Lab 6: PAM Receiver with Matched Filter and Symbol Timing Extraction

1 Introduction

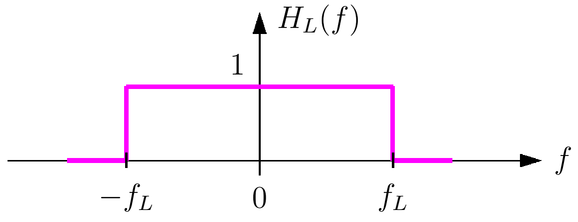
Communication without noise would be trivial. You could take the text of a whole encyclopedia, encode it in ASCII, and make a long binary string by concatenating the resulting bits. To transmit this string as a single symbol you would interpret it as the binary representation of a number which determines the amplitude of a voltage or current pulse sent by the transmitter. The receiver would then convert the received amplitude back into a number whose binary representation is the received bit string. If the bit string is 7 bits long (from one ASCII character), then the receiver needs to distinguish 128 different amplitude levels, and the scheme actually works quite well. But if a whole text results in 1 million bits, say, then the receiver needs to be able to distinguish $2^{1,000,000} \approx 10^{300,000}$ levels, which is impossible in the presence of the slightest amount of noise. Thus, there is no other choice than to transmit a long text as a sequence of smaller numbers and, to minimize the probability of error, to build the receiver front-end such that it maximizes the signal-to-noise ratio (SNR) for each reception. An associated problem that comes from transmitting sequences of numbers rather than a single number is that the receiver must synchronize itself to the transmitter. In practice this means in most cases that symbol timing information must be extracted from the received signal waveform itself.

1.1 Trapezoidal Lowpass Filter

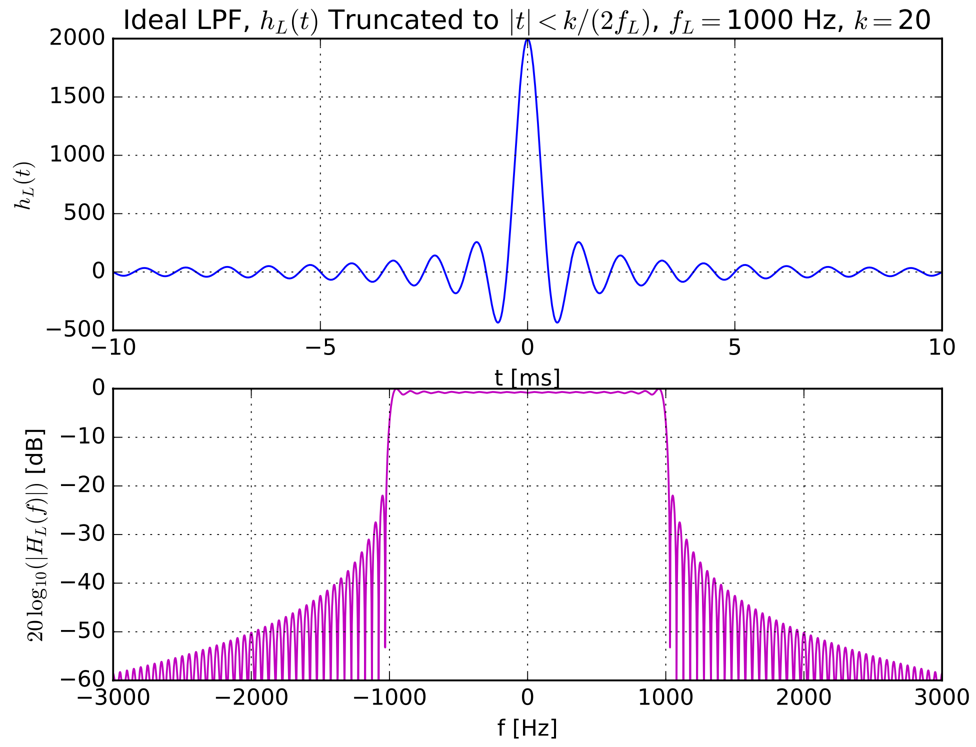
Lowpass filters (LPF) are used for many different tasks in communication systems. Among them are the separation of a desired signal from noise and interference and symbol timing extraction at the receiver.

The impulse response $h_L(t)$ and the frequency response $H_L(f)$ of an ideal LPF with cutoff frequency f_L are shown in the figure below.

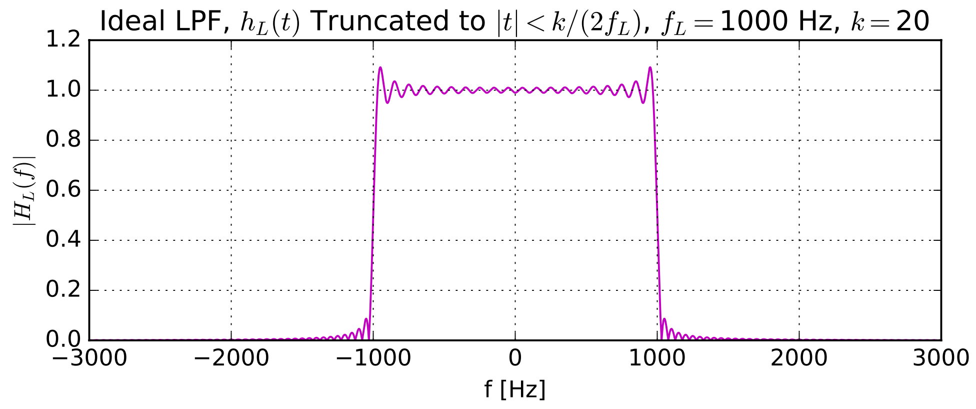
$$h_L(t) = \frac{\sin(2\pi f_L t)}{\pi t} \quad \Longleftrightarrow$$



For practical applications the problem with the ideal LPF is that $h_L(t)$ needs to be truncated, e.g., to the time interval $-k/(2f_L) \leq t \leq k/(2f_L)$ for some integer k, which leads to substantial sidelobes in the frequency response whenever k is finite. The following plots of $h_L(t)$ and $20\,\log_{10}(|H_L(f)|)$ (in dB) show an example for $f_L = 1000$ Hz and $k = 20$.

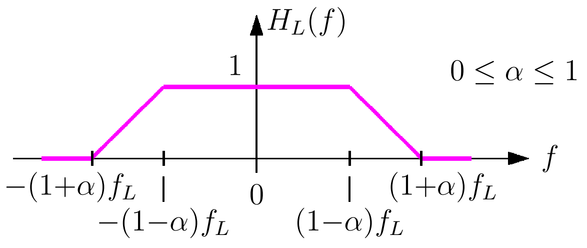


If the magnitude of the frequency response $|H_L(f)|$ is displayed on a linear scale, we see a ripple in both the passband and the stopband of the filter as a result of the truncation of $h_L(t)$.

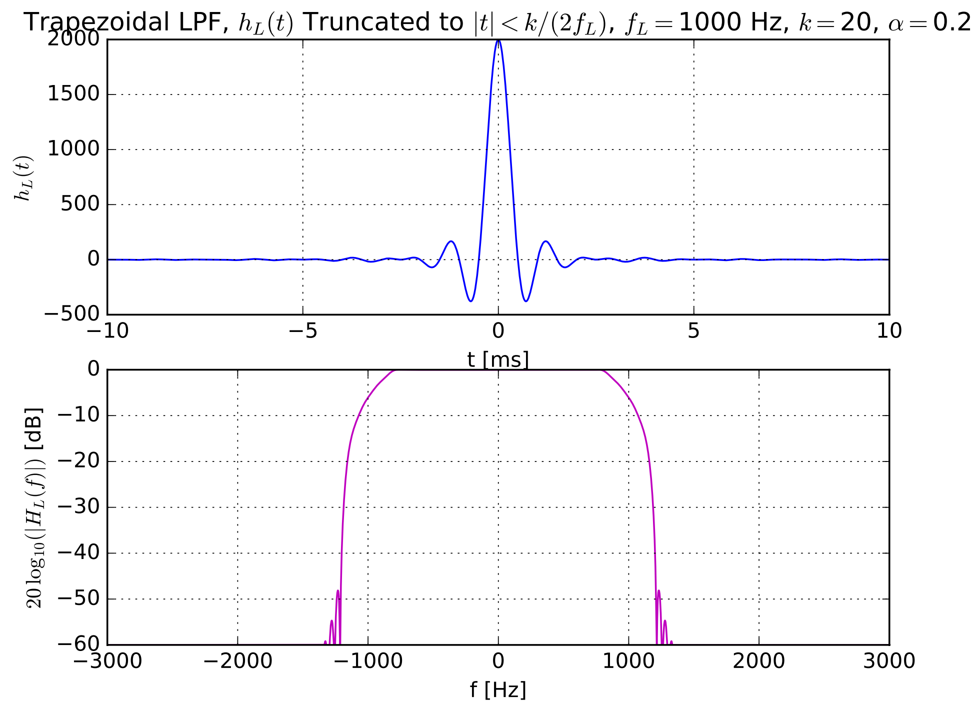


By making the transition from the passband to the stopband less abrupt, the situation can be improved considerably. One possibility is to use an LPF with a trapezoidal frequency response as shown next.

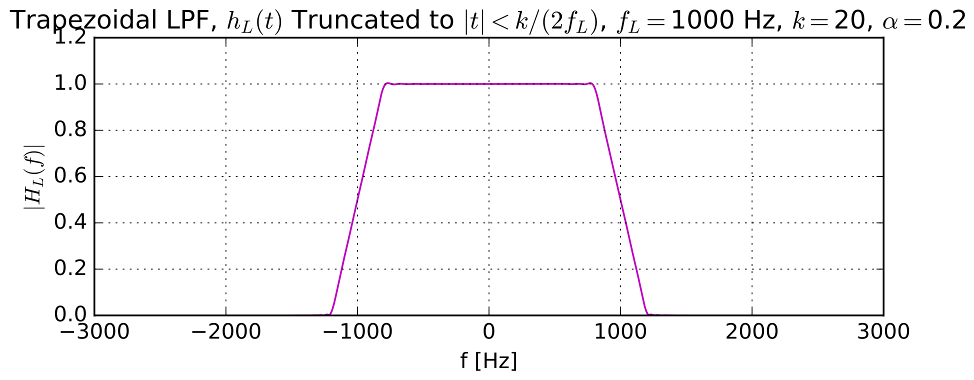
$$h_L(t) = \frac{\sin(2\pi f_L t)}{\pi t}\,\frac{\sin(2\pi \alpha f_L t)}{2\pi \alpha f_L t} \quad \Longleftrightarrow$$



As the parameter α is varied from 0 to 1, the frequency response goes from an ideal LPF to a $H(f)$ with triangular shape. An example with $f_L = 1000$ Hz, $k = 20$, and $\alpha = 0.2$ is shown below.

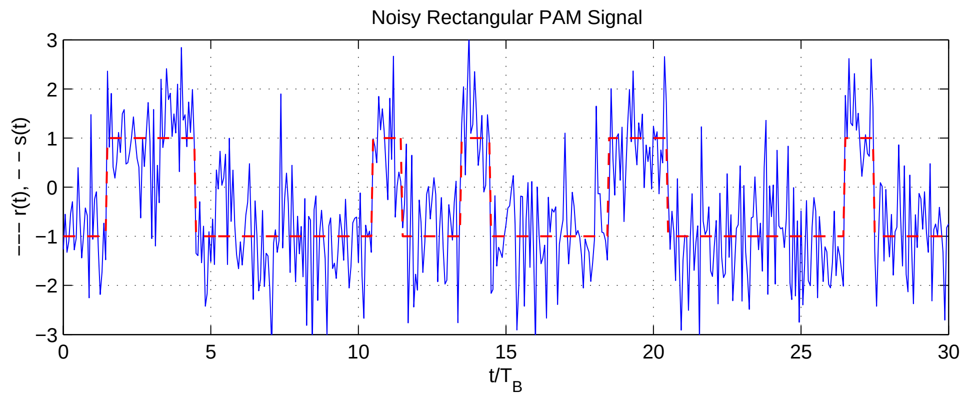


An additional advantage of a LPF with trapezoidal frequency response is that its transition region around f_L can be used as a frequency discriminator, i.e., as a device that converts linearly from frequency to amplitude. A linear plot of the magnitude of the frequency response of the same filter that was used as an example above is plotted in the next figure.

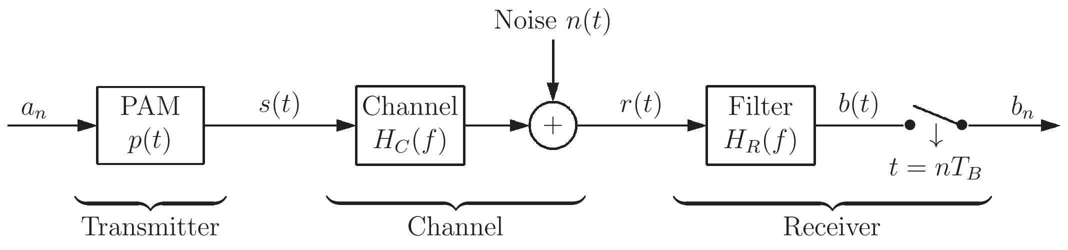


1.2 Filtering PAM Signals

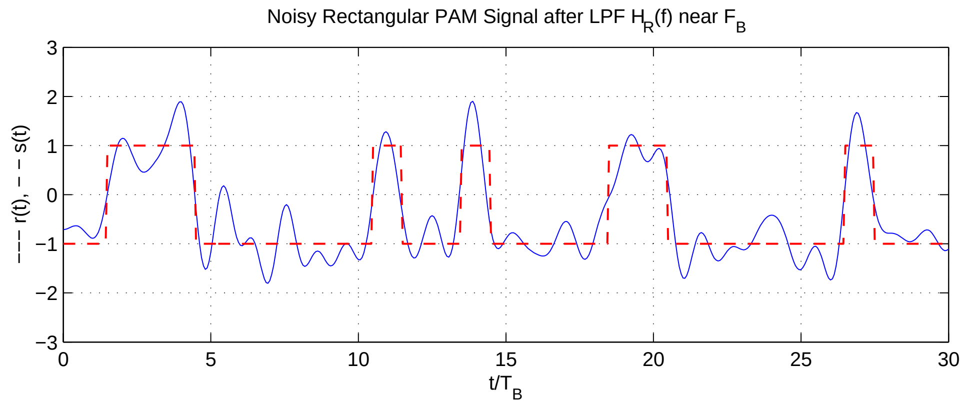
Suppose you received a noisy rectangular PAM signal $r(t)$ with baud rate $F_B = 1/T_B$, like the one shown in the graph below.



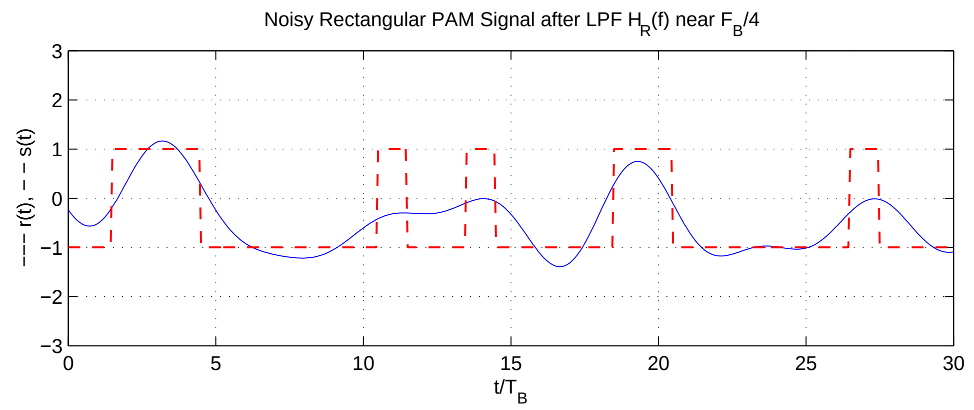
Before sampling $r(t)$ at times $t = nT_B$, it certainly is a good idea to perform some kind of averaging operation. This is the function of the receiver filter $h_R(t) \Leftrightarrow H_R(f)$ shown in the following blockdiagram of a general PAM communication system.



If $H_R(f)$ is a LPF with cutoff frequency near F_B, then the output $b(t)$ of the receiver filter from the same $r(t)$ as shown above looks as follows.



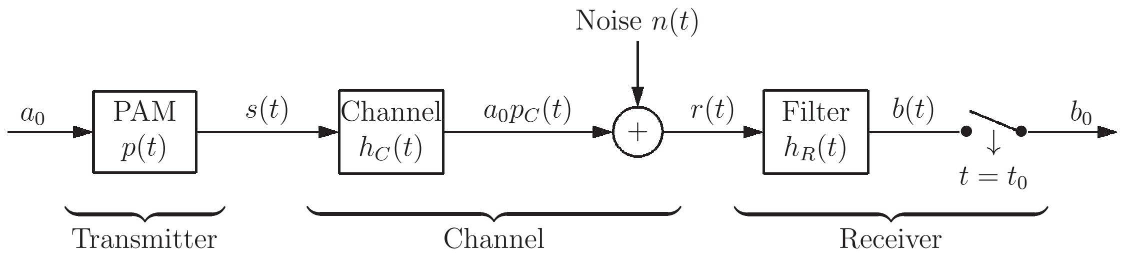
This is quite an improvement, and it is tempting to see if more filtering would lead to further improvements. The following graph shows $r(t)$ after lowpass filtering near $F_B/4$.



That does remove more noise, but at the price of loosing signal energy and introducing intersymbol interference (ISI). Thus, a compromise needs to be made between rejecting as much noise as possible, while keeping most of the signal energy and avoiding ISI. A properly chosen LPF is a good initial choice, but to maximize the signal-to-noise ratio (SNR) at the output of the receiver filter, a matched filter, as described in the next section, needs to be used.

1.3 Matched Filter

Suppose a transmitter sends the PAM signal $s(t) = a_0\, p(t)$ over a channel with unit impulse response $h_C(t) \Leftrightarrow H_C(f)$. It is assumed that $p(t)$ is a deterministic pulse, but a_0 is a random variable. In the absence of noise the received signal is thus $a_0\, p(t) * h_C(t) = a_0\, p_C(t)$, where $p_C(t) = p(t) * h_C(t)$. If an additive noise model is used for the channel, then the received signal is $r(t) = a_0\, p_C(t) + n(t)$, where $n(t)$ is the noise signal. The question now is how to design a receiver that estimates a_0 as accurately as possible. The following block diagram depicts the situation graphically.



Note that, in order to avoid the issue of ISI, this model uses the so called “one shot” approach, i.e., only the single sample a_0 is transmitted. If the overall pulse $q(t) = p_C(t) * h_R(t)$ satisfies Nyquist’s first criterion, then the results from the “one shot” approach directly generalize to the transmission of sequences $a_0, a_1, a_2, \ldots$ of symbols.

Assume that the random variable a_0 and the noise $n(t)$ are uncorrelated and that $n(t)$ is a wide-sense stationary (WSS) CT random process with zero mean, i.e., $E[n(t)] = 0$ and autocorrelation function $R_n(t,t-\tau) = E[n(t)\,n^*(t-\tau)] = R_n(\tau)$, all t. The power spectral density (PSD) $S_n(f)$ of the noise is the FT of the correlation function $R_n(\tau)$, i.e.,

$$S_n(f) = \int_{-\infty}^{\infty} R_n(\tau) e^{-j2\pi f\tau}\, d\tau \ .$$

For **white noise** $R_n(\tau) = (\mathcal{N}_0/2)\delta(\tau) \Leftrightarrow S_n(f) = \mathcal{N}_0/2$, all f, i.e., white noise has a flat (2-sided) PSD.

One optimality criterion for selecting the receiver filter response $h_R(t)$ is the maximization of the signal-to-noise ratio (SNR) of b_0 after sampling at $t = t_0$. If $r(t) = a_0\, p_C(t) + n(t)$, then

$$b(t) = h_R(t) * \big(a_0\, p_C(t) + n(t)\big) = a_0 \int_{-\infty}^{\infty} H_R(f)\, P_C(f)\, e^{j2\pi ft}\, df + \int_{-\infty}^{\infty} h_R(\mu)\, n(t-\mu)\, d\mu \ .$$

Therefore

$$\begin{aligned} E[b(t)] &= E[a_0] \int_{-\infty}^{\infty} H_R(f)\, P_C(f)\, e^{j2\pi ft}\, df + \int_{-\infty}^{\infty} h_R(\mu)\, \underbrace{E[n(t-\mu)]}_{=\,0}\, d\mu \\ &= E[a_0] \int_{-\infty}^{\infty} H_R(f)\, P_C(f)\, e^{j2\pi ft}\, df \ . \end{aligned}$$

In the absence of noise (i.e., $n(t) = 0$)

$$\begin{aligned} E[|b(t)|^2] &= E\Big[\big(a_0 \int_{-\infty}^{\infty} H_R(f)\, P_C(f)\, e^{j2\pi ft}\, df\big)\big(a_0 \int_{-\infty}^{\infty} H_R(\nu)\, P_C(\nu)\, e^{j2\pi \nu t}\, d\nu\big)^*\Big] \\ &= E[|a_0|^2] \left| \int_{-\infty}^{\infty} H_R(f)\, P_C(f)\, e^{j2\pi ft}\, df \right|^2 , \end{aligned}$$

and thus the signal power at $t = t_0$ is

$$E[|b_0|^2] = E[|b(t_0)|^2] = E[|a_0|^2] \left| \int_{-\infty}^{\infty} H_R(f)\, P_C(f)\, e^{j2\pi f t_0}\, df \right|^2 .$$

The autocorrelation function $R_b(t_1, t_2)$ at the output of the receiver filter when $r(t) = n(t)$

(noise only, no pulse present or $a_0 = 0$) is obtained as follows

$$\begin{aligned}
R_b(t_1, t_2) &= E[b(t_1)\, b^*(t_2)] = E\big[\big(h_R(t_1) * n(t_1)\big)\, \big(h_R(t_2) * n(t_2)\big)^*\big] \\
&= E\big[\int_{-\infty}^{\infty} h_R(\mu)\, n(t_1 - \mu)\, d\mu \int_{-\infty}^{\infty} h_R^*(\nu)\, n^*(t_2 - \nu)\, d\nu\big] \\
&= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h_R(\mu)\, h_R^*(\nu) \underbrace{E[n(t_1-\mu)\, n^*(t_2-\nu)]}_{= R_n(t_1 - t_2 - \mu + \nu) = \int S_n(f)\, e^{j2\pi f(t_1-t_2-\mu+\nu)} df} d\mu\, d\nu \\
&= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h_R(\mu)\, h_R^*(\nu)\, S_n(f)\, e^{j2\pi f(t_1-t_2-\mu+\nu)}\, df\, d\mu\, d\nu \\
&= \int_{-\infty}^{\infty} S_n(f) \int_{-\infty}^{\infty} h_R(\mu)\, e^{-j2\pi f\mu}\, d\mu \int_{-\infty}^{\infty} h_R^*(\nu)\, e^{j2\pi f\nu}\, d\nu\, e^{j2\pi f(t_1-t_2)}\, df \\
&= \int_{-\infty}^{\infty} S_n(f)\, |H_R(f)|^2\, e^{j2\pi f(t_1-t_2)}\, df = R_b(t_1 - t_2)\, .
\end{aligned}$$

The variance of the noise at time t at the output of the filter $h_R(t)$ is therefore

$$\sigma_b^2 = R_b(0) = \int_{-\infty}^{\infty} S_n(f)\, |H_R(f)|^2\, df\, .$$

Assuming white noise with $S_n(f) = \mathcal{N}_0/2$ for all f, this simplifies to

$$\sigma_b^2 = R_b(0) = \frac{\mathcal{N}_0}{2} \int_{-\infty}^{\infty} |H_R(f)|^2\, df\, .$$

The SNR after sampling at $t = t_0$ in the receiver can now be expressed as

$$\frac{S}{N} = \frac{E[|b_0|^2]}{\sigma_b^2} = \frac{2E[|a_0|^2]}{\mathcal{N}_0}\, \frac{\left|\int_{-\infty}^{\infty} H_R(f)\, P_C(f)\, e^{j2\pi f t_0}\, df\right|^2}{\int_{-\infty}^{\infty} |H_R(f)|^2\, df}\, .$$

To maximize S/N by choice of $h_R(t) \Leftrightarrow H_R(f)$, the following theorem comes in handy.

Theorem: Schwartz Inequality. Let $U(f)$ and $V(f)$ be real- or complex-valued and have finite energy. Then

$$\Big|\int_{-\infty}^{\infty} U(f)\, V(f)\, df\Big|^2 \le \int_{-\infty}^{\infty} |U(f)|^2\, df \int_{-\infty}^{\infty} |V(f)|^2\, df\, ,$$

with equality iff $V^*(f) = A\, U(f)$ for some real- or complex-valued constant A. □

In the context of the PAM receiver filter this implies that

$$\Big|\int_{-\infty}^{\infty} H_R(f)\, P_C(f)\, e^{j2\pi f t_0}\, df\Big|^2 \le \int_{-\infty}^{\infty} |H_R(f)|^2\, df \int_{-\infty}^{\infty} \underbrace{|P_C(f)\, e^{j2\pi f t_0}|^2}_{= |P_C(f)|^2}\, df\, ,$$

and thus

$$\frac{S}{N} \le \frac{2E[|a_0|^2]}{\mathcal{N}_0} \, \frac{\int_{-\infty}^{\infty} |H_R(f)|^2 \, df \, \int_{-\infty}^{\infty} |P_C(f)|^2 \, df}{\int_{-\infty}^{\infty} |H_R(f)|^2 \, df} = \frac{2E[|a_0|^2]}{\mathcal{N}_0} \int_{-\infty}^{\infty} |P_C(f)|^2 \, df \, ,$$

with equality iff

$$P_C^*(f) \, e^{-j2\pi f t_0} = A \, H_R(f) \qquad \Longrightarrow \qquad H_R(f) = \frac{P_C^*(f) \, e^{-j2\pi f t_0}}{A} \, .$$

Substituting this in the equation for $E[b(t)]$ at $t = t_0$ yields

$$E[b_0] = \frac{E[a_0]}{A} \int_{-\infty}^{\infty} |P_C(f)|^2 \, df \qquad \Longrightarrow \qquad A = \int_{-\infty}^{\infty} |P_C(f)|^2 \, df = \int_{-\infty}^{\infty} |p_C(t)|^2 \, dt \, ,$$

for $E[b_0] = E[a_0]$. Altogether this proves the following

Theorem: Matched Filter. Let $r(t) = a_0 \, p_C(t) + n(t)$ be a received PAM signal with random amplitude a_0, known pulse $p_C(t)$, and additive white noise $n(t)$ with PSD $S_n(f) = \mathcal{N}_0/2$ for all f. Assume that a_0 and $n(t)$ are uncorrelated. Then the filter

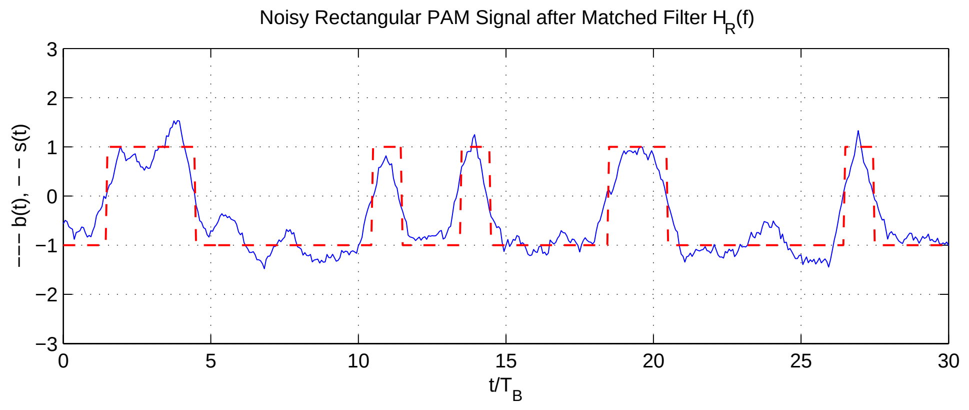
$$h_R(t) = \frac{p_C^*(t_0 - t)}{\int_{-\infty}^{\infty} |p_C(\mu)|^2 \, d\mu} \qquad \Longleftrightarrow \qquad H_R(f) = \frac{P_C^*(f) \, e^{-j2\pi f t_0}}{\int_{-\infty}^{\infty} |P_C(\nu)|^2 \, d\nu}$$

which is called a matched filter, maximizes the SNR at its output when sampled at $t = t_0$. The expected output value (at $t = t_0$) is $E[a_0]$, with SNR

$$\frac{S}{N} = \frac{2 \, E[|a_0|^2]}{\mathcal{N}_0} \int_{-\infty}^{\infty} |p_C(\mu)|^2 \, d\mu = \frac{2 \, E[|a_0|^2]}{\mathcal{N}_0} \int_{-\infty}^{\infty} |P_C(\nu)|^2 \, d\nu \, .$$

$\square$

The plot below shows the output of the matched filter for the same noisy rectantgular PAM signal $r(t)$ that was used as an example in a the section.



The improvement over simple lowpass filtering is clearly visible.

1.4 Root Raised Cosine in Frequency Pulse

The overall pulse $q(t)$ in a PAM communication system with matched filter receiver is

$$q(t) = p_C(t) * h_R(t) \qquad \Longleftrightarrow \qquad Q(f) = P_C(f)\, H_R(f)$$

Assuming $t_0 = 0$ and white noise with $S_n(f) = \mathcal{N}_0/2$, yields

$$Q(f) = \frac{|P_C(f)|^2}{\int_{-\infty}^{\infty} |P_C(\nu)|^2\, d\nu} \, .$$

Thus, to satisfy Nyquist's 1'st criterion for no ISI after the matched filter, $P_C(f)$ has to satisfy

$$\sum_{k=-\infty}^{\infty} |P_C(f - kF_B)|^2 = K \, , \qquad \text{all } f \, ,$$

for some constant K. Since the RCf pulse is bandlimited and behaves well in the time domain, it is a natural starting point for finding a bandlimited $P_C(f)$ that satisfies the above formula. Recall that that the RCf pulse with parameter $0 \le \alpha \le 1$ is defined as

$$p(t) = \frac{\sin(\pi t/T_B)}{\pi t/T_B} \, \frac{\cos(\pi \alpha t/T_B)}{1 - (2\alpha t/T_B)^2} \, ,$$

with Fourier transform

$$P(f) = \begin{cases} T_B \, , & |f| \le \dfrac{1-\alpha}{2T_B} \, , \\[2ex] \dfrac{T_B}{2} \Big[1 + \cos\Big(\dfrac{\pi T_B}{\alpha}\big(|f| - \dfrac{1-\alpha}{2T_B}\big)\Big)\Big] \, , & \dfrac{1-\alpha}{2T_B} \le |f| \le \dfrac{1+\alpha}{2T_B} \, , \\[2ex] 0 \, , & |f| \ge \dfrac{1+\alpha}{2T_B} \, . \end{cases}$$

To obtain the square root of this $P(f)$, use the trigonometric identity

$$\cos^2 \alpha = \frac{1}{2}(1 + \cos(2\alpha)) \quad \Longrightarrow \quad \sqrt{\frac{1 + \cos\beta}{2}} = \cos\big(\frac{\beta}{2}\big) \, .$$

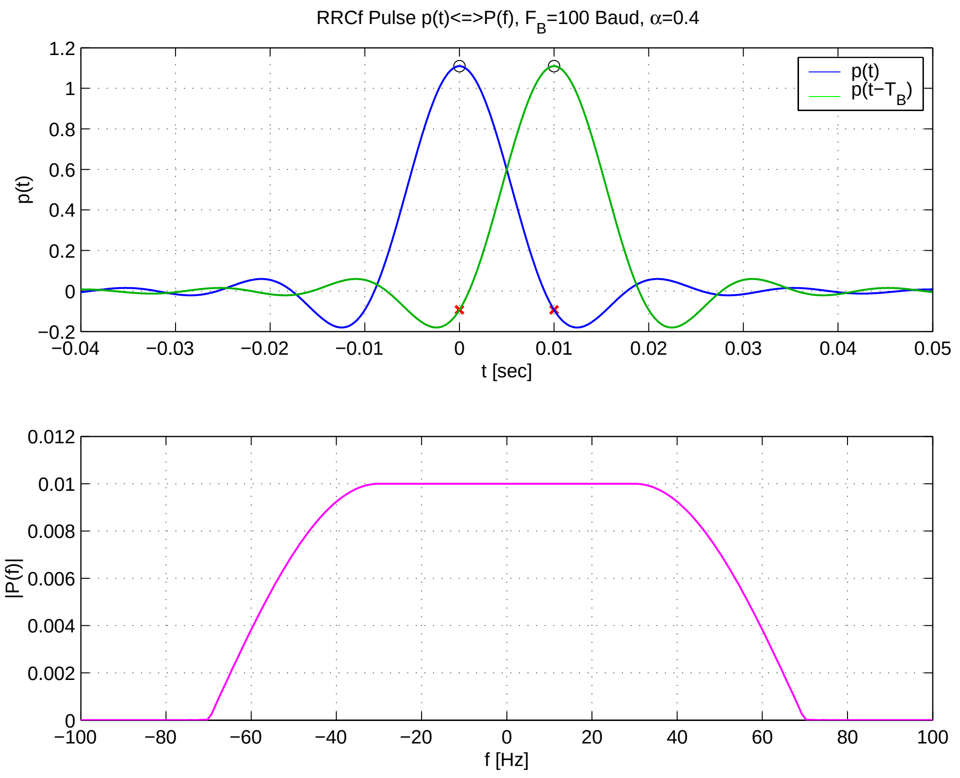
In addition, multiply the resulting spectrum by $\sqrt{T_B}$ (to keep the dc component unchanged) so that

$$P_C(f) = \begin{cases} T_B \cos\Big(\dfrac{\pi T_B}{2\alpha}\big(|f| - \dfrac{1-\alpha}{2T_B}\big)\Big) \, , & \dfrac{1-\alpha}{2T_B} \le |f| \le \dfrac{1+\alpha}{2T_B} \, , \\[2ex] T_B \, , & |f| \le \dfrac{1-\alpha}{2T_B} \, , \\[2ex] 0 \, , & \text{otherwise} \, . \end{cases}$$

which is proportional to the **square root of the RCf spectrum**. In the time domain this corresponds to

$$p_C(t) = \begin{cases} \dfrac{T_B}{\pi} \, \dfrac{\sin\big((1-\alpha)\pi t/T_B\big) + (4\alpha t/T_B)\, \cos\big((1+\alpha)\pi t/T_B\big)}{\big(1-(4\alpha t/T_B)^2\big)\, t} \,, & t \neq 0, \pm\dfrac{T_B}{4\alpha} \,, \\ 1 - \alpha + \dfrac{4\alpha}{\pi} \,, & t = 0 \,, \\ \dfrac{\alpha}{\sqrt{2}} \Big[\big(1+\dfrac{2}{\pi}\big) \sin\big(\dfrac{\pi}{4\alpha}\big) + \big(1-\dfrac{2}{\pi}\big) \cos\big(\dfrac{\pi}{4\alpha}\big)\Big] \,, & t = \pm\dfrac{T_B}{4\alpha} \,, \end{cases}$$

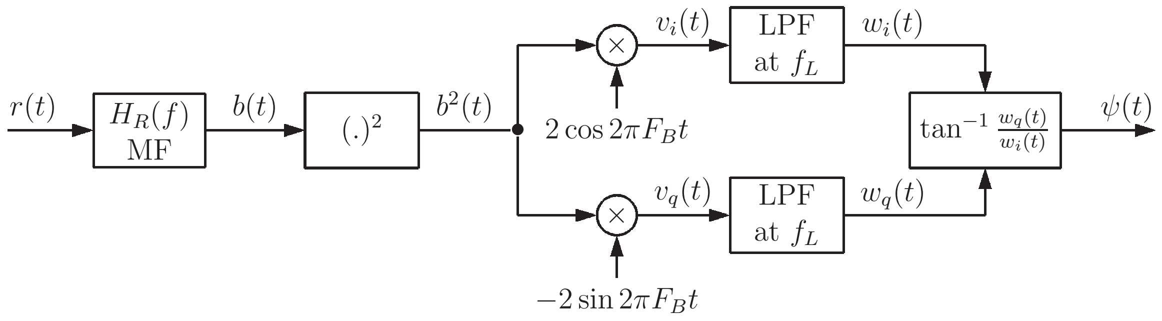
where $0 \leq \alpha \leq 1$. Because of its spectral shape and origin, this pulse will be called the **root raised cosine in frequency (RRCf)** pulse. An example of a RRCf pulse $p(t)$ and $p(t - T_B)$ and its Fourier transform $|P(f)|$ for $\alpha = 0.4$ is shown in the graphs below.



Note that this $p(t)$ creates intersymbol interference, as indicated by the red crosses at the sampling times (black circles), during transmission. After matched filtering at the receiver, however, this intersymbol interference disappears.

1.5 Symbol Timing Synchronization

As shown previously, a spectral component at the baud rate F_B can be extracted from most practical PAM signals by squaring the received signal $r(t)$. To remove as much noise as possible, it is best to use a matched filter (MF) $H_R(f)$ first and then to square its output $b(t)$ as shown in the following block diagram.



The reason why symbol timing synchronization at the receiver is necessary is that the baud rate F_B used (initially) at the receiver differs from the baud rate $F_B + f_e$ used at the transmitter by the frequency error f_e. In addition, there may also be a phase error θ_e. Reliable reception of a PAM signal is only possible if $f_e = 0$ and θ_e is sufficiently small.

Let F_B be the baud rate used by the receiver and let

$$b_B(t) = A\,\cos\big(2\pi(f_B + f_e)t + \theta_e\big)\ ,$$

be the term in $b^2(t)$ that contains the spectral component at the baud rate $F_B + f_e$ used by the transmitter. To estimate f_e and θ_e, $b^2(t)$ is multiplied by $2\cos 2\pi F_B t$ and by $-2\sin 2\pi F_B t$ to obtain

$$\begin{aligned} v_i(t) &\approx 2b_B(t)\cos 2\pi f_B t = A\big[\cos(2\pi f_e t + \theta_e) + \cos(2\pi(2F_B + f_e)t + \theta_e)\big]\ ,\\ v_q(t) &\approx 2b_B(t)\cos 2\pi f_B t = A\big[\sin(2\pi f_e t + \theta_e) - \sin(2\pi(2F_B + f_e)t + \theta_e)\big]\ ,\end{aligned}$$

where the approximation comes from the fact that $b^2(t)$ also contains other terms than the (strong) spectral component in $b_B(t)$. After lowpass filtering with cutoff frequency $f_L < F_B$

$$\begin{aligned} w_i(t) &\approx A\cos(2\pi f_e t + \theta_e)\ ,\\ w_q(t) &\approx A\sin(2\pi f_e t + \theta_e)\ .\end{aligned}$$

Thus after taking the inverse tangent of $w_q(t)/w_i(t)$ (resolved modulo 2π) and unwrapping the phase at jumps by $\pm 2\pi$,

$$\psi(t) \approx 2\pi f_e t + \theta_e\ .$$

This is the (estimated) error between the symbol timing at the transmitter and the symbol timing at the receiver. To obtain a pulse train $c(t)$ for sampling the received signal $b(t)$ at the (estimated) best time instants, compute

$$g(t) = \frac{d}{dt}\big\{\,\mathrm{sgn}\left[\sin\big(2\pi F_B t + \psi(t)\big)\right]\big\}\ ,$$

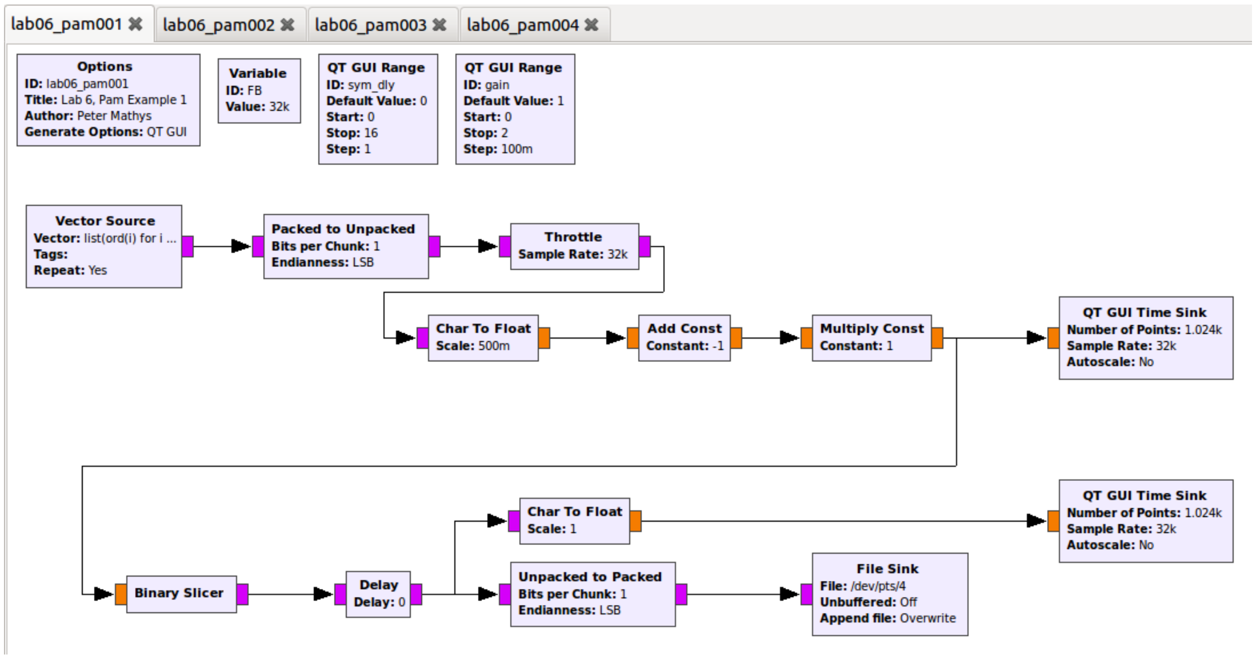
where sgn(.) is the signum function ($\mathrm{sgn}(x) = -1$ if $x < 0$, $\mathrm{sgn}(x) = +1$ otherwise), and then set

$$c(t) = \frac{g(t) + |g(t)|}{4}\ .$$

Thus, $c(t)$ has impulses of area 1 at the positive zero crossings of $\sin\big(2\pi F_B t + \psi(t)\big)$.

1.6 Baseband PAM using GNU Radio

In this section our goal is to use the GNU Radio Companion (GRC) to implement an end-to-end communication system for transmitting and receiving an ASCII encoded text using PAM signals with different pulse shapes over a noisy communication channel. The first thing we need to be able to do is generate a text message, convert it to a serial stream of binary polar symbols, attenuate (or amplify) these symbols during transmission, and then convert the received symbol stream back to an ASCII encoded text which is displayed in a text window. The figure below shows a GRC flowgraph, called `lab06_pam001` that can accomplish this first task.



Options
ID: lab06_pam001
Title: Lab 6, Pam Example 1
Author: Peter Mathys
Generate Options: QT GUI

Variable
ID: FB
Value: 32k

QT GUI Range
ID: sym_dly
Default Value: 0
Start: 0
Stop: 16
Step: 1

QT GUI Range
ID: gain
Default Value: 1
Start: 0
Stop: 2
Step: 100m

Vector Source
Vector: list(ord(i) for i ...
Tags:
Repeat: Yes

Packed to Unpacked
Bits per Chunk: 1
Endianness: LSB

Throttle
Sample Rate: 32k

Char To Float
Scale: 500m

Add Const
Constant: -1

Multiply Const
Constant: 1

QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 32k
Autoscale: No

Binary Slicer

Delay
Delay: 0

Char To Float
Scale: 1

QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 32k
Autoscale: No

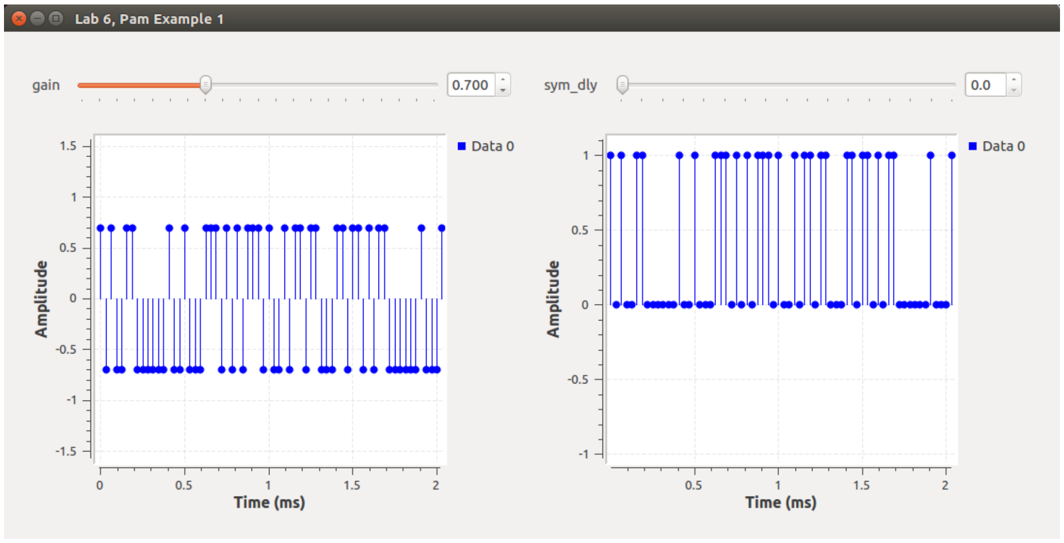
Unpacked to Packed
Bits per Chunk: 1
Endianness: LSB

File Sink
File: /dev/pts/4
Unbuffered: Off
Append file: Overwrite

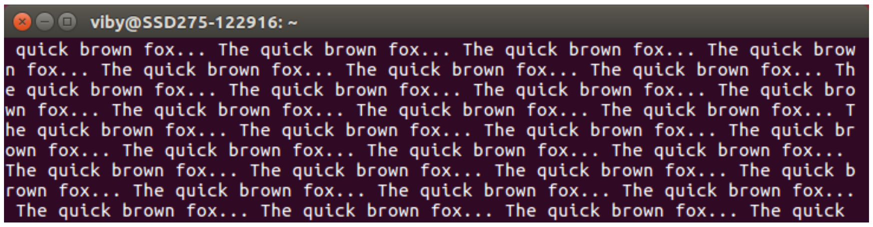
The text `"The quick brown fox..."` is generated by the "Vector Source" block at the top left using the Python statement `list(ord(i) for i in 'The quick brown fox...')` for the "Vector" entry. The output type of this block is set to "Byte" and thus one byte is output for each character. The "Packed to Unpacked" block breaks the bytes up into chunks of 1 bit each, with LSB or MSB first selectable for the "Endianness" parameter. The output of this block is another byte with values of only 0 or 1 if the "Bits per Chunk" value is set to 1. Next, this stream of 0/1 bytes is passed through a "Throttle" block with "Sample Rate" `FB` (symbol or baud rate, set to 32000 in the flowgraph shown). The next blocks, "Char to Float" (with "Scale" 0.5) followed by "Add Const" (with "Constant" = -1) converts the uniploar binary signal to a bipolar ± 1.0 floating point signal. Note that the "Scale" value in the "Char to Float" block **divides** the input signal (thus, a value of 0.5 amplifies the output by a factor of 2). To simulate the attenuation/gain effect of a real channel, a "Multiply Const" block with adjustable "Constant" factor `gain` (in the range from 0 to 2 with step size 0.1) is used. The output of this block can be displayed using the first "QT GUI Time Sink".

The receiver portion is shown in the bottom half of the flowgraph. The first block is a "Binary Slicer" which takes the floating point input and outputs 0 for negative inputs and 1 for non-negative inputs. Now we have a stream of bits that we need to convert back to bytes to retrieve the ASCII message. But in general we do not know where the byte boundaries in the incoming bit stream are. Thus, we need a "Delay" block with variable (integer) "Delay" value `sym_dly` to start with bit 0, or bit 1, or bit 2, etc, as the first bit in the bit to byte conversion. To verify that we only get zeros and ones at the output of the "Delay" block, we use the second "QT GUI Time Sink" together with a "Char to Float" type converter. The actual bit to byte conversion then takes place in the "Unpacked to Packed" block with the "Bits per Chunk" and the "Endianness" parameters set to the same values as for the "Packed to Unpacked" block. The bytes at the output of the "Unpacked to Packed" block can now be displayed and interpreted as ASCII characters using a "File Sink" with a terminal "File" name of the type `/dev/pts/n` where `n` is a number like 1 or 4 or 18. To find an appropriate `n`, open a new Linux terminal and type `tty` followed by "Enter". This will display `/dev/pts/n` with `n` set to the number specific to the terminal that you just opened.

Now we are ready for a test run of `lab06_pam001`. The first graph below shows the displays of the two "QT GUI Time Sinks".

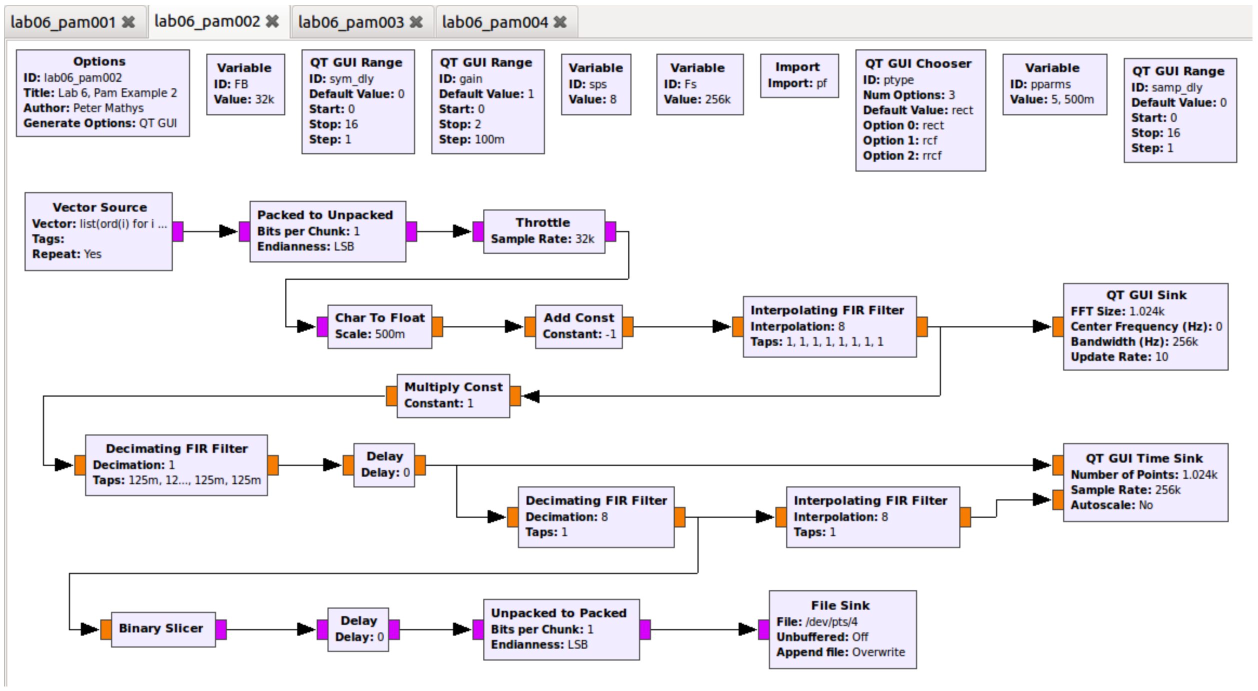


We can clearly see that the "received" DT signal on the left is a (slightly) attenuated polar binary signal with the signal on the right its true binary 0/1 counterpart after the binary slicer. The next figure shows the text display when the bit delay (`sym_delay`) is adjusted correctly.


```
viby@SSD275-122916: ~
 quick brown fox... The quick brown fox... The quick brown fox... The quick brow
n fox... The quick brown fox... The quick brown fox... The quick brown fox... Th
e quick brown fox... The quick brown fox... The quick brown fox... The quick bro
wn fox... The quick brown fox... The quick brown fox... The quick brown fox... T
he quick brown fox... The quick brown fox... The quick brown fox... The quick br
own fox... The quick brown fox... The quick brown fox... The quick brown fox...
The quick brown fox... The quick brown fox... The quick brown fox... The quick b
rown fox... The quick brown fox... The quick brown fox... The quick brown fox...
 The quick brown fox... The quick brown fox... The quick brown fox... The quick
```

Change the `sym_delay` delay value from its correct value to see how profoundly the text display deteriorates with the wrong byte alignment within the received bit stream.

The next GRC flowgraph, called `lab06_pam002` incorporates the DT sequence to CT waveform and back to DT sequence conversions.



Options
ID: lab06_pam002
Title: Lab 6, Pam Example 2
Author: Peter Mathys
Generate Options: QT GUI

Variable
ID: FB
Value: 32k

QT GUI Range
ID: sym_dly
Default Value: 0
Start: 0
Stop: 16
Step: 1

QT GUI Range
ID: gain
Default Value: 1
Start: 0
Stop: 2
Step: 100m

Variable
ID: sps
Value: 8

Variable
ID: Fs
Value: 256k

Import
Import: pf

QT GUI Chooser
ID: ptype
Num Options: 3
Default Value: rect
Option 0: rect
Option 1: rcf
Option 2: rrcf

Variable
ID: pparms
Value: 5, 500m

QT GUI Range
ID: samp_dly
Default Value: 0
Start: 0
Stop: 16
Step: 1

Vector Source
Vector: list(ord(i) for i ...
Tags:
Repeat: Yes

Packed to Unpacked
Bits per Chunk: 1
Endianness: LSB

Throttle
Sample Rate: 32k

Char To Float
Scale: 500m

Add Const
Constant: -1

Interpolating FIR Filter
Interpolation: 8
Taps: 1, 1, 1, 1, 1, 1, 1, 1

QT GUI Sink
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 256k
Update Rate: 10

Multiply Const
Constant: 1

Decimating FIR Filter
Decimation: 1
Taps: 125m, 12..., 125m, 125m

Delay
Delay: 0

QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 256k
Autoscale: No

Decimating FIR Filter
Decimation: 8
Taps: 1

Interpolating FIR Filter
Interpolation: 8
Taps: 1

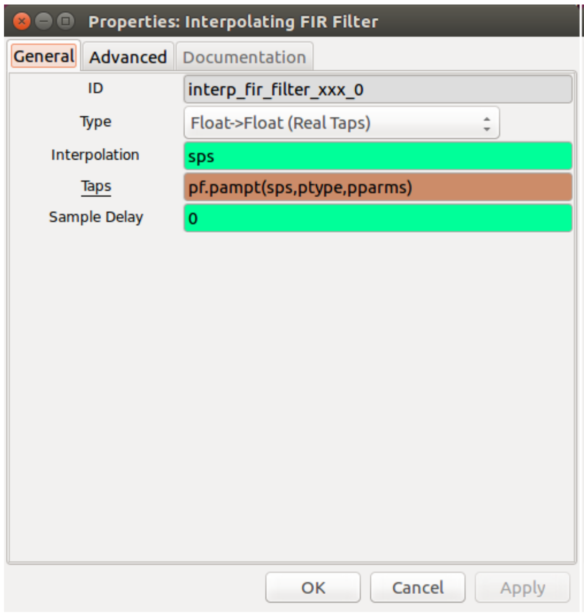
Binary Slicer

Delay
Delay: 0

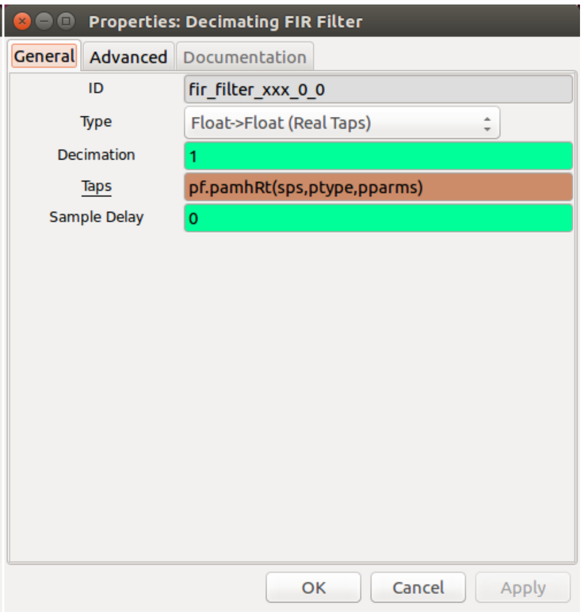
Unpacked to Packed
Bits per Chunk: 1
Endianness: LSB

File Sink
File: /dev/pts/4
Unbuffered: Off
Append file: Overwrite

The new blocks are “Interpolating FIR Filter” and ”Decimating FIR Filter” blocks and a second “Delay” block. The properties windows of the main “Interpolating FIR Filter” (which implements the PAM transmitter with pulse shapes `'rect'`, `'rcf'`, and `'rrcf'`) and the main “Decimating FIR Filter” (which implements the matched filter receiver for the same pulse shapes) are shown below.

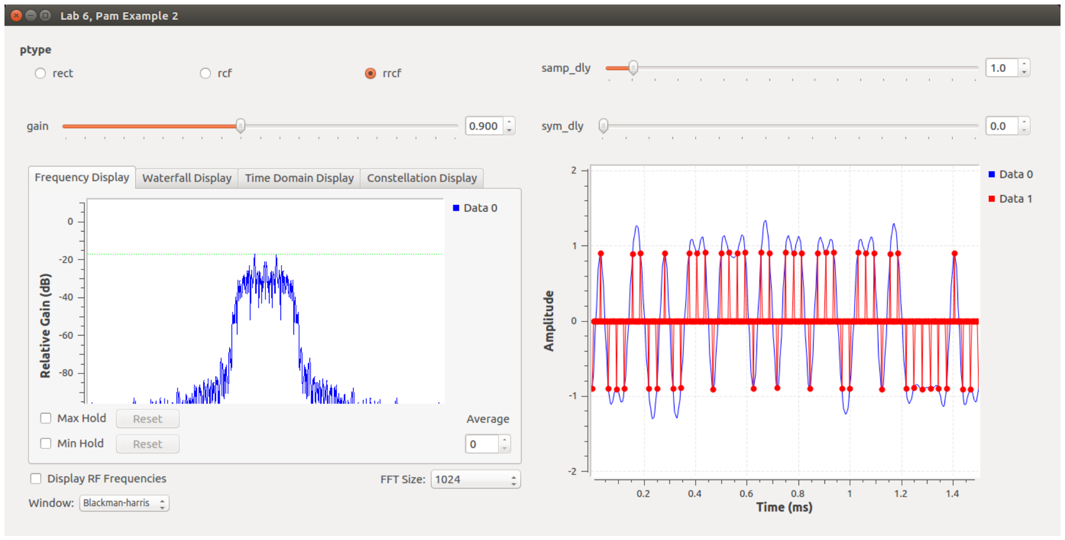

Properties: Interpolating FIR Filter

ID	interp_fir_filter_xxx_0
Type	Float->Float (Real Taps)
Interpolation	sps
Taps	pf.pampt(sps,ptype,pparms)
Sample Delay	0


Properties: Decimating FIR Filter

ID	fir_filter_xxx_0_0
Type	Float->Float (Real Taps)
Decimation	1
Taps	pf.pamhRt(sps,ptype,pparms)
Sample Delay	0

The `pampt` and the `pamhRt` functions which define $p(t)$ and $h_R(t)$ are implemented in the `ptfun.py` module which you started in an earlier lab (you will need to update this module with the `'rcf'` and the `'rrcf'` pulses). The pulse types `ptype` are selectable using a "QT GUI Chooser" and the pulse parameters `pparms=[k,alpha]` are set to `[5,0.5]` using a "Variable" block. The `ptfun.py` module needs to be installed in a directory referenced by `$PYTHONPATH` (typically `/usr/local/lib/python2.7/site-packages`) so that the GRC can find it. In the flowgraph above this module is imported used the "Import" statement `import ptfun as pf`. Note that the PAM transmitter filter uses an upsampling factor `sps` (samples per symbol) of 8 and thus the sampling rate `Fs` of the transmitted "CT waveform" is `8*32000=256000` Hz. The "Decimating FIR Filter" which implements the receiver operates at `Fs` and uses a decimation factor of 1. It is followed by a "Delay" block with selectable value `samp_dly` in the range 0 to `2*sps`. This is used to adjust the optimal sampling time at the output of the matched filter. A separate "Decimating FIR Filter" with decimation factor `sps` and a filter tap of just a single `1` is then used to obtain the received DT sequence that is fed into the "Binary Slicer" and displayed in the "File Sink" terminal window. To be able to see the received signal from the output of the matched filter before and after decimation by `sps`, a "QT GUI Time Sink" with two inputs is used. Since the "QT GUI Time Sink" does not work with inputs that have different sampling rates, an "Interpolating FIR Filter" with interpolation factor `sps` and a single filter tap of 1 is used to upsample the received DT sequence to rate `Fs`. The resulting display for `'rrcf'` PAM with the `samp_dly` (and the `sym_dly`) correctly adjusted looks like this.


Lab 6, Pam Example 2

ptype: rect, rcf, [✓ rrcf]

samp_dly: 1.0

gain: 0.900

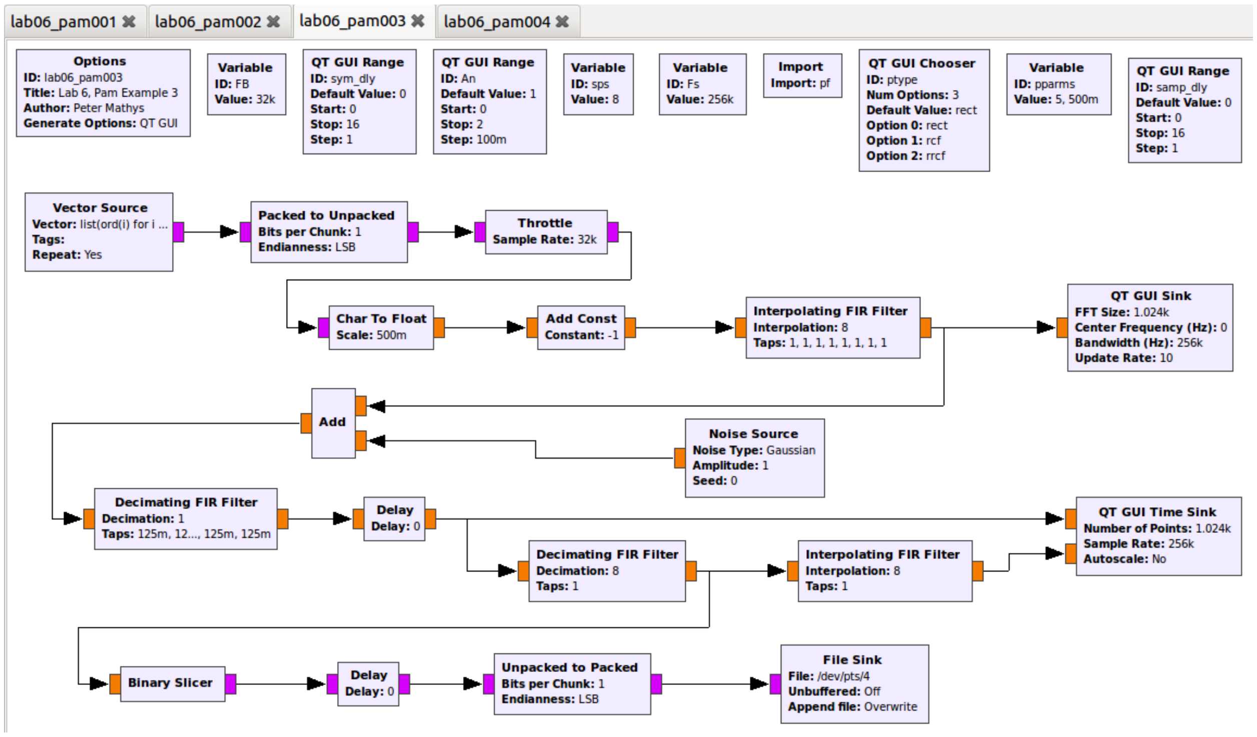
sym_dly: 0.0

Max Hold, Min Hold, Average: 0

Display RF Frequencies, FFT Size: 1024

Window: Blackman-harris

The next step is to see how this whole system performs in the presence of additive white Gaussian noise (AWGN). The corresponding flowgraph, called `lab06_pam003`, replaces the "Multiply Const" block that simulated the channel attenuation/gain with an adder and a "Noise Source" with "Noise Type" Gaussian and adjustable "Amplitude" `An`.


lab06_pam001 | lab06_pam002 | lab06_pam003 | lab06_pam004

Options — ID: lab06_pam003; Title: Lab 6, Pam Example 3; Author: Peter Mathys; Generate Options: QT GUI

Variable — ID: FB; Value: 32k

QT GUI Range — ID: sym_dly; Default Value: 0; Start: 0; Stop: 16; Step: 1

QT GUI Range — ID: An; Default Value: 1; Start: 0; Stop: 2; Step: 100m

Variable — ID: sps; Value: 8

Variable — ID: Fs; Value: 256k

Import — Import: pf

QT GUI Chooser — ID: ptype; Num Options: 3; Default Value: rect; Option 0: rect; Option 1: rcf; Option 2: rrcf

Variable — ID: pparms; Value: 5, 500m

QT GUI Range — ID: samp_dly; Default Value: 0; Start: 0; Stop: 16; Step: 1

Vector Source — Vector: list(ord(i) for i ...; Tags: ; Repeat: Yes

Packed to Unpacked — Bits per Chunk: 1; Endianness: LSB

Throttle — Sample Rate: 32k

Char To Float — Scale: 500m

Add Const — Constant: -1

Interpolating FIR Filter — Interpolation: 8; Taps: 1, 1, 1, 1, 1, 1, 1, 1

QT GUI Sink — FFT Size: 1.024k; Center Frequency (Hz): 0; Bandwidth (Hz): 256k; Update Rate: 10

Add

Noise Source — Noise Type: Gaussian; Amplitude: 1; Seed: 0

Decimating FIR Filter — Decimation: 1; Taps: 125m, 12..., 125m, 125m

Delay — Delay: 0

Decimating FIR Filter — Decimation: 8; Taps: 1

Interpolating FIR Filter — Interpolation: 8; Taps: 1

QT GUI Time Sink — Number of Points: 1.024k; Sample Rate: 256k; Autoscale: No

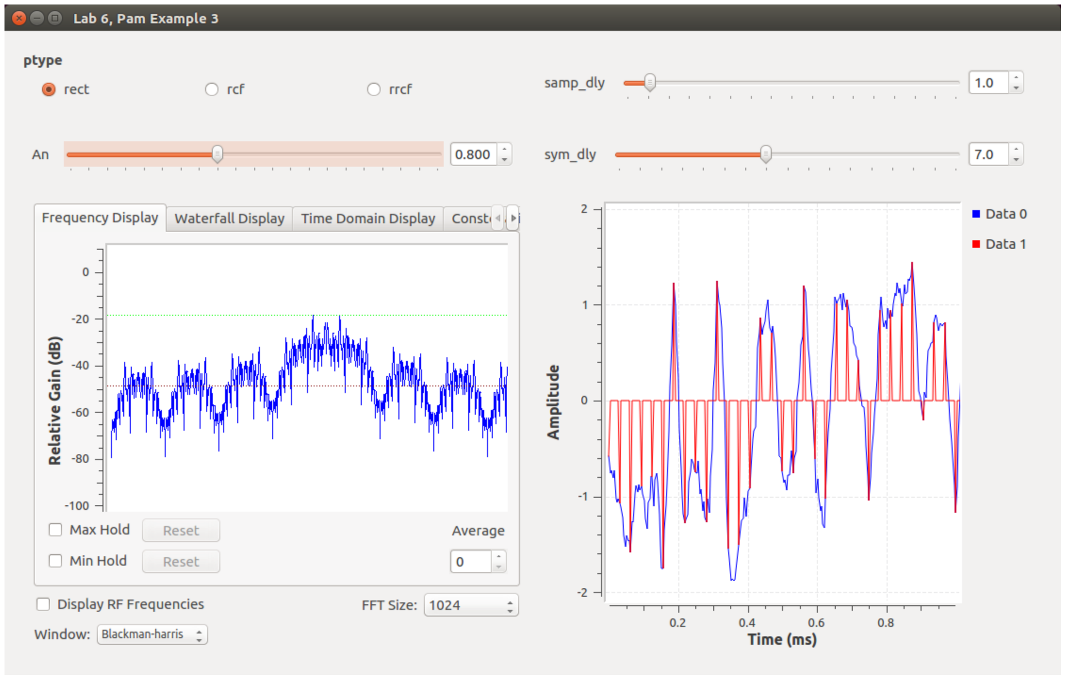
Binary Slicer

Delay — Delay: 0

Unpacked to Packed — Bits per Chunk: 1; Endianness: LSB

File Sink — File: /dev/pts/4; Unbuffered: Off; Append file: Overwrite

For a signal to noise ratio of about 8 dB (corresponding to `An = 0.8`), `'rect'` PAM, and all delays adjusted optimally, the displays in the frequency and time domains look as follows.



ptype: [✓ rect] rcf rrcf

samp_dly: 1.0

An: 0.800

sym_dly: 7.0

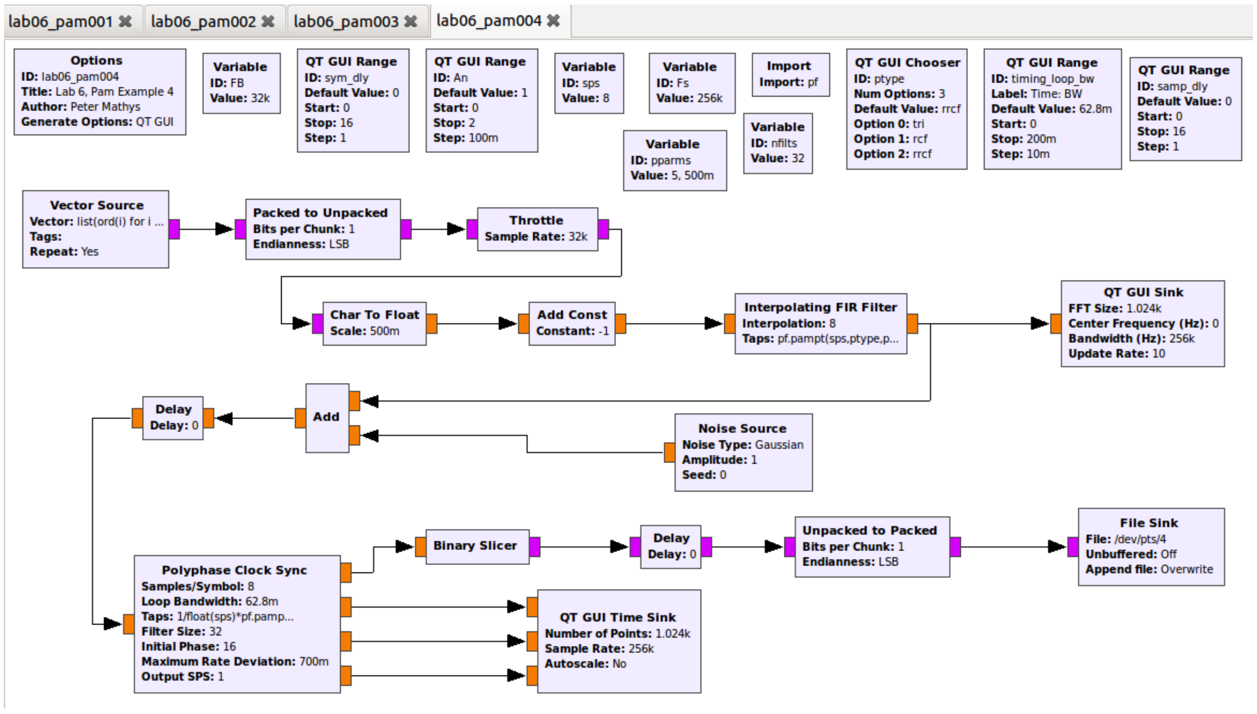
Max Hold ☐ Min Hold ☐ Average: 0

Display RF Frequencies ☐ FFT Size: 1024

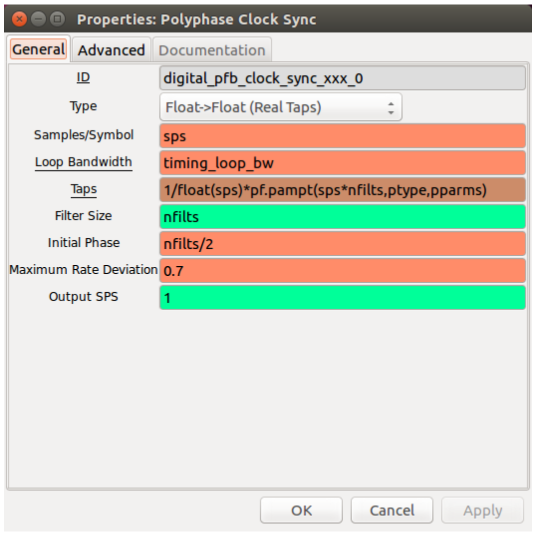
Window: Blackman-harris

The noise is clearly visible in the display on the right, but the text still reads correctly (the probability of error for a polar binary signal and 8 dB SNR is about 1 bit in 10,000). But try what happens if you use an `'rcf'` or an `'rrcf'` pulse with the same settings. What results would you expect?

Finally, we are now asking the question whether the adjustments of the `samp_dly` and the `sym_dly` delays could be automated. For the `samp_dly` we can use the product of the waveform at the output of the matched filter and its derivative to build a control loop similar to a PLL as described in the paper F.J. Harris and M. Rice, *"Multirate Digitial Filters for Symbol Timing Synchronization in Software Defined Radios,"* IEEE JSAC, Vol. 19, No. 12, Dec. 2001, pp. 2346–2357. In the GRC this is available in the form of a "Polyphase Clock Sync" block as shown in the flowgraph `lab06_pam004` below.

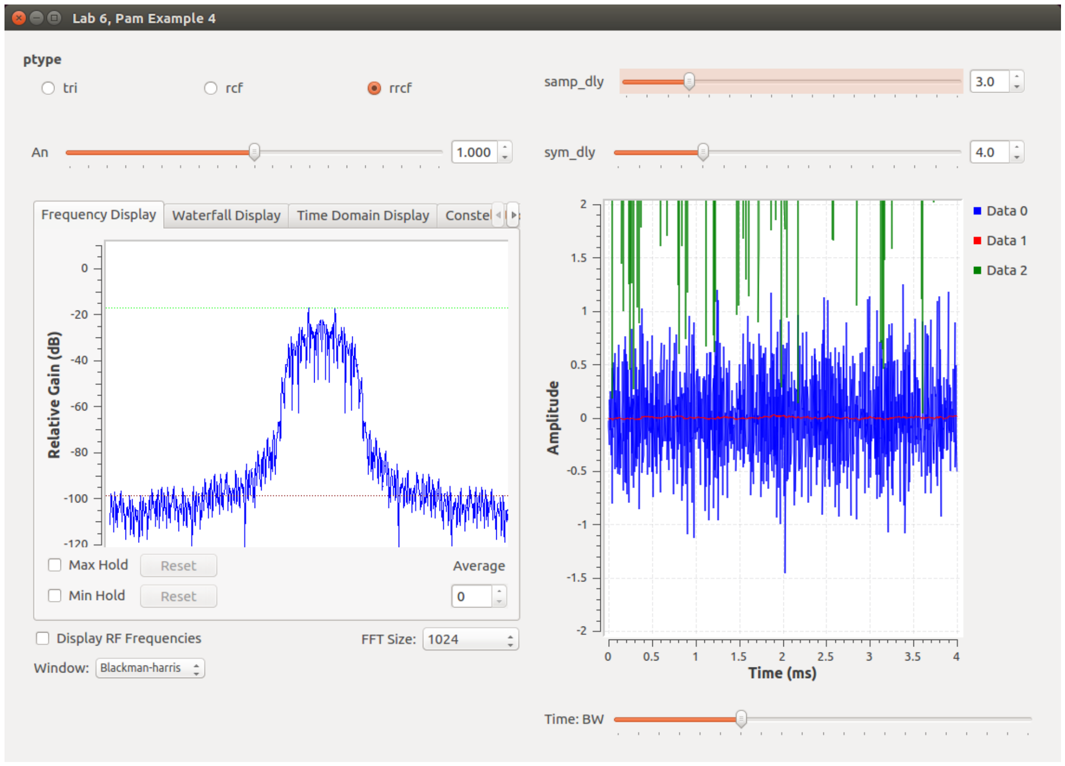


Now the matched filter, the delay, and the downsampling are all packed into one block. The "Properties" window of the "Polyphase Clock Sync" block looks as follows.



ID	digital_pfb_clock_sync_xxx_0
Type	Float->Float (Real Taps)
Samples/Symbol	sps
Loop Bandwidth	timing_loop_bw
Taps	1/float(sps)*pf.pampt(sps*nfilts,ptype,pparms)
Filter Size	nfilts
Initial Phase	nfilts/2
Maximum Rate Deviation	0.7
Output SPS	1

Note that the filter "Taps" are derived from the `pampt` function in the `ptfun.py` module. But since there is a problem in the "Polyphase Clock Sync" implementation taking the derivative of the `'rect'` pulse type, we substituted the `'tri'` pulse as one of the PAM pulse choices instead. Running the flowgraph for `'rrcf'` with a signal to noise ratio of about 6 dB yields the following graphs.


Lab 6, Pam Example 4

ptype: tri rcf [✓ rrcf]

samp_dly: 3.0

An: 1.000

sym_dly: 4.0

Max Hold: ☐

Min Hold: ☐

Average: 0

Display RF Frequencies: ☐

FFT Size: 1024

Window: Blackman-harris

The display on the left shows the PSD of the transmitted signal as before. The display on the right shows the instantaneous value of the product of the output of the matched filter with its derivative in blue and the average symbol rate deviation in red. Given th relatively low signal to noise ratio the algorithm performs very well for `'rrcf'` pulses. For `'tri'` and `'rcf'` pulses the performance deteriorates rapidly for low signal to noise ratio because of the intersymbol interference introduced by these pulse shapes. It should be noted that the `sym_dly` delay still needs to be adjusted manually even if the `samp_dly` delay is adjusted automatically. To make the `sym_dly` adjustment automatic in a real system, a higher level protocol that transmits some synchronization pattern at the beginning of a transmission is needed.

1.7 How does the Polyphase Clock Sync Work?

In short, it uses M (default $M = 32$) matched filters with delays staggered by $\Delta = T_B/M$ to produce M versions of $b(t - k\Delta)$, $k \in \{-M/2, M/2\}$. Assuming that $b(t)$ is the output that is currently selected, the product $v(t) = b(t)\, db(t)/dt$ at $t = nT_B$ is computed and lowpass filtered to produce output $w_n = w(nT_B)$. The cumulative sum $x_n = \sum_m w_m$ is then used to select the best phase (or delay) from the outputs of the M matched filters. If $x_n > 0$, the current filter output is too early for sampling and the output that produces $b(t - \Delta)$ is selected. Conversely, if $x_n < 0$, the current filter output is too late and the output that produces $b(t + \Delta)$ is selected.

— To be completed —

2 Lab Experiments

E1. LPF with Trapezoidal Frequency Response. (a) The unit impulse response of an LPF with trapezoidal frequency response and -6dB frequency f_L is

$$h(t) = \frac{\sin(2\pi f_L t)}{\pi t}\,\frac{\sin(2\pi\alpha f_L t)}{2\pi\alpha f_L t}\,, \qquad 0 \le \alpha \le 1\,.$$

The linear "rolloff" region of this LPF extends from $(1-\alpha)f_L$ to $(1+\alpha)f_L$. Make a Python module called `filtfun` and start the module with a function called `trapfilt`. This fcuntion should generate a truncated version $h_T(t)$ of the impulse response given above and use it to filter an input signal $x(t)$ and produce a delay-compensated output signal $y(t)$. More specifically

$$h_T(t) = \begin{cases} h(t)\,, & -k/(2f_L) \le t \le k/(2f_L)\,, \\ 0\,, & \text{otherwise}\,, \end{cases}$$

for some integer k and

$$\angle H_T(f) = 0\,, \quad -f_L \le f \le f_L\,.$$

Note that delay compensation is important, for instance, when processing PAM signals, so that the optimum sampling times are not shifted by the lowpass filter. The header and a skeleton for the `trapfilt` function are shown below.

```
# File: filtfun.py
# Module for filter functions
from pylab import *
from scipy.signal import butter, lfilter
import ecen4652 as ecen

def trapfilt(sig_xt, fL, k, alfa):
    """
    Delay compensated FIR LPF/BPF filter with trapezoidal
    frequency response.
    >>>>> sig_yt, n = trapfilt(sig_xt, fL, k, alfa) <<<<<
    where  sig_yt: waveform from class sigWave
           sig_yt.signal():  filter output y(t), samp rate Fs
           n:    filter order
           sig_xt: waveform from class sigWave
           sig_xt.signal():  filter input x(t), samp rate Fs
           sig_xt.get_Fs():  sampling rate for x(t), y(t)
           fL:   LPF cutoff frequency (-6 dB) in Hz
           k:    h(t) is truncated to |t| <= k/(2*fL)
           alfa: frequency rolloff parameter, linear rolloff
                 over range (1-alfa)fL <= |f| <= (1+alfa)fL
    """
    xt = sig_xt.signal()       # Input signal
    Fs = sig_xt.get_Fs()       # Sampling rate
    ixk = round(Fs*k/float(2*fL))  # Tail cutoff index
    tth = arange(-ixk,ixk+1)/float(Fs)  # Time axis for h(t)
    n = len(tth)-1             # Filter order
    # ***** Generate impulse response ht here *****
    yt = lfilter(ht, 1, hstack((xt, zeros(ixk))))/float(Fs)
                               # Compute filter output y(t)
    yt = yt[ixk:]              # Filter delay compensation
    return ecen.sigWave(yt, Fs, sig_xt.get_t0()), n
                               # Return y(t) and filter order
```

(b) To test `trapfilt`, let $F_s = 16000$ Hz, use a time axis from -0.5 to 0.5 sec, and generate a unit impulse at $t = 0$. Note that this has to be (an approximation to) a CT unit impulse (with area 1 under the impulse). Use this unit impulse as input for `trapfilt` with $f_L = 1000$ Hz, $k = 20$ and $\alpha = 0.2$. Then use your `showft` function to plot the magnitude and the phase of the unit impulse response. Display the magnitude both linear on an absolute scale, and relative in dB. Compare your graphs to the ones given in the introduction. In particular, verify that the passband has a magnitude of 1 and that the transition from passband to stopband is linear. Verify also that the filter is properly delay-compensated (explain how you did this). How do the results change when you set $\alpha = 0$?

(c) The file `pamsig601.wav` contains a noisy polar binary PAM signal. Look at the PSD of this signal and then use `trapfilt` to lowpass filter the signal with $f_L \approx 1000$ Hz, $k \approx 10$, and $\alpha \approx 0.2$. Determine the baud rate F_B (look at the PSD of the lowpass filtered signal squared), and then use this information to display the eye diagram (use a width of 3 T_B and superimpose about 100 traces) of the lowpass filtered PAM signal. Change f_L to find the value that gives you the largest eye opening. How is this f_L related to F_B?

(d) Implementation and Measurement of 'trapfilt' in GNU Radio. Use the header specification shown below to start a new Python module `filtspecs.py` with a function called `trapfilt_taps` that computes the taps for a FIR LPF with trapezoidal frequency response.

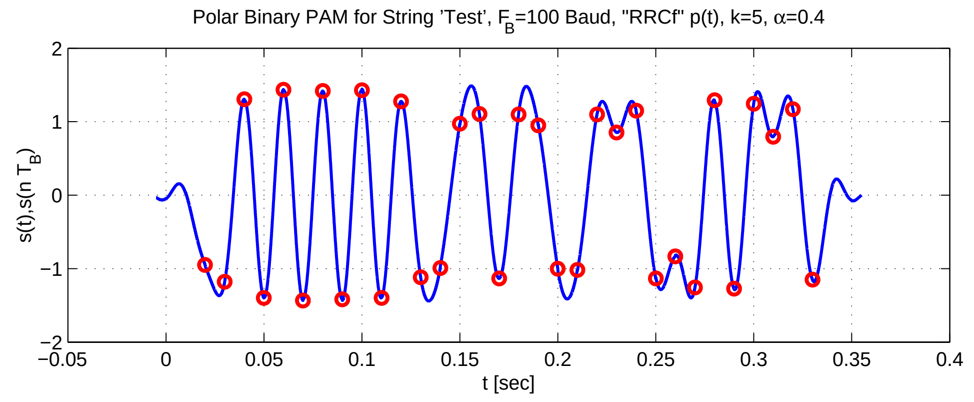
```
# File: filtspecs.py
# Module for filter specfifications
# FIR: Determine filter taps
# IIR: Determine numerator (b) and denominator (a)
# polynomial coefficients
from pylab import *
from scipy.signal import butter

def trapfilt_taps(N, phiL, alfa):
    """
    Returns taps for order N FIR LPF with trapezoidal frequency
    response, normalized cutoff frequency phiL = fL/Fs, and rolloff
    parameter alfa.
    >>>>> hLn = trapfilt_taps(N, phiL, alfa) <<<<<
    where  N:    filter order
           phiL: normalized cutoff frequency (-6 dB)
           alfa: frequency rolloff parameter, linear rolloff
                 over range (1-alfa)phiL <= |f| <= (1+alfa)phiL
    """
```

Place the `filtspecs.py` module in one of the directories in your `PYTHONPATH` and generate a GNU Radio flowgraph that can be used to test the `trapfilt` taps in a decimating or interpolating FIR filter block. Describe your measurement procedure and how the results that you measured verify the correct implementation of the LPF with a trapezoidal frequency response in GNU Radio.

E2. PAM Transmitter/Receiver. (a) Update your `pam11` function to include the **RRCf** (root raised cosine in frequency) pulse which is defined as

$$p(t) = \begin{cases} \dfrac{T_B}{\pi}\,\dfrac{\sin\big((1-\alpha)\pi t/T_B\big) + (4\alpha t/T_B)\,\cos\big((1+\alpha)\pi t/T_B\big)}{\big(1-(4\alpha t/T_B)^2\big)\,t}\,, & t \neq 0, \pm\dfrac{T_B}{4\alpha}\,, \\[2ex] 1-\alpha+\dfrac{4\alpha}{\pi}\,, & t = 0\,, \\[2ex] \dfrac{\alpha}{\sqrt{2}}\Big[\big(1+\dfrac{2}{\pi}\big)\,\sin\big(\dfrac{\pi}{4\alpha}\big) + \big(1-\dfrac{2}{\pi}\big)\,\cos\big(\dfrac{\pi}{4\alpha}\big)\Big]\,, & t = \pm\dfrac{T_B}{4\alpha}\,, \end{cases}$$

where $0 \leq \alpha \leq 1$. Call the new function `pam12` (PAM, V1.2). The `ptype` for the RRCf pulse is `'rrcf'` and the parameter format is `pparms = [k alpha]`, with the same meanings as for the RCf pulse. An example of polar binary PAM using an RRCf pulse with $\alpha = 0.4$ and "tail length" $k = 4$ is shown in the following graph.



Note that the RRCf pulse has ISI at the sampling time instants. After filtering with a matched filter, however, the ISI disappears.

(b) For the module `pamfun.py`, write a Python function called `pamrcvr10` (PAM receiver V1.0) which implements a receiver, with matched filter and sampler, for (noisy) PAM signals. Here is the header for this function:

```
def pamrcvr10(sig_rt, FBparms, ptype, pparms=[]):
    """
    Pulse amplitude modulation receiver with matched filter:
    r(t) -> b(t) -> bn.
    V1.0 for 'man', 'rcf', 'rect', 'rrcf', 'sinc', and 'tri'
    pulse types.
    >>>>> sig_bn, sig_bt, ixn = pamrcvr10(sig_rt, FBparms, ptype, pparms) <<<<<
    where  sig_rt: waveform from class sigWave
           sig_rt.signal():   received (noisy) PAM signal r(t)
           sig_rt.timeAxis(): time axis for r(t)
           FBparms: = [FB, dly]
           FB:    Baud rate of PAM signal, TB=1/FB
           dly:   sampling delay for b(t) -> b_n as a fraction of TB
                  sampling times are t=n*TB+t0 where t0 = dly*TB
           ptype: pulse type from list
                  ('man','rcf','rect','rrcf','sinc','tri')
           pparms not used for 'man','rect','tri'
           pparms = [k, alpha]  for 'rcf','rrcf'
           pparms = [k, beta]  for 'sinc'
           k:     "tail" truncation parameter for 'rcf','rrcf','sinc'
                  (truncates p(t) to -k*TB <= t < k*TB)
           alpha: rolloff parameter for ('rcf','rrcf'), 0<=alpha<=1
           beta:  Kaiser window parameter for 'sinc'
           sig_bn: sequence from class sigSequ
           sig_bn.signal(): received DT sequence after sampling at t=n*TB+t0
           sig_bt: waveform from class sigWave
           sig_bt.signal(): received PAM signal b(t) at output of matched filter
           ixn:   indexes where b(t) is sampled to obtain b_n
    """
```

The following commands implement `pamrcvr10` for rectangular $p(t)$.

```
    if type(FBparms)==int or type(FBparms)==float:
        FB, t0 = FBparms, 0    # Get FBparms parameters
    else:
        FB, t0 = FBparms[0], 0
        if len(FBparms) > 1:
            t0 = FBparms[1]/float(FB)
    Fs = sig_rt.get_Fs()       # Sampling rate
    rt = sig_rt.signal()       # Received signal r(t)
    tt = sig_rt.timeAxis()     # Time axis for r(t)
    nn0 = int(ceil((tt[0]-t0)*FB))  # First data index
                                    # Integer multiple of 1/FB
    ixnn0 = argmin(abs(tt-(nn0/float(FB)+t0)))
    N = int(floor((tt[-1]-tt[ixnn0])*FB)) + 1  # Number of data symbols
    # ***** Set up matched filter response h_R(t) *****
    ptype = ptype.lower()      # Convert ptype to lowercase
                               # Set left/right limits for p(t)
    if (ptype=='rect'):        # Rectangular p(t)
          kL = -0.5; kR = -kL
    else:
        kL = -1.0; kR = -kL    # Default left/right limits
    ixpL = int(ceil(Fs*kL/float(FB)))  # Left index for p(t) time axis
    ixpR = int(ceil(Fs*kR/float(FB)))  # Right index for p(t) time axis
    ttp = arange(ixpL,ixpR)/float(Fs)  # Time axis for p(t)
    pt = zeros(len(ttp))       # Initialize pulse p(t)
    if (ptype=='rect'):        # Rectangular p(t)
        ix = where(logical_and(ttp>=kL/float(FB), ttp<kR/float(FB)))[0]
        pt[ix] = ones(len(ix))
    else:
        print("ptype '%s' is not recognized" % ptype)
    hRt = pt[::-1]             # h_R(t) = p(-t)
    hRt = Fs/sum(np.power(pt,2.0))*hRt  # h_R(t) normalized
    # ***** Filter r(t) with matched filter h_R(t) *****
    bt = convolve(rt,hRt)/float(Fs)  # Matched filter b(t)=r(t)*h_R(t)
    bt = bt[-ixpL:len(tt)-ixpL]  # Trim b(t)
    # ***** Sample b(t) at t=n*TB+t0 to obtain b_n *****
    ixn = ixnn0 + array(around(Fs*arange(N)/float(FB)),int)
                               # Sampling indexes
    bn = bt[ixn]               # DT sequence sampled at t=n*TB+t0
    return ecen.sigSequ(bn,FB,nn0), ecen.sigWave(bt,Fs,tt[0]), ixn
```

Complete this function by adding the remaining pulse types. Note that the generation of $p(t)$ in `pamrcvr10` is essentially identical to the $p(t)$ generation in `pam12`.

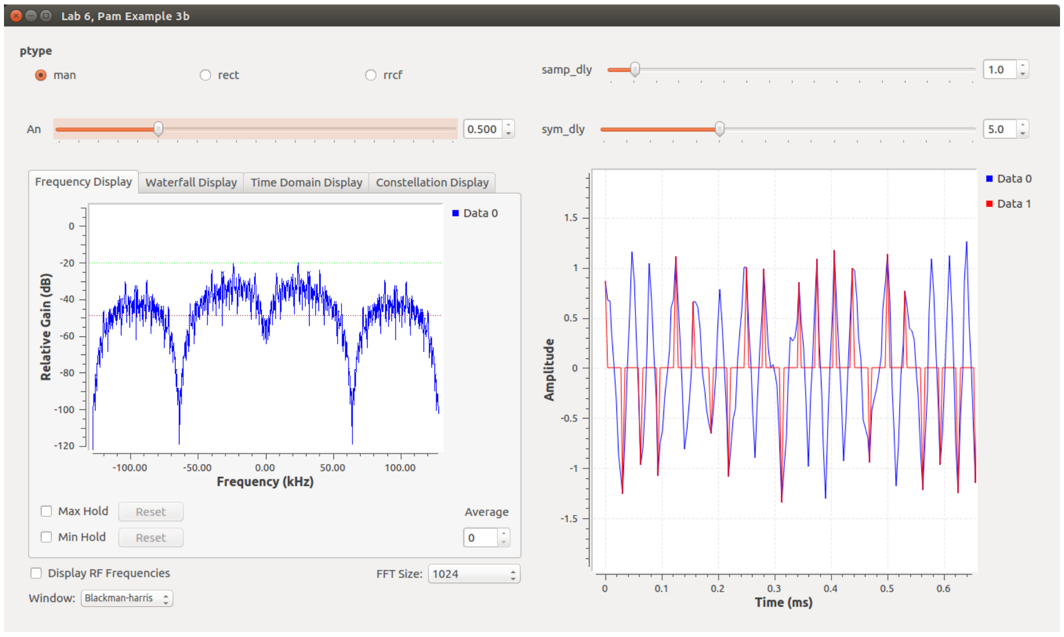
(c) Reception of noisy PAM signals. The wav-files `pamsig601.wav`, `pamsig602.wav`, and `pamsig603.wav` contain received PAM signals $r(t)$ with additive white Gaussian noise. The data in the PAM signals is ASCII text, 8 bits/character, LSB-first, transmitted as polar binary with logical $0 \rightarrow -A$ and logical $1 \rightarrow +A$ where A is an amplitude level that was chosen so that the wav-files (with the noise added) have amplitude less than 1. Use `showpsd` in combination with `trapfilt` and time domain plots to analyze the PAM signals sufficiently so that you can use your `pamrcvr10` function to recover the ASCII text in each case.

(d) Extend your `ptfun.py` module to include a function `pamhRt` for the matched filter response $h_R(t)$ (same as $p(t)$, but time reversed and normalized) and add the `'man'`, `'rcf'`, and `'rrcf'` pulses to both the `pampt` and the `pamhRt` functions. The headers for the functions in the `ptfun.py` module look as follows.

```
# File: ptfun.py
# Functions for gnuradio-companion PAM p(t) and h_R(t)
# (matched filter MF) generation
import numpy as np

def pamhRt(sps, ptype, pparms=[]):
    """
    PAM normalized matched filter (MF) receiver filter
    h_R(t) = h_R(n*TB/sps) generation
    >>>>> hRt = pamhRt(sps, ptype, pparms) <<<<<
    where  sps:
           ptype: pulse type from list
                  ('man', 'rcf', 'rect', 'rrcf', 'sinc', 'tri')
           pparms not used for 'man', 'rect', 'tri'
           pparms = [k, alpha] for 'rcf', 'rrcf'
           pparms = [k, beta] for 'sinc'
           k:     "tail" truncation parameter for 'rcf', 'rrcf', 'sinc'
                  (truncates p(t) to -k*sps <= n < k*sps)
           alpha: Rolloff parameter for 'rcf', 'rrcf', 0 <= alpha <= 1
           beta:  Kaiser window parameter for 'sinc'
           hRt:   MF impulse response h_R(t) at t=n*TB/sps
    Note: In terms of sampling rate Fs and baud rate FB,
          sps = Fs/FB
    """
```

```
def pampt(sps, ptype, pparms=[]):
    """
    PAM pulse p(t) = p(n*TB/sps) generation
    >>>>> pt = pampt(sps, ptype, pparms) <<<<<
    where  sps:
           ptype: pulse type from list
                  ('man', 'rcf', 'rect', 'rrcf', 'sinc', 'tri')
           pparms not used for 'man', 'rect', 'tri'
           pparms = [k, alpha] for 'rcf', 'rrcf'
           pparms = [k, beta] for 'sinc'
           k:     "tail" truncation parameter for 'rcf', 'rrcf', 'sinc'
                  (truncates p(t) to -k*sps <= n < k*sps)
           alpha: Rolloff parameter for 'rcf', 'rrcf', 0 <= alpha <= 1
           beta:  Kaiser window parameter for 'sinc'
           pt:    pulse p(t) at t=n*TB/sps
    Note: In terms of sampling rate Fs and baud rate FB,
          sps = Fs/FB
    """
```

(e) Build the flowgraph referred to as `lab06_pam003` in the introduction, but change the pulse types that can be selected to `'man'`, `'rect'`, and `'rrcf'`. Set the "Default Value" to `'rrcf'`. A typical display resulting from running the flowgraph is shown below.



ptype: [✓ man] rect rrcf

samp_dly: 1.0

An: 0.500

sym_dly: 5.0

Average: 0

FFT Size: 1024

Window: Blackman-harris

Try out all three pulse types as follows. First reduce the noise amplitude `An` to zero and adjust the sampling delay `samp_delay` to its optimal value. Then adjust the `sym_delay` delay such that the correct text is displayed in the terminal at the receiver. Take a screenshot of the frequency and time domain displays when everything is adjusted correctly and include it in your jupyter notebook. Finally, increase the noise amplitude to the point where you just see an occasional error (approximately one error per 100 characters) in the text display. Note the value of `An` where this happens for the different pulse types. Should this value of `An` be the same for all pulse types or should it be different? What is your interpretation?

(f) Build the flowgraph referred to as `lab06_pam004` (with the automatic `samp_delay` adjustment using the "Polyphase Clock Sync") in the introduction, but change the pulse types that can be selected again to `'man'`, `'rect'`, and `'rrcf'`. Set the "Default Value" to `'rrcf'`. Run the flowgraph for `'rrcf'` PAM with `An` set to zero and adjust the `sym_delay` delay to obtain the correct text at the receiver terminal. Then increase `An` until you just see an occasional error in the text display. How does this compare to the value of `An` that you found in part (e)? What happens if you change the pulse type to `'man'` or `'rect'`?

E3. Matched Filters and ISI. (Experiment for ECEN 5002, optional for ECEN 4652) Devise a test using `pam12.m`, `pamrcvr10.m`, and `showeye.m` to determine for which of the pulse types (`'man', 'rcf', 'rect', 'rrcf', 'sinc', 'tri'`) there is ISI (intersymbol interference) either in the received PAM signal $r(t)$ and/or in the signal $b(t)$ after the matched filter. If there is ISI, estimate (in percent of the full size) by how much the "eye" in the eye diagram gets closed because of ISI.

(b) Use the same setup as for E2e, but use `'tri'`, `'rcf'`, and `'rrcf'` as the pulse types that can be selected (with `'rrcf'` as default). For each pulse type adjust `samp_delay` and `sym_delay` to their optimum values when `An=0`. Then increase `An` until to the point where you just see an occasional error in ther terminal that displays the received text. How and why do the corresponding `An` values for the three different pulse types differ? What happens if you set `alpha=0` for the `'rcf'` pulse? Explain!

(c) Repeat (b) for the flowgraph with the "Polyphase Clock Sync" (flowgraph `lab06_pam004`). What are reasons why the automatic clock sync might fail?