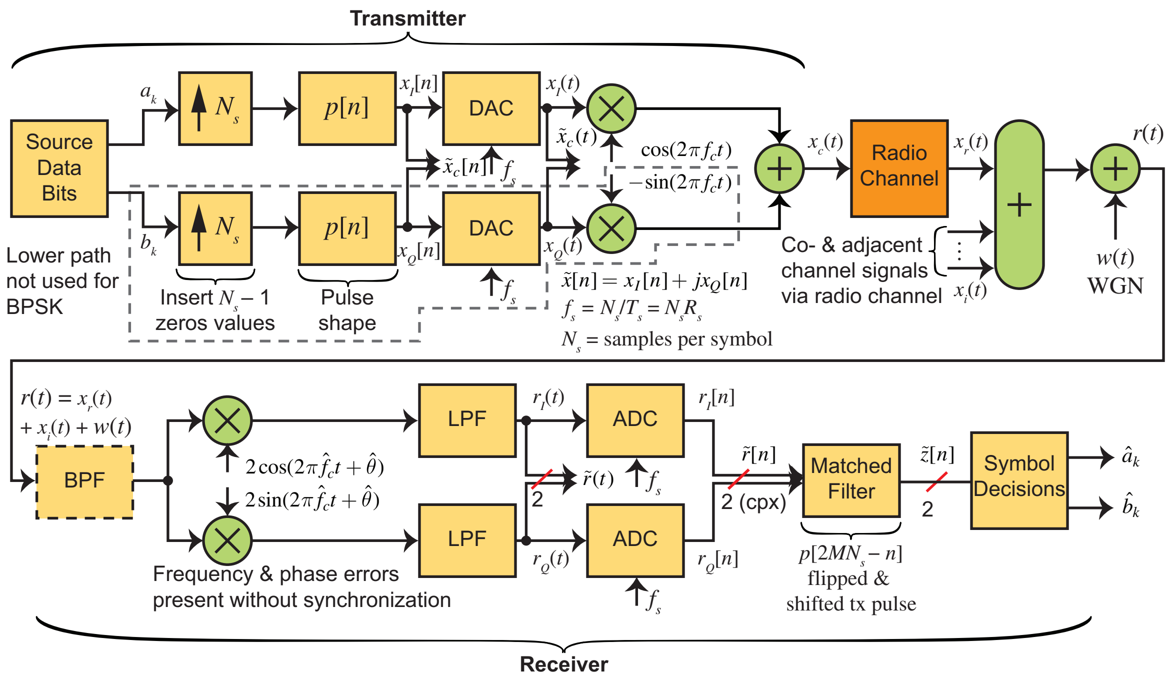
ECE 5625/4625 Python Project 2

Introduction

In this team project you will studying PCM encoding and decoding of speech signals over an additive white Gaussian noise channel and then binary phase-shift keying (BPSK) in an adjacent channel interference environment. Recorded speech waveforms will be used to provide test messages to be encoded into a serial bit stream of ones and zeros. Random bit streams will also be employed. The entire simulation will be constructed in an Jupyter notebook. A template notebook which includes a code framework is provided in additional to the document you are reading now. will be provided along with interface requirements, so that the code submitted with the report project can be easily tested using the speech waveform message signals. The Jupyter notebook and any support `*.py` modules will be available in a ZIP package on the course Web Site.

Honor Code: The project teams will be limited to at most *three* members. Teams are to work independent of one another. Bring questions about the project to me. 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 10, at 5:00 pm, 2017. Note the final exam is Thursday, early afternoon, May 11, 2017 from 12:40–2:40.

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



Transmitter
Source Data Bits
a_k
b_k
N_s
$p[n]$
$x_I[n]$
$x_Q[n]$
DAC
$x_I(t)$
$x_Q(t)$
$\tilde{x}_c(t)$
$\tilde{x}_c[n]$
f_s
$\cos(2\pi f_c t)$
$-\sin(2\pi f_c t)$
$x_c(t)$
Radio Channel
$x_r(t)$
$x_i(t)$
$r(t)$
$w(t)$
WGN
Lower path not used for BPSK
Insert $N_s - 1$ zeros values
Pulse shape
$\tilde{x}[n] = x_I[n] + jx_Q[n]$
$f_s = N_s/T_s = N_s R_s$
N_s = samples per symbol
Co- & adjacent channel signals via radio channel
$r(t) = x_r(t) + x_i(t) + w(t)$
BPF
$2\cos(2\pi \hat{f}_c t + \hat{\theta})$
$2\sin(2\pi \hat{f}_c t + \hat{\theta})$
Frequency & phase errors present without synchronization
LPF
$r_I(t)$
$r_Q(t)$
$\tilde{r}(t)$
2
ADC
$r_I[n]$
$r_Q[n]$
$\tilde{r}[n]$
2 (cpx)
Matched Filter
$p[2MN_s - n]$ flipped & shifted tx pulse
$\tilde{z}[n]$
2
Symbol Decisions
$\hat{a}_k$
$\hat{b}_k$
Receiver

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. At the receiver demodulation of the a_k and b_k bit streams occurs. For the purposes of this project the focus is on the a_k stream, which is the upper path in the transmitter block diagram. With BPSK only one data stream is employed and $a_k = \pm 1$. In general the a_k and b_k bit streams may each take on M amplitude levels and hence be termed *symbols* that carry more than one bit of information. In the receiver both paths are needed as frequency and phase errors in the demodulation process require both signals to allow a PLL to form an error signal that estimates the phase and frequency error and then drives the error to zero. The PLL carrier phase tracking is not shown in Figure 1.

Before getting into the digital modulation and demodulation details, we take a detour to consider a possible interface that produces a bit stream from an analog message source. Here the message source is speech and encoding processing is pulse code modulation (PCM), which is essentially analog to digital conversion (ADC). At the receive end the bit stream is converted back to an analog message using a PCM decoder, which is very much like a digital-to-analog converter (DAC).

Part I: PCM Encode/Decode

In this first part of the project the focus is on PCM encoding/decoding and the fidelity of the decoded message relative to the encoded message. The influence of bit errors is also considered.

White Gaussian Noise Channel

To get started you will consider a PCM encode/decoder pair with an additive white Gaussian noise channel model sitting in between. The block diagram for this system is shown in Figure 2.

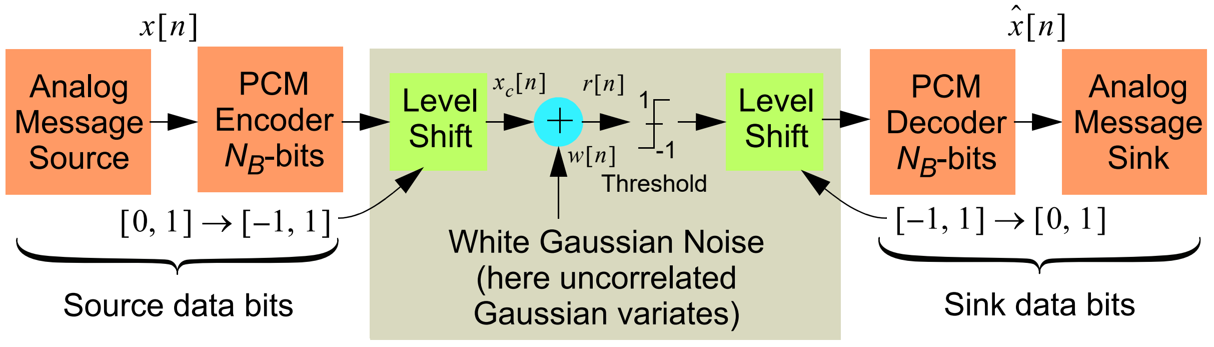


Figure 2: Discrete Gaussian binary channel model to represent sending bits in a noisy communications environment.

In a practical wireless scheme, the serial bits from the PCM encoder would also need to be packetized and then modulated onto a carrier as a waveform. To keep things simple in this project, packet formation has been omitted. The case of a full waveform channel is considered later in this project overview, but for now is simplified or abstracted to just sending ± 1 bit values or one sample per bit. In any case the 0/1 bits are mapped to a continuous-time waveform/sequence with each bit having a duration of T_b seconds. The bit period is inversely proportional to the channel bit rate R_b. 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 bit stream (in discrete-time form), the AWGN channel produces at the receiver

$$r[n] = x_c[n] + w[n] \tag{1}$$

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 $w[n_1]$, $n_0 \neq n_1$, 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.

Here the mapping is 0/1 to ± 1 values, which is known as *binary antipodal* signaling, which it turns out is equivalent to baseband BPSK. In Python code details of the channel can be found in the function.

```python
def AWGN_chan(x_bits,EBN0_dB):
    """
    ////////////////////////////////////////////////////////////////////////
     x_bits = serial bit stream of 0/1 values.
    EBNO_dB = energy per bit to noise power density ratio in dB of the
              serial bit stream sent through the AWGN channel. Frequently
              we equate EBN0 to SNR in link budget calculations
     y_bits = received serial bit stream following hard decisions. This bit
              will have bit errors. To check the estimated bit error
              probability use digitalcom.BPSK_bep() or simply
              >> Pe_est = sum(xor(x_bits,y_bits))/length(x_bits);
    ////////////////////////////////////////////////////////////////////////

    Mark Wickert, March 2015
    """
    x_bits = 2*x_bits - 1 # convert from 0/1 to -1/1 signal values
    var_noise = 10**(-EBN0_dB/10)/2;
    y_bits = x_bits + np.sqrt(var_noise)*np.random.randn(size(x_bits))

    # Make hard decisions
    y_bits = np.sign(y_bits) # -1/+1 signal values
    y_bits = (y_bits+1)/2 # convert back to 0/1 binary values
    return y_bits
```

At the receive end of the channel the noisy values $r[n]$ need to be converted back to hard decision values of ± 1. These values are then converted back to 0/1 values via level shifting. The fact that noise is present means that some bit values sent as +1 will be < 0 at the receive end, and some bits sent as -1 will be > 0 at the receive end. 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 probability of a bit error is

$$P_{e,\,\text{thy}} = \frac{1}{2}\text{erfc}\left(\sqrt{\frac{E_b}{N_0}}\right) = \frac{1}{2}\text{erfc}\left(\sqrt{10^{\text{SNR}_{\text{dB}}/10}}\right) \tag{2}$$

where

$$\text{erfc}(u) = \frac{2}{\sqrt{\pi}}\int_u^{\infty} e^{-v^2}\,dv \tag{3}$$

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.,

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

In Python you use scipy.special to get access to erfc():

```
import scipy.special as special
# 1/2*special.erfc(sqrt(10**(EbN0_dB/10)))
EbN0_dB = 5
Pe_thy = 1/2*special.erfc(sqrt(10**(EbN0_dB/10)))
print('Pe_thy = %1.3e' % Pe_thy)
```

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

The last stage of processing in Figure 2 is PCM decoding of the serial source bit stream. The decoding operation groups N_B bits together to form a binary word that is then converted into a signed integer. The decoded signal sample value is denoted $\hat{x}[n]$ to signify the fact that sending the original message $x[n]$ over the AWGN channel has resulted in quantization errors due to the PCM encoding and individual bit errors due to the AWGN channel.

A PCM encoder (think ADC) has a full-scale range that the conversion takes place over. For the Python functions developed here the full scale input and hence output range is ± 1. We assume a bipolar conversion range because the message signal is assumed to have a zero mean. In order to have a quantization level at zero it is customary as in fixed-point numbers, to have one less quantization level on the positive side, thus the quantization levels cover the interval $[-1, 1 - 2^{-(N_B - 1)}]$. The encoding and decoding function listing, as found in the corresponding Jupyter notebook, are given below:

```
def PCM_encode(x,N_bits):
    """
    x_bits = PCM_encode(x,N_bits)
    /////////////////////////////////////////////////////////////
         x = signal samples to be PCM encoded
    N_bits = bit precision of PCM samples
    x_bits = encoded serial bit stream of 0/1 values. MSB first.
    /////////////////////////////////////////////////////////////
    Mark Wickert, Mark 2015
    """
    x = np.int16(np.rint(x*2**(N_bits-1)))
    x_bits = np.zeros((N_bits,len(x)))
    for k, xk in enumerate(x):
        x_bits[:,k] = tobin(xk,N_bits)
    # Reshape into a serial bit stream
    x_bits = np.reshape(x_bits,(1,len(x)*N_bits))
    return int16(x_bits.flatten())

# A helper function for PCM_encode
def tobin(data, width):
    data_str = bin(data & (2**width-1))[2:].zfill(width)
    return map( int, tuple( data_str ) )
```

```
def PCM_decode(x_bits,N_bits):
    """
    xhat = PCM_decode(x_bits,N_bits)
    /////////////////////////////////////////////////////////////
    x_bits = serial bit stream of 0/1 values. The length of
             x_bits must be a multiple of N_bits
    N_bits = bit precision of PCM samples
      xhat = decoded PCM signal samples
    /////////////////////////////////////////////////////////////
    Mark Wickert, March 2015
    """
    N_samples = len(x_bits)//N_bits
    # Convert serial bit stream into parallel words with each
    # column holdingthe N_bits binary sample value
    x_bits = np.reshape(x_bits,(N_bits,N_samples))
    # Convert N_bits binary words into signed integer values
    xq = np.zeros(N_samples)
    w = 2**np.arange(N_bits-1,-1,-1) # binary weights for bin
                                     # to dec conversion
    for k in range(N_samples):
       xq[k] = np.dot(x_bits[:,k],w) - x_bits[0,k]*2**N_bits
    return xq/2**(N_bits-1)
```

Mean-Squared Error

A performance measure that characterizes the distortion introduced by the quantization error and the channel bit errors, is the mean-squared error (MSE) between $x[n]$ and $\hat{x}[n]$:

$$\text{MSE} = \frac{1}{N}\sum_{n=1}^{N}(\hat{x}[n]-x[n])^2, \tag{5}$$

where N is the length of the message array in Python. For high channel SNR we expect that the MSE is due just to the PCM quantization error, which is inversely proportional to the word length, N_B. In the project tasks you will be investigating 1/MSE in dB as a measure of the decoded PCM message signal quality.

Part II: Pulse-Shaped BPSK Waveform Modeling

In this second part of the project you study in more detail how at the waveform level, the binary PCM stream can be placed on a carrier and then recovered. The digital modulation considered here is BPSK. A baseband BPSK signal is easily formed by *impulse train modulating* the translated ± 1 PCM data bits. The impulse train modulator stuffs $N_s - 1$ zero samples in between each ± 1 data bit. 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 3. Baseband BPSK exits the pulse shape filter and

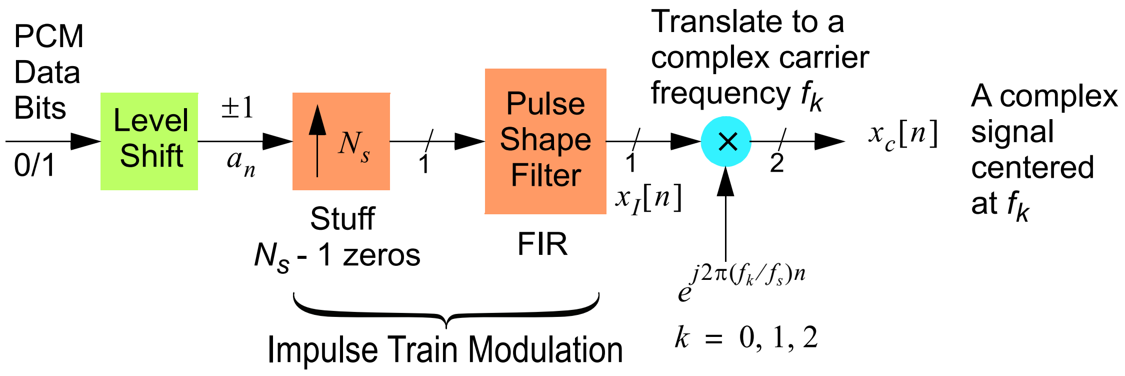


Figure 3: Impulse train modulator and complex frequency translator for BPSK.

is frequency translated to a carrier frequency f_k relative to a sampling rate of f_s Hz. Note also that the bit period is $T_b = N_s/f_s$ s so the bit rate is $R_b = 1/T_b = f_s/N_s$ bps. 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_b to represent each ± 1 bit. The transmitted signal spectrum will have the form $T_b \text{sinc}^2(fT_b)$, 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 BPSK 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) are each used at the transmitter and receiver. 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 4 below shows this Tx/Rx configuration [1]. In Python all of this is pre-build

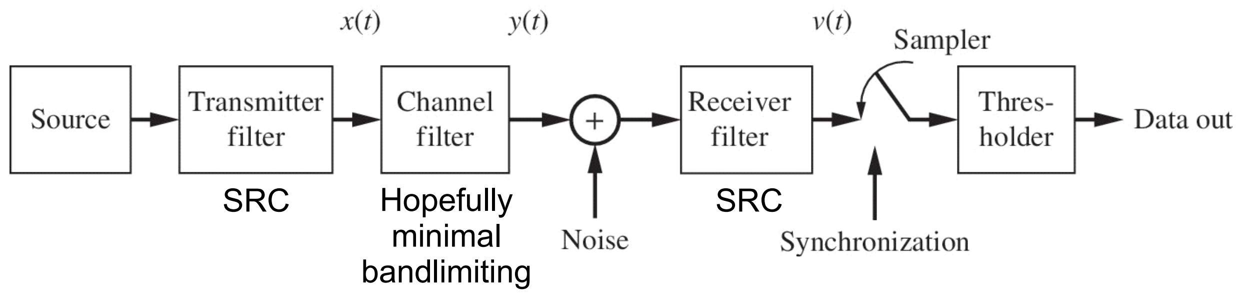


Figure 4: A simplified transceiver block diagram showing how pulse shaping is spread between the transmitter and receiver [1].

using functions available in `ssd.py` and `digitalcom.py`. When synchronization is needed functions are available in `synchronization.py`.

As an example the code below and plot of Figure 5 generates baseband BPSK with SRC pulse shaping and then translates the signal to $f_c = 4R_b$ for the case of $N_s = 16$ samples per bit.

```
#dc.NRZ_bits(N_bits, Ns, pulse='rect', alpha=0.25, M=6)
xbb1,b,d = dc.NRZ_bits(10000,16,'src')
n = arange(0,len(xbb0))
# Translate baseband to fc1/fs = 4.0/16
# Relative to Ns = 16 samps/bit & Rb = 1 bps
xc1 = xbb0*exp(1j*2*pi*4.0/16*n)
```

```
figure(figsize=(6,2.5))
psd(xc1,2**10,16);
ylim([-60,5])
xlabel(r'Normalized Frequency $f/R_b$')
title(r'Spectrum SRC Pulse Shaped BPSK at $f_{c1} \
      = 4R_b$ and $N_s = 16$');
```

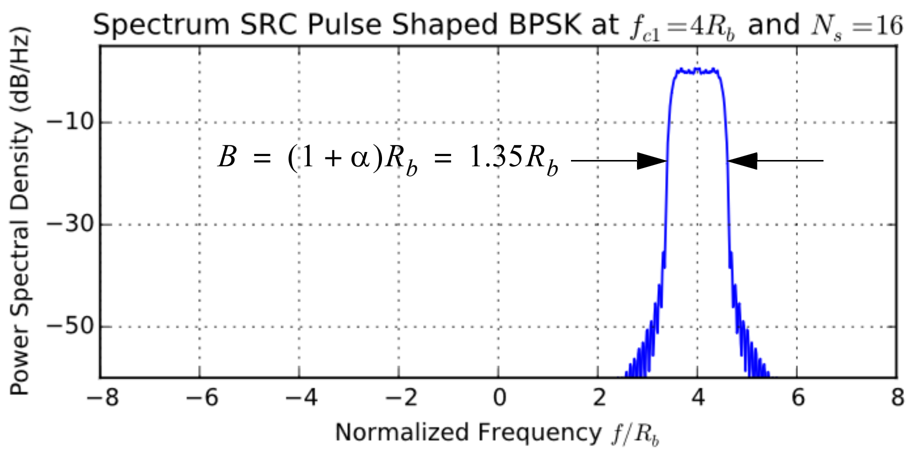


Figure 5: Sample SRC shaped BPSK spectrum shifted to $f_{c1} = 4R_b$.

As Figure 1 shows, the receiver undoes the action of the transmitter, but also must 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 bit timing is perfect, this is the Tx and Rx clocks are synchronous. A simplified receiver block diagram is shown in Figure 6.

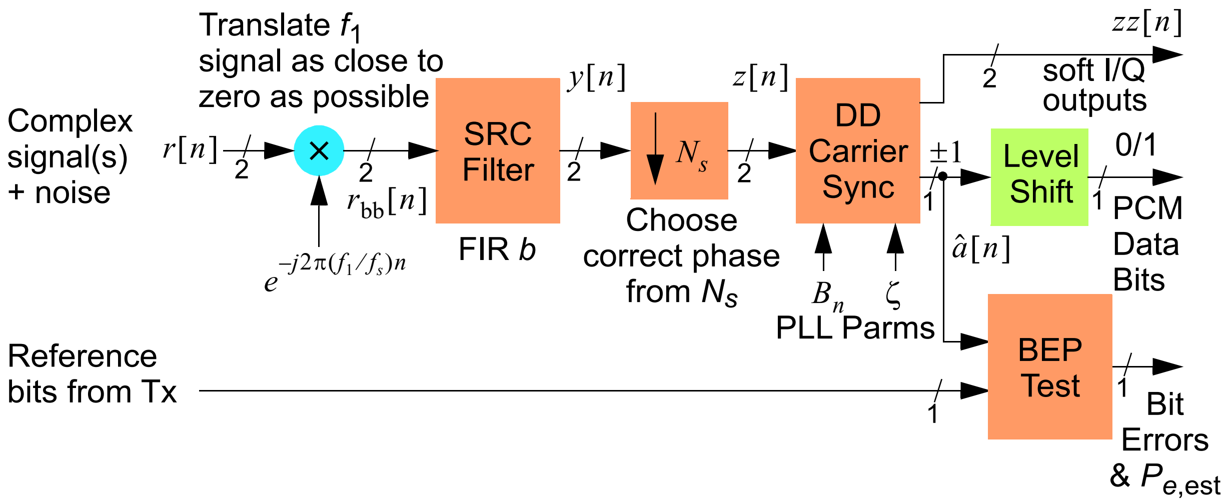


Figure 6: BPSK receiver/demodulator assuming perfect bit timing and carrier phase tracking.

With the software building blocks configured as shown in Figure 6 there are many options for digital communications studies. By level shifting the data bits contained in $\hat{a}[n]$ (the real part of `a_hat`) you can drive the PCM decoder to complete a speech data link using BPSK. Probability of bit error (BEP) studies can be conducted by sending the data bits into the BEP testing function

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

where `tx_data` and `rx_data` are the ± 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 BPSK_BEP() automates this by cross-correlating the input arrays to find the correct code alignment. The optional argument `Ncorr` sets search interval and the fourth argument can be used to skip over initial bits in the receive array where transients may exist. An example of a transient you may want to skip over is the PLL acquisition of the receiver carrier phase.

As a complete link example consider the following Python code:

```
In [167]: Nstart = 22000; # start here
          Nsamp = 20000 # number of sample to process
          fs,m0 = ssd.from_wav('OSR_uk_000_0050_8k.wav')
          m0 = m0[Nstart:Nstart+Nsamp]

# Set the Eb/N0 value in dB for the channel
EbN0_dB = 5
Ns = 16
```

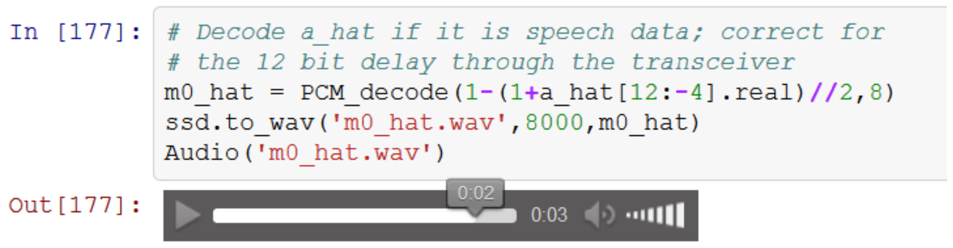
```python
# Choose the random bits or PCM encoded speech
# Uncomment two lines below to select random bits
# Enter the number of bits simulated, here 10000
#x,b,a_in = dc.NRZ_bits(10000,Ns,'src')
#
# Uncomment the following two lines to select encoded speech
a_in = PCM_encode(m0,8)
x,b = dc.NRZ_bits2(a_in,Ns,'src')

# AWGN channel set via EbNo_dB at top
y = dc.cpx_AWGN(x,EbN0_dB,Ns)

# Introduce frequency error into the baseband BPSK signal
# setting Df = 0.012 makes actual Df = 0.012*Rb
n = arange(0,len(x))
Df = 0.012;
y *= exp(1j*2*pi*Df/Ns*n) # Df = 0.012*Rb or 1.2% of Rb

# Incorporate adjacent channel signals by
# uncommenting the next five lines
#f_adj = 1.4
#x_lo,b,d_lo = dc.NRZ_bits(len(a_in),Ns,'src')
#x_hi,b,d_hi = dc.NRZ_bits(len(a_in),Ns,'src')
#y = y + x_lo*exp(-1j*2*pi*f_adj/Ns*n) \
#       + x_hi*exp(1j*2*pi*f_adj/Ns*n)

# Matched filter
z = signal.lfilter(b,1,y)
# Sample MF output at once per bit
# The third argument controls the sampling phase or offset mod Ns
zd = ssd.downsample(z,Ns,0)

# Carrier phase tracking
# DD carrier phase sync: 2<=>BPSK, 0.05=loop BW/Rb,0.70=loop damping
zz,a_hat,e_phi,theta_hat = pll.DD_carrier_sync(zd,2,0.05,0.707)

# Estimate the bit error probability (BEP)
Nbits,Nerrors = dc.BPSK_BEP(2.*a_in-1,a_hat.real)
print('BEP: Nbits = %d, Nerror = %d, Pe_est = %1.3e' % \
      (Nbits, Nerrors, Nerrors/Nbits))

# Calculate the corresponding theoretical BEP, but
# note this is ideal as the carrier phase tracking is
# not modeled.
Pe_thy = 1/2*special.erfc(sqrt(10**(EbN0_dB/10)))
```

```
print('Pe_thy = %1.3e' % Pe_thy)
```

Cell Output:

```
          kmax =  0, taumax = 12
          BEP: Nbits = 159988, Nerror = 940, Pe_est = 5.875e-03
          Pe_thy = 5.954e-03
```

For PCM audio playback the following is added to another cell:



```
In [177]: # Decode a_hat if it is speech data; correct for
          # the 12 bit delay through the transceiver
          m0_hat = PCM_decode(1-(1+a_hat[12:-4].real)//2,8)
          ssd.to_wav('m0_hat.wav',8000,m0_hat)
          Audio('m0_hat.wav')
Out[177]:
```

Waveforms within the simulation provide useful information about the transmitted and received BPSK signal in the time and frequency domains. Appendix A describes eye plots and scatter plots, tools you will practice with in the Project Tasks. Appendix B describes the details of the decision directed (DD) carrier phase tracking algorithm employed in the simulation.

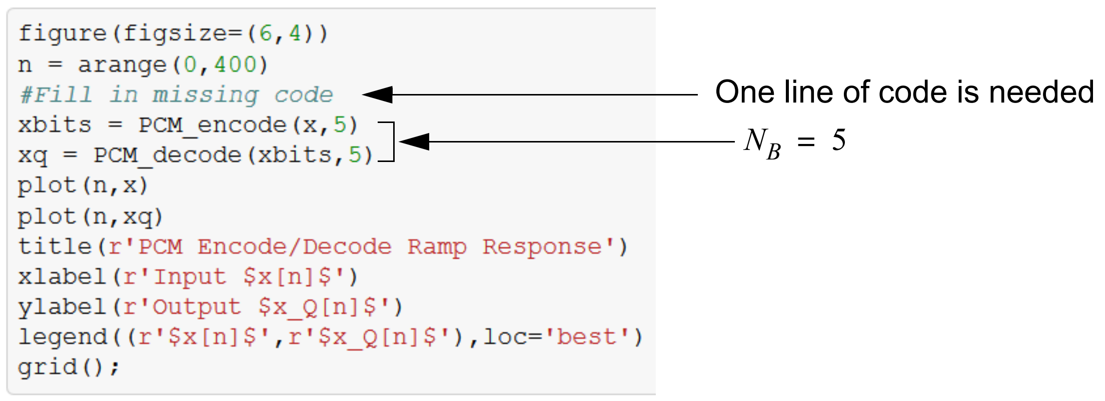
Project Tasks

The speech message source used in all parts is `OSR_uk_000_0050_8k.wav`.

Part I

1. When working with `PCM_encode()` you need to understand that the dynamic range is limited just like with an ADC.

   a) To be clear on the impact of this behavior design an input signal, $x[n]$, that is a ramp that sweeps linearly from -2 to +2 over 400 steps. In Python this can be constructed and plotted as follows:



```
figure(figsize=(6,4))
n = arange(0,400)
#Fill in missing code
xbits = PCM_encode(x,5)
xq = PCM_decode(xbits,5)
plot(n,x)
plot(n,xq)
title(r'PCM Encode/Decode Ramp Response')
xlabel(r'Input $x[n]$')
ylabel(r'Output $x_Q[n]$')
legend((r'$x[n]$',r'$x_Q[n]$'),loc='best')
grid();
```

   In your notebook fill in the missing code, produce the plot, and then explain the plot. What is the largest quantization level when $N_B = 5$ bits?

   b) Modify `PCM_encode()` so that it saturates or clips the input signal and avoids the problems you see in Task 1a. **Hint**: The numpy function `clip()` makes this easy to do, e.g.,

   `x_clip = x.clip(min_val,max_val)`

   Modify the function interface by including an optional argument `sat_mode` taking the default value True, e.g.,

   `def PCM_encode(x,N_bits, sat_mode = True):`

   To verify that your code enhancement works, re-plot the Part (a) results.

2. To understand the limitations of N_B-bit PCM encoding, without the impact of channel bit errors, measure and then plot $10\log_{10}\{1/\mathrm{MSE}\}$ versus N_B for the speech input message, as N_B varied from 4 to 16 bits, stepping one bit at a time. Make sure to set the channel SNR to greater than 50 dB (so the bit errors occur probability is is very small) and gain level the speech input to just fit in the dynamic range of the PCM encoder. **Hint**: You will likely want to make use of the numpy `max()` and/or `min()` functions to find the needed scaling constant g to form $m_{0,\,\text{scaled}} = g \cdot m_0$.

   a) Comment on your results. Note that 16 bits is the resolution of the original audio CD recording format. Note also that 1/MSE is proportional to the signal-to-quantization noise ratio (SNR_Q).

b) Comment also on the speech quality you hear through sound playback, as the N_B value is varied. What is the lowest bits/sample value that you feel provides just acceptable intelligibility?

3. In this task you will plot $10\log_{10}(1/\text{MSE})$ versus channel SNR in dB. The plots of (a), (b), and (c) described below should be placed on a single graph for comparison purposes.

   a) Set $N_B = 8$ and vary the channel SNR in dB over the range of 0 dB to 20 dB. In particular plot $10\log_{10}(1/\text{MSE})$ versus channel SNR in dB for the speech message source. Again gain level the speech signal as you did in Task 2.

   b) Repeat part (a) with $N_B = 6$.

   c) Repeat part (a) with $N_B = 10$.

   d) Comment on your results. You may want to take a look Text Chapter 8, Section 8.5 entitled *Noise in Pulse-Code Modulation*. Your plots should look similar to Figure 8.17. Note: A detailed understanding of the theory is not part of this problem, as you are producing experimental results only.

Part II

4. Configure the waveform level BPSK simulation according to the example given at the end of the Part II section. This code is included in the Jupyter notebook, so there is really not too much work in setting it up.

   a) By commenting and uncommenting code blocks at the top of the simulation, configure the simulation for random data bits as opposed to PCM speech bits. Set the number of bits in `NRZ_bits()` (`N_bits`) to just 1000. Set `EbN0_dB` to 25 dB and frequency error `Df` to 0.012 and obtain a scatter plot of the `DD_carrier_sync()` block `zz` output waveform. Enter a 12 sample delay to compensate for the filter delays. You should observe a transient. Explain what is happening. For hints look at Appendix B. You may want to take a look at the tracking loop error waveform `e_phi` in Figure 13.

```
plot(zz[12:].real,zz[12:].imag,'.')
axis('equal');
xlim([-1.25,1.25]);
grid();
```

   b) Repeat part (a) except now increase the PLL loop bandwidth from `BW/Rb` = 0.05 to 0.1. Observe the changes in the scatter plot transient. Also comment on the size of the Gaussian cloud relative to the vertical decision boundary as described in the last figure of Appendix A. How likely are bit errors given the size of the cloud?

5. Modify the simulation to now include two adjacent channel signals, at the same power level as the desired baseband signal that you will keep near $f = 0$. Initially place the adjacent channel signals on carrier frequencies at $f_{\text{adj}} = \pm 1.4 R_b$, i.e.,

```
# Incorporate adjacent channel signals (just before the matched filter)
f_adj = 1.4
```

```
x_lo,b,d_lo = dc.NRZ_bits(len(a_in),Ns,'src')
x_hi,b,d_hi = dc.NRZ_bits(len(a_in),Ns,'src')
y = y + x_lo*exp(-1j*2*pi*f_adj/Ns*n) \
      + x_hi*exp(1j*2*pi*f_adj/Ns*n)
```

a) Plot the power spectrum of the received signal `y` using 10000 bits and `EbN0_dB` = 25 dB. Scale the frequency axis in terms of the normalized bit rate, i.e., $f/R_b \Rightarrow f_s = N_s$:

```
psd(y,2**10,Ns);
xlabel(r'Normalized Frequency $f/R_b$')
```

Verify that you see three distinct complex baseband BPSK carrier signals.

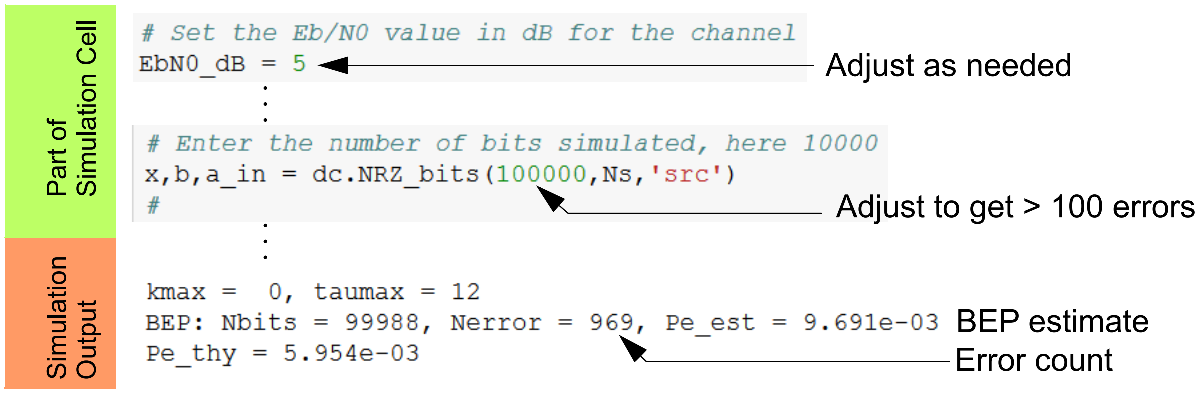
b) In this part you will construct an eye plot of the matched filter output `z` following the adjacent channel interference (ACI) injection point. Set `EbN0_dB` to 100 to just focus on the influence of the adjacent signal interference as viewed in the eye plot. You will need to set the frequency error, `Df` = 0. The eye should be fully open (± 1). Now decrease the channel spacing of the adjacent signals until the eye closes to about ± 0.5. Make note of this frequency. Here `z` is complex so you will be plotting just `z.real eye_plot()`, and return to using just 1000 bits in the simulation.

c) Plot the combined power spectrum at point `y` under the channel spacing of part (b). Are you surprised?

d) Bit error probability measurement (BEP) with ACI. The goal is to plot measured BEP results and compare to ideal BPSK similar to that of Figure 15 in Appendix C. The plot set-up is the following:

```
# Data for BEP theory plot
EbN0_dB_plot = arange(0,10.6,0.5)
Pe_thy_plot = 1/2*special.erfc(sqrt(10**(EbN0_dB_plot/10)))

figure(figsize=(5,6))
semilogy(EbN0_dB_plot,Pe_thy_plot)
# Sample/place-holder measured data
semilogy([6,8,10],[1e-3,1e-3,1e-3],'rd')
ylim([1e-6,1])
ylabel(r'Probability of Bit Error')
xlabel(r'Received $E_b/N_0$ (dB)')
title(r'BPSK BEP with Adjacent Channel Interference')
legend((r'Ideal Theory',r'Experimental with ACI'),loc='best')
grid();
```

Collect **three** experimental BEP data points using the `f_adj` value you found in part (b). Continue to keep Df = 0. Choose `EbN0_dB` values that produce BEP estimates in the interval $[10^{-5}, 10^{-3}]$. For statistical confidence you need to run enough bits through the system to produce at least 100 bit errors. Configure the simulation cell accordingly:



```
# Set the Eb/N0 value in dB for the channel
EbN0_dB = 5
     ⋮
# Enter the number of bits simulated, here 10000
x,b,a_in = dc.NRZ_bits(100000,Ns,'src')
#
     ⋮
kmax =  0, taumax = 12
BEP: Nbits = 99988, Nerror = 969, Pe_est = 9.691e-03
Pe_thy = 5.954e-03
```

How many dB to the right of the BPSK ideal theory curve does your BEP cross 10^{-4}? As a result of the ACI impairment, we expect the BEP experiment values to be shifted to the right as described in Appendix C .

e) Now switch the simulation back to the speech signal keeping the ACI signals as they were in part (b), except set `Df` back to $0.012R_b$. Listen to the short audio segment with just ACI and $E_b/N_0 = 100$ dB as in the measurements of (b). **Note**: Please stay with the audio file parsing given in the code at the bottom of page 8. You should hear the speaker say “The little tales they tell are false.” Decrease `EbN0_dB` down to just 7 dB and listen again. Comment on what you hear.

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 7.

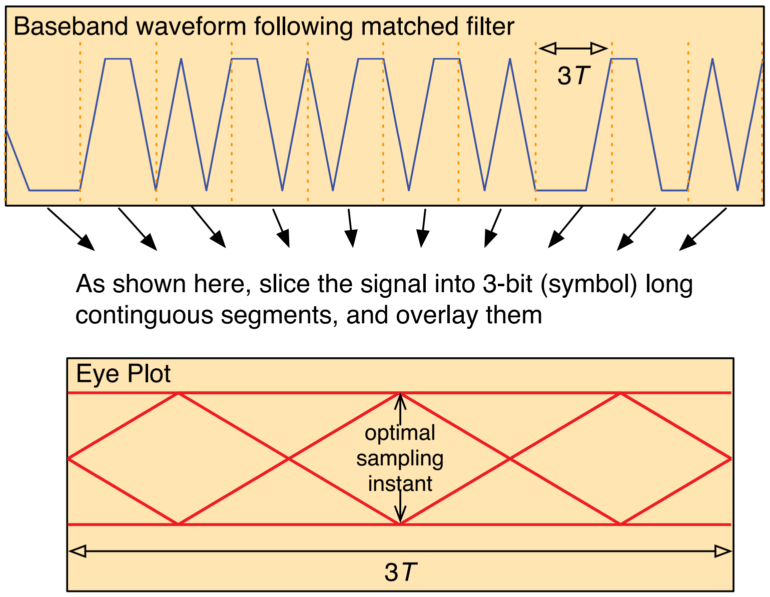

Baseband waveform following matched filter
3T
As shown here, slice the signal into 3-bit (symbol) long contingUous segments, and overlay them
Eye Plot
optimal sampling instant
3T

Figure 7: 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 8, 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)
```

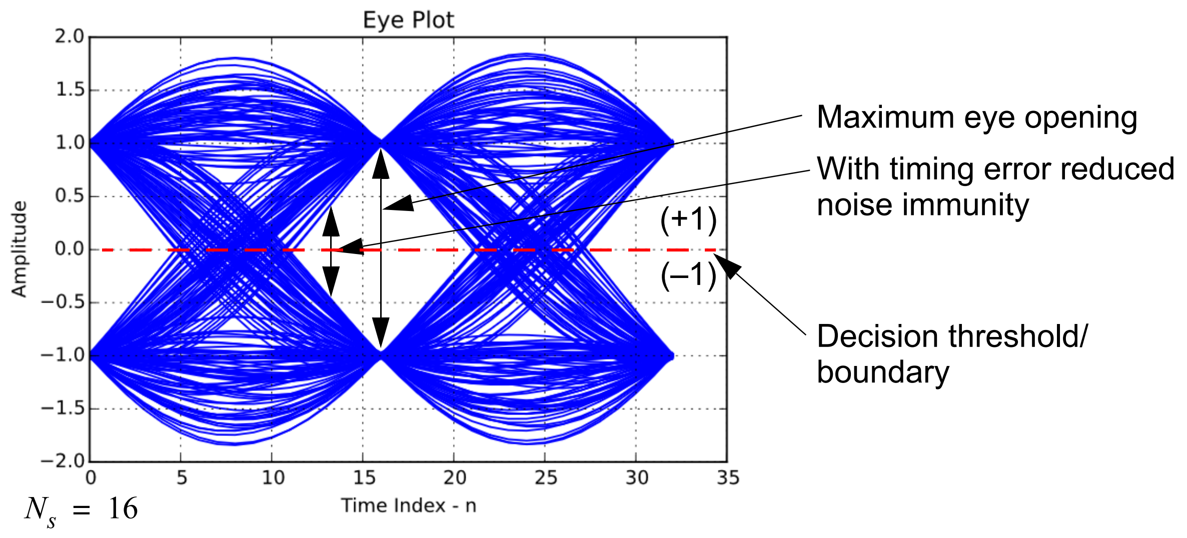


Figure 8: Example eye plot taken at very high E_b/N_0.

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 seconds or N_s samples if in the discrete-time domain 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 9), resulting in bit errors.

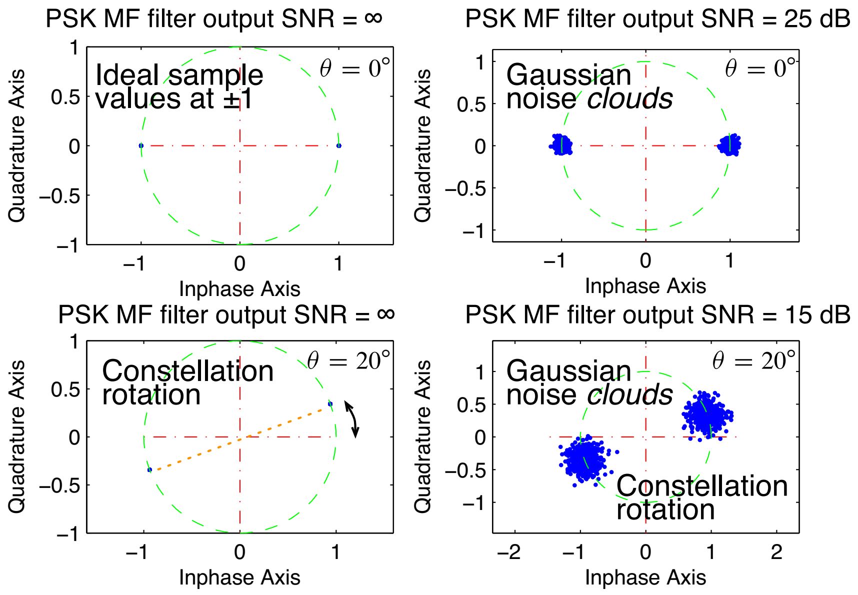


Figure 9: A collection of scatter plots for BPSK under various operating conditions.

Consider now a specific example in Python. Figure 10 shows you can use the function

```python
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

`dc.scatter()` or just use plot and break out the real and imaginary parts on your own. You

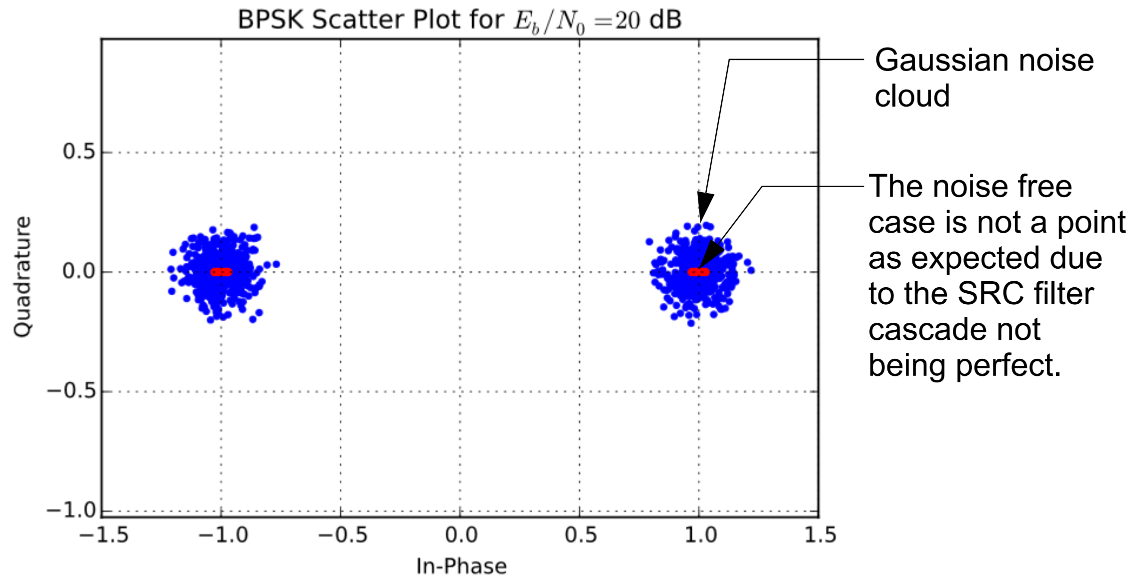


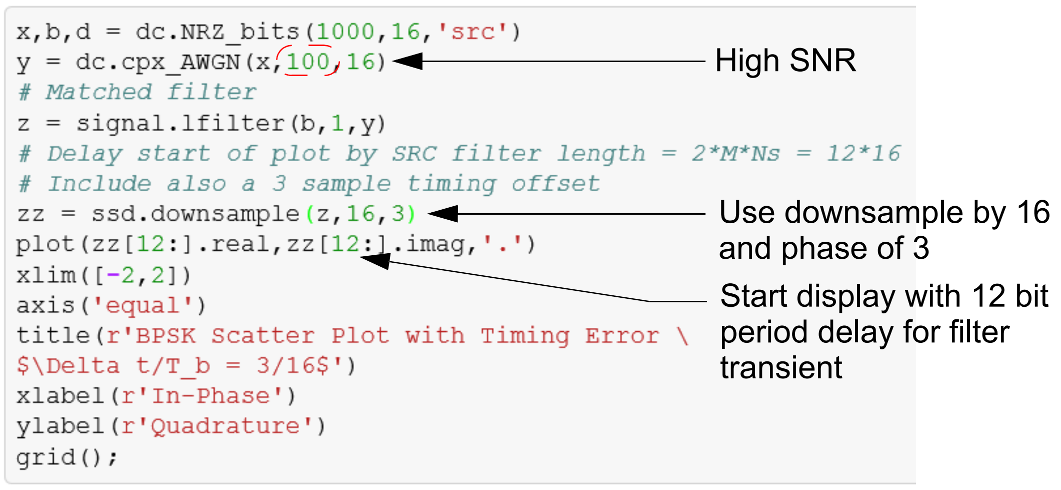
Figure 10: An over plot of two scatter plots, one at 20 dB and one at very high SNR.

will also have choose the sampling instant and the stride interval, e.g,

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

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

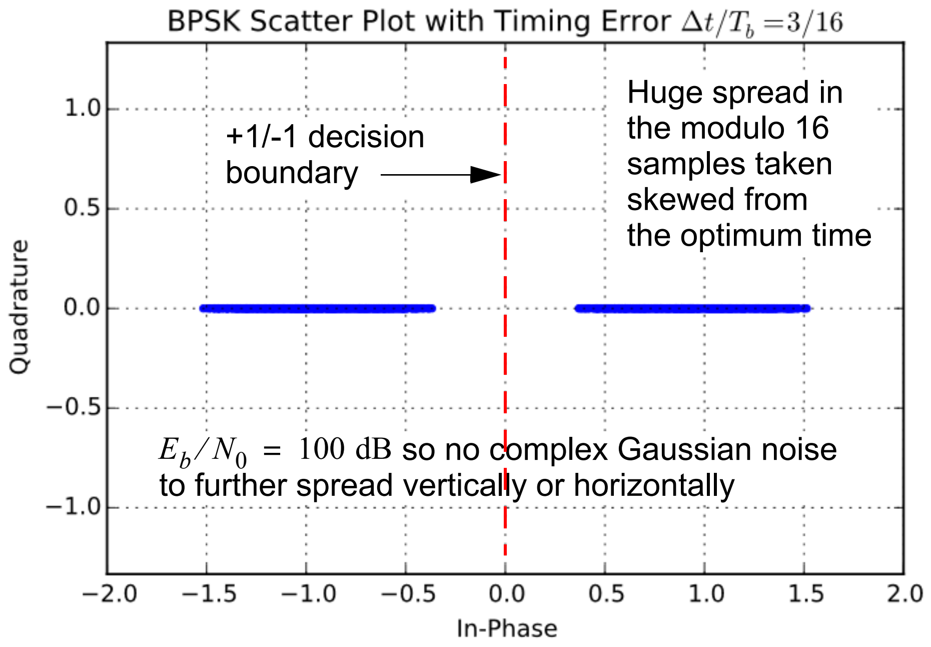


Figure 11: Scatter plot with timing error of $\Delta T/T_b = 3/16$.

Appendix B: Decision Directed Carrier Phase Tracking

The carrier phase tracking algorithm inside of `pll.DD_carrier_sync()` is a decision directed (DD) algorithm [2]. What this means is ±1 decisions are made on the I/Q (complex baseband signal) and then compared with the input I/Q signal in order to form a tracking error measurement. A phase-locked loop (PLL) is configured with the error signal as the input followed by a loop filter, and digital VCO (direct digital synthesis algorithm), that drives a phase rotator (complex multiply by $e^{-j\hat{\theta}}$). The block diagram of the system is shown in Figure 12 below.

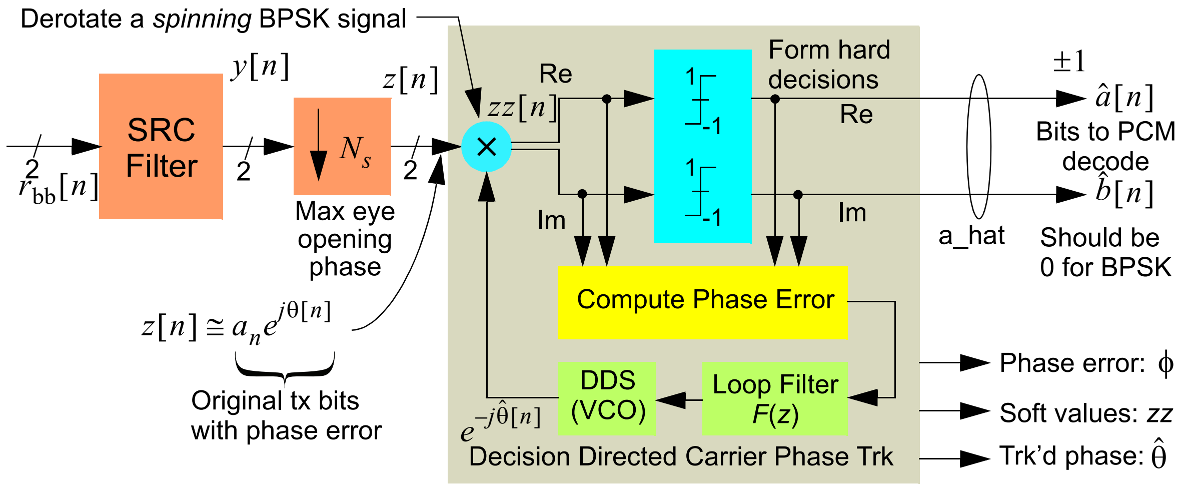


Figure 12: Decision directed (DD) carrier phase tracking system.

The function `DD_carrier_sync()` takes four inputs: (1) the complex baseband input signal, assumed to be sampled at the bit rate at the maximum eye opening, (2) the modulation order,

M, which for BPSK is 2, (3) the PLL loop bandwidth in Hz normalized by the bit rate, and (4) the second-order PLL damping factor ζ.

A simple test of carrier phase tracking is to generate baseband BPSK, introduce a small frequency error (frequency uncertainty can come from a variety of sources), then matched filter signal (MF), sample at the maximum eye opening, then enter the one sample per bit signal into the DD phase tracking system. Figure 13 shows plots for $E_b/N_0 = 100$ dB and frequency error $\Delta f = 0.012R_b$.

```python
x,b,d = dc.NRZ_bits(1000,16,'src')
# AWGN channel at high Eb/No = SNR
y = dc.cpx_AWGN(x,100,16)
# Introduce frequency error
n = arange(0,len(x))
Df = 0.012/16;
y *= exp(1j*2*pi*Df*n)  # Df = 0.012*Rb or 12% of Rb
# Matched filter
z = signal.lfilter(b,1,y)
# Sample at once per bit (Ns=16)
z = ssd.downsample(z,16,0)
# DD carrier phase sync: 2<=>BPSK, 0.05=loop BW/Rb,0.70=loop damping
zz,a_hat,e_phi,theta_hat = pll.DD_carrier_sync(z,2,0.1,0.707)
```

```python
subplot(221)
plot(z[12:].real,z[12:].imag,'.')
axis('equal')
title(r'Phase Track Input')
xlabel(r'In-Phase')
ylabel(r'Quadrature')
grid();
subplot(222)
plot(e_phi)
title(r'Phase Error $\hat{\theta}$')
xlim([0,250])
xlabel(r'Bits')
ylabel(r'Radians')
grid();
subplot(223)
plot(zz[100:].real,zz[100:].imag,'.')
axis('equal')
title(r'Phase Track Output')
xlabel(r'In-Phase')
ylabel(r'Quadrature')
grid();
subplot(224)
plot(theta_hat)
title(r'Tracked Phase $\hat{\theta}$')
xlabel(r'Bits')
ylabel(r'Radians')
#xlim([0,250])
grid();
```

to ignore the transient start plotting here

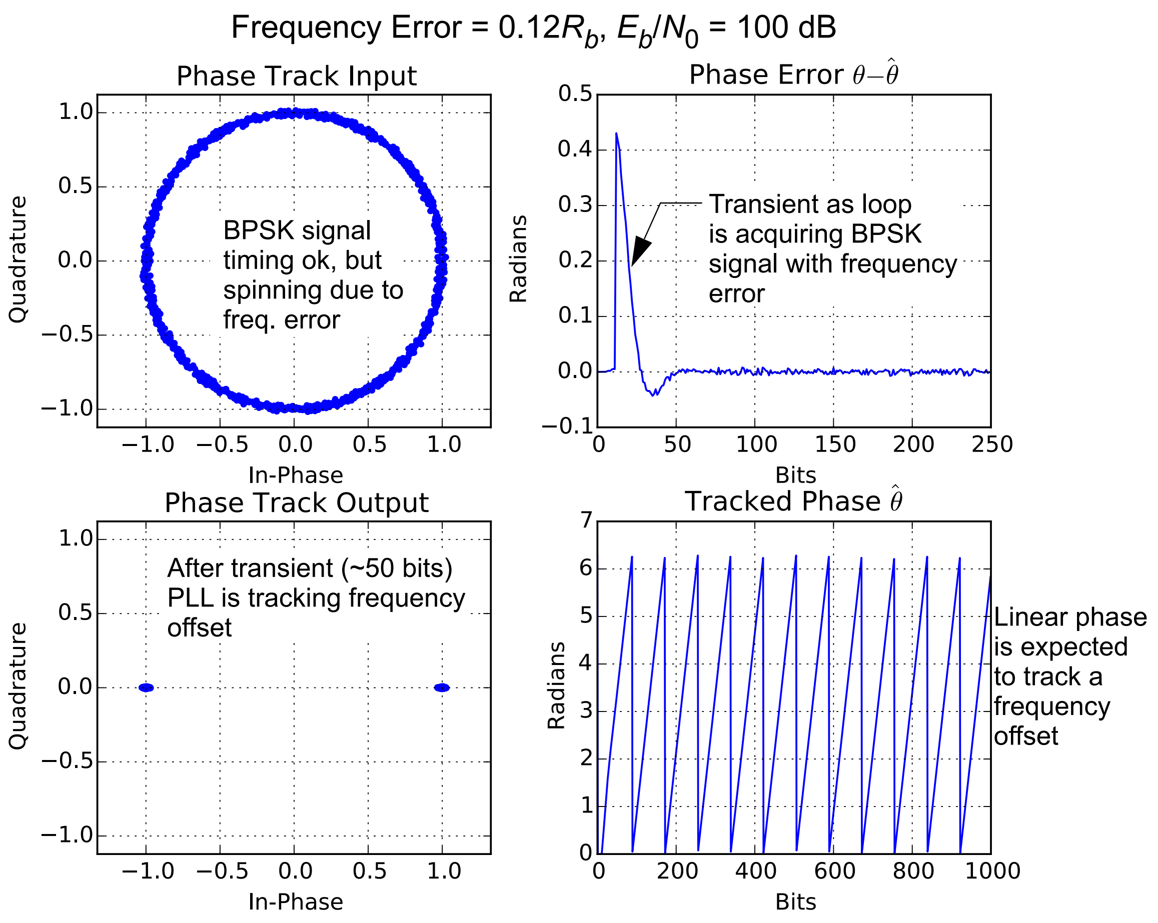


Figure 13: Measurements taken around the DD carrier phase synchronization system at high SNR.

As E_b/N_0 (SNR) is decreased, say down to 20 dB, the tracking error variance increases and the scatter plots, before and after carrier phase tracking, show evidence of the channel noise. This

```
x,b,d = dc.NRZ_bits(1000,16,'src')
# AWGN channel at high Eb/No = SNR
y = dc.cpx_AWGN(x,20,16)
# Introduce frequency error
n = arange(0,len(x))
Df = 0.012/16;
y *= exp(1j*2*pi*Df*n) # Df = 0.012*Rb or 12% of Rb
# Matched filter
z = signal.lfilter(b,1,y)
# Sample at once per bit (Ns=16)
z = ssd.downsample(z,16,0)
# DD carrier phase sync: 2<=>BPSK, 0.05=loop BW/Rb,0.70=loop damping
zz,a_hat,e_phi,theta_hat = pll.DD_carrier_sync(z,2,0.1,0.707)
```

is exemplified in Figure 14.

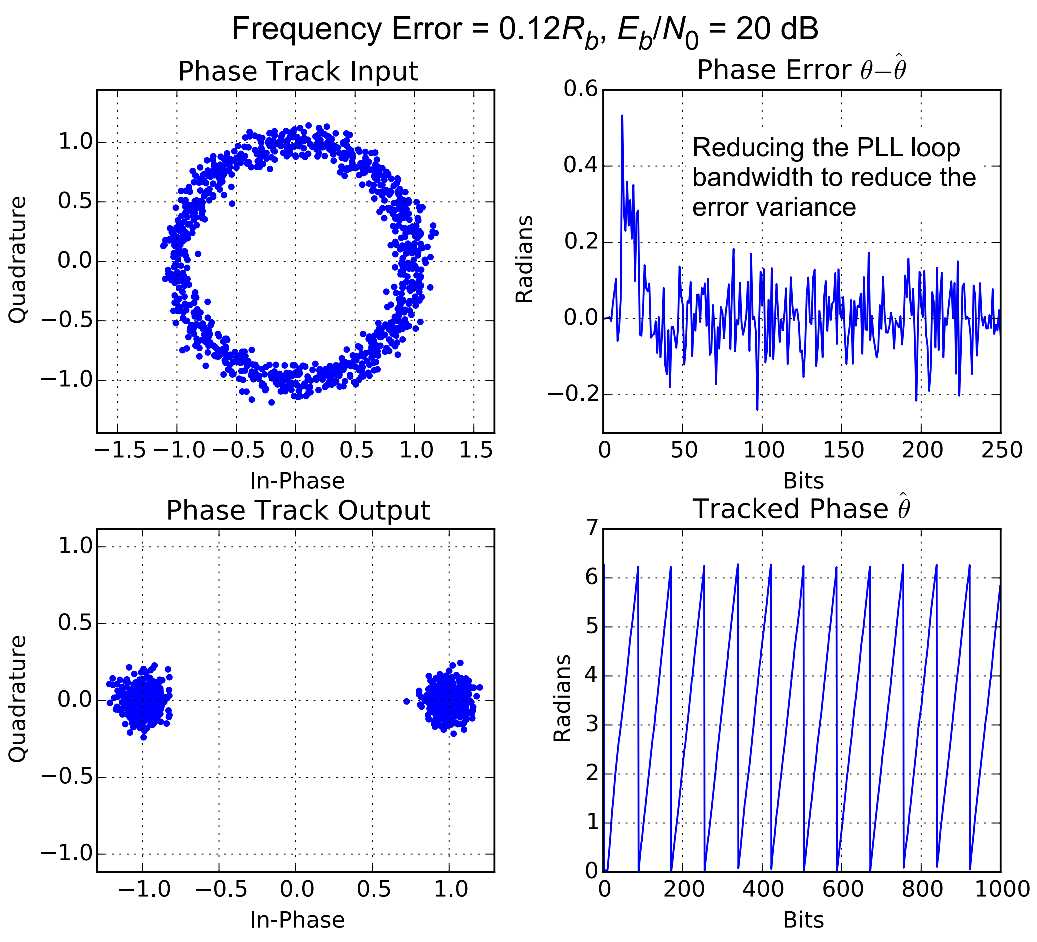


Figure 14: Measurements taken around the DD carrier phase synchronization system at $E_b/N_0 = 20\,\text{dB}$.

The error variance noticeable in the phase error signal (`e_phi`) can be reduced by decreasing the PLL loop bandwidth (third argument) in the function call to `pll.DD_carrier_sync()`, This is good from a tracking standpoint, but now the loop will take longer to settle and will have a harder time locking to larger frequency errors Δf. Recall that a second-order PLL can track a signal with large frequency error, but initial acquisition may take a long time.

Another aspect of the DD phase tracking algorithm is that there are multiple stable lock points. For BPSK ($M = 2$) there are two stable lock points $0°$ and $180°$. The PLL locks at $180°$ the output data bits will be inverted from the way they entered at the transmitter. *Ambiguity resolution* is required to resolve the inverted data bits scenario. There are various ways of taking care of this in practice. For the purposes of this project inverted data bits end up inverting the analog message, which is not really noticeable.

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 therm BEP. In any case, the theoretical curve of interest here is

$$P_{e,\,\text{thy}} = \frac{1}{2}\text{erfc}\left(\sqrt{\frac{E_b}{N_0}}\right) = \frac{1}{2}\text{erfc}\left(\sqrt{10^{\text{SNR}_{\text{dB}}/10}}\right) \tag{6}$$

versus $(E_b/N_0)_{\text{dB}} = \text{SNR}_{\text{dB}}$. The curve is plotted in Figure 15 using a log scale for the probability axis.

```
EbN0_dB = arange(0,12,.2)
Pe = 1/2*special.erfc(sqrt(10**(EbN0_dB/10)))
figure(figsize=(5,5))
semilogy(EbN0_dB,Pe) # ideal theory
semilogy(5,5.875e-3,'rd')
xlabel(r'$E_b/N_0$ (dB)')
ylabel(r'Probability of Bit Error')
title(r'BPSK BEP in AWGN')
xlim([0,14])
ylim([1e-6,1e-1])
legend((r'Ideal Theory',r'Measured'),loc='upper right')
grid();
```

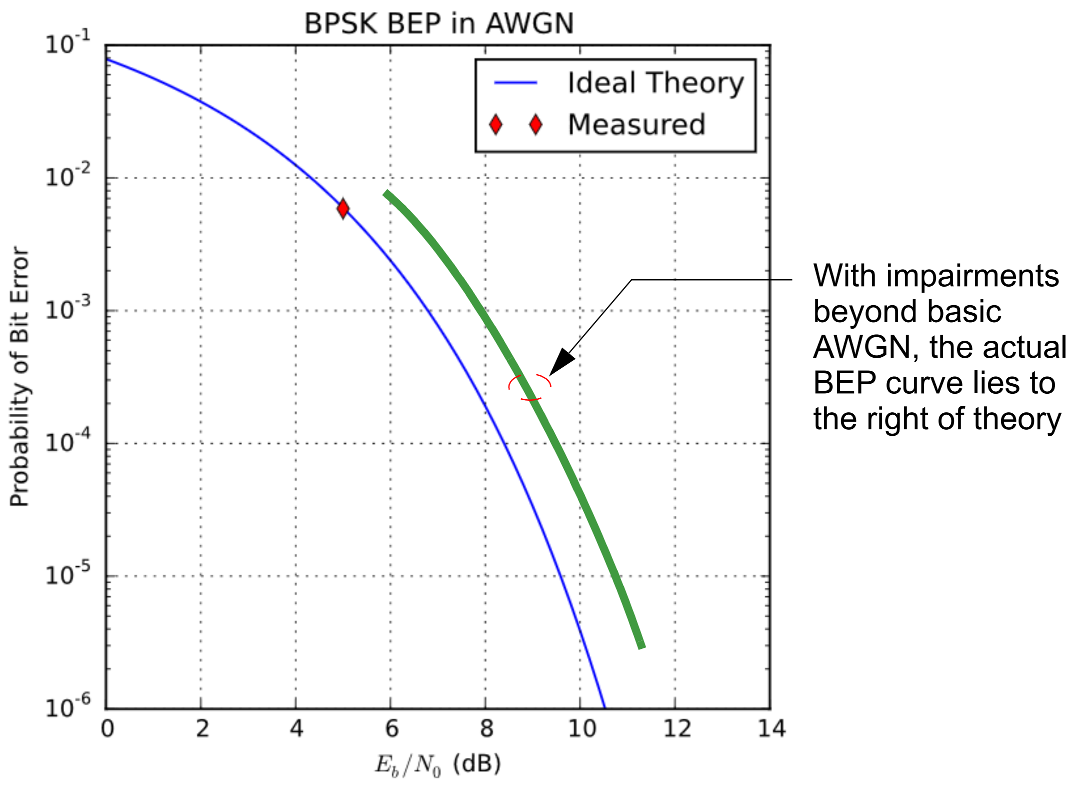


Figure 15: Theoretical BPSK BEP curve including one measured point.

In characterizing various impairments in digital comm, such as 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 15. Note for experimental points statistical accuracy requires at least 100 error events be observed. Here 940 errors were counted.