Data Transmission in Noise

June 30, 2025

1 Data Transmission in Noise

1.1 Z&T Chapter 9

[1]:
```
%pylab inline
#%matplotlib qt
import sk_dsp_comm.sigsys as ss
import scipy.signal as signal
import sk_dsp_comm.digitalcom as dc
import scipy.special as special
import scipy.stats as stats
%config InlineBackend.figure_formats=['svg'] # SVG inline viewing
```

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

1.2 Binary Antipodal Probability of Bit Error

1.2.1 Define the Q-Function In Terms of the erfc

[2]:
```
def Q(x):
    return 1./2*special.erfc(x/sqrt(2.0))
    #return stats.norm.sf(x)
```

Alternatively use the `scipy.stats.norm` package and in particular the functions `sf()` and `isf()`:

- `sf(x, loc=0, scale=1)` is the survival function (also defined as 1 - cdf, but `sf` is sometimes more accurate). It is equivalent to the `Q()` function in the text.
- `isf(q, loc=0, scale=1)` is the inverse survival function (inverse of sf), or the inverse `Q()` function.
- Another alternative is to use the $Q()$ function that is inside `digitalcom`: `dc.q_fctn()`.

[3]:
```
x = arange(0,4,.5)
Q(x)
```

[3]:
```
array([5.00000000e-01, 3.08537539e-01, 1.58655254e-01, 6.68072013e-02,
       2.27501319e-02, 6.20966533e-03, 1.34989803e-03, 2.32629079e-04])
```

[4]:
```
x = arange(0,4,.5)
dc.q_fctn(x)
```

[4]:
```
array([5.00000000e-01, 3.08537539e-01, 1.58655254e-01, 6.68072013e-02,
       2.27501319e-02, 6.20966533e-03, 1.34989803e-03, 2.32629079e-04])
```

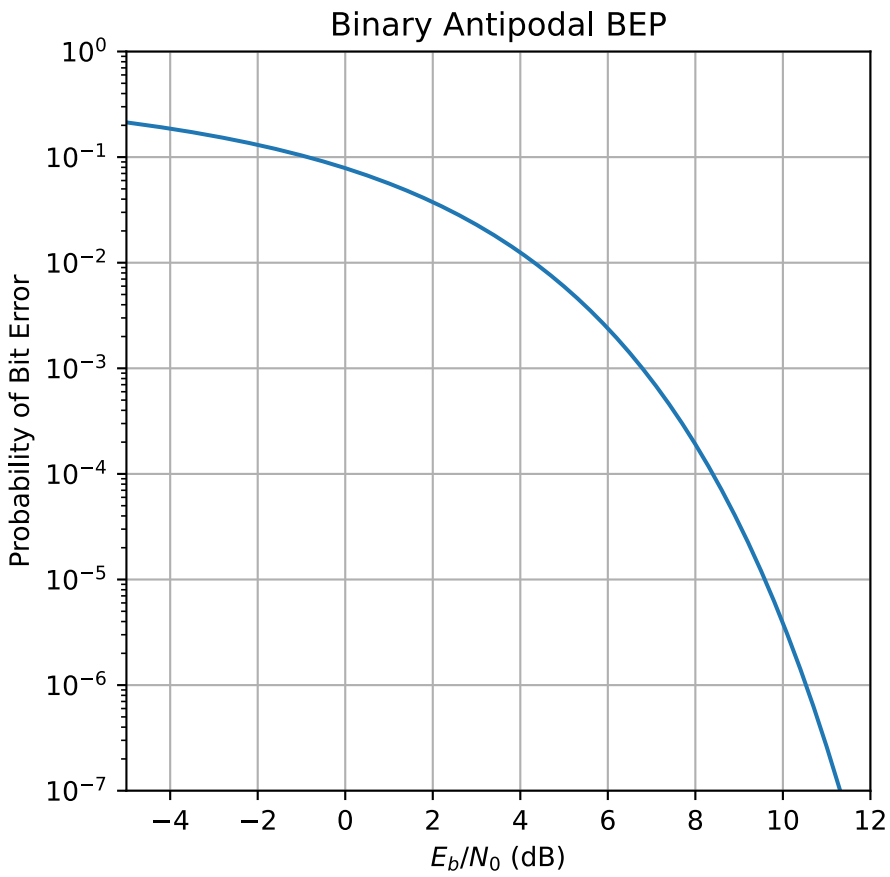
[5]:
```
# Compare with norm.sf
stats.norm.sf(x)
```

[5]:
```
array([5.00000000e-01, 3.08537539e-01, 1.58655254e-01, 6.68072013e-02,
       2.27501319e-02, 6.20966533e-03, 1.34989803e-03, 2.32629079e-04])
```

[6]:
```
# Check the inverse Q or norm.isf
x_inv = stats.norm.sf(x)
stats.norm.isf(x_inv)
```

[6]:
```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5])
```

[7]:
```
figure(figsize=(5,5))
EbN0 = arange(-5,12.1,.1)
semilogy(EbN0,Q(sqrt(2*10**(EbN0/10))))
grid()
xlim([-5,12])
ylim([1e-7, 1])
xlabel(r'$E_b/N_0$ (dB)')
ylabel('Probability of Bit Error')
title('Binary Antipodal BEP');
```



1.2.2 Find Eb/N0 Values in dB to Achieve a Particular BEP Value

For binary antipodal we know that

$$P_e = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

Using the inverse Q-function we can first solve for E_b/N_0 and then convert to dB:

$$\frac{E_b}{N_0} = \frac{1}{2}\left[Q^{-1}\left(P_e\right)\right]^2,$$

so in dB

$$\left(\frac{E_b}{N_0}\right)_{\text{dB}} = 10\log_{10}\left[\frac{1}{2}\left[Q^{-1}\left(P_e\right)\right]^2\right]$$

[8]:
```
Pe = [1e-4,1e-5,1e-6,1e-7]
for n, Pe_n in enumerate(Pe):
    EbN0_dB = 10*log10(stats.norm.isf(Pe[n])**2/2)
    print('Pe = %2.1e <==> (Eb/No) = %6.3f dB' % (Pe_n,EbN0_dB))
```

```
Pe = 1.0e-04 <==> (Eb/No) =  8.398 dB
Pe = 1.0e-05 <==> (Eb/No) =  9.588 dB
Pe = 1.0e-06 <==> (Eb/No) = 10.530 dB
Pe = 1.0e-07 <==> (Eb/No) = 11.309 dB
```

1.3 Simple Numerical Optimzation

In Set #3 and in general, numerical optimization comes in handy. Two situations of interest are in finding the max/min of a function of finding a root or zero crossing. Within SciPy is the module *optimize*. There are many powerful routines to choose from. From the simple scenario described above we just need to consider two functions.

1.) Find roots or the zero crossing of a function using *bisection* or *brent*
2.) Find the minimum of a function (to find the max take the negative of the function) using *minimize_scalar*

Consider an objective functions to try out (1) and (2) on. Then, code the functions and find the solutions. As a sample let

$$y = f(x) = \cos(2\pi x)e^{-ax},\ 0 \le x \le 2 \tag{1}$$

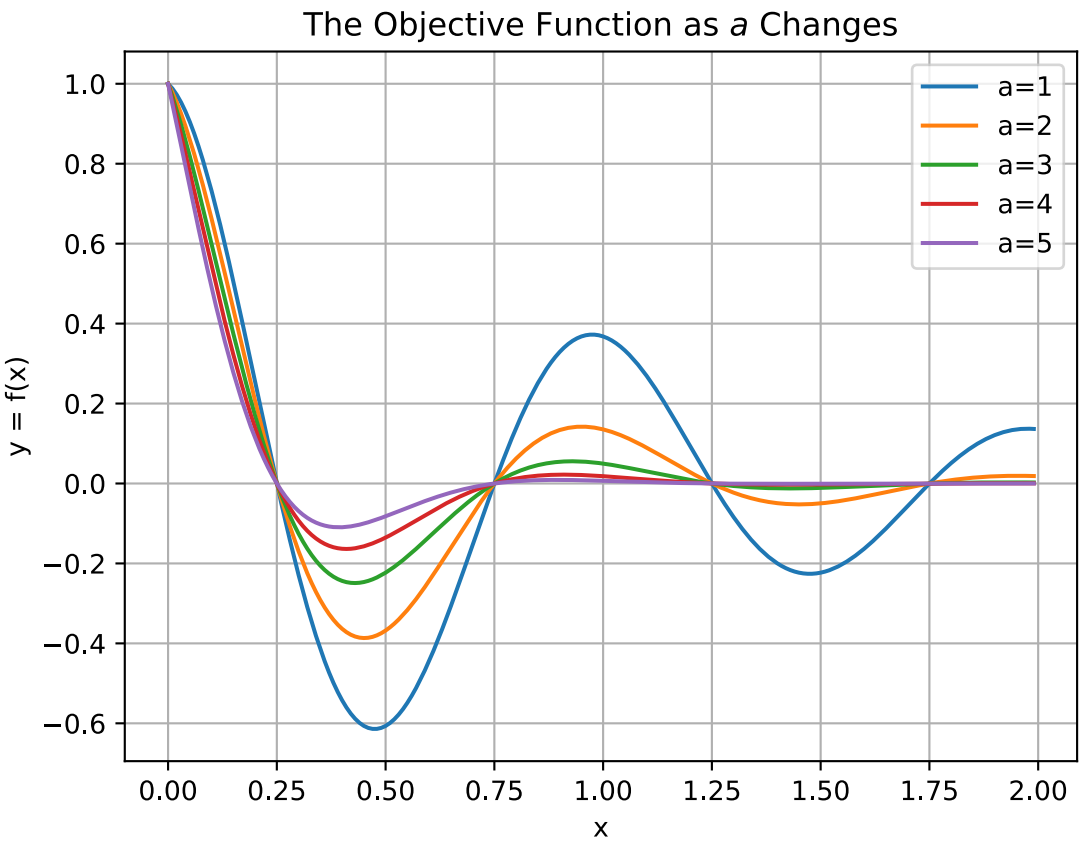
where a is a parameter.

```
Definition:  optimize.minimize_scalar(fun, bracket=None, bounds=None,
args=(), method='brent', tol=None, options=None)
Definition:  optimize.brentq(f, a, b, args=(), xtol=1e-12,
rtol=4.4408920985006262e-16, maxiter=100,
full_output=False, disp=True)
```

[9]:
```
# The objective function with a as a parameter
def y_test(x,a):
    return cos(2*pi*x)*exp(-a*x)
```

Plot the Function

[10]:
```
x = arange(0,2,.01)
a = arange(1,5+1)
for k,ak in enumerate(a):
    plot(x,y_test(x,ak))
grid()
legend((r'a=1',r'a=2',r'a=3',r'a=4',r'a=5'),loc='best')
xlabel('x')
ylabel('y = f(x)')
title('The Objective Function as $a$ Changes');
```



Find the Miniumum: $a = 3.0$

[11]:
```
import scipy.optimize as optimize
min = optimize.minimize_scalar(y_test,(0,0.5),args=(3,)) #set a = 3.0
min # note min is a dictionary containing 4 entries
```

[11]:
```
  message:
            Optimization terminated successfully;
            The returned value satisfies the termination criteria
            (using xtol = 1.48e-08 )
  success: True
      fun: -0.24907729778054052
        x: 0.4291032378728843
      nit: 13
     nfev: 16
```

Find the Minimum: As a Steps over $1.0, 2.0, \ldots, 5.0$

Consider the minimum over a range of parameter a values. I will print a simple table as a is stepped from 1 to 5 with a step size of 1.

[12]:
```
a = arange(1,5+1,1)
for k,ak in enumerate(a):
    min = optimize.minimize_scalar(y_test,(0,0.5),args=(ak,))
    print('a = %6.4f, x_min = %6.4f, y_min = %6.4f' % (ak,min['x'],min['fun']))
```

```
a = 1.0000, x_min = 0.4749, y_min = -0.6142
a = 2.0000, x_min = 0.4510, y_min = -0.3867
a = 3.0000, x_min = 0.4291, y_min = -0.2491
a = 4.0000, x_min = 0.4098, y_min = -0.1638
a = 5.0000, x_min = 0.3930, y_min = -0.1097
```

Find the Second Zero Crossing: $a = 3.0$

[13]:
```
optimize.brentq(y_test,0.5,1.0,args=(3.0,))
```

[13]: 0.75

Find the Third Zero Crossing: $a = 3.0$

[14]:
```
optimize.brentq(y_test,1.0,1.5,args=(3.0,))
```

[14]: 1.25

1.4 Using SymPy for Root Finding

[16]:
```
from IPython.display import display
from sympy.interactive import printing
printing.init_printing(use_latex='mathjax')
import sympy as sym
```

Define some symbolic variables for use in the remainder of this example.

[17]:
```
Pe, SNR, SNRdB, SJR, SJRdB = sym.symbols("P_E, SNR, SNR_dB, SJR,␣
↪SJR_dB",positive=True)
x = sym.symbols("x", real=True)
```

Define the Q(x) Function

[18]:
```
Qsymp = 1/2*sym.erfc(x/sym.sqrt(2))
Qsymp
```

[18]:
$$0.5\,\mathrm{erfc}\left(\frac{\sqrt{2}x}{2}\right)$$

Problem Definition Here we conider a simple model for the BEP of BPSK receiver impaired by a bradband noise jammer. The BEP is modeled with the addition of term N_j which represents a jamming noise spectral density. The BEP expression is

$$P_E = Q\left(\sqrt{\frac{2E_b}{N_0 + N_j}}\right) = Q\left(\sqrt{\frac{2}{N_0/E_b + N_j/E_b}}\right)$$

Next we write the P_E expression using the Q-function defined using `sympy` functions, where we define SNR $= E_b/N_0$ and SJR $= E_b/N_j$. In working with `SymPy` it is very common to make variable substitutions, such as in the creation of the `Pe` symbolic function below:

[19]:
```
Pe = Qsymp.subs({x:sym.sqrt(2/(1/SNR+1/SJR))})
Pe
```

[19]:
$$0.5\,\mathrm{erfc}\left(\frac{1}{\sqrt{\frac{1}{SNR}+\frac{1}{SJR}}}\right)$$

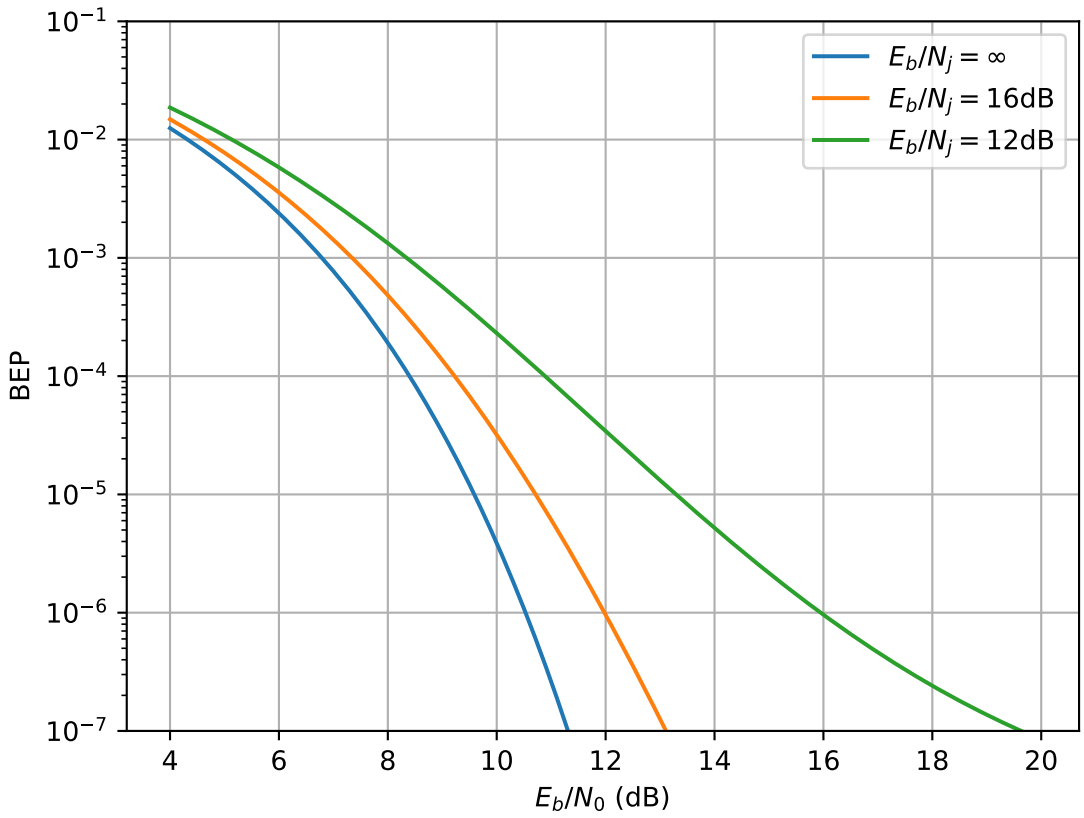
Next we plot some BEP curves by filling arrays using a `for` loop, since the `sympy` function is not immediately vectoriazable as it would be in pure `numpy`. NOte another option is use `sym.lambdify()` on the symbolic expression, but I an running into errors with `erfc` not found. Keep it simple for now, just not as fast using a `for` loop.

We choose a range of 4 to 20 dB in 0.1 dB steps and use SJR values in dB of ∞, 16, and 12 dB:

[20]:
```
SNR_dB_num = arange(4,20,.1)
Pe_val1 = zeros_like(SNR_dB_num)
Pe_val2 = zeros_like(SNR_dB_num)
Pe_val3 = zeros_like(SNR_dB_num)
for k,SNR_dB_k in enumerate(SNR_dB_num):
    Pe_val1[k] = Pe.subs({SNR:10**(SNR_dB_k/10),SJR:sym.oo})
    Pe_val2[k] = Pe.subs({SNR:10**(SNR_dB_k/10),SJR:10**(16/10)})
    Pe_val3[k] = Pe.subs({SNR:10**(SNR_dB_k/10),SJR:10**(12/10)})
```

Plot the results:

[21]:
```
semilogy(SNR_dB_num,Pe_val1)
semilogy(SNR_dB_num,Pe_val2)
semilogy(SNR_dB_num,Pe_val3)
ylabel(r'BEP')
xlabel(r'$E_b/N_0$ (dB)')
ylim([1e-7,1e-1])
legend((r'$E_b/N_j = \infty$',r'$E_b/N_j = 16$dB',r'$E_b/N_j =␣
 ↪12$dB'),loc='best')
grid();
```



The above lets use see how SJR pushed the BEP curves to the right and causes flaring. Suppose now we want to use `sym.nsolve` to find the BEP degradation in dB at $P_E = 10^{-6}$. As a quick check we can use `nsolve` to find the E_b/N_0 value in dB when SJR $\to \infty$:

[22]:
```
sym.nsolve(1e-6 - Pe.subs({SNR:10**(SNRdB/10),SJR:sym.oo}),10)
```

[22]:
10.5298316995714

Note the second argument is the initial guess on the solution. Here I have chosen 10dB. If the guess is too far away from the solution/root, an error occurs. Also in `Sympy` infinity is denoted using `sym.oo`.

Moving on, build up a solution for the degradation when $\text{SJR}_{dB} = 12$ dB. Here we use `sym.N()` to convert to a numerical answer with 6 digit precision.

[23]:
```
SJRdB = 12
Pe_thresh = 1e-6
sym.N(sym.nsolve(Pe_thresh - Pe.subs({SNR:10**(SNRdB/10),SJR:10**(SJRdB/
  ↪10)}),18) \
-sym.nsolve(Pe_thresh - Pe.subs({SNR:10**(SNRdB/10),SJR:sym.oo}),10),6)
```

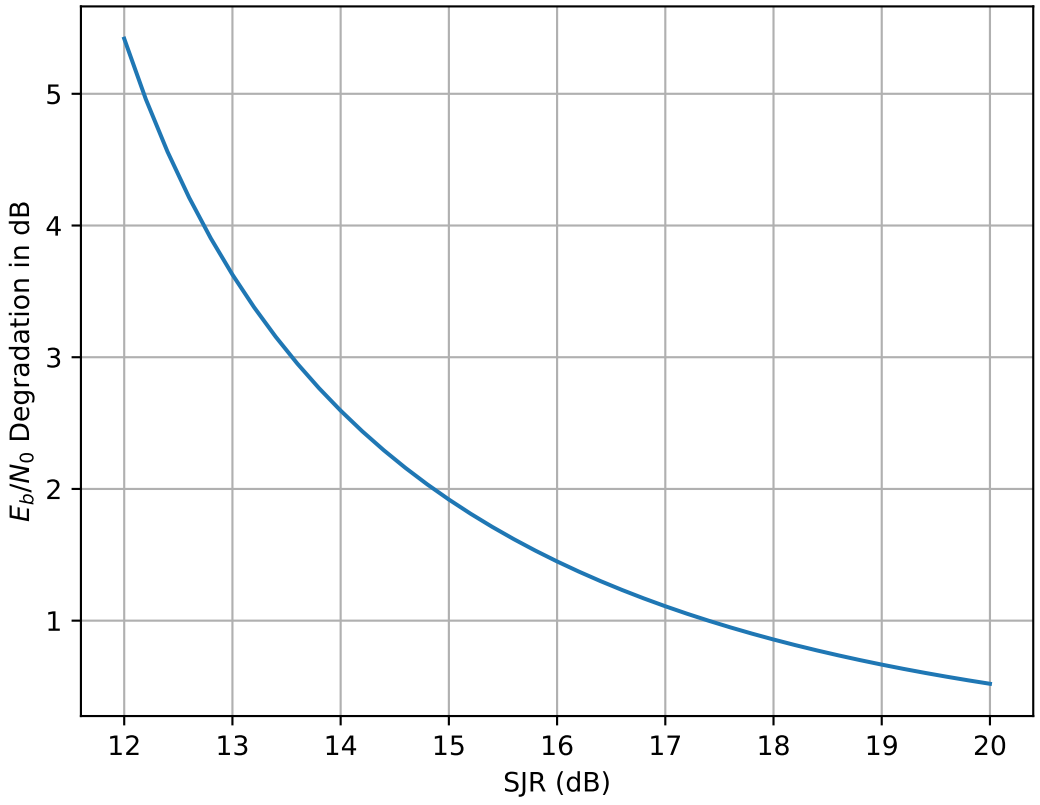
[23]:
5.41854

Create a degradation plot in terms of SJR in dB. Note we must have $\text{SJR}_{dB} > \text{SNR}_{dB}$ otherwise

there is no value of E_b/N_0 that can overcome the jammer power. Some added parameters are included on the first use of `nsolve` to help with convergence over a range of operating points, e.g.,

```
# The (10,20) sets a range of values expected for the solution
# The solver type is set to bisection
# Error reporting is turned off with verify=False,
# which means you are on your own to verify the correct solution
sym.N(sym.nsolve(Pe_thresh - Pe.subs({SNR:10**(SNRdB/10),SJR:10**(SJRdB_k/10)}),
                                    (10,20),solver='bisect',verify=False)
```

[24]:
```
SJRdB = arange(12,20+.2,0.2)
Pe_thresh = 1e-6
Pe_deg_dB = zeros_like(SJRdB)
for k, SJRdB_k in enumerate(SJRdB):
    Pe_deg_dB[k] = sym.N(sym.nsolve(Pe_thresh - Pe.subs({SNR:10**(SNRdB/10),SJR:
 ↪10**(SJRdB_k/10)}),
                                    (10,20),solver='bisect',verify=False) \
    -sym.nsolve(Pe_thresh - Pe.subs({SNR:10**(SNRdB/10),SJR:sym.oo}),10),6)
plot(SJRdB,Pe_deg_dB)
ylabel(r'$E_b/N_0$ Degradation in dB')
xlabel(r'SJR (dB)')
grid();
```



Hopefully the above will help you get started with using `sym.nsolve` for other applications.

1.5 More BEP Analysis and Modeling

1.5.1 Coherent BPSK with Phase Jitter

Consider BPSK with an Imperfect Phase Reference (notes 5-29-31). Here I perform the numerical integration using the function `quad()` found in the SciPy module *integrate*.

```
Definition:  integrate.quad(func, a, b, args=(), full_output=0,
epsabs=1.49e-08, epsrel=1.49e-08, limit=50, points=None,
weight=None, wvar=None, wopts=None, maxp1=50, limlst=50)

Returns
-------
y : float
    The integral of func from `a` to `b`.
abserr : float
    An estimate of the absolute error in the result.
```

[25]:
```
def PE(phi,sig_phi_deg,SNR):
    sigma = pi/180*sig_phi_deg
    pdf = exp(-(phi**2)/(2*sigma**2))/sqrt(2*pi*sigma**2)
    return 2*Q(sqrt(2*SNR*cos(phi)**2))*pdf
```

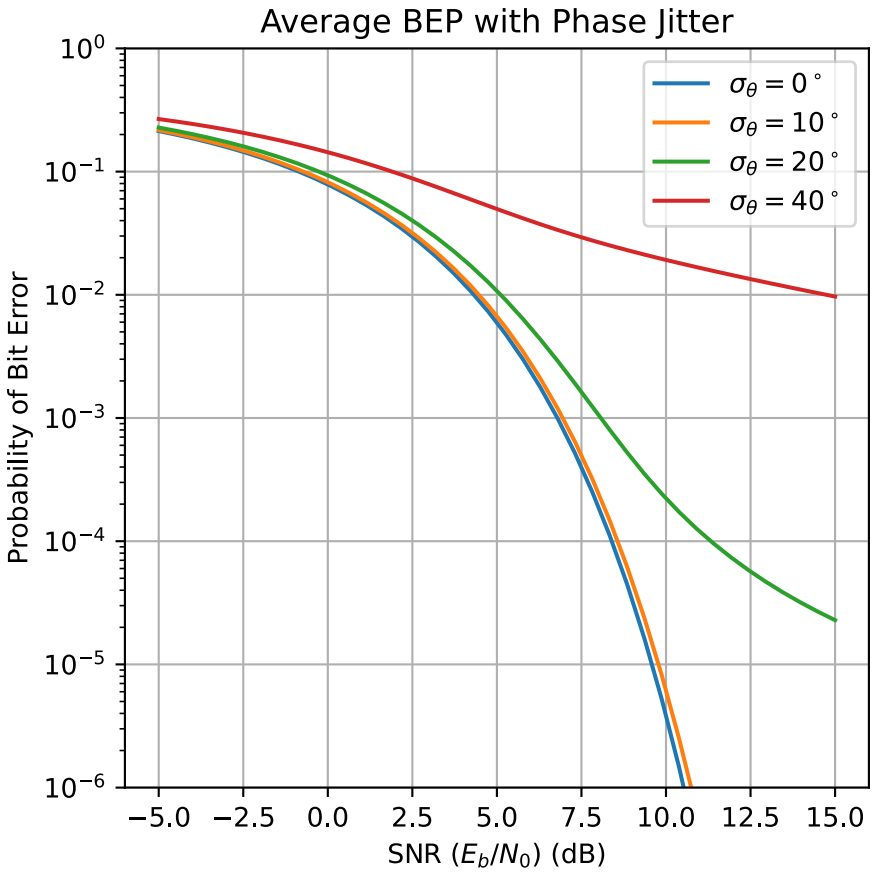
I will plot a few curves similar to those of notes page 5-31 with the aid of the function `PE_jitter`. The `PE_jitter` functions sweeps over a range of SNR values in dB for a fixed value of σ_θ in degrees.

[26]:
```
import scipy.integrate as integrate
def PE_jitter(SNR_range,sig_phi_deg):
    # Add a small value to sig_phi_deg in case the user enters zero
    sig_phi_deg += 0.1
    BEP = zeros_like(SNR_range)
    for k,SNRk in enumerate(SNR_range):
        BEP[k], abserr = integrate.quad(PE,0,pi,args=(sig_phi_deg,10**(SNRk/10.
 ↪)))
    return BEP
```

[27]:
```
figure(figsize=(5,5))
SNR_dB = linspace(-5,15,40)
semilogy(SNR_dB,PE_jitter(SNR_dB,0.0))
semilogy(SNR_dB,PE_jitter(SNR_dB,10.0))
semilogy(SNR_dB,PE_jitter(SNR_dB,20.0))
semilogy(SNR_dB,PE_jitter(SNR_dB,40.0))
ylim([1e-6,1])
xlabel(r'SNR ($E_b/N_0$) (dB)')
ylabel('Probability of Bit Error')
title('Average BEP with Phase Jitter')
legend((r'$\sigma_\theta = 0^\circ$',
```

```
        r'$\sigma_\theta = 10^\circ$',
        r'$\sigma_\theta = 20^\circ$',
        r'$\sigma_\theta = 40^\circ$'),loc='best')
grid();
```

```
/var/folders/nh/4pcjlwf17gd87xk2z4f449ph0000gn/T/ipykernel_83954/4182068714.py:7
: IntegrationWarning: The integral is probably divergent, or slowly convergent.
  BEP[k], abserr = integrate.quad(PE,0,pi,args=(sig_phi_deg,10**(SNRk/10.)))
```



More..

2 Complex Baseband Modeling of ASK, BPSK, and FSK

In this section of I will attempt to parallel the m-code presented in pages 5-41 to 5-64 of the notes. The code is not a 1-1 match as the existing `digitalcom.py` module does things differently, and hopefully more efficiently. As a note, I have added and/or updated some of the existing function found in `digitalcom.py` to complete the task at hand. For this first iteration, some of the code is written into the notebook itself. The original intent of this development was to move the code into the module `sk_dsp_comm.digitalcom`. This is still a possibility.

2.1 Binary ASK

I begin with *ASK* modulation at complex baseband. The *m-code* for ASK contains two functions: (1) for basic ASK generation and a second function that serves as a top level simulation of a transmitter, additive white Gaussian noise channel (AWGN), and a receiver. Probability of bit error (BEP) calculation is also performed.

[28]:
```
def ask(Nbit, Ns, theta):
    """
    x,data = ask(Nbit, Ns, theta)

    Complex baseband binary ask modulator
    ////////////////////////////////////////////////////////////
    Nbit = number of symbols processed
    Ns = the number of samples per bit
    theta = Rotate the complex baseband signal by theta radians
    x = Complex baseband transmit signal vector
    data = Binary {0,1} data being sent; used for error detecting
    ////////////////////////////////////////////////////////////
    Mark Wickert October 2014
    """
    A = 1
    y,b,data = dc.rz_bits(Nbit, Ns) # Rect pulse shape by default
    x = A*y*exp(1j*theta)
    return x, b, data
```

2.1.1 A Top Level Simulation Function

[29]:
```
def ask_sim(SNR,theta,Nbits,timing):
    """
    Pe,Bit_errors,Bit_count,z = ask_sim(SNR,theta,Nbits,timing)

    Coherent ASK modem end-to-end simulation function
    ////////////////////////////////////////////////////////////
           SNR = Set the received signal SNR in dB; for on-off keying
                 the SNR is defined interms of the average signal
                 power, e.g., 1/2 the peak symbol energy
         theta = Rotate the complex baseband signal by theta radians
         Nbits = The number of bits simulated
        timing = Sets the sampler timing to give the minimum BEP;
                 find value using eyeplot_mark function
            Pe = Estimated bit error probability (BEP)
    Bit_errors = Number of bit errors found (good to obtain at least
                 100 errors over an AWGN channel
     Bit_count = The number of bits processed y bit_errors function;
                 If you create a 'top-level' function 'errors' and
                 'RecBits' can be used to combine results from multiple
                  runs
```

```
            z = Receiver decision statistic; use as input to eyeplot
                for further analysis, e.g., find a good value
                for 'timing'
    ////////////////////////////////////////////////////////////////
    Mark Wickert October 2014
    """
    Ns = 16 # Fix the number of samples per bit to 16

    # Generate ASK tx signal
    x,b,data = ask(Nbits, 16, theta)
    y = dc.cpx_awgn(x,SNR,16) # AWGN channel

    # Enter receiver
    # Coherent receiver (at complex baseband)
    #////////////////////////////////
    # Matched filter is a rectangle pulse
    b_REC = b/sum(b) # make filter unity gain at dc
    # Process complex BB signal through filter to form decision statistic
    z = signal.lfilter(b_REC,1,y)
    z_det = z.real
    # Decision logic
    z_det = z_det - mean(z_det) # bias about zero for detection
    d_hat = (sign(z_det[timing::16])+1)/2 # {0,1} logic levels
    # Error detect
    Bit_count, Bit_errors = dc.bit_errors(data,d_hat)
    Pe = Bit_errors/float(Bit_count)
    return Pe,Bit_errors,Bit_count,z
```

Run the simulation and generate a BEP data point:

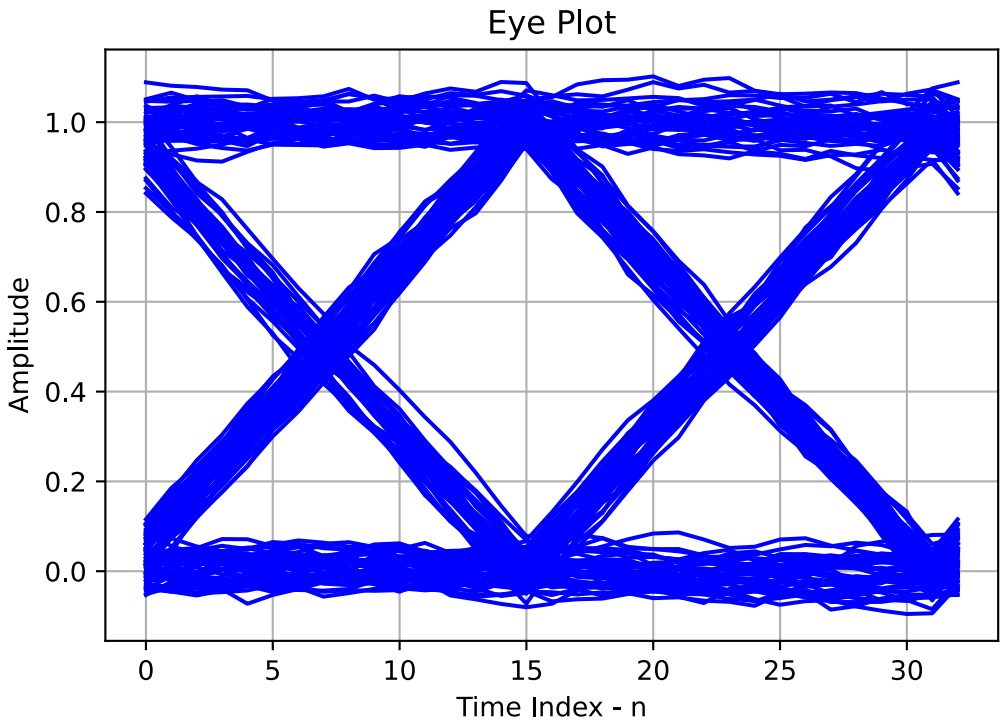
[30]:
```
Pe,Bit_errors,Bit_count,z = ask_sim(20,0*pi/180.,10000,15)
print(' Bit Count = %d' % Bit_count)
print('Bit Errors = %d' % Bit_errors)
print('        Pe = %6.3e' % Pe)
```

```
 Bit Count = 10000
Bit Errors = 0
        Pe = 0.000e+00
```

2.1.2 Consider the Eye Plot

[31]:
```
dc.eye_plot(z[:5000].real,32,16)
```

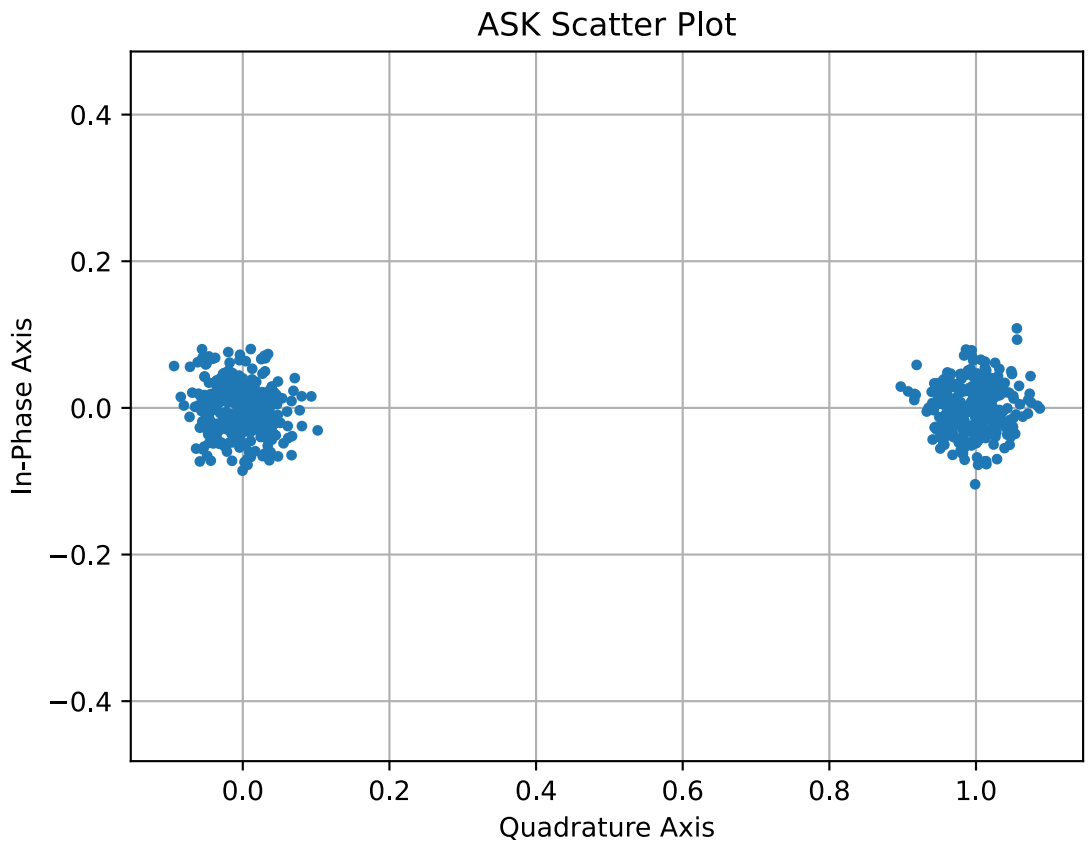
[31]:
0



2.1.3 Consider the Scatter Plot

Two ways to do this: (1) with some help from `digitalcom` and (2) just do it on your own.

[32]:
```
# xI,xQ = dc.scatter(z,16,15)
# plot(xI,xQ,'.')
plot(z[15:10000:16].real,z[15:10000:16].imag,'.')
grid()
title('ASK Scatter Plot')
xlabel('Quadrature Axis')
ylabel('In-Phase Axis')
axis('equal');
```



2.2 Binary PSK

Note that by setting $m = 0$ in the function `psk()` you can obtain ordinary BPSK.

[33]:
```
def psk(m, Nbit, Ns, theta):
    """
    x,data = ask(, Nbit, Ns, theta)

    Complex baseband binary psk modulator
    ////////////////////////////////////////////////////////////////
    m = modulation index; m = 0 ==> BPSK
    Nbit = number of symbols processed
    Ns = the number of samples per bit
    theta = Rotate the complex baseband signal by theta radians
    x = Complex baseband transmit signal vector {-1,+1}
    data = Binary {0,1} data being sent; used for error detecting
    ////////////////////////////////////////////////////////////////
    Mark Wickert October 2014
    """
    A = 1
```

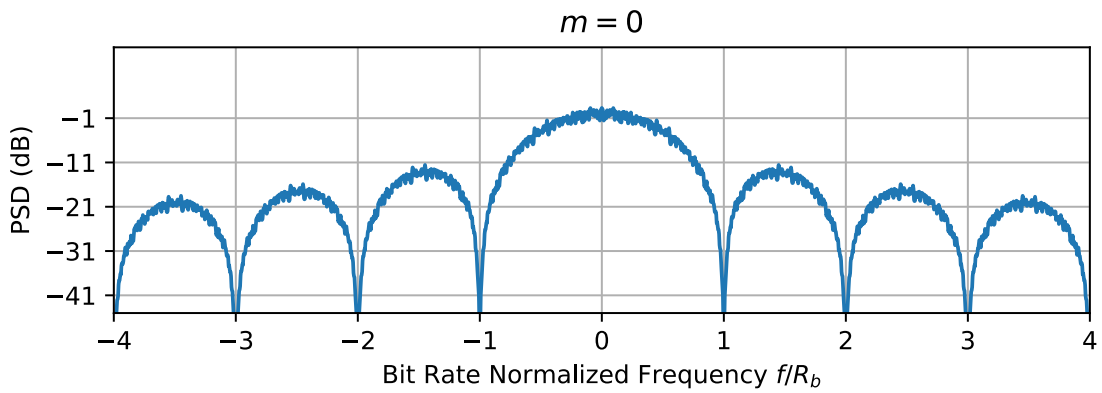
```
    y,b,data = dc.nrz_bits(Nbit, Ns) # Rect pulse shape by default
    x = A*(sqrt(1 - m**2)*y+1j*m)*exp(1j*theta)
    return x, b, data
```

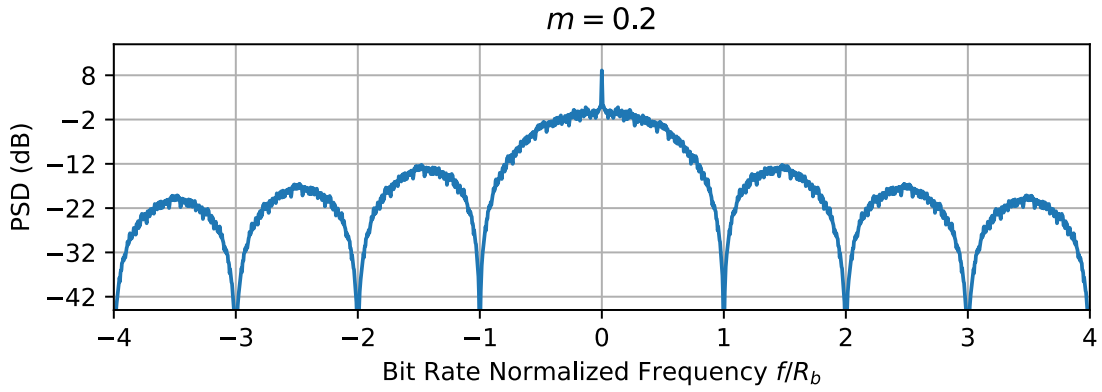
2.3 Binary PSK Modulator Output

By running just the modulator function, `x, b, data = psk(m, Nbit, Ns, theta)` you can look at the power spectrum of the received signal.

[34]:
```
x_m0,b,data0 = psk(0,10000,16,0.0) # m= 0 so BPSK
x_m02,b,data02 = psk(0.2,10000,16,0.0) # m <> 0 so some carrier present
subplot(211)
psd(x_m0,2**12,16);
xlim([-4,4])
ylim([-45,15])
ylabel('PSD (dB)')
xlabel(r'Bit Rate Normalized Frequency $f/R_b$')
title(r'$m = 0$')
subplot(212)
psd(x_m02,2**12,16);
xlim([-4,4])
ylim([-45,15])
title(r'$m = 0.2$')
ylabel('PSD (dB)')
xlabel(r'Bit Rate Normalized Frequency $f/R_b$')
tight_layout()
```

```
/var/folders/nh/4pcjlwf17gd87xk2z4f449ph0000gn/T/ipykernel_83954/632695361.py:4:
MatplotlibDeprecationWarning: Passing the NFFT parameter of psd() positionally
is deprecated since Matplotlib 3.10; the parameter will become keyword-only in
3.12.
  psd(x_m0,2**12,16);
/var/folders/nh/4pcjlwf17gd87xk2z4f449ph0000gn/T/ipykernel_83954/632695361.py:11
: MatplotlibDeprecationWarning: Passing the NFFT parameter of psd() positionally
is deprecated since Matplotlib 3.10; the parameter will become keyword-only in
3.12.
  psd(x_m02,2**12,16);
```





[35]:
```
def psk_sim(SNR,m,theta,Nbits,timing):
    """
    Pe,Bit_errors,Bit_count,z = psk_sim(SNR,theta,Nbits,timing)

    Coherent PSK modem end-to-end simulation function
    ////////////////////////////////////////////////////////////////
           SNR = Set the received signal SNR in dB; for on-off keying
                 the SNR is defined interms of the average signal
                 power, e.g., 1/2 the peak symbol energy
             m = modulation index, m= 0 ==> BPSK
         theta = Rotate the complex baseband signal by theta radians
         Nbits = The number of bits simulated
        timing = Sets the sampler timing to give the minimum BEP;
                 find value using eyeplot_mark function
            Pe = Estimated bit error probability (BEP)
    Bit_errors = Number of bit errors found (good to obtain at least
                 100 errors over an AWGN channel
     Bit_count = The number of bits processed y bit_errors function;
                 If you create a 'top-level' function 'errors' and
                 'RecBits' can be used to combine results from multiple
                  runs
```

```
            z = Receiver decision statistic; use as input to eyeplot
                for further analysis, e.g., find a good value
                for 'timing'
    ////////////////////////////////////////////////////////////////
    Mark Wickert October 2014
    """
    Ns = 16 # Fix the number of samples per bit to 16

    # Generate PSK tx signal
    x,b,data = psk(m, Nbits, 16, theta)
    y = dc.cpx_awgn(x,SNR,16) # AWGN channel
    print(var(y))
    # Enter receiver
    # Coherent receiver (at complex baseband)
    #////////////////////////////////
    # Matched filter is a rectangle pulse
    b_REC = b/sum(b) # make filter unity gain at dc
    # Process complex BB signal through filter to form decision statistic
    z = signal.lfilter(b_REC,1,y)
    z_det = z.real
    # Decision logic
    d_hat = (sign(z_det[timing::16])+1)/2 # {0,1} logic levels
    # Error detect
    Bit_count, Bit_errors = dc.bit_errors(data,d_hat)
    Pe = Bit_errors/float(Bit_count)
    return Pe,Bit_errors,Bit_count,z
```

[36]:
```
Pe,Bit_errors,Bit_count,z = psk_sim(20,0.0,0*pi/180.,100000,15)
print(' Bit Count = %d' % Bit_count)
print('Bit Errors = %d' % Bit_errors)
print('        Pe = %6.3e' % Pe)
```

```
1.1598010190062908
 Bit Count = 100000
Bit Errors = 0
        Pe = 0.000e+00
```

Perform a quick theory check:

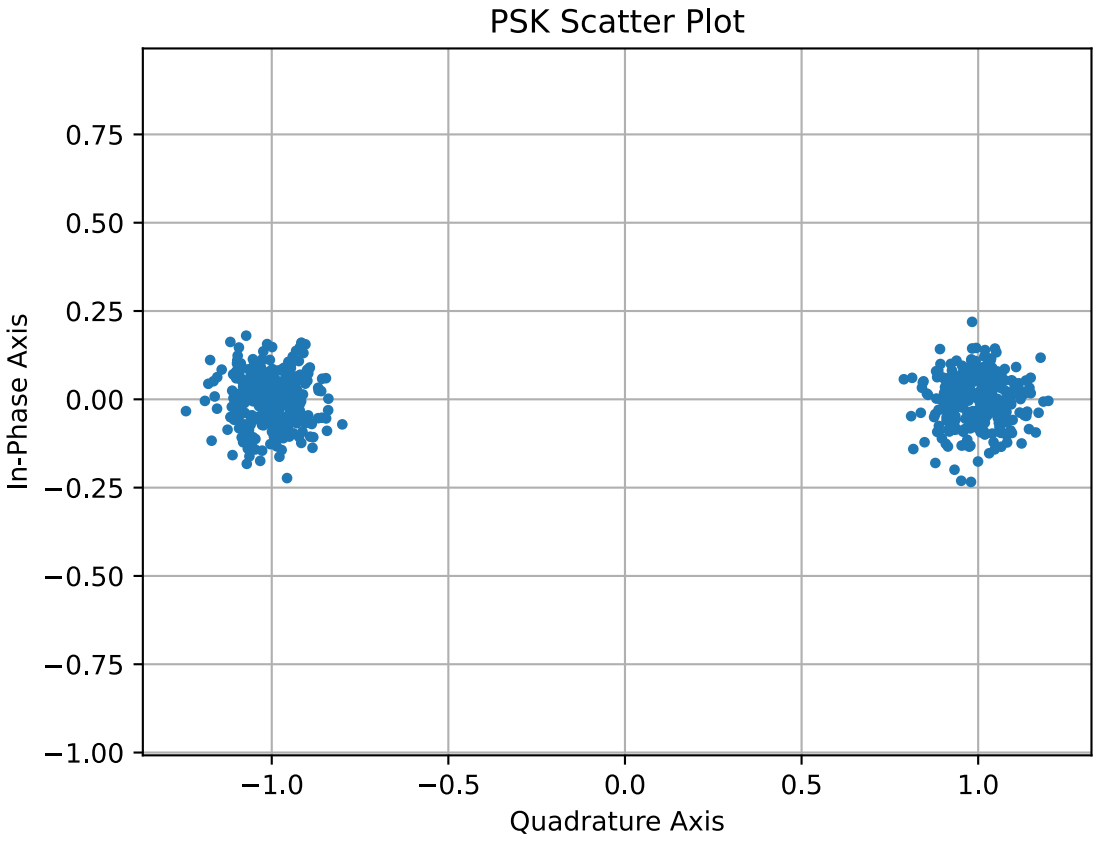
I know that $P_E = Q(\sqrt{2E_b/N_0})$. I set $E_b/N_0 = 10^{(E_b/N_0)_{\text{dB}}/10}$ so I can input dB values for E_b/N_0.

[37]:
```
print('Pe estim = %4.3e' % Q(sqrt(2*10**(8/10.))))
```

```
Pe estim = 1.909e-04
```

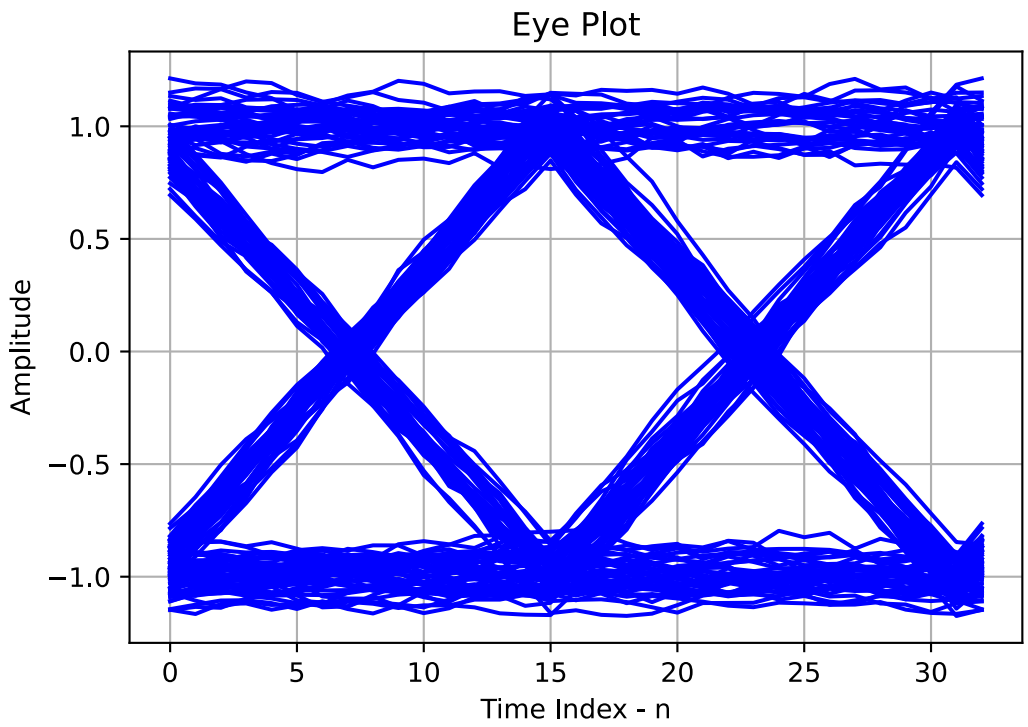
[38]:
```
plot(z[15:10000:16].real,z[15:10000:16].imag,'.')
grid()
title('PSK Scatter Plot')
xlabel('Quadrature Axis')
```

```
ylabel('In-Phase Axis')
axis('equal');
```



[39]:
```
dc.eye_plot(z.real[:5000],32,16)
```

[39]:
0



2.4 Binary FSK

[40]:
```
def fsk(Df, Nbit, Ns, theta):
    """
    x,b,data,x_ref = fsk(Df, Nbit, Ns, theta)

     Complex baseband binary FSK modulator
    ////////////////////////////////////////////////////////////////////
         Df = peak-to-peak frequency deviation normalized to bit rate
       Nbit = number of symbols processed
        Ns  = the number of samples per bit
      theta = Rotate the complex baseband signal by theta radians

          x = Complex baseband transmit signal vector
       data = Binary {0,1} data being sent; used for error detecting
      x_ref = Complex baseband reference signal used for coherent
              demodulation of FSK (does not contain data)
    ////////////////////////////////////////////////////////////////////
    Mark Wickert October 2014
    """
    A = 1
    y,b,data = dc.nrz_bits(Nbit,Ns)
    n = arange(0,len(y)) # time axis for generating FM
```

```
    x = A*exp(1j*2*pi*Df/2.*y*n/float(Ns))*exp(1j*theta)
    # reference for coherent correlation at receiver
    x_ref = A*exp(1j*2*pi*Df/2.*n/float(Ns))
    return x,b,data,x_ref
```

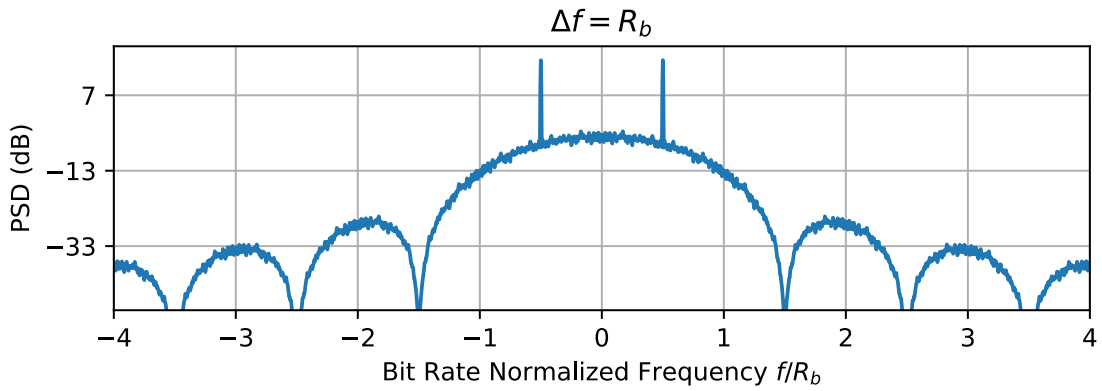
[41]:
```
# This function can be found in
# sk_dsp_comm.rtlsdr_helper
def discrim(x):
    """
    function disdata = discrim(x)
    where x is an angle modulated signal in complex baseband form.

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

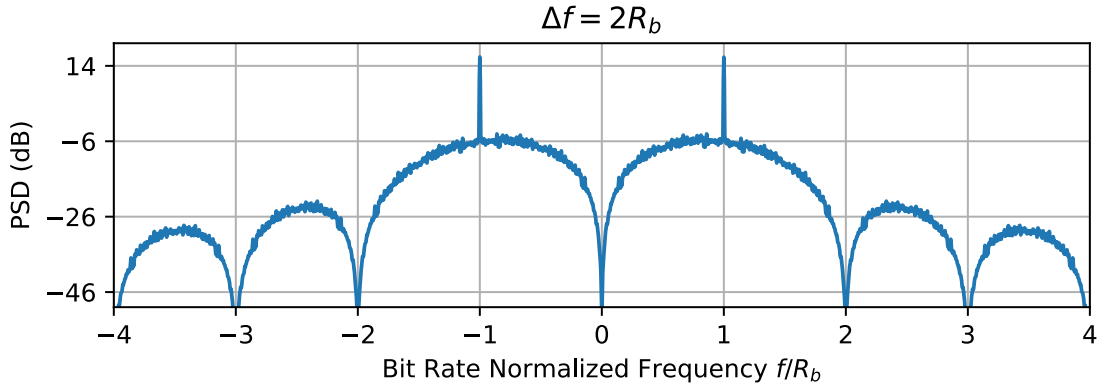
2.5 Binary FSK Modulator Output

By running just the modulator function, x, b, data = fsk(Df, Nbit, Ns, theta) you can look at the power spectrum of the received signal.

[42]:
```
x_D1,b,dataD1,xref = fsk(1.0,10000,16,0.0) # Df = 1.0
x_D2,b,dataD2,xref = fsk(2.0,10000,16,0.0) # Df = 2.0
subplot(211)
psd(x_D1,2**12,16);
xlim([-4,4])
ylim([-50,20])
title(r'$\Delta f = R_b$')
ylabel('PSD (dB)')
xlabel(r'Bit Rate Normalized Frequency $f/R_b$')
subplot(212)
psd(x_D2,2**12,16);
xlim([-4,4])
ylim([-50,20])
title(r'$\Delta f = 2R_b$')
ylabel('PSD (dB)')
xlabel(r'Bit Rate Normalized Frequency $f/R_b$')
tight_layout()
```

```
/var/folders/nh/4pcjlwf17gd87xk2z4f449ph0000gn/T/ipykernel_83954/3416923653.py:4
: MatplotlibDeprecationWarning: Passing the NFFT parameter of psd() positionally
is deprecated since Matplotlib 3.10; the parameter will become keyword-only in
3.12.
  psd(x_D1,2**12,16);
/var/folders/nh/4pcjlwf17gd87xk2z4f449ph0000gn/T/ipykernel_83954/3416923653.py:1
1: MatplotlibDeprecationWarning: Passing the NFFT parameter of psd()
positionally is deprecated since Matplotlib 3.10; the parameter will become
keyword-only in 3.12.
  psd(x_D2,2**12,16);
```





FSK Simulation Top level

[43]:
```
def fsk_sim(SNR,Df,Nbits,timing,rec_type):
    """
    [Pe,errors,N_RecBits,z] = fsk_sim(SNR,Df,Nbits,timing,rec_type)

    Coherent and Noncoherent FSK modem end-to-end simulation function
    ////////////////////////////////////////////////////////////////////
         SNR = Set the received signal SNR in dB; for on-off keying the
               SNR is defined interms of the average signal power, e.g.,
               1/2 the peak symbol energy
```

```
        Df = Peak-to-peak frquency deviation normalized by the bit rate
     Nbits = The number of bits simulated
    timing = Sets the sampler timing to give the minimum BEP; find value
             using eyeplot_mark function
  rec_type = Receiver type: 'coherent', bpf_env_det', or 'lim_discrim'

        Pe = Estimated bit error probability (BEP)
    errors = Number of bit errors found (good to obtain at least 100
             errors over an AWGN channel
   RecBits = The number of bits processed y bit_errors function; If you
             create a 'top-level' function 'errors' and 'RecBits' can be
             used to combine results from multiple runs
         z = Receiver decision statistic; use as input to eyeplot for
             further analysis, e.g., find a good value for 'timing'
////////////////////////////////////////////////////////////////////////
Mark Wickert October 2014
Rework for Python October 2016-2018
"""
Ns = 16 # Fix the number of samples per bit to 16

# Generate FSK tx signal
x,b,data,x_ref = fsk(Df,Nbits, Ns, 0)
y = dc.cpx_awgn(x,SNR,Ns) # AWGN channel

# Enter receiver
if rec_type.lower() == 'coherent':
    # Correlator intgrator via rectangle pulse filter
    b_REC = b/float(Ns) # make filter unity gain at dc
                        # correlation with s1(t)
    r1 = signal.lfilter(b_REC,1,y*x_ref)
    # correlation with s2(t)
    r2 = signal.lfilter(b_REC,1,y*conj(x_ref))
    # Form decision statistic
    z = real(r1 - r2);
    # Decision logic
    d_hat = (sign(z[timing::Ns])+1)/2 # {0,1} logic levels
    # Error detect
    Bit_count, Bit_errors = dc.bit_errors(data,d_hat)
    Pe = Bit_errors/float(Bit_count)
    ###################################
elif rec_type.lower() == 'bpf_env_det':
    # BPF Envelope detector receiver
    #//////////////////////////////
    # Design the receive filter using fdatool written m-file function
    b_win = signal.firwin(33,2*1.5/16) # FIR lpf with hamming window
    # Process complex BB signal through filters via complex frequency
    # translation to +/-
```

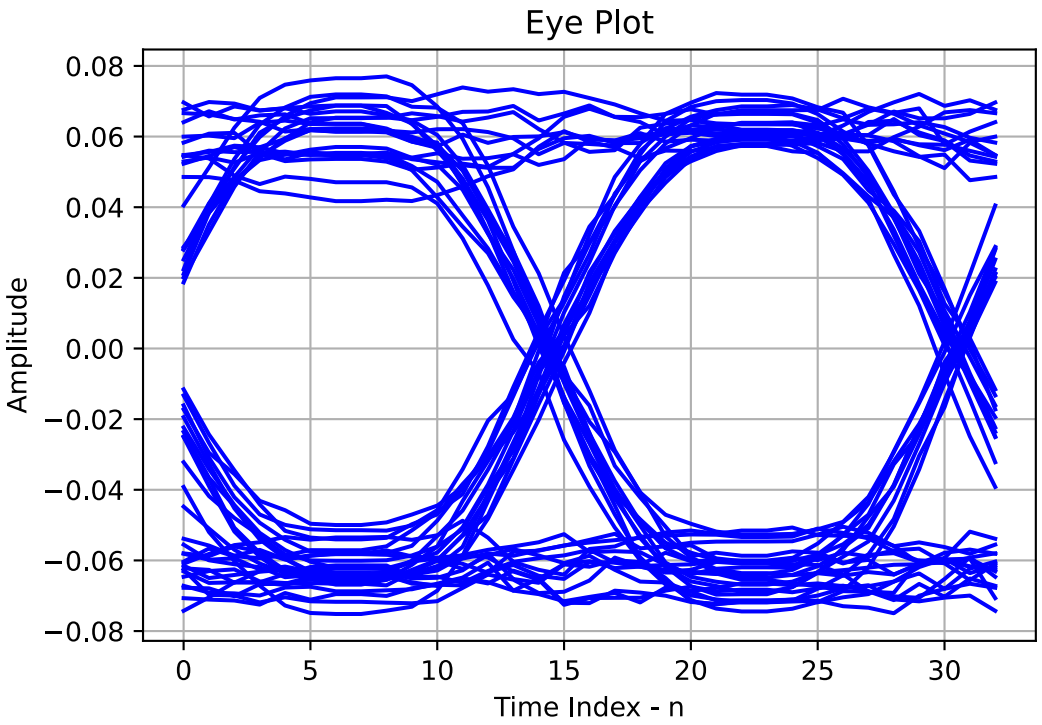
```
        n = arange(len(x))
        r1 = signal.lfilter(b_win,1,y*exp(-1j*2*pi*n*(Df/2)/Ns));
        r2 = signal.lfilter(b_win,1,y*exp(1j*2*pi*n*(Df/2)/Ns));
        # Form decision statistic
        z = abs(r1)-abs(r2);
        # Decision logic
        d_hat = (sign(z[timing::Ns])+1)/2; # {0,1} logic levels
        # Error detect
        Bit_count, Bit_errors = dc.bit_errors(data,d_hat)
        Pe = Bit_errors/float(Bit_count)
        ###################################
    elif rec_type.lower() == 'lim_discrim':
        # Limiter discriminator receiver
        #//////////////////////////////////
        # Design the receive filter using fdatool written m-file function
        b_win = signal.firwin(33,2*1.5/16)
        # Process complex BB signal through receive filter
        #r1 = lfilter(Hd2.numerator,1,y);
        r1 = signal.lfilter(b_win,1,y);
        # Form decision statistic via limiter discriminator
        z = discrim(r1);
        # Decision logic
        d_hat = (sign(z[timing::Ns])+1)/2; # {0,1} logic levels
        # Error detect
        Bit_count, Bit_errors = dc.bit_errors(data,d_hat)
        Pe = Bit_errors/float(Bit_count)
    else:
        print('Unknown receiver type')
    return Pe,Bit_errors,Bit_count,z
```

[44]:
```
#                   SNR,Df,Nbits,timing,rec_type = 'coherent', 'lim_discrim',␣
 ↪'bpf_env_det'
Pe,Bit_errors,Bit_count,z = fsk_sim(20,1.0,100,16,'coherent')
print(' Bit Count = %d' % Bit_count)
print('Bit Errors = %d' % Bit_errors)
print('        Pe = %6.3e' % Pe)
```

```
 Bit Count = 99
Bit Errors = 0
        Pe = 0.000e+00
```

[45]:
```
dc.eye_plot(z,32,9)
```

[45]:
0



3 A Simple Simulation Case Study

In this mini case study I consider BPSK at complex baseband, or quivalently binary anipodal signalling using rectangular pulse shapes. The receiver matched filter, however, will be replaced by a 3rd-order Butterworth filter. The 3dB frequency of the Butterworth frequency will be a free variable. The `signal.butter()` function from the SciPy signal module will be used to design the filter.

A custom version of the `psk_sim()` function will be created to contain the top level software simulation function.

[46]:
```
def psk_sim_butter3(SNR,f3Tb,theta,Nbits,timing):
    """
    Pe,Bit_errors,Bit_count,z = psk_sim(SNR,theta,Nbits,timing)

    Coherent PSK modem end-to-end simulation function
    ////////////////////////////////////////////////////////////////
           SNR = Set the received signal SNR in dB; for on-off keying
                 the SNR is defined interms of the average signal
                 power, e.g., 1/2 the peak symbol energy
         fsTb = modulation index, m= 0 ==> BPSK
        theta = Rotate the complex baseband signal by theta radians
        Nbits = The number of bits simulated
```

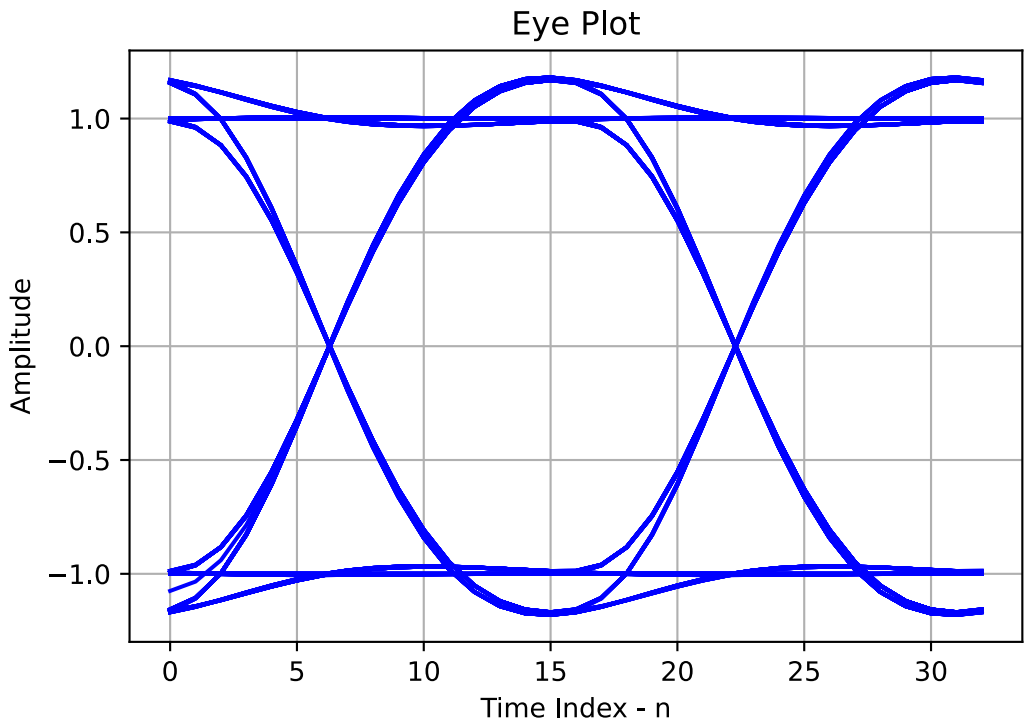
```
        timing = Sets the sampler timing to give the minimum BEP;
                 find value using eyeplot_mark function
            Pe = Estimated bit error probability (BEP)
    Bit_errors = Number of bit errors found (good to obtain at least
                 100 errors over an AWGN channel
     Bit_count = The number of bits processed y bit_errors function;
                 If you create a 'top-level' function 'errors' and
                 'RecBits' can be used to combine results from multiple
                  runs
             z = Receiver decision statistic; use as input to eyeplot
                 for further analysis, e.g., find a good value
                 for 'timing'
    ////////////////////////////////////////////////////////////////
    Mark Wickert October 2014
    """
    Ns = 16 # Fix the number of samples per bit to 16
    m = 0 # fix the modulation index to produce BPSK

    # Generate PSK tx signal
    x,b,data = psk(m, Nbits, 16, theta)
    y = dc.cpx_awgn(x,SNR,16) # AWGN channel

    # Enter receiver
    # Coherent receiver (at complex baseband)
    #////////////////////////////////
    # Matched filter is approximated with a 3rd-order Butterworth
    # filter
    #b_REC = b/sum(b) # make filter unity gain at dc
    b_butter3,a_butter3 = signal.butter(3,2*f3Tb/Ns)
    # Process complex BB signal through filter to form decision statistic
    z = signal.lfilter(b_butter3,a_butter3,y)
    z_det = z.real
    # Decision logic
    d_hat = (sign(z_det[timing::16])+1)/2 # {0,1} logic levels
    # Error detect
    Bit_count, Bit_errors = dc.bit_errors(data,d_hat)
    Pe = Bit_errors/float(Bit_count)
    return Pe,Bit_errors,Bit_count,z
```

[47]:
```
Pe,Bit_errors,Bit_count,z = psk_sim_butter3(100,0.8,0,1000,15)
print(' Bit Count = %d' % Bit_count)
print('Bit Errors = %d' % Bit_errors)
print('        Pe = %6.3e' % Pe)
```

```
 Bit Count = 1000
Bit Errors = 0
        Pe = 0.000e+00
```

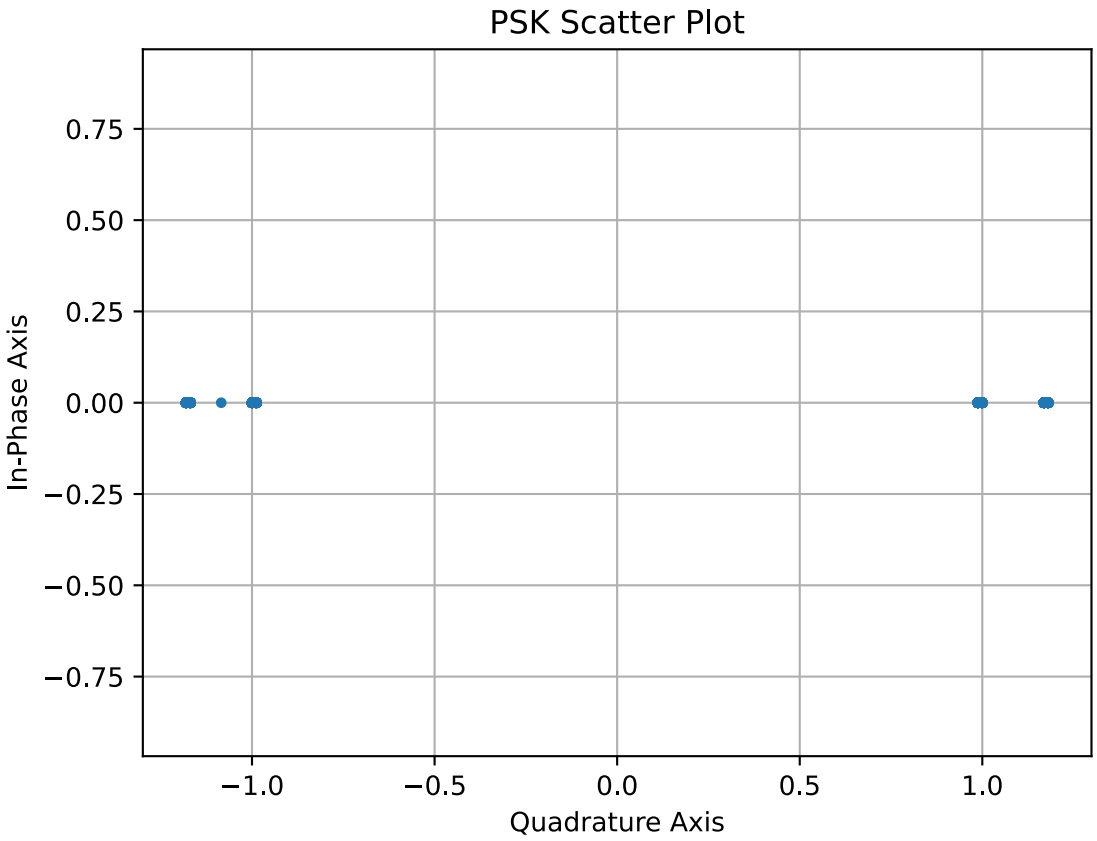
[48]:
```
dc.eye_plot(z.real[:5000],32,16)
```

[48]:
0



3.1 Timing Adjustment

[49]:
```
timing = 15
plot(z[timing::16].real,z[timing::16].imag,'.')
grid()
title('PSK Scatter Plot')
xlabel('Quadrature Axis')
ylabel('In-Phase Axis')
axis('equal');
```



3.2 Some BEP Experiments for f3Tb = 0.8

[50]:
```
Pe,Bit_errors,Bit_count,z = psk_sim_butter3(3,0.85,0,10000,14)
print(' Bit Count = %d' % Bit_count)
print('Bit Errors = %d' % Bit_errors)
print('        Pe = %6.3e' % Pe)
```

```
 Bit Count = 10000
Bit Errors = 540
        Pe = 5.400e-02
```

[51]:
```
Pe,Bit_errors,Bit_count,z = psk_sim_butter3(5,0.85,0,10000,14)
print(' Bit Count = %d' % Bit_count)
print('Bit Errors = %d' % Bit_errors)
print('        Pe = %6.3e' % Pe)
```

```
 Bit Count = 10000
Bit Errors = 211
        Pe = 2.110e-02
```

[38]:
```
Pe,Bit_errors,Bit_count,z = psk_sim_butter3(7,0.85,0,100000,14)
print(' Bit Count = %d' % Bit_count)
print('Bit Errors = %d' % Bit_errors)
print('        Pe = %6.3e' % Pe)
```

```
kmax =  0, taumax = 0
 Bit Count = 100000
Bit Errors = 577
        Pe = 5.770e-03
```

[182]:
```
Pe,Bit_errors,Bit_count,z = psk_sim_butter3(9,0.85,0,500000,14)
print(' Bit Count = %d' % Bit_count)
print('Bit Errors = %d' % Bit_errors)
print('        Pe = %6.3e' % Pe)
```

```
kmax =  0, taumax = 0
 Bit Count = 500000
Bit Errors = 423
        Pe = 8.460e-04
```

[191]:
```
Pe,Bit_errors,Bit_count,z = psk_sim_butter3(10,0.85,0,500000,14)
print(' Bit Count = %d' % Bit_count)
print('Bit Errors = %d' % Bit_errors)
print('        Pe = %6.3e' % Pe)
```

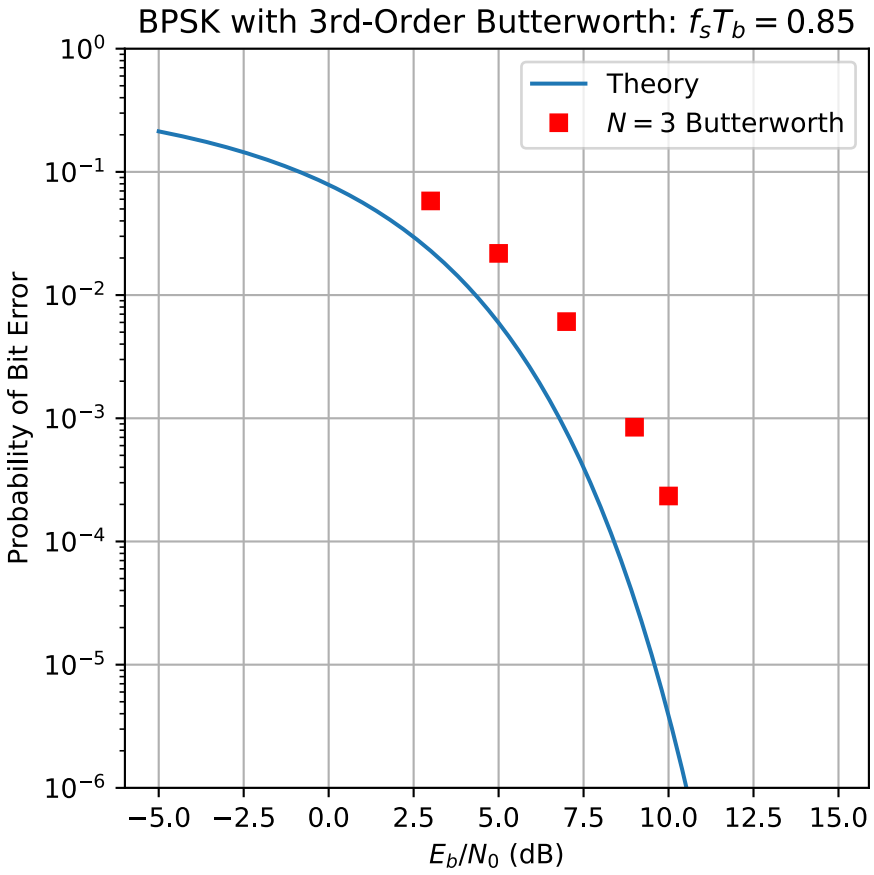
```
kmax =  0, taumax = 0
 Bit Count = 500000
Bit Errors = 117
        Pe = 2.340e-04
```

3.3 Simulation BEP Plot Compared with BPSK Ideal Theory

When plotting experimental BEP curves it is frequently best to collect the simulation points and then plot the curve. The probability points are usually collected over a long simulation run. These *valuable* points may be added to the curve as they become available. You might for example dedicate one computer just for running your simulation points and a second computer for writing documentation and developing new models.

[52]:
```
EBN0_dB = arange(-5,15,.1)
Pe_thy = dc.q_fctn(sqrt(2*10**(EBN0_dB/10.)))
EBN0_dB_exp = array([3,5,7,9,10])
Pe_exp = array([5.810e-02,2.180e-02,6.090e-03,8.460e-04,2.340e-04])
figure(figsize=(5,5))
semilogy(EBN0_dB,Pe_thy)
semilogy(EBN0_dB_exp,Pe_exp,'rs')
grid()
xlabel(r'$E_b/N_0$ (dB)')
ylabel('Probability of Bit Error')
title(r'BPSK with 3rd-Order Butterworth: $f_sT_b = 0.85$')
```

```
legend((('Theory'),(r'$N = 3$ Butterworth')),loc='best')
ylim([1e-6,1]);
```



4 Nyquist Pulse Shaping

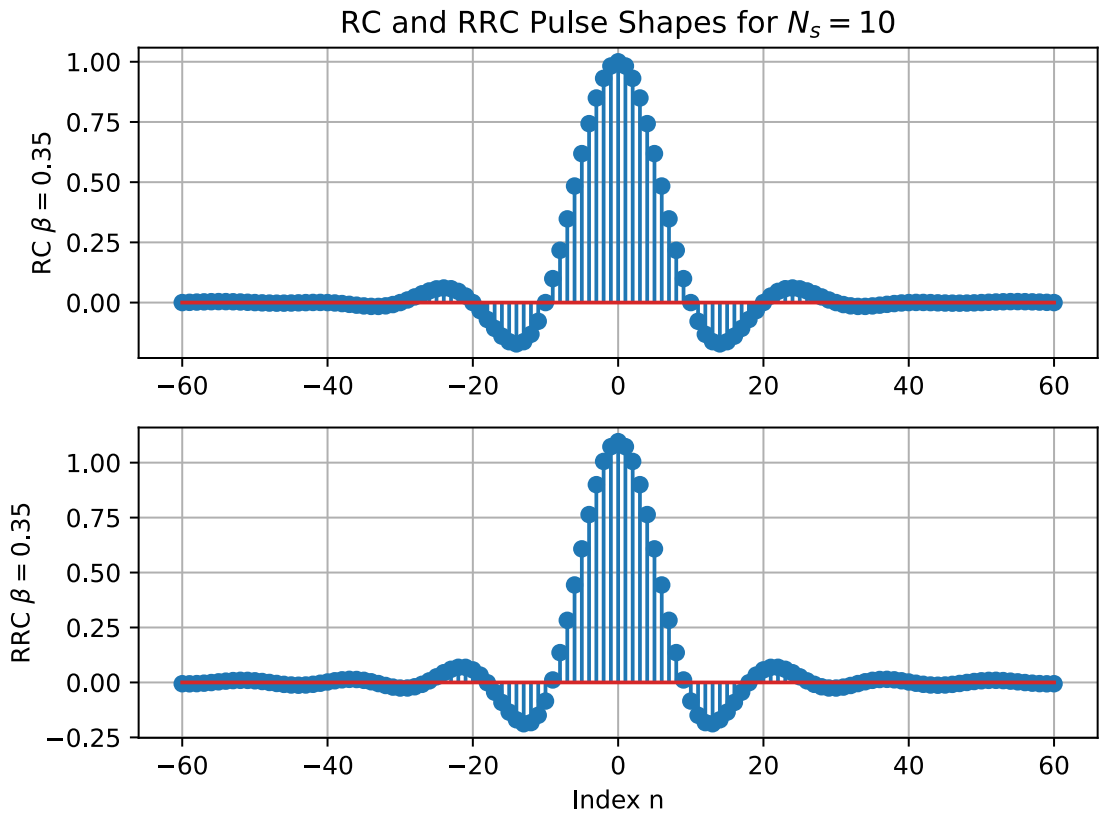
Nyquist pulse shaping, both the raised cosine and the square root raised cosine functions can be found in `sigsys.py` and `digital_com.py`.

```
"""
Definition:  sigsys.rc_imp(Ns, alpha, M=6)
Docstring:
A truncated raised cosine pulse used in digital communications.

The pulse shaping factor 0< alpha < 1 is required as well as the
truncation factor M which sets the pulse duration to be 2*M*Tsymbol.

Parameters
----------
Ns : number of samples per symbol
```

```
alpha : excess bandwidth factor on (0, 1), e.g., 0.35
M : equals RC one-sided symbol truncation factor

Returns
-------
b : ndarray containing the pulse shape

Notes
-----
The pulse shape b is typically used as the FIR filter coefficients
when forming a pulse shaped digital communications waveform.

Examples
--------
"""
>>> # ten samples per symbol and alpha = 0.35
>>> b = rc_imp(10,0.35)
>>> n = arange(-10*6,10*6+1)
>>> stem(n,b)

"""
Definition:  sigsys.sqrt_rc_imp(Ns, alpha, M=6)
Docstring:
A truncated square root raised cosine pulse used in digital communications.

The pulse shaping factor 0< alpha < 1 is required as well as the
truncation factor M which sets the pulse duration to be 2*M*Tsymbol.


Parameters
----------
Ns : number of samples per symbol
alpha : excess bandwidth factor on (0, 1), e.g., 0.35
M : equals RC one-sided symbol truncation factor

Returns
-------
b : ndarray containing the pulse shape

Notes
-----
The pulse shape b is typically used as the FIR filter coefficients
when forming a pulse shaped digital communications waveform. When
square root raised cosine (SRC) pulse is used generate Tx signals and
at the receiver used as a matched filter (receiver FIR filter), the
received signal is now raised cosine shaped, this having zero
intersymbol interference and the optimum removal of additive white
noise if present at the receiver input.
```

```
    Examples
    --------
    """
    >>> # ten samples per symbol and alpha = 0.35
    >>> b = sqrt_rc_imp(10,0.35)
    >>> n = arange(-10*6,10*6+1)
    >>> stem(n,b)
```

[53]:
```
subplot(211)
b_rc = ss.rc_imp(10,0.35)
n = arange(-10*6,10*6+1)
stem(n,b_rc)
grid()
title(r'RC and RRC Pulse Shapes for $N_s = 10$')
ylabel(r'RC $\beta = 0.35$');
subplot(212)
b_src = ss.sqrt_rc_imp(10,0.35)
n = arange(-10*6,10*6+1)
stem(n,b_src)
grid()
xlabel('Index n')
ylabel(r'RRC $\beta = 0.35$');
tight_layout()
```

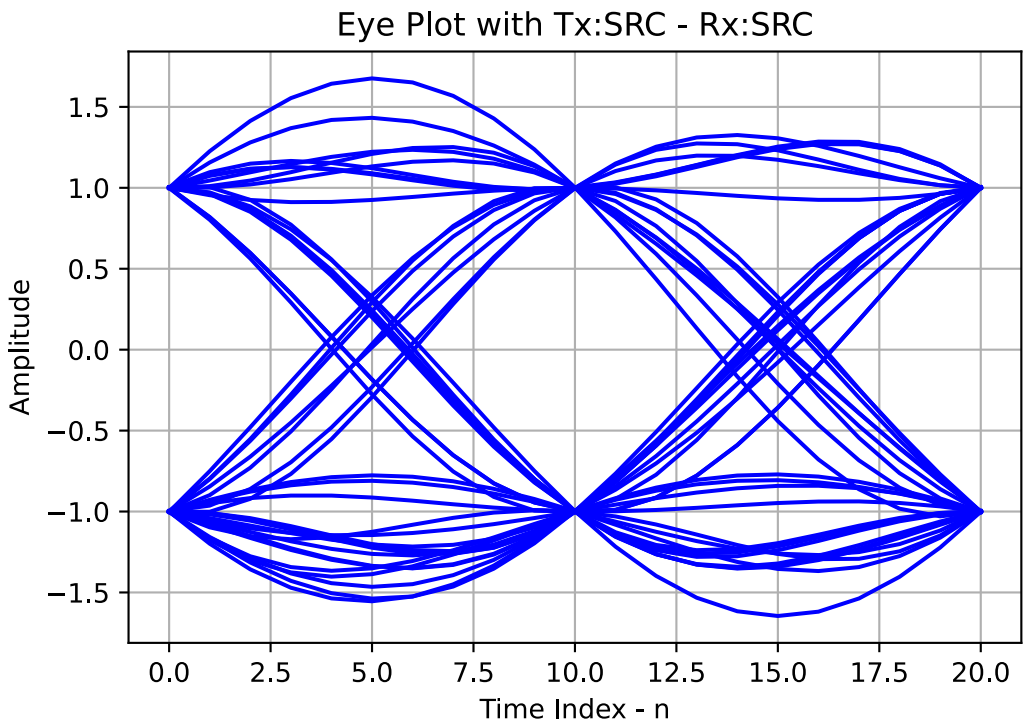


4.1 Simple Example Using `dc.NRZ_bits()`

[54]:
```
Ns = 10 # samples per bit
x,b,data = dc.nrz_bits(100, Ns,'src',alpha=0.35)
y = dc.cpx_awgn(x,100,Ns) # Eb/N0 = 100 dB
z = signal.lfilter(b,1,y)
```

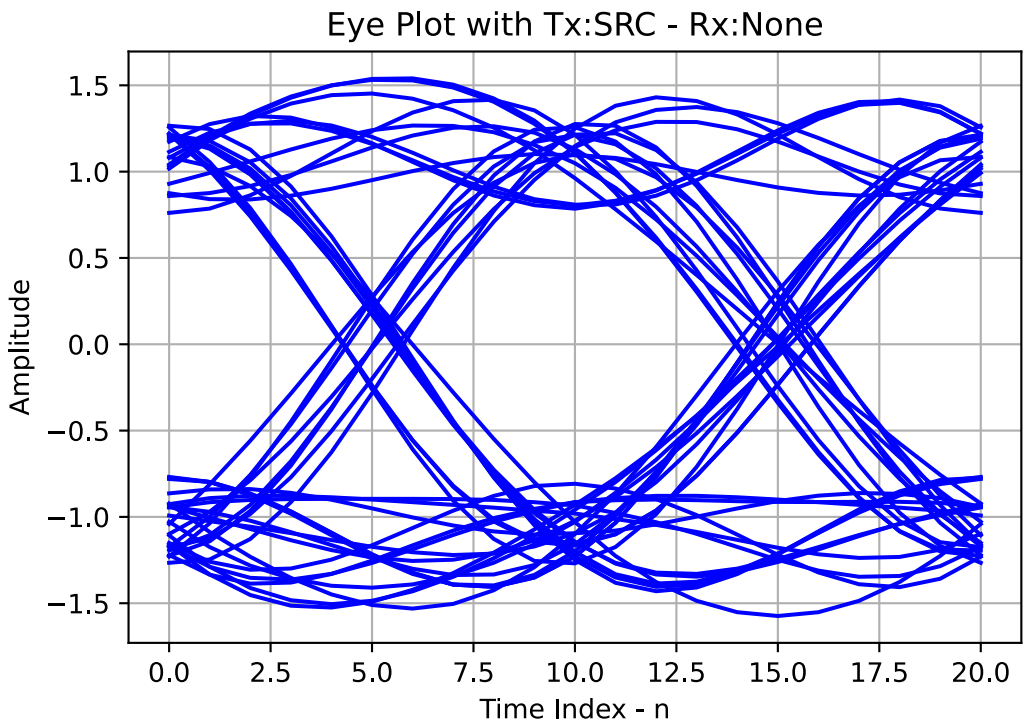
[55]:
```
dc.eye_plot(z.real,2*10,12*10)
title(r'Eye Plot with Tx:SRC - Rx:SRC')
```

[55]:
```
Text(0.5, 1.0, 'Eye Plot with Tx:SRC - Rx:SRC')
```



[56]:
```
dc.eye_plot(y.real,2*10,12*10)
title(r'Eye Plot with Tx:SRC - Rx:None')
```

[56]:
```
Text(0.5, 1.0, 'Eye Plot with Tx:SRC - Rx:None')
```



5 Flat Fading

With Rayleigh flat fading the fades are slow relative to the bit rate, and frequency independent, so each received bit has a multiplicative Rayleigh random variable. The Rayleigh weights are assumed to be independent from bit-to-bit. We can write the decision statistic, z_k, as:

$$z_k = R \cdot d_k + N \tag{2}$$

where R has a Rayleigh PDF with mean $r_m = 1$ and N is AWGN having variance σ_n^2 such that $\bar{E}_b/N_0 = 1/\sigma_n^2$.

The random subpackage of `numpy` has Rayleigh random variates (so does `scipy.stats`):

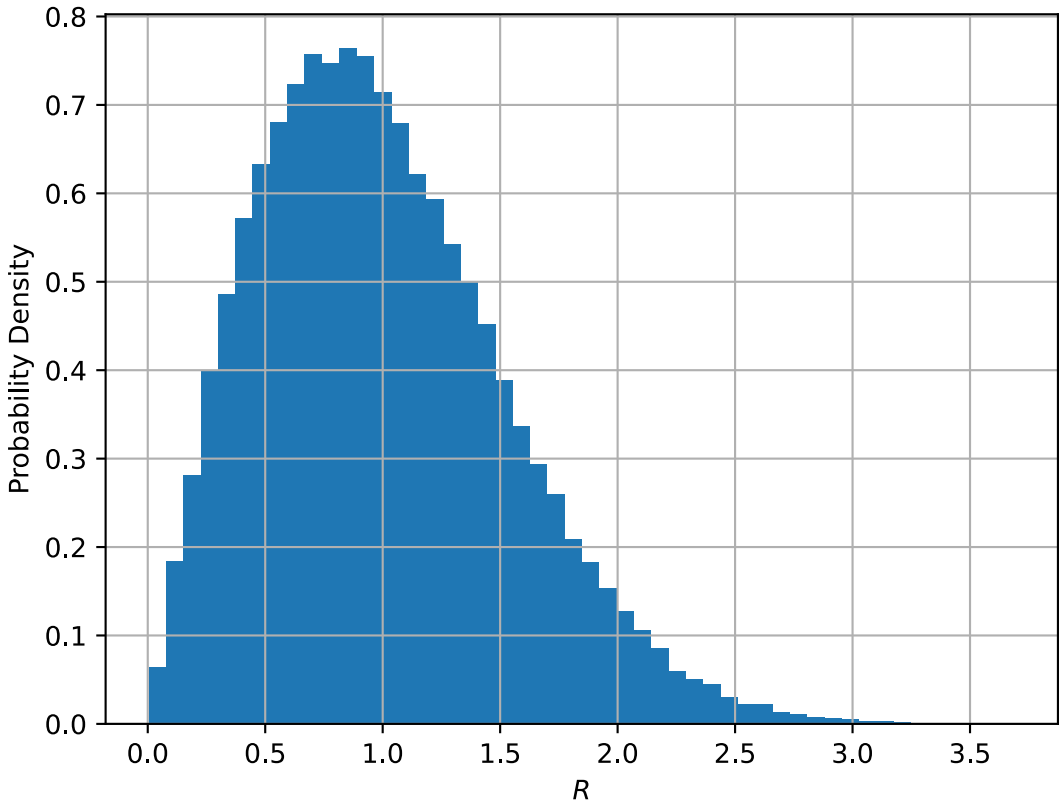
```
r = random.rayleigh(10)
```

[57]:
```
r = random.rayleigh(sqrt(2/pi),10000)
(mean(r),mean(r**2))
```

[57]: (0.988092191703537, 1.2462288315526)

[58]:
```
r = random.rayleigh(sqrt(2/pi),100000)
hist(r,bins=50,density=True);
ylabel(r'Probability Density')
```

```
xlabel(r'$R$')
grid();
```



[59]:
```
def psk_rayleigh_sim(SNR,m,theta,Nbits,timing):
    """
    Pe,Bit_errors,Bit_count,z = psk_rayleigh_sim(SNR,theta,Nbits,timing)

    Coherent PSK modem end-to-end simulation function
    ////////////////////////////////////////////////////////////////
           SNR = Set the received signal SNR in dB; for on-off keying
                 the SNR is defined interms of the average signal
                 power, e.g., 1/2 the peak symbol energy
             m = modulation index, m= 0 ==> BPSK
         theta = Rotate the complex baseband signal by theta radians
         Nbits = The number of bits simulated
        timing = Sets the sampler timing to give the minimum BEP;
                 find value using eyeplot_mark function
            Pe = Estimated bit error probability (BEP)
    Bit_errors = Number of bit errors found (good to obtain at least
                 100 errors over an AWGN channel
```

```
     Bit_count = The number of bits processed y bit_errors function;
                 If you create a 'top-level' function 'errors' and
                 'RecBits' can be used to combine results from multiple
                  runs
             z = Receiver decision statistic; use as input to eyeplot
                 for further analysis, e.g., find a good value
                 for 'timing'
    ////////////////////////////////////////////////////////////////
    Mark Wickert November 2018
    """
    Ns = 16 # Fix the number of samples per bit to 1

    # Generate PSK tx signal
    x,b,data = psk(m, Nbits, Ns, theta)
    # Rayleigh flat fading
    r = random.rayleigh(sqrt(2/pi),len(data))
    r_up = signal.lfilter(ones(Ns),1,ss.upsample(r,Ns))
    y = dc.cpx_awgn(x*r_up,SNR,Ns) # AWGN channel that incorporates fading

    # Enter receiver
    # Coherent receiver (at complex baseband)
    #////////////////////////////////
    # Matched filter is a rectangle pulse
    b_REC = b/sum(b) # make filter unity gain at dc
    # Process complex BB signal through filter to form decision statistic
    z = signal.lfilter(b_REC,1,y)
    z_det = z.real
    # Decision logic
    d_hat = (sign(z_det[timing::16])+1)/2 # {0,1} logic levels
    # Error detect
    Bit_count, Bit_errors = dc.bit_errors(data,d_hat)
    Pe = Bit_errors/float(Bit_count)
    return Pe,Bit_errors,Bit_count,z,r
```

[60]:
```
Pe,Bit_errors,Bit_count,z,r = psk_rayleigh_sim(25,0.0,0*pi/180.,200000,15)
print(' Bit Count = %d' % Bit_count)
print('Bit Errors = %d' % Bit_errors)
print('        Pe = %6.3e' % Pe)
```

```
 Bit Count = 200000
Bit Errors = 144
        Pe = 7.200e-04
```

Perform a quick theory check:

I know that for flat fading

$$P_E = \frac{1}{2}\left(1 - \sqrt{\frac{\bar{E}_b/N_0}{1+\bar{E}_b/N_0}}\right)$$

I set $E_b/N_0 = 10^{(E_b/N_0)_{\text{dB}}/10}$ so I can input dB values for E_b/N_0.

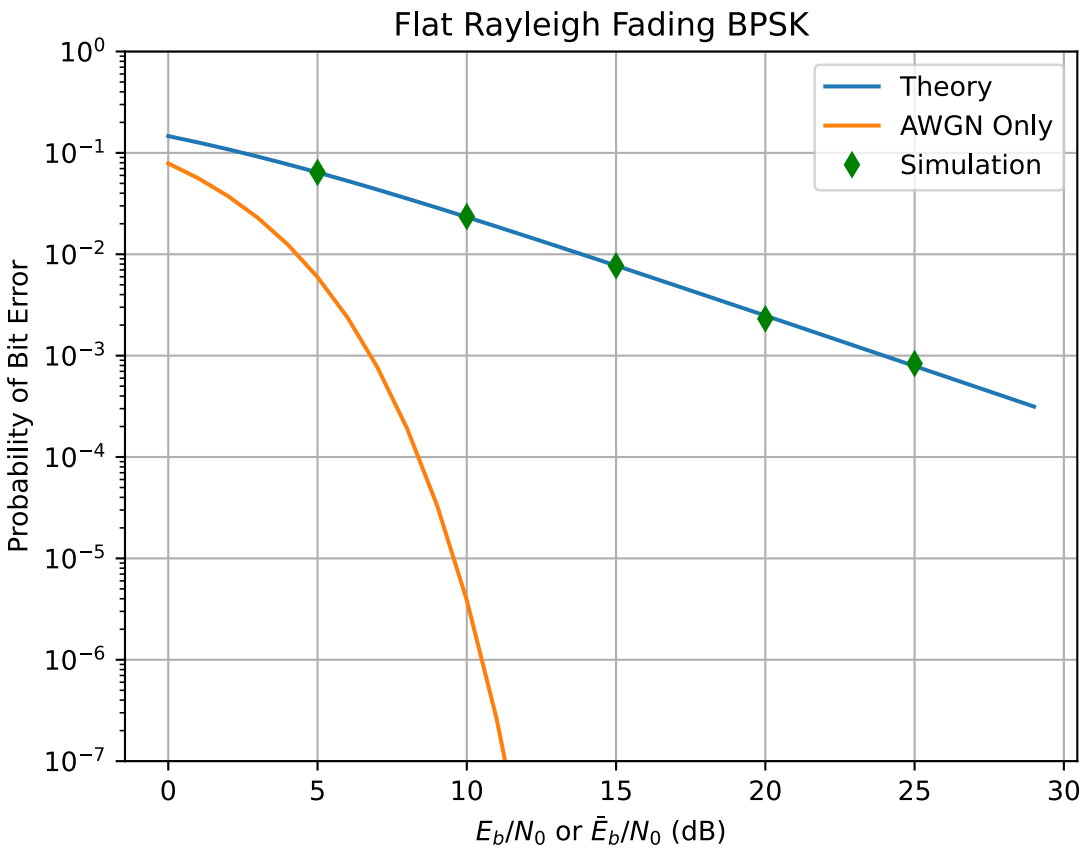
[61]:
```
def Pe_flatfade(EbNo_bar):
    """
    Pe_bar for Rayleigh flat fading BPSK
    Mark Wickert November 2018
    """
    Pe_bar = 1/2*(1 - sqrt(10**(EbN0_bar/10)/(1 + 10**(EbN0_bar/10))))
    return Pe_bar
```

[62]:
```
EbN0_bar = 25
print('Pe estim = %4.3e' % (Pe_flatfade(EbN0_bar),))
```

```
Pe estim = 7.887e-04
```

5.0.1 Theory versus Simulation BEP

[63]:
```
EbN0_bar = arange(0,30,1)
Pe_fade_thy = Pe_flatfade(EbN0_bar)
EbN0_bar_sim = [5,10,15,20,25]
Pe_fade_sim = [6.420e-02,2.352e-2,7.700e-03,2.310e-03,8.350e-04]
Pe_thy = Q(sqrt(2*10**(EbN0_bar/10)))
semilogy(EbN0_bar,Pe_fade_thy)
semilogy(EbN0_bar,Pe_thy)
semilogy(EbN0_bar_sim,Pe_fade_sim,'gd')
ylim([1e-7,1])
title(r'Flat Rayleigh Fading BPSK')
ylabel(r'Probability of Bit Error')
xlabel(r'$E_b/N_0$ or $\bar{E}_b/N_0$ (dB)')
legend((r'Theory',r'AWGN Only',r'Simulation'),loc='best')
grid();
```



6 Equalizers

Moved to a separate Jupyter notebook.

[ ]: