Motivating Undergraduate Communication Theory Using GNU Radio

Peter Mathys MATHYS@COLORADO.EDU
University of Colorado Boulder, Department of ECEE, UCB 425, Boulder, CO 80309-0425 USA

Abstract

Typical undergraduate communication theory textbooks start from amplitude, frequency, and phase modulation using continuous time signal processing, followed by an introduction to random processes, before digital (and discrete time) communications systems are introduced in the second half of the book. This has led to a drop in enrollment in our undergraduate communications track sequence, largely due to students being overwhelmed with the mathematics of linear systems and probability theory before they have been motivated enough by seeing the applicability of communication theory to the wireless devices that they have come to embrace in their daily lives. We promote a modified approach that uses GNU Radio flowgraphs early in the course to build and analyze a basic digital communication system in parallel with developing the necessary communication theory background. Since GNU Radio is a complex and sophisticated software development toolkit, we place a lot of emphasis on making it a tool that is as transparent and as easy to use as possible by creating carefully designed hierarchical blocks that partition the communication system into subsystems with well defined interfaces and parameter settings.

1. Introduction

Typical undergraduate communication theory textbooks (Lathi & Ding, 2009), (Haykin & Moher, 2009), (Proakis & Salehi, 2002), start out with a review of continuous time linear systems, followed by amplitude, frequency and phase modulation. In some cases, probability theory and random processes also appear prominently at the beginning of the book. Digital data communication and digital signal processing are usually deferred until the second half of the course and the first complete digital communication system that resembles what is actually used in smartphones and other modern wireless systems appears only some 300 to 400 pages into the book. That makes it difficult to motivate students who are hesitant to begin from a mostly abstract mathematical exposition to the topic and we are thus promoting an approach where we start with a simple ideal communication system, e.g., using binary phase shift keying to transmit ASCII code text messages. In subsequent steps we can then introduce practical constraints and impairments such as channel bandwidth, noise, and timing synchronization. Along the way such concepts as the matched filter, signal space, and phase locked loops can be introduced naturally. To give the students opportunity to experiment and explore 'what-if' scenarios, GNU Radio and the GNU Radio Companion (GRC) provide an ideal and very affordable platform. But there is a 'chicken and egg' problem. If you already know communication theory, GNU Radio is a great tool for experimentation, but if you are new to the field there is a steep learning curve. Just to demonstrate the concept of signal space and what happens if there is noise and the transmitter and receiver are not exactly synchronized, one quickly fills an entire flowgraph screen with some 30 blocks whose function is not always self-evident. Thus, some tailored blocks along the lines of an idealized textbook exposition to communications are needed to demonstrate the applicability of the material and to let students gain confidence in their ability to analyze and design such systems. In the following sections we develop GNU Radio blocks with well defined interfaces and parameters to explore a basic digital communication system in four acts (which could easily be extended if there was no page limit).

2. Basic Digital Communication in Four Acts

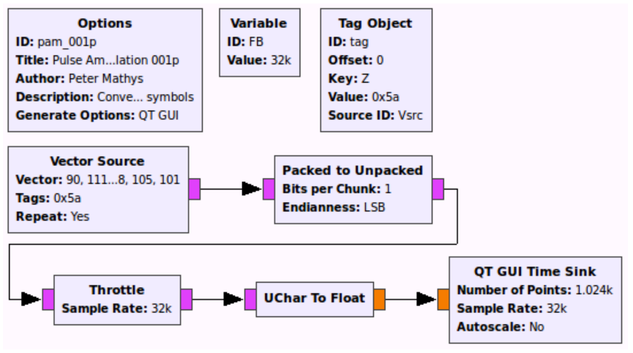
You are new to communications and you are given the task to transmit a text, say `'Zombie'`, over a pair of wires. What are the fundamental operations that you need to perform in order to accomplish this task? We break this up into four acts. In act 1 the ASCII coded text is converted to a serial stream of M-ary symbols, e.g., for $M = 2, 4$. In act 2 the M-ary symbols are converted to waveforms using pulse amplitude modulation (PAM) with different pulse shapes. In act 3 we transmit the PAM signal over a noisy

Proceedings of the 6^{th} GNU Radio Conference, Copyright 2016 by the author(s).

channel and observe the resulting errors. In act 4 we develop the matched filter as the receiver's solution to maximize the signal-to-noise ratio (SNR).

2.1. Act 1, ASCII and Parallel Serial Conversion

The ASCII (American Standard Code for Information Interchange) is a well-established 7-bit code for standard English text, punctuation, and symbols. Because digital computers typically use wordlengths in multiples of 8 bits, ASCII encoded text usually uses 8 bits per character with the most significant bit (MSB) set to 0. In hexadecimal notation, the code for `Zombie` is `5a, 6f, 6d, 62, 69, 65`. To transmit these codes we have to break them up into symbols that can be sent to the receiver. In the simplest case we just transmit the 0's and 1's that make up the 8 bits, but we might also be able to combine pairs of bits and transmit symbols for 00, 01, 10, and 11 (e.g., as 0, 1, 2, and 3). We call these M-ary symbols with $M = 2$ for binary and $M = 4$ for pairs of bits or $M = 3$ for triplets of bits, etc. These symbols have to be sent serially to the receiver and the process of converting from ASCII code to M-ary symbols is called parallel-to-serial conversion. Similarly, the process of reconstructing ASCII codes from received symbols is called serial-to-parallel conversion. The rate at which symbols are transmitted is called Baud rate (or symbol rate) and denoted with the symbol F_B. In the GRC we can use the flowgraph in Figure 1 to generate a stream of binary symbols from the text `Zombie`.


Options
ID: pam_001p
Title: Pulse Am...lation 001p
Author: Peter Mathys
Description: Conve... symbols
Generate Options: QT GUI

Variable
ID: FB
Value: 32k

Tag Object
ID: tag
Offset: 0
Key: Z
Value: 0x5a
Source ID: Vsrc

Vector Source
Vector: 90, 111...8, 105, 101
Tags: 0x5a
Repeat: Yes

Packed to Unpacked
Bits per Chunk: 1
Endianness: LSB

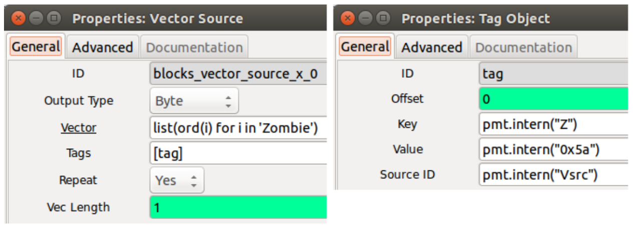
Throttle
Sample Rate: 32k

UChar To Float

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

Figure 1. Parallel to Serial Conversion of ASCII Text

Each GRC flowgraph has an options block which is the top-level hierarchical block representing the flowgraph. It is used to set global parameters, e.g., to use the QT GUI for the display of graphs, or specification of parameters during flowgraph execution. The Vector Source is used to generate the ASCII code bytes. We also specified a tag using the Tag Object so that we can identify the beginning of the text later in a time display. The detailed settings for these two blocks are shown in Figure 2.



ID	blocks_vector_source_x_0
Output Type	Byte
Vector	list(ord(i) for i in 'Zombie')
Tags	[tag]
Repeat	Yes
Vec Length	1

ID	tag
Offset	0
Key	pmt.intern("Z")
Value	pmt.intern("0x5a")
Source ID	pmt.intern("Vsrc")

Figure 2. Properties of Vector Source and Tag Object

The Packed to Unpacked block performs the actual parallel-to-serial conversion. Specifying 1 bit per chunk creates binary symbols, 2 bits per chunk would create 4-ary symbols, etc. Whenever a parallel-to-serial conversion is performed there are two ends to start from, either LSB (least significant bit) first or MSB first. In the Packed to Unpacked block this is specified as LSB (1) or MSB (0) Endianness. The Throttle block together with the `FB` Variable limits the flow of the symbols generated to the specified Baud rate F_B. Finally, to check that the correct sequence of symbols is generated, we use a QT GUI Time Sink. Since this block only accepts Float or Complex number inputs, it is preceded by a UChar to Float type converter. The graph of the generated binary time sequence is shown in Figure 3.

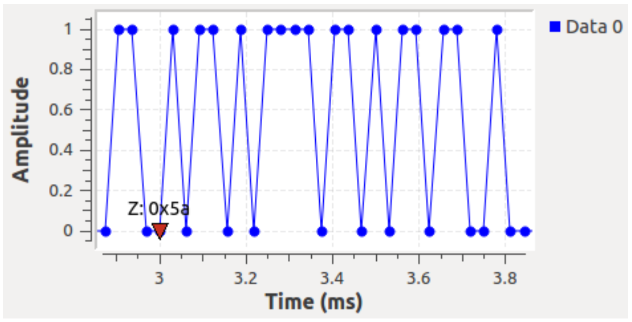
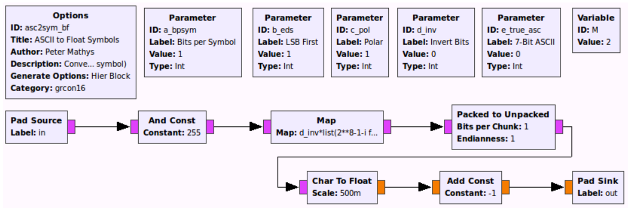


Figure 3. Generated Serial Stream of Binary Symbols

Note the tag `Z:0x5a` that was set in the Vector Source to indicate the beginning of the text (which is repeated over and over again). To trigger on the tag and to display the actual sample points as circles, click on the middle mouse button and select the appropriate menu item. Use the mouse to select and zoom in on an area of the graph.

Now that we have the basic conversion from parallel ASCII to serial symbols in place, we see that there are a number of parameters that could be set in different ways and if we want our transmitter to be reasonably universal we need some additional settings. These include (i) the number of bits per symbol, (ii) LSB or MSB first, (iii) unipolar (e.g., 0,1 or 0,1,2,3) or polar (e.g., -1,+1 or -3,-1,+1,+3), (iv) bit inversion of the ASCII code (to accomodate polarity reversals), and (v) 8-bit or 7-bit (true) ASCII code generation. In addition we may want to send other data than ASCII text, e.g., random bytes for making different kinds of measurements. This will require a few additional blocks and it is cumbersome to repeat these every time we need a serial symbol stream. Luckily, the GRC includes a feature where several blocks can be combined into one hierarchical block with specified inputs, outputs, and parameters. This is shown in the next flowgraph in Figure 4 for a ASCII to Float Symbols conversion block which incorporates all the additional options that we listed above in the form of parameters.



Options
ID: asc2sym_bf
Title: ASCII to Float Symbols
Author: Peter Mathys
Description: Conve... symbol)
Generate Options: Hier Block
Category: grcon16

Parameter
ID: a_bpsym
Label: Bits per Symbol
Value: 1
Type: Int

Parameter
ID: b_eds
Label: LSB First
Value: 1
Type: Int

Parameter
ID: c_pol
Label: Polar
Value: 1
Type: Int

Parameter
ID: d_inv
Label: Invert Bits
Value: 0
Type: Int

Parameter
ID: e_true_asc
Label: 7-Bit ASCII
Value: 0
Type: Int

Variable
ID: M
Value: 2

Pad Source
Label: in

And Const
Constant: 255

Map
Map: d_inv*list(2**8-1-i f...

Packed to Unpacked
Bits per Chunk: 1
Endianness: 1

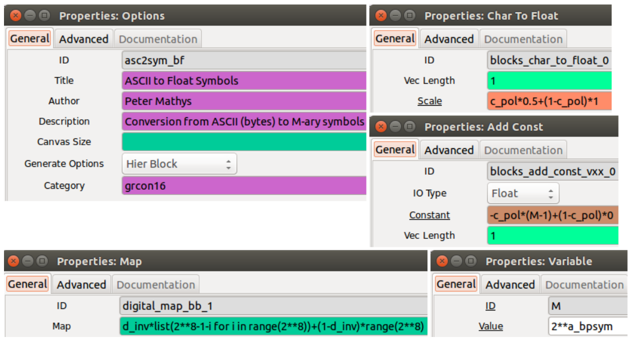
Char To Float
Scale: 500m

Add Const
Constant: -1

Pad Sink
Label: out

Figure 4. ASCII to Float Symbols Hierarchical Block

The type of the top-level Options block is now set to Hier Block. The number M of symbols generated is computed as `2**a_bpsym`. This, together with the properties of the other blocks is shown in Figure 5.



Properties: Options
General | Advanced | Documentation

ID	asc2sym_bf
Title	ASCII to Float Symbols
Author	Peter Mathys
Description	Conversion from ASCII (bytes) to M-ary symbols
Canvas Size	
Generate Options	Hier Block
Category	grcon16

Properties: Char To Float
General | Advanced | Documentation

ID	blocks_char_to_float_0
Vec Length	1
Scale	c_pol*0.5+(1-c_pol)*1

Properties: Add Const
General | Advanced | Documentation

ID	blocks_add_const_vxx_0
IO Type	Float
Constant	-c_pol*(M-1)+(1-c_pol)*0
Vec Length	1

Properties: Map
General | Advanced | Documentation

ID	digital_map_bb_1
Map	d_inv*list(2**8-1-i for i in range(2**8))+(1-d_inv)*range(2**8)

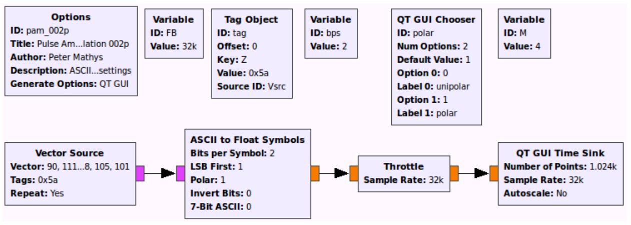
Properties: Variable
General | Advanced | Documentation

ID	M
Value	2**a_bpsym

Figure 5. Properties of Blocks in ASCII to Float Symbols Block

Now we can use this hierarchical block to generate different forms of symbol streams for transmission from a given source. Some of the parameters, like polar versus unipolar can be changed on the fly and the effect can be viewed directly as the simulation is ongoing. The next flowgraph, shown in Figure 6, uses the text `Zombie` to generate 4-ary symbols with selectable unipolar (0,1,2,3) or polar (−3,−1,+1,+3) values.

The Bits per Symbol parameter in the ASCII to Float Symbols block is set to the Variable `bps` and the Po-



Options
ID: pam_002p
Title: Pulse Am...lation 002p
Author: Peter Mathys
Description: ASCII...settings
Generate Options: QT GUI

Variable
ID: FB
Value: 32k

Tag Object
ID: tag
Offset: 0
Key: Z
Value: 0x5a
Source ID: Vsrc

Variable
ID: bps
Value: 2

QT GUI Chooser
ID: polar
Num Options: 2
Default Value: 1
Option 0: 0
Label 0: unipolar
Option 1: 1
Label 1: polar

Variable
ID: M
Value: 4

Vector Source
Vector: 90, 111...8, 105, 101
Tags: 0x5a
Repeat: Yes

ASCII to Float Symbols
Bits per Symbol: 2
LSB First: 1
Polar: 1
Invert Bits: 0
7-Bit ASCII: 0

Throttle
Sample Rate: 32k

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

Figure 6. 4-Ary Symbol Generation Unipolar and Polar Selectable

lar parameter is set to the variable `polar` which is the ID of the QT GUI Chooser. The resulting graph in the polar setting is shown in Figure 7 in the form of a stem plot (using the middle mouse button for the graph type selection). Note that `'Z'=5a` and `'o'=6f` convert to `-1,-1,+1,+1,+3,+3,-1,+1` 4-ary polar and LSB first ASCII.

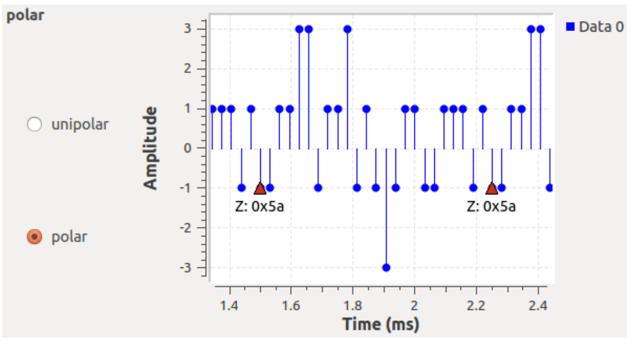


Figure 7. 4-Ary Polar Symbol Sequence for `'Zombie'`

Now we are ready to tackle the reverse problem of receiving a serial stream of symbols and converting them back to parallel bytes that represent ASCII characters. We use the same general principle of identifying the parameters that may need adjustment as for the parallel-to-serial conversion and then building a hierarchical block that can accomplish the task. At the receiver there are some new challenges in addition to the possible options that we considered for the transmitter. First of all, the signal level will most likely change along any realistic transmission path and we need an adjustable gain parameter. Then there is the important issue of synchronization. If we use binary transmission as an example, then each ASCII character is 8 bits long. For various reasons we may lose a few bits during transmission and the first bit that arrives may not be at a boundary between ASCII characters. Thus, an adjustable symbol delay is needed so that synchronism can be established at the receiver. A hierarchical block that can accomplish this task is shown in Figure 8.

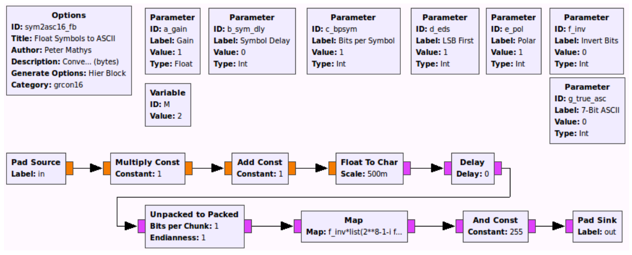

Options
ID: sym2asc16_fb
Title: Float Symbols to ASCII
Author: Peter Mathys
Description: Conve... (bytes)
Generate Options: Hier Block
Category: grcon16
Parameter ID: a_gain Label: Gain Value: 1 Type: Float
Parameter ID: b_sym_dly Label: Symbol Delay Value: 0 Type: Int
Parameter ID: c_bpsym Label: Bits per Symbol Value: 1 Type: Int
Parameter ID: d_eds Label: LSB First Value: 1 Type: Int
Parameter ID: e_pol Label: Polar Value: 1 Type: Int
Parameter ID: f_inv Label: Invert Bits Value: 0 Type: Int
Parameter ID: g_true_asc Label: 7-Bit ASCII Value: 0 Type: Int
Variable ID: M Value: 2
Pad Source Label: in
Multiply Const Constant: 1
Add Const Constant: 1
Float To Char Scale: 500m
Delay Delay: 0
Unpacked to Packed Bits per Chunk: 1 Endianness: 1
Map Map: f_inv*list(2**8-1-i f...
And Const Constant: 255
Pad Sink Label: out

Figure 8. Float Symbols to ASCII Conversion Hierarchical Block

The next interesting question is how to display the output of the Float Symbols to ASCII conversion block. To complete this first act of exploring the basics of communication with a sense of accomplishment we don't want to see just numbers or graphs, we want to be able to read the text (which we of course already know) at the receiving end. Luckily Linux treats all devices, including the terminal, as files. Thus, we can use a File Sink and use `/dev/pts/n`, where `n` is a number, as file name. To obtain `n` for a specific terminal, open it and type `tty` at the command prompt. The flowgraph in Figure 9 shows our completed discrete-time baseband system for transmitting an ASCII text using serial M-ary symbol transmission.

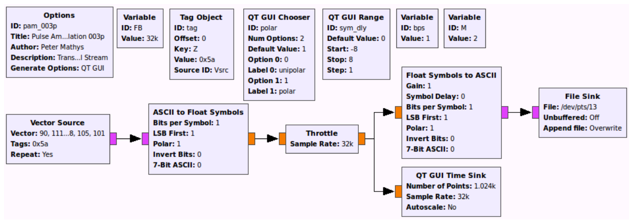
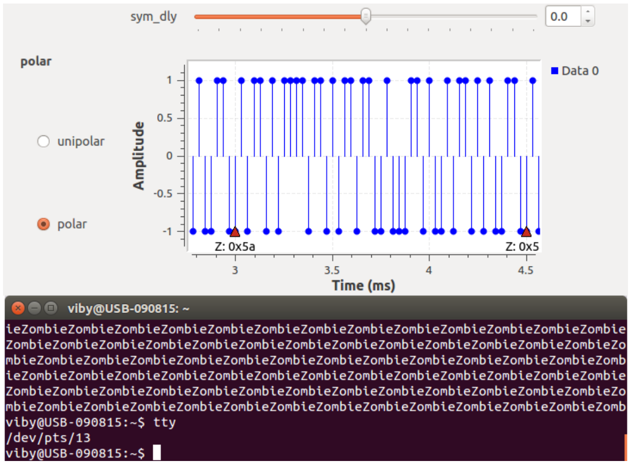

Options
ID: pam_003p
Title: Pulse Am...lation 003p
Author: Peter Mathys
Description: Trans...l Stream
Generate Options: QT GUI
Variable ID: FB Value: 32k
Tag Object ID: tag Offset: 0 Key: Z Value: 0x5a Source ID: Vsrc
QT GUI Chooser ID: polar Num Options: 2 Default Value: 1 Option 0: 0 Label 0: unipolar Option 1: 1 Label 1: polar
QT GUI Range ID: sym_dly Default Value: 0 Start: -8 Stop: 8 Step: 1
Variable ID: bps Value: 1
Variable ID: M Value: 2
Vector Source Vector: 90, 111...8, 105, 101 Tags: 0x5a Repeat: Yes
ASCII to Float Symbols Bits per Symbol: 1 LSB First: 1 Polar: 1 Invert Bits: 0 7-Bit ASCII: 0
Throttle Sample Rate: 32k
Float Symbols to ASCII Gain: 1 Symbol Delay: 0 Bits per Symbol: 1 LSB First: 1 Polar: 1 Invert Bits: 0 7-Bit ASCII: 0
File Sink File: /dev/pts/13 Unbuffered: Off Append file: Overwrite
QT GUI Time Sink Number of Points: 1.024k Sample Rate: 32k Autoscale: No

Figure 9. Discrete Time Serial Baseband Transmission System

You can try changing the number of bits per symbol, change the Symbol Delay at the receiver, change polar to unipolar, etc. It is also instructive to see how garbled up the received text becomes when there is a mismatch in the parameters between the transmit and receive sides. If all goes well the QT GUI Time Sink display and the tty terminal display should look as shown in Figure 10.



```
ieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombie
ZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZo
mbieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZomb
ieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombie
ZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZo
mbieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZomb
viby@USB-090815:~$ tty
/dev/pts/13
viby@USB-090815:~$
```

Figure 10. Received Signal in the Time Domain and on the tty Screen

2.2. Act 2, Symbol to Waveform Conversion

A pair of wires is an analog communication channel that needs to have a defined signal level at all instants of time. Thus, a discrete-time (DT) sequence of symbols needs to be converted to a continuous-time (CT) waveform for proper transmission. A simple way to accomplish this is to use pulse amplitude modulation (PAM) which can be expressed mathematically as

$$s(t) = \sum_n a_n\, p(t - nT_B) \tag{1}$$

where a_n are the values of the DT symbols, $T_B = 1/F_B$ is the symbol time, $p(t)$ is a CT pulse shape and $s(t)$ is the CT PAM signal. Inside GNU Radio all signals are sampled and we cannot have a true CT signal. However, we can use a sampling rate F_s that is sufficiently larger than the Baud rate F_B to obtain "CT" waveforms that work well for simulation and visualization purposes. The ratio F_s/F_B is equal to the number of samples per symbol (F_s/F_B = `sps`). For actual software radio implementations `sps` may be chosen as low as 2. But to generate "nice" looking waveforms during simulations `sps` should be chosen higher, e.g., `sps=10`. Figure 11 shows the hierarchical block that we use to produce PAM waveforms for different pulse types.

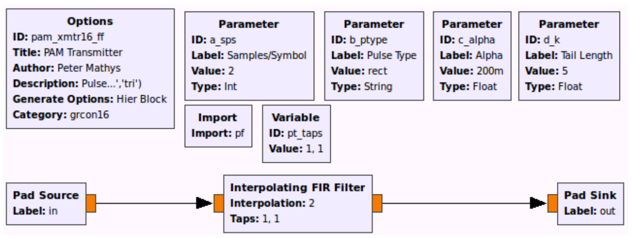
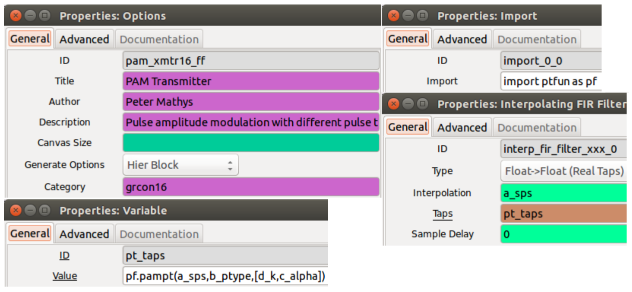

Options
ID: pam_xmtr16_ff
Title: PAM Transmitter
Author: Peter Mathys
Description: Pulse...','tri')
Generate Options: Hier Block
Category: grcon16
Parameter ID: a_sps Label: Samples/Symbol Value: 2 Type: Int
Parameter ID: b_ptype Label: Pulse Type Value: rect Type: String
Parameter ID: c_alpha Label: Alpha Value: 200m Type: Float
Parameter ID: d_k Label: Tail Length Value: 5 Type: Float
Import Import: pf
Variable ID: pt_taps Value: 1, 1
Pad Source Label: in
Interpolating FIR Filter Interpolation: 2 Taps: 1, 1
Pad Sink Label: out

Figure 11. PAM Transmitter Hierarchical Block: DT Symbol to CT Waveform Conversion

The increase in sampling rate from F_B to F_s for the DT to CT conversion is achieved using an Interpolating FIR Filter with the Interpolation value set to `sps`. The Property settings of the blocks in the PAM Transmitter Hierarchical Block are shown in Figure 12.



ID: pam_xmtr16_ff
Title: PAM Transmitter
Author: Peter Mathys
Description: Pulse amplitude modulation with different pulse t
Canvas Size:
Generate Options: Hier Block
Category: grcon16

ID: import_0_0
Import: import ptfun as pf

ID: interp_fir_filter_xxx_0
Type: Float->Float (Real Taps)
Interpolation: a_sps
Taps: pt_taps
Sample Delay: 0

ID: pt_taps
Value: pf.pampt(a_sps,b_ptype,[d_k,c_alpha])

Figure 12. Block Properties of PAM Transmitter Blocks

The most important part is setting the Taps of the Interpolating FIR Filter to obtain a PAM pulse $p(t)$ of the desired shape. This is done through the Variable `pt_taps` which generates the taps from our own Python module `ptfun.py` that contains the function `pampt`:

```
# File: ptfun.py
import numpy as np
def pampt(sps, ptype, pparms=[]):
  if ptype == 'rect':
    nn = np.arange(sps)
    pt = np.ones(len(nn))
  elif ptype == 'rcf':
    nk = round(pparms[0]*sps)
    nn = np.arange(-nk,nk)
    pt = np.sinc(nn/float(sps))
    if len(pparms) > 1:
      p2t = 0.25*np.pi*np.ones(len(nn))
      atFB = pparms[1]/float(sps)*nn
      atFB2 = np.power(2*atFB,2.0)
      ix = np.where(atFB2 != 1)[0]
      p2t[ix] = np.cos(np.pi*atFB[ix])
      p2t[ix] = p2t[ix]/(1-atFB2[ix])
      pt = pt*p2t
  else:
    pt = np.ones(1)  # default value
  return pt
```

This generates the taps for rectangular (`'rect'`) and raised cosine in frequency (`'rcf'`) PAM pulses. Other pulses of interest which we included in `pampt` are the manchester (`'man'`) and triangular (`'tri'`) PAM pulses. The simplest strategy is to use rectangular pulses, i.e., keep the level of the PAM signal $s(t)$ at value a_n for the n-th symbol interval. But looking at the Fourier transform (FT)

$$S(f) = \int_{-\infty}^{\infty} s(t)\, e^{-j2\pi f t}\, dt = A(fT_B)\, P(f)$$

(where $A(fT_B)$ is the DT-FT of a_n) of a general PAM signal $s(t)$, we see that the spectrum of the signal is shaped by the FT $P(f)$ of the pulse $p(t)$. For a rectangular pulse of amplitude 1 and width T_B, $P(f) = \sin(\pi f T_B)/(\pi f)$ which has a 95% power bandwidth of $2F_B$. To reduce the bandwidth to $F_B/2$, the `'sinc'` pulse $p(t) = \sin(\pi t/T_B)/(\pi t/T_B)$, or the related `'rcf'` pulse

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

with bandwidth control parameter $0{<=}\alpha{<=}1$ can be used (the case $\alpha{=}0$ corresponds to the `'sinc'` pulse). The -6 dB bandwidth of `'rcf'` PAM pulses is $(1{+}\alpha)F_B/2$. Plots of `'rcf'` pulses in the time domain are shown in Figure 13 for different values of α.

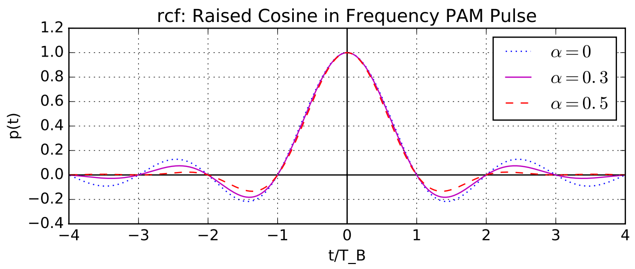


Figure 13. Raised Cosine in Frequency Pulses for Different Values of α

Note that ideal `'rcf'` pulses extend from $t = -\infty$ to $t = +\infty$. For practical purposes the "tails" of the pulses must be truncated, e.g., to k zero crossings to the left and right of the main lobe as we did in the `pampt` function. Thus, for `'sinc'` and `'rcf'` we use `pparms`$= [k, \alpha]$, whereas for `'man'`, `'rect'` and `'tri'` we set `pparms`$= []$.

The flowgraph in Figure 14 implements a PAM transmitter for different pulse shapes and an impulse sampling receiver. Impulse sampling is implemented using a Decimating FIR filter with decimation factor `sps` and a unit impulse response function $h_n = \delta_n$. That is the equivalent of using a switch that closes for one time period $T_s = 1/F_s$ and stays open for sps-1 time periods T_s. An adjustable time delay is put in front of this filter so that the optimal sampling time (e.g., tip of triangular pulse) can be chosen.

The sampled signal at rate F_B is passed on to the Float Symbols to ASCII conversion block. It is also upsampled again by `sps` with the help of an Interpolating FIR Filter with impulse response δ_n so that the received waveform and its sampled (at rate F_B) version can be viewed together using a 2-input QT GUI Time Sink (note that the Time Sink does not work well when it receives signals with different sampling rates). Figure 15 shows the spectrum of a rectangular polar binary PAM signal on the left and the received PAM signal (blue) and its sampled version (red) in the time domain on the right side. The correct sampling time instants are easily found by adjusting the sampling delay and

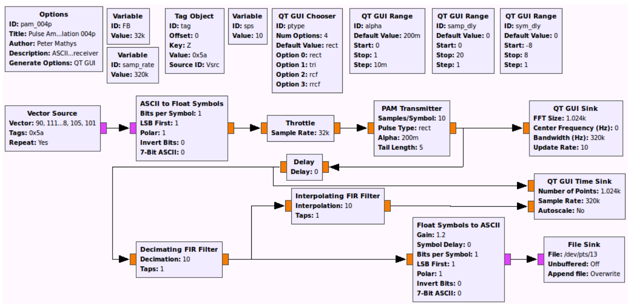

Options
ID: pam_004p
Title: Pulse Am...lation 004p
Author: Peter Mathys
Description: ASCII...receiver
Generate Options: QT GUI
Variable ID: FB Value: 32k
Variable ID: samp_rate Value: 320k
Tag Object ID: tag Offset: 0 Key: Z Value: 0x5a Source ID: Vsrc
Variable ID: sps Value: 10
QT GUI Chooser ID: ptype Num Options: 4 Default Value: rect Option 0: rect Option 1: tri Option 2: rcf Option 3: rrcf
QT GUI Range ID: alpha Default Value: 200m Start: 0 Stop: 1 Step: 10m
QT GUI Range ID: samp_dly Default Value: 0 Start: 0 Stop: 20 Step: 1
QT GUI Range ID: sym_dly Default Value: 0 Start: -8 Stop: 8 Step: 1
Vector Source Vector: 90, 111...8, 105, 101 Tags: 0x5a Repeat: Yes
ASCII to Float Symbols Bits per Symbol: 1 LSB First: 1 Polar: 1 Invert Bits: 0 7-Bit ASCII: 0
Throttle Sample Rate: 32k
PAM Transmitter Samples/Symbol: 10 Pulse Type: rect Alpha: 200m Tail Length: 5
QT GUI Sink FFT Size: 1.024k Center Frequency (Hz): 0 Bandwidth (Hz): 320k Update Rate: 10
Delay Delay: 0
QT GUI Time Sink Number of Points: 1.024k Sample Rate: 320k Autoscale: No
Interpolating FIR Filter Interpolation: 10 Taps: 1
Decimating FIR Filter Decimation: 10 Taps: 1
Float Symbols to ASCII Gain: 1.2 Symbol Delay: 0 Bits per Symbol: 1 LSB First: 1 Polar: 1 Invert Bits: 0 7-Bit ASCII: 0
File Sink File: /dev/pts/13 Unbuffered: Off Append file: Overwrite

Figure 14. ASCII Transmission Using PAM and Impulse Sampling at Receiver

observing the locations of the resulting samples (shown as red diamonds in the graph).

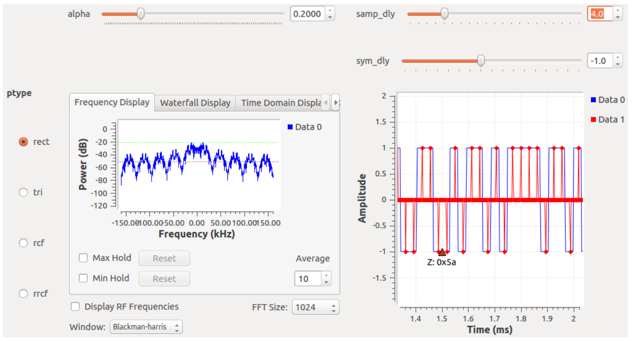


Figure 15. Noiseless PAM Signal for Rectangular Pulse Shape

2.3. Act 3, Channel Noise

Now let's see what happens if we use impulse sampling at the receiver when the PAM signal is transmitted through a noisy channel. In practical implementations a substantial portion of the noise is actually generated by the circuitry of the receiver front end, but for the analysis of communication systems it is more convenient to attribute all the noise to the channel. The flowgraph in Figure 16 is similar to the one in Figure 14, with the exception of the Fast Noise Source that has been added. This source generates white Gaussian noise and the channel model that results from adding this noise to the transmitted waveform is called AWGN (additive white Gaussian noise) model.

The amplitude An of the noise is controlled by one of the QT GUI Range sliders. The graphs in Figure 17 show the spectrum on the left, and the received as well as the sampled PAM signal on the right when An=0.32 (corresponding to a noise power of 0.1).

Figure 18 shows the resulting errors in the received ASCII

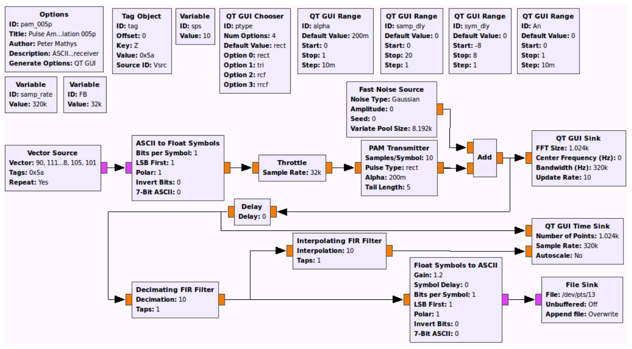

Options
ID: pam_005p
Title: Pulse Am...lation 005p
Author: Peter Mathys
Description: ASCII...receiver
Generate Options: QT GUI
Tag Object ID: tag Offset: 0 Key: Z Value: 0x5a Source ID: Vsrc
Variable ID: sps Value: 10
QT GUI Chooser ID: ptype Num Options: 4 Default Value: rect Option 0: rect Option 1: tri Option 2: rcf Option 3: rrcf
QT GUI Range ID: alpha Default Value: 200m Start: 0 Stop: 1 Step: 10m
QT GUI Range ID: samp_dly Default Value: 0 Start: 0 Stop: 20 Step: 1
QT GUI Range ID: sym_dly Default Value: 0 Start: -8 Stop: 8 Step: 1
QT GUI Range ID: An Default Value: 0 Start: 0 Stop: 1 Step: 10m
Variable ID: samp_rate Value: 320k
Variable ID: FB Value: 32k
Fast Noise Source Noise Type: Gaussian Amplitude: 0 Seed: 0 Variate Pool Size: 8.192k
Vector Source Vector: 90, 111...8, 105, 101 Tags: 0x5a Repeat: Yes
ASCII to Float Symbols Bits per Symbol: 1 LSB First: 1 Polar: 1 Invert Bits: 0 7-Bit ASCII: 0
Throttle Sample Rate: 32k
PAM Transmitter Samples/Symbol: 10 Pulse Type: rect Alpha: 200m Tail Length: 5
Add
QT GUI Sink FFT Size: 1.024k Center Frequency (Hz): 0 Bandwidth (Hz): 320k Update Rate: 10
Delay Delay: 0
QT GUI Time Sink Number of Points: 1.024k Sample Rate: 320k Autoscale: No
Interpolating FIR Filter Interpolation: 10 Taps: 1
Decimating FIR Filter Decimation: 10 Taps: 1
Float Symbols to ASCII Gain: 1.2 Symbol Delay: 0 Bits per Symbol: 1 LSB First: 1 Polar: 1 Invert Bits: 0 7-Bit ASCII: 0
File Sink File: /dev/pts/13 Unbuffered: Off Append file: Overwrite

Figure 16. PAM Transmitter, Channel Noise, and Impulse Sampling at Receiver

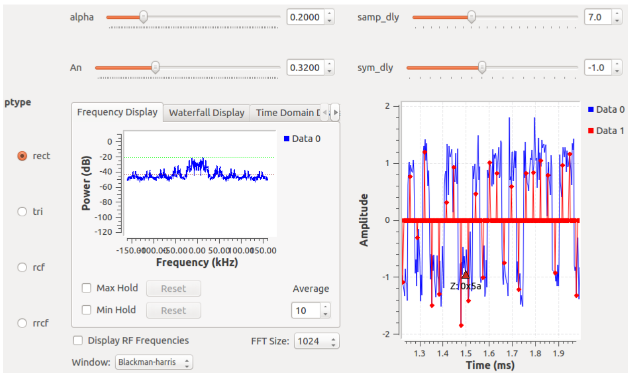


Figure 17. Noisy PAM Spectrum and Waveform for Rectangular Pulse Type

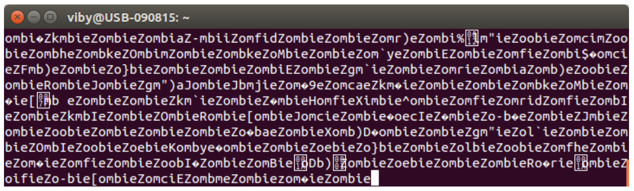
text.

This is clearly non-negligible and depends very much on whether the one sample per symbol that the receiver uses happens to be affected or spared by the noise.

2.4. Act 4, Matched Filter and SNR

The flaw of impulse sampling at the receiver is that it relies on a single sample per symbol rather than taking into account that the waveforms of all possible symbols are actually known to both transmitter and receiver. Thus, the key idea is to use a filter with impulse response $h_R(t)$ at the receiver that somehow incorporates this knowledge before sampling at rate F_B. The block diagram of a PAM communication system is shown in Figure 19.

To keep things simple, it is assumed that only a single symbol with value a_0 is transmitted over a noisy channel with impulse response $h_C(t)$. The received noisy signal is $r(t) = a_0 p_C(t) + n(t)$, where $p_C(t) = p(t) * h_C(t)$ is the convolution of the PAM pulse $p(t)$ with the impulse re-



```
viby@USB-090815: ~
ombi�ZkmbieZombieZombiaZ-mbiiZomfidZombieZombieZomr)eZombi%▯m"ieZoobieZomcimZoo
bieZombheZombkeZOmbimZombieZombkeZoMbieZombieZom`yeZombiEZombieZomfieZombi$�omci
eZFmb)eZombieZo}bieZombieZombieZombiEZombieZgm`ieZombieZomrieZombiaZomb)eZoobieZ
ombieRombieJombieZgm")aJombieJbmjieZom�9eZomcaeZkm�ieZombieZombieZombkeZoMbieZom
�ie[▯b eZombieZombieZkm`ieZombieZ�mbieHomfieXimbie^ombieZomfieZomridZomfieZombI
eZombieZkmbIeZombieZOmbieRombie[ombieJomcieZombie�oecIeZ�mbieZo-b�eZombieZJmbieZ
ombieZoobieZombieZombieZombieZo�baeZombieXomb)D�ombieZombieZgm"ieZol`ieZombieZom
bieZOmbIeZoobieZoebieKombye�ombieZombieZoebieZo}bieZombieZolbieZoobieZomfheZombi
eZom�ieZomfieZombieZoobI�ZombieZomBie▯Db)▯ombieZoebieZombieZombieRo�rie▯mbieZ
oifieZo-bie[ombieZomciEZombmeZombiezom�ieZombie
```

Figure 18. Rectangular PAM: Errors in Received Text due to Channel Noise

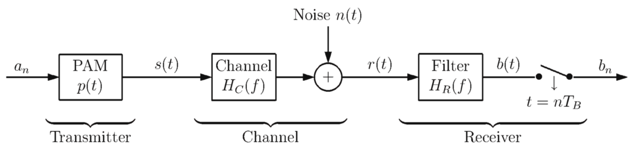


Figure 19. Block Diagram of PAM Communication System Model

sponse of the transmission channel. The receiver filters $r(t)$ and produces the CT signal $b(t) = r(t) * h_R(t)$ which is then sampled at time t_0 to produce the received symbol b_0 which is hopefully a good estimate of a_0. In this model a_0 is a random variable (e.g., taking on values $+1$ or -1) and $n(t)$ is a random process (e.g., with power spectral density $S_n(f) = N_0/2$ for all f for the AWGN channel).

A simple intuitive choice for the receiver would be to use a low-pass filter (LPF) of bandwidth $\approx F_B$. But it turns out that we can do better by explicitly looking for a filter response $h_R(t)$ that maximizes the signal-to-noise ratio (SNR) in b_0 for a given $p(t)$, after sampling $b(t)$ at the receiver. To develop this approach, some basic knowledge of probability theory and random processes is needed. Here we skip the details and state the relevant results. In the absence of noise the expected signal power is

$$\begin{aligned} E[|b_0|^2] &= E[|b(t_0)|^2] \\ &= E[|a_0|^2]\,\Big|\int_{-\infty}^{\infty} H_R(f)\,P_C(f)\,e^{j2\pi f t_0}\,df\Big|^2 \end{aligned}$$

where $H_R(f)$ and $P_C(f)$ are the FTs of $h_R(t)$ and $p_C(t)$, respectively. In the presence of noise only (i.e., $a_0 = 0$), the expected noise power for the AWGN channel is

$$E[b(t_1)\,b^*(t_2)]\Big|_{t_1=t_2} = \sigma_b^2 = \frac{N_0}{2}\int_{-\infty}^{\infty} |H_R(f)|^2\,df$$

where N_0 is the (one-sided) power spectral density of the white Gaussian noise assumed for the AWGN channel. Thus, the SNR for the received symbol b_0 is

$$\frac{S}{N} = \frac{E[|a_0|^2]\,\Big|\int_{-\infty}^{\infty} H_R(f)\,P_C(f)\,e^{j2\pi f t_0}\,df\Big|^2}{N_0/2\,\int_{-\infty}^{\infty} |H_R(f)|^2\,df}$$

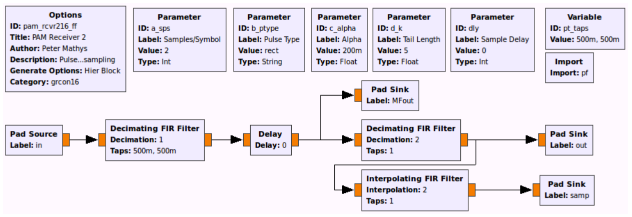
The thought of maximizing this by choice of $h_R(t) \Leftrightarrow H_R(f)$ looks intimidating at first, but it turns out that this can be done relatively easily by invoking the Schwartz Inequality (see, for example, (Haykin & Moher, 2009), p. 282). Skipping the details, the resulting optimal receiver filter is the (normalized) **matched filter** (MF) with unit impulse response

$$h_R(t) = \frac{p_C^*(t_0 - t)}{\int_{-\infty}^{\infty} |p_C(\mu)|^2\,d\mu}$$

where * denotes complex conjugate. The resulting (minimum) SNR is

$$\frac{S}{N} = \frac{2\,E[|a_0|^2]\,\int_{-\infty}^{\infty} |p_C(\mu)|^2\,d\mu}{N_0}$$

A hierarchical GRC block, called PAM Receiver 2, that implements the (normalized) MF is shown in Figure 20.



Options
ID: pam_rcvr216_ff
Title: PAM Receiver 2
Author: Peter Mathys
Description: Pulse...sampling
Generate Options: Hier Block
Category: grcon16

Parameter
ID: a_sps
Label: Samples/Symbol
Value: 2
Type: Int

Parameter
ID: b_ptype
Label: Pulse Type
Value: rect
Type: String

Parameter
ID: c_alpha
Label: Alpha
Value: 200m
Type: Float

Parameter
ID: d_k
Label: Tail Length
Value: 5
Type: Float

Parameter
ID: dly
Label: Sample Delay
Value: 0
Type: Int

Variable
ID: pt_taps
Value: 500m, 500m

Import
Import: pf

Pad Source
Label: in

Decimating FIR Filter
Decimation: 1
Taps: 500m, 500m

Delay
Delay: 0

Pad Sink
Label: MFout

Decimating FIR Filter
Decimation: 2
Taps: 1

Pad Sink
Label: out

Interpolating FIR Filter
Interpolation: 2
Taps: 1

Pad Sink
Label: samp

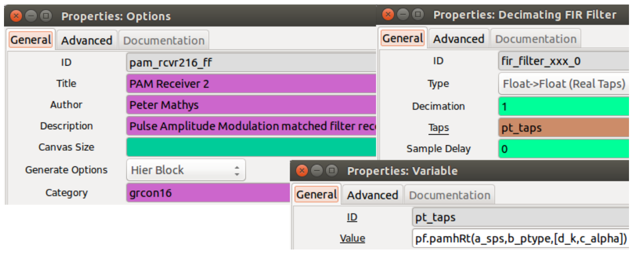
Figure 20. PAM Receiver Hierarchical Block: Matched Filter

This block has one input, three outputs, and 5 parameters. The main processing path goes from the Pad Source `in` to the Decimating (with factor 1) FIR Filter which acts as MF followed by the (adjustable Sample) Delay and another Decimating (with factor `sps`) FIR Filter to the Pad Sink `out`. The other two Pad Sinks are used to observe the waveform $b(t)$ after the MF (`MFout`) and the sample values b_n after after the sampler (`samp`). Note that the latter output is upsampled to the same rate as `MFout` so that both outputs can be displayed on the same QT GUI Time Sink. The most important block settings of the blocks inside PAM Receiver 2 are shown in Figure 21.

Note that the MF taps are obtained from the Python function `pamhRt` in the module `ptfun`. This is the same function as `pampt`, except that $p(t)$ is time reversed and normalized to obtain $h_R(t)$:

```
def pamhRt(sps, ptype, pparms=[]):
  pt = pampt(sps, ptype, pparms)
  hRt = pt[::-1]    # h_R(t) = p(-t)
  hRt = 1.0/sum(np.power(pt,2.0))*hRt
  return hRt
```

Returning to the block diagram in Figure 19 we can see that, assuming an ideal channel with $h_C(t) = \delta(t)$ and



Properties: Options

General | Advanced | Documentation

ID	pam_rcvr216_ff
Title	PAM Receiver 2
Author	Peter Mathys
Description	Pulse Amplitude Modulation matched filter rec
Canvas Size	
Generate Options	Hier Block
Category	grcon16

Properties: Decimating FIR Filter

General | Advanced | Documentation

ID	fir_filter_xxx_0
Type	Float->Float (Real Taps)
Decimation	1
Taps	pt_taps
Sample Delay	0

Properties: Variable

General | Advanced | Documentation

ID	pt_taps
Value	pf.pamhRt(a_sps,b_ptype,[d_k,c_alpha])

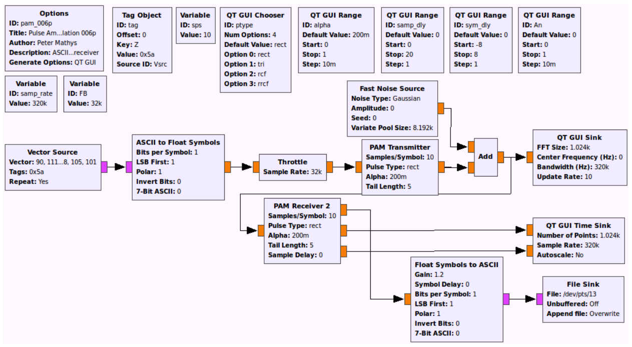
Figure 21. Block Properties of PAM Receiver 2 Blocks

$h_R(t) = p(-t)$ for simplicity, the received signal at $b(t)$ is essentially the convolution $p(t) * p(-t)$, scaled by a_0. If we transmit a whole sequence a_n of symbols then we get intersymbol interference (ISI) after sampling $b(t)$, unless $p(t) * p(-t) = 0$ for $t = nT_B$ for all integers n, except $n = 0$. This works fine for some pulses, e.g., `'rect'` and (ideal) `'sinc'`, but not for others, e.g., `'rcf'` pulses with $\alpha > 0$. To keep the bandwidth requirement below F_B, we need to come up with a pulse $p(t)$ for which $p(t) * p(-t)$ has the shape of the `'rcf'` pulse. The resulting "root raised cosine in frequency" (`'rrcf'`) PAM pulse is

$$p(t) = \frac{T_B}{\pi} \frac{\sin((1-\alpha)\pi t/T_B) + \frac{4\alpha t}{T_B}\cos((1+\alpha)\pi t/T_B)}{(1-(4\alpha t/T_B)^2)\, t}$$

with bandwidth control parameter $\alpha<=0<=1$.

Now we are ready to put PAM signaling with a matched filter receiver to the test on a noisy channel. The complete GRC flowgraph is shown in Figure 22.



Options ID: pam_006p; Title: Pulse Am...lation 006p; Author: Peter Mathys; Description: ASCII...receiver; Generate Options: QT GUI

Tag Object ID: tag; Offset: 0; Key: Z; Value: 0x5a; Source ID: Vsrc

Variable ID: sps; Value: 10

QT GUI Chooser ID: ptype; Num Options: 4; Default Value: rect; Option 0: rect; Option 1: tri; Option 2: rcf; Option 3: rrcf

QT GUI Range ID: alpha; Default Value: 200m; Start: 0; Stop: 1; Step: 10m

QT GUI Range ID: samp_dly; Default Value: 0; Start: 0; Stop: 20; Step: 1

QT GUI Range ID: sym_dly; Default Value: 0; Start: -8; Stop: 8; Step: 1

QT GUI Range ID: An; Default Value: 0; Start: 0; Stop: 1; Step: 10m

Variable ID: samp_rate; Value: 320k

Variable ID: FB; Value: 32k

Fast Noise Source Noise Type: Gaussian; Amplitude: 0; Seed: 0; Variate Pool Size: 8.192k

Vector Source Vector: 90, 111...8, 105, 101; Tags: 0x5a; Repeat: Yes

ASCII to Float Symbols Bits per Symbol: 1; LSB First: 1; Polar: 1; Invert Bits: 0; 7-Bit ASCII: 0

Throttle Sample Rate: 32k

PAM Transmitter Samples/Symbol: 10; Pulse Type: rect; Alpha: 200m; Tail Length: 5

Add

QT GUI Sink FFT Size: 1.024k; Center Frequency (Hz): 0; Bandwidth (Hz): 320k; Update Rate: 10

PAM Receiver 2 Samples/Symbol: 10; Pulse Type: rect; Alpha: 200m; Tail Length: 5; Sample Delay: 0

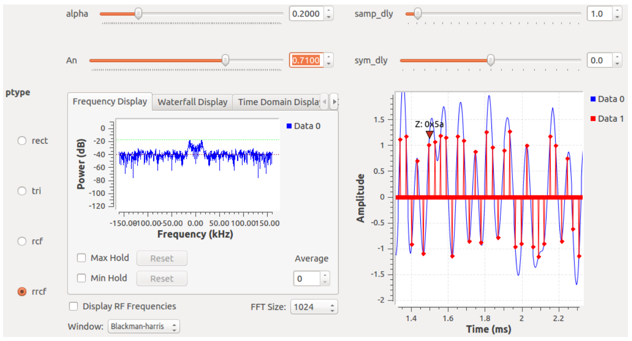
QT GUI Time Sink Number of Points: 1.024k; Sample Rate: 320k; Autoscale: No

Float Symbols to ASCII Gain: 1.2; Symbol Delay: 0; Bits per Symbol: 1; LSB First: 1; Polar: 1; Invert Bits: 0; 7-Bit ASCII: 0

File Sink File: /dev/pts/13; Unbuffered: Off; Append file: Overwrite

Figure 22. PAM Transmitter, Channel Noise, and Matched Filter Receiver

Using the `'rrcf'` instead of the `'rect'` PAM pulse for variation, but the same noise amplitude as in Act 3, we now obtain the results shown in Figure 23. On the left is the spectrum of the transmitted PAM signal and on the right is the received signal after the MF (blue) and after sampling (red).



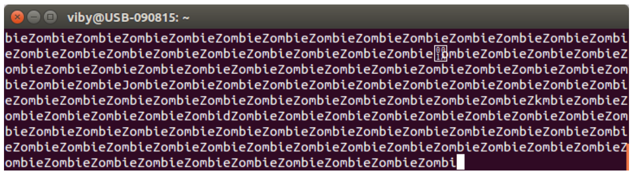
alpha 0.2000; samp_dly 1.0; An 0.7100; sym_dly 0.0

ptype: rect, tri, rcf, [✓ rrcf]

Max Hold; Min Hold; Average 0; Display RF Frequencies; FFT Size: 1024; Window: Blackman-harris

Figure 23. Noisy PAM Spectrum and Waveform after MF for 'rrcf' Pulse Type

As shown in Figure 24, there are much fewer errors in the received ASCII text when a MF receiver is used.



viby@USB-090815: ~

```
bieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombi
eZombieZombieZombieZombieZombieZombieZombieZombieZombie[illegible]ombieZombieZombieZombieZ
ombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZom
bieZombieZombieJombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombi
eZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZkmbieZombieZ
ombieZombieZombieZombieZombidZombieZombieZombieZombieZombieZombieZombieZombieZom
bieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombi
eZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZombieZ
ombieZombieZombieZombieZombieZombieZombieZombieZombieZombi
```

Figure 24. 'rrcf' PAM with MF: Errors in Received Text due to Channel Noise

One thing that is interesting to try with the flowgraph in Figure 22 is to use PAM with different pulse shapes, e.g., `'rect'` or `'tri'`, and to compare the amount of errors in the received text. Some pulse shapes (ask yourself which) will create a sizeable amount of errors due to ISI after the MF.

3. Act x, for $x > 4$

In act 5 it is time to make the transition from baseband to bandpass communication systems by using amplitude modulation (AM) to put the PAM signal $s(t)$ onto a carrier with frequency f_c. Now wireless communication with software-defined radios (SDR) becomes possible. Issues to be treated in act 6 are the use of complex baseband signals and signal constellations which translate to quadrature amplitude modulation (QAM) for bandpass signals. Wireless communication between physically separated transmitters and receivers which have a slightly different perception of time leads to treating synchronization issues in act 7. Mobility of transmitters and/or receivers creates multipath propagation scenarios which necessitate discussion of more sophisticated channel models, channel equalization, and different modulation techniques (e.g., orthogonal frequency-division multiplexing or OFDM) in act 8 and beyond.

References

Haykin, Simon and Moher, Michael. *Communication Systems*. John Wiley and Sons, 5th edition, 2009.

Lathi, B. P. and Ding, Zhi. *Modern Digital and Analog Communication Systems*. Oxford University Press, 4th edition, 2009.

Proakis, John G. and Salehi, Masoud. *Communication Systems Engineering*. Prentice Hall, 2nd edition, 2002.