ECE5630_Equalizers

June 30, 2025

1 Equalizers

In this notebook we consider the equalizer Examples 5.11 and 5.12 found in notes Chapter 5. The original MATLAB code in converted to Python and is embedded into the cells of this notebook. Later this code may be re-worked and placed into a Python code module.

- Zero forcing
- Linear decision-directed (DD) MMSE
- Non-linear with decision feedback (DFE) under DD MMSE

[1]:
```
%pylab inline
#%matplotlib qt
import sk_dsp_comm.sigsys as ss
import sk_dsp_comm.digitalcom as dc
import scipy.signal as signal
from IPython.display import Audio, display
from IPython.display import Image, SVG

%config InlineBackend.figure_formats=['svg'] # SVG inline viewing
#%config InlineBackend.figure_formats=['pdf'] # render pdf figs for LaTeX
```

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

Basic Linear Algebra using `numpy.linalg` and `scipy.linalg`

[3]:
```
A = array([[1,2,3],[4,5,6],[-7,8,-9]])
A
```

[3]:
```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [-7,  8, -9]])
```

[8]:
```
Ainv = linalg.inv(A)
Ainv
```

[8]:
```
array([[-0.96875   ,  0.4375    , -0.03125   ],
       [-0.0625    ,  0.125     ,  0.0625    ],
       [ 0.69791667, -0.22916667, -0.03125   ]])
```

[9]:
```
A @ Ainv
```

[9]:
```
array([[ 1.00000000e+00, -5.55111512e-17, -1.38777878e-17],
       [-4.44089210e-16,  1.00000000e+00, -2.77555756e-17],
       [ 2.22044605e-16,  1.66533454e-16,  1.00000000e+00]])
```

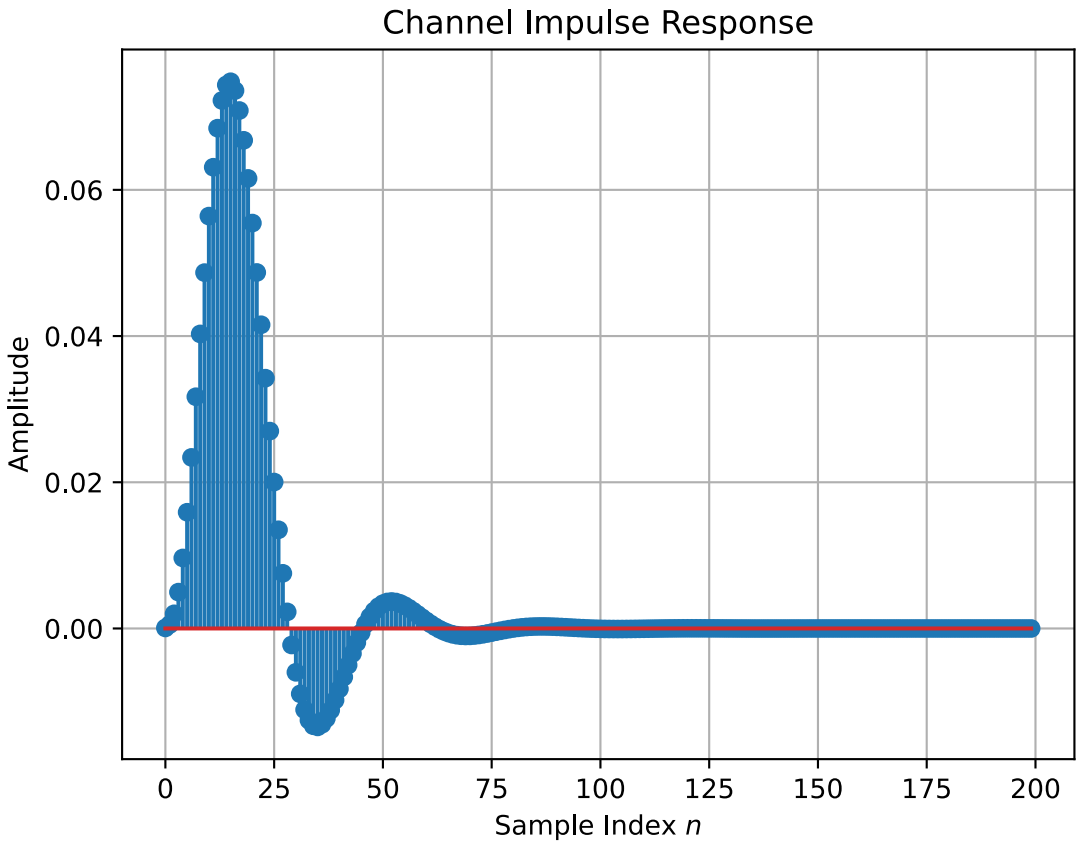
[10]:
```
dot(A,Ainv)
```

[10]:
```
array([[ 1.00000000e+00, -5.55111512e-17, -1.38777878e-17],
       [-4.44089210e-16,  1.00000000e+00, -2.77555756e-17],
       [ 2.22044605e-16,  1.66533454e-16,  1.00000000e+00]])
```

1.1 Channel Models

The initial channel model is that of a Butterworth IIR filter. A bit/symbol-spaced FIR channel model used in Set #5 homework problems is given below. To use this channel with oversampling of `Ns` samples per bit we can upsample these values.

[3]:
```
# A 4th-order Butterworh channel distortion model
# Butterworth lowpass filter.
# Set fc = 0.5 times the bit rate.
Ns = 16
b_chan,a_chan = signal.butter(4,2 * 0.5/Ns)

# Z&T 9.46 Modified Channel
# b_chan = ss.upsample(array([0,0,0.02,0,0,-0.11,0.1,-0.6,1.0,-0.4,0.5,0.
 ↪18,0,0,0.01,0,0]),16)
# a_chan = array([1])

nd = arange(200)
xd = ss.dimpulse(nd)
xc = signal.lfilter(b_chan,a_chan,xd)
stem(nd,xc)
title(r'Channel Impulse Response')
ylabel('Amplitude')
xlabel(r'Sample Index $n$')
grid();
```



1.2 Zero Forcing

The function `shaped_psk_ZFEQ_sim()` implements a baseband BPSK simulation that incorporates a transmitter with SRC pulse shaping, a bandlimiting channel filter, a receiver matched filter, a symbol-spaced zero forcing equalizer, and a probability of bit error calculator. The zero forcing equalizer employs $2N+1$ taps where presently $N = 5$. The sampling rate is set to $N_s = 16$ samples per bit. The equalizer is designed to force the ISI to zero at $\pm N$ bits either side of the center bit. The impulse response is obtained by sampling the pulse response from transmitter through the channel filter, through the receiver matched filter at the bit rate. The sampling of the pulse response is chosen to include the maximum or peak of the end-to-end (less the equalizer filter of course) pulse response $p_c[n] = \mathcal{F}^{-1}\{P_{SRC}(e^{j2\pi f/f_s})H_{chan}(e^{j2\pi f/f_s})P_{SRC}(e^{j2\pi f/f_s})\}$, plus N samples either side of it.

[4]:
```
import scipy.linalg as linalg
def  shaped_psk_ZFEQ_sim(SNR,N_bits,timing,pulse,beta=0.35):
    """
     Pe,errors,N_RecBits,z,b_eq =␣
 ↪shaped_psk_ZFEQ_sim(SNR,N_bits,timing,pulse,beta)

     Coherent Binary BPSK modem ZF-EQ end-to-end simulation function
```

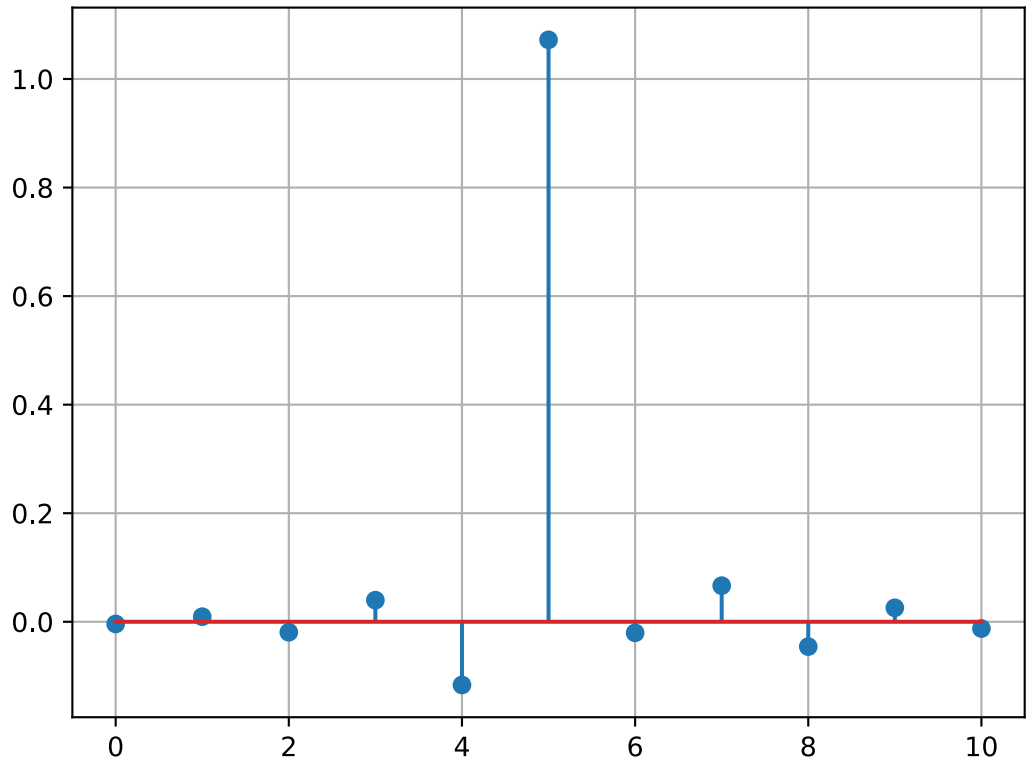
```
////////////////////////////////////////////////////////////////////////
      SNR = Set the received signal SNR in dB; for on-off keying the SNR
            is defined interms of the average signal power, e.g., 1/2 the
            peak symbol energy
   N_bits = The number of bits simulated
   timing = Sets the sampler timing to give the minimum BEP; find value
            using eyeplot function

       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_mark for
            further analysis, e.g., find a good value for 'timing'
////////////////////////////////////////////////////////////////////////

Mark Wickert November 2010, code to Python November 2016
"""

Ns = 16 # Fix the number of samples per bit to 16
N = 5 # Number of equalizer taps is 2*N+1

# Generate PSK tx signal
x,b,data = dc.nrz_bits(N_bits, 16, pulse, beta)

# Introduce channel distortion via a 4th-order
# Butterworth lowpass filter.
# Set fc = 0.5 times the bit rate.
b_chan,a_chan = signal.butter(4,2 * 0.5/Ns)
x = signal.lfilter(b_chan,a_chan,x)

# AWGN channel
y = dc.cpx_awgn(x,SNR,16)

# Enter receiver

# Coherent receiver (at complex baseband)
#////////////////////////////////
# Matched filter is designed by the Tx function as vector b
# Process complex BB signal through filter to form decision statistic
z = signal.lfilter(b,1,y)

#////////////////////////////////
# Design a zero forcing equalizer using side information
b_eq,p_c = zero_force(b*Ns,b_chan,a_chan,Ns,N) # Ntaps = 2N+1 = 2*2+1 = 11
```

```
    # Apply equalizer as an upsampled filter to better see timing needs
    # If line is commented out, no equalizer is applied
#     z = signal.lfilter(dc.upsample(b_eq,Ns),1,z)

    z_det = real(z)
    # Decision logic
    d_hat = (sign(z_det[timing::16])+1)/2 # {0,1} logic levels
    # Error detect
    N_RecBits,errors= dc.bit_errors(data,d_hat)
    Pe = errors/N_RecBits
    return Pe,errors,N_RecBits,z,p_c,b_eq

#////////////////////////////////////////////////////////////////////////
def zero_force(tx_pulse,b_chan,a_chan,Ns,N):
    """
    Aopt = zero_force(tx_pulse,b_chan,a_chan,Ns,N)
    Zero Force Function

    Mark Wickert November 2011
    Recoded to Python November 2016
    ////////////////////////////////////////////////////////////////////////
    """

    Npre = N*Ns; Npost = 4*N*Ns;
    # Zero pad either side of pulse
    p = hstack([zeros(Npre), tx_pulse, zeros(Npost)])
    p = signal.lfilter(b_chan,a_chan,p) # Filter twice to get end-to-end
    p = signal.lfilter(tx_pulse/Ns,1,p)#/Ns # (Tx/Rx pulse together) response
    # Find the max and index of max
    p_max,I_max = max(p),argmax(p)
    p_vec = p[I_max-Ns*2*N:I_max+Ns*2*N+1:Ns] # Symbol spaced channel samples
    p_c = p[I_max-Ns*3*N:I_max+Ns*4*N+1:Ns] # Extended range symbol spaced␣
 ↪samples
    N_tap = 2*N+1
    Pc = zeros((N_tap,N_tap))
    for k in range(2*N+1):
        # Fill up the Pc matrix with channel pulse response samples
        Pc[:,k] = p_vec[2*N-(k):2*N-(k)+N_tap]

    Peq = hstack((zeros(N), [1.0], zeros(N))).T
    Aopt = linalg.solve(Pc,Peq)
    return Aopt, p_c
```

[5]:
```
Pe,errors,N_RecBits,z,p_c,b_eq = shaped_psk_ZFEQ_sim(50,1000,15,'src')
print('Pe Est = %1.2e' % Pe)
print('B_RecBits = %d, errors = %d' % (N_RecBits,errors))
```

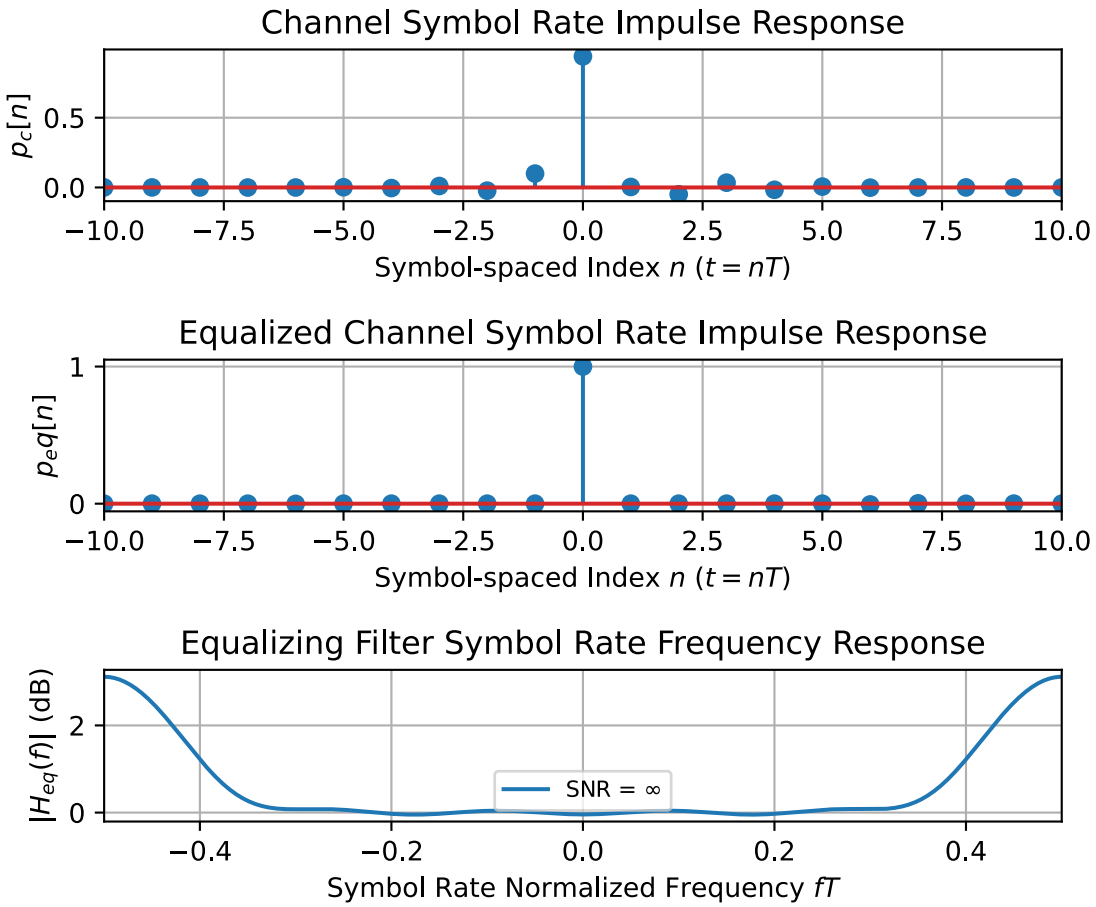
```
Pe Est = 0.00e+00
B_RecBits = 988, errors = 0
```

[6]:
```
stem(b_eq)
grid();
```



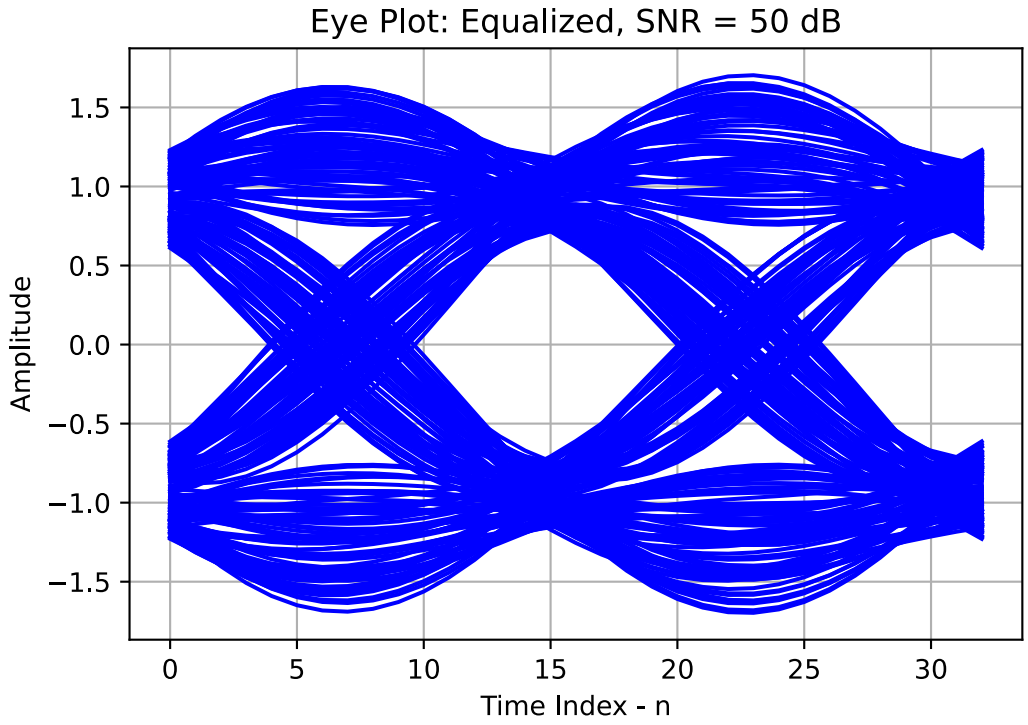
[7]:
```
N = 5 # number of eq taps is 2*N+1
figure(figsize=(6,5))
subplot(311)
n = arange(len(p_c))
n -= argmax(p_c)
stem(n,p_c)
xlim([-2*N,2*N])
title(r'Channel Symbol Rate Impulse Response')
ylabel(r'$p_c[n]$')
xlabel(r'Symbol-spaced Index $n$ ($t=nT$)')
grid();
subplot(312)
p_eq = signal.lfilter(b_eq,1,p_c)
n = arange(len(p_eq))
n -= argmax(p_eq)
stem(n,p_eq)
```

```
xlim([-2*N,2*N])
title(r'Equalized Channel Symbol Rate Impulse Response')
ylabel(r'$p_eq[n]$')
xlabel(r'Symbol-spaced Index $n$ ($t=nT$)')
grid();
subplot(313)
f = arange(-0.5,0.5,1/2048)
w, B_eq = signal.freqz(b_eq,1,2*pi*f)
plot(f,20*log10(abs(B_eq)))
xlim([-.5,.5])
title(r'Equalizing Filter Symbol Rate Frequency Response')
ylabel(r'$|H_{eq}(f)|$ (dB)')
xlabel(r'Symbol Rate Normalized Frequency $fT$')
legend((r'SNR = $\infty$',),loc='best',fontsize='small')
grid()
tight_layout()
```



[8]:
```
dc.eye_plot(real(z[400*16:]),2*16,0)
title(r'Eye Plot: Equalized, SNR = 50 dB')
```

[8]:
```
Text(0.5, 1.0, 'Eye Plot: Equalized, SNR = 50 dB')
```



1.3 Decision Directed (DD) MMSE with LMS Adaptation

The function `shaped_psk_DDEQ_sim()` implements a baseband BPSK simulation that incorporates a transmitter with SRC pulse shaping, a bandlimiting channel filter, a receiver matched filter, a decision-directed (DD) symbol-spaced minimum mean-squared error (MMSE) equalizer or simply DD-MMSE equalizer, and a probability of bit error calculator. The DD-MMSE equalizer employs $2N + 1$ taps where presently $N = 5$. The sampling rate is set to $N_s = 16$ samples per bit. The equalizer is designed minimize the mean-square error relative to a DD criterion. The impulse response is obtained by sampling the pulse response from transmitter through the channel filter, through the receiver matched filter at the bit rate. The sampling of the pulse response is chosen to include the maximum or peak of the end-to-end (less the equalizer filter of course) pulse response $p_c[n] = \mathcal{F}^{-1}\{P_{SRC}(e^{j2\pi f/f_s})H_{chan}(e^{j2\pi f/f_s})P_{SRC}(e^{j2\pi f/f_s})\}$, plus N samples either side of it.

[9]:
```
def shaped_psk_DDEQ_sim(SNR,N_bits,timing,DD_i,pulse,beta=0.35):
    """
    [Pe,errors,N_RecBits,z,DD_o] =␣
↪shaped_psk_DDEQ_sim(SNR,Nbits,timing,beta,shape,DD_i)
```

```
 Coherent Binary BPSK modem ZF-EQ 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
   N_bits = The number of bits simulated
   timing = Sets the sampler timing to give the minimum BEP; find value
            using eyeplot function
     DD_i = structure variable hold DD-LMS parameters:
            DD_i.N  = number of single-sided taps; total taps = 2*N+1
            DD_i.mu = LMS algorithm convergence parameter, around 1e-3
            DD_i.Nstart = Start BEP measuring with a delay due to LMS
                          not convergenced yet, 500 to 1000 good
                          but check MSE plot

       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_mark for
            further analysis, e.g., find a good value for 'timing'
     DD_o = DD-LMS output variables:
            DD_o.z_eq = Equalizer output at symbol spacing
            DD_o.d_hat = Hard decision data
            DD_o.e = Error sequence (complex)
            DD_o.Aopt = Tap weigh vector at end of simulation (complex)
////////////////////////////////////////////////////////////////////////

Mark Wickert November 2010
Translated to Python November 2016
"""

Ns = 16; # Fix the number of samples per bit to 16

# Generate PSK tx signal
x,b,data = dc.nrz_bits(N_bits, 16, pulse, beta)

# Introduce channel distortion via a 4th-order
# Butterworth lowpass filter.
# Set fc = 0.5 times the bit rate.
b_chan,a_chan = signal.butter(4,2 * 0.5/Ns)
x = signal.lfilter(b_chan,a_chan,x)

# AWGN channel
y = dc.cpx_awgn(x,SNR,16)
```

```
    # Enter receiver

    # Coherent receiver (at complex baseband)
    #////////////////////////////////
    # Matched filter is designed by the Tx function as vector b
    # Process complex BB signal through filter to form decision statistic
    z = signal.lfilter(b,1,y)

    #////////////////////////////////
    # Design a MMSE equalizer using a DD-LMS
    DD_o = DD_LMS(z[timing::16],Ns,DD_i.N,DD_i.mu)
    # Apply as an upsampled filter to better see timing needs
    z = signal.lfilter(dc.upsample(DD_o.Aopt,Ns),1,z)

    z_det = real(z)
    # Decision logic
    d_hat = (sign(z_det[timing::16])+1)/2 # {0,1} logic levels
    # Error detect
    N_RecBits,errors= dc.bit_errors(data[DD_i.Nstart:],d_hat[DD_i.Nstart:])
    Pe = errors/N_RecBits
    return Pe,errors,N_RecBits,z,DD_o

#////////////////////////////////////////////////////////////////////////
def DD_LMS(y,Ns,N,mu):
    """
    [z_eq,e,Aopt] = DD_LMS(y,Ns,N,mu)
    DD LMS Function

    Mark Wickert November 2011
    Converted to Python November 2016
    ////////////////////////////////////////////////////////////////////////
    """
    Nsim = len(y)
    DD_A = 1.0
    constellation = array([1, -1])*DD_A; # BPSK complex baseband constellation
    a = 1 # 1/sqrt(2) for QPSK;
    constellation = constellation*a

    z_eq = zeros(len(y),dtype=complex128)
    d_hat = zeros(len(y),dtype=complex128)
    e = zeros(len(y),dtype=complex128)
    Aopt = hstack((zeros(N), [1], zeros(N)))
    y_states = zeros(2*N+1)

    # Begin the LMS simulation sample-by-sample outer loop
    for k in range(Nsim):
        # Update input state vector
```

```
        y_states = hstack(([y[k]], y_states[:-1]))

        # Apply equalizer FIR filter sample-by-sample
        z_eq[k] = dot(Aopt,y_states)

        # Form the hard decisions using the filter outputs
        DD_error = z_eq[k] - constellation
        min_error,index = min(abs(DD_error)), argmin(abs(DD_error))
        d_hat[k] = constellation[index] # recovered data in {-1,1} format
        #d_hat.I(k) = fix(real(z_hat(k))/a)
        #d_hat.Q(k) = fix(imag(z_hat(k))/a)

        e[k] = d_hat[k] - z_eq[k]

        # Update the filter coefficients (LMS update eqn)
        Aopt = Aopt + mu*conj(y_states)*e[k]
    return DD_outputs(z_eq,d_hat,e,Aopt)

# Data structure for LMS inputs
class DD_inputs(object):
    """
    A class used used to hold DD input parameters
    """
    def __init__(self,N=5,mu=1e-3,Nstart=500):
        self.N = N # number of single-sided taps; total taps = 2*N+1
        self.mu = mu # LMS algorithm convergence parameter, around 1e-3
        self.Nstart = Nstart # Start BEP measuring with a delay due to LMS
                             # not convergenced yet, 500 to 1000 good
                             # but check MSE plot
    # This class has no methods

# Data structure to hold DD output variables
class DD_outputs(object):
    """
    A class used to hold DD output variables
    """
    def __init__(self,z_eq,d_hat,e,Aopt):
        self.z_eq = z_eq    # Equalizer output at symbol spacing
        self.d_hat = d_hat  # Hard decision data
        self.e = e          # Error sequence (complex)
        self.Aopt = Aopt    # Tap weight vector at end of simulation (complex)
    # This class has no methods
```

1.3.1 SNR = 50 dB

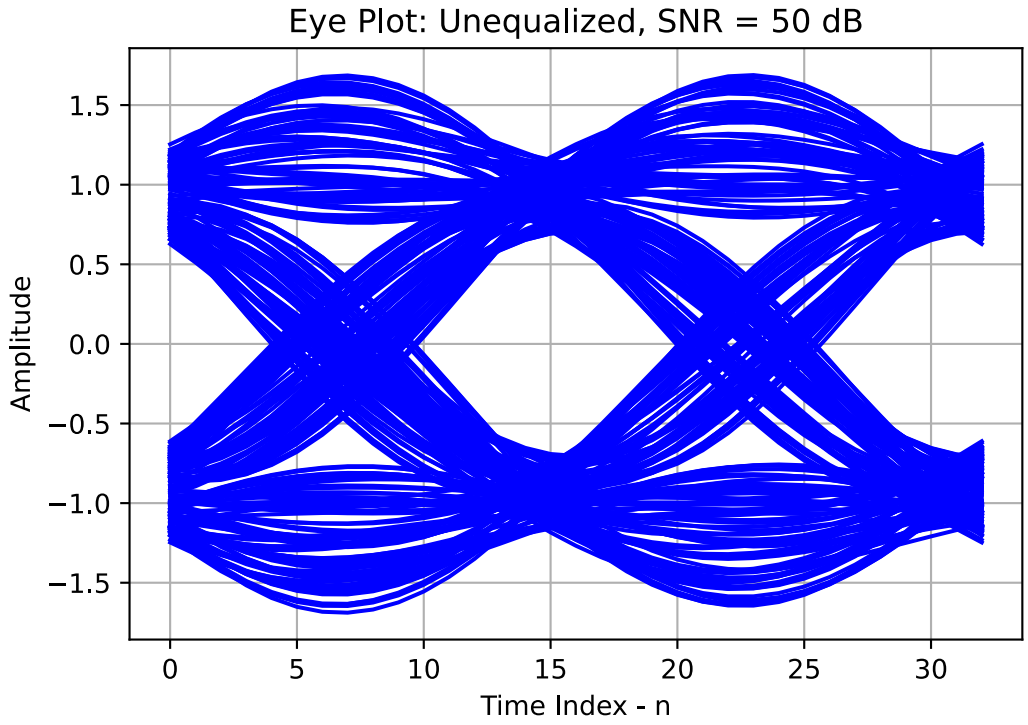
- Begin with $\mu = 0$ to see the eye plot without equalization
- Next turn on the equalizer by setting $\mu = 0.005$

[10]:
```
DD_i = DD_inputs(5,0.00,Nstart=0) # 2*5+1 taps, mu = 0.005
Pe,errors,N_RecBits,z,DD_o = shaped_psk_DDEQ_sim(50,1000,15,DD_i,'src',beta=0.
↪35)
print('Pe Est = %1.2e' % Pe)
print('B_RecBits = %d, errors = %d' % (N_RecBits,errors))
```

```
Pe Est = 0.00e+00
B_RecBits = 983, errors = 0
```

[11]:
```
dc.eye_plot(z.real[400*16:],2*16,0)
title(r'Eye Plot: Unequalized, SNR = 50 dB')
```

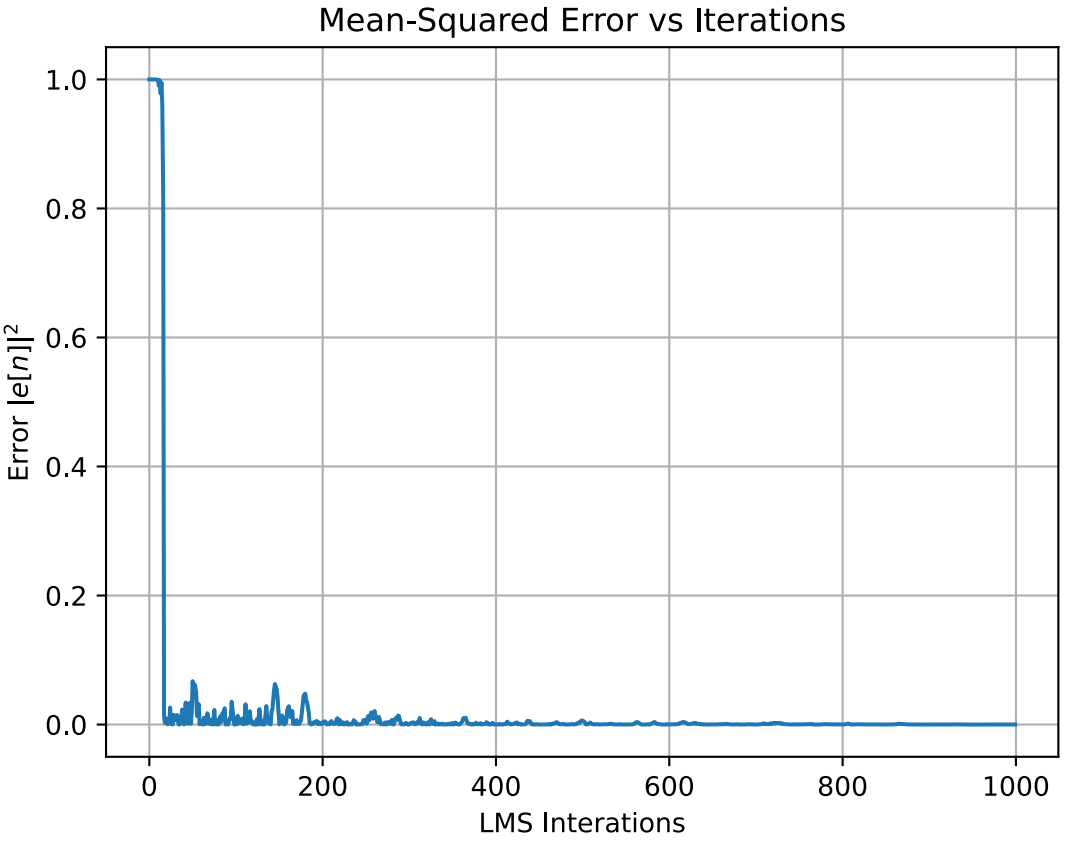
[11]:
```
Text(0.5, 1.0, 'Eye Plot: Unequalized, SNR = 50 dB')
```



[12]:
```
DD_i = DD_inputs(5,0.005,Nstart=0) # 2*5+1 taps, mu = 0.005
Pe,errors,N_RecBits,z,DD_o = shaped_psk_DDEQ_sim(50,1000,15,DD_i,'src',beta=0.
↪35)
print('Pe Est = %1.2e' % Pe)
print('B_RecBits = %d, errors = %d' % (N_RecBits,errors))
```

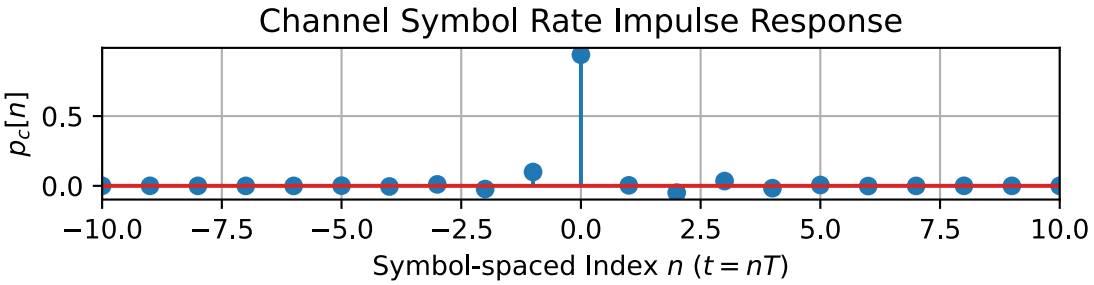
```
Pe Est = 0.00e+00
B_RecBits = 983, errors = 0
```

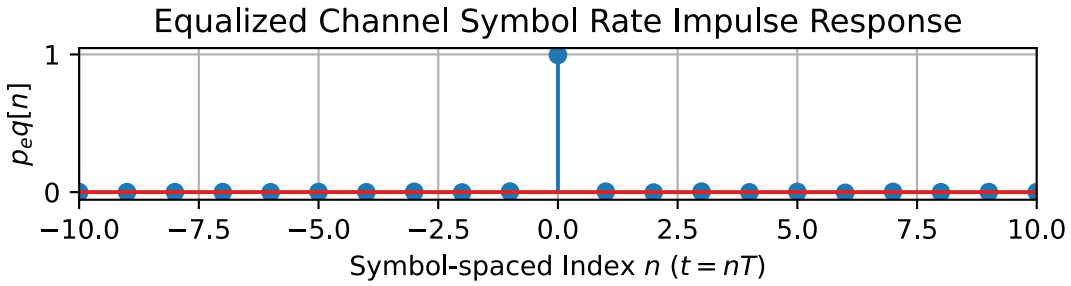
[13]:
```
plot(abs(DD_o.e)**2)
#ylim([0,.1])
title(r'Mean-Squared Error vs Iterations')
ylabel(r'Error $|e[n]|^2$')
xlabel(r'LMS Interations')
grid();
```

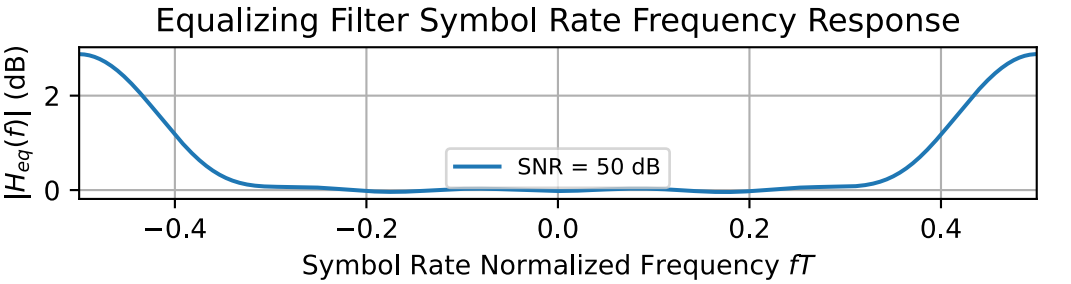


[14]:
```
N = 5 # number of eq taps is 2*N+1
figure(figsize=(6,5))
subplot(311)
n = arange(len(p_c))
n -= argmax(p_c)
stem(n,p_c)
xlim([-2*N,2*N])
title(r'Channel Symbol Rate Impulse Response')
ylabel(r'$p_c[n]$')
xlabel(r'Symbol-spaced Index $n$ ($t=nT$)')
grid();
subplot(312)
p_eq = signal.lfilter(DD_o.Aopt,1,p_c).real #show real part only
```

```
n = arange(len(p_eq))
n -= argmax(p_eq)
stem(n,p_eq)
xlim([-2*N,2*N])
title(r'Equalized Channel Symbol Rate Impulse Response')
ylabel(r'$p_eq[n]$')
xlabel(r'Symbol-spaced Index $n$ ($t=nT$)')
grid();
subplot(313)
f = arange(-0.5,0.5,1/2048)
w, B_eq = signal.freqz(DD_o.Aopt,1,2*pi*f)
plot(f,20*log10(abs(B_eq)))
xlim([-.5,.5])
title(r'Equalizing Filter Symbol Rate Frequency Response')
ylabel(r'$|H_{eq}(f)|$ (dB)')
xlabel(r'Symbol Rate Normalized Frequency $fT$')
legend((r'SNR = 50 dB',),loc='best',fontsize='small')
grid()
tight_layout()
```





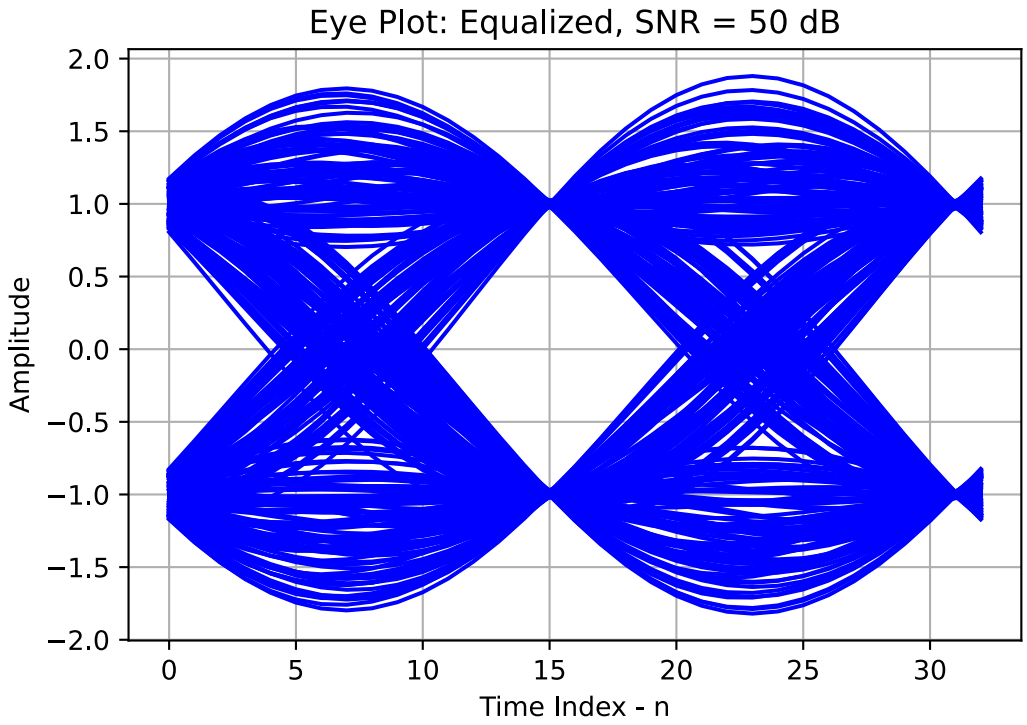


[345]:
```
DD_o.Aopt
```

[345]:
```
array([-0.00147916-2.29504243e-04j,  0.00595861-9.51842242e-05j,
       -0.01532085+1.91246732e-04j,  0.03462337-2.51842023e-04j,
       -0.11055305+3.46894429e-04j,  1.06675219-2.55361815e-04j,
       -0.01557311-3.76698632e-05j,  0.0607617 -5.02211337e-05j,
       -0.04068762-3.41210212e-05j,  0.02142057+2.53536826e-04j,
       -0.00932713+1.01959013e-04j])
```

[15]:
```
dc.eye_plot(z.real[400*16:],2*16,0)
title(r'Eye Plot: Equalized, SNR = 50 dB')
```

[15]:
```
Text(0.5, 1.0, 'Eye Plot: Equalized, SNR = 50 dB')
```



1.3.2 SNR = 10 dB

- Begin with $\mu = 0$ to see the eye plot without equalization and noise
- Next turn on the equalizer by setting $\mu = 0.005$

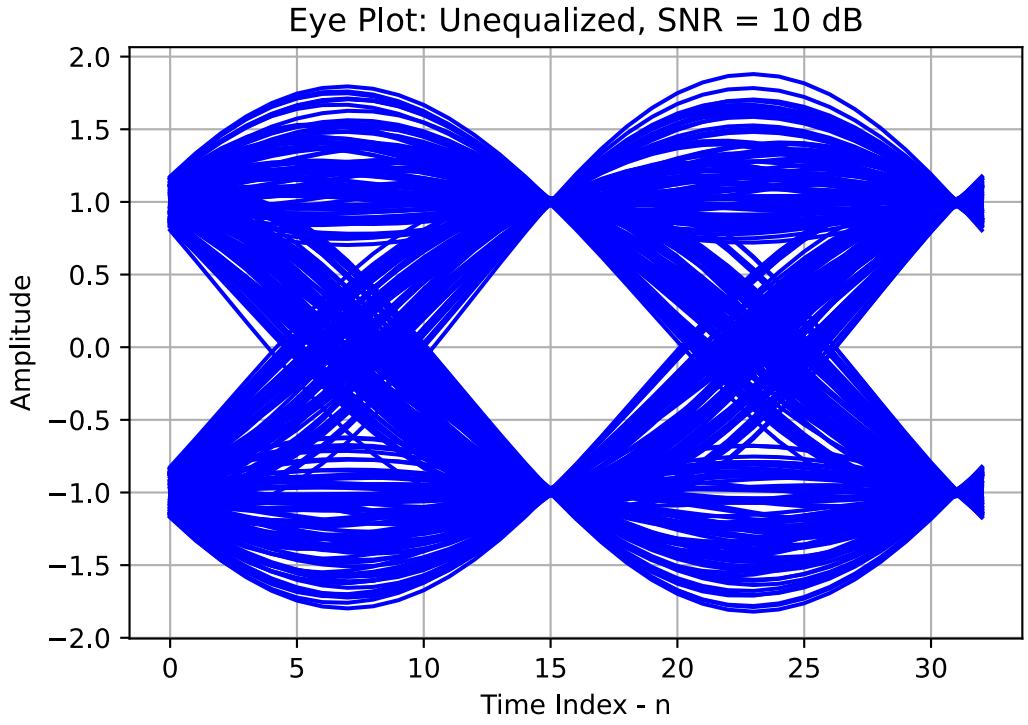
[34]:
```
DD_i = DD_inputs(5,0.00,Nstart=0) # 2*5+1 taps, mu = 0.005
Pe,errors,N_RecBits,z,DD_o = shaped_psk_DDEQ_sim(10,1000,15,DD_i,'src',beta=0.
↪35)
```

```
print('Pe Est = %1.2e' % Pe)
print('B_RecBits = %d, errors = %d' % (N_RecBits,errors))
```

```
Pe Est = 0.00e+00
B_RecBits = 983, errors = 0
```

[16]:
```
dc.eye_plot(z.real[400*16:],2*16,0)
title(r'Eye Plot: Unequalized, SNR = 10 dB')
```

[16]:
```
Text(0.5, 1.0, 'Eye Plot: Unequalized, SNR = 10 dB')
```

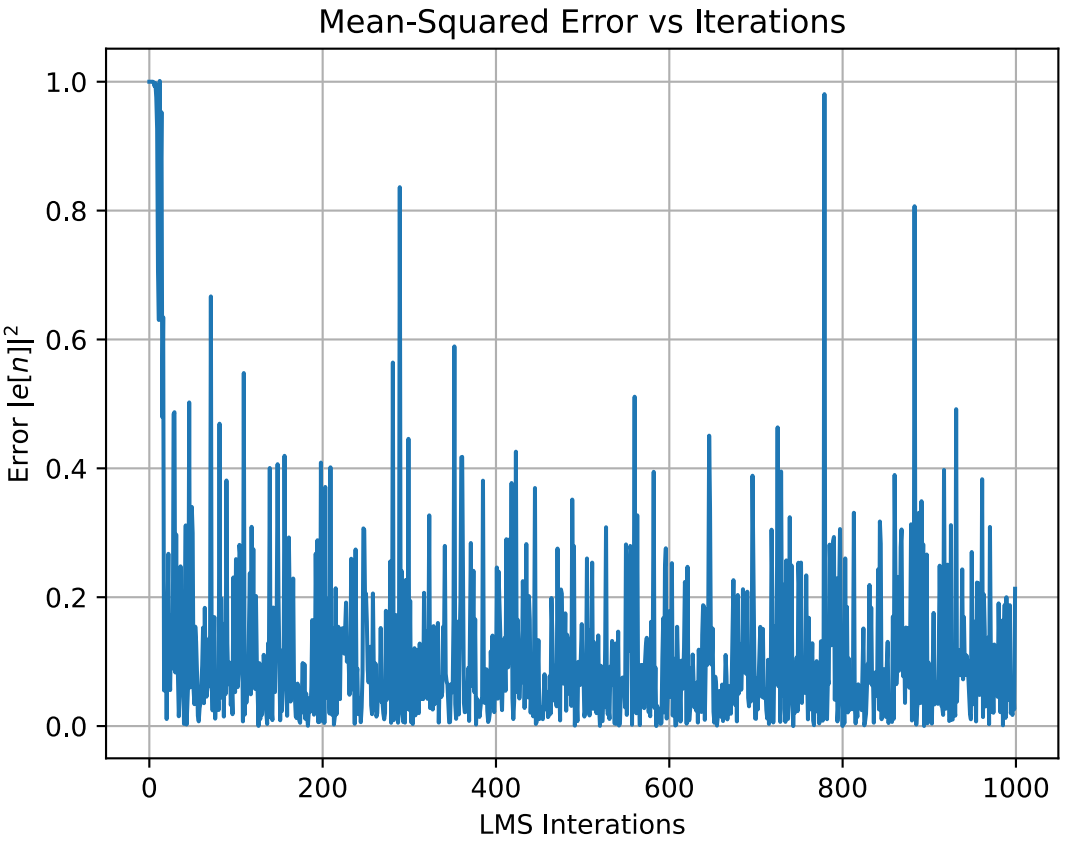


[17]:
```
DD_i = DD_inputs(5,0.005,Nstart=0) # 2*5+1 taps, mu = 0.005
Pe,errors,N_RecBits,z,DD_o = shaped_psk_DDEQ_sim(10,1000,15,DD_i,'src',beta=0.
↪35)
print('Pe Est = %1.2e' % Pe)
print('B_RecBits = %d, errors = %d' % (N_RecBits,errors))
```

```
Pe Est = 1.02e-03
B_RecBits = 983, errors = 1
```

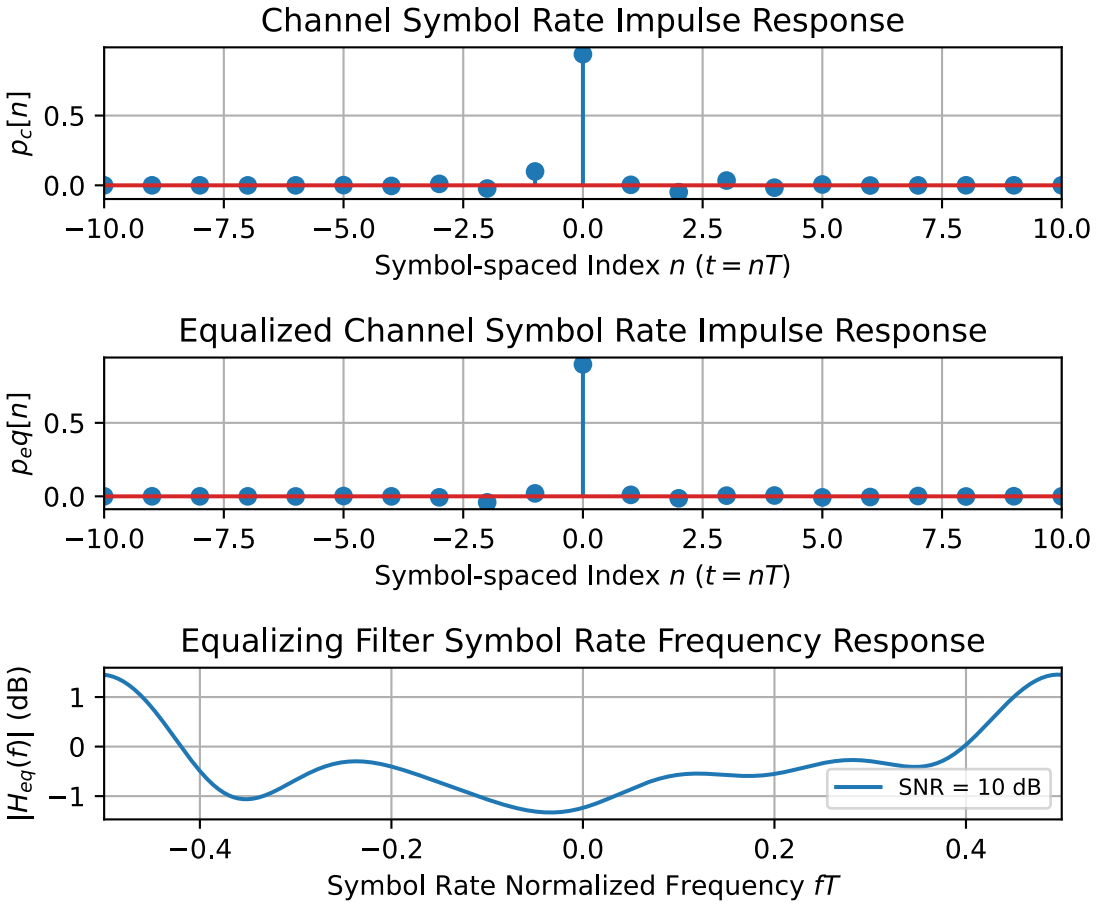
[18]:
```
plot(abs(DD_o.e)**2)
#ylim([0,.1])
title(r'Mean-Squared Error vs Iterations')
ylabel(r'Error $|e[n]|^2$')
```

```
xlabel(r'LMS Interations')
grid();
```



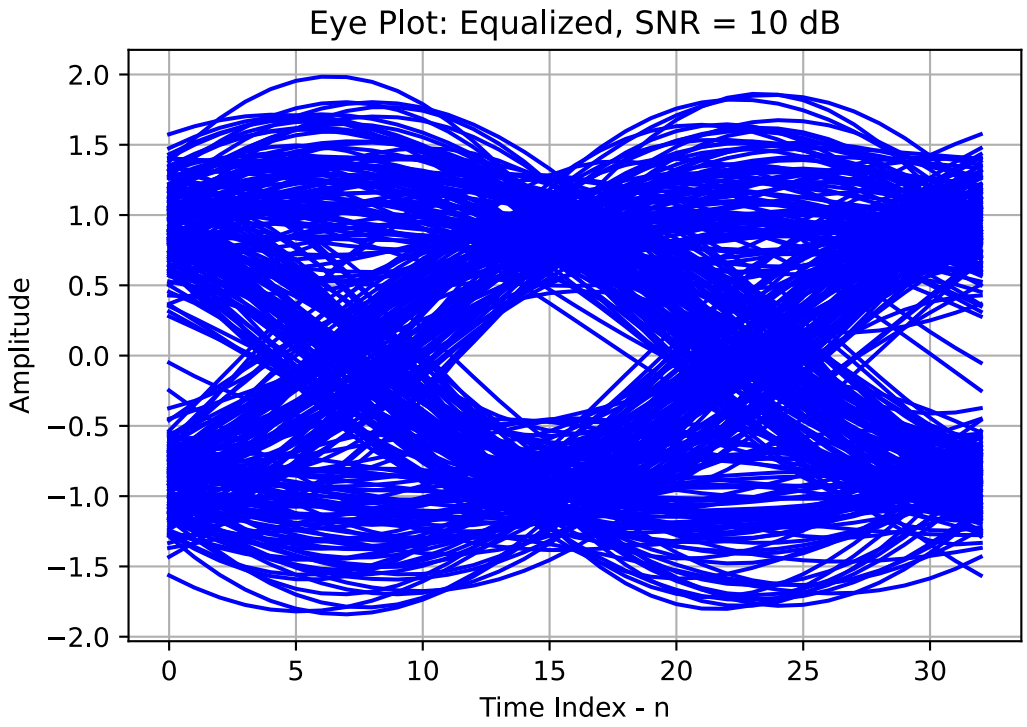
[19]:
```
N = 5 # number of eq taps is 2*N+1
figure(figsize=(6,5))
subplot(311)
n = arange(len(p_c))
n -= argmax(p_c)
stem(n,p_c)
xlim([-2*N,2*N])
title(r'Channel Symbol Rate Impulse Response')
ylabel(r'$p_c[n]$')
xlabel(r'Symbol-spaced Index $n$ ($t=nT$)')
grid();
subplot(312)
p_eq = signal.lfilter(DD_o.Aopt,1,p_c).real #show real part only
n = arange(len(p_eq))
n -= argmax(p_eq)
stem(n,p_eq)
xlim([-2*N,2*N])
```

```
title(r'Equalized Channel Symbol Rate Impulse Response')
ylabel(r'$p_eq[n]$')
xlabel(r'Symbol-spaced Index $n$ ($t=nT$)')
grid();
subplot(313)
f = arange(-0.5,0.5,1/2048)
w, B_eq = signal.freqz(DD_o.Aopt,1,2*pi*f)
plot(f,20*log10(abs(B_eq)))
xlim([-.5,.5])
title(r'Equalizing Filter Symbol Rate Frequency Response')
ylabel(r'$|H_{eq}(f)|$ (dB)')
xlabel(r'Symbol Rate Normalized Frequency $fT$')
legend((r'SNR = 10 dB',),loc='best',fontsize='small')
grid()
tight_layout()
```



[20]:
```
dc.eye_plot(z.real[400*16:],2*16,0)
title(r'Eye Plot: Equalized, SNR = 10 dB')
```

[20]: Text(0.5, 1.0, 'Eye Plot: Equalized, SNR = 10 dB')



1.4 Decision Feedback (DFE) MMSE with LMS Adaptation

The function `shaped_psk_DFE_EQ_sim()` implements a baseband BPSK simulation that incorporates a transmitter with SRC pulse shaping, a bandlimiting channel filter, a receiver matched filter, a decision-directed (DD) symbol-spaced minimum mean-squared error (MMSE) equalizer or simply DD-MMSE equalizer, and a probability of bit error calculator. The DFE-MMSE equalizer employs $2N+1$ taps where presently $N = 5$. The sampling rate is set to $N_s = 16$ samples per bit. The equalizer is designed minimize the mean-square error relative to a DD criterion. The impulse response is obtained by sampling the pulse response from transmitter through the channel filter, through the receiver matched filter at the bit rate. The sampling of the pulse response is chosen to include the maximum or peak of the end-to-end (less the equalizer filter of course) pulse response $p_c[n] = \mathcal{F}^{-1}\{P_{SRC}(e^{j2\pi f/f_s})H_{chan}(e^{j2\pi f/f_s})P_{SRC}(e^{j2\pi f/f_s})\}$, plus N samples either side of it.

[21]:
```
def shaped_psk_DFE_EQ_sim(SNR,N_bits,timing,DFE_i,pulse,beta=0.35):
    """
    [Pe,errors,N_RecBits,z,DD_o] =␣
 ↪shaped_psk_DDEQ_sim(SNR,Nbits,timing,beta,shape,DD_i)

     Coherent Binary BPSK modem DFE-EQ 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
   N_bits = The number of bits simulated
   timing = Sets the sampler timing to give the minimum BEP; find value
            using eyeplot function
    DFE_i = structure variable to hold DFE-LMS parameters:
            DFE_i.N  = number of precursor taps; total taps = N+1
            DFE_i.M  = number of postcursor taps; total taps = M
            DFE_i.mu = LMS algorithm convergence parameter, around 1e-3
            DFE_i.Nstart = Start BEP measuring with a delay due to LMS
                           not convergenced yet, 500 to 1000 good
                           but check MSE plot

       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_mark for
            further analysis, e.g., find a good value for 'timing'
    DFE_o = DFE-LMS output variables:
            DFE_o.z_eq = Equalizer output at symbol spacing
            DFE_o.d_hat = Hard decision data
            DFE_o.e = Error sequence (complex)
            DFE_o.Aopt = Tap weigh vector at end of simulation (complex)
//////////////////////////////////////////////////////////////////////////

Mark Wickert October 2021
"""

Ns = 16; # Fix the number of samples per bit to 16

# Generate PSK tx signal
x,b,data = dc.nrz_bits(N_bits, 16, pulse, beta)

# Introduce channel distortion via a 4th-order
# Butterworth lowpass filter.
# Set fc = 0.5 times the bit rate.
b_chan,a_chan = signal.butter(4,2 * 0.5/Ns)
x = signal.lfilter(b_chan,a_chan,x)

# AWGN channel
y = dc.cpx_awgn(x,SNR,16)
# Enter receiver

# Coherent receiver (at complex baseband)
```

```
    #////////////////////////////////
    # Matched filter is designed by the Tx function as vector b
    # Process complex BB signal through filter to form decision statistic
    z = signal.lfilter(b,1,y)

    #////////////////////////////////
    # Design a MMSE equalizer using a DFE-LMS
    DFE_o = DFE_LMS(z[timing::16],Ns,DFE_i.N,DFE_i.M,DFE_i.mu)
    # Apply as an upsampled filter to better see timing needs
    z = signal.lfilter(dc.upsample(DFE_o.Aopt,Ns),1,z)

    z_det = real(z)
    # Decision logic
    d_hat = (sign(z_det[timing::16])+1)/2 # {0,1} logic levels
    # Error detect
    N_RecBits,errors= dc.bit_errors(data[DFE_i.Nstart:],d_hat[DFE_i.Nstart:])
    Pe = errors/N_RecBits
    return Pe,errors,N_RecBits,z,DFE_o

#////////////////////////////////////////////////////////////////////////
def DFE_LMS(y,Ns,N,M,mu):
    """
    [z_eq,e,Aopt] = DFE_LMS(y,Ns,N,M,mu)
    DFE LMS Function

    Mark Wickert October 2021
    #////////////////////////////////////////////////////////////////////
    """
    Nsim = len(y)
    DFE_A = 1.0
    constellation = array([1, -1])*DFE_A; # BPSK complex baseband constellation
    # constellation = array([3, 1, -1, -3])*DFE_A; # PAM4 complex baseband␣
 ↪constellation
    # constellation = array([1+1j,-1+1j,-1-1j,1-1j])/sqrt(2)*DFE_A; # QPSK␣
 ↪complex baseband constellation
    a = 1 # 1/sqrt(2) for QPSK;
    constellation = constellation*a

    z_eq = zeros(len(y),dtype=complex128)
    d_hat = zeros(len(y),dtype=complex128)
    e = zeros(len(y),dtype=complex128)
    Aopt = hstack((zeros(N), [1], zeros(M)))
    y_pre_pres_states = zeros(N+1)
    y_post_states = zeros(M)
#     y_states = hstack((y_pre_pres_states,y_post_states))

    # Begin the LMS simulation sample-by-sample outer loop
```

```
    for k in range(Nsim):
        # Update input state vector
        # The pre & present portion of the states array
        # can be updated once we have new input sample
        y_pre_pres_states = hstack(([y[k]], y_pre_pres_states[:-1]))

        # Apply equalizer FIR filter sample-by-sample
        # Note: On the first pass y_post_states will be zero
        # as no hard decision have been made yet
        y_states = hstack((y_pre_pres_states,y_post_states))
        z_eq[k] = dot(Aopt,y_states)

        # Form the hard decisions using the filter output
        DFE_error = z_eq[k] - constellation
        min_error,index = min(abs(DFE_error)), argmin(abs(DFE_error))
        # Update the y_post_states to be used in filtering on the next iteration
        y_post_states = hstack(([constellation[index]], y_post_states[:-1]))
        # Store a new hard decision output value
        d_hat[k] = constellation[index] # recovered data in {-1,1} format
        # Form the error
        e[k] = d_hat[k] - z_eq[k]
        # Update the filter coefficients (LMS update eqn)
        Aopt = Aopt + mu*conj(y_states)*e[k]
        # In testing use the below to turn off selected filter taps
        # Aopt[N+1:] = zeros(M)
        # Aopt[2*N+1:] = zeros(M)
    return DFE_outputs(z_eq,d_hat,e,Aopt)

# Data structure for LMS inputs
class DFE_inputs(object):
    """
    A class used to hold DFE input parameters
    """
    def __init__(self,N=5,M=5,mu=1e-3,Nstart=500):
        self.N = N # number of single-sided taps; total taps = N+1
        self.M = M # number of DFE taps
        self.mu = mu # LMS algorithm convergence parameter, around 1e-3
        self.Nstart = Nstart # Start BEP measuring with a delay due to LMS
                             # not fully converged, 500 to 1000 good
                             # but check MSE plot
    # This class has no methods

# Data structure to hold DFEoutput variables
class DFE_outputs(object):
    """
    A class used to hold DFE output variables
    """
```

```
    def __init__(self,z_eq,d_hat,e,Aopt):
        self.z_eq = z_eq    # Equalizer output at symbol spacing
        self.d_hat = d_hat  # Hard decision data
        self.e = e          # Error sequence (complex)
        self.Aopt = Aopt    # Tap weigh vector at end of simulation (complex)
    # This class has no methods
```

1.4.1 SNR = 50 dB

- Begin with $\mu = 0$ to see the eye plot without equalization
- Next turn on the equalizer by setting $\mu = 0.005$

[22]:
```
DFE_i = DFE_inputs(5,5,0.005,Nstart=0) # 2*5+1 taps, mu = 0.005
Pe,errors,N_RecBits,z,DFE_o =␣
 ↪shaped_psk_DFE_EQ_sim(100,1000,15,DFE_i,'src',beta=0.35)
print('Pe Est = %1.2e' % Pe)
print('B_RecBits = %d, errors = %d' % (N_RecBits,errors))
```

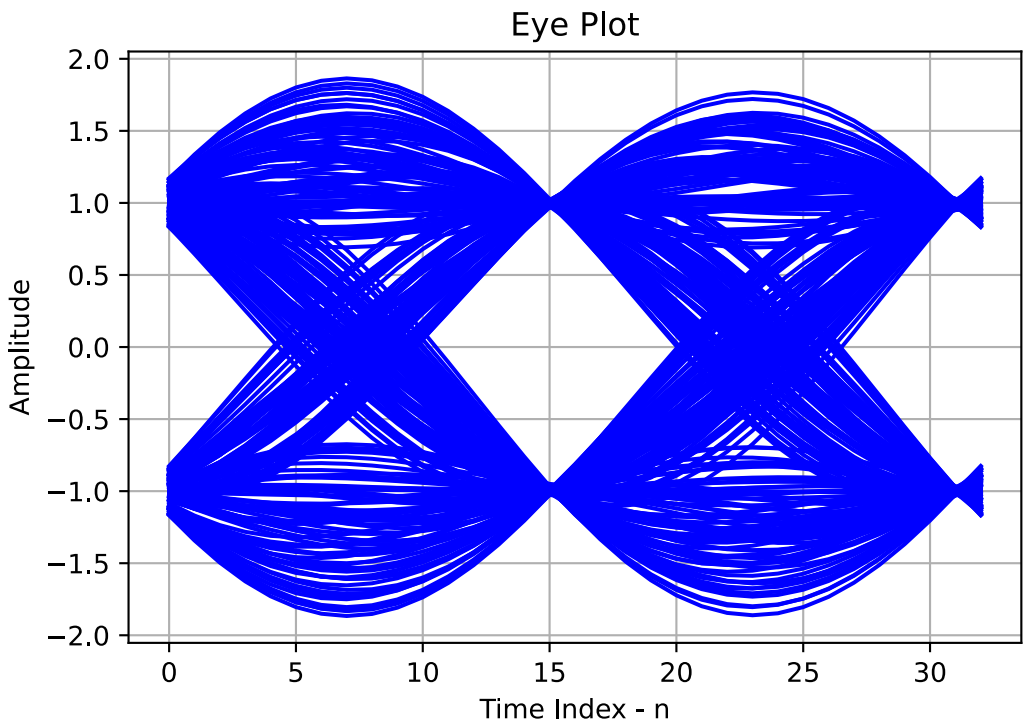
```
Pe Est = 0.00e+00
B_RecBits = 983, errors = 0
```

[23]:
```
DFE_o.Aopt
```

[23]:
```
array([-0.00245973+2.32918652e-07j,  0.00705037-2.01679473e-07j,
       -0.01647129+3.77540340e-07j,  0.03639587-2.93641227e-07j,
       -0.11138035+5.20083378e-07j,  1.06505329+3.09083753e-07j,
       -0.01059294-3.37076360e-07j,  0.05591987-3.02760138e-07j,
       -0.03802959-1.96103315e-07j,  0.01860325+4.24846412e-07j,
       -0.00670105+2.08876910e-07j])
```

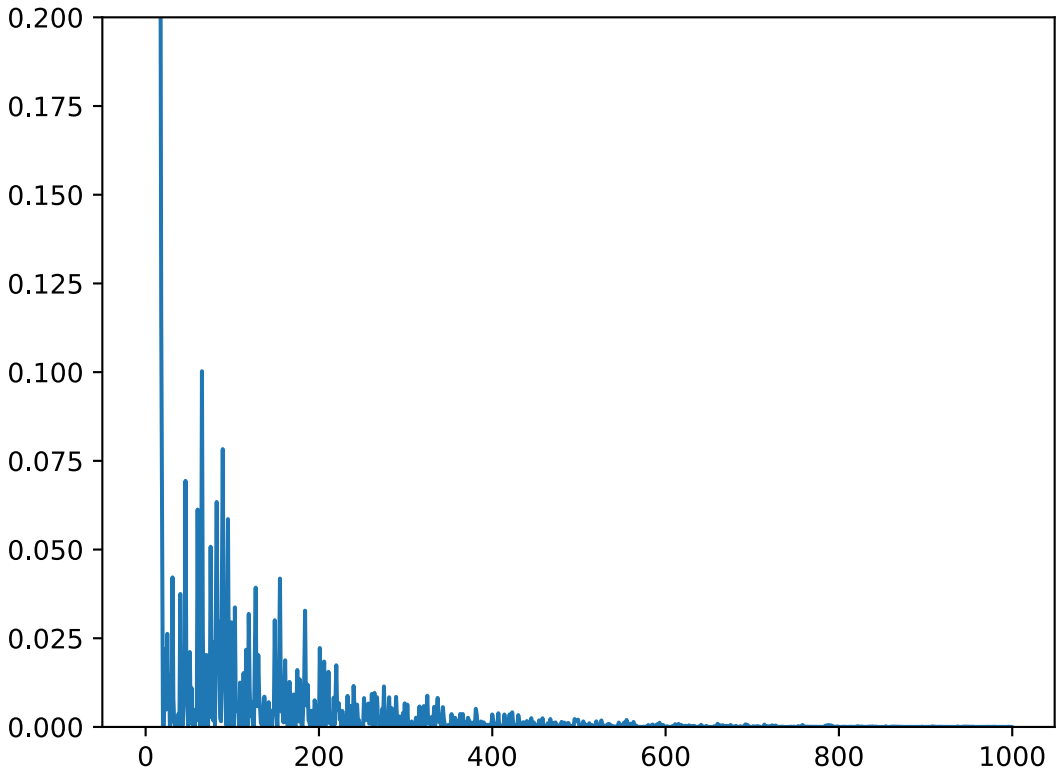
[24]:
```
dc.eye_plot(z[400*16:].real,2*16,0)
# grid()
```

[24]:
```
0
```



[25]:
```
plot(abs(DFE_o.e)**2)
ylim([0,.2])
```

[25]: (0.0, 0.2)


0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0
200
400
600
800
1000

[ ]: