Chapter 10

Communications Applications

Introduction

The great demand for Internet connectivity by consumers and the wireless revolution have pushed the demand for DSP based communication solutions to a very high level. What this chapter attempts to do is work through some of the basic issues in using real-time DSP to handle both analog and digital modulation schemes. The first step in this process is getting analog communication waveforms into and out of the discrete-time domain. Next follows a discussion of some basic signal processing functions, such as a DSP Costas-Loop for coherent carrier recovery.

The environment where the communication signal resides, the size weight and power requirements, and the market where the communication services reside, dictates how to choose a particular implementation. These is no simple one approach satisfies all solution. The dividing line between analog and DSP based portions of a particular implementation is always moving towards DSP, but where this line is placed depends on the above mentioned factors. Once you decide to use DSP, then you must decide which portions, if any are implemented using ASICs and which can be done on a general purpose DSP. By visiting a few vendor Web sites, you soon find that there are a vast array of

wireless communications oriented ASICs just waiting for you.

Summary of Factors Relating to Implementation Choices

How to structure the material presented here in this chapter is thus a challenge in itself. Factors which may be important in a DSP based communication system design include:

The Communications Channel

- Wired bandlimited channel, e.g., wireline/PSTN modems
- RF/Microwave line-of-sight, e.g., satellite, common carrier terrestrial
- Mobile radio
- Fiber optic
- others

Size, Weight, and Power Requirements

- Cellular base station
- Cellular portable
- PC PSTN modem; desktop and portable
- Satellite and satellite earth station
- Terrestrial point-to-point
- Wireless data networks; access node and remote
- others

Market Where Service Resides

- All consumer, e.g., cell phones
- Government services non-military
- Military
- others

DSP Technology

- ASIC
- General purpose DSP
- A combination of the two

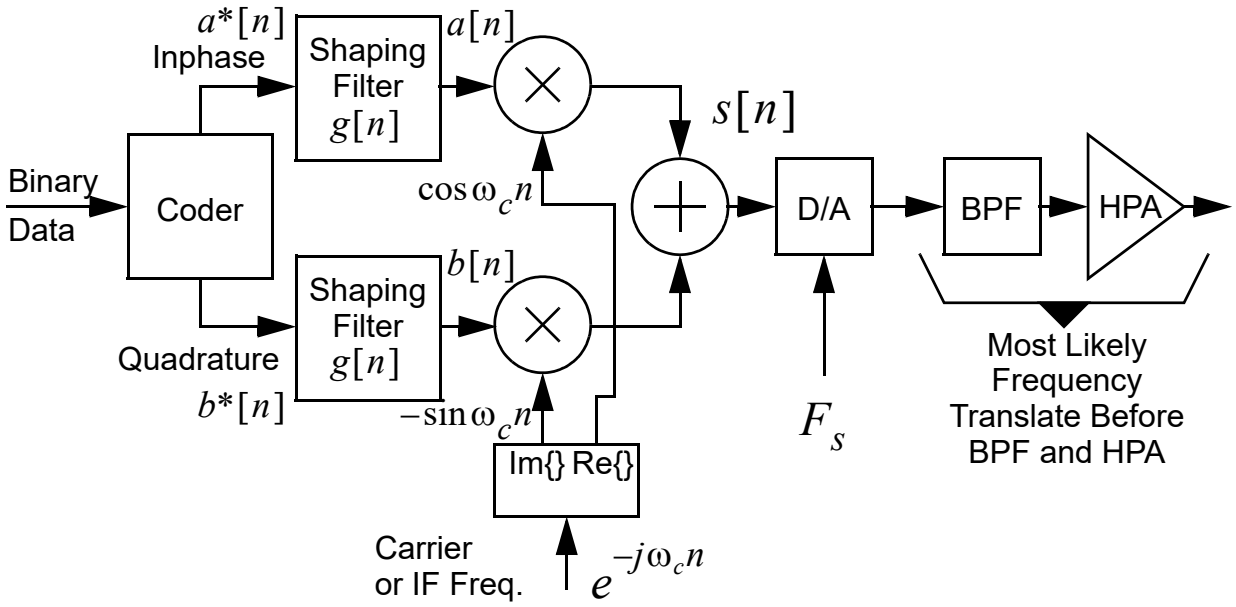
Transmitting Signals

In the book by Tretter[1] DSP techniques for communication system design is discussed, with emphasis on wireline modems. A good blend of theory and practice is presented. From the introductory section we know that with the great variety of communication system scenarios there are more considerations.

Baseband/IF Transmitter

- The most direct approach for using DSP at the transmitter is to simply modulate directly onto a discrete-time carrier and pass the composite signal out through the D/A

1. S. Tretter, *Communication System Design Using DSP Algorithms with Laboratory Experiments for the TMS320C6713™*, Kluwer Academic/Plenum Publishers, 2008.



- The coder output is likely to be an impulse train with the shaping filter, $g[n]$, used to control the baseband transmitted spectral occupancy
- The discrete-time signal prior to the D/A is of the form

$$s[n] = a[n]\cos(\omega_c n) - b[n]\sin(\omega_c n) \tag{10.1}$$

where $a[n] = a^*[n] * g[n]$, $b[n] = b^*[n] * g[n]$,

$$a^*[n] = \sum_{k=-\infty}^{\infty} a_k \delta[n - kN_s] \tag{10.2}$$

$$b^*[n] = \sum_{k=-\infty}^{\infty} b_k \delta[n - kN_s] \tag{10.3}$$

and N_s is the number of samples per symbol

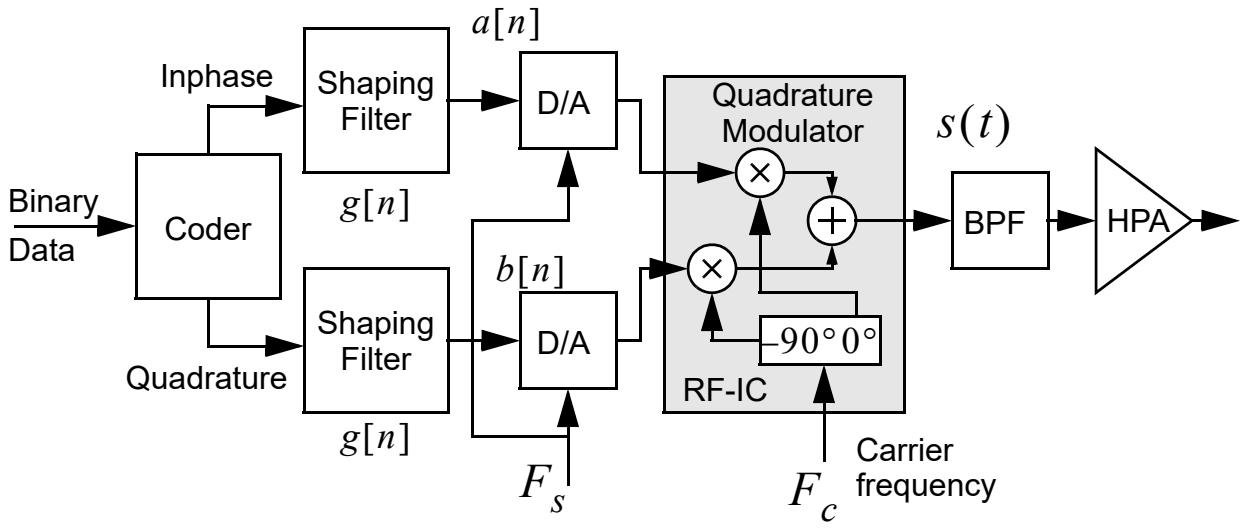
- The points in the 2-dimensional space defined by the point pairs (a_k, b_k), define the *signal constellation*
- In practice these point pairs are viewed in the complex plane as $c_k = a_k + jb_k$
- The coder block groups consecutive blocks of J-bits into J-bit binary words that define a 2^J element alphabet
  - The operation of J-bits being formed into a J-bit word is really nothing more than a serial-to-parallel converter
  - In other applications it might be that $K < J$ input bits form the J-bit word; in this case a rate K/J error correcting code may be involved (block or convolutional)
- The complex envelope or complex baseband signal corresponding to $s[n]$ is

$$\tilde{s}[n] = a[n] + jb[n] \tag{10.4}$$

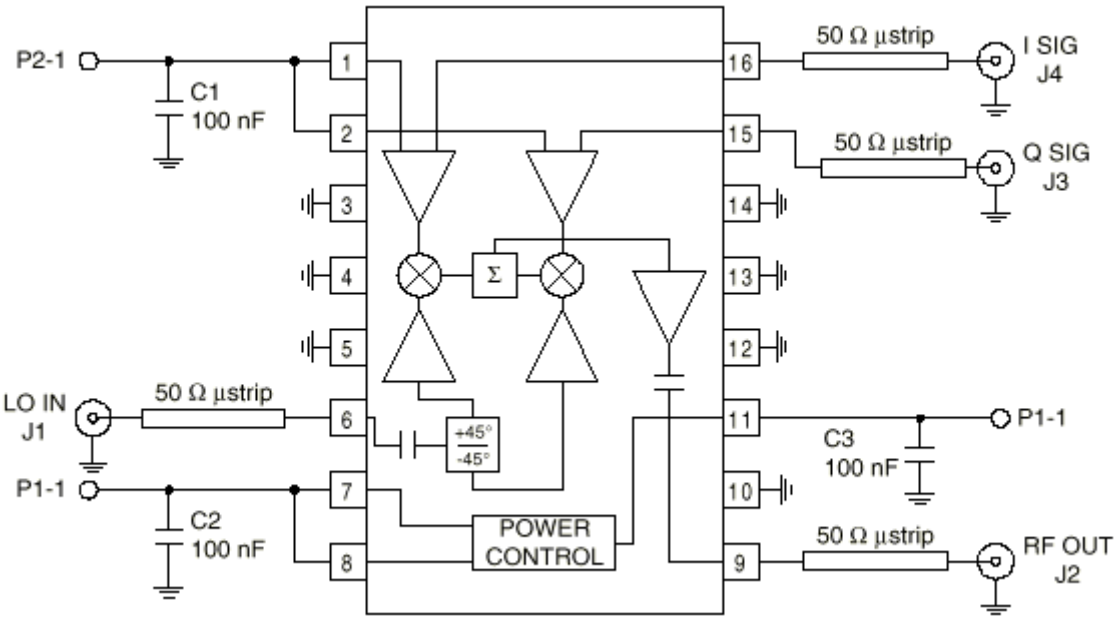
- For a wireline modem application the carrier frequencies are very low and may not need to be translated as depicted in the above figure
- For most other cases the D/A output would be considered at an IF frequency, and would need analog signal processing components to translate the signal to the proper center frequency, followed by high power amplification (HPA)

Complex Baseband Transmitter

- A more complex, yet also more flexible approach, is to have the DSP create the signal in complex baseband (complex envelope) format



- In this implementation the DSP is not responsible for carrier generation, but the AIC must have two D/A output channels
- Below 2.4 GHz the quadrature modulator is typically an RF integrated circuit which contains the mixers, power combiner, and phase splitting circuit for generating $\cos 2\pi f_c t$ and $\sin 2\pi f_c t$
  - As an example consider the *RF MicroDevices* part number RF2422



RF Microdevices RF2422
2.4 GHz Quadrature Modulator

- In this implementation the signals $a[n]$ and $b[n]$ are the impulse modulator outputs convolved with the pulse shaping filter

$$a[n] = \sum_{k=-\infty}^{\infty} a_k g[n - kN_s] \tag{10.5}$$

$$b[n] = \sum_{k=-\infty}^{\infty} b_k g[n - kN_s] \tag{10.6}$$

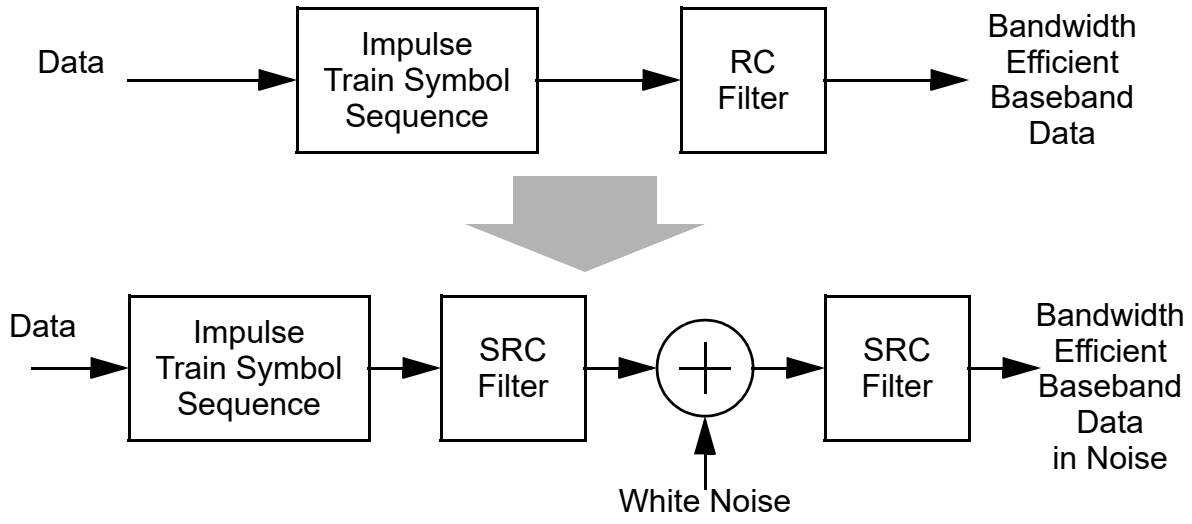
- Following the quadrature modulator we have the analog signal

$$s(t) = a(t)\cos(2\pi F_c t) - b(t)\sin(2\pi F_c t) \tag{10.7}$$

  – The signals $a(t)$ and $b(t)$ are $a[n]$ and $b[n]$ along with the influences of the D/A analog reconstruction filters

Practical Pulse Shaping

- A common form of pulse shaping is one that satisfies the Nyquist criterion for zero intersymbol interference (ISI)
- Zero ISI means that pulses corresponding to adjacent symbols do not interfere with each other at symbol spaced sampling instants
- A popular baseband shaping filter is the raised cosine, which has a parameter $\alpha \in [0, 1]$, known as the *excess bandwidth factor*
- When the channel frequency response is flat across the signal bandwidth and the noise is white (flat spectrum), it is best to equally split the raised cosine (RC) frequency response shape into the product of two square-root raised cosine (SRC) frequency responses



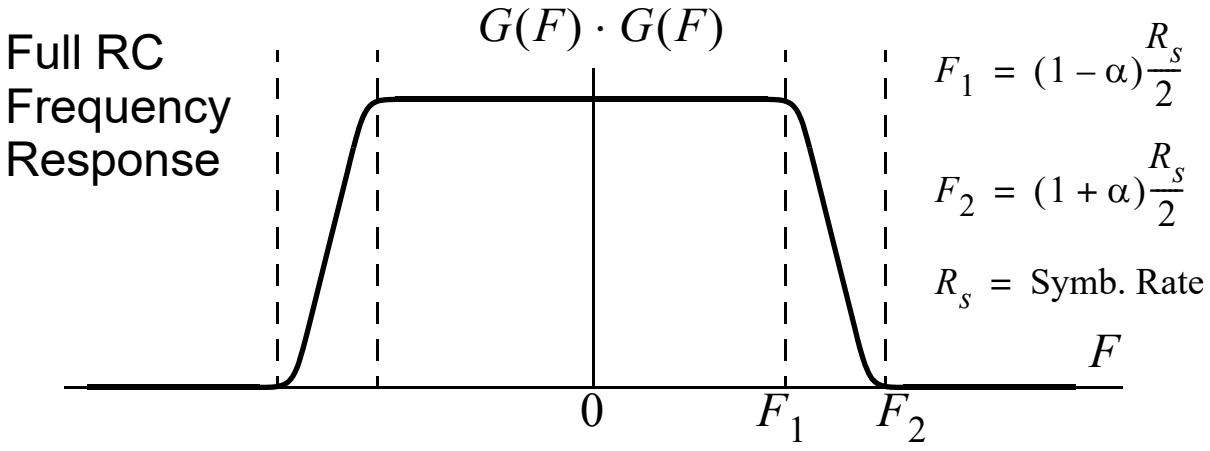
Example: MATLAB Implementation of Shaped BPSK Modulation using an FIR SRC filter

- The SRC impulse response can be implemented as an FIR filter by truncating the true SRC impulse response

$$g(t) = \frac{4\alpha\cos\left[(1+\alpha)\pi\frac{t}{T_s}\right] + \frac{\sin\left[(1-\alpha)\pi\frac{t}{T_s}\right]}{4\alpha t/T_s}}{\pi\sqrt{T_s}[1-16\alpha^2 t^2/T_s^2]} \qquad (10.8)$$

  where T_s is the symbol duration

- The frequency response of the full RC pulse (equivalent to $G(F)G(F)$) is shown in the following figure

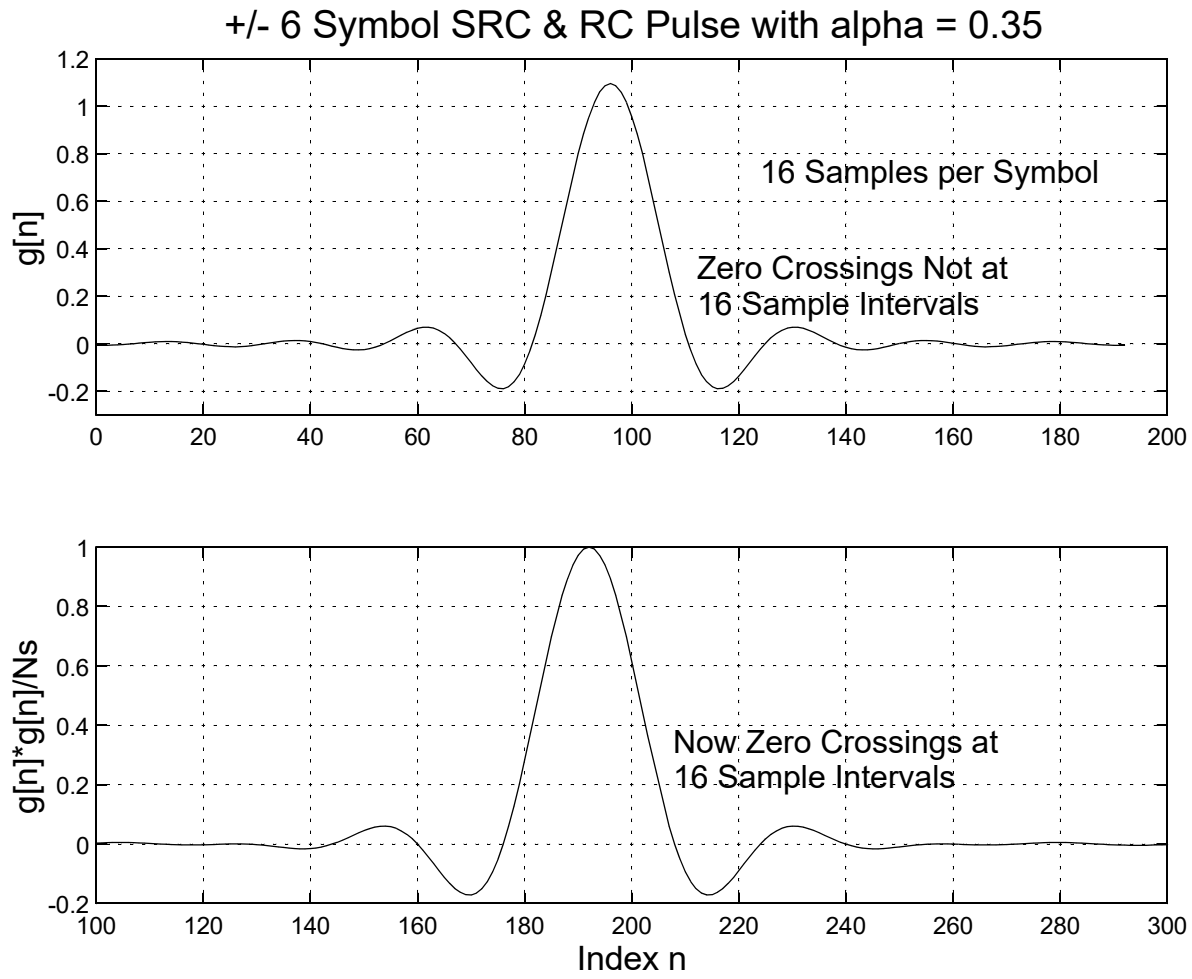


- A popular truncation interval for this impulse response is ±6 symbols, which implies a total impulse response duration of 12 symbol periods at N_s samples per symbol

- A MATLAB function for creating such a pulse shaping filter is the following

```
function b = sqrt_rc_imp(Ns,alpha,M)
%        b = sqrt_rc_imp(Ns,alpha,M)
%   Sqrt-Raised-Cosine Impulse Response Filter
%    Ns = number of samples per symbol
% alpha = excess bandwidth factor = 0.35 for IS 136
%     M = equals sqrt(RC) one-sided symbol truncation factor

% Design the filter
n = -M*Ns:M*Ns;
b = zeros(size(n));
a = alpha;
for i=1:length(n),
   if (1 - 16*a^2*(n(i)/Ns)^2) == 0
      b(i) = 1/2*((1+a)*sin((1+a)*pi/(4*a))- ...
          (1-a)*cos((1-a)*pi/(4*a))+(4*a)/pi*sin((1-a)*pi/(4*a)));
   else
      b(i) = 4*a./(pi*(1 - 16*a^2*(n(i)/Ns).^2));
      b(i) = b(i).*(cos((1+a)*pi*n(i)/Ns) + ...
          sinc((1-a)*n(i)/Ns)*(1-a)*pi/(4*a));
   end %end if statement
end %end for loop
```

- A plot of $g[n]$ at 16 samples per symbol (more typically this would be 4 or so), ± 6 symbol length, and $\alpha = 0.35$ is shown below along with the composite RC pulse $g[n] * g[n]$

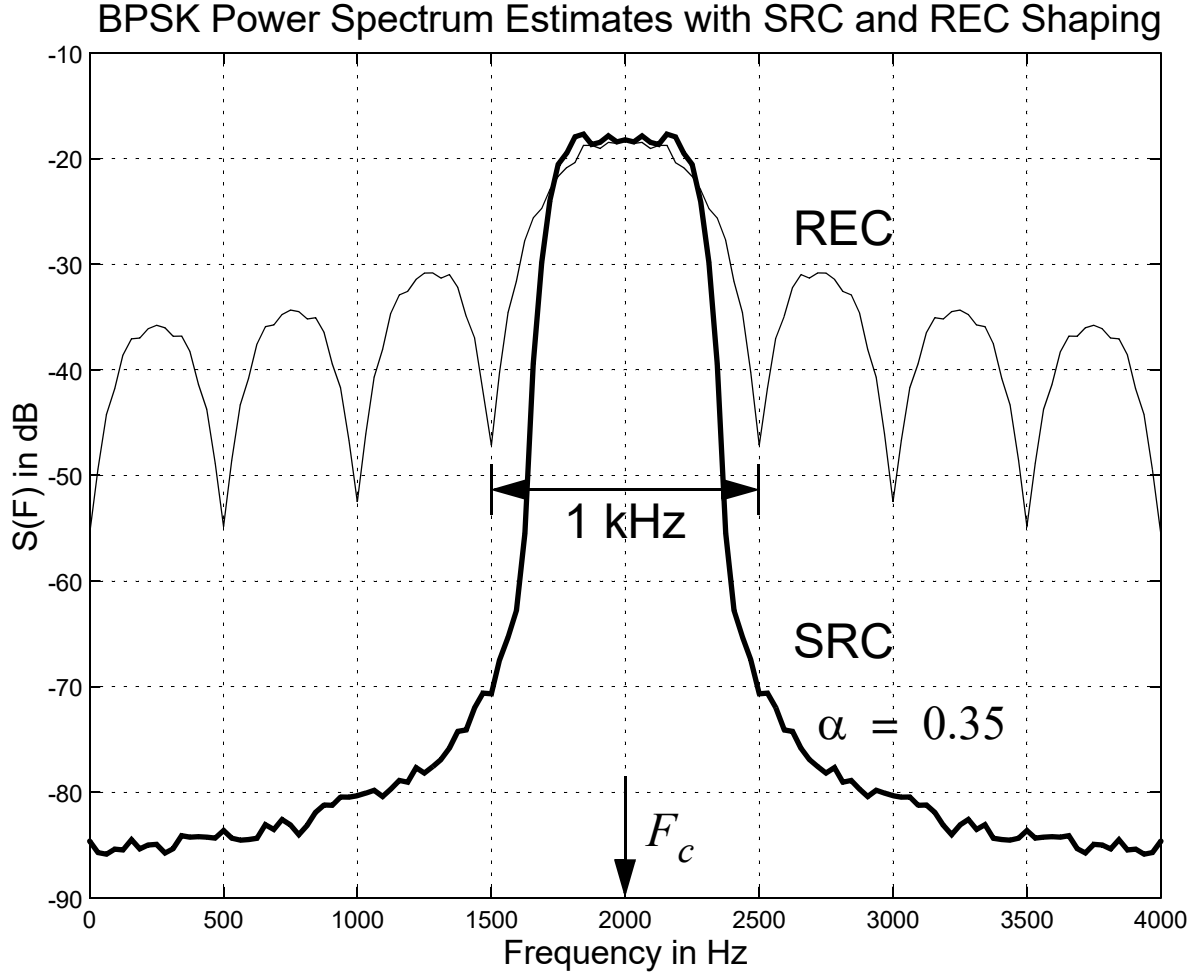


- To investigate further we will now implement a binary phase-shift keyed (BPSK) modulator ($a_k = \pm 1, b_k = 0$) using both rectangular and SRC pulse shaping

```
% A Square-Root Raised Cosine Pulse Shaping
% Demo that Creates a BPSK waveform. src_bpsk.m
%
% Mark Wickert 5/98
%
% Set Fs = 8 kHz
% Set Rb = 500 bits/sec
% Create a record of M = 1000 symbols using m_ary.m
% which creates symbols of amplitude 0 & 1
Fs = 8000;
Rb = 500;
M = 1000;
```

```
d = m_ary(2,M,1);
d = 2*d - 1; % translate to -1/+1 amplitudes
d = [d zeros(M,Fs/Rb-1)]; % Set up for impulse modulation
d = reshape(d',1,M*Fs/Rb);
n = 0:(M*Fs/Rb-1);
% Square-root raised cosine filter and then modulate
% on a cosine carrier of 2000 Hz
b_src = sqrt_rc_imp(Fs/Rb,0.35,6);
df = filter(b_src*Rb/Fs,1,d); %Filter with unity DC gain
Fc = 2000;
x = df.*cos(2*pi*Fc/Fs*n); %SRC Shaped BPSK
% Rectangle Pulse Shape filter and then modulate
% on a cosine carrier of 2000 Hz
b_rec = ones(1,Fs/Rb)/(Fs/Rb); %Make unity gain rec
dff = filter(b_rec,1,d);
y = dff.*cos(2*pi*Fc/Fs*n); %Rectangle Pulse BPSK

function [data,y] = m_ary(M, K, Ns)
% M_ary   [data,y] = m_ary(M, K, Ns): Create an M-level, e.g.,
M=2,4,8,16,
%        data sequence with Ns samples per symbol and containing K
%         total symbols; Ns * K total samples.
%        data is a vector of symbols taking on values from 0 to M-1
%         with Ns samples per symbol
%         y is a vector of symbol values at one sample per symbol

% create the data sequence
x = rand(K,1);
y = round((M*x)-0.5);

% create a zero padded (interpolated by Ns) symbol sequence
symb = [y zeros(K,Ns-1)]';
symb = reshape(symb,Ns*K,1);

% filter symb with a moving average filter of length Ns to fill-in
% the zero samples
data = filter(ones(1,Ns),1,symb);
```

- Compare the power spectral densities of the two pulse shaping schemes

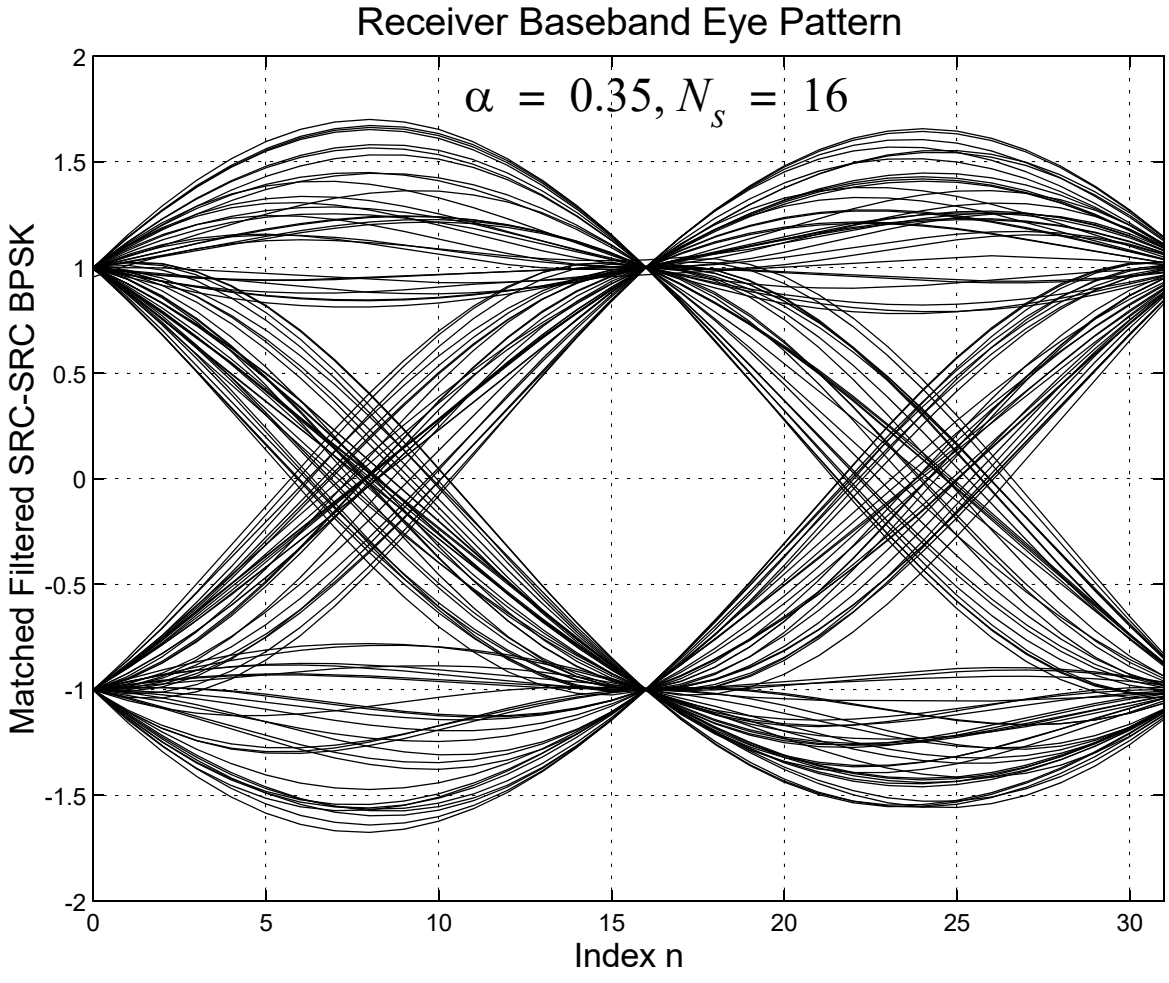


- The eye pattern of the baseband shaped impulse train modulation, following a second SRC or matched filter, is shown below as a result of using the MATLAB function `eyeplot()`

```
function eyeplot(data, W, OFS)
% eyeplot eyeplot(data, W, offset): Plot the eye pattern of a
% digital communication waveform using a window length of W
% and offset OFS

% define some variables
x = 0:W-1;
```

```
L = length(data);
% set axis limits
Limit = max([abs(min(data)) abs(max(data))]);
axis([0,W,-Limit,Limit]);
while L >= W+OFS,
        temp = data(1+OFS:W+OFS);
        plot(x,temp);
        hold on
        data = data(W+1:L);
        L = length(data);
end
hold off
```

```
» eyeplot(filter(b_src,1,df(1000:5000)),32,105)
```

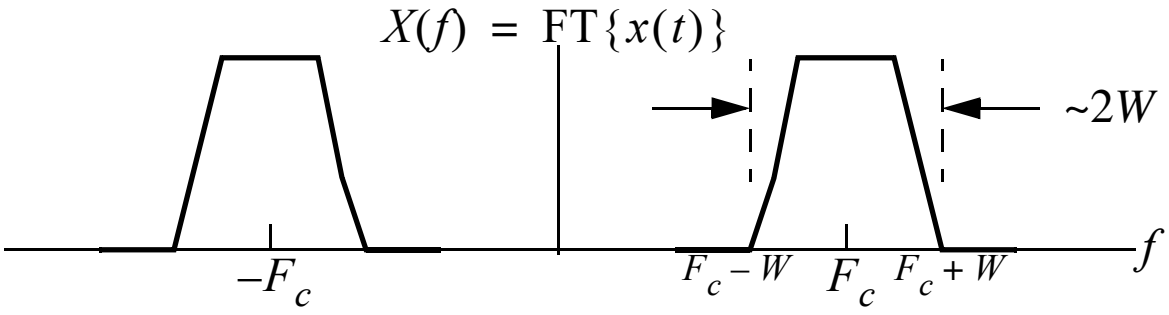


Receiving Signals

Signal processing at the receiver is typically more complex than at the transmitter. There are a lot of details involved. Only the main points will be considered here, or at least for now. In addition to the wireline modem design details given in Tretter, a very good source for more general information on DSP implementation of digital communication receivers is Meyr[1].

Complex Envelope Representation

- A communication signal such as $x(t)$, wireless or wired, is very often in the form of a bandpass signal



- The *complex envelope* representation of a bandpass signal can be written as

$$x(t) = \mathrm{Re}[\tilde{x}(t)e^{j\Omega_c t}] \qquad (10.9)$$

where $\tilde{x}(t)$ is the complex envelope and $\Omega_c = 2\pi F_c$ is the carrier frequency in rad/s

1. H. Meyr, M. Moeneclaey, and S. Fetchel, *Digital Communication Receivers*, John Wiley, New York, 1998.

- The signal $\tilde{x}(t)$ is a complex baseband representation of $x(t)$
- We can easily expand (10.9) in rectangular form as

$$x(t) = x_I(t)\cos(\Omega_c t) - x_Q(t)\sin(\Omega_c t) \qquad (10.10)$$

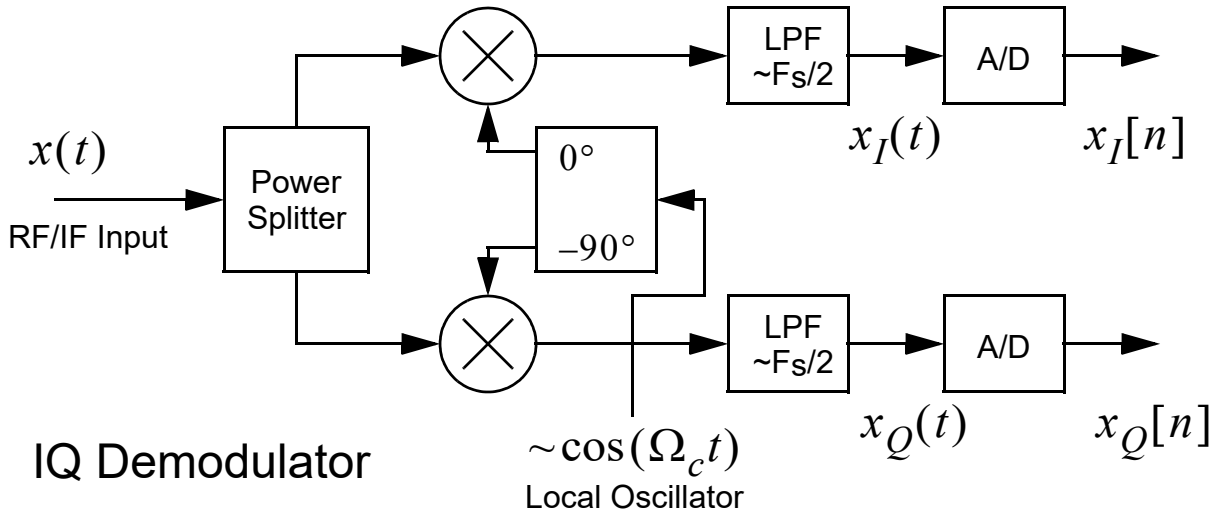
from which it follows that

$$\tilde{x}(t) = x_I(t) + jx_Q(t) \qquad (10.11)$$

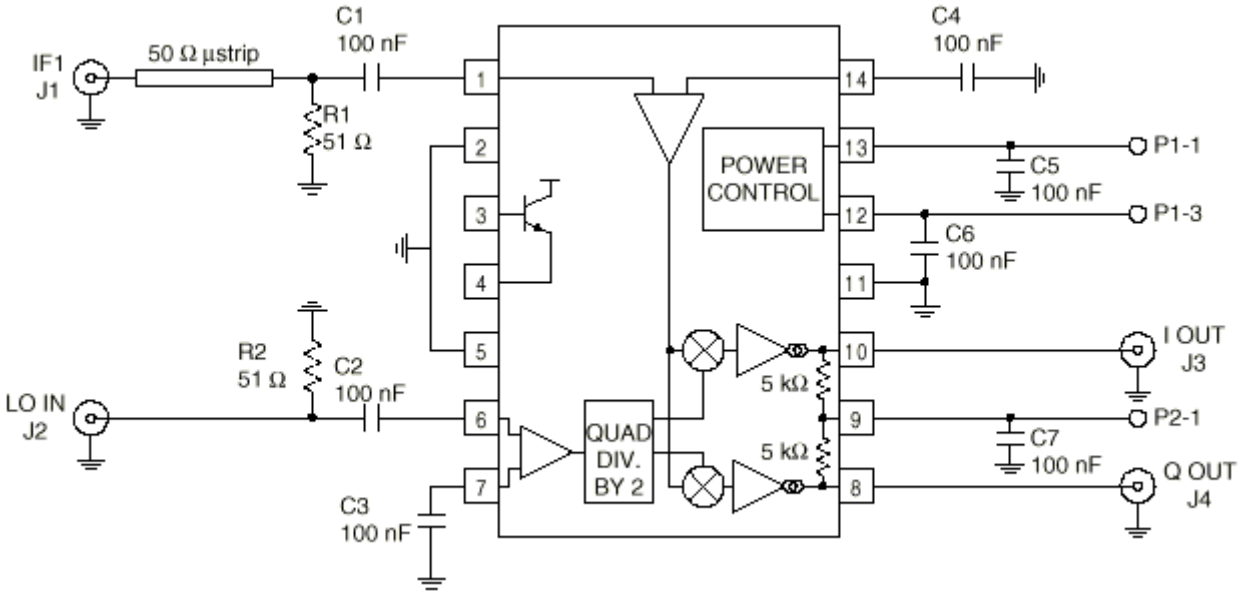
– Note: $x_I(t)$ and $x_Q(t)$ are referred to as the inphase and quadrature parts of the complex baseband signal, respectively

Standard I-Q Demodulation

- Given two channels of analog input on the DSP, a popular approach in receiver design is to convert the modulated RF or IF carrier to complex baseband form and then A/D convert $x_I(t)$ and $x_Q(t)$ into $x_I[n]$ and $x_Q[n]$ respectively



– As an example consider the RF MicroDevices part number RF2711



RF Microdevices RF2711 Quadrature Demodulator

- In the IQ demodulator the local oscillator does not have to remove the carrier completely
- Once in the discrete-time domain a carrier tracking loop can be used to remove a small frequency offset and track phase errors
- For wireline modems, with the carrier frequencies being very low, it is possible to A/D convert the entire modulated carrier signal and then remove the carrier in the discrete-time domain

Using the Hilbert Transform

- The complex envelope can also be obtained by using the Hilbert transform, which is defined by

$$\hat{x}(t) = x(t) * \frac{1}{\pi t} = \frac{1}{\pi}\int_{-\infty}^{\infty} \frac{x(\tau)}{t-\tau} d\tau \qquad (10.12)$$

- In theory the Hilbert transform (HT) of a signal is obtained by passing it through a filter with impulse response

$$h(t) = \frac{1}{\pi t} \tag{10.13}$$

and frequency response

$$H(\Omega) = -j\,\mathrm{sign}\Omega \tag{10.14}$$

where

$$\mathrm{sign}\Omega = \begin{cases} 1, & \Omega > 0 \\ 0, & \Omega = 0 \\ -1, & \Omega < 0 \end{cases} \tag{10.15}$$

- Simply put, an ideal Hilbert transforming filter is a $90°$ phase shifter

$$\tilde{X}(\Omega) = H(\Omega)X(\Omega) = (-j\,\mathrm{sign}\Omega)X(\Omega) \tag{10.16}$$

- A practical Hilbert transforming filter can be designed in the discrete-time domain as an FIR filter
  - MATLAB will be used to carry out this design shortly
- Simple HT relationships are:

$$\mathrm{HT}\{\cos\Omega_c t\} = \sin\Omega_c t \tag{10.17}$$

$$\mathrm{HT}\{\sin\Omega_c t\} = -\cos\Omega_c t \tag{10.18}$$

- A useful theorem for bandpass signals is that if $m(t)$ is low-pass with $M(\omega) \cong 0$ for $f > W_1$ and $c(t)$ is highpass with $c(t) \cong 0$ for $f < W_2$ and $W_1 < W_2$, then

$$\mathrm{HT}\{m(t)c(t)\} = m(t)\hat{c}(t) \tag{10.19}$$

– A practical example is

$$\mathrm{HT}\{m(t)\cos\Omega_c t\} = m(t)\sin\Omega_c t,\ F_c > W_1$$

- The signal

$$x_+(t) = x(t) + j\hat{x}(t) \tag{10.20}$$

is known as the *analytic signal* or *pre-envelope* associated with $x(t)$

- The Fourier transform of $x_+(t)$ is

$$X_+(\Omega) = 2X(\Omega)u(\Omega) \tag{10.21}$$

where $u(\Omega)$ is the usual step function, except now in the frequency domain

- Referring back to the complex envelope in (10.9), we see that

$$\tilde{x}(t) = x_+(t)e^{-j\Omega_c t} = [x(t) + j\hat{x}(t)]e^{-j\Omega_c t} \tag{10.22}$$

and

$$x_I(t) = x(t)\cos(\Omega_c t) + \hat{x}(t)\sin(\Omega_c t) \tag{10.23}$$

$$x_Q(t) = -j[x(t)\sin(\Omega_c t) - \hat{x}(t)\cos(\Omega_c t)] \tag{10.24}$$

Example: A Simple MATLAB Simulation

- In this first example we generate a simple amplitude modulated carrier

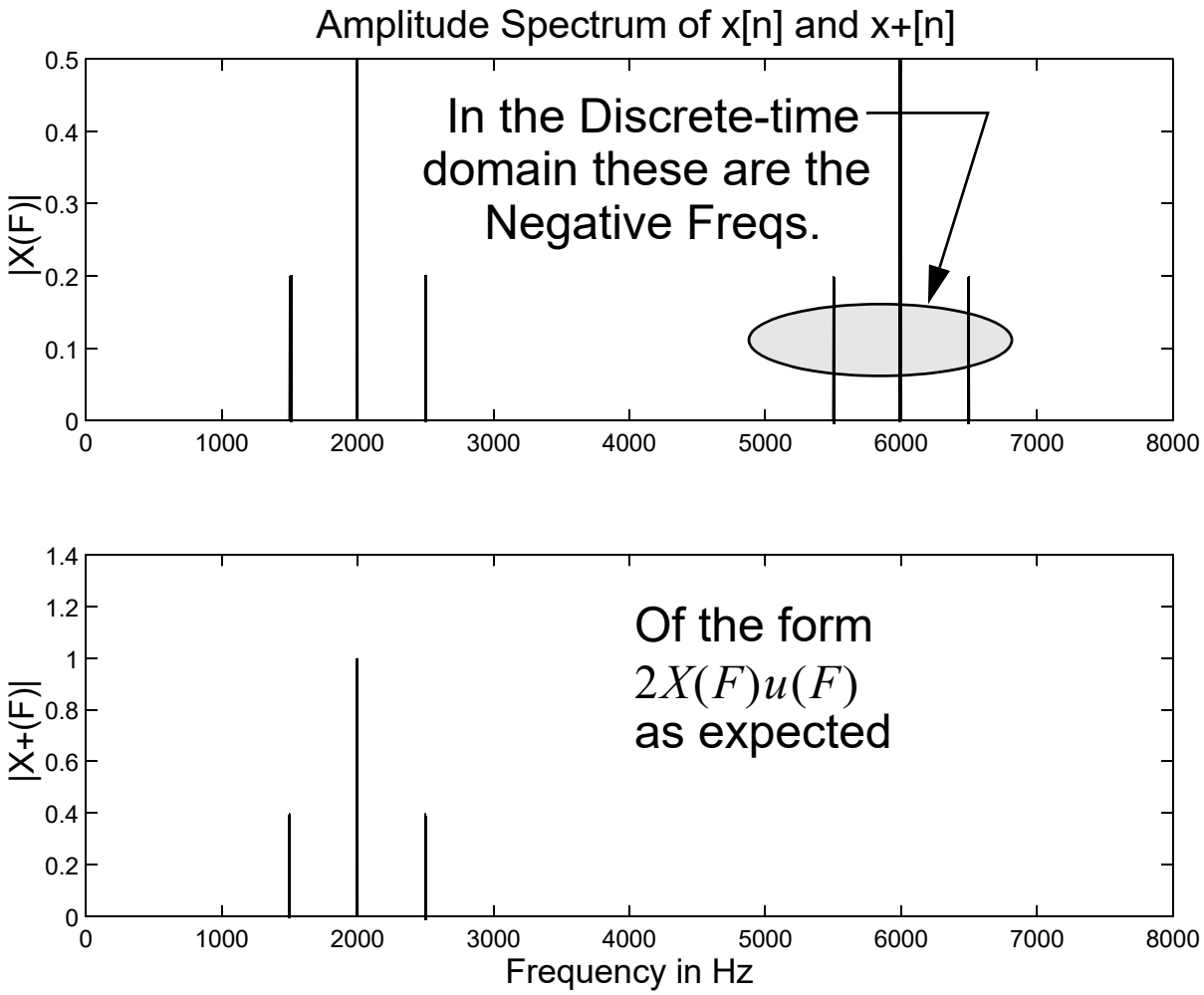
$$x[n] = \left[1 + 0.8\cos\left(2\pi\frac{500}{8000}n\right)\right]\cos\left(2\pi\frac{2000}{8000}n\right) \tag{10.25}$$

with modulation index 0.8, $F_m = 500\,\text{Hz}$, $F_c = 2000\,\text{Hz}$, and a sampling rate of $F_s = 8000\,\text{Hz}$

```
% A Simple Demo Using the Hilbert Transform
% on an AM waveform. hilbert_am.m
%
% Mark Wickert 5/98
%
% Set Fs = 8 kHz
% Set Fc = 2 kHz
% Set Fm = 500 Hz bits/sec
% Create a record of M = 2048 samples
Fs = 8000;
Fc = 2000;
Fm = 500;
M = 2048;
n = 0:M-1;
x = (1+ 0.8*cos(2*pi*Fm/Fs*n)).*cos(2*pi*Fc/Fs*n);
```

- Using the FFT we can plot the amplitude spectrum of $x[n]$ and the corresponding analytic signal $x_+[n]$
- MATLAB makes it very easy to obtain the analytic signal using the function `hilbert()`, which returns an analytic sequence $x_+[n] = x[n] + j\hat{x}[n]$

```
» subplot(211);
» plot(n*Fs/M,abs(fft(x))/M)
» subplot(212);
» plot(n*Fs/M,abs(fft(hilbert(x)))/M)
```



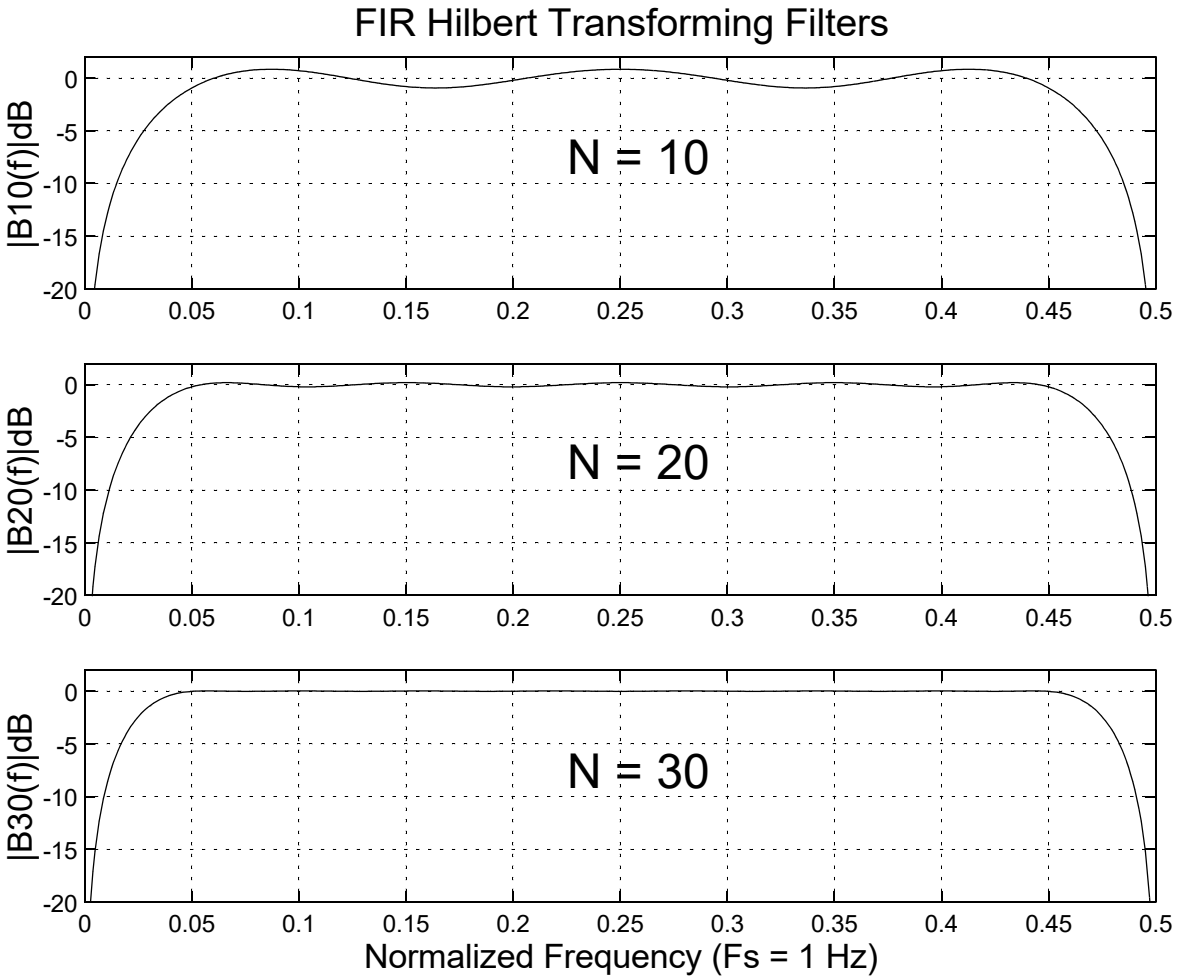
- To obtain an FIR based HT filter we can use the MATLAB function `remez()`, e.g.,

```
» b=remez(N,[.1 .9],[1 1],'Hilbert')
```

  which designs an order N filter ($N + 1$ taps) that has approximately unity gain for $0.1F_s/2 \le F \le 0.9F_s/2$

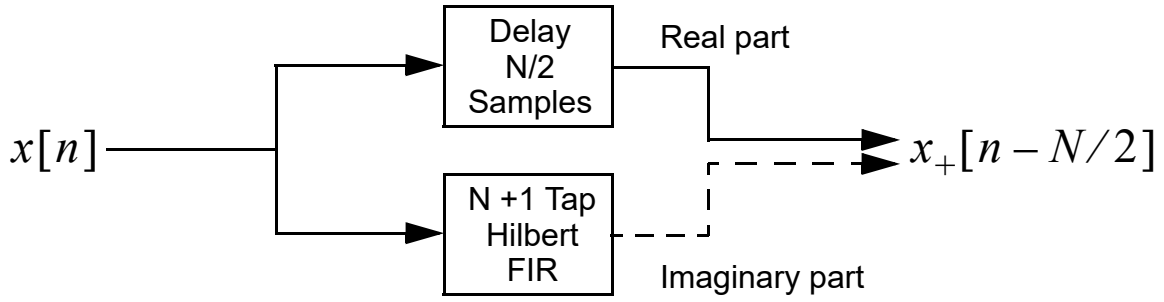
- The gain flatness for three different filter orders is considered below:

```
» b10=remez(10,[.1 .9],[1 1],'Hilbert');
» b20=remez(20,[.1 .9],[1 1],'Hilbert');
» b30=remez(30,[.1 .9],[1 1],'Hilbert');
```



  - Real-time implementation of the above is easy

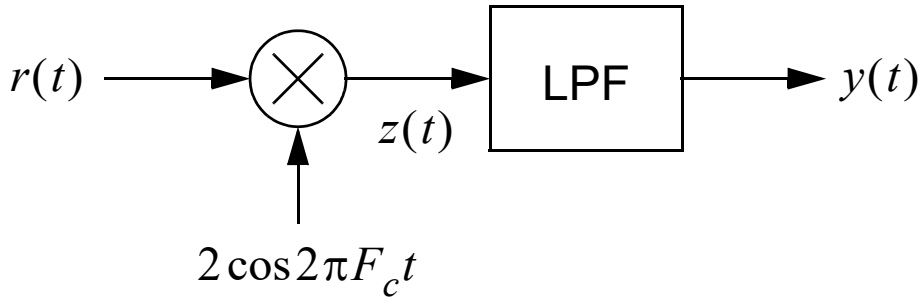
- To properly create an analytic signal using an HT FIR filter also requires that the real part be delayed by $N/2$ (assume N is even, number of taps is odd)



- Once we have the analytic signal we can obtain the complex baseband signal via (10.22)

System Application: A DSP Based Costas Loop for Coherent Carrier Recovery

- In the BPSK example given earlier, the receiver requires knowledge of the carrier frequency and phase to perform coherent demodulation
- Ideally we have:



- Assuming no noise is present

$$r(t) = a(t)\cos 2\pi F_c t \tag{10.26}$$

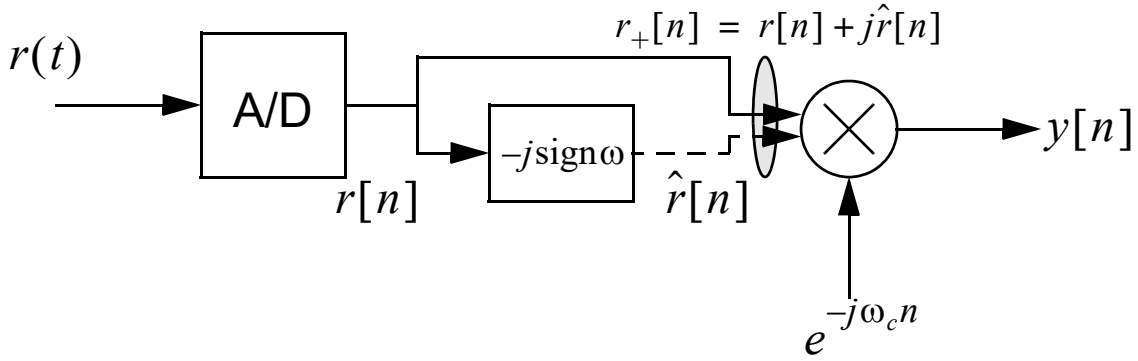
and

$$\begin{aligned} z(t) &= 2a(t)\cos^2(2\pi F_c t) \\ &= a(t) + a(t)\cos[2\pi(2F_c)t] \end{aligned} \tag{10.27}$$

- Following the lowpass filter the $\cos[2\pi(2F_c)t]$ term is removed leaving

$$y(t) = a(t) \tag{10.28}$$

- Another approach is to first form the analytic signal and then multiply by the complex exponential $\exp(-j2\pi F_c t)$
- If this was being done in the discrete-time domain it would be as follows



- From (10.19)

$$\begin{aligned} r_+[n] &= a[n]\cos(\omega_c n) + ja[n]\sin(\omega_c n) \\ &= a[n]e^{j\omega_c n} \end{aligned} \tag{10.29}$$

so

$$y[n] = r_+[n]e^{-j\omega_c n} = a[n] \tag{10.30}$$

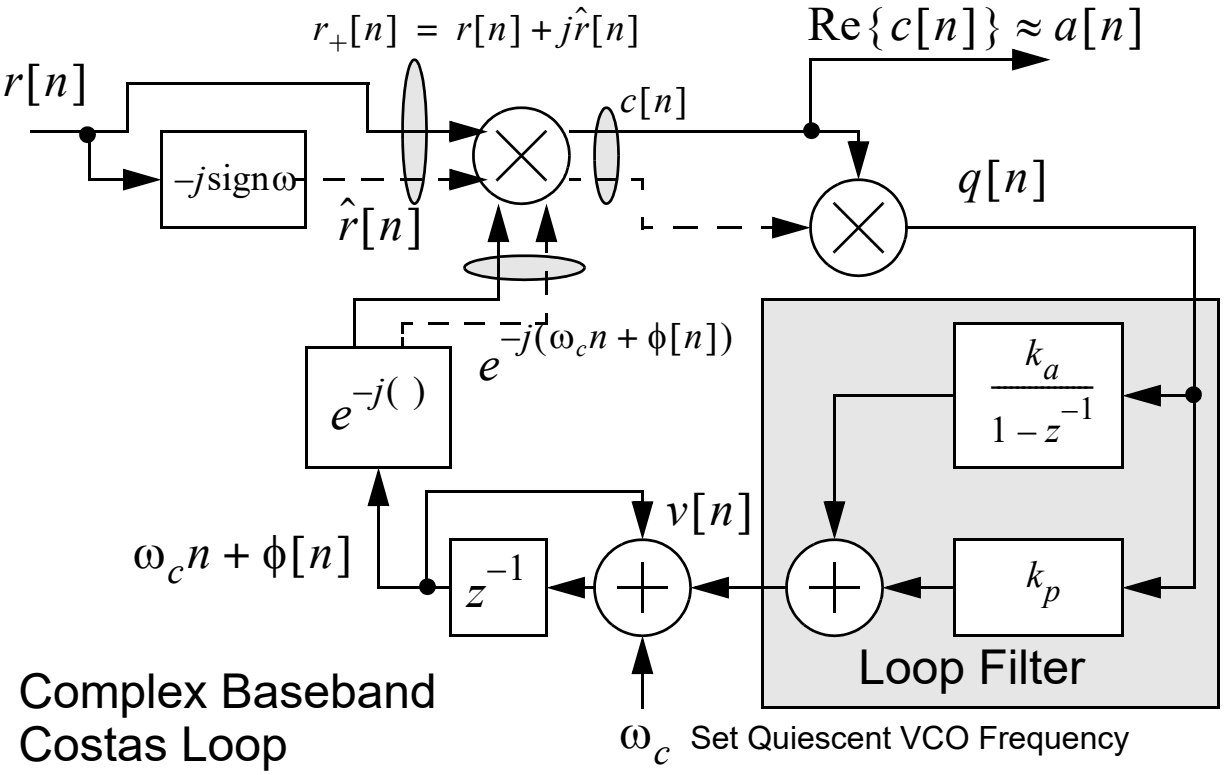
- A more practical model for $r[n]$ is

$$r[n] = a[n]\cos(\omega_c n + \theta[n]) \tag{10.31}$$

where ω_c is still the nominal carrier frequency (in the DSP domain), and $\theta[n]$ is a constant or slowly varying phase

– Note: $\theta[n]$ may also include a frequency error, e.g., $\theta[n] = \Delta\omega n = 2\pi\Delta F/F_s \times n$

- A feedback control system for tracking $\theta[n]$ is the Costas Loop (shown here as a DSP based implementation)



- To understand the operation of the Costas carrier tracking loop we begin by writing out $r_+[n]$ and then $c[n]$

$$r_+[n] = a[n]e^{j(\omega_c n + \theta[n])} \tag{10.32}$$

$$\begin{aligned} c[n] &= r_+[n]e^{-j(\omega_c n + \phi[n])} \\ &= a[n]e^{j(\theta[n] - \phi[n])} \end{aligned} \tag{10.33}$$

- The demodulated data signal is the real part of $c[n]$

$$\mathrm{Re}\{c[n]\} = a[n]\cos(\theta[n] - \phi[n]) \approx a[n] \tag{10.34}$$

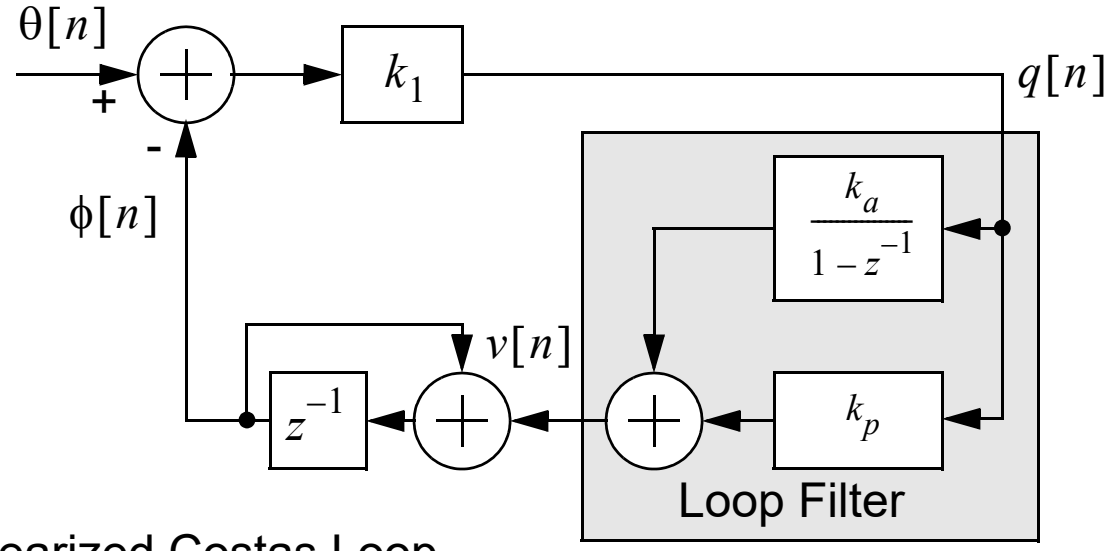
for $\theta[n] - \phi[n] \approx 0$

- The control signal, $q[n]$, for the Costas loop is the product of the real and imaginary parts of $c[n]$

$$\begin{aligned} q[n] &= a^2[n]\cos(\theta[n]-\phi[n])\sin(\theta[n]-\phi[n]) \\ &= 0.5a^2[n]\sin\{2(\theta[n]-\phi[n])\} \end{aligned} \tag{10.35}$$

- For a small phase error, $|\theta[n]-\phi[n]| \ll 1$, we can linearize the control loop since

$$q[n] \cong a^2[n](\theta[n]-\phi[n]) \tag{10.36}$$



Linearized Costas Loop

  - In the linearized model, the lowpass filtering action of the closed-loop response allows us to replace $a^2[n]$ with $k_1 = E\{a^2[n]\}$, which for BPSK is just one

- The closed-loop system performance can be obtained from the system function

$$H(z) = \frac{\Phi(z)}{\Theta(z)} \tag{10.37}$$

- The loop filter chosen here is second-order and is presently in the form of an accumulator (integrator) in parallel with a gain or proportional block

$$\frac{V(z)}{Q(z)} = k_p + \frac{k_a}{1-z^{-1}} = (k_p + k_a)\frac{\left(1 - \frac{k_p}{k_p + k_a}z^{-1}\right)}{1-z^{-1}} \qquad (10.38)$$

- The equivalent to a voltage controlled oscillator (VCO) is the accumulator with input $v[n]$ and output $\phi[n]$

$$\frac{\Phi(z)}{V(z)} = \frac{z^{-1}}{1-z^{-1}} \qquad (10.39)$$

- Closing the loop we have

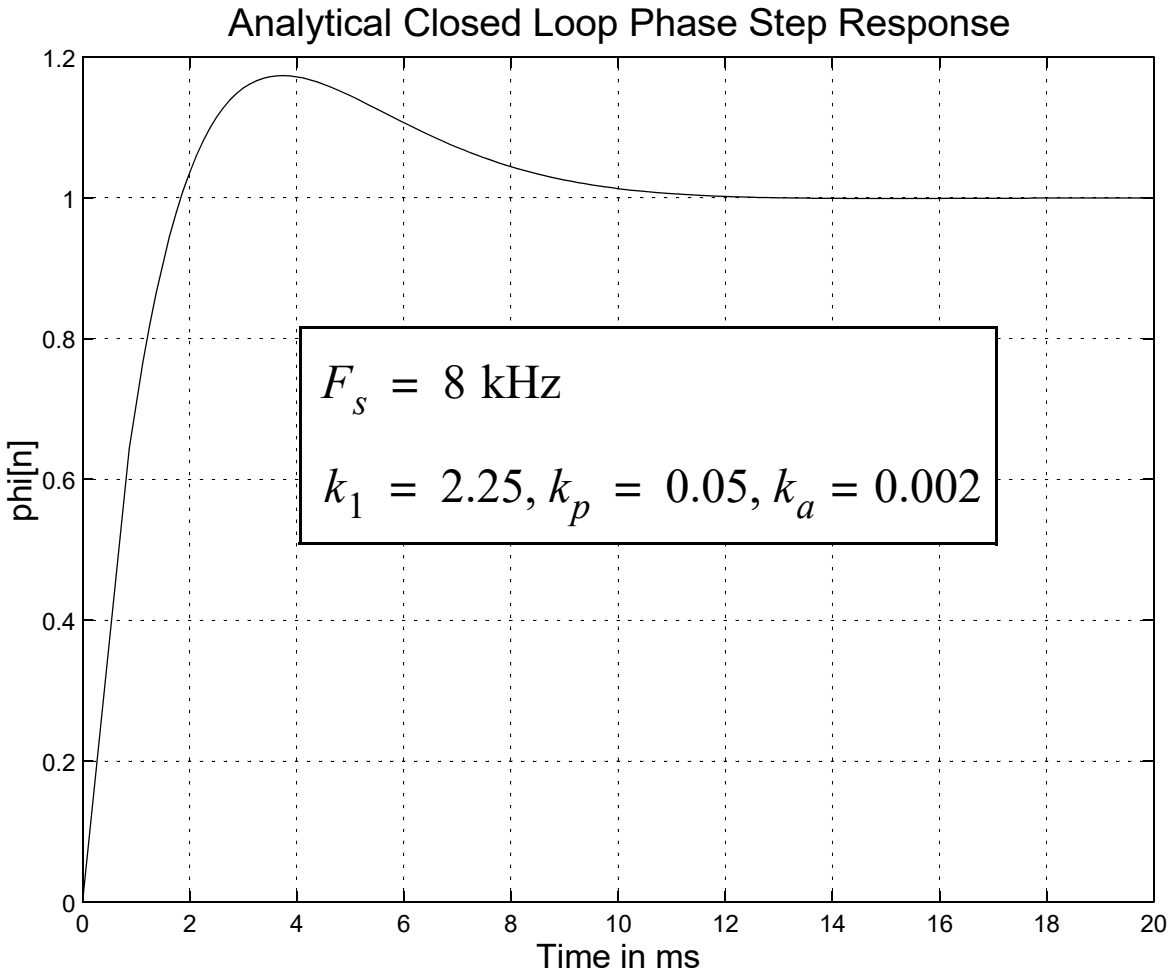
$$\Phi(z) = [\Theta(z) - \Phi(z)]k_1(k_p + k_a)z^{-1}\frac{\left(1 - \frac{k_p}{k_p + k_a}z^{-1}\right)}{(1-z^{-1})^2} \qquad (10.40)$$

or finally,

$$H(z) = \frac{k_1(k_p + k_a)z^{-1}\left(1 - \frac{k_p}{k_p + k_a}z^{-1}\right)}{(1-z^{-1})^2 + k_1(k_p + k_a)z^{-1}\left(1 - \frac{k_p}{k_p + k_a}z^{-1}\right)} \qquad (10.41)$$

$$= \frac{k_1(k_p + k_a)z^{-1} - k_1k_pz^{-2}}{1 - [2 - k_1(k_p + k_a)]z^{-1} + (1 - k_1k_p)z^{-2}}$$

- Assuming a sampling frequency of 8 kHz the analytical closed loop response step response is obtained using MATLAB



MATLAB Simulation

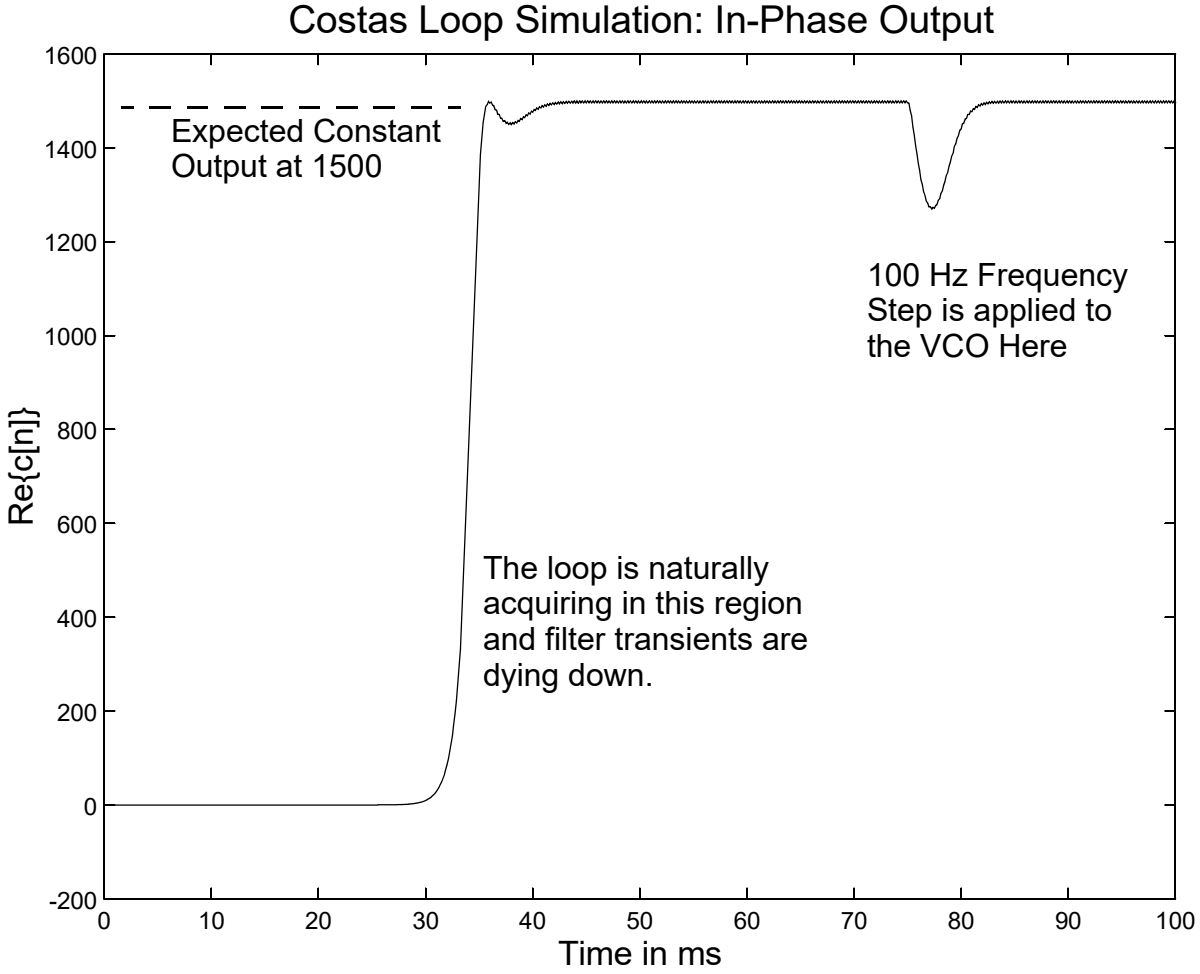
- A complete simulation of the loop using an $M = 30$ Hilbert transforming FIR filter is constructed to verify proper loop performance with simulated signals
- For the simulation an unmodulated carrier at 2000 Hz with amplitude 1500 is input to the system
  - Note: 1500 is a representative amplitude for digitized sinu-

soids once inside the DSK

```
% Costas Loop Simulation
%
% Mark Wickert 5/11/98

Fs = 8000;   % Sampling Frequency
F0 = 2000;   % Carrier frequency
N = 1000;    % Number of simulation points
kp = 0.05;   % Loop filter proportional gain
ka = 0.002;  % Loop filter accumulator gain

b=remez(30,[.1 .9],[1 1],'Hilbert'); %Hilbert 31-tap FIR

n = 0:N;
%Generate carrier at F0 with amplitude similar to DSK
%x = 1500*cos(2*pi*50/Fs*n).*cos(2*pi*F0/Fs*n);%with/DSB Mod.
x = 1500*cos(2*pi*F0/Fs*n); %Unmodulated carrier
%Generate analytic signal
xa = filter([zeros(1,15),1],1,x) + j*filter(b,1,x);

q = 0; q_old = 0; v = 0; v_old = 0;
accum = 0; %NCO accumulator
out = zeros(1,N+1) + j*zeros(1,N+1);
out_i = zeros(1,N+1);
q = zeros(1,N+1);
v = zeros(1,N+1);
%Begin simulation loop to model feedback system
for i=1:N+1
   out(i) = xa(i)*exp(-j*accum);
   out_i(i) = real(out(i));
   q(i) = real(out(i))*imag(out(i));
   q(i) = q(i)*1e-6;
   %Loop filter
   v(i) = v_old + (kp+ka)*q(i) - kp*q_old;
   %Update filter states
   v_old = v(i); q_old = q(i);
   %Apply a frequency step at 600 samples &
   %Wrap the phase in accumulator
   if i <= 600
      accum = mod(accum + 2*pi*F0/Fs + v(i),2*pi);
```

```
   else
      accum = mod(accum + 2*pi*(F0-100)/Fs + v(i),2*pi);
   end
end %Simulation loop
```

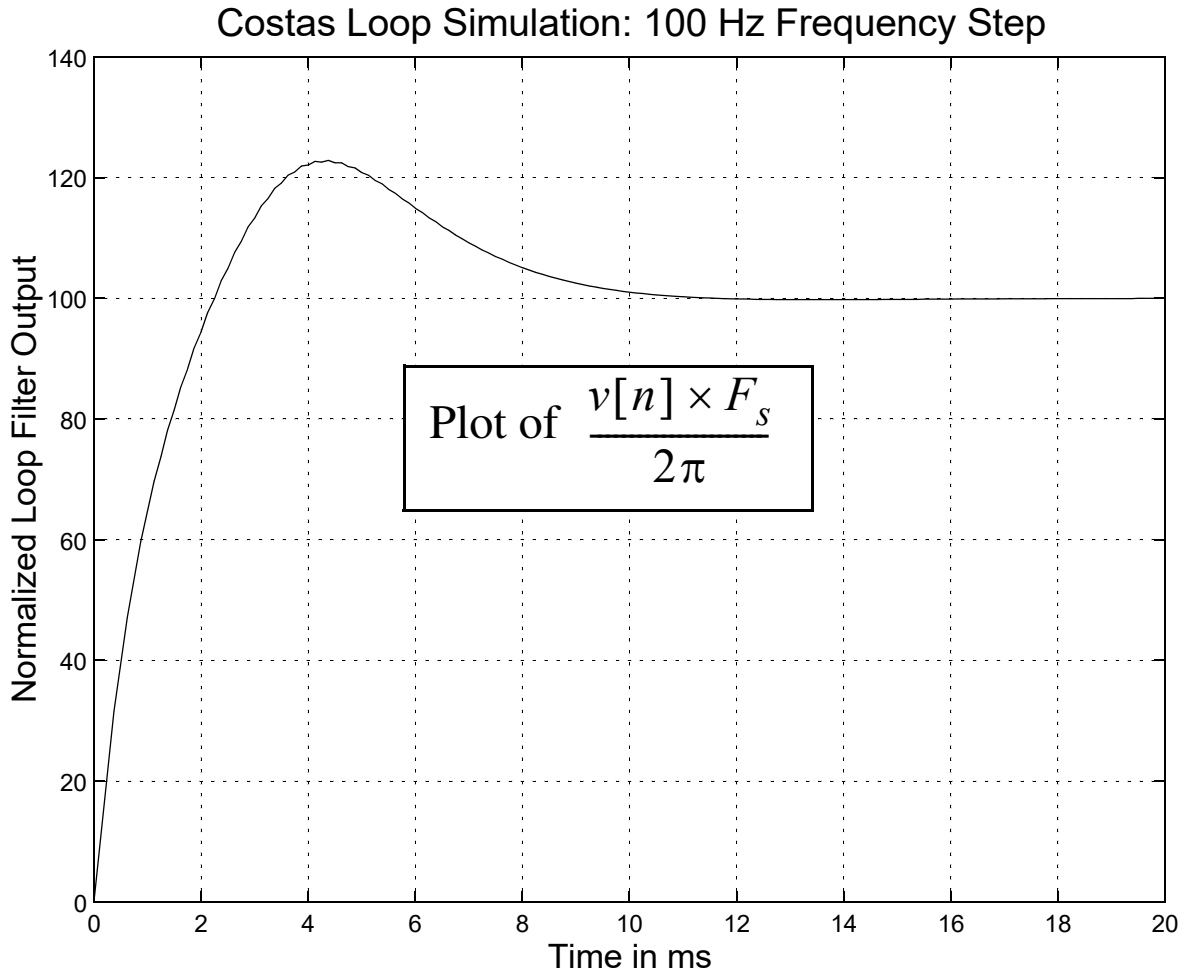
- A nominal sampling rate of 8 kHz is assumed
- A VCO quiescent frequency of 2000 Hz is used to allow the loop to naturally acquire phase lock
- At 600 samples into the simulation the VCO quiescent frequency is shifted down by 100 Hz
- The recovered modulation, in this case a constant is shown below



- The loop control signal $v[n]$ response with a corresponding

positive step to drive the loop back into lock

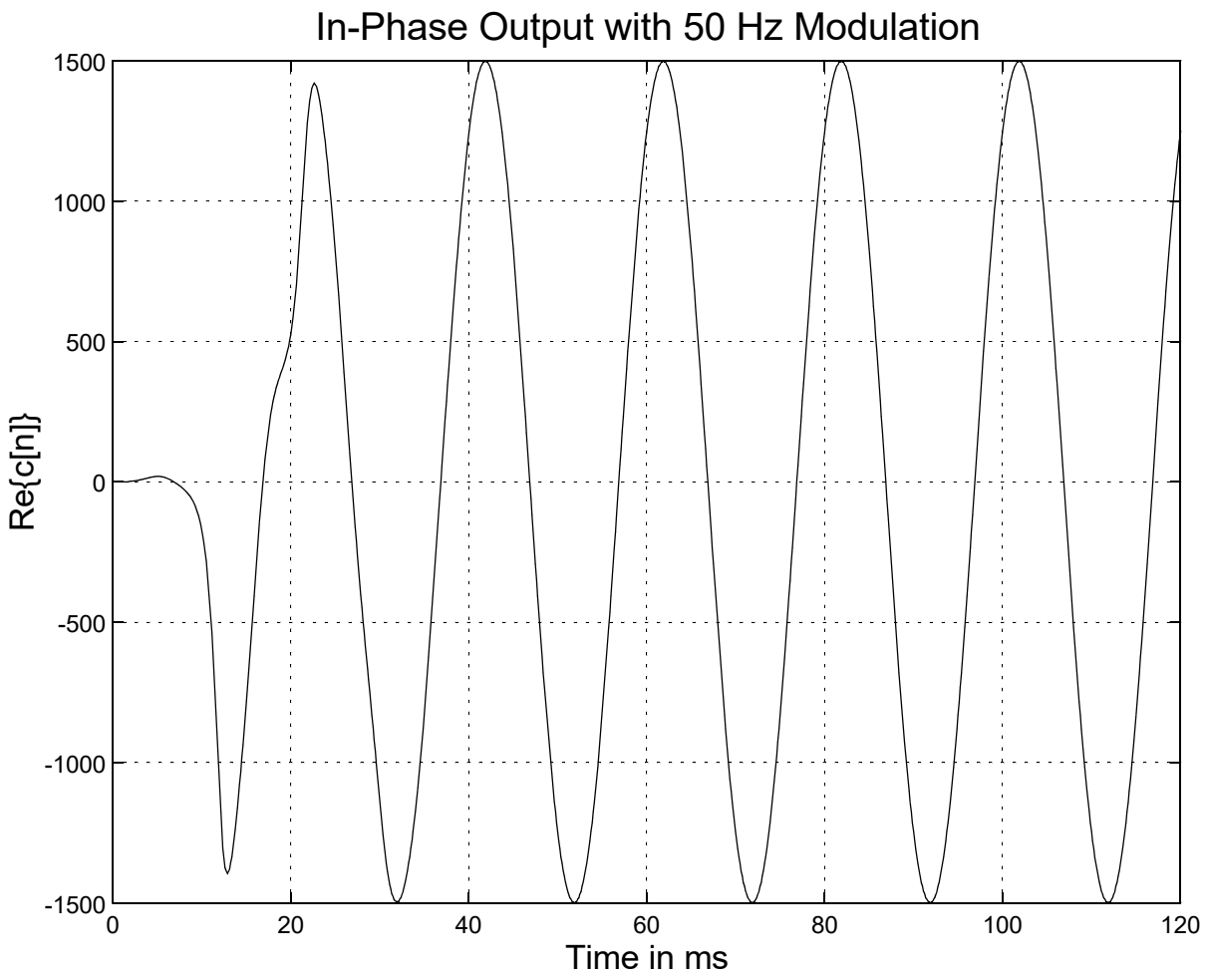
- The signal $v[n]$ scaled to represent frequency and time shifted so zero corresponds to the start of the 100 Hz step is shown below



- If the unmodulated carrier is replaced by a 50 Hz sinusoidal modulation carrier, and the loop is given an initial 0.1 Hz step to aid acquisition, e.g.,

```
if i <= 1000
    accum = mod(accum + 2*pi*(F0+.1)/Fs + v(i),2*pi);
```

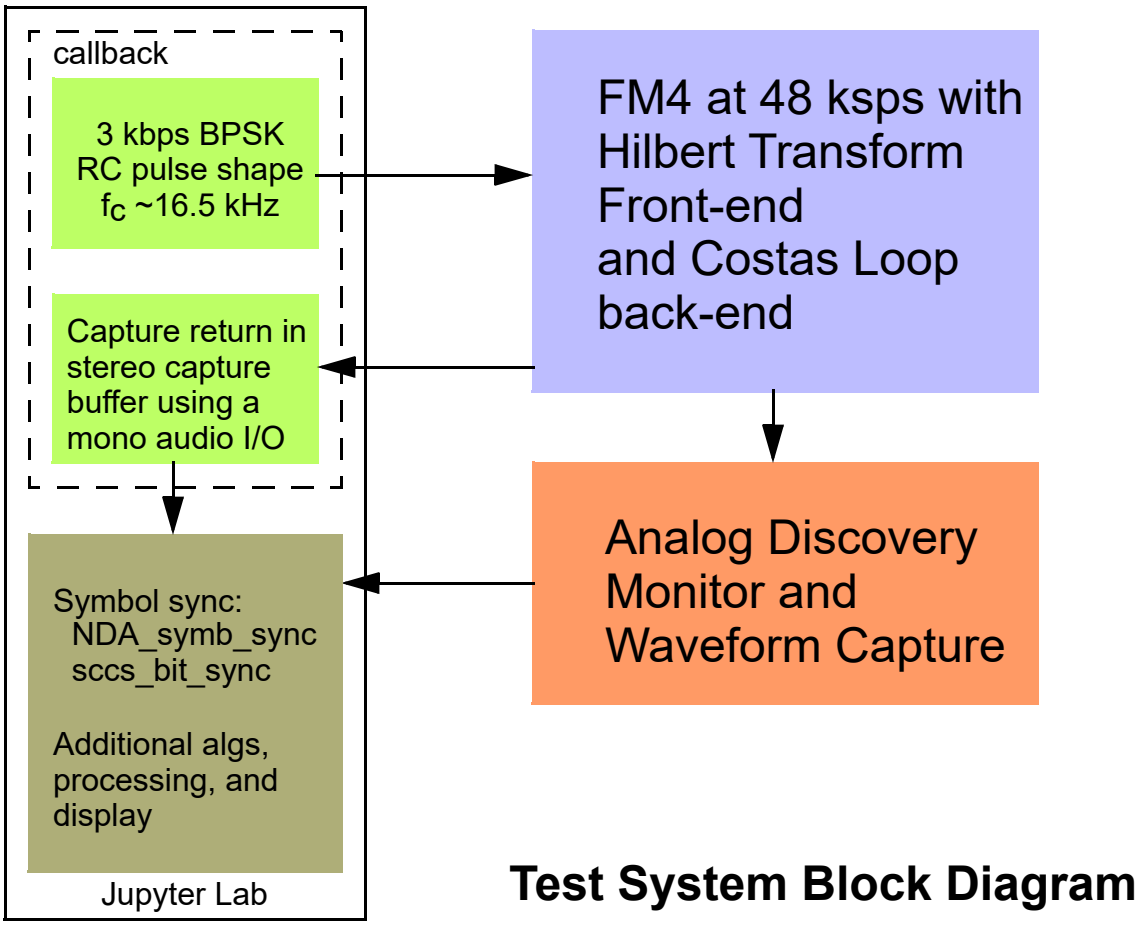
the following output is obtained



FM4 Costas Loop Example

Here we develop an FM4-based version of the Costas loop for carrier recovery on a 3 kbps BPSK at ~16.5 kHz. The sampling rate is fixed at 48ksps. The test configuration is Python centric with `pyaudio_helper` being used to generate the transmit signal. The pyaudio_helper callback also captures the FM4 output waveform so that symbol synchronization be performed in Python and the recovered 63-bit PN data stream can checked for bit errors at high SNR.

The system block diagram is shown below:



Test System Block Diagram

Transmitter using pyaudio_helper

- We want to transmit shaped BPSK with f_s = 48 ksps. The desired bit rate is 3 kbps, so we choose to have 16 samples per bit in the signal generation
- To maintain continuity with the pulse shaping filter we will maintain a separate transmit filter than can update its states after each PyAudio frame is processed
- The Jupyter notebook `5655_Costas_Project.ipynb` contains the all of the special code and the pyaudio_helper callback

and widget code

- In detail the transmitter uses a 63-bit long PN code to create a repeating message stream
- The bit stream is upsampled by 16 and passed through a raised cosine (RC) pulse shaping filter
- The filter excess bandwidth is $\alpha = \beta = 0.25$ and duration ± 6 bits
- The signal stream modulates a DDS carrier source outputting $\cos(2\pi(f_c/f_s)n)$
- Since pyaudio_helper is a frames-based signal processing architecture the DDS phase accumulator maintains phase continuity
- The states of the pulse shaping filter are properly maintained using the capabilities of `scipy.signal.lfilter`

- Check for available audio devices:

```
pah.available_devices()
```

```
{0: {'name': 'iMic USB audio system', 'inputs': 2, 'outputs': 2},
 1: {'name': 'MacBook Pro Microphone', 'inputs': 1, 'outputs': 0},
 2: {'name': 'MacBook Pro Speakers', 'inputs': 0, 'outputs': 2}}
```

- Define the GUI widgets

```
# ipywidgets code
Playback_gain = widgets.FloatSlider(description = 'PB Gain',
                 continuous_update = True,
                 value = 20.0,
                 min = 0.0,
                 max = 30.0,
                 step = 0.01,
                 orientation = 'horizontal')
f_c = widgets.FloatSlider(description = 'Carrier',
                 continuous_update = True,
```

```
                value = 16500.0,
                min = 0.0,
                max = 18000.0,
                step = 10.0,
                orientation = 'horizontal')
widgets.HBox([Playback_gain, f_c])
```

- Initialize global variables

```
# Initialization for pyaudio_helper code
# Generate PN bits
pn6 = ss.m_seq(6)
# Use 16 samples per bit
Ns = 16
x_pn6, b = ss.nrz_bits2(pn6,Ns,'rc')
zi = signal.lfiltic(b,1,[0])
f_c = 16500
DDS1 = DDS(f_c,48000)
```

- Define the callback used to process real-time audio frames

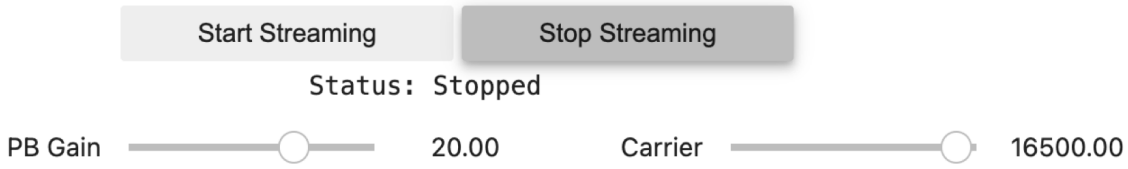
```
# define callback (3)
# Here we configure the callback to play back a wav file
def callback(in_data, frame_count, time_info, status):
    global DSP_IO, DDS1, pn6, zi, b, f_c
    DSP_IO.DSP_callback_tic()

    # convert byte data to ndarray
    in_data_nda = np.frombuffer(in_data, dtype=np.int16)
    # Ignore in_data when generating output only
    #***********************************************
    x_rx = in_data_nda.astype(float32)
    x, zi = signal.lfilter(b,1,ss.upsample(2*pn6-1,16),zi=zi)
    DDS1.set_fcenter(f_c.value)
    # Form the carrier array
    carrier = zeros(frame_count)
    for n in range(frame_count):
        carrier[n] = 10000*DDS1.output_cos()
        DDS1.update(0.0)
    y = Playback_gain.value*x*carrier
    # Note wav is scaled to [-1,1] so need to rescale to int16
    #y = 32767*x.get_samples(frame_count)
    # Perform real-time DSP here if desired
    #
    #***********************************************
    # Save data for later analysis
    DSP_IO.DSP_capture_add_samples_stereo(y,x_rx)
    # accumulate a new frame of samples
    DSP_IO.DSP_capture_add_samples(y)
    #***********************************************
```

```
    # Convert from float back to int16
    y = y.astype(int16)
    DSP_IO.DSP_callback_toc()
    return y.tobytes(), pah.pyaudio.paContinue
```

  - The above callback uses a mono audio stream, but it allows both the output and input waveforms to be logged in stereo capture buffer `DSP_IO.DSP_capture_add_samples_stereo(y,x_rx)`

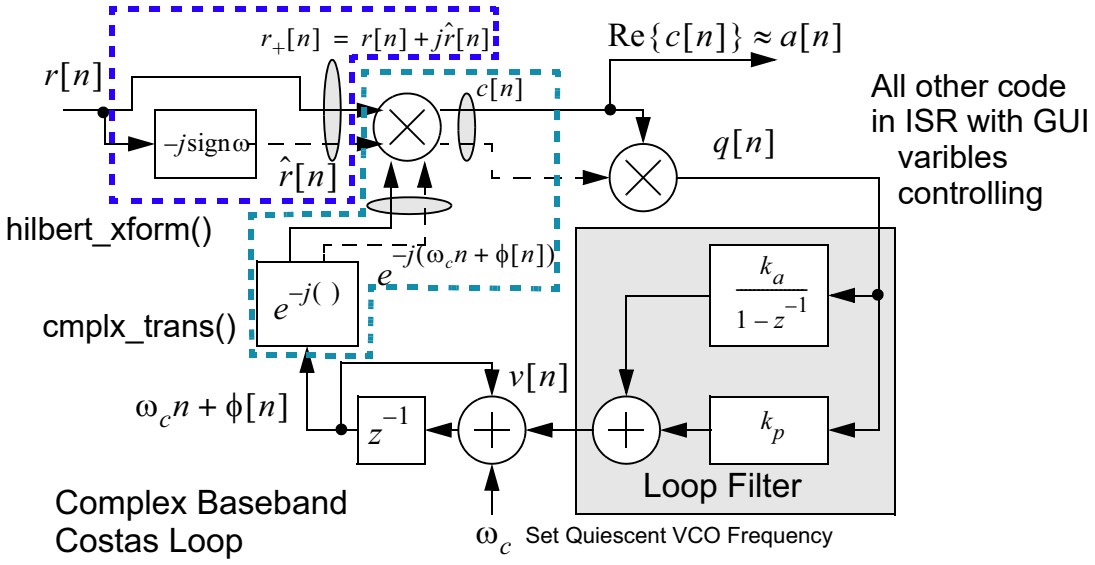
- Create the DSP_IO object, run it, and display the widgets

```
# Set up the stream and then start it
fsamp = 48000
frame_length = 1008 #496 # = (2**5 - 1)*16
T_cap = 0
DSP_IO = pah.DSP_io_stream(callback,0,0,frame_length=frame_length,
                           fs=fsamp,Tcapture=T_cap)
DSP_IO.interactive_stream(Tsec=T_cap)
widgets.HBox([Playback_gain, f_c])
```



FM4 Receiver Code

- To quickly review, the C-code implements:



- On the FM4 the C-code is modularized to some extent
  - The function `hilbert_xform()` encapsulates the analytic signal formation
  - The function `complex_trans()` encapsulates complex frequency translation which is the heart of the carrier phase tracking and conversion to baseband I/Q signal
  - All of the remaining function blocks are implemented in the body of the ISR for processing signal samples
- The code listing is:

```
// costas_main

#include "fm4_wm8731_init.h"
#include "FM4_slider_interface.h"

#include "Hilbert_bpf_31_s18.h" // Hilbert 31 and delay by 15 coefficients


int rand_int(void);

float32_t Fs = 48000; //32000; //8000.0 // desired Fs

// Some global variables
float32_t DLY[M_HFIR]; //buffer for delay samples
float32_t in, out_i, out_q;
float32_t accum, q, q1, q1_old, v;
float32_t twopi = 6.283185307179586;
float32_t twopiT = 0.0001308996938995747;
float32_t kv, kp, ka;
int16_t s_indx;
float32_t samples[128];
int16_t select_output, select_buffer;
float32_t noise_level;

void hilbert_xform()
{
    int i;
    float32_t acc;
    acc = 0.0;
    DLY[0] = in;

    for (i=0; i<M_HFIR; i++)
    {
        acc += h_Hil[i] * DLY[i];
```

```
    }

    out_i = DLY[15];
    out_q = -acc;

    for (i=M_HFIR-1; i>0; i--)
    {
      DLY[i] = DLY[i-1]; /*Update Buffers*/
    }

}

void cmplx_trans(float *arg)
{
    float32_t temp;
    float32_t cos_arg = cosf(*arg);
    float32_t sin_arg = sinf(*arg);
    temp = out_i;
   out_i = temp * cos_arg + out_q * sin_arg;
   out_q = out_q * cos_arg - temp * sin_arg;

    // See if NCO/DDS is working
    //out_i = 16000.0f * cos_arg;
    //out_q = -16000.0f * sin_arg;
}

// Create (instantiate) GUI slider data structure
struct FM4_slider_struct FM4_GUI;

// Analog I/O working variables
volatile int16_t audio_chR=0;    //16 bits audio data channel right
volatile int16_t audio_chL=0;    //16 bits audio data channel left

void PRGCRC_I2S_IRQHandler(void)
{
  union WM8731_data sample;
  int16_t xL, xR;
  int16_t yL, yR;

  gpio_set(DIAGNOSTIC_PIN,HIGH);
  // Get L/R codec sample
  sample.uint32bit = i2s_rx();

  // Breakout and then process L and R samples

  xL = sample.uint16bit[LEFT];
  xR = sample.uint16bit[RIGHT];
  in = (float32_t) xL;
  ////Use the next line add noise to the input signal
  //in += 0.05*noise_level*((int16_t) rand_int());
  /*Hilbert transform the input signal*/
  hilbert_xform();
  // POINT 1 AUDIO CAPTURE
  //xL = (int16_t) out_i;
```

```
  //xR = (int16_t) out_q;
  /*Multiply by the NCO/VCO Output exp(-j*accum)*/
  cmplx_trans(&accum);
  // POINT 2 AUDIO CAPTURE
  xL = (int16_t) (FM4_GUI.P_vals[0] * out_i);
  // xR = (int16_t) (FM4_GUI.P_vals[1] * out_q);
  /*Form Costas Loop Error Signal and Scale*/
  q = out_q * out_i * kv; // Reduce detector gain by kv immediately
    // Loop filter accumulator
    q1 = ka*q + q1_old;
    q1_old = q1;
  //Finish the loop filter by combiniing the proportional and accumulator
terms
  v = kp*FM4_GUI.P_vals[4]* q + ka*FM4_GUI.P_vals[5] * q1_old;
    //v = kp*FM4_GUI.P_vals[4]* q;
  //xL = (int16_t)(v*FM4_GUI.P_vals[1]*1000.0f);
  /*Drive the NCO/VCO with v[n]*/
  // Use twopi*FM4_GUI.P_vals[4]/fs as wc
  accum += FM4_GUI.P_vals[2]*twopiT + v;
  while (accum >= twopi) accum -= twopi;//
  // Use GUI controls to allow selection of various
  // output signal combinations. Also consider making
  // some of the loop filter parameters under slider control.
  // Always output out_i
  // With slider acting as a selector switch choose between:
  // (1) out_q, (2) v, and (3) in
  // Consider making kv variable and perhaps kp and/or ka//xL = (int16_t)
out_q;
  if (FM4_GUI.P_vals[3] == 3) {
    yL = (int16_t) (FM4_GUI.P_vals[1] * out_i);
       yR = yL;
    }
  else if (FM4_GUI.P_vals[3] == 2) {
    yL = (int16_t) (v*FM4_GUI.P_vals[1]*1000.0f);
       yR = xR;
    }
  else {
    yL = (int16_t) (FM4_GUI.P_vals[1] * out_i);
       yR = xR;
    }

  // Return L/R samples to codec via C union
    sample.uint16bit[LEFT] = yL;
    sample.uint16bit[RIGHT] = yR;
  i2s_tx(sample.uint32bit);

  NVIC_ClearPendingIRQ(PRGCRC_I2S_IRQn);

    gpio_set(DIAGNOSTIC_PIN,LOW);
}

int main(void)
{
    // Initialize the slider interface by setting the baud rate (460800 or
```

```
921600)
    // and initial float values for each of the 6 slider parameters
    init_slider_interface(&FM4_GUI,460800, 1.0, 1.0, 16500.0, 1.0, 1.0, 1.0);

    // Send a string to the terminal
    write_uart0("Hello FM4 World!\r\n");

    accum = 0.0;
    q1_old=0.0;
    v=0.0;
    //twopi = 8*atan(1);
    //twopiT = twopi/Fs; //Sampling Period
    // add slider to kv with slider from 1 to 100
    kv = 1e-9, kp = 13.282, ka = 0.00122; // see Jupyter notebook
    s_indx = 0;
    select_output = 1; //in-phase output select
    select_buffer = 1; //load accum into samples[] buffer
    noise_level = 0; //initialize noise level at 0
    //comm_intr(); //init DSK, codec, McBSP
    //audio_init (hz48000, line_in, intr, I2S_HANDLER);
    // Some #define options for initializing the audio codec interface:
    // FS_8000_HZ, FS_16000_HZ, FS_24000_HZ, FS_32000_HZ, FS_48000_HZ,
FS_96000_HZ
    // IO_METHOD_INTR, IO_METHOD_DMA
    // WM8731_MIC_IN, WM8731_MIC_IN_BOOST, WM8731_LINE_IN
  fm4_wm8731_init (FS_48000_HZ,                // Sampling rate (sps)
                   WM8731_LINE_IN,               // Audio input port
                   IO_METHOD_INTR,               // Audio samples handler
                   WM8731_HP_OUT_GAIN_0_DB,    // Output headphone Gain (dB)
                   WM8731_LINE_IN_GAIN_0_DB); // Line-in input gain (dB)

    while(1){ //infinite loop

        // Update slider parameters
        update_slider_parameters(&FM4_GUI);
    }
}
```

- To configure the loop parameters the phase detector gain needs to be determined
  - The GUI slider control allows scaling of k_p and k_a
  - The first step is calculating the parameters in Python

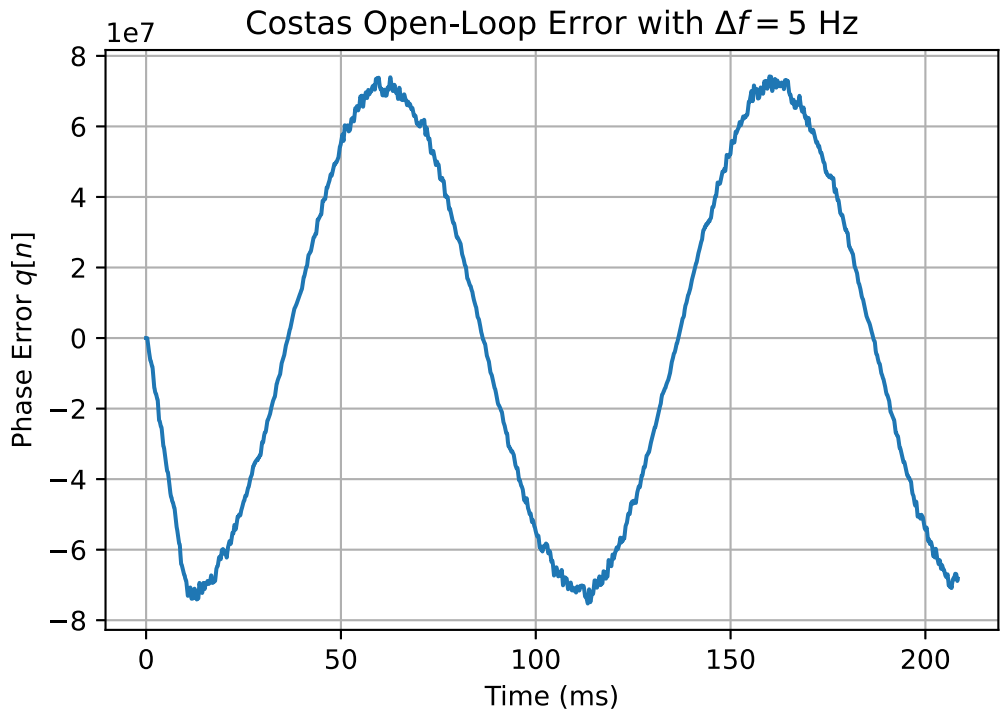
```
1 kd = 1.4e6 #
2 kc = 1.0
3 wn = 2*pi*100
4 zeta = 0.707
5 kp = (2*zeta*wn/fs + (wn/fs)**2/2)/kc/kd
6 ka = (wn/fs)**2/kc/kd
7 (kp/1e-9,ka/1e-9)
```

```
1 # Loop gain parameters kp and ka
2 (13.282064547653482, 0.122390927592874)
```

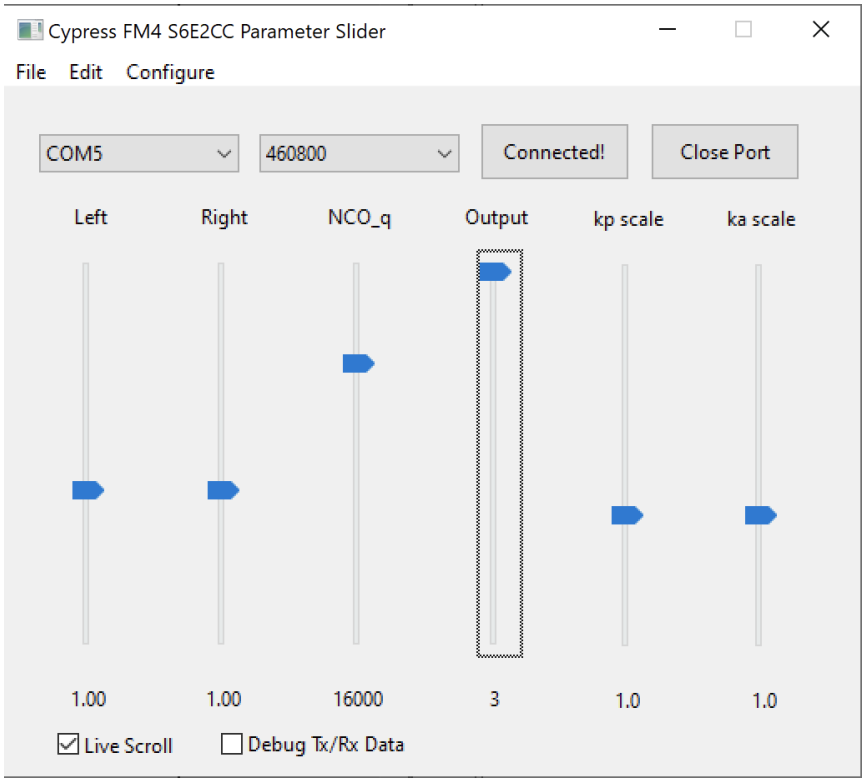
  - Note in this calculation we need k_d which is the phase detector gain

- An open-loop measurement is made using a synthesized measurement of $q[n]$ using a 5 Hz frequency error
- The signal below approximates the error signal $q[n]$ which takes the form $A_q \sin(2\theta)$, with θ the phase error (in general a function of time)

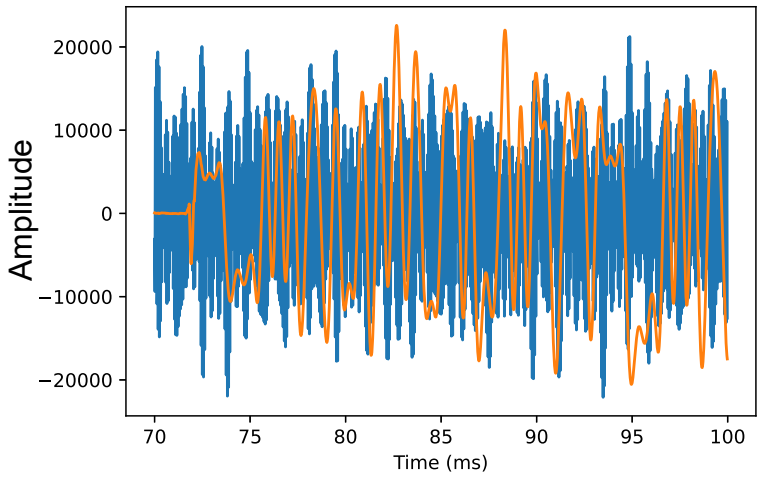
```
N_samps = 10000
fs_h = 48 # kHz for the BPF filter only, while the others effectively have fs = 1 Hz
b31_bpf = signal.remez(31,[0,10,15,18,23,fs_h/2],[0,1,0],weight=[10,100,10],
                    fs=fs_h,type='hilbert')
n = arange(N_samps)
x_analytic = signal.lfilter(hstack((zeros(15),ones(1))),1,x_tx[:N_samps]) \
                        - 1j*signal.lfilter(b31_bpf,1,x_tx[:N_samps])
x_DDC = x_analytic * exp(-1j*2*pi*16495/48000*n)
q = x_DDC.real * x_DDC.imag
plot(n/48,signal.lfilter(ones(500)/500,1,q))
title(r'Costas Open-Loop Error with $\Delta f = 5$ Hz')
ylabel(r'Phase Error $q[n]$')
xlabel(r'Time (ms)')
grid();
```



- From the above smoother open-loop phase error plot we estimate the slope at zero to be $k_d = 2A_q \approx 1.4\times10^6$ V/rad
- The FM4 GUI slider interface configuration:



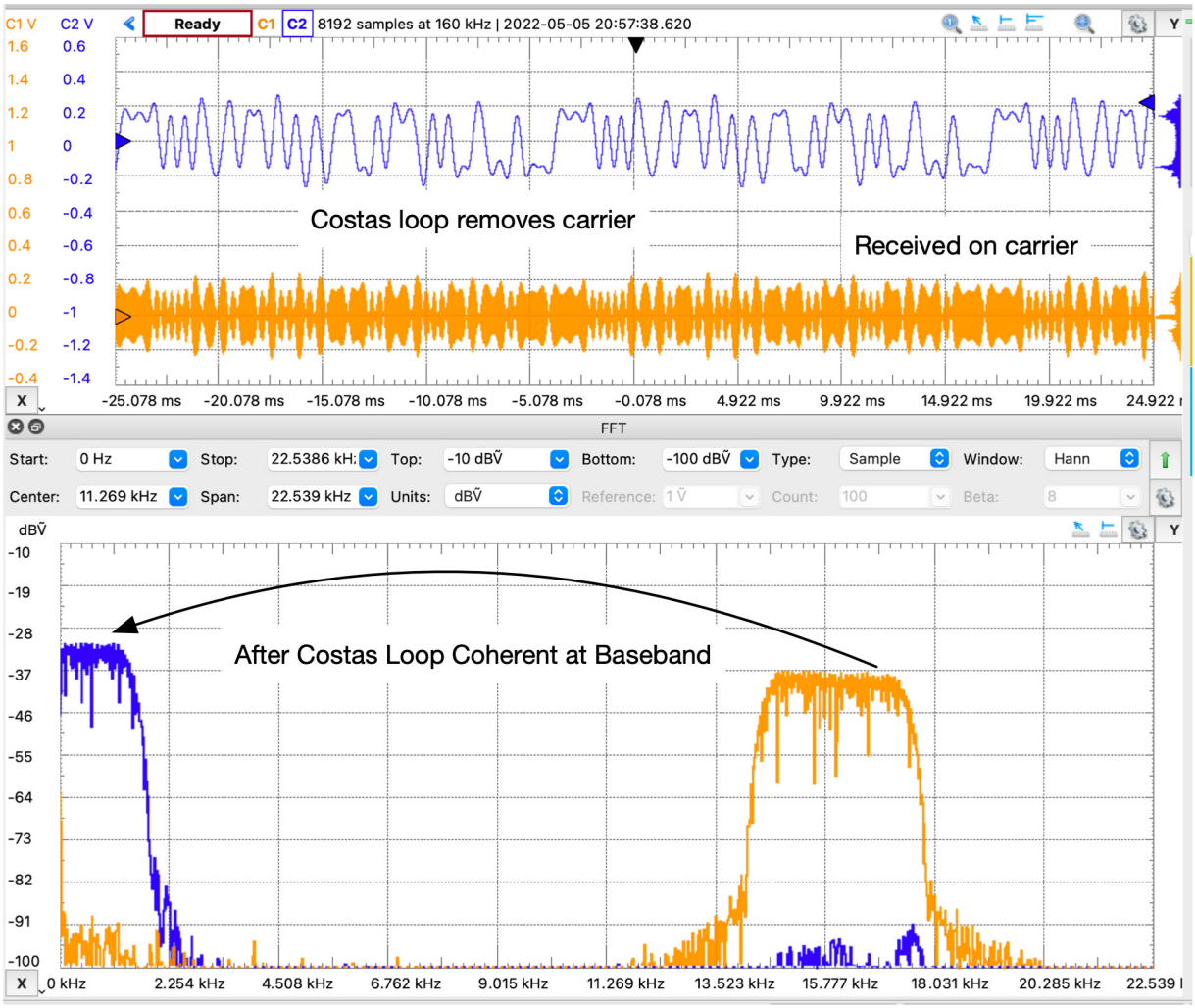
- **Quick Test**: Referring to the test system block diagram, when `pyaudio_helper` app is sending a signal to FM4 line-in jack and the FM4 headphone jack is returning a signal back to pyaudio we can see the following:



blue = input; 16.5kHz
orange = output; BB

Note: The latency on the return demodulated BPSK; this is due to pyaudio

- Analog Discovery view of the quick test:



- A complete simulation test in Python can be found in the Jupyter notebook sample; similar results were obtained
- A more detailed analysis of the returned baseband waveforms will now be done in Python

Post Processing in Python

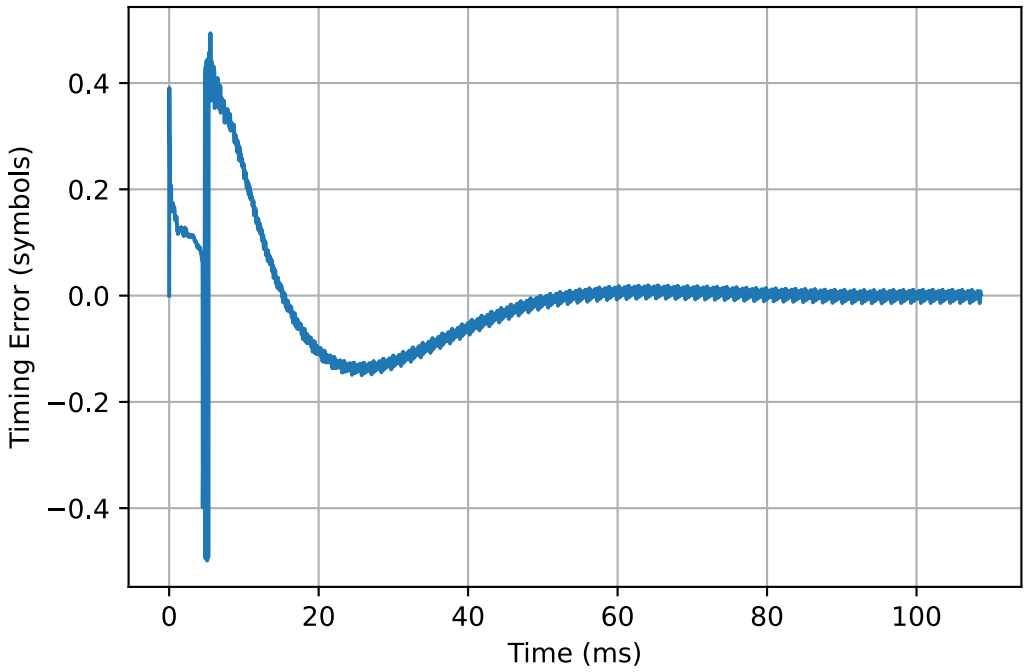
- The carrier phase being tracked as desired, the next step is to account for the asynchronous clocks by implmenting symbol synchronization

- In the future this can be attempted on the FM4
- Presently we will operate on the captured waveform in Python using tools from `scikit-dsp-comm`
  - Load the `synchronization` module & use `NDA_symb_sync`
  - Pass the

```
import sk_dsp_comm.synchronization as sync
```

```
zz,e_tau = sync.NDA_symb_sync(DSP_IO.data_capture_right,16,100,0.001)
```

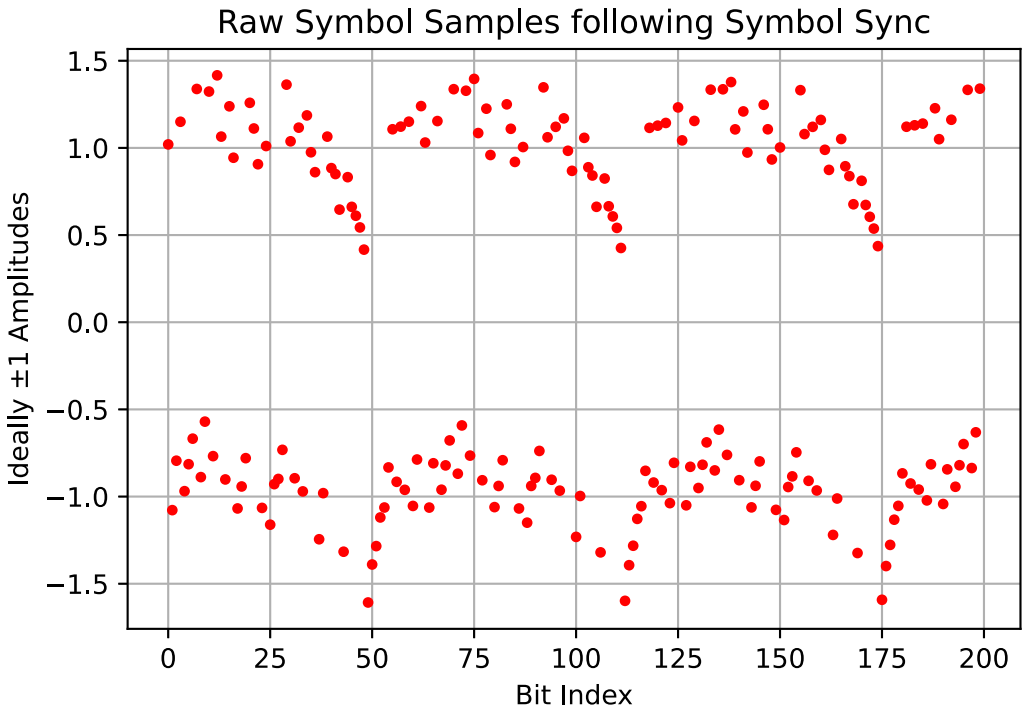
```
plot(arange(len(e_tau))/48000*1000,e_tau)
ylabel(r'Timing Error (symbols)')
xlabel(r'Time (ms)')
grid();
```



  - The loop bandwidth is very narrow so it becomes evident that clock jitter related to audio frame scheduling via PyAudio is not great

– Moving beyond 60 ms we consider the bit sync to be locked and settled

– Looking at the bit samples, `zz`, recovered from `NDA_symb_sync` we see this clearly (ideally ± 1):



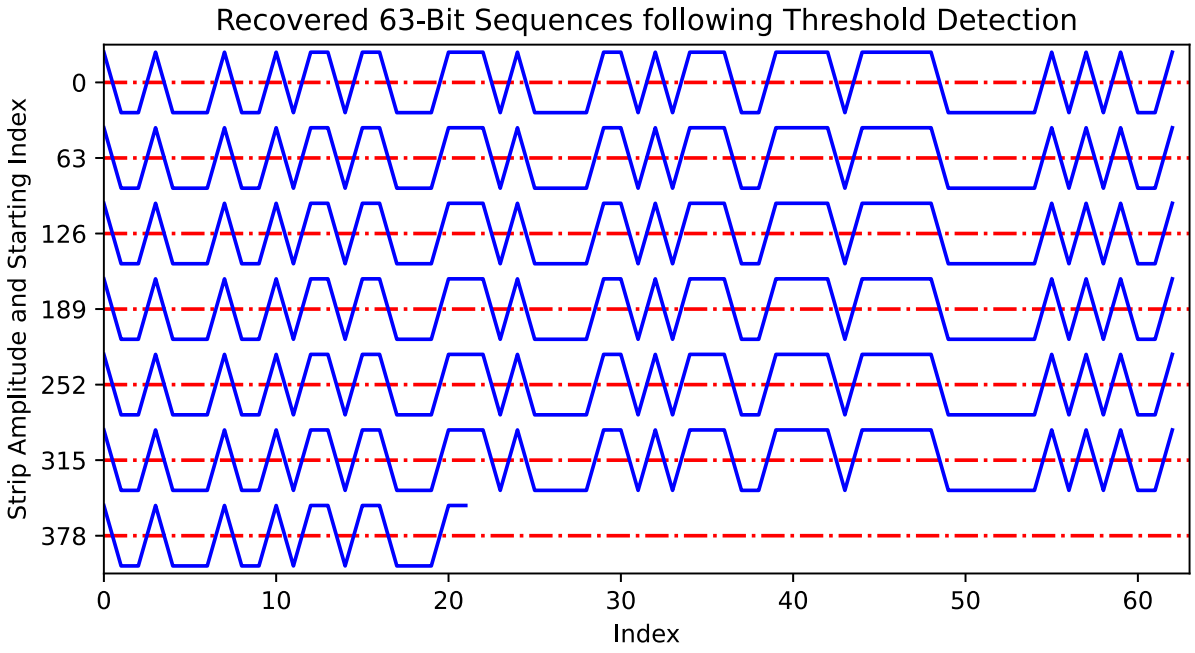
– In spite of the jitter we press forward to verify the PN63 bits are indeed recovered

– Here we use the `strips()` function which is in the module `digitalcom`:

```
import sk_dsp_comm.digitalcom as dc
```

```
dc.strips(sign(zz[2000:2400].real),63,fig_size=(8,4))
title(r'Recovered 63-Bit Sequences following Threshold Detection')
```

– From a 5s capture we consider just 2000 to 2400 samples and find under noise free conditions the PN63 is being recovered:



- An alternate and much simpler bit synchronizer is considered the Jupyter notebook sample
- In this case the capture was taken using the Analog Discovery