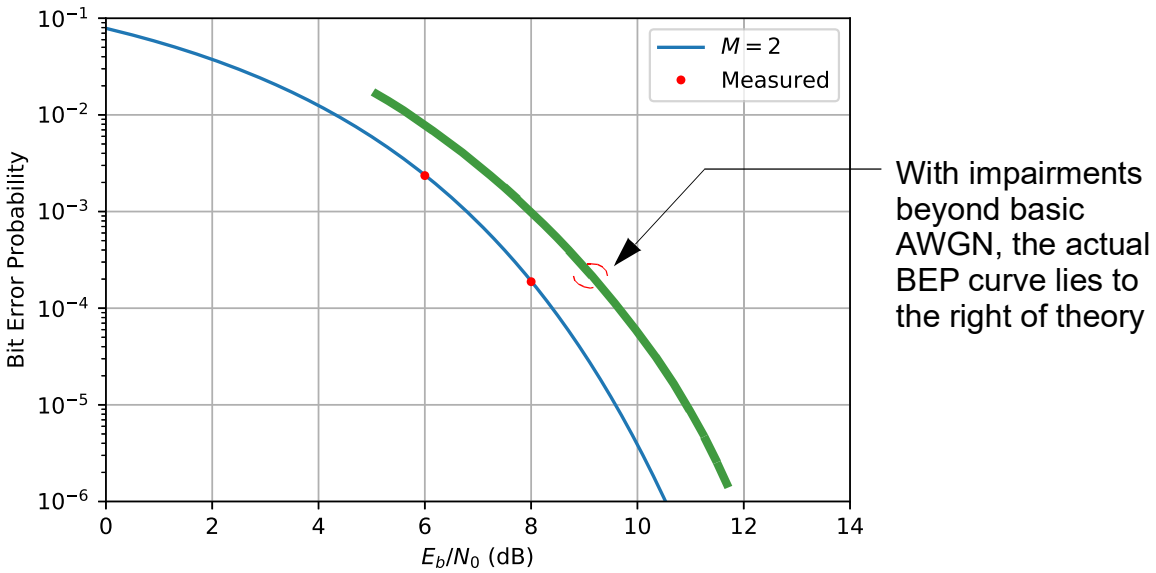

Figure 22: Theoretical BPSK BEP curve including one measured point.

In characterizing various impairments in digital comm, such as phase error, timing error, or adjacent channel interference, you typically find the actual or measured BEP curve shifted to the right by a few tenths or more of a dB. See the green curve in Figure 22. Note for experimental points statistical accuracy requires at least 100 error events be observed. Here 2354 and 188 errors were counted respectively.

Project 2 Spring 2026 Point Assignments

Task	Part	Points
1a	Five scatter plots in 2x3 array: $M = 2, 4, 16, 64, 256$	10
1b	Family of waterfall BEP curves vs $(E_b/N_0)_{\text{dB}}$ for $M = 2, 4, 16, 64, 256$	10
1c	Explain $M = 2$ and 4 BEP plot overlap	10
2a	64-QAM phase noise sensitivity starting with $\sigma_\phi = 0°$	10
2b	64-QAM phase noise sensitivity starting with $\sigma_\phi = 1°$	10
2c	64-QAM phase noise sensitivity starting with $\sigma_\phi = 2°$	10
3a	2% timing error $(E_b/N_0)_{\text{dB}}$ degradation at $P_e = 10^{-4}$	10
3b	4% timing error $(E_b/N_0)_{\text{dB}}$ degradation at $P_e = 10^{-4}$	10
4	256-QAM, `N_PCM = 16` PCM audio loop through	10
Total		**90**