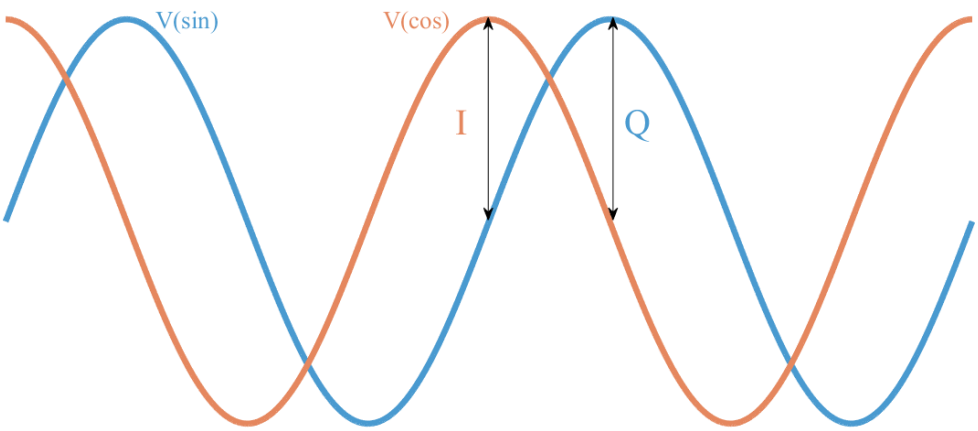
USRP Hardware Lab #1

EELE 445 Telecommunication Systems

Montana State University

Basics of SDRs and IQ Signaling



Author

Alex Brisson

Preface

At Montana State University, EELE 445 Telecommunication Systems is a course which aims to introduce undergraduate students to the theory behind many types of communications systems. Most notably, the course focuses on modulation techniques to optimize channel capacity, distortions introduced by channel effects, and the structures of the transmitter and receiver that enable such systems. An integral part in solidifying the understanding of these concepts is to simulate and analyze practical communication systems in the lab. By creating real-time simulations that incorporate telecommunications hardware, such as the Universal Software Radio Peripheral (USRP), students will be exposed to practical implementations and the engineering challenges involved in designing and building several types of communications systems.

Pre-Lab

Please read the following introduction. It is necessary that you have gained a basic understanding of how the SDR works before the first lab.

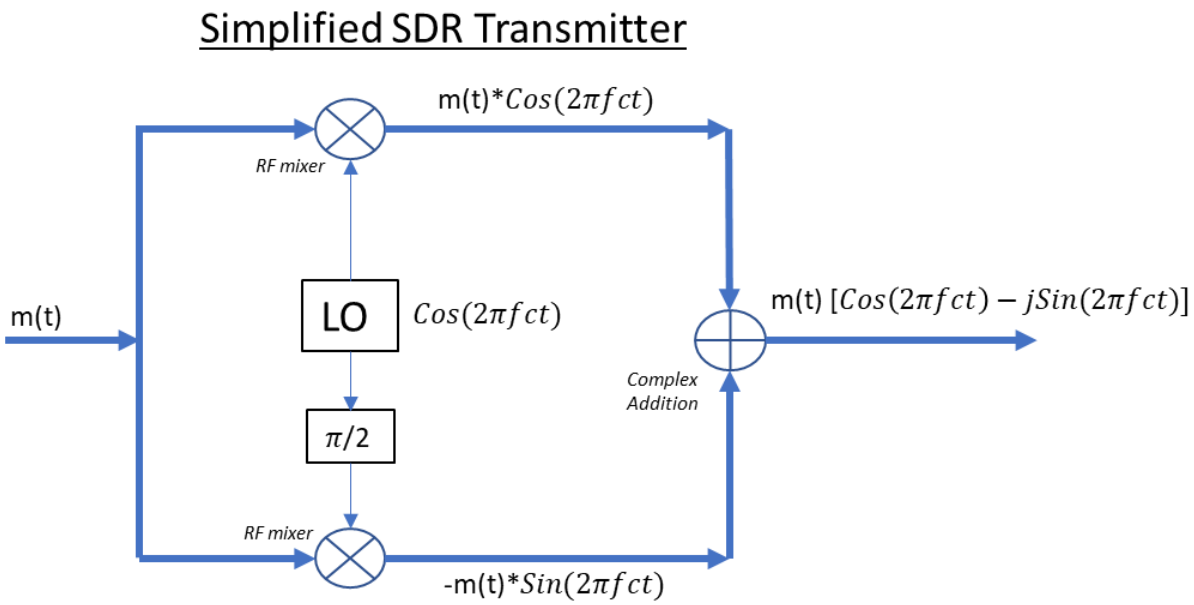
Introduction

SDR Structure and Operation

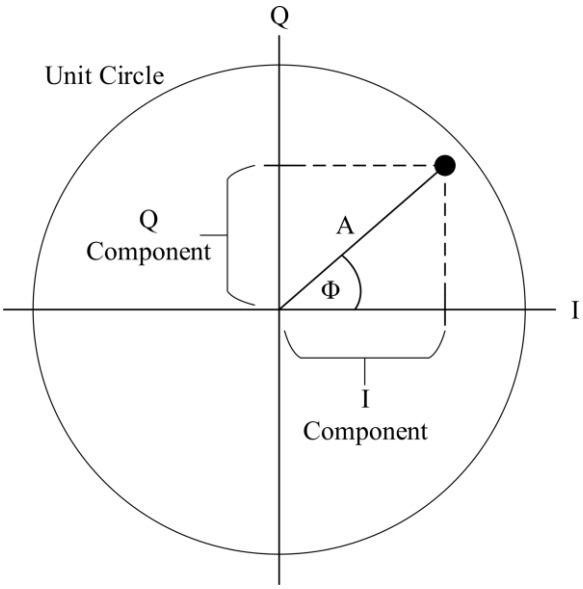
The USRP, or more generally the software defined radio (SDR), is a tool that will allow us to explore some of the fundamentals of communications systems. In this lab, our goal is to learn how to prepare a "baseband" message signal m(t) such that it can be efficiently transmitted to our receiver. However, before we jump straight into the lab activities, it is important to understand what the SDR does and appreciate why its methods of operation are so significant in the realm of communications.

So, what is an SDR? An SDR is a radio transceiver, which means it can transmit and receive information in the lower part of the radio frequency band of the electromagnetic spectrum ~MHz. Below is an image of our device and the basic structure of its transmitter.

Ettus Research USRP N210

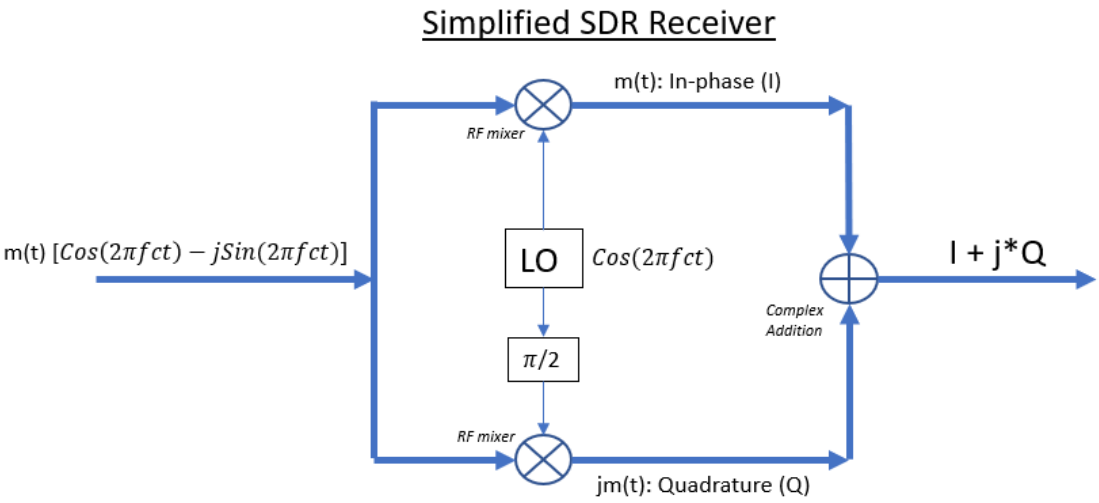


Using a local oscillator, or more simply, a locally generated sinusoid (LO), the SDR transmitter takes the baseband signal (centered around 0Hz) m(t) and up-converts it to a higher carrier frequency, fc. Not obviously, it does this up-conversion in two parts. The SDR multiplies m(t) by the LO in the top branch and again by the LO shifted in 90 degrees in phase in the bottom branch. The two up-converted signals are added to form a complex signal. You may be wondering why the SDR messes around with complex numbers. Signals that we actually transmit are *real* in nature, right? So why do we need complex numbers? It turns out, complex numbers reveal their significance and perhaps, *necessity*, when it comes to implementing modulation formats such as QPSK, 8PSK, 16QAM and so on. Basically, by forming a complex carrier "cos + j*sin" we generate–in the analog domain (real signals)–two sinusoids that oscillate in quadrature (90 degrees apart in phase) at the carrier frequency. By adjusting *just* the amplitudes of the sine and the cosine (we do this by generating m(t)), their superposition yields a symbol vector in the complex plane that is mapped to a specific amplitude and phase. See the figure below. 'A' represents the complex symbol vector with amplitude and phase and can be decomposed by the amplitudes of its components, I and Q.



The SDR's job is then to detect the amplitudes of the sine and the cosine at the receiver, in other words, the in-phase component of the vector A (I) and the quadrature component of the vector A (Q). With just these two pieces of information, we are able to decode the amplitude and phase of our modulated waveform which carries our encoded information or bits.

So, what about the receiver?



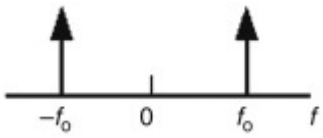
At the receiver port, the SDR does a very similar thing to what it does at the transmitter port. The SDR multiplies the incoming complex waveform by two phase-shifted LOs at fc. Through a trig identity, we find that this multiplication shifts the spectral content of m(t) centered at fc back to baseband. Now that m(t) is shifted down to lower frequencies, the electronics inside the SDR samplers can detect I and Q with ease. The only thing left to do is add up I and Q and determine

the encoded amplitude and phase, right? If only things were that easy… As we'll see in this lab, many signal impairments happen along the way as the signal propagates through the channel to the receiver. Additionally, the samplers or "ADC and DAC clocks" at the transmitter and receiver are not exactly matched in frequency and phase. This causes timing errors that can lead to incorrect estimates of I and Q. These are some of the topics we will be looking at and exploring in the following activities and hardware labs.

The Significance of I and Q

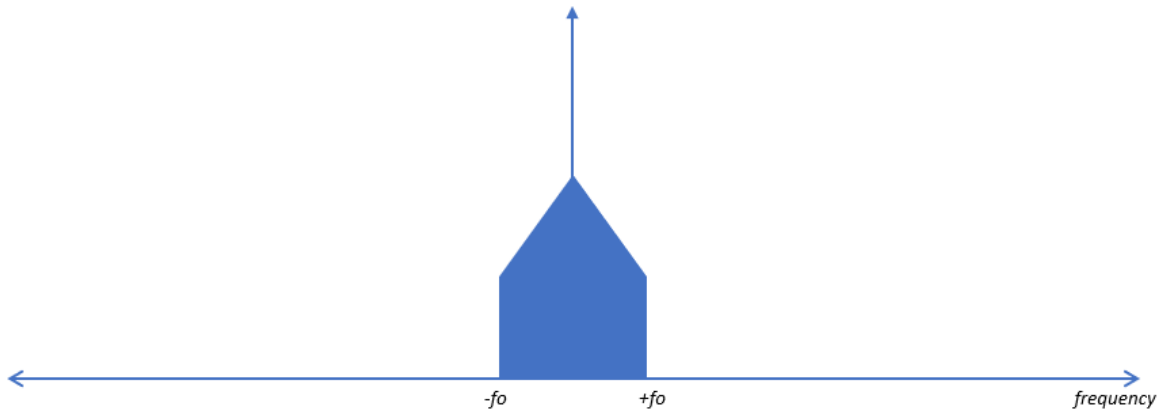
As I pointed out before, our main goal is to learn how to prepare an information-carrying baseband message signal to be transmitted, received, and successfully decoded. It turns out the most commercially successful way, or rather, the *most natural* way of doing this has been found through transmitting and detecting I and Q signals. Not only does I and Q give us the ability to encode information using the carrier phase and amplitude, the superposition of I and Q expresses a unique property that deem it *necessary* for implementations of coherent communications as mentioned earlier. To understand this, we must first consider the spectrum of a real signal in its most primitive form: the cosine. The cosine exhibits two spectral peaks at +fo and -fo as shown below.

$$cos(2\pi fot) = \frac{1}{2}[e^{+j2\pi fot} + e^{-j2\pi fot}]$$

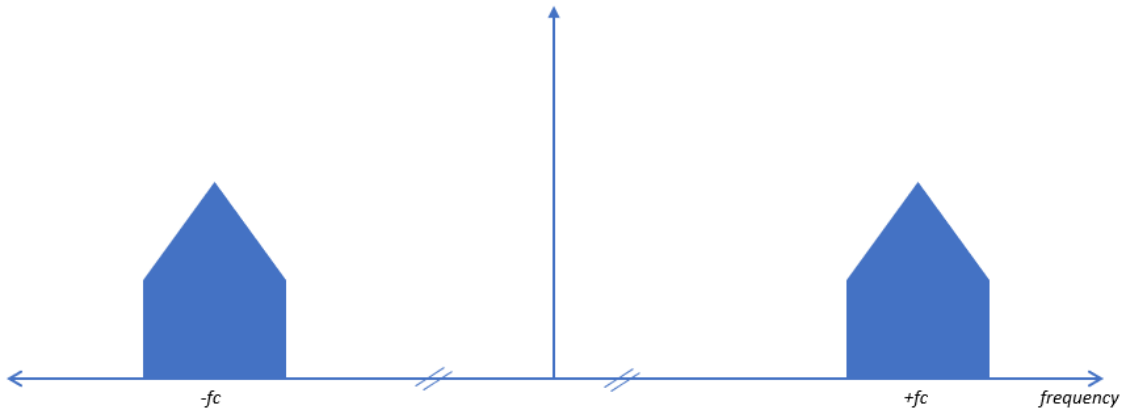


You can then imagine adding up all the cosines composite to any real signal and realize that *all* real signals exhibit this property and that their spectrum consists of mirrored images at positive and negative frequencies. Okay, so what? Well, the *main function* of the SDR is to up-convert *and* down-convert the spectral content of our message, m(t), to and from baseband centered at 0 Hz, and I told you that our signal m(t), and hence its spectrum, can become distorted as it propagates through the channel. Therefore, if we up-convert m(t) to a carrier frequency and subsequently down-convert it to baseband after it has become distorted by the channel, its spectrum can no longer be represented by purely real signals. Basically, this means the spectrum of the received signal is no longer symmetric about 0 Hz and we have no hope in compensating the full effects of the channel. Now, I understand it's possible none of that helped, so I'll attempt to draw diagrams of what happens to the spectrum of m(t) to recap.

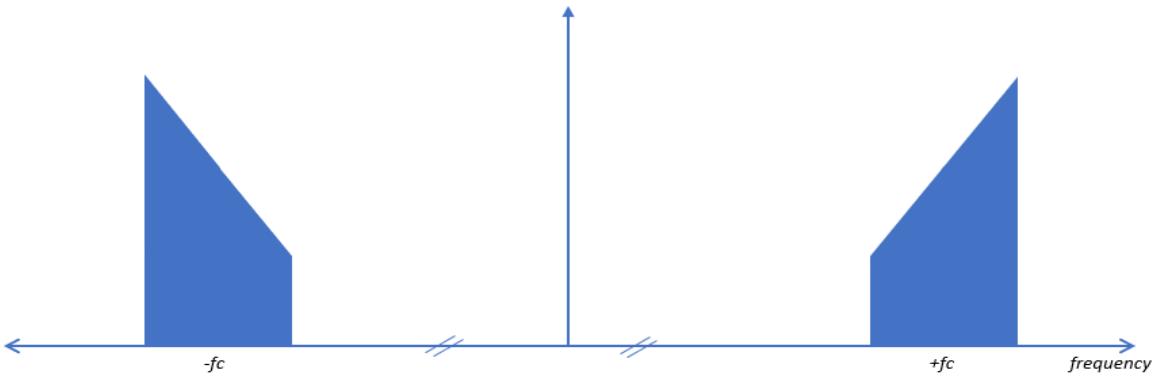
1.Below is an example of what the spectrum could look like of our *real* signal m(t). Notice it is symmetric about 0 Hz at positive and negative frequencies.



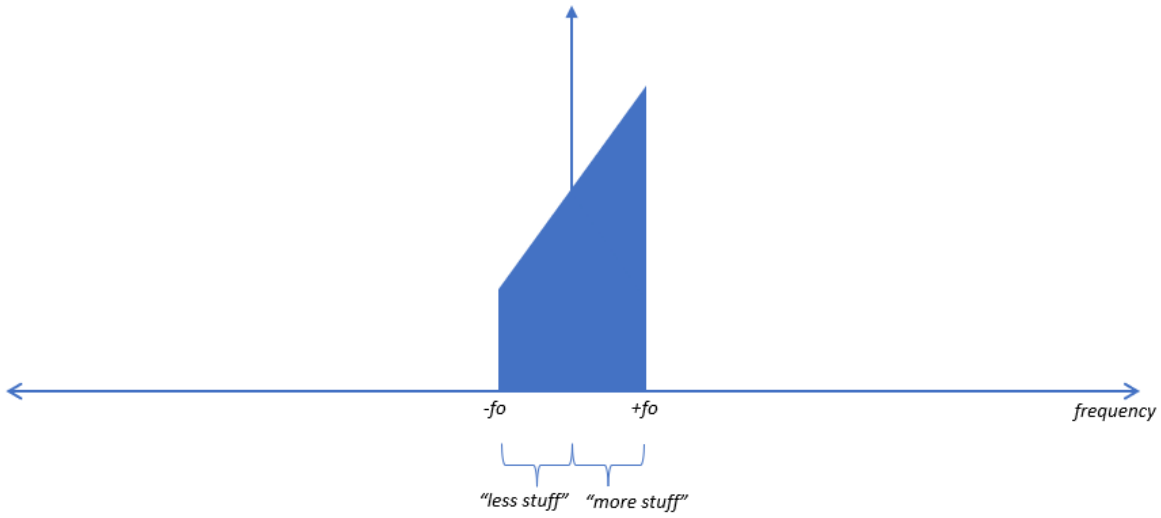
2.When we up-convert the spectrum to fc, we get copies at +fc and -fc (again, because we transmit a real signal).



3.The channel then distorts the signal, and hence its spectrum in some way, but remains symmetric.



4. The SDR then down-converts the spectrum at fc to baseband. Notice, this spectrum is no longer symmetric about 0 Hz and cannot be represented by a real signal. For example, there is more "stuff" or energy in the positive frequencies than in the negative frequencies and as a result, we *need* complex numbers to completely represent this signal at the receiver.



In a nutshell, the superposition of I and Q forms a single complex exponential at either a positive or negative frequency depending on the sign of the sine. We can therefore use this property to construct the spectrum above. This property has been proven by the famous mathematician, Leonhard Euler and his famous formula, Euler's formula.

$$e^{\pm j\,(2\pi fot)} = cos(2\pi fot) \pm jsin(2\pi fot) = I \pm jQ$$

Gnuradio Download and Install Instructions: Optional

Gnu Radio Companion is an open-source graphical user interface with python host language. The application allows the user to simulate communications systems and control many types of SDRs with the capability to display real-time performance visualizations.

1. Proceed to the following webpage: Releases · ryanvolz/radioconda (github.com)
2. Find the latest release (top of page) and under the "Assets" tab, download the executable file (.exe) "radioconda" for your computer's operating system.
3. Follow instructions to install. Once completed, you should have access to the application *gnu radio companion.*

Additional Note: It is recommended you come to lab with a printed copy of this manual since you will be using your lab computer to do the lab. You are also free to use your own computer, so long as you have access to an ethernet port for the USRP.

Part 1: Sinusoids and Complex Down-Conversion

1. Open gnuradio from your computer's start menu. Once the software opens, click *file -> save as* and save your file to a folder you can locate later with a descriptive name, i.e. "usrp_lab1_sinusoidIQ". Next, you should see an options block in the top left of the workspace, double click and title your workspace something like "Basics of SDRs and IQ". Next, you will change the baseband sample rate by double clicking on the box that has ID "samp_rate" and set the variable to 500 kHz and click "Apply" and "OK" (you can do this by typing "500e3" or type out all the zeros: "500000").

2. You will create a new variable for the center frequency of the USRP. This center frequency is the same frequency 'fc' as discussed in the introduction and is commonly referred to as the *tuning frequency* of the radio. Using the search icon in the top right above the tab section, search for the block "variable". Click, hold, and drag the block to the workspace. Double click and change the ID to "center_freq" and set its value to 10 MHz.

3. At this point you should have 2 variables: one for the baseband sample rate and another for the tuning frequency of the radio. Next, search "usrp source" and place the UHD source into the workspace- this block represents the USRP receiver and will give us the raw output data we use to analyze our signals. Under the "general" tab, change the parameter "sync" to "no sync" and under the "RF options" tab, change "Center Freq" to your variable for the tuning frequency, "center_freq".

4. You will now place down a time scope and frequency analyzer to visualize what the USRP will give us once we send it a signal. Search for the block "gui time sink" and "gui frequency sink" and place both blocks next to the output port of the usrp source block. Click and drag an arrow from the output port of the usrp source block to the input ports of the time and frequency sink blocks. Your workspace should now look like the following figure.

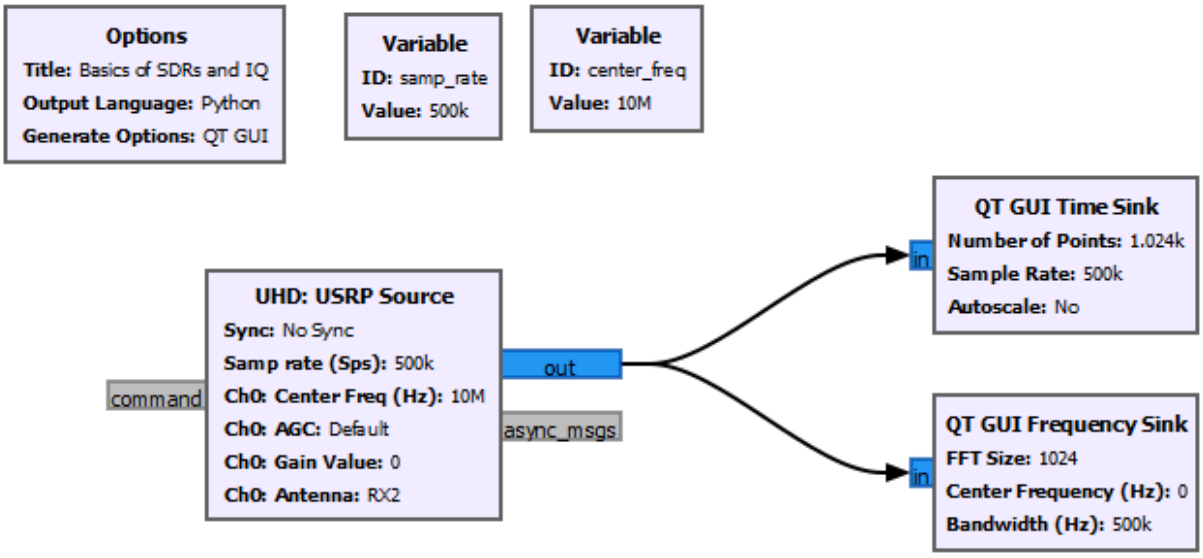
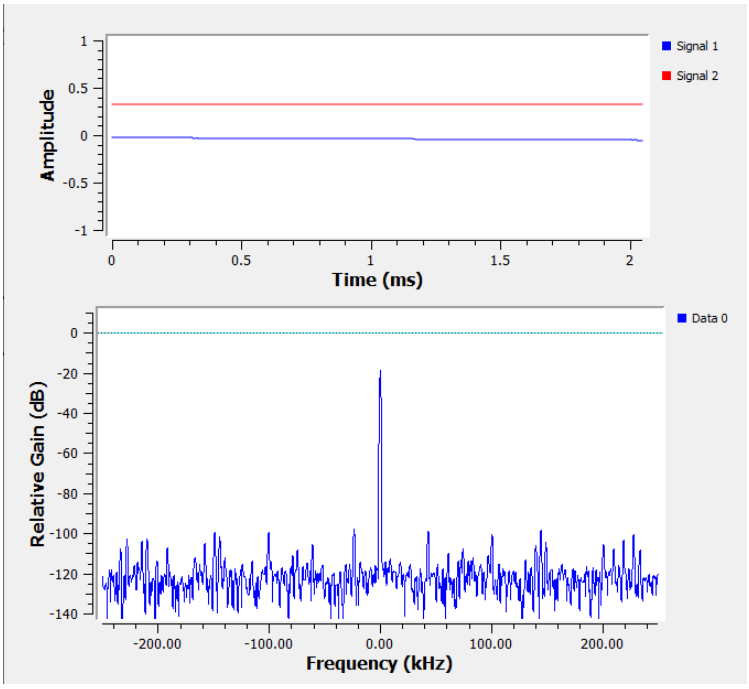


Figure 1: Sinusoidal IQ Down-Conversion

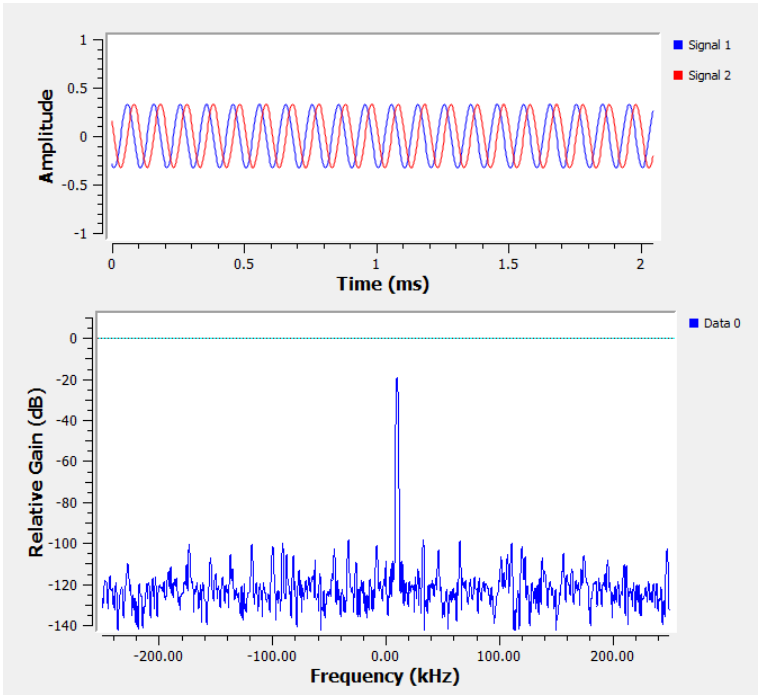
5. Connect the coaxial cable from channel 1 of the function generator to the RF2 port (receive port) of the USRP. Set the function generator's function to sine wave with frequency 10MHz, amplitude 1V, and 0V offset.

6. Once you have configured the function generator's settings, turn on the generator's output and click the white play button in the tab section of gnuradio. You should see two red and blue lines moving up and down in the time scope and a distinct spectral peak at 0Hz. Now, try increasing the function generator frequency up by 10kHz. Those red and blue lines should start to rotate much faster. Next, aim your mouse cursor over the time scope and click down on your mouse scroll wheel. This should open the scope menu. Click "stop" to stop the acquisition. Using your mouse cursor, verify the timestamps between two adjacent peaks of either the blue or red curve. This value should be ~0.1ms or 10kHz. Now, try changing the function generator frequency to 10kHz below 10MHz (9.99MHz), and make the same verification.

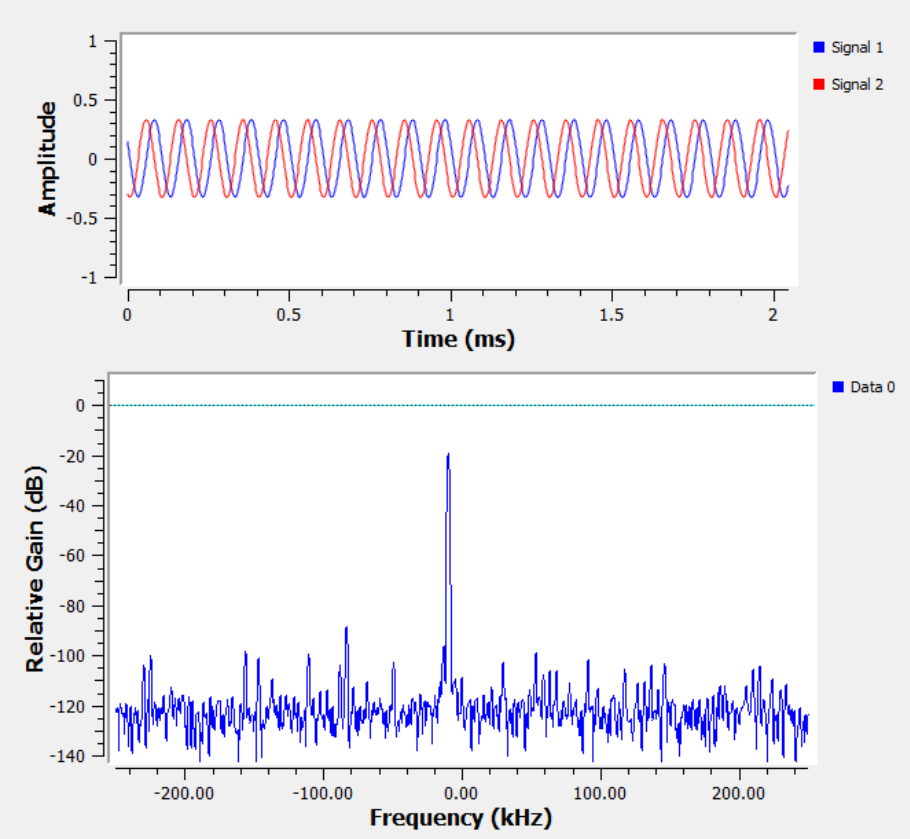
#6 Results



AWG freq = 10MHz



AWG freq = 10.01MHz



AWG freq = 9.99MHz

7. Ok, so at this point, let's back up for a second and think about what just happened. You sent a sinusoid of frequency 10MHz to the USRP's receiver port and the USRP told you your signal is centered at 0Hz. This makes sense because your radio is also tuned to 10MHz, so it down-converted your 10MHz real signal from the function generator and converted that signal to a complex signal at baseband (centered at 0Hz). This becomes more obvious when you increase the frequency by +/- 10kHz. Those sinusoids represent the in-phase and quadrature components (I and Q) of your down-converted complex signal. Okay, so did you catch what happened in the spectrum when you changed the frequency by +/- 10kHz? What about the red and blue curves? Make a note/drawing in your notebook about what changed in each case and explain in your own words why you think it is advantageous for the SDR to use complex numbers (ie I and Q) to represent baseband signals. Feel free to include Euler's identity in your explanation and determine which color the I and Q signals correspond to.

Looking at the IQ waveforms from part 6, we should notice that any frequencies in our baseband signal that are below the carrier frequency after up-conversion are subsequently down-converted by the SDR to baseband as negative frequencies. This property constitutes the necessity for complex numbers which are constructed using Euler's formula. We should also note the sign of the quadrature i.e., for positive frequencies, the sine (Q) is positive and hence leads the cosine (I)

in phase. For negative frequencies, the sine flips, and lags the cosine in phase. I is blue and Q is red.

Part 2: Square Pulses, Complex Down-Conversion, and Frequency Error

After transmitting a simple sinusoid, you should now have a pretty good idea of what the SDR receiver is doing. So now, let's try and make things a little more interesting by transmitting a BPSK signal. As a reminder, the BPSK constellation looks like the following figure. With BPSK, we transmit pulses to represent amplitudes of either +1 or -1 of the real (I) component. These symbols can either represent the bits '1' when I = 1 or '0' when I = -1. Notice, we don't transmit a quadrature component with BPSK modulation, i.e. Q = 0.

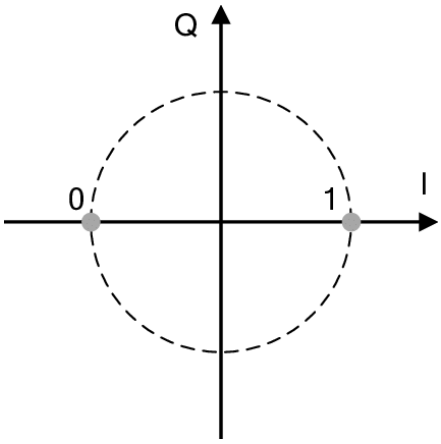


Figure 2: BPSK Constellation Diagram

1. Open the provided gnuradio file "usrp_bpsk.grc" and save a local version of the file using "save as". You can keep the file name but add your name, so you know it's yours. This file generates random samples (+1, -1) to represent the amplitude of the real component (I). Each sample is interpolated (repeated) 32 times so that each symbol is 32 samples long. The symbols are then sent to the usrp transmitter and subsequently received by the usrp receiver. Time scopes and frequency analyzers are placed to view the transmit signal and receive signal.

2. Run the file and you should see something like the following figure.

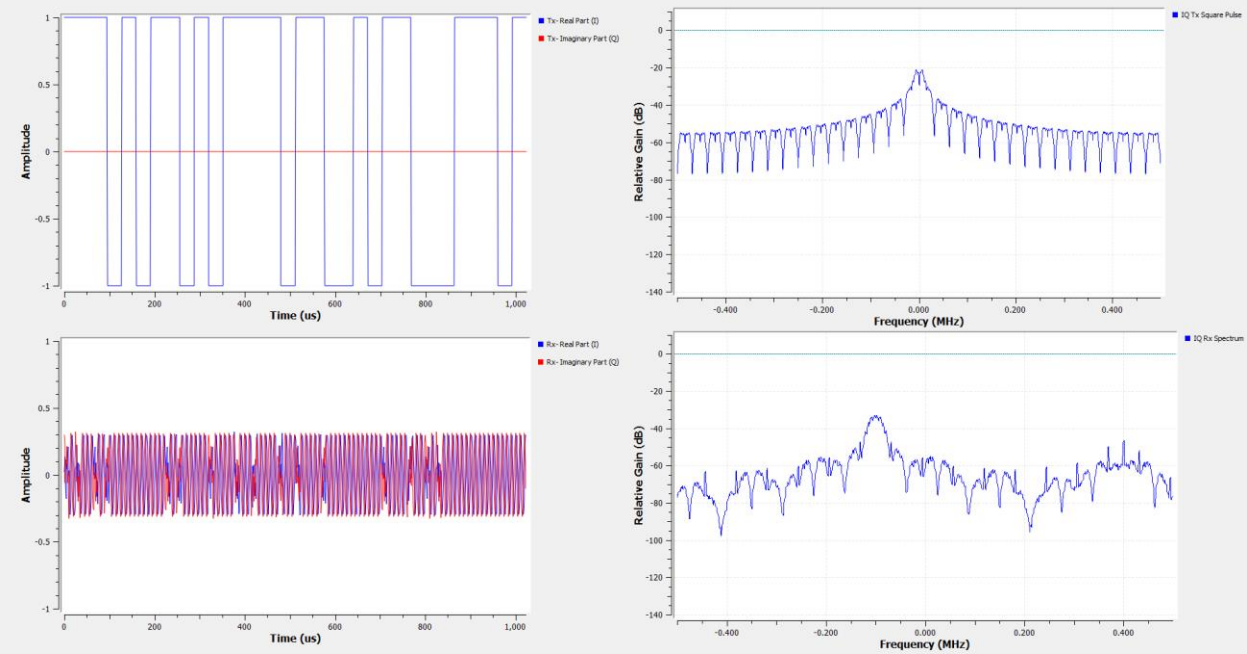
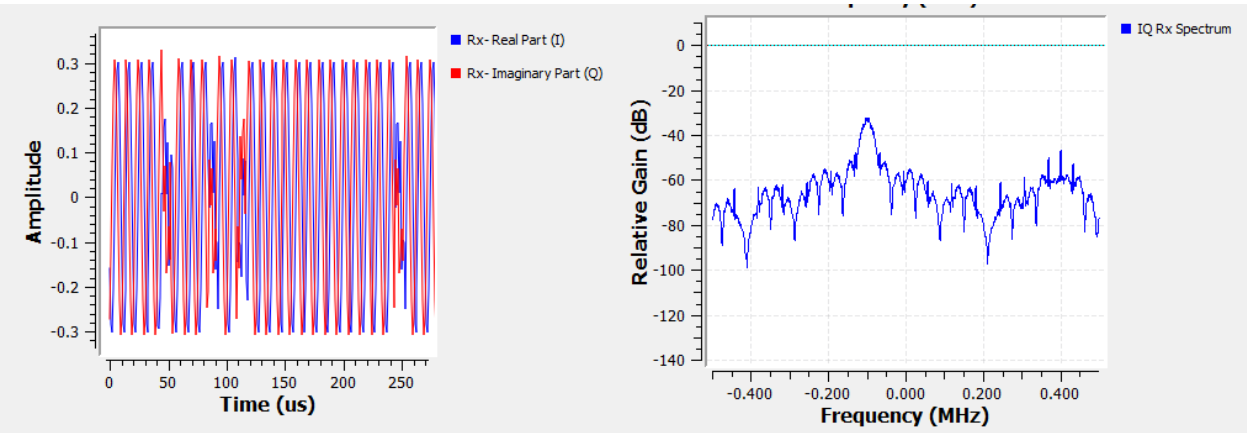


Figure 3: USRP BPSK

3. Use the scope menu to stop the receive time scope acquisition and zoom the window into ~200us. You should be able to see the discontinuities in the time domain (which represent the phase transitions of the BPSK signal), but obviously, something is wrong here. Describe what doesn't make sense about the received time scope data. Should there be an imaginary part?

4. We can make a similar observation that something is wrong by looking at the spectrum of our received pulses (bottom right). Describe what could be the problem with the received data by observing its spectrum and what could have caused this problem. What should the SDR receiver do? Think back to part 1.

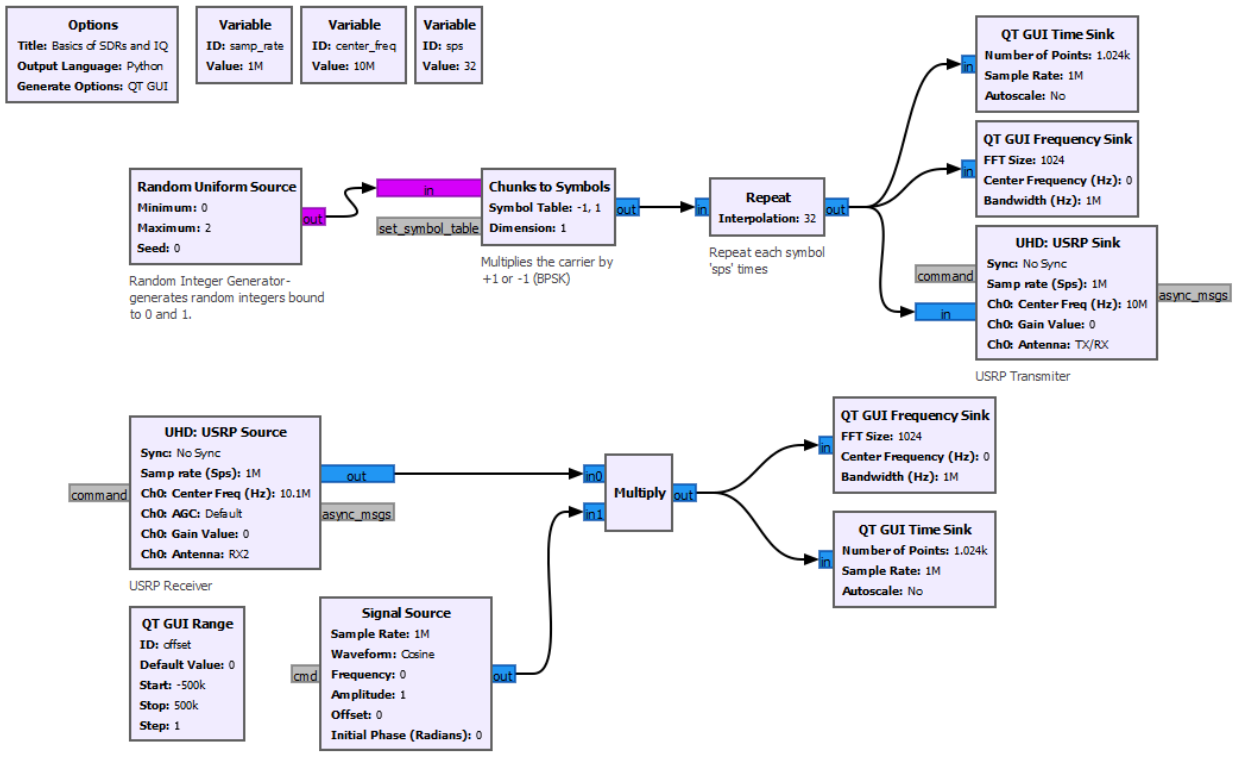
#3/4 Results



If the modulation scheme is BPSK, we would expect there to be no imaginary part (Q), despite the time domain indicating that there exists an imaginary component in the received signal. The SDR

is expected to down-convert the center of the transmitted spectrum to baseband at 0Hz, although it appears the spectrum came back coarsely shifted away from baseband, hence why we see oscillating sines and cosines. Therefore, this shift must be corrected by multiplying the received signal by a complex exponential with the correct compensating frequency.

5. You will now place some additional blocks to correct the error in the received signal. Search and place the blocks "Signal Source", "QT Gui Range", and "Multiply". You will need to multiply your received signal by a complex exponential (the signal source) to compensate for the frequency offset in the received signal. Use the range block to create a tunable variable called "offset". Set its default value to 0 and allow the range to span [-samp_rate/2, +samp_rate/2] and step size 1. Set the frequency parameter inside your signal source to "offset" and use the multiply block to multiply the offset signal with the output of the USRP receiver. Connect the multiplied result to the time scope and frequency analyzer. Once you are finished, your workspace should look like the following figure.

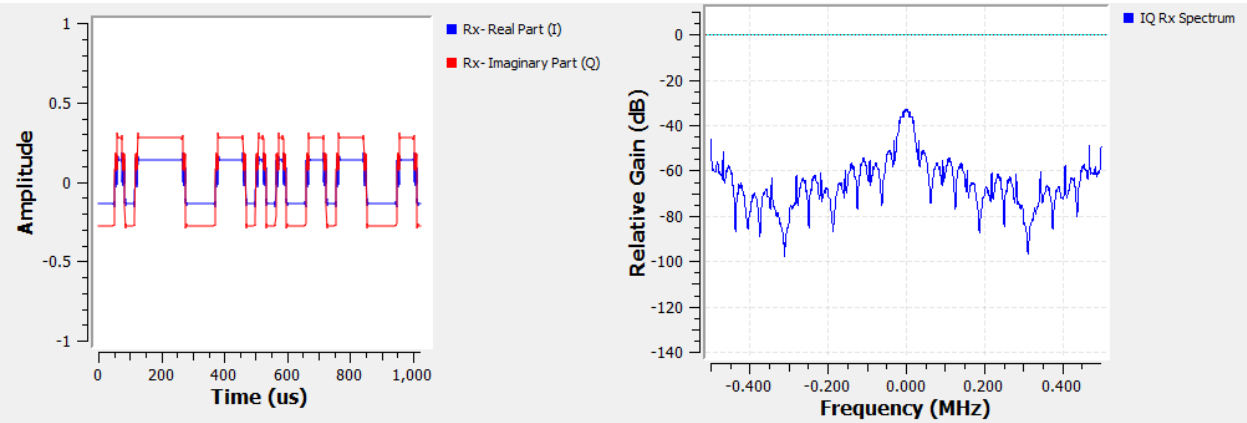


6. Re-run the file and adjust the 'offset' slider until you have corrected the frequency offset in the received spectrum. What was the offset? What happened to the received signal in the time domain?

7. You have just corrected your signal for any *coarse* frequency offsets between the transmitter and receiver, but is your signal still fully representative of the one you sent? What is still happening

to I and Q in the time domain of your received signal? Do you have any thoughts as to why this is happening? Again, think back to what was mentioned in the introduction.

#6/7 Results



The coarse shift was compensated using a complex signal generator at center frequency 100kHz. This frequency offset was manually inserted into the tuning frequency of the receiver for demonstration purposes. One should notice the difference in the time domain, notably, square pulses as opposed to oscillating sines and cosines. Slow sloshing in amplitude is resultant of residual clock drift between the Tx and Rx.

8. Open the USRP source block and tab over to "RF Options". Remove the frequency offset. Yes, I know, I just tuned the receiver LO to a different frequency which is why our spectrum came back shifted away from 0Hz. It turns out, the transmitter and receiver LOs are *relatively* well matched in the USRP, but in real systems, this is rarely ever the case.

Even after doing a *coarse* frequency correction, the transmitter and receiver clock frequencies still drift around with respect to one another. The two clocks will never be exactly in phase with the same frequency, partly due to differences in environmental conditions. This is why we see a slow "sloshing in amplitude" between I and Q in the time domain, or in the case of the sinusoid, the red and blue lines moving up and down slowly when in theory they should be constant. You can also think of this as very small frequency errors that are changing all the time. It turns out we must use more adaptive algorithms to "lock" the frequency and stop this sloshing around. These techniques will be discussed in future labs.

Part 3: Spectral Efficiency and Pulse Shaping

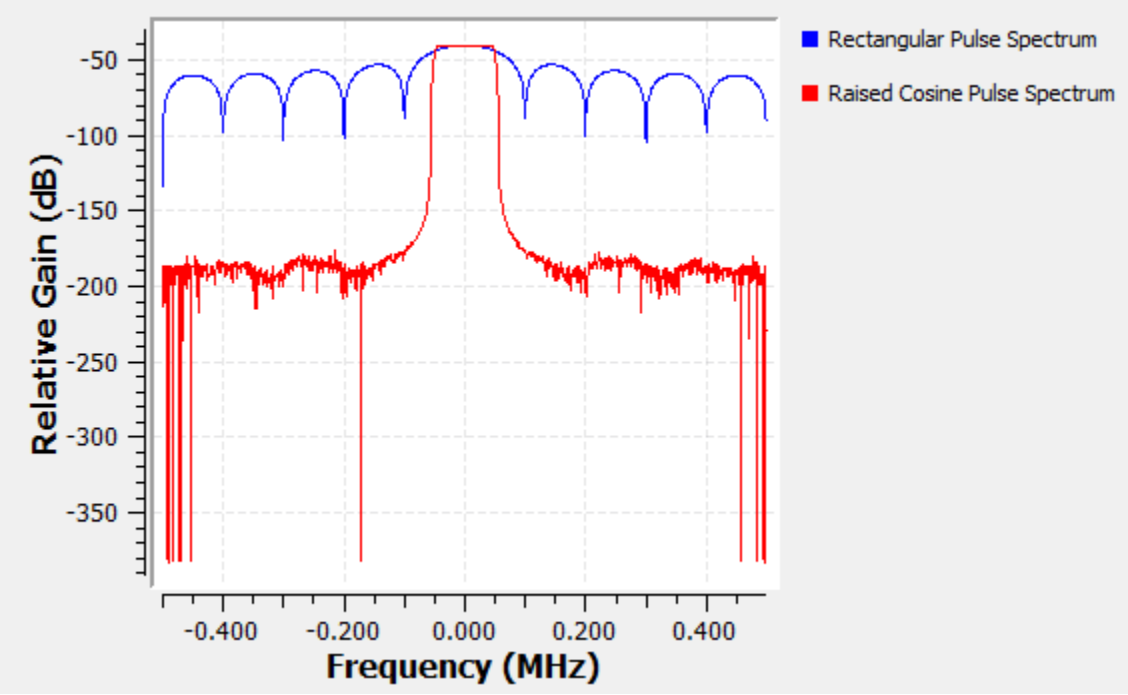
One downside you may have noticed about transmitting square pulses in part 2 is the large amount of bandwidth that is necessary to construct them. A major challenge engineers face when designing and implementing communications systems is in maximizing the *spectral efficiency*, which is to say the rate of information that can be transmitted over a given bandwidth [bit/s/Hz].

Communications engineers always try to maximize this number using many different techniques. Perhaps one way is to include higher-dimensional modulation schemes such as QPSK and 16QAM to encode more bits/symbol, however, we can also work to try and reduce the effective bandwidth necessary to represent our transmitted *pulses* or the symbols they represent. This technique is known as pulse shaping and lends itself to many interesting aspects in communications engineering design.

1. Open the provided file "raisedcosine.grc" and run the file.

2. You should see 4 different plots, but we will start by just looking at the top two plots. The plot in the top left displays two different pulses: a rectangular pulse (the pulse shape we transmitted in part 2), and a raised cosine pulse. If you zoom in (click and hold) on the two pulses, you should see that the raised cosine reaches its peak right in the middle of the square pulse but exhibits oscillating tails that seem to go on forever. The plot to the left is the spectrum of the two pulses. Using your mouse, open the frequency analyzer menu and select auto scale to view the full spectral range. What differences do you notice about the two pulses in terms of the frequency domain? Specifically, after the first main lobe of the square pulse, how much more attenuation does the raised cosine pulse provide?

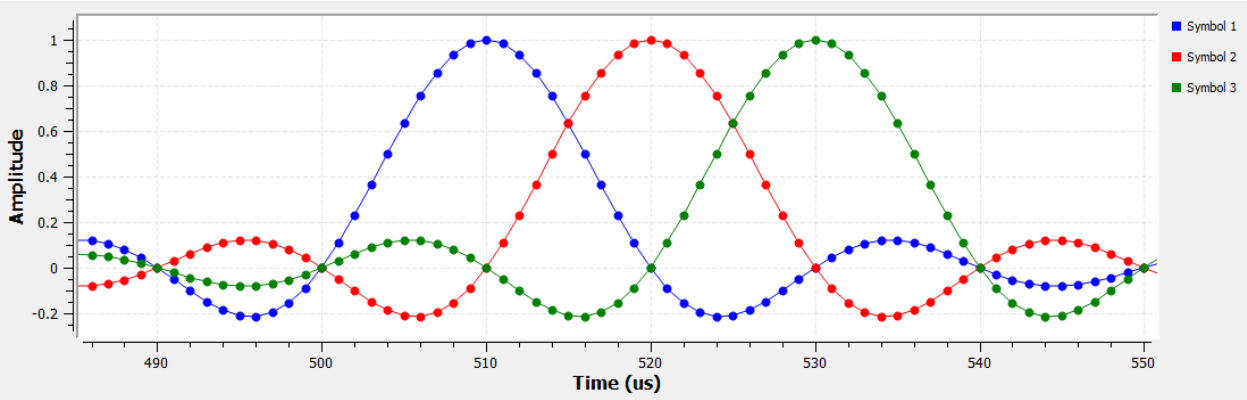
#2 Results



This is an extreme case of RC filtering with excess bandwidth parameter set to 0.1 for "ideal box car" frequency response. One should notice this spectrum is ideal because it doesn't exhibit large side lobes like the rectangular pulse and spectrum for pulses can now be efficiently allocated. Attenuation after first main lobe is ~90dB.

3. It's obvious that the raised cosine pulse gives us a very ideal "box car" like spectrum in the frequency domain so that we can reduce the effective bandwidth that makes up our pulses, but what about the time domain? Are these oscillating tails a problem? It turns out what really makes the raised cosine pulse so special has not only to do with its close-to-ideal frequency response, but more uniquely, the intervals of transitions of those oscillating tails in its impulse response. To clarify this point, consider the middle plot that displays 3 subsequent symbols (supposed to represent values of +1) and imagine them being received by our SDR receiver. Zoom in to the window between ~400-600us. Each symbol has dots which would represent the samples taken by the ADC of our receiver. The receiver's job is then to decide which sample best represents the amplitude of the pulse we transmitted. Now, consider which sample the receiver should pick. Assuming the receiver samples at *exactly* this sample, are the adjacent tails a problem? Why or why not?

#3 Results

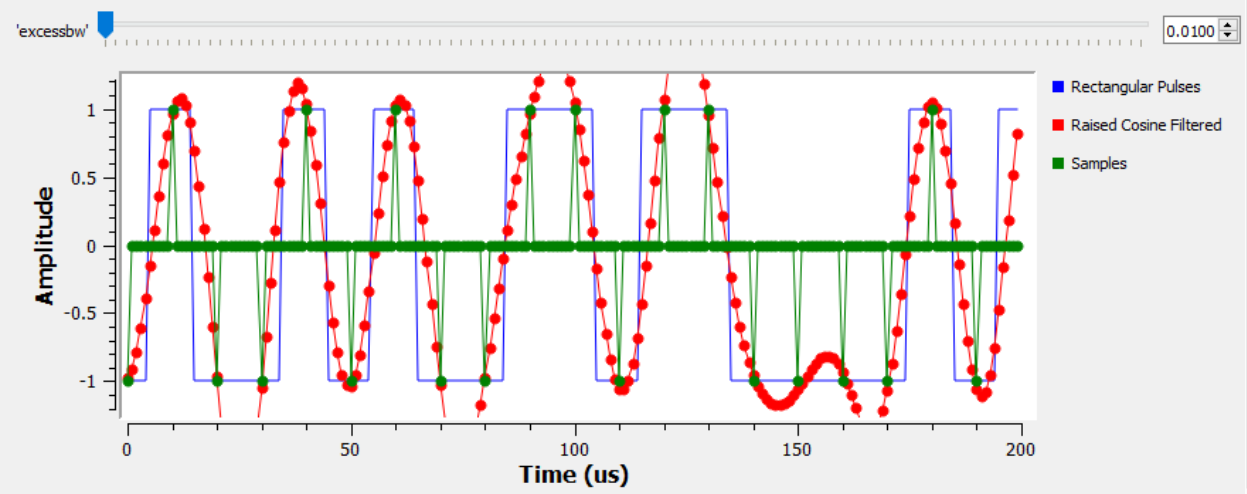


The receiver should pick the sample corresponding to when all the additions of the adjacent tails sum to zero. This occurs at the peak of each symbol, between symbol transitions. If the clock samples at exactly this point, the adjacent tails are not a problem because there will be zero ISI between adjacent symbols.

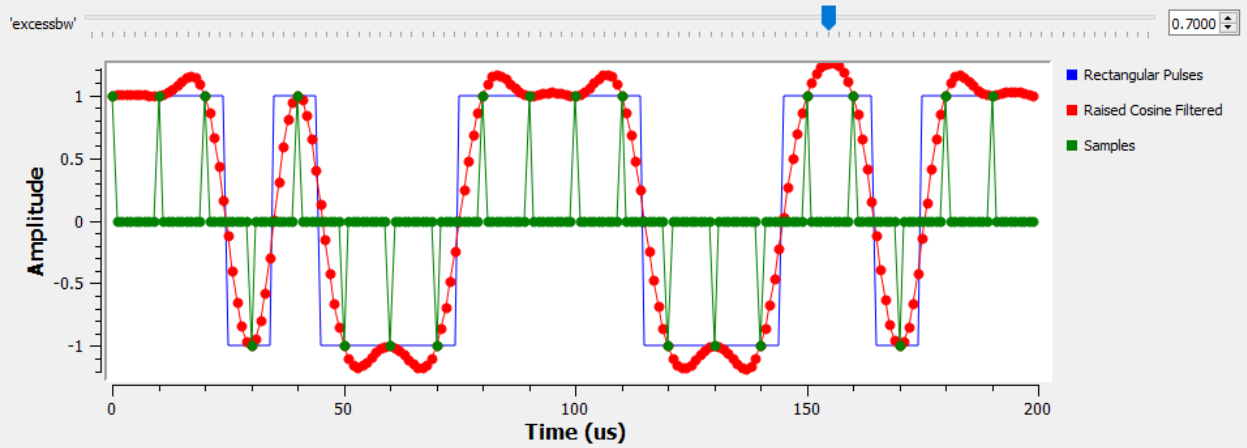
4. Let's now look at the last two plots in the window. The plot to the bottom left shows the sampling instances (green) of our raised cosine and square pulses. One thing you'll notice right away is that the raised cosine pulses have massive fluctuations in amplitude around the nominal symbol amplitudes of (+1,-1), as compared to our generic square pulses. These large fluctuations can become very problematic if our green sampling instances are slightly off. Luckily for us, we have a tuning knob to tame these fluctuations and smoothen out the transitions between a symbol +1 and symbol -1. This tuning knob is known as the *excess bandwidth parameter* and is used to control the amount of "extra frequencies" or excess bandwidth we will allow into our pulses to attempt to smoothen them out. Tune the 'excessbw' variable to ~0.7. Notice, the raised cosine pulses are no longer clipping and exhibit much smoother symbol transitions. What happened to the frequency

and impulse responses when you adjusted the excess bandwidth parameter? Do they still exhibit the same properties as before?

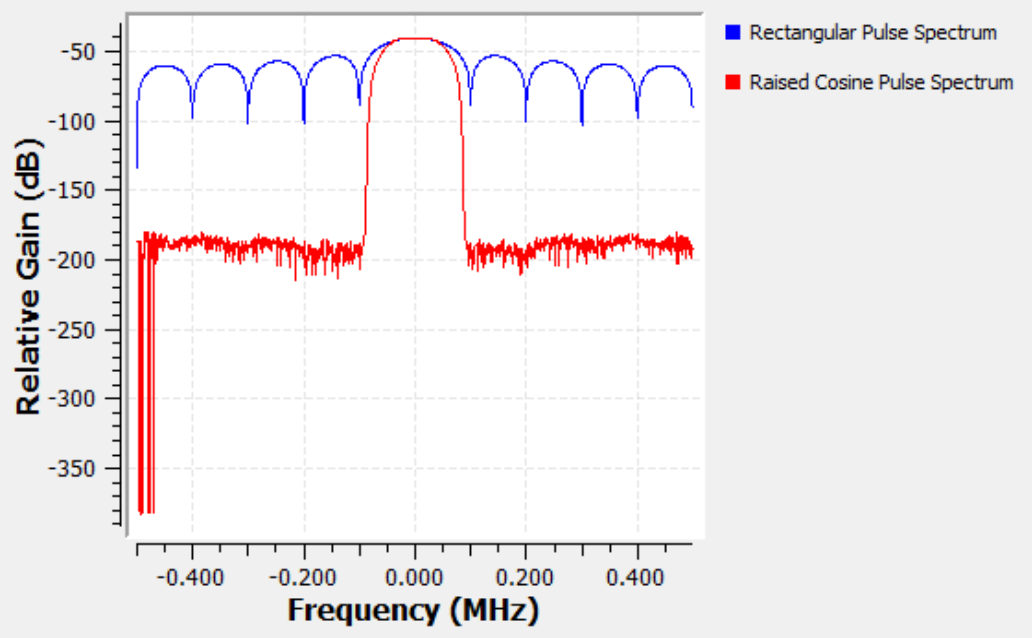
#4 Results



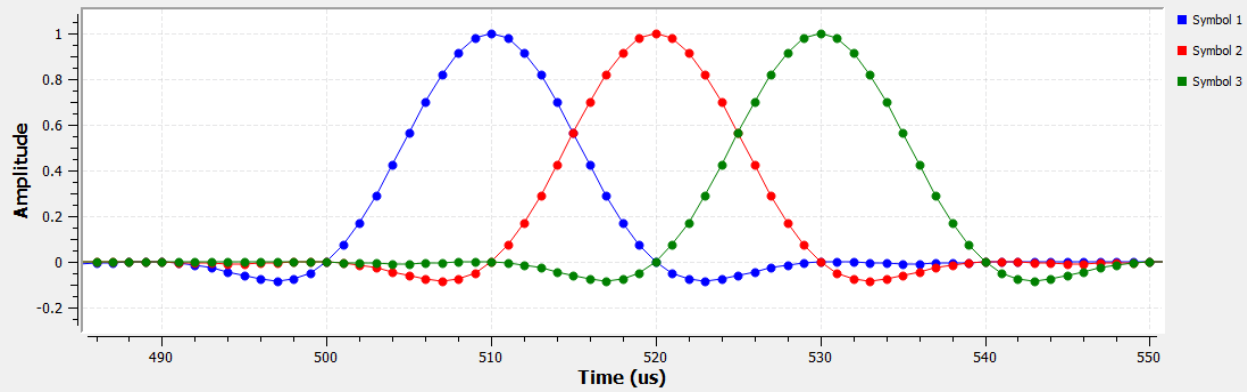
Excess bandwidth = 0.01. we can see massive fluctuations in amplitude in the pulses which leads to clipping.



Excess bandwidth = 0.7. Smoother transitions between symbols without as much clipping.



Excess bandwidth = 0.7. The RC pulses occupy more bandwidth than before but are still contained within the main lobe of the square pulse. Slightly better side lobe attenuation ~97dB.



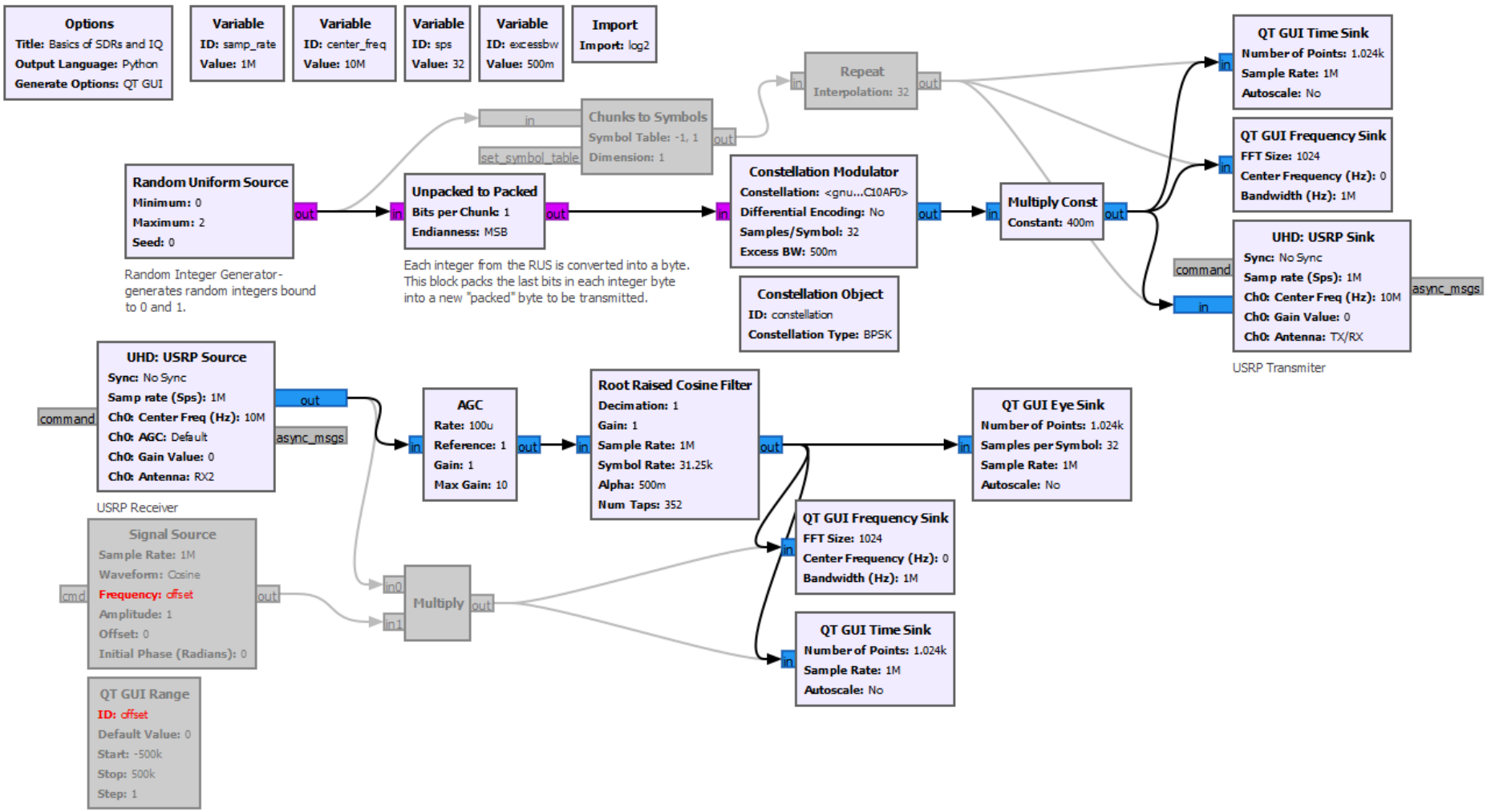
Excess bandwidth = 0.7. The impulse response tails are die off sooner and still maintain zero ISI benefit.

5. Now, try increasing the excess bandwidth to the maximum value. What differences did you notice? What would be one downside of using an excess bandwidth that is too high?

The transitions are very smooth and are well contained within the range of the scope, the impulse response still exhibits zero ISI with smaller tails, and the spectrum occupies most of the main lobe of the square pulse. The major downside to large excess bandwidth is exactly that. Compared to the case when the excess bandwidth was 0.1, at maximum excess bandwidth (1), the pulses occupy nearly twice the amount of spectrum, thus sacrificing spectral efficiency in the frequency domain to obtain more idealistic pulses in the time domain.

6. Let's now make some changes to your BPSK script from part 2 to implement raised cosine pulses.

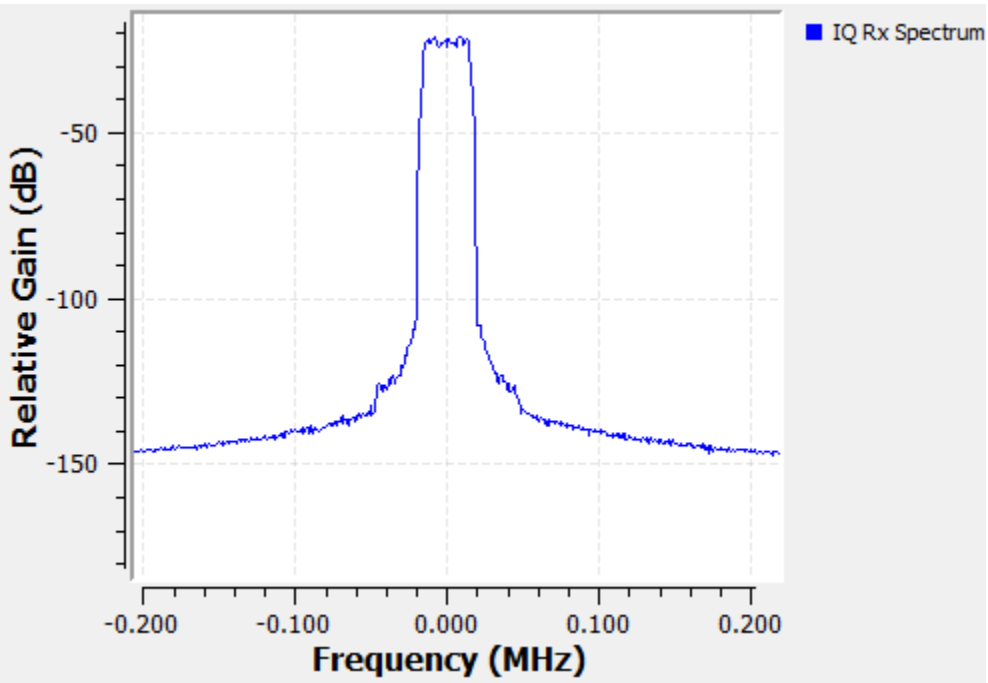
a. First, comment out the following blocks by clicking on them and selecting the ethernet unplug button above the tab section: "chunks to symbols", "repeat", "signal source", "QT GUI Range", and "multiply".
b. Add a variable to the workspace called 'excessbw' and set its value to 0.5.
c. Search for the block "Import" and import the math operator Log2() by typing "from math import log2" in the import space.
d. Search for the block "Unpacked to Packed". Change its data type to 'byte' and bits per chunk value to 1. Connect the Random Uniform Source output to this block's input.
e. Search "Constellation Modulator". Type 'constellation' into the "Constellation" space. Change differential encoding to 'No', samples per symbol value to "samp_rate/sps", and "Excess BW" to 'excessbw'. This block will take our symbols and encode our desired modulation object, which we have named 'constellation'. The block will then filter the symbols using a root raised cosine pulse impulse response. The "root" means we take the square root of the raised cosine impulse response at the transmitter so that we can subsequently apply the square root filter at the receiver to achieve the complete raised cosine pulse. Connect the output of the unpacked to packed block to the input of this block.
f. Search "Constellation Object". Type 'constellation' into its ID space and change the constellation type to BPSK.
g. Search "Multiply Const" and set its value to 0.4. This block will scale the pulses to remain within the bounds of our transmitter. Connect the output of the constellation modulator block to the input of the multiply const block. Then, connect the multiply const block to the time, frequency, and USRP sink blocks.
h. Search "AGC". Set the reference and gain to 1 and set the max gain to 10. Connect the USRP source output to the input of this block. This block will apply automatic gain control to the receive pulses relative to a reference power level.
i. Search "Root Raised Cosine Filter". Set the Symbol Rate to "samp_rate/sps", set alpha to "excessbw", and NumTaps to 11*sps. This block will apply the second root to form the raised cosine pulses.
j. Connect the AGC output to the input of the RRC filter block. Connect the RRC to the frequency and time sink blocks.
k. Search "QT GUI Eye Sink" and connect the output of the RRC block to this block. The eye sink will allow us to view the eye diagrams (a persistent stacking of our raised cosine pulses) for both quadratures, I and Q. Once you are finished, your workspace should look like the following figure.

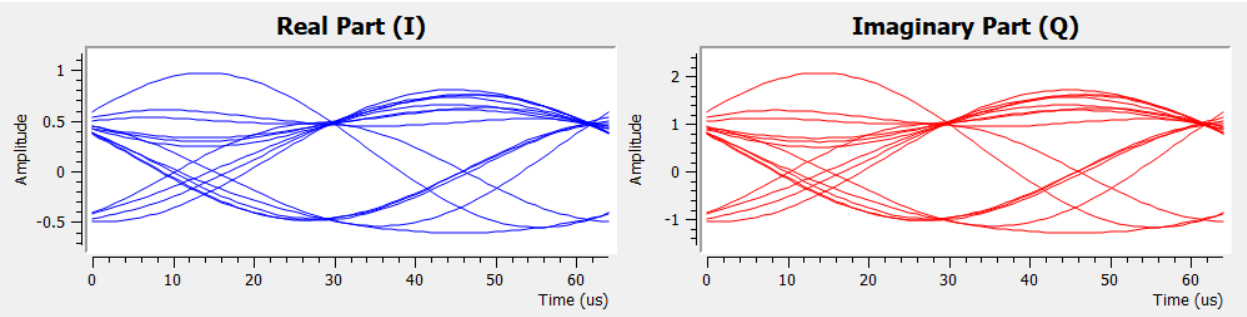

Options
Title: Basics of SDRs and IQ
Output Language: Python
Generate Options: QT GUI
Variable
ID: samp_rate
Value: 1M
Variable
ID: center_freq
Value: 10M
Variable
ID: sps
Value: 32
Variable
ID: excessbw
Value: 500m
Import
Import: log2
Random Uniform Source
Minimum: 0
Maximum: 2
Seed: 0
out
Random Integer Generator- generates random integers bound to 0 and 1.
in
Unpacked to Packed
Bits per Chunk: 1
Endianness: MSB
out
Each integer from the RUS is converted into a byte. This block packs the last bits in each integer byte into a new "packed" byte to be transmitted.
in
set_symbol_table
Chunks to Symbols
Symbol Table: -1, 1
Dimension: 1
out
in
Repeat
Interpolation: 32
out
Constellation Modulator
Constellation: <gnu...C10AF0>
Differential Encoding: No
Samples/Symbol: 32
Excess BW: 500m
out
Constellation Object
ID: constellation
Constellation Type: BPSK
in
Multiply Const
Constant: 400m
out
QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 1M
Autoscale: No
QT GUI Frequency Sink
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 1M
command
in
UHD: USRP Sink
Sync: No Sync
Samp rate (Sps): 1M
Ch0: Center Freq (Hz): 10M
Ch0: Gain Value: 0
Ch0: Antenna: TX/RX
async_msgs
USRP Transmiter
command
UHD: USRP Source
Sync: No Sync
Samp rate (Sps): 1M
Ch0: Center Freq (Hz): 10M
Ch0: AGC: Default
Ch0: Gain Value: 0
Ch0: Antenna: RX2
out
async_msgs
USRP Receiver
in
AGC
Rate: 100u
Reference: 1
Gain: 1
Max Gain: 10
out
in
Root Raised Cosine Filter
Decimation: 1
Gain: 1
Sample Rate: 1M
Symbol Rate: 31.25k
Alpha: 500m
Num Taps: 352
out
in
QT GUI Eye Sink
Number of Points: 1.024k
Samples per Symbol: 32
Sample Rate: 1M
Autoscale: No
Signal Source
Sample Rate: 1M
Waveform: Cosine
Frequency: offset
Amplitude: 1
Offset: 0
Initial Phase (Radians): 0
cmd
out
in0
in1
Multiply
out
in
QT GUI Frequency Sink
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 1M
in
QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 1M
Autoscale: No
QT GUI Range
ID: offset
Default Value: 0
Start: -500k
Stop: 500k
Step: 1

7. Run your file and observe the differences between the received pulses and the received spectrum. Note: you may need to auto scale your plots to see the full ranges.

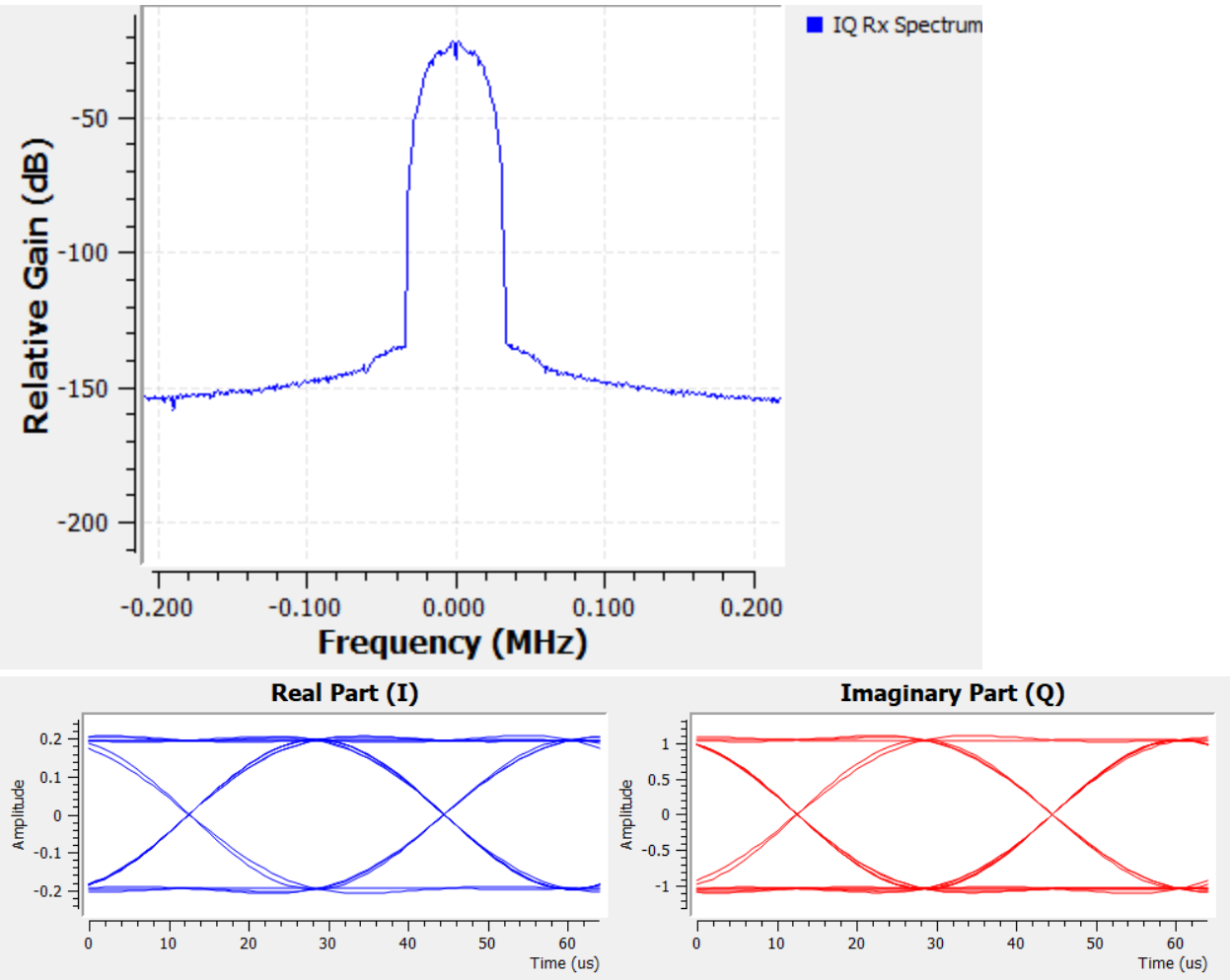
8. Now, stop the simulation and try changing your 'excessbw' to a very large number i.e. 1.0 and a very small number i.e. 0.1. What differences do you see when changing the excess bandwidth? Specifically, are your results consistent with what you expected? How about the eye diagram? What happens to the "Eye" when you decrease the excess bandwidth? Why do you think this is happening? Feel free to take screen captures to use in your lab reports for justification.

#8 Results





Excess Bandwidth = 0.1



Excess Bandwidth = 1

We can see the tradeoffs between ideal pulses and what it costs in the frequency domain. For instance, with a low excess bandwidth, the spectrum is restricted from the higher frequency components to “smooth” out the transitions between symbols, and as a result, we see many different transitions between symbol states (+1,-1) for BPSK. In the case where the excess bandwidth is maximized, more of the higher frequencies are allowed into the pulses and hence a more apparent smoothing effect. This “smoothing” can lead to eye opening and better estimates of symbol amplitudes.