Guided Tutorial PSK Demodulation

Contents [hide]

1 Objectives
2 Prerequisites
3 Introduction
4 Transmitting a Signal
4.1 Constellation Modulator
5 Adding a Channel
6 Recovering Timing
6.1 Details of the PFB Clock Recovery Block
6.2 Using the PFB Clock Recovery Block in Our Receiver
7 Multipath
8 Equalizers
9 LMS-DD Equalizer
9.1 More Equalizers
10 Phase and Fine Frequency Correction
11 Decoding
12 Quiz

Objectives

- Understand issues of signal distortion and channel effects.
- Recognize the stages required to recover signals.
    - Timing recovery
    - Multipath channels
    - Phase and frequency correction
- Decoding symbols and bit ordering

Prerequisites

- Basic knowledge of radio and RF
    - Our Suggested Reading list
    - The ARRL Handbook
- Basic signal process / DSP concepts
- Previous Tutorials recommended
    - Tutorials Introductions
    - Using GNU Radio with Hardware

Introduction

In this tutorial, we will be building off of the previous tutorials to see how we deal with issues of receiving digitally modulated signals. Though we're focused on simulation and not over-the-air, we want to understand many of the issues involved with what goes on when transmitting and then receiving real signals through real hardware and channel effects. Here, we'll walk through the stages of setting up our simulation and then step-by-step walk through how to recover the signal.

As we walk through the recovery stages, we should keep in mind that this is just *one way* of handling digital signal reception. There are various algorithms and methods that have been designed for these steps, and different types of digital signals will behave differently. Here, we go through a set of stages and use of algorithms readily available in GNU Radio for PSK signal reception and demodulation. This tutorial, however, should in no way be meant to suggest that this is the only way to accomplish this task.

Transmitting a Signal

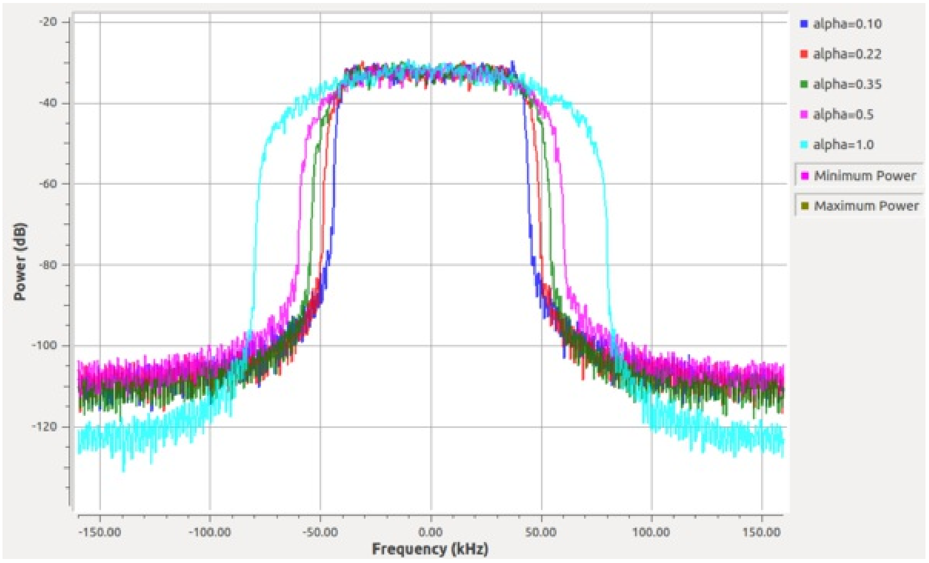
The first stage is transmitting the signal. We take a stream of bits and modulate it onto a complex constellation. To do this, we use the Constellation Modulator block, which takes in a constellation object and other settings that establish how we want to control the transmitted signal. First, let's look at the constellation object.

Constellation Modulator

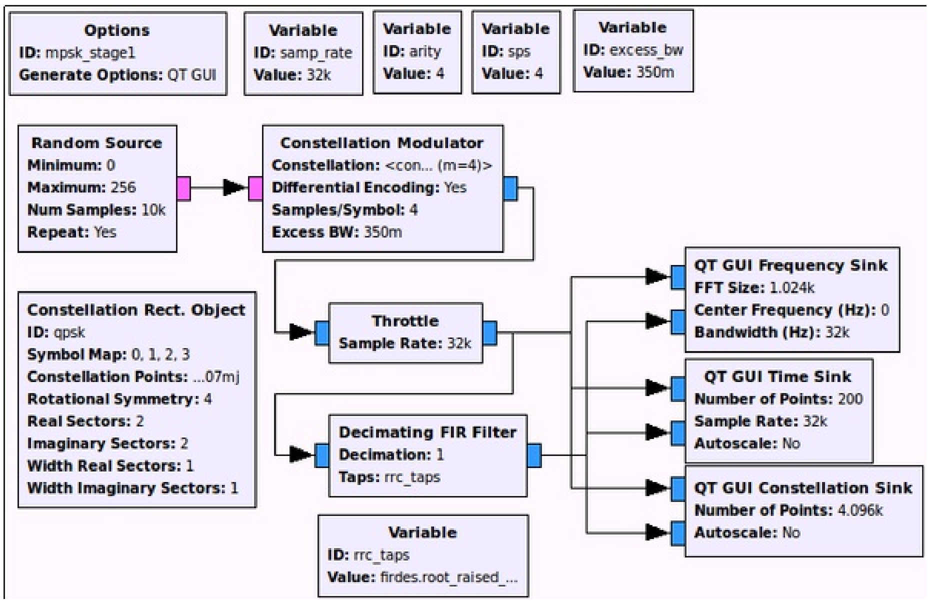
Tutorial 4 already introduced us to the modulation side of our QPSK modem. We have changed this tutorial to use the Constellation Modulator block with a Constellation Object because of the greater flexibility in defining the modulation. We can set the constellation points as well as how the symbols are mapped to those points. The constellation object allows us to determine how the symbols are coded and if we want to use Gray coding or not. The modulator block can then use this modulation scheme with or without differential encoding. The constellation modulator expects packed bytes, so we have a random source generator providing bytes with values 0 - 255. The manual page on digital modulation explains more of these concepts.

When dealing with the number of samples per symbol, we have two criteria. First, we want to keep this value as small as possible with a minimum value of 2. Generally, we can use this value to help us match the desired bit rate with the sample rate of the hardware device we'll be using. Since we're starting in simulation mode, the samples per symbol is only important in making sure we match this rate at all important points. We'll use 4 here, which is greater than what we need (e.g., 2) but useful to visualize the signal in the different domains.

Finally, we set the excess bandwidth value. The constellation modulator uses a root raised cosine (RRC) pulse shaping filter, which gives us a single parameter to adjust the roll-off factor of the filter, often known mathematically as 'alpha.' The following figure is generated from the **mpsk_rrc_rolloff.grc** example in "gr-tutorial:https://github.com/gnuradio/gr-tutorial and shows different values of the excess bandwidth. Typical values these days are between 0.2 and 0.35.



The example file **mpsk_stage1.grc** transmits a pulse shape-filtered, up-sampled QPSK constellation. This flowgraph plots both the transmitted signal as well as part of the receiver chain in time, frequency, and the constellation plot.



Options
ID: mpsk_stage1
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 32k

Variable
ID: arity
Value: 4

Variable
ID: sps
Value: 4

Variable
ID: excess_bw
Value: 350m

Random Source
Minimum: 0
Maximum: 256
Num Samples: 10k
Repeat: Yes

Constellation Modulator
Constellation: <con... (m=4)>
Differential Encoding: Yes
Samples/Symbol: 4
Excess BW: 350m

Constellation Rect. Object
ID: qpsk
Symbol Map: 0, 1, 2, 3
Constellation Points: ...07mj
Rotational Symmetry: 4
Real Sectors: 2
Imaginary Sectors: 2
Width Real Sectors: 1
Width Imaginary Sectors: 1

Throttle
Sample Rate: 32k

Decimating FIR Filter
Decimation: 1
Taps: rrc_taps

Variable
ID: rrc_taps
Value: firdes.root_raised_...

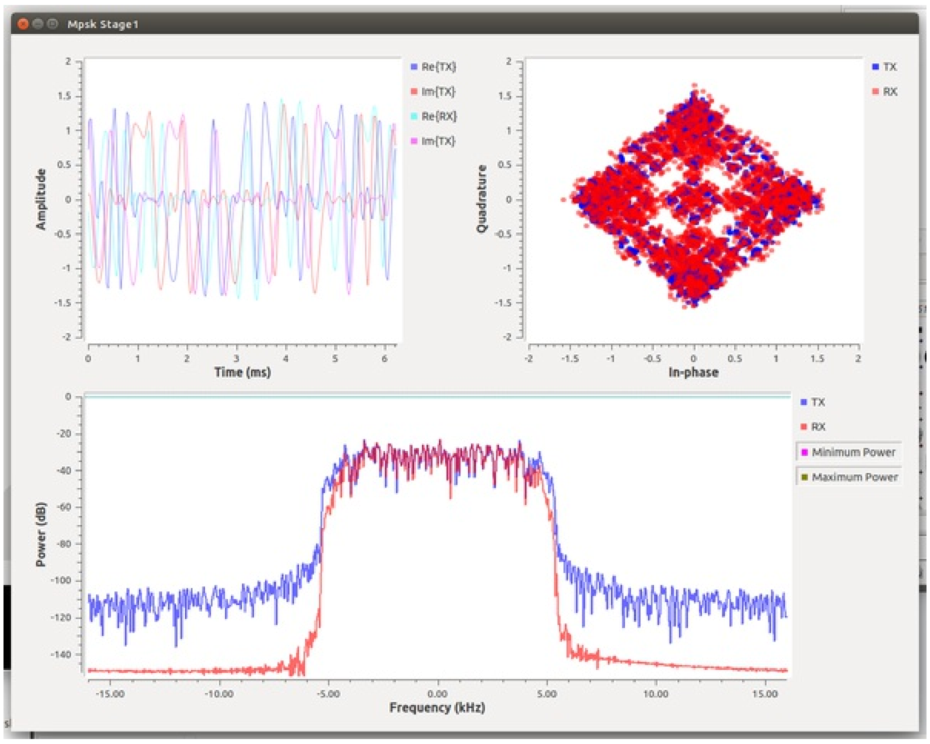
QT GUI Frequency Sink
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 32k

QT GUI Time Sink
Number of Points: 200
Sample Rate: 32k
Autoscale: No

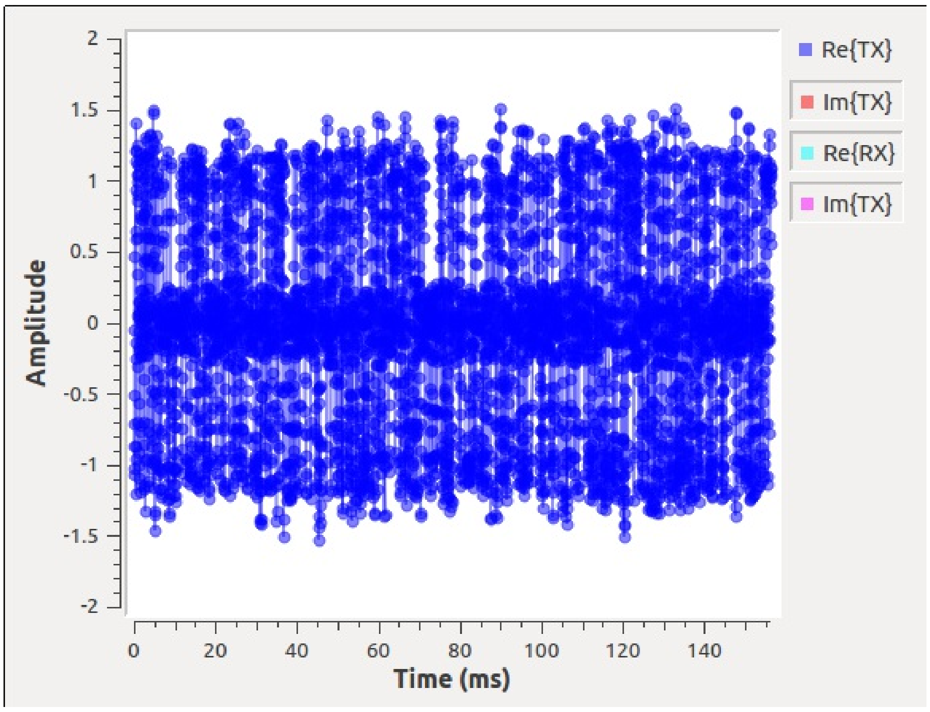
QT GUI Constellation Sink
Number of Points: 4.096k
Autoscale: No

When running, first, let's turn off all of the RX signals by just clicking on them in the legends.

In the constellation plot, we see the effects of the up-sampling and filtering process. In this case, the RRC filter adds intentional self-interference, known as inter-symbol interference (ISI). ISI is bad for a received signal because it blurs the symbols together, and we'll look into this in-depth during the timing recovery section. Right now, let's just see what we're doing to the signal. If you are just looking at the transmitted signals from this graph, then you should see a number of things happening. First and foremost, the frequency plot is showing a signal that has a nice shape to it and that rolls-off into the noise. If we didn't put a shaping filter on the signal, we would be transmitting square waves that produce a lot of energy in the adjacent channels. On your own, try creating a simple flowgraph of a signal source set to generate a square wave into a frequency sink and you'll see what we mean. By reducing the out-of-band emissions, our signal now stays nicely within our channel's bandwidth. But there is a slight problem with this signal that can be best seen in the time domain plot.

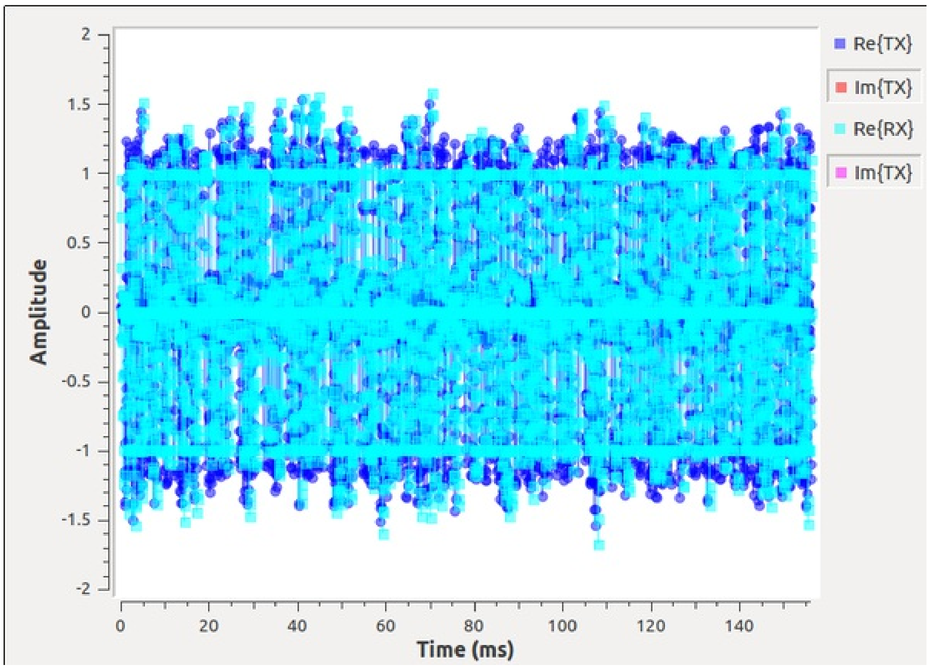


Let's turn off all of the signals in the time domain plot except the real part of the transmitted signal (i.e., "Re{TX}"). Next, by clicking the middle mouse button to open a menu, increase the number of point from 200 to 5000. Using the same menu, under the line label Re{TX}, go to "Line Marker" and click circle. What you should see is a bit of a mess of points. How are we supposed to receive these samples when we thought we were transmitting 1's and 0's? Use the drop-down button to show the transmitted signal.



What we are seeing in this image is actually the ISI that we mentioned before. We get rid of ISI by using another filter at the receiver. Basically, what we've done is purposefully used a filter on the transmitter, the RRC filter, that creates the ISI. But when we convolve two RRC filters together, we get a raised cosine filter, which is a form of a Nyquist filter. So, knowing this property of the RRC filter, we can use another RRC filter at the receiver. Filtering is just a convolution here, so the output of the receive-side RRC filter is a raised cosine pulse shaped signal with minimized ISI. The other benefit is that absent effects of the channel, what we are also doing is using a matched filter at the receiver.

So what does the signal received through the matched RRC filter look like? In the time plot, we can turn back on the "Re{RX}" signal and using the drop-down menu controls add a symbol to this output. As opposed to the mess of the transmitted signal, this received version has three distinct lines at +1, -1, and 0 (the zeros might be difficult to see, so you'll have to look closely). Use the drop-down button to show the cleaned-up signal at the receiver. For those keeping score, this matched filter satisfies the Nyquist ISI criterion. We'll see this again in a different format during the timing recovery stage.



Adding a Channel

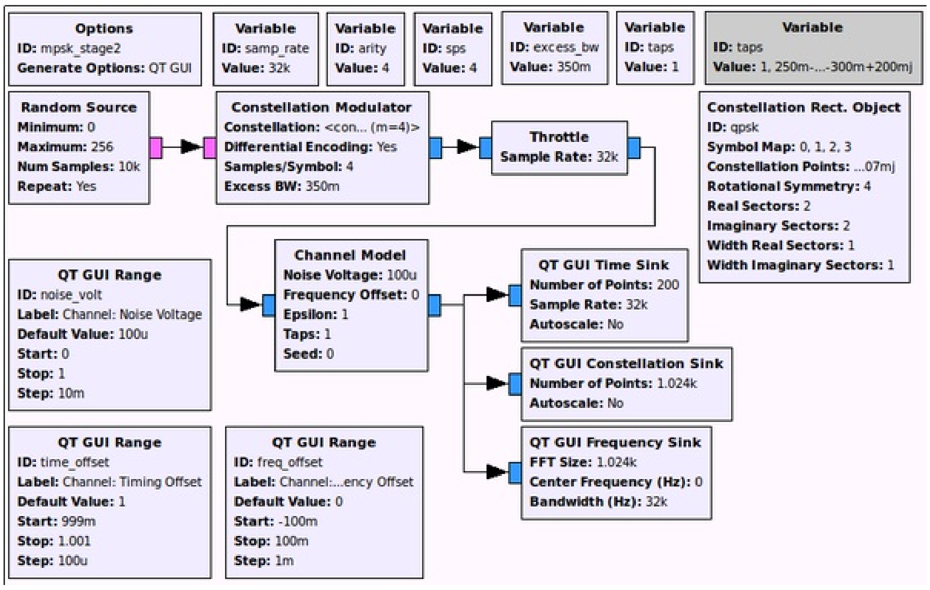
That first stage example only dealt with the particulars of the mechanics of transmitting a QPSK signal. We'll now start to look into the effects of the channel and how the signal is distorted between when it was transmitted and when we see the signal in the receiver. The first step is to add a channel model, which is done using the example **mpsk_stage2.grc**. To start with, we'll use the most basic Channel Model block of GNU Radio.

This block allows us to simulate a few main issues that we have to deal with. The first, fundamental issue with receivers is noise. Thermal noise in our receivers causes noise that we know of as additive white Gaussian noise AWGN. We set the noise power by adjusting the noise voltage value of the channel model. We specify the voltage here instead of power because we need to know the bandwidth of the signal in order to the power properly. One of the defining aspects of GNU Radio is the independence of the blocks, so the channel model doesn't know anything about the incoming signal itself. Instead, we use the voltage concept because of its independence to the signal itself. We can calculate the noise voltage from a desired power level ourselves knowing the other parameters of simulation.

Another significant problem between two radios is different clocks, which drive the frequency of the radios. The clocks are, for one thing, imperfect, and therefore different between radios. One radio transmits nominally at fc (say, 450 MHz), but the imperfections mean that it is really transmitting at fc + f_delta_1. Meanwhile, the other radio has a different clock and therefore a different offset, f_delta_2. When it's set to fc, the real frequency is at fc + f_delta_2. In the end, the received signal will be f_delta_1 + f_delta_2 off where we think it should be (these deltas may be positive or negative).

Related to the clock problem is the ideal sampling point. We've up-sampled our signal in the transmitter and shaped it, but when receiving it, we need to sample the signal at the original sampling point in order to maximize the signal power and minimize and inter-symbol interference. Like in our stage 1 simulation after adding the second RRC filter, we can see that among the 4 samples per symbol, one of them is at the ideal sampling point of +1, -1, or 0. But again, the two radios are running at two different speeds, and so the ideal sampling point is an unknown.

The second stage of our simulation allows us to play with these concepts of additive noise, frequency offset, and timing offset. When we first run this graph, shown below, we have turned all of these effects off but have sliders to adjust the settings. We can add a bit of noise (0.2), some frequency offset 0.025), and some timing offset (1.0005) to see the resulting signal from where we started.


Options
ID: mpsk_stage2
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 32k

Variable
ID: arity
Value: 4

Variable
ID: sps
Value: 4

Variable
ID: excess_bw
Value: 350m

Variable
ID: taps
Value: 1

Variable
ID: taps
Value: 1, 250m-...-300m+200mj

Random Source
Minimum: 0
Maximum: 256
Num Samples: 10k
Repeat: Yes

Constellation Modulator
Constellation: <con... (m=4)>
Differential Encoding: Yes
Samples/Symbol: 4
Excess BW: 350m

Throttle
Sample Rate: 32k

Constellation Rect. Object
ID: qpsk
Symbol Map: 0, 1, 2, 3
Constellation Points: ...07mj
Rotational Symmetry: 4
Real Sectors: 2
Imaginary Sectors: 2
Width Real Sectors: 1
Width Imaginary Sectors: 1

QT GUI Range
ID: noise_volt
Label: Channel: Noise Voltage
Default Value: 100u
Start: 0
Stop: 1
Step: 10m

Channel Model
Noise Voltage: 100u
Frequency Offset: 0
Epsilon: 1
Taps: 1
Seed: 0

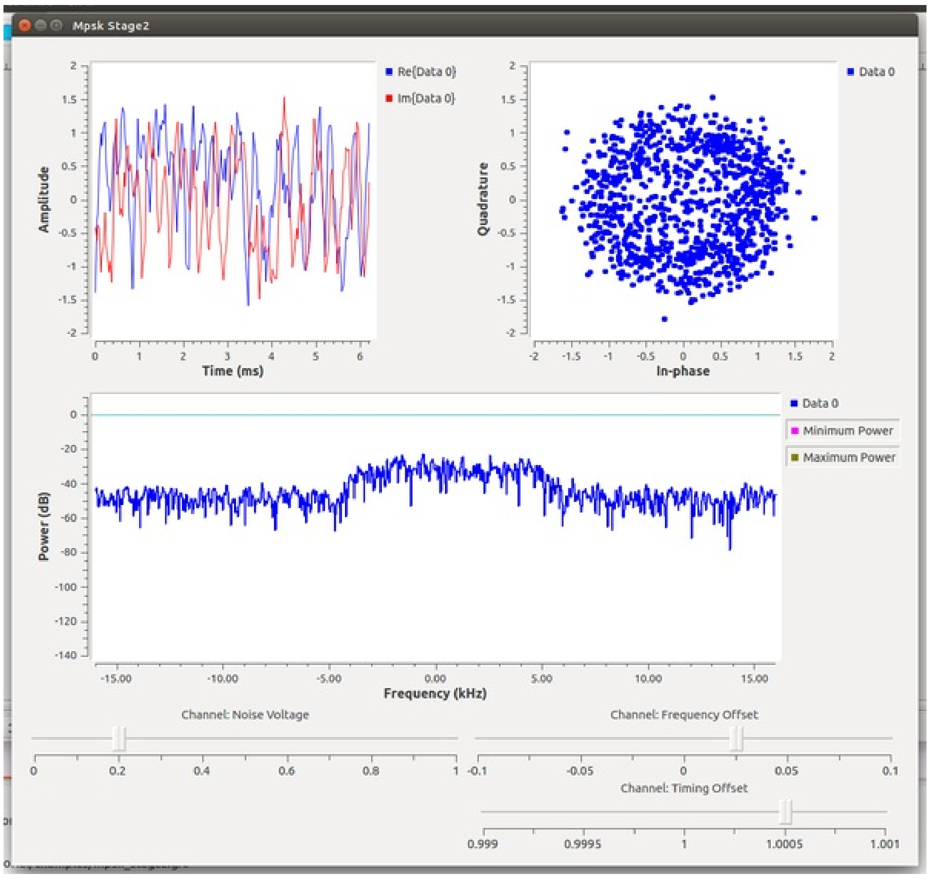
QT GUI Time Sink
Number of Points: 200
Sample Rate: 32k
Autoscale: No

QT GUI Constellation Sink
Number of Points: 1.024k
Autoscale: No

QT GUI Frequency Sink
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 32k

QT GUI Range
ID: time_offset
Label: Channel: Timing Offset
Default Value: 1
Start: 999m
Stop: 1.001
Step: 100u

QT GUI Range
ID: freq_offset
Label: Channel:...ency Offset
Default Value: 0
Start: -100m
Stop: 100m
Step: 1m



The constellation plot shows us a cloud of samples, far worse that what we started off with in the last stage. From this received signal, we now have to undo all of these effects.

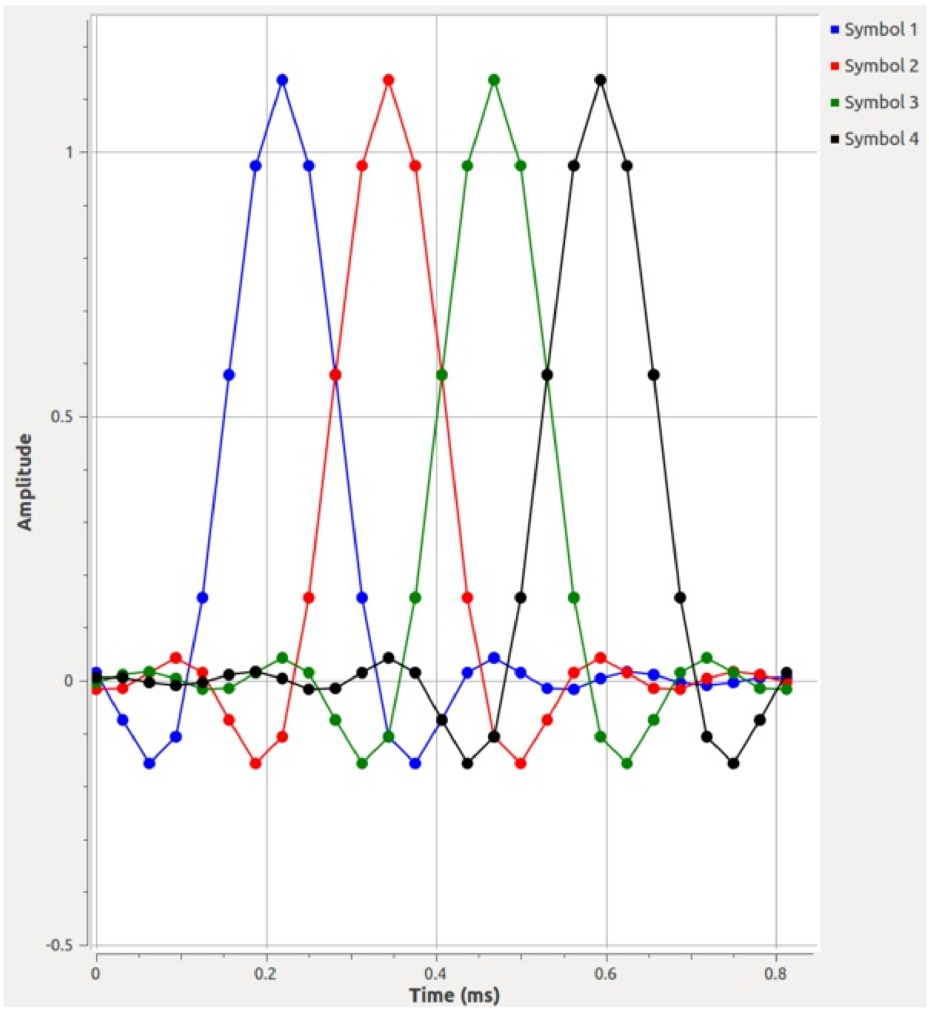
Recovering Timing

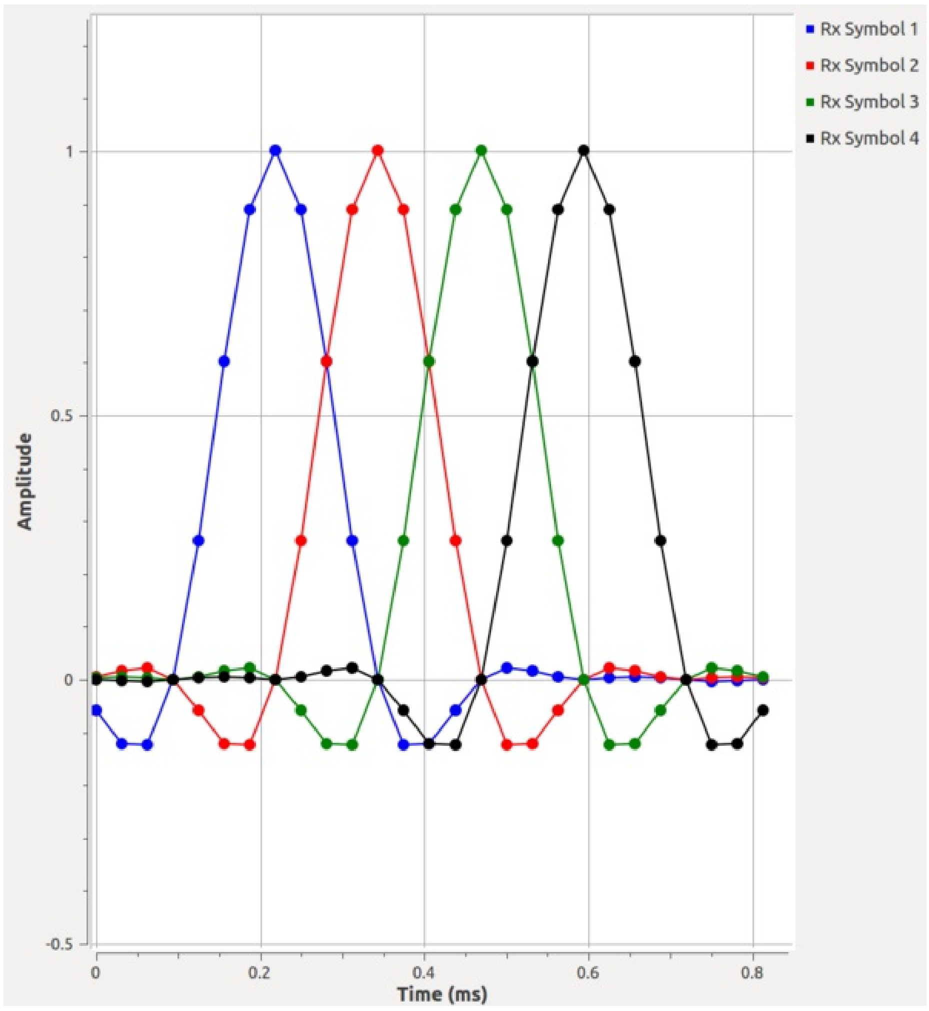
We'll now walk step-by-step through the recovery process. Keep in mind, though, that there are many algorithms we could potentially use for recovery of each stage. Some, even, can do joint recovery of multiple stages at the same time. We will specifically use the polyphase clock recovery algorithm here.

In our stages here, we start off with timing recovery. We're trying to find the best time to sample the incoming signals, which will maximize the SNR of each sample as well as reduce the effects of inter symbol interference (ISI).

We can illustrate the ISI problem using the example flowgraph **symbol_sampling.grc** where we simply create four separate symbols of 1's in row then filter them. The first stage of filtering performs up-sampling to the 'sps' samples per symbol and uses a root raised cosine filter. We follow this with another root raised cosine filter that does no rate changes. The second RRC filter here converts the signals from using the non-Nyquist RRC filter to a Nyquist raised cosine (RC) filter as we discussed in the first stage of this tutorial. The output, shown in the figures below, shows the differences between the RRC- and RC-filtered symbols. Without Nyquist filtering, we can see how at the ideal sampling point of each symbol, the other symbols have some energy. If we summed these symbols together like we would in a continuous stream of samples, the energy of those other samples add together and distort the symbol at that point.

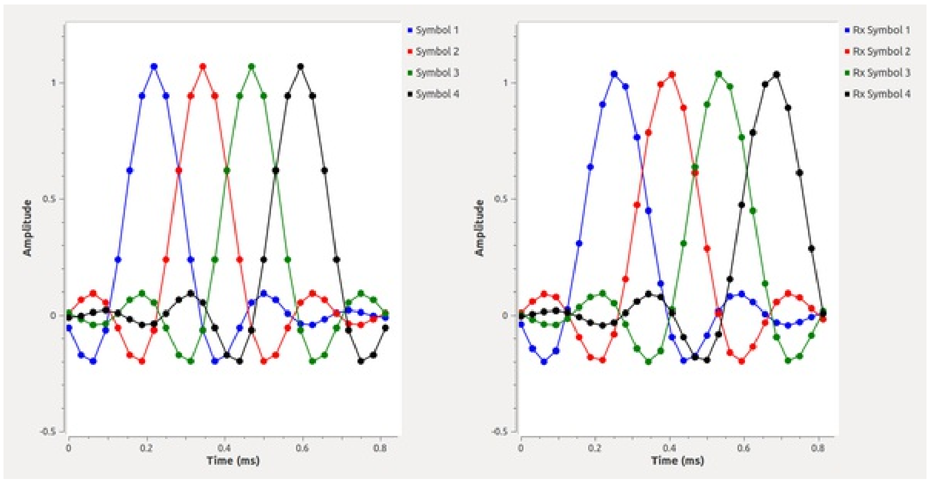
Conversely, in the RC filtered output, the energy from the other samples are at 0 at the ideal sampling point for the given symbol in time. That means that if we sample at exactly the correct sample point, we only get energy from the current symbol with no interference from the other symbols in the stream. Again, what we're seeing is how the timing recovery applies a matched filter to satisfy the Nyquist ISI criterion.





This simulation allows us to easily adjust things like the number of samples per symbol, excess bandwidth of the RRC filters, and the number of taps. We can then play with these different values to see how they affect the behavior of the sampling point.

Next, let's look at what happens due to the different clocks affecting the sampling points between the transmitter and receiver. Using the example flowgraph in **symbol_sampling_diff.grc**, we simulate the effect of the different clocks in the transmitter and receiver. Each clock is imperfect and so a) will start at a different point in time and b) drift relative to the other clocks. We simulate this by adding a resampler that adjusts the symbol sampling time slightly between the transmitted signal (in the transmit image above) and the receiver, shown below. The clock difference shown here of 1.125 is extreme as a way of showing it in this setup as a visualization technique. In reality, timing differences are on the order or parts per million. But here, notice that with the samples being collected at different points in time, the ideal sampling period is not known and any sampling done will also include ISI.

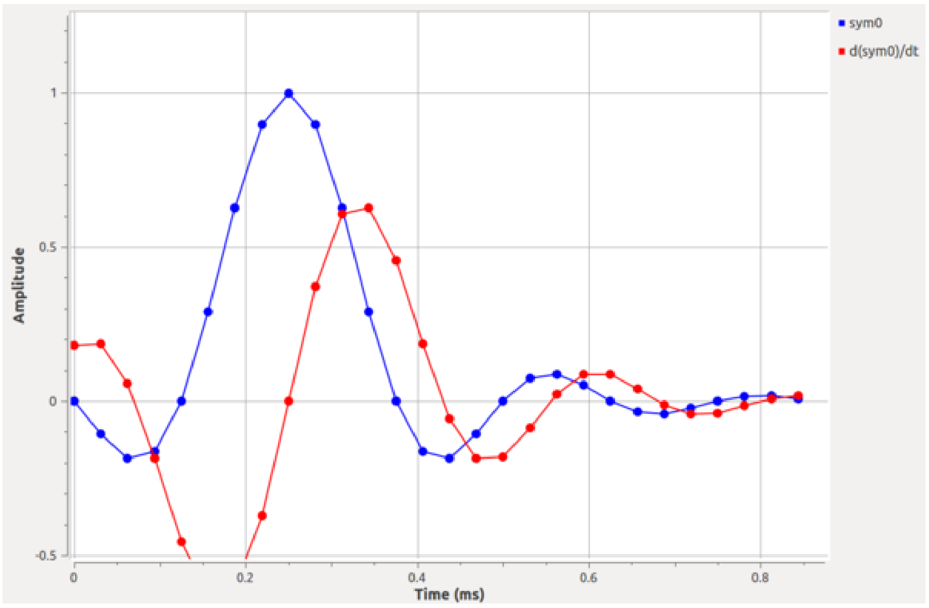


Our task here is to synchronize the transmit and receiver clocks using only information at the receiver from the incoming samples. This job is known as clock or timing recovery.

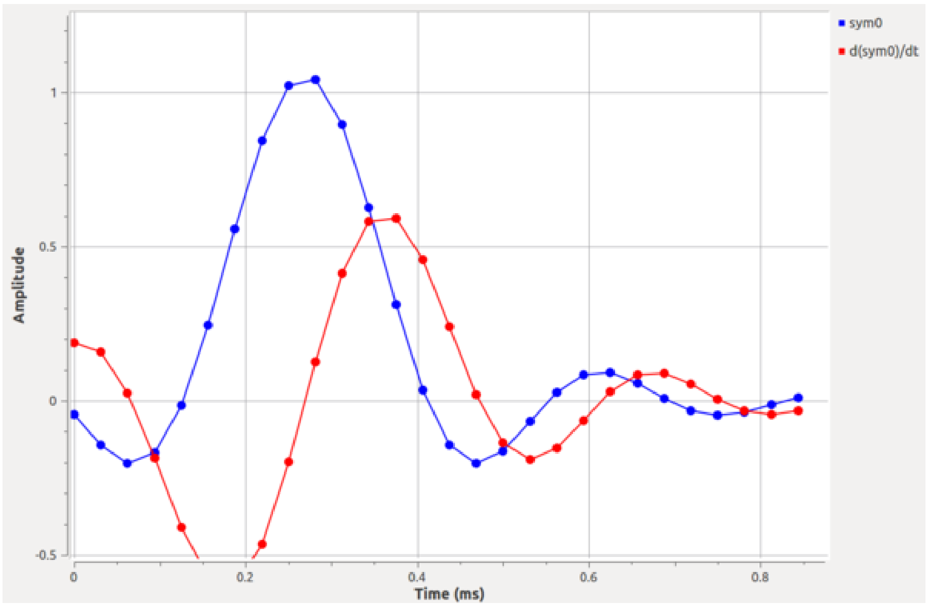
Details of the PFB Clock Recovery Block

There are various algorithms that we can use to recovery the clock at the receiver, and almost all of them involve some kind of feedback control loop. Those that don't are generally data aided using a known word like a preamble. We'll use a polyphase filterbank clock recovery technique that can be found in *Multirate Signal Processing for Communications Systems* by fred harris. This block does three things for us. First, it performs the clock recovery. Second, it does the receiver matched filter to remove the ISI problem. Third, it down-samples the signal and produces samples at 1 sps.

The block works by calculating the first differential of the incoming signal, which will be related to its clock offset. If we simulate this very simply at first, we can see how the differential filter will work for us. First, using the example flowgraph **symbol_differential_filter.grc**, we can see how everything looks perfect when our rate parameter is 1 (i.e., there is no clock offset). The sample we want is obviously at 0.25 ms. The difference filter ([-1, 0, 1]) generates the differential of the symbol, and as the following figure shows, the output of this filter at the correct sampling point is 0. We can then invert that statement and instead say when the output of the differential filter is 0 we have found the optimal sampling point.

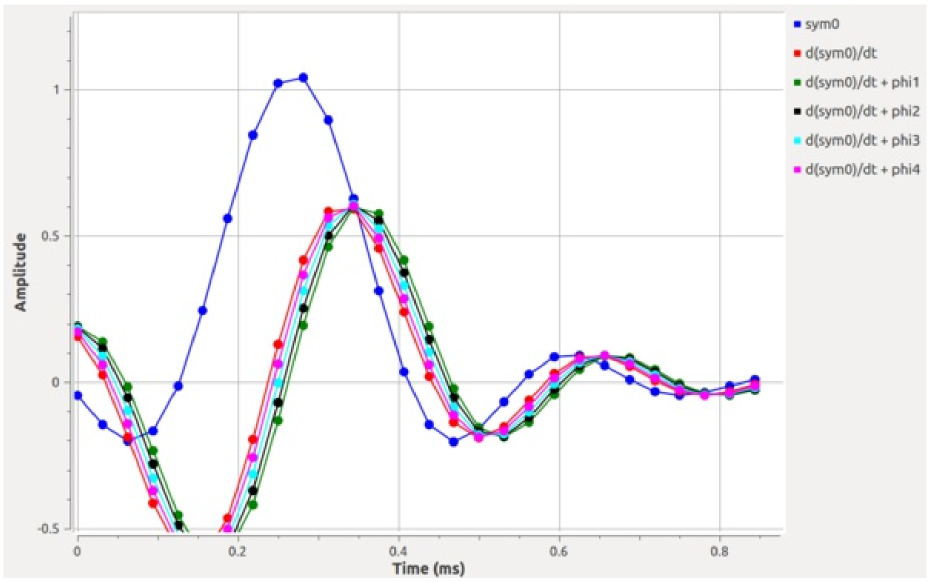


What happens when we have a timing offset? That output is shown below shows that the timing offset where the peak of the symbol is off and the derivative filter does not show us a point at zero.



Instead of using a single filter, what we can do is build up a series of filters, each with a different phase. If we have enough filters at different phases, one of them is the correct filter phase that will give us the timing value we desire. Let's look at a simulation that builds 5 filters, which means 5 different phases. Think of each filter as segmenting the unit circle (0 to 2pi) into 5 equal slices. Using the example flowgraph **symbol_differential_filter_phase.grc**, we can see how this helps us. Notice here that we are using the fractional_resampler_ff here because it makes it easy to do the phase shift (between 0 and 1), but it also changes the filter delays of the signals, so we correct for that using the follow-on delay blocks.

The figure below now gives us an idea of what we're dealing with, although it's a bit inexact. What we can see is that the signal labeled as d(sym0)/dt + phi3 has a sample point at exactly 0. This tells us that our ideal sampling point occurs at this phase offset. Therefore, if we take a the RRC filter of our receiver and adjust its phase by phi3 (which is 3*2pi/5), then we can correct for the timing mismatch and select the ideal sampling point at this sample time.



But as we have discussed, this is only a simulated approximation; in reality, the samples of each filter wouldn't occur at the same point in time. We have to up-sample by the number of filter (e.g., 5) to really see this behavior. However, that can clue us into what's happening a bit farther. We can look at these different filters as parts of one big filter that is over-sampled by M, where M=5 in our simple example here. We could up-sample our incoming signal by this much and select the point in time where we get the 0 output of the difference filter. The trouble with that is we are talking about a large amount of added computational complexity, since that is proportional to our sample rate. Instead, we're working on filters of different phases at the incoming sample rate, but with the bank of them at these different phases, we can get the effect of working with the over-sampled filter without the added computational cost.

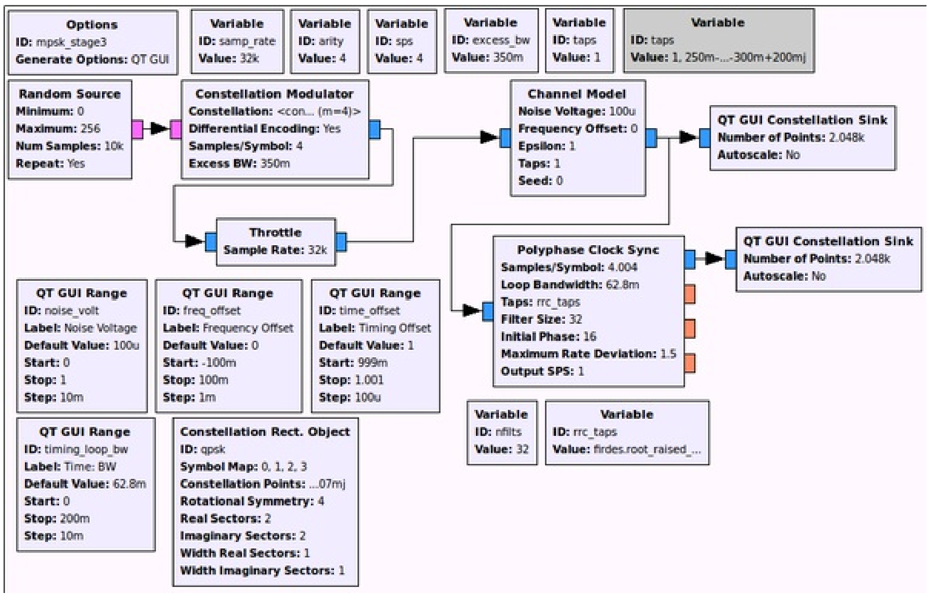
Ok, so in our example above, we offset our sampling rate by some known factor of 1.2 and found that we could use one of five filters as the ideal sampling point. Unfortunately, we really only have 5 different phases we can exactly produce and correct for here. Any sampling offset between these phases will still produce a mistimed sample with added ISI as we explored previously. So instead, we use way more than 5 filters in our clock recovery algorithm. Without exploring the math (see harris' book referenced above), we can use 32 filters to give us a maximum ISI noise factor that is less than the quantization noise of a 16 bit value. If we want more than 16 bits of precision, we can use more filters.

Alright, so what? We have a large bank of filters where one of them is at (or very close to) the ideal sampling phase offset. How do we automatically find that? Well, we use a 2nd order control loop, like we almost always do in these recovery situations. The error signal for the recovery is the output of the differential filter. The control loop starts at one of the filters and calculates the output as the error signal. It then moves its way up or down the bank of filters proportionally to the error signal, and so we're trying to find where that error signal is closest to 0. This is our optimal filter for the sampling point. And because we expect the transmit and receive clocks to drift relative to each other, we use a second order control loop to acquire both the correct filter phase as well as the rate difference between the two clocks.

GNU Radio comes with an example found in the digital examples directory called source:gr-digital/examples/example_timing.py. You can run this script on your own to see the convergence behavior of the source:gr-digital/lib/pfb_clock_sync_ccf_impl.cc recovery block.

Using the PFB Clock Recovery Block in Our Receiver

Now let's put this block to use in our simulation. The example flowgraph **mpsk_stage3.grc** script takes the output of the channel model and passes it through our Polyphase Clock Sync block. This block is setup with 32 filters, for the reasons we discussed above, and a loop bandwidth of 2pi/100. The block also takes in a value for the expected samples per symbol, but this is just our guess at what we think this value should be. Internally, the block will adapt around this value based on the rates of the incoming signal. Notice, however, that I have set this simulation up where the estimate is slightly off of the 4 sps we transmit with. This is to simulate an initial timing offset between the transmitter and receiver since we initialize our Timing Offset control to 1.0. It makes things slightly harder so that we can observe the convergence of the constellation.



Options
ID: mpsk_stage3
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 32k

Variable
ID: arity
Value: 4

Variable
ID: sps
Value: 4

Variable
ID: excess_bw
Value: 350m

Variable
ID: taps
Value: 1

Variable
ID: taps
Value: 1, 250m-...-300m+200mj

Random Source
Minimum: 0
Maximum: 256
Num Samples: 10k
Repeat: Yes

Constellation Modulator
Constellation: <con... (m=4)>
Differential Encoding: Yes
Samples/Symbol: 4
Excess BW: 350m

Channel Model
Noise Voltage: 100u
Frequency Offset: 0
Epsilon: 1
Taps: 1
Seed: 0

QT GUI Constellation Sink
Number of Points: 2.048k
Autoscale: No

Throttle
Sample Rate: 32k

Polyphase Clock Sync
Samples/Symbol: 4.004
Loop Bandwidth: 62.8m
Taps: rrc_taps
Filter Size: 32
Initial Phase: 16
Maximum Rate Deviation: 1.5
Output SPS: 1

QT GUI Constellation Sink
Number of Points: 2.048k
Autoscale: No

QT GUI Range
ID: noise_volt
Label: Noise Voltage
Default Value: 100u
Start: 0
Stop: 1
Step: 10m

QT GUI Range
ID: freq_offset
Label: Frequency Offset
Default Value: 0
Start: -100m
Stop: 100m
Step: 1m

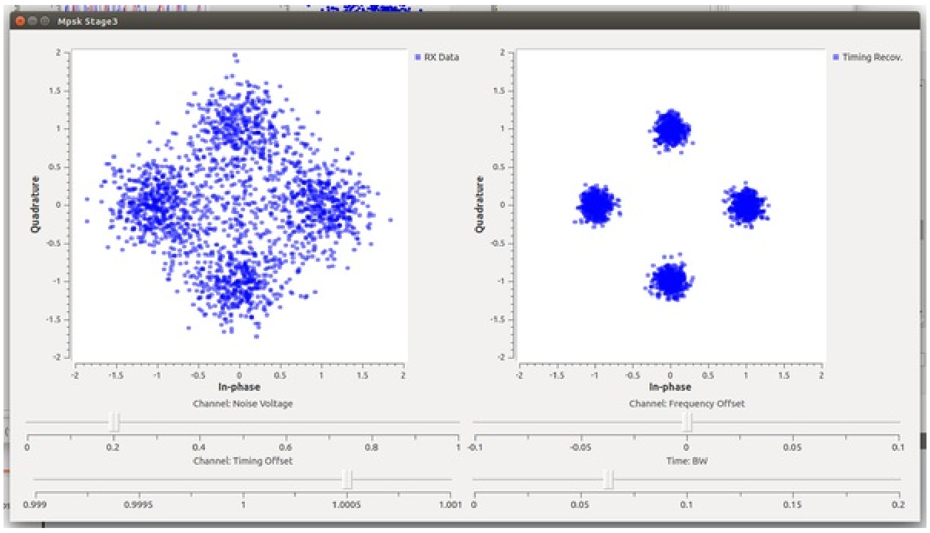
QT GUI Range
ID: time_offset
Label: Timing Offset
Default Value: 1
Start: 999m
Stop: 1.001
Step: 100u

Variable
ID: nfilts
Value: 32

Variable
ID: rrc_taps
Value: firdes.root_raised_...

QT GUI Range
ID: timing_loop_bw
Label: Time: BW
Default Value: 62.8m
Start: 0
Stop: 200m
Step: 10m

Constellation Rect. Object
ID: qpsk
Symbol Map: 0, 1, 2, 3
Constellation Points: ...07mj
Rotational Symmetry: 4
Real Sectors: 2
Imaginary Sectors: 2
Width Real Sectors: 1
Width Imaginary Sectors: 1



When running this script, we see the constellation on the left as the received signal before timing recovery and on the right after timing recovery. It's still a little noisy as a result of the ISI after the 32 filters, which is quickly absorbed by noise once we adjust the channels Noise Voltage setting to be more than 0.

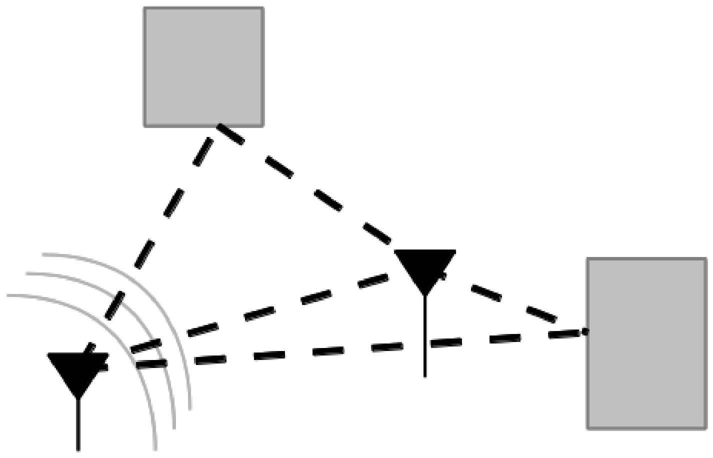
We can then play around with changing the timing and frequency offset. Moving the timing bar around shows us how the clock sync block keeps the signal locked in time and outputs samples at (or very near) the ideal constellation points. When we add frequency offset, we can see that the constellation becomes a circle. The constellation is still on the unit circle, so we know that it's still keeping the correct timing, but the block isn't allowing us to correct for a frequency offset. We still need to handle this, but later.

Likewise, we can change the multipath simulation environment by changing which version of the taps variable we use. Adding multipath will show us that the clock recovery block is robust to multipath but won't correct for it, so again, we need something else to handle that.

Multipath

Let's first understand what multipath is. There is already quite a lot written on the subject of multipath, we'll just explore it enough here to get a general sense of where it comes from and how it affects our communications capabilities. We won't be going into details about real fading channels or how to analyze their properties.

Multipath results from that fact that in most communication environments, we don't have a single path for the signal to travel from the transmitter to the receiver. Like the cartoon below shows, any time there is an object that is reflective to the signal, a new path can be established between the two nodes. Surfaces like buildings, signs, trees, people, cats, etc. can all produce signal reflections. Each of these reflective paths will show up at the receiver at different times based on the length of the path. Summing these together at the receiver causes distortions, both constructively and destructively.

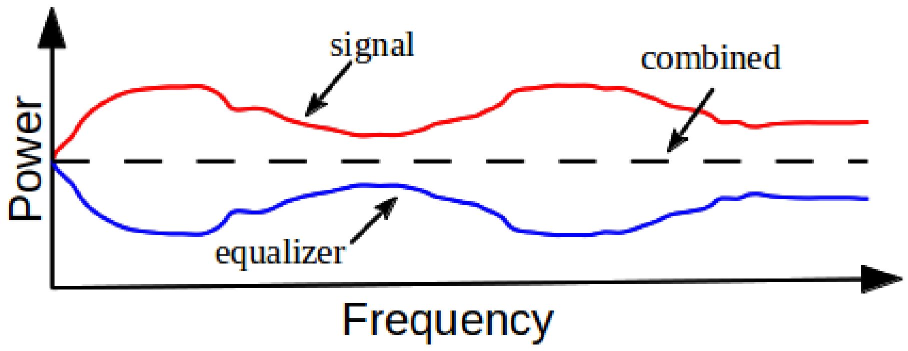


The impact of the combination of these signals at the receiver is distortions of the signal. If the difference in time between reflections is small enough relative to the width of the symbol, the distortion can be within the symbol -- intra-symbol interference. When the reflections are longer than the symbol time, the reflection from one symbol will affect the signals following -- another reason for inter-symbol interference.

We need to correct for this behavior, and we can do so using a mechanism very much like a stereo equalizer. In fact, we call them equalizers. With a stereo equalizer, we can change the gain of certain frequencies to either suppress or enhance those signals -- bass and treble being the common ones. I've created a very simple example called **multipath_sim.grc** to help us explore what this looks like in the frequency domain.

The **multipath_sim.grc** simply sets up a channel model to provide a channel with five equalizer knobs, four of which we can change. These knobs are simply set up equally in frequency and we can adjust them from 0 to 1. At a value of 1, the knob will allow those frequencies to pass without hindrance. At a value of 0, they will produce a deep null in the spectrum, which will affect all those frequencies around it. When running the script, it's best to set the frequency plot to average (by right clicking the graph and selecting "Average"; a value of "Medium" is good for this).

While in this example, we are controlling the frequency domain explicitly, what we're really playing with is the ability to create an equalizer that can correct or adjust the frequency response of a received signal. Ultimately, the goal is shown in the figure below where the multipath channel is creating some distortion in the signal as shown in the frequency domain. The task of the equalizer is to invert that channel. Basically, we want to undo the distortion that's caused by the channel such that the output of the equalizer is flat. But, instead of adjusting the taps by hand, we have algorithms that update these taps for us. Our job is to use the right equalizer algorithm and set up the parameters. One important parameter here is the number of taps in the equalizer. As we can see in our simulation, five taps gives fairly coarse control over the frequency response. Alternatively, the more taps, the more time it takes to both compute the taps as well as run the equalizer against the signal.

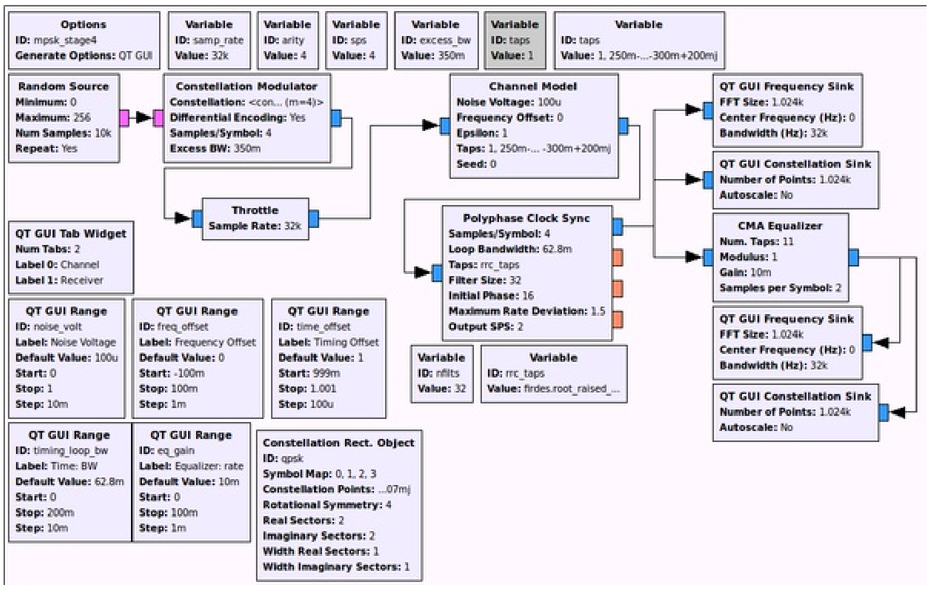


Equalizers

GNU Radio comes with two easily usable equalizers. The CMA Equalizer and the LMS DD Equalizer. The CMA, or Constant Modulus Algorithm, is a blind equalizer, but it only works on signals that have a constant amplitude, or modulus. This means that digital signals like MPSK are good candidates since they have points only on the unit circle (think back to the experiment we did where we locked the signal timing but had a frequency offset; what we were seeing was the unit circle).

The CMA algorithm accepts the number of taps to use in the equalizer, which will be based on some combination of an educated guess, known best practices, and maybe some actual knowledge of the channel itself. We want to keep this number small to reduce the overhead of the algorithm while making sure there are enough degrees of freedom to correct for our channel.

In the **mpsk_stage4.grc** example, we use the CMA algorithm with 11 taps. This is a simulation, and that number has worked well in the past. Play around with it and see how it affects performance, both from a computational and signal standpoint.



Options
ID: mpsk_stage4
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 32k

Variable
ID: arity
Value: 4

Variable
ID: sps
Value: 4

Variable
ID: excess_bw
Value: 350m

Variable
ID: taps
Value: 1

Variable
ID: taps
Value: 1, 250m-...-300m+200mj

Random Source
Minimum: 0
Maximum: 256
Num Samples: 10k
Repeat: Yes

Constellation Modulator
Constellation: <con... (m=4)>
Differential Encoding: Yes
Samples/Symbol: 4
Excess BW: 350m

Throttle
Sample Rate: 32k

Channel Model
Noise Voltage: 100u
Frequency Offset: 0
Epsilon: 1
Taps: 1, 250m-... -300m+200mj
Seed: 0

QT GUI Frequency Sink
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 32k

QT GUI Constellation Sink
Number of Points: 1.024k
Autoscale: No

Polyphase Clock Sync
Samples/Symbol: 4
Loop Bandwidth: 62.8m
Taps: rrc_taps
Filter Size: 32
Initial Phase: 16
Maximum Rate Deviation: 1.5
Output SPS: 2

CMA Equalizer
Num. Taps: 11
Modulus: 1
Gain: 10m
Samples per Symbol: 2

QT GUI Frequency Sink
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 32k

QT GUI Constellation Sink
Number of Points: 1.024k
Autoscale: No

QT GUI Tab Widget
Num Tabs: 2
Label 0: Channel
Label 1: Receiver

QT GUI Range
ID: noise_volt
Label: Noise Voltage
Default Value: 100u
Start: 0
Stop: 1
Step: 10m

QT GUI Range
ID: freq_offset
Label: Frequency Offset
Default Value: 0
Start: -100m
Stop: 100m
Step: 1m

QT GUI Range
ID: time_offset
Label: Timing Offset
Default Value: 1
Start: 999m
Stop: 1.001
Step: 100u

Variable
ID: nfilts
Value: 32

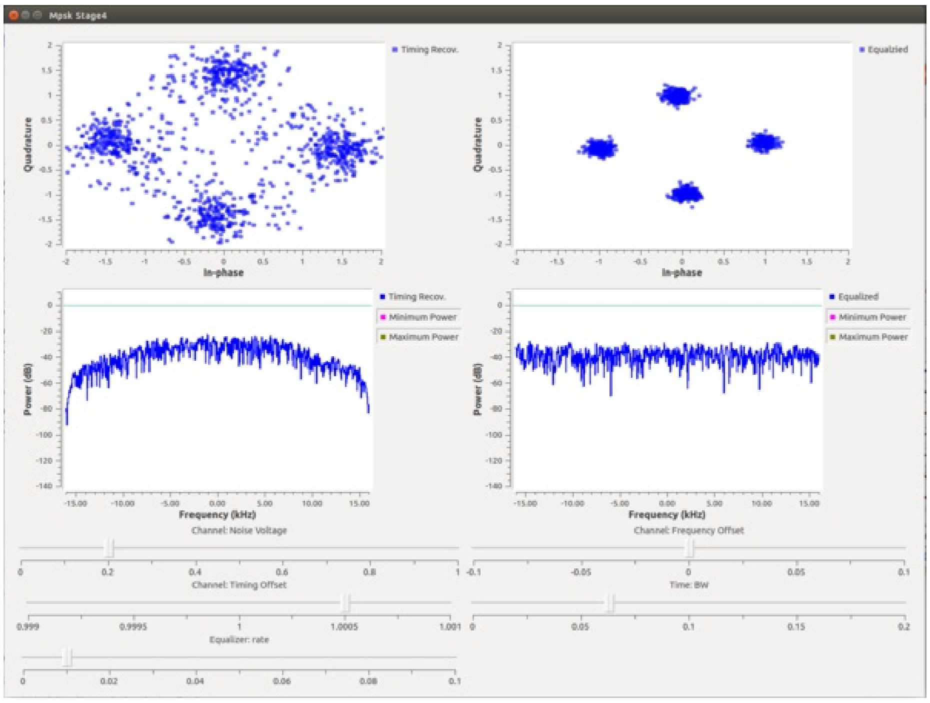
Variable
ID: rrc_taps
Value: firdes.root_raised_...

QT GUI Range
ID: timing_loop_bw
Label: Time: BW
Default Value: 62.8m
Start: 0
Stop: 200m
Step: 10m

QT GUI Range
ID: eq_gain
Label: Equalizer: rate
Default Value: 10m
Start: 0
Stop: 100m
Step: 1m

Constellation Rect. Object
ID: qpsk
Symbol Map: 0, 1, 2, 3
Constellation Points: ...07mj
Rotational Symmetry: 4
Real Sectors: 2
Imaginary Sectors: 2
Width Real Sectors: 1
Width Imaginary Sectors: 1

We can watch the CMA algorithm converge. Note, too, that since we have both a clock sync and equalizer block, they are converging independently, but the one stage will affect the next stage. So there is some interaction going on here while both are locking on to the signal. In the end, though, we can see the effect of the time-locked multipath signal before and after the equalizer. Before the equalizer, we have a very ugly signal, even without noise. The equalizer nicely figures out how to invert and cancel out this channel so that we have a nice, clean signal again. We can also see the channel itself and how it flattens out nicely after the equalizer.



Play with the taps provided to the channel model block to change the multipath. The current taps were simply randomly generated to provide a multipath profile with no real mathematical basis for them. For more complex channel models with fading simulations, see the Channel Models page in the GNU Radio manual.

LMS-DD Equalizer

A good challenge now is to use the Least Mean Squared Decision-Directed (LMS-DD) equalizer block. There is a lot of overlap between the parameters, except for one major feature. Instead of a blind equalizer like the CMA, this equalizer requires knowledge of the received signal. The equalizer needs to know the constellation points in order to correct, and it uses decisions about the samples to inform how to update the taps for the equalizer.

This equalizer is great for signals that don't fit the constant modulus requirement of the CMA algorithm, so it can work with things like QAM-type modulations. On the other hand, if the SNR is bad enough, the decisions being made are incorrect, which can ruin the receiver's performance. The block is also more computationally complex in its performance. When the signal is of good quality, though, this equalizer can produce better quality signals because it has direct knowledge of the signal. A common model is to use a blind equalizer for initial acquisition to get the signal good enough for a directed equalizer to take over. We won't try and do something like that here, though.

As a challenge, take the **mpsk_stage4.grc** file and replace the CMA equalizer recovery with the LMS-DD equalizer and see if you can get it to converge. This block uses a Constellation Object, so part of the difficulty here is creating the proper constellation object for the QPSK signal and applying that.

More Equalizers

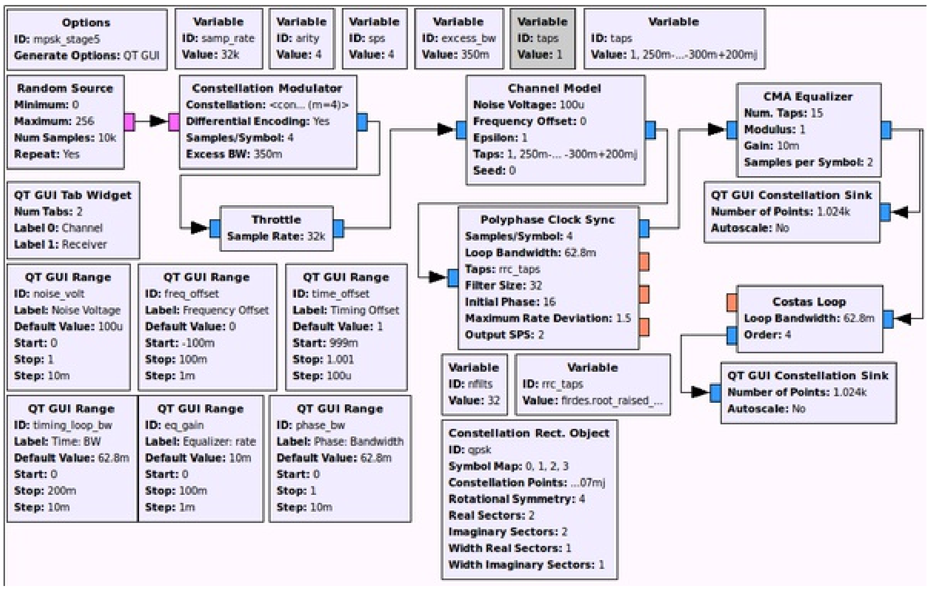
There are plenty of other equalizer algorithms out there that are not yet part of the main GNU Radio release.

Phase and Fine Frequency Correction

Given that we've equalized the channel, we still have a problem of phase and frequency offset. Equalizers tend not to adapt quickly, and so a frequency offset can be easily beyond the ability of the equalizer to keep up. Also, if we're just running the CMA equalizer, all it cares about is converging to the unit circle. It has no knowledge of the constellation, so when it locks, it will lock at any given phase. We now need to correct for any phase offset as well as any frequency offset.

Two things about this stage. First, we'll use a second order loop so that we can track both phase and frequency, which is the derivative of the phase over time. Second, the type of recovery we'll deal with here assumes that we are doing *fine* frequency correction. So we must be sure that we are already within a decent range of the ideal frequency. If we are too far away, our loop here won't converge and we'll continue to spin. There are ways to do coarse frequency correction, but we won't get getting into those here.

For this task, we're going to use the Costas Loop in example **mpsk_stage5.grc** (the Constellation Receiver can also be used here). The Costas Loop block can synchronize BPSK, QPSK, and 8PSK. The Constellation Receiver will lock to any given constellation object, though depending on the constellation, the decision making function may be more or less complex.


Options
ID: mpsk_stage5
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 32k

Variable
ID: arity
Value: 4

Variable
ID: sps
Value: 4

Variable
ID: excess_bw
Value: 350m

Variable
ID: taps
Value: 1

Variable
ID: taps
Value: 1, 250m-....-300m+200mj

Random Source
Minimum: 0
Maximum: 256
Num Samples: 10k
Repeat: Yes

Constellation Modulator
Constellation: <con... (m=4)>
Differential Encoding: Yes
Samples/Symbol: 4
Excess BW: 350m

Channel Model
Noise Voltage: 100u
Frequency Offset: 0
Epsilon: 1
Taps: 1, 250m-.... -300m+200mj
Seed: 0

CMA Equalizer
Num. Taps: 15
Modulus: 1
Gain: 10m
Samples per Symbol: 2

QT GUI Tab Widget
Num Tabs: 2
Label 0: Channel
Label 1: Receiver

Throttle
Sample Rate: 32k

Polyphase Clock Sync
Samples/Symbol: 4
Loop Bandwidth: 62.8m
Taps: rrc_taps
Filter Size: 32
Initial Phase: 16
Maximum Rate Deviation: 1.5
Output SPS: 2

QT GUI Constellation Sink
Number of Points: 1.024k
Autoscale: No

QT GUI Range
ID: noise_volt
Label: Noise Voltage
Default Value: 100u
Start: 0
Stop: 1
Step: 10m

QT GUI Range
ID: freq_offset
Label: Frequency Offset
Default Value: 0
Start: -100m
Stop: 100m
Step: 1m

QT GUI Range
ID: time_offset
Label: Timing Offset
Default Value: 1
Start: 999m
Stop: 1.001
Step: 100u

Costas Loop
Loop Bandwidth: 62.8m
Order: 4

Variable
ID: nfilts
Value: 32

Variable
ID: rrc_taps
Value: firdes.root_raised_...

QT GUI Constellation Sink
Number of Points: 1.024k
Autoscale: No

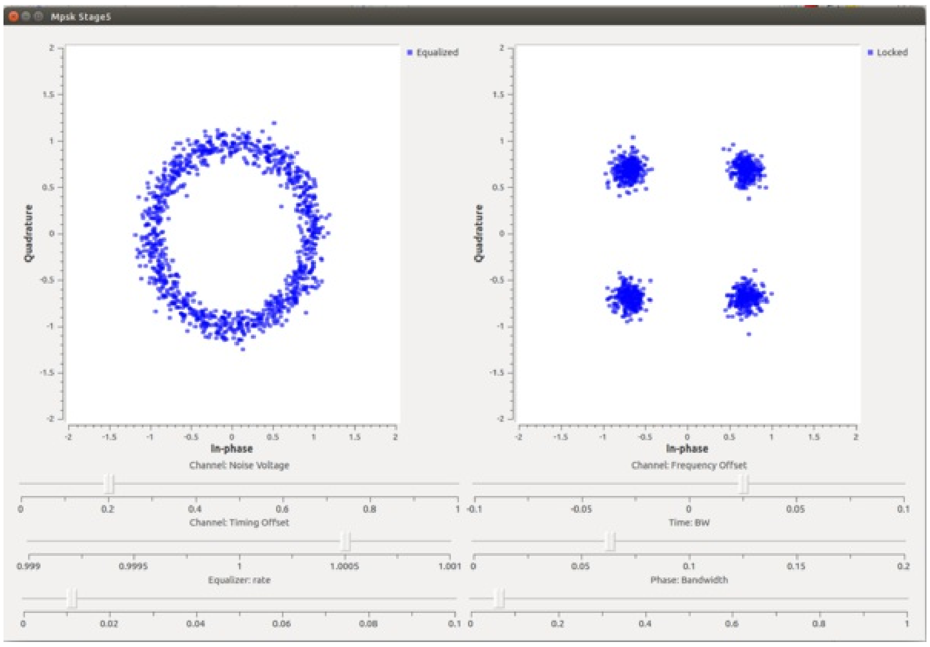
QT GUI Range
ID: timing_loop_bw
Label: Time: BW
Default Value: 62.8m
Start: 0
Stop: 200m
Step: 10m

QT GUI Range
ID: eq_gain
Label: Equalizer: rate
Default Value: 10m
Start: 0
Stop: 100m
Step: 1m

QT GUI Range
ID: phase_bw
Label: Phase: Bandwidth
Default Value: 62.8m
Start: 0
Stop: 1
Step: 10m

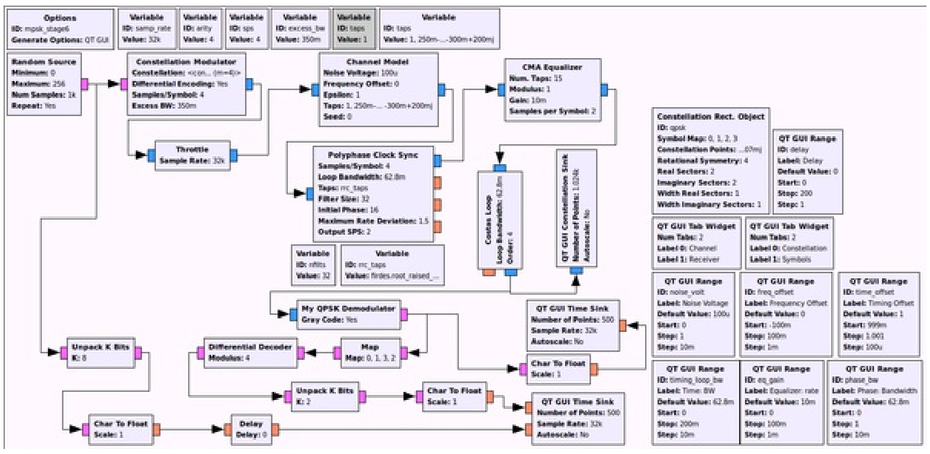
Constellation Rect. Object
ID: qpsk
Symbol Map: 0, 1, 2, 3
Constellation Points: ...07mj
Rotational Symmetry: 4
Real Sectors: 2
Imaginary Sectors: 2
Width Real Sectors: 1
Width Imaginary Sectors: 1

This block, like all of our others, uses a second order loop and is therefore defined with a loop bandwidth parameter. The other thing it needs to know is the order of the PSK modulation, so 2 for BPSK, 4 for QPSK, and 8 for 8PSK. In the next image, we have set noise, timing offset, a simple multipath channel, and a frequency offset. After the equalizer, we can see that the symbols are all on the unit circle, but rotating due to the frequency offset that nothing is yet correcting for. At the output of the Costas loop block, we can see the locked constellation like we started with plus the extra noise, which we can't do anything about.



Decoding

Now that the hard part is done, we get to decode the signal. Using the **my_qpsk_demodulator** block from Tutorial 4, we have a locked constellation that can be properly turned back into bits. Using the **mpsk_stage6.grc** example flowgraph, we insert the demodulator after the Costas loop, but our work is not quite done. At this point, we get out symbols from 0 to 3 because this is the size of our alphabet in a QPSK scheme. Furthermore, of those 0-3 symbols, how do we know for sure that we have the same mapping of symbols to constellation points that we did when we transmitted? Notice in our discussion above that nothing we did had any knowledge of the transmitted symbol-to-constellation mapping, which means we might have an ambiguity of 90 degrees in the constellation. Luckily, we avoided this problem by transmitting *differential* symbols. We didn't actually transmit the constellation itself, we transmitted the difference between symbols of the constellation by setting the Differential setting in the PSK Mod block to True. So now we undo that.



Options
ID: mpsk_stage6
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 32k

Variable
ID: arity
Value: 4

Variable
ID: sps
Value: 4

Variable
ID: excess_bw
Value: 350m

Variable
ID: taps
Value: 1

Variable
ID: taps
Value: 1, 250m-...-300m+200mj

Random Source
Minimum: 0
Maximum: 256
Num Samples: 1k
Repeat: Yes

Constellation Modulator
Constellation: <(con... (m=4)>
Differential Encoding: Yes
Samples/Symbol: 4
Excess BW: 350m

Channel Model
Noise Voltage: 100u
Frequency Offset: 0
Epsilon: 1
Taps: 1, 250m-... -300m+200mj
Seed: 0

CMA Equalizer
Num. Taps: 15
Modulus: 1
Gain: 10m
Samples per Symbol: 2

Throttle
Sample Rate: 32k

Polyphase Clock Sync
Samples/Symbol: 4
Loop Bandwidth: 62.8m
Taps: rrc_taps
Filter Size: 32
Initial Phase: 16
Maximum Rate Deviation: 1.5
Output SPS: 2

Costas Loop
Loop Bandwidth: 62.8m
Order: 4

QT GUI Constellation Sink
Number of Points: 1.024k
Autoscale: No

Constellation Rect. Object
ID: qpsk
Symbol Map: 0, 1, 2, 3
Constellation Points: ...07mj
Rotational Symmetry: 4
Real Sectors: 2
Imaginary Sectors: 2
Width Real Sectors: 1
Width Imaginary Sectors: 1

QT GUI Range
ID: delay
Label: Delay
Default Value: 0
Start: 0
Stop: 200
Step: 1

QT GUI Tab Widget
Num Tabs: 2
Label 0: Channel
Label 1: Receiver

QT GUI Tab Widget
Num Tabs: 2
Label 0: Constellation
Label 1: Symbols

Variable
ID: nfilts
Value: 32

Variable
ID: rrc_taps
Value: firdes.root_raised_...

My QPSK Demodulator
Gray Code: Yes

QT GUI Time Sink
Number of Points: 500
Sample Rate: 32k
Autoscale: No

QT GUI Range
ID: noise_volt
Label: Noise Voltage
Default Value: 100u
Start: 0
Stop: 1
Step: 10m

QT GUI Range
ID: freq_offset
Label: Frequency Offset
Default Value: 0
Start: -100m
Stop: 100m
Step: 1m

QT GUI Range
ID: time_offset
Label: Timing Offset
Default Value: 1
Start: 999m
Stop: 1.001
Step: 100u

Unpack K Bits
K: 8

Differential Decoder
Modulus: 4

Map
Map: 0, 1, 3, 2

Char To Float
Scale: 1

QT GUI Range
ID: timing_loop_bw
Label: Time: BW
Default Value: 62.8m
Start: 0
Stop: 200m
Step: 10m

QT GUI Range
ID: eq_gain
Label: Equalizer: rate
Default Value: 10m
Start: 0
Stop: 100m
Step: 1m

QT GUI Range
ID: phase_bw
Label: Phase: Bandwidth
Default Value: 62.8m
Start: 0
Stop: 1
Step: 10m

Unpack K Bits
K: 2

Char To Float
Scale: 1

QT GUI Time Sink
Number of Points: 500
Sample Rate: 32k
Autoscale: No

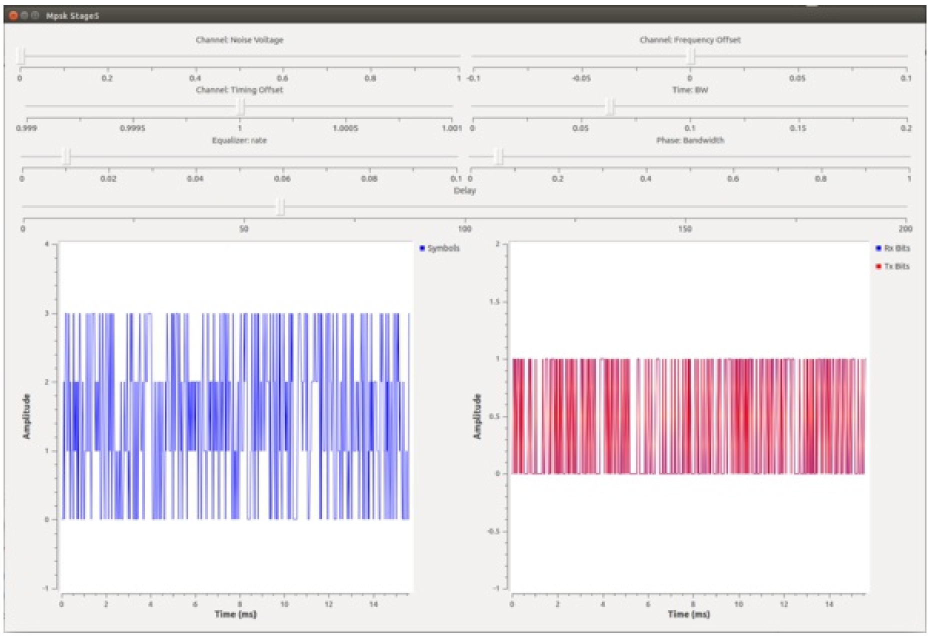
Char To Float
Scale: 1

Delay
Delay: 0

The flowgraph uses the Differential Decoder block to translate the differential coded symbols back to their original symbols due to the phase transitions, not the absolute phase itself. But even out of here, our symbols are not exactly right. This is the hardest part about demodulation, really. In the synchronization steps, we had basic physics and math on our side. Now, though, we have to interpret some symbol based on what someone else said it was. We, basically, just have to know this mapping. And luckily we do, so we use the map:"http://gnuradio.org/doc/doxygen/classgr_1_1digital_1_1map__bb.html%22 block to convert the symbols from the differential decoder to the original symbols we transmitted. At this point, we now have the original symbols from 0-3, so lets unpack those 2 bits per symbol into bits using the unpack bits block. Now, we have the original bit stream of data!

But how do we know that it's the original bit stream? To do so, we'll compare to the input bit stream, which we can do because this is a simulation and we have access to the transmitted data. But of course, the transmitter produced *packed bits*, so we again use the unpack bit block to unpack from 8-bits per byte to 1-bit per byte. We then convert these streams to floating point values of 0.0 and 1.0 simply because our time sinks only accept float and complex values. Comparing these two directly would show us... nothing. Why? Because the receiver chain has many blocks and filters that delay the signal, so the received signal is some number of bits behind. To compensate, we have to delay the transmitted bits by the same amount using the delay block. I have programmed this to set the delay to 0 to start with.

You can then adjust the delay to find the correct value and see how the bits synchronize.



For the answer click here

You can also subtract one signal from the other to see when they are synchronized as the output will be 0. Adding noise and other channel affects can then be easily seen as bit errors whenever this signal is not 0.

As a final experiment, notice that we are using a finite length random number generator, so we should be able to see the pattern in the received signal. Using the Time Raster QTGUI plotting tool, set it up so that you can see this pattern. Keep in mind that the Time Raster plot, like all of our plotters, samples the stream, and so the resulting display might not be exactly what you would expect. But the pattern itself should be visible if you have set it up correctly.

(Also note that the Time Raster plotter does not work under QWT5, and it will warn you if you are trying to run it using that library version. QWT6 fixed some raster plotting issues to make this usable.)

Quiz

- What does the roll-off factor change?
- Why do we need synchronization, and which types of synchronization are there?
- What can we do about noise (AWGN)?
- What would we need to change if the encoder we use in the final example was not differential?
- Why do we put the map block in front of the differential decoder?

Category: Guided Tutorials

Tutorials > Guided Tutorials