Practical Tutorial for SDR using GNU Radio

Andr Silva[‡], Marco A. C. Gomes[‡] João P. Vilela[*]
[‡]Instituto de Telecomunicações, Department of Electrical and Computer Engineering,
University of Coimbra, Coimbra, Portugal. Email: uc2015228086@student.uc.pt, marco@co.it.pt
[*]CISUC and Department of Informatics Engineering, University of Coimbra, Coimbra, Portugal.
Email: jpvilela@dei.uc.pt

***Abstract*—FALTA O ABSTRACT**

I. Introduction

Everyday, more and more people communicate with each other in different ways for different needs, for instance, communications with phone or internet calls, messaging, socializing in social networks, emergency calls, among several other ways and needs. When people are interacting with a phone or a laptop either by Wi-Fi or Global System for Mobile communication (GSM), in reality, they are transmitting and receiving data which is created by them or another terminal. A radio is a device that can transmit/receive signals in electromagnetic waves with information encoded in it, making all these interactions possible.

Software Defined Radio (SDR) can be defined in many ways. One of them was introduced in a work of the Wireless Innovation Forum [1] in collaboration with the Institute of Electrical and Electronics Engineers (IEEE) where it was defined as "Radio in which some or all of the physical layer functions are software defined" [2]. "Software defined" means, in short, that the mechanisms previously statically placed on hardware are now implemented in software. Thus, SDR is a system for radio communication where several components (like modulators/demodulators, filters, amplifiers, equalizers, among others) are now implemented in software other than hardware, hence, much more configurable and useful for many reasons, like research, personal prototypes among others. One type of these radios is the Universal Software Radio Peripheral (USRP) which will be used for this tutorial.

There are different tools to work with the software component in a SDR device such as:

- *MATLAB & Simulink*: The MATLAB [3] is a paid software developed by MathWorks which combines iterative analysis with a programming language based on matrix and arrays calculus, being a quick learning and fast running language, thus being widely used by engineers. The Simulink [4] is a plugin that makes possible designing and simulating the system quickly and intuitively by enabling the creation of the design by blocks and a set of connections between these blocks.
- *LabView*: The LabView [5] is another paid software created by National Instruments which is owner of Ettus Research who dominates the SDR market. Although it is more directed to automation, it is flexible and can be used for this purpose. It has the advantage to allow directly program on Field-Programmable Gate Array (FPGA), thus, being faster at signal processing level.
- *GNU Radio*: The GNU Radio (GR) [6] is a free and open-source software to work with signal processing, where it is possible to simulate all steps by creating a FlowGraph (FG) by positioning blocks and connecting them. There exists some blocks to do some pre-defined algorithms, although if it is not yet implemented then it is also possible to create new blocks using `Python` or `C++` (preferable `C++` for being the fastest language) and use `XML` to link the source code to the graphical interface. This is the main problem for some users because it requires a lot more effort to code comparing with other tools.
  There are two main problems about this software, namely the lack of documentation to help the user starting and creating his own FG (information about the functioning of these blocks is dispersed). Also, the continuously improving and changing of the stock blocks causes some blocks to be deprecated and hence an outdated FG, needing a constant update by the FG's author of the used blocks.

In this tutorial we will show how a transmission of any file type (text, image, and even video streaming) it is created and implemented both in simulating mode and in real-world prototype by using using USRP B210 [7] with VERT 2450 [8] antennas attached (both from Ettus Research). Furthermore, it will be used the GR as the software component and we address the GR problems above described through the creation of the FG step-by-step explaining them in a detailed way (which includes encoding, modulation, transmission and recover the original signal). Besides that I will show the faced problems and the respective solutions in a easily understandable manner (e.g. creating our binary packets and even creating our own blocks (by crating the Out-Of-Tree (OOT) Module)).

II. Getting Started

Before starting, we need to install all dependencies needed for GR, this dependencies can be found here: [9].

Then you need to decide if you install an external Vector-Optimized Library of Kernels (VOLK) or the internal one, if choose for install the internal then jump this step, however I recommend install the external one to avoid some troubles. The VOLK can found here: [10].

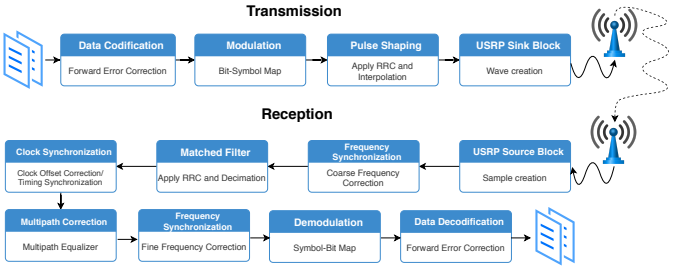


Fig. 1. General concepts of a wireless transceiver.

To deploy the project on GR using the SDR is mandatory to have a proper driver taking into account the hardware that is being used, for instance, to the USRP is required the USRP Hardware Driver (UHD) Driver [11], so you need to install it: [12].

Now we are ready to install the GR itself which can be found how it is done here: [13]

If you have a Ubuntu and you still didn't have installed GNU Radio then you can find a more detailed installation on Appendix A tested both in Ubuntu 16.02 and 18.02.

III. Overview

The final goal of this tutorial is enabling a transmission of some data (text, images, even video files) between two USRP terminals. A communication always assumes the existence of two sides: the Transmitter (TX) that manages to adapt the information to be sent to the characteristics of the transmission channel, making the transmission possible; and one Receiver (RX) where the sent data is received (wireless or not) in analog/digital information and recovers the original information from there. The main block components of these sides for a SDR implementation are depicted in the Fig. 1.

In the TX side the goal is pick the desired data and make the transmission possible, thus requiring four main operations: First, is added redundancy using the error correction codes that enables detection and/or correction of error that may occur during transmission making it more resilient to noise. Low-Density Parity-Check Code (LDPC) or Convolutional Code (CC) are examples of algorithms for this purpose. Next, the data is modulated where the binary information is mapped (using some mapping rule, usually called as constellation) into some characteristics (amplitude, phase or frequency) of the transmitting wave (called carrier) adapted to the transmission over the medium. This results in a large bandwidth which needs to be narrowed by using a pulse shaping filter. This raises some problems which results in the application of the two Root Raised Cosine (RRC) filters, where one is applied in the TX side and another upon the reception. Finally, the digital information is ready to be converted on analog waveform and up-converted to high frequency to be transmitted over the antenna. This is done by using the "USRP Sink" block available in GR.

At this moment we have our data in the channel (in particular, the air) thus some effects are induced in the signal namely *noise*, *timing offset* (due to SDR's clock differences), *frequency offset* (since they are two physically separated devices) and even the *multipath* problem (multiple paths to receptor).

The RX side is substantially more complicated than the TX side because it is critical to undo these distortions. First step is to convert the analog signal that came from the antenna to digital signal with the "USRP Source" block in GR. Then, it is necessary to take care aspects related to synchronization, firstly by grossly correcting the frequency offset, next applying a matched pulse shaping filter (RRC filter), then correcting the clock difference of the SDR's (in a grossly way), followed by correcting the multipath problem and finally by performing fine frequency correction. Finally, the signal is ready for being successfully demodulated and decoded for recovering the transmitted bits.

IV. GNU Radio Internal Working

Before going to the tutorial itself it is necessary to understand the GR internal's working. The data is treated as a stream which has a sample rate defined in a "Source" block. The data flows through a connection and is feed into one block that processes the data. Then the data is output which flows to another block using connections, and so on until get to a "Sink" block.

When we do data streaming we are implicitly using buffers to hold the data between blocks. This buffers are of finite size, and as so before the block does anything, it checks the amount of available data in input buffer and space to write in the output buffer. If there is not enough data or space available in such buffers the block does not do anything. This is called "Back-pressure" because it is pressured back the data that is incoming, slowing down the flow. This is how GR controls the data flow, giving data to the block when it is necessary and holding it when block is busy working.

There are source/sink blocks that produces/consumes data at a given fixed sample rate, as for example the "Audio" and "USRP" blocks. These blocks are called "clock" blocks, and a FG has typically only one source or one sink block, and not both because thus usually leads to a problem called as "2 Clock Problem". This problem happens when the two blocks have different fixed sample rates leading an asynchronous clock sources which causes "Underruns"/"Overflows" depending on difference between the production and consumption rates. When none of this "clock" blocks is used then it is required to use the "Throttle" block to act as a clock. Without it, the FG will run as fast as the CPU can process resulting in non-responsive program and other problems like hiccups. This means that while it is in simulating mode it is necessary to use the "Throttle" block to control the flow, although when we go to a real transmission this block is no longer required.

V. Guided Tutorial - Simulation

A. *Modulation - Tutorial_1.grc*

In order to transmit data between SDR's it is necessary modulate the signal. The modulation is the act of convert/map bits into an electromagnetic wave (called carrier) by varying one

or more of its properties in a manner adapted to the medium of transmission (wireless, optic, copper, among others). One popular form of modulation is called as Phase-Shift Keying (PSK), where the data is encoded by varying the phase of the wave keeping the amplitude and frequency unaltered. This type of modulation uses both the sine and the cosine wave together resulting in two orthogonal carriers making possible the representation of the tuple with their phase as a symbol in a complex plane resulting in a constellation plot. The number of bits carried in each symbol depends of the modulation order M, i.e. the number of symbols of the constellation with $\log_2 M$ bits for each symbol. For example, with an order of $M = 2$, this is, Binary Phase-Shift Keying (BPSK) is encoded one bit for each phase, thus a symbol carries 1 bit and the constellation plot has 2 points (only 2 symbols possible: the symbol 0 and the symbol 1). Going for Quadrature Phase-Shift Keying (QPSK) ($M = 4$), the phase changes in four possible ways, resulting in four constellation points, each carrying 2 bits. In the same way, 8PSK will result in 8 constellation points carrying 3 bits length per symbol, and 16PSK has 16 constellation points with 4 bits length per symbol.

Going to using GR itself, first we need to understand the working of some blocks. We can convert bytes in complex numbers using the "Chunks to Symbols" block. This block has a parameter called "Symbol Table" where is defined the mapping between a byte and the constellation point of the symbol. However, it is necessary first to do two things:

First, it is required unpack the byte, this means split the 8 significant bits per byte into M significant bits per byte. The M value is how many bits is necessary for the symbol (taking into consideration the constellation) as previously mentioned. (In short: BPSK $M = 1$, QPSK $M = 2$, 8PSK $M = 3$, 16PSK $M = 4$). This is possible using the "Repack Bits" block where is defined 3 important parameters: (1) the "Bits per Input Byte" where we define how many bits we pick in an incoming byte, (2) the "Bits per Output Byte" where we define how many significant bits will have the output byte (the remaining bits will be 0's), (3) and finally the parameter "Endianness" to determinate if we want write in Most Significant Byte (MSB) or Least Significant Byte (LSB) (always read in LSB). For example if we input the byte "01000010" with 8:2 in MSB we got 4 bytes: "00000**01**", "00000**00**", "00000**00**", "00000**10**". However, if we change to LSB (also with 8:2) we got 4 reversed bytes: "00000**10**", "00000**00**", "00000**00**", "00000**01**". Note that this block leads to an interpolation of "Bits per Output Byte"/"Bits per Input Byte", in this case $\frac{8}{2} = 4$, this is, for each 1 bytes outputs 4 bytes.

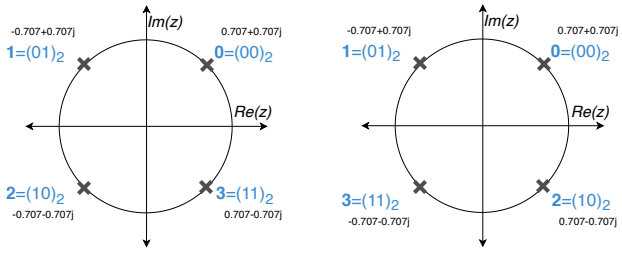
Second, we need to map the unpacked bits into the desired symbols using the "Map" block that basically does: `output[i] = map[input[i]]`.

Particularly, in a QPSK modulation: First, the 8 bits per byte is sliced into 2 significant bits per byte using the "Repack bits" block with 8 "Bits per Input Byte" and 2 "Bits per output Byte" with MSB as endianness. Currently, the byte has the first 6 bits with 0's and the next 2 bits with 2 bits of data. Then we create the "Constellation Object"/"Constellation Rect. Object" where is defined the "Symbol Map", which is the map between the incoming bits and the symbol (in binary form), and the "Constellation Points", which is the map between the symbol (in binary) and the constellations point (in a complex number form). The we can use the combination of "Map" and "Chunks to Symbols" blocks to modulate the signal.

Since it is a simulation, the last block is "Throttle" and it is explained why it is needed in Section IV. To finalize the TX side we send the data to a "Virtual Sink" block, that redirects the data to the "Virtual Source" block with the same ID, the only purpose of this block is making the FG more clear, elegant and readable.

In the RX side we can use one of two decoders: an hard decoder or a soft decoder. Both receive the complex values of the signal, however, the hard decoder maps into binary bits, while a soft decoder outputs the probability of some bit being 0 or 1 (decision based on the LUT table in constellation object). The probability can be useful when using "Forward error correction decoder" block or it is possible to convert then in binary bits by using the "Binary Slicer" block.

Taking the hard decoder into consideration, the "Constellation Decoder" block maps the incoming complex numbers into symbols (using the parameters defined in the "Constellation Object"/"Constellation Rect. Object" object), and then we use "Map" block to map the symbols into bits of data. Finally, we pick the 2 significant bits and pack them in a byte, exactly the inverse operation of before.



(a) Natural Mapping. (b) Gray Code Mapping.

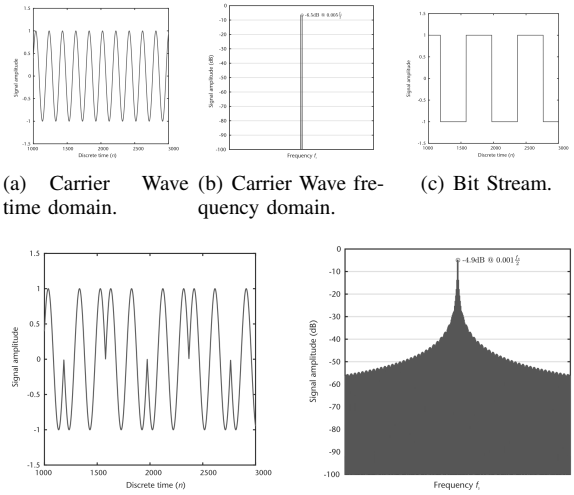
Fig. 2. Difference between Natural Mapping 2(a) and Gray Code Mapping 2(b) on a QPSK constellation.

Furthermore, regarding the mapping between symbol and constellation points, we can do it with "Natural Mapping" represented in Fig. 2(a) or use one concept called "Gray Code Mapping" represented in Fig. 2(b). The gray code has the advantage that neighbor symbols only differ one bit, reducing the amount of wrong bits if occur one error in the transmission. This happens because when occur one error, in the demodulator, instead mapping the right constellation point maps one of the neighbors with high probability, hence, resulting in maximum, only one-bit error instead of two or more bits (if using the natural mapping). Both are implemented on the given FG.

One last tricky thing, if you desire work with 8PSK modulation, you will need to use the "Constellation Object" and manually insert all the parameters (symbol map and

constellation points), otherwise it will not work correctly due to GR's bugs. I created the object which does correctly the 8PSK modulation.

B. Modulation - Tutorial_2.grc



(a) Carrier Wave time domain. (b) Carrier Wave frequency domain. (c) Bit Stream.

(d) Information carrier wave in the time domain. (e) Information carrier wave in the frequency domain.

Fig. 3. At a simple carrier wave 3(a) with a narrow frequency band 3(b) is multiplied the bit stream 3(c) resulting in a ready to transmit carrier wave 3(d) where has an infinite frequency band 3(e). All images are from [14].

The step that follows modulation is pulse shaping. To understand why this is needed [14] let's consider the case of a BPSK transmission. Take a look to the time domain plot of the carrier wave without the added bit stream in Fig. 3(a) where a perfect sinusoidal wave is shown and then taking a look to the frequency domain Fig. 3(b) we observe that a single strike appears at the frequency of the carrier. However, the spectrum of the modulated signal shown in Fig. 3(e) occupies a large bandwidth since the modulated signal results from the multiplication of the carrier by the bit stream shown in Fig. 3(c) resulting Fig. 3(d) (Note that polar encoding is used with bits "0" or "1" being mapped to voltages $\pm 1V$). In fact the spectrum of the modulated signal is theoretically infinite due to the rectangular shape of Fig. 3(c). This large bandwidth brings several problems, such as:

- Costs of operation. Spectrum is a scarce resource and there is the need to buy it to a regulation entity;
- Difficult to transmit because of the large bandwidth and the physical channels (e.g. wireless) being selectively in the frequency;
- Problems of inter channel interference if transmission bandwidth exceeds the assigned one causing distortion in neighbour channels;
- More noise upon reception, since thermal noise, usually moduled as additive white Gaussian noise is proportional to the channel bandwidth.

Hopefully, according to the Nyquist criteria, the minimum bandwidth required to transmit at a symbol rate of R_s (Baud) is $\frac{R_s}{2}[Hz]$. The function of the pulse shaping is to limit the bandwidth of the channel toward the minimum Nyquist Bandwidth. The problem with narrowing the bandwidth is that the symbols become wide in the time domain and their tail will interfere with the beginning of the neighbouring symbol inducing Inter-Symbol Interference (ISI). Therefore the goal is narrow the bandwidth while keeping the lowest ISI possible. Hopefully, Nyquist has also defined a set of requirements on pulse shaping filters to avoid ISI, known as Nyquist filters. A commonly used pulse shaping filter is Raised Cosine (RC) filter with frequency response [14]:

$$H_{RC}(f) = \begin{cases} 1, & \text{if } |f| \leq \frac{1-\beta}{2T_s} \\ \frac{1}{2}[1 + \cos(\frac{\pi T_s}{\beta}[|f| - \frac{1-\beta}{2T_s}])], & \text{if } \frac{1-\beta}{2T_s} < |f| \leq \frac{1+\beta}{2T_s} \\ 0, & \text{if } otherwise \end{cases}$$

Where $Ts = \frac{1}{R_s}$ is the symbol period, and β is the roll-off factor that measures the excess of bandwidth with respect to the minimum Nyquist bandwidth $\frac{R_s}{2}$, thus, $0 \leq \beta \leq 1$ and $\beta \geq \frac{1}{2T_s}$. When we set more roll-off (e.g 1), it is easier for the receptor to decode the signal but we are narrowing less the bandwidth (which is against our objective), on the other hand, smaller roll-off will result in a narrower bandwidth, however its side lobes increases so attenuation in stop band is reduced [15] being more difficult to receiver to decode. I provide the FG called as `roll_off.grc` inside of `Extra` folder, where you can analyse the plot of 5 different frequencies waves with different roll-off factors. In this tutorial we will use the 0.22 value to this parameter for being the middle ground in terms of trade-off, hence, the most used for communication systems.

In order to maximize the Signal Noise Ratio (SNR) upon reception, this RC filter is usually divided in two square RRC ones, with frequency response $H_{RRC}(f) = \sqrt{H_{RC}(f)}$, where one is used upon transmission and other upon reception filtering out the receiving noise.

By applying the filter upon reception we are doing the matched filter operation, which is obtained by correlating a known delayed signal (called as template, this the RRC filter on TX side) with the received signal (mixed with noise) to detect the presence of that template in that signal. The digital implementation of this filters is usually done by windowing the Finite Impulse Filter (FIR) where more window means more number of symbols that will be used to calculate the energy of the wave, thus, better approximation of the analog signal filter. Considering a minimum duration of 11 symbols (center of the filter, plus the five close neighbors for each side).

Another thing included in modulation is the interpolation which is the act of increasing the Samples Per Symbol (SPS) in order to increase resolution of the signal. Basically we can set an high number of SPS leading to less probability of errors, however, we are inducting more overhead to our transmission (we are transmitting more information). The interpolation construct new points in the wave between the existing ones, thereafter when the signal will be reconstructed will have more points, thus, will be a lot more precise. To find this parameter we have 2 criteria: First, leaving this values small as possible

taking into account that is mandatory to be 2 or higher. Second, we can use this value to help us match the desired bit rate taking into account the sample rate of the hardware we are using (currently none but we will use USRP's). We will use $SPS = 4$ for this parameter due to its performance being advantageous compared to the introduced overhead.

Moving to RRC filter in GR, it uses taps to apply against the signal in order to shaping it, there is two ways to implement. (1) Implement as a normal filter using the "Interpolating FIR Filter" block on TX side and the "Decimating FIR Filter" block on RX side setting the desired window and SPS value to calculate the number of taps (resulting only in $11 * 4 = 44$ taps). This works perfectly for now because we do no yet have introduced channel problems, after these problems this number of thaps is not enough.

(2) The other way is using the "Polyphase Arbitary Resampler" block in the TX side and the "Polyphase Clock Sync" block on the RX. Here, the number of taps are calculated based in the number of filters, window value and the SPS value. Theoretically, we can define an infinite number of taps, hence we can infinite attenuate the stop band. However, in reality, we need to define the limit according with processing power we have. We will use this blocks to the remaining tutorial for having good results when inducting real-world problems. This block instead of apply only one filter, it applies how many filters we desire to work with, creating more filters where each one has a different phase, resulting in a big filterbank. First we select the number of filters in the "Number of Filters" parameter. We set 32 filters `(N_Filters)` because it is a default and recommended value [16]. However, it is possible to increase this value if we want more precision (and less ISI) in the recovery, e.g: 64 filters, but will be a lot more computationally heavy. Taking this into account now we need to create the "RRC Filter Taps" object to be input in "Taps" parameter. In this object is defined all the specifications of the RRC itself, namely the "Gain", the "Sample Rate" to 4 (because we will use 4 to our SPS), the "Symbol Rate" to 1 (we will send 1 symbol for each 4 samples), the "Excess bw" as 0.22 (which is the chosen roll-off as discussed above), and finally the "Num taps" as `Window * SPS * N_Filters`, resulting in $11*4*32 = 1408$. To finish this block, the "Sample Delay" parameter is only used to right place the tags if they are being used.

Like we have mentioned, in RX it is required to apply the matched filter. We can do that by using the "Polyphase Clock Sync" block which not only applies the filter we want, but also decimate the signal to a desired number of output SPS and even do clock synchronization like it will be addressed ahead. Previously we have interpolated the signal by 4, thus, we set 4 in the "Samples/Symbol" parameter and 1 in the "Output SPS" parameter which results in the signal without any interpolation. Besides that, as it was used 32 filters it is required to set the 32 value in the "Filter Size" parameter and in the "Gain" parameter (on "RRC Filter Taps" object). The "Sample Rate" is set `SPS * N_Filters)`, which is the incoming samples rate, the "Symbol Rate" to 1 (only one symbol at a time), the roll-of factor to 0.22 (as discussed above), and finally, the "Number of taps" results in `Window * SPS * N_Filters` which is the same as the TX side. (This means that for each sample we got 32 filters, using the present symbol, 5 symbols backward and 5 more forward). Note that we can not get rid of all the ISI because of the noise added by the channel and the precision (this is the N_Filters of the filterbank).

C. Modulation - Tutorial_3.grc

In order to achieve more approximation to reality, it is possible to implement a "Channel Model" block where is simulated different properties of a channel that we will need to take care at the receptor:

- Noise - We can add Additive White Gaussian Noise (AWGN) in parameter "Noise Voltage" where we set a level as a voltage;
- Frequency Offset - The "Frequency_offset" parameter where 0 is no offset and 0.25 would be a digital modem (1/4 of the symbol rate);
- Timing Offset - The "Epsilon" parameter allows emulating different sample clocks between the transmitter e the receiver and 1.0 is no offset added.
- Multipath - We can emulate multipath delay by adding taps of a FIR filter in the "Taps" parameter.

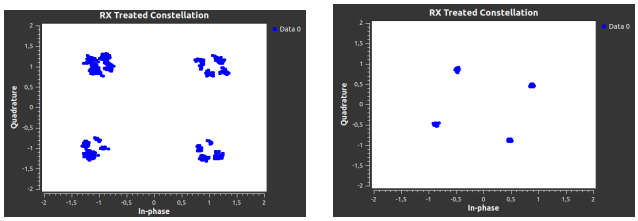
Remember that the analog signal is got from the antenna and sampled into digital one with the Analog to Digital Converter (ADC) inside of the USRP. This offset arises from sampling not happening in the ideal instant, creating timing offsets, resulting different samples than the ideal ones. This impacts ultimately on more disperse constellation points. Also, we have added in `Extra` folder an example named `data_timing_offset.grc` where inputs repeated data and simulates a clock offset on the "channel model" block. It is possible to visualize that the data is sampled in different instants, thus having associated the timing offset.

Fortunately, we already take care of this problem (along with noise) by using the "Polyphase Clock Sync" block. This block uses two polyphase filterbanks, one contains the matched filter with different phases and another one contains the derivatives of the first filterbank. Note that the first filterbank is the RRC "template" previously described and the peaks are known, hence, we have the knowledge of the ideal sampling points. This is conjugated with the property of the derivative of these peaks being zero, then the main objective is to align the output signal of this block to be sampled at exactly the peak of the first filterbank. Besides that, the region around the peak is relatively linear. All these facts are used to generate the error signal which is value of how far away a sample is from the zero in the derivative signal, this is used to recover the signal.

For this it is necessary a second order loop similar with Phase-locked loop (PLL), where it is defined two variables, the number of filters on the filterbank (the "Filter Size" parameter which is 32 in our case), and how far we want to traverse the path filters to keep the receiver locked (forward or backward)

in the ”Maximum Rate Deviation” parameter (positive and negative from 0). I have set 1.5 because it is the default value, however, it is possible to increase this value to set the loop going further for looking the ideal sample spot, thus, increasing the amount of power needed for this operation. Besides that, a parameter we can also adjust is the ”Loop Bandwidth” that is used by the second order loop, I have set to $6.28/100$ as suggested in [17].

There is also the multipath problem which is having different paths to the receptor. In a wireless communication the signal reflects on environment objects, for instance, walls, and the signal arrives to the receptor a little after of the direct path signal, thereafter the receptor receives the same signal various times with slight timing differences. This, obviously, affects the transmission causing the dispersion of the points of the constellation increasing the probability of receiving a wrong symbol.



(a) Multipath effect in a constellation.

(b) Multipath correction with a 15 taps equalizer.

Fig. 4. Multipath effect in a constellation plot 4(a) and its correction after being applied an equalizer with 15 taps 4(b).

In order to easily understand this concept we have provided the one FG called as `multipath_problem.grc` inside of `Extra` folder, where there is two paths on the receptor, one with (upper path) the ”CMA Equalizer” block and another without it. Also, it has two variables about the taps to be set on the ”Channel Model” where one is with almost no taps and another emulates the multipath problem. Running the FG with the taps variable it is possible to visualize the constellation plot of the path with the equalizer block with more aggregated points, thus, better possibilities of correctly demodulate the signal.

Going to the main FG, if you bypass ”CMA Equalizer” block (not that is also required to set the ”Output SPS” parameter to 1 in the ”Polyphase Clock Sync” block) and enable the first taps variable (which is almost without taps) and then you can play with the ”Time Offset” and ”Noise Amp” tweaks in order to visualize the plot labeled as ”Rx Treated constellation” a nice and clear constellation (although a little rotated). Now enabling the second taps variable which simulates the multipath property and run again the FG, you can now see the effect of these taps which causes a dispersion on the constellation points as visualized on Fig. 4(a).

In order to tackle this problem it will be used an equalizer called ”CMA Equalizer”, whose performance usually depends on the number of taps, where more taps mean better final results, however more overhead to the algorithm. Thus, this number must be small but enough for correcting the channel which is experimentally found. It is required the ”Samples per Symbol” of the input signal to be set as 2, thus, in the ”Output SPS” parameter of ”Polyphase Clock Sync” block is also set to 2. Then if you compare the constellation plot after being applied the equalizer it is possible to observe a constellation significantly more converged and cleaner like is visualized in Fig. 4(b).

Note that the constellation is rotating a little since the ”CMA Equalizer” all it cares about is converging in a unit circle without any knowledge of the constellation, so when it locks, locks in a given phase. Besides that if you tweak the ”Freq Offset” you can see that the constellation become a circle, that is because we are not yet treating the frequency offset, it will be taken care next.

D. Modulation - Tutorial_4.grc

Due to the transmitter and the receiver being two distinct and spatially separated devices the transmission over the channel induce a frequency or phase offset in the transmitted signal. In literature the carrier recovery is defined as carrier phase recover or carrier frequency recover, and has the same goal, which is attain a stable constellation on the output of the synchronizer, so the demodulator can correctly decode the symbols. The frequency and phase are related, so recover the frequency it is indeed equivalent to recover the phase, because the angular frequency ω (equivalent of $2\pi f$) is only a measure of a changing phase σ over time: $\omega = \frac{d\sigma}{dt} = 2\pi f$ [14].

What it is desired in the end is the constellation points well positioned and aggregated, thus, solving the rotation problem previously mentioned which requires estimation and compensation of the frequency offset. Since this deviation can be large, this procedure is usually carried in two-stages frequency recover [14] namely coarse and fine frequency recover.

We will start by treating the second one. It is possible to track the offset by using a second order loop when the signal is close of the correct frequency, if it is not close enough then the constellation will keep spinning. The Costas Loop is a method that works directly over signal [18] which relies on a feedback concepts. This method has the ability to find the right phase and self-correct in order to keep the carrier correctly recovered being used a loop in order to achieve that. Because of this operation mode it has the only disadvantage of needing some prior time to settle the loop, although after it is settled, it feeds itself to keep track the correct frequency. This said, we will use the ”Costas Loop” block where we have to set the mainly two parameters: ”Loop bandwidth” and the ”Order”. The first one is relative to the second order loop (I will use $6.28/100.0$ which is recommended, however, this value adapts to the signal in runtime). The second is the order of the used constellation, thus, 4 in our example. Now if you disable the ”FFL Band-Edge” block and run the FG you will note that the constellation is no more rotating and if you tweak the Frequency Offset note that it is not getting a circle anymore,

instead that, the constellation keeps without rotation, clear and converged.

As I have said, the fine frequency correction needs to be close to the real frequency, otherwise it will no correct. If you continuously keep increasing the ”Frequency Offset” tweak (without the ”FLL Band-Edge” block) you will see that at certain point (close to 0.034) the constellations starts to be a lot noisier. To correct that it is required do the first stage of frequency recover which is ”Coarse frequency recover” by using the ”FFl Band-Edge” block. The Frequency-locked loop (FLL) technique derives of band-edge filter [19]. A band-edge filter covers the upper and lower bandwidths of a modulated signal taking into account the range of the roll-off factors and the interpolation already discussed determining the frequency placement of the band-edges. In the FLL itself, it filters the upper and lower band-edges and calculates the error resulting in an error directly proportional to the carrier frequency, thus, it is possible recover in a grossly form the frequency offset by using a second order loop. In the ”FLL Band-Edge” block on the FG first we set the ”Samples Per Symbol” to 4 and the roll-off factor to 0.22. We also need to set the ”Loop Bandwidth” parameter value, for this first set a temporary big filter size (e.g. 60), then start by using the recommended value which is $6.28/100$. However, the constellation is no aggregated enough (it looks more like an ellipse than a circle), thus, we set change the divisor component until we found the correct one (the one that results in aggregated constellation points). We set to $6.28/200$. Finally we need to set definitely the number of filter taps that we will generate to apply against the signal, to find the best value we need to start at some low value (e.g. 5) which will result in a circle, and keep increasing until we get a clear and aggregated constellation points. We have set to 20 this value.

Finally, now you can tweak the ”Frequency Offset” value beyond what ”Costas loop” alone was capable of handle and confirm a clean, aggregated and steady constellation as it was the objective.

E. Packet communications - Tutorial_5.grc

Now we have successfully modulated and transmitted the signal, but taking a look in the output file, it will only appear junk. This happens because it is not writing the bits in the supposed byte. For better understanding taking into consideration the byte ”01010101” is sent and for some motive receives the first 4 bits at 0’s followed by the sent byte, then it results in two bytes: ”0000**0101**”, ”**0101**XXXX”. Because of this, all of the next sent bits are not wrote in the aligned byte, thereafter only junk is written on the file. This problem is a consequence of the ”Polyphase Clock Sync”/”Costas Loop” blocks need some prior time to adapt, meanwhile outputs junk. It is required a method to receive all the bytes aligned. The solution is packetize the data and send the packet, consequently, it is possible to detect an incoming packet and extract the data correctly. Note that we will lose the first packet but the next ones will be received correctly and with the bits aligned.

For this purpose it will be used the ”Correlate Access Code - Tag Stream” block, thus, lets analyze how it works. This block expects unpacked data (only 1 significant bit per byte) and scans this input in order to find a sequence called ”Access Code” with 64 bits. In this scan it is possible to have X different bits between the input and the sequence, thus, X is the ”Threshold” parameter to allow X different bits. After find this sequence the next 16 bits establishes the payload length (that will output) that will be extracted, where is repeated twice (32 bits in total). Resulting in a header with 64 bits of access code, 16 bits of payload length and another 16 bits again of payload length. It is important to note that the payload length is set in bytes. Now that we have the shape of the header we need to create it.

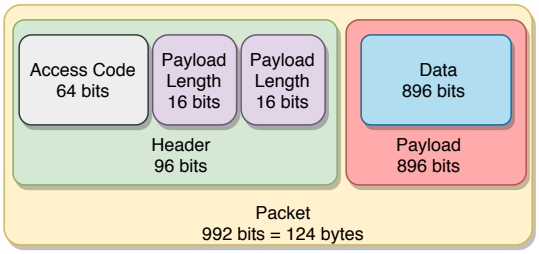


Fig. 5. Packet content and its length representation.

Assuming that we want transmit packets with 112 bytes (896 bits) of payload size, then we need to create the header to this particular payload. To create the header we will use the ”Vector Source” block with the ”Repeat” parameter set to ”Yes”, and the vector will be 64 bits of the access code (we will use the default which can be accessed here: ”digital.packet_utils.default_access_code”), followed by $(0000000001110000)_2 = (112)_{10}$ and then $(0000000001110000)_2 = (112)_{10}$ again. With the header ready we need to split the input in 1 bit per byte using ”Repack Bits” block previously mentioned and then concatenate the 96 bits of the header with 896 bits of the payload using ”Stream Mux” block. All the packet transformation including their respective length (in bits) can be visualized on Fig. 5. Finally, we set the packet to be modulated by repacking again to 2 bits per byte (since we are using QPSK). Note that if we will not use any Forward Error Correction (FEC) code than it is possible to optimize this process by using 2 bits per byte creating the packet with this in mind. However, I will use FEC ahead which will requires the usage of unpacked bytes.

At the RX side we just need to unpack the incoming byte (2 bits/byte into 1 bit/byte) to use the ”Correlate Access Code - Tag Stream” block where is extracted the payload of the packet. Finally pack all bit together (to 8 bits/byte) it is possible to analyse the aligned bytes of the content of the output file.

F. Out-Of-Tree Modules - Tutorial_6.grc

As I have mentioned, now we are getting the bits aligned, however, in return the first packet is lost. This can be solved by transmitting a vector of bits before the transmission of the

real data. There is just a big problem: the GR does not has that implemented but allow us to extend the GR stock blocks by enabling us to create our own (using the OOT Modules funcitonality). This blocks can be created either using `Python` or in `C++` (preferable `C++` for being the fastest language) and use `XML` to link the source code to the graphical interface. This is the main problem for some users because it requires a lot more effort to code comparing with other tools. I will teach how to create this type of blocks in GR.

First, in the folder where you want to keep the OOT blocks run the following commands:

```
$gr_modtool newmod insert_vec_cpp
$cd gr-insert_vec_cpp
$gr_modtool add new_vec
```

Now we need to choose the type of block we want to create, such as:

- Source - With this blocks you can produce output items.
- Sink - With this blocks you can consume input items.
- General -This block is a general version, where you can define the rate of consumption/production.
- Interpolator/Decimator - Use this type when you want a fixed multiple rate of consumption/production.
- Hier - This is called "Hierarchical blocks" where you can aggregate multiple existing blocks in one block. You can also do this in a graphical way on FG by setting the parameter "Generate Options" as a "Hier Block" located in Options of the FG.
- No Block - You can also create FG without any graphical blocks, only code.

For our purpose, I have selected **general**, and then we need to select the language (`Python` or in `C++`), I have selected **cpp**. Then you need to input the arguments that your block will take, as I want only a vector of bytes I set: **const std::vector<unsigned char>&vec**. Finally, you select if you want Quality Assurance (QA) code which is a template where you can do unit testing on your code (I selected no).

Here it is created three important files:

```
File 'lib/new_vec_impl.h'
File 'lib/new_vec_impl.cc'
File 'grc/insert_vec_cpp_new_vec.xml'
```

The first one is the header file of the source code file. The second is the source code where we will write the code and the last one is a `XML` file where we define some aspects on the graphical interface and link the source code to the GR itself.

Lets carefully analyse the source file, in the constructor we have:

```
gr::io_signature::make(<+MIN_IN+>, <+
    MAX_IN+>, sizeof(<+ITYPE+>)),
gr::io_signature::make(<+MIN_OUT+>, <+
    MAX_OUT+>, sizeof(<+OTYPE+>)))
```

The tool adds <+ and +> in places that you need to replace for values that you need. The first line is where you select how many input connections you need, namely the minimum, the maximum, and the size of data type that will be on that connections. In our case we want to get in only one connection (so min=1 and max=1) and treat the data as a byte (to set an `unsigned char`). The second line is where you select how many output connections you need, namely the minimum, the maximum, and the size of the data type that will be in each connection, exactly as before.

The code itself will be written in the "general_work" function, however, to use the variables there we need to firstly initialize them. Thus, the first thing to add is the vector that will be used to store the data that will be concatenated to the incoming information. Second, we will use a flag to know if all the vector was sent (initialize it at 0) and finally an index to track how many bits of the vector we already sent. This results in changes both on source file and on the header file. In the source file:

```
gr::io_signature::make(1, 1, sizeof(
    unsigned char)),
gr::io_signature::make(1, 1, sizeof(
    unsigned char))),
d_data(vec),
flag(0),
track_oo(0)
```

In header file, these variables are declared:

```
std::vector<unsigned char> d_data;
int flag;
int track_oo;
```

Making the respective getters and setters (yet in the header file):

```
void set_data(const std::vector<unsigned
    char> &vec){
     d_data=vec;}
int get_flag(){
     return flag;}
void set_flag(){
     flag=1;}
void set_track(int a){
     track_oo=a;}
int get_track(){
     return track_oo;}
```

Returning to the source file, the function "forecast" is where you define the production/consumption rate you want. We will want 1:1 so you need to make that possible by adding:

```
new_vec_impl::forecast (int noutput_items,
     gr_vector_int &ninput_items_required)
    {
```

```
    unsigned ninputs =
    ninput_items_required.size ();
    for(unsigned i = 0; i < ninputs; i++)
        ninput_items_required[i] =
    noutput_items;
}
```

Moving on for the core work, as previously mentioned, the `general_work()` function is the one that will perform something on the block. The template show us:

```
const <+ITYPE+> *in = (const <+ITYPE+> *)
    input_items[0];
<+OTYPE+> *out = (<+OTYPE+> *)
    output_items[0];
consume_each (noutput_items);
return noutput_items;
```

Again, we need to replace content of the <+ and +> for our type of data: `unsigned char`. The `consume_each()` function is where we set the number of items that we will consumed. The number of items produced is set at front of the return.

Going for the code itself: If all the vector was already inserted, then we just pass through the items doing nothing to them, using the `memcpy()`. If there is any items in the vector to be sent, then we need to send it taking into account that we can only send the maximum size available on the output buffer. In case of there is not enough space on that buffer, we sent what it is possible update the variable to keep track of the last byte sent in order to the next iteration knowing the initial position that we will start sending the vector. Note that here we do not consume any item, we are just producing them. In case there is enough space on the buffer, then the remaining content of the vector is sent and the flag is set to 1 in order to know that we can make the items pass through directly on the block.

So, in `general_work()` we will add this code:

```
int ii=0;
int oo=0;
if(get_flag()==1){ //Vector already
    inserted
    ii=noutput_items;
    oo=noutput_items;
    memcpy(&out[0], &in[0], sizeof(
    unsigned char)*noutput_items);
}
else{ //First time/Vector not fully sent.
    int max_copy = std::min (noutput_items
    , (((int) d_data.size()) - get_track()
    ) ); //Check for space in buffers to
    use the remaining vector (len(vec) -
    used)
    oo=max_copy; //That is what I will
    output (produce: all vector/all buffer
    )
    ii=0; //I do not want consume anything
    memcpy(&out[0],&d_data[get_track()],
    sizeof(unsigned char)*max_copy); //
    Output starting from where I stopped
    the last time (Starting with 0)
    if(max_copy == (((int) d_data.size())
    - get_track()) ){ //If I will use last
     piece of the vector (len(vec)-used)
    then set the flag in order to start
    passing directly through the block.
        set_flag();
    }
    set_track(get_track()+oo); //Increment
     the track to know where to start
    copying the vector the next time (
    Where I was plus what I will produce
    now)
    }
consume_each (ii);
return oo;
```

In last file (`XML`) it is required to set some things in order to link the created code to the GR. The first tricky thing is remove the `&` on the parameter in `<make>` tag. In `<param>` is where is defined the parameters that will be parsed in the GR, so we will replace this piece of the template for this:

```
<param>
    <name>Vector</name>
    <key>vec</key>
    <type>int_vector</type>
</param>
```

Then, in `<sink>` and in `<source>` has a parameter called `<type>` where we set what type of data that is input and output, in this case will be `byte`.

Now that we have all code written we need compile it and link to GR with the following commands:

```
$mkdir build && cd build
$cmake ../ && make
$sudo make install
$sudo ldconfig
```

Finally, we just need to click in ”Reload Block” in the GR’s toolbar and use the block like any other. To create a vector we can use a ”Variable” block and use the Python input `[int(random.random()*X) for i in range(Y)]` where X is the alphabet of the vector (I want between 0 and 3), and Y is the vector’s length (value that needs to be experimentally found until transmit correctly the first packet).

If you prefer, instead doing manually all this code, I let you the all OOT ready to use in `Extra/OOT/Insert Vector` where you can just unzip it and run `/script.sh` inside `build` folder. To uninstall the OOT you need to do it manually by running `sudo make uninstall` on the same folder.

G. Forward Error Correction - CC - Tutorial_7_CC.grc

In a communication system the probability of having errors is high, so, it is necessary a mechanism which enable recovering the wrong bits. FEC is usually employed, where the messages to be sent are coded by introducing redundancy (added extra bits) that allows to detect and/or correct wrong bits upon reception. These check bits are usually linear combinations of several bits of the message. It is this knowledge and relation between them, that enables the message being correctly recovered in RX side, even with some corrupted bits of the codeword. Errors that occur during transmission are typically of two types: a single-bit error where some randomly spaced bit is wrongly received; or a burst error where some amount of continuous bits are wrongly received. The error correction codes are better prepared for the first type, thus, the second being the worst type but common.

The GR offers a module called FEC where exists four encoding blocks: "FEC Encoder", "FEC Extended Encoder", "FEC Tagged Encoder" and "FEC Extended Tagged Encoder". There is only a few differences between them: the first one striclty applies the algorithm and do nothing else (inputs the information data and outputs the codeword), the second one is a wrap (a heir block) of the first one where is added a puncture pattern (common in this type of encoding) and it is properly converted the input/output data into the necessary types. The third one, instead using the "Frame Bits" parameter defined in the encoder object block for knowing how many bits is input into FEC, uses a stream previously tagged and automatically increases the length of that tag. Finally, the last one is similar to the third however, takes care of puncture and some data conversion aspects. It will be used the "FEC Extended Encoder" block for this tutorial which is the most used since it takes care of the data type conversion. In order to use this block it is necessary specify as a parameter an encoder object which specify the algorithm that is used, such as:

- Dummy Encoding - The dummy encoding simply redirects the input to output with no error correction. This simply allow us to use and test the the encoder block.
- Repetition Encoding - This encoding only repeats the same bit how many times we define, thus, has a low performance taking into account that we multiply the amount of bits needed to transmit.
- Convolutional Code - This code generates parity symbols by sliding and polynomial function to our data stream. Here we can use 2 blocks, or the "CC Encoder Definition" object which is a generic implementation where it is prepared to use different polynomials, rate and constraint length (currently, the GR has only implemented the Voyager code), or use the "CCSDC Encoder Definition" object which is an highly optimized implementation of the Voyager code.
- Low-Density-Parity-Check - Here we have two types that we can use, or we use an already created Matrix (using "LDPC Encoder Definition (via Parity Check)") or we use a Generator Matrix (using "LDPC Encoder Definition (via Generator)").

The "FEC Extended Encoder" block expects unpacked bytes, so it is necessary to repack the information previously (8 bits per byte into 1 bit per byte) in order to use it. Now we need to fill the parameters, first we set a puncture pattern of '11' (where there is no puncture), then the "Threading Type" parameter we set to "None" for now (I will be back here). Regrading to the "Encoder Object" we will use the "CC Encoder Definition". This block has a lot of important parameters which is only understandable by knowing the behaviour of the CC algorithm.

The CC [20] takes mainly two parameters, the constraint length K and R generator functions represented as polynomials $P_1, P_2, ..., P_R$. The algorithm works by sliding the data stream over these polynomials generating parity symbols. In more detail, for each bit b of the message M generates R parity bits $(p_1, p_2, ..., p_R)$ by applying the polynomials into the current bit ($b[n]$) and $K-1$ previous bits ($b[n-1], b[n-2], ..., b[n-K-1]$). Then is transmitted the R resulted parity bits and the algorithm goes to the next bit. It is intuitively known that for every 1 bit of the message it will result in R final bits, thus resulting in $\frac{1}{R}$ rate. One of the most used parameters are $K = 7, R = 2, P_1 = 109$ and $P_2 = 79$ which is the Voyager Code from National Aeronautics and Space Administration (NASA), and it is implemented in GR.

Going back to the "CC Encoder Definition", we set 440 in the "Frame Bits" parameter which specify how many bits we desire to input of each work. The "Constraint Length (K)", the "Rate Inverse ($1/R$)", and the "Polynomials" I have set 7,2 and $[109, 79]$ respectively. This values is the Voyager Code which is the only implemented code at this moment.

In the "Streaming Behaviour" parameter we have four options to manage the data stream and the algorithm:

- Streaming: Expects uninterrupted flow of bits to the encoder, where the output is continually encoded.
- Terminated: Used for packet-based, however this mode adds `rate*(K-1)` bits to the output to help flush the decoder.
- Tailbiting: Used for packet-based, however instead adding bits, uses the final bits of the packet when we are decoding.
- Truncated: Here the registers are reset between frames.

As we have mentioned, the goal is packetize the data, so we need a packet-based behavior which only 2 of this modes to attend this requirement, namely the "Terminated" and the "Tailbiting". One important thing to keep in mind is that we want send the encoded packet independent of the last one,

so if one packet is lost, it will not start a chain of corrupted packets, thus, the choice will be "Terminated".

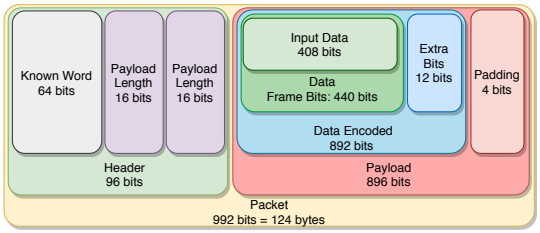


Fig. 6. Packet content and its length representation using the CC.

Taking into consideration the discussed aspects, at the output of the encoder there will be the input multiplied by the rate plus the added bits to flush decoder resulting in: (Input $*$ Rate) $+$ (Rate$*$(K-1) = 440 bits * $2+(2*(7-1))$ = 892 bits. Now that we have the payload encoded we just need to create the proper header exactly as before, but we have a problem here: in the header we need to define the payload in bytes, however, we can not convert 892 bits in bytes ($892/8 = 111.5$ bytes). To solve this we need to add 4 bits resulting in 896 bits = 112 bytes. We can add this using a "Vector Source" Block concatenating the vector with a "Stream Mux" block. All the packet transformation including their respective length (in bits) can be visualized on Fig. 6.

We can now transmit our packet with the payload already encoded. In reception it is necessary to apply the inverse operations, namely, first by using only the first 892 bits of 896 total bits with the "Keep M in N" block, then using the "FEC Extended Decoder" along with "CC Decoder Definition" object. Note that the decoder blocks now uses soft decision bits (floats between -1 and 1) on the input, but as we have an hard demodulator so we got hard decision bits (0's and 1's), thus, we can convert in floats by using the "Map" along with the "Char to Float" blocks.

If you run the FG and look at the Central Process Unit (CPU) processing you can see that only one thread is running slowing down our FG. It is possible to use multithreading by creating multiple encoder variables with same settings defined in "Parallelism" parameter of the encoder object. This value can be 1 or 2, where 1 creates a list of variables and 2 create a list of lists of variables. The length of that list is defined in "Dimension 1" parameter also there. The "FEC Encoder" block must know how to handle with these variables, for this we can set one of the two options: "Capillary" or "Ordinary" in "Threading Type" parameter. In the first type all the data stream is divided in N sub-streams, where each one is passed to one encoder variable to process it in a parallel way. After the processing work, the resulted sub-stream is interleaved back together making again a single stream. The second type is similar, however, it creates a tree where each branch launches 2 more branches, thus N must be a factor of 2 performing better than the former. With this in mind we have set to 1 in "Parallelism" where it is set 4 variables to the encoder and 8 variables to the decoder (because the decoder is heavier) on the "Dimension 1" parameters.

Furthermore, I have added an example (found in: [21]) in `Extra` folder called `ber_curve_gen` where you can analyse the Bit Error Rate (BER) curve in different algorithms. I advise you to run directly from console using python to avoid some problems.

H. Forward Error Correction - LDPC - Tutorial_7_LDPC.grc

A simple coding "Single Parity Check (SPC) code" [22] is a code that adds a single bit to a message, with that bit dependent of the message (usually a linear combination of some bits of the message). LDPC [23] codes are linear codes which is basically the combination of several SPC codes, defined by a parity check matrix H. A (n, k) LDPC code means that for each k bits of the message will result in a n codeword length (with redundancy added) using H as parity-check matrix. The matrix H has $m = (n-k)$ rows with each one corresponding to a parity check equation and has n columns with each one corresponding a bit of the codeword.

The generator matrix of the code represented as G is used to the encoding process, for a u vector which holds k message bits, then the codeword to u can be found using the equation $c = uG$, where G is $k*n$ with k/n ratio. Thus, G for a code with H as parity-check matrix can be found by performing Gauss-Jordan elimination resulting in the end $GH^T = 0$ as described in [22]. Finding G is not practical, thus, generally the LDPC code is restricted to be systematic, where the codeword is composed by the message unchanged appended to the parity bits, which correspond to H matrix with a well defined structure regarding the parity bits [24]. This structure allow the computation of the parity bits directly from the H matrix and the message m, precluding the need for finding G. This matrix can be randomly generated as long as the requirements are meet, or it can be generated using known algorithms like Gallager parity-check matrix and the MacKay Neal's algorithm proposed in [24].

With that in mind, lets go to the existing GR block to encode a stream using the LDPC code, there is three objects defined:

- "LDPC Encoder Definition": use a parity check algorithm using an alist file and specified matrix gap.
- "LDPC Encoder Definition (via Parity Check)": receives a prebuilt H matrix from the "LDPC Parity Check Matrix". The "LDPC Parity Check Matrix" constructs a parity check matrix, H, from a given alist file and matrix gap. This is the same as before but cleaner.
- "LDPC Encoder Definition (via Generator)": receives a prebuilt G matrix from the "LDPC Generator Matrix". In the "LDPC Generator Matrix" constructs a generator matrix, G, from a given alist file.

The LDPC via Parity Check uses a reduced complexity algorithm compared to the LDPC via Generator which requires more heavier operations an each encoding step. This is accomplished by completing a significant amount of the complex matrix manipulation (including inverse, multiplication, and Gaussian elimination operations) during preprocessing. The

disadvantage of this encoder is that it requires a specially formatted matrix. [25]

Note that there is a prebuilt alist files distributed and installed with GR which can be found in:

```
$/usr/local/share/gnuradio/fec/ldpc
```

For the implementation I have choose to use the "LDPC Encoder Definition (via Parity Check)" due to better performance and use a matrix already provided by GR ($H(1100, 442)$ with 24 as Gap). To encode a stream there is need the "FEC Extended Encoder" block and feed it with 442 bits of data to encode it. If we want to transmit a text file we would not want to encode half of byte on two separate packets since one of them can be last. For better explanation, 442 bits is 55.25 bytes, thus, we want only pick 55 bytes and not a quarter of the next byte. A work around of this is simply pick the first 440 bits of data (55 bytes) and add 2 bits to pad in order to complete the necessary amount of bits to use in the LDPC encoding.

The LDPC codes are represented mainly by a Tanner graph which consists in two vertices: n bits of the codeword called as *Bit Nodes*, and m vertices to the parity-check equations called as *Check Nodes*. The decoding process of a LDPC code is usually iterative of type *message-passing algorithms* since the operation can be explained by exchanging information between bit nodes and check nodes by passing of messages in edges of a Tanner graph. Some of these messages are hard-decision (binary) like *bit-flipping decoding*, others are soft-decision (probabilistic) like *belief propagation decoding*, which represents the level of belief about the value of the codeword bits. A bit node sends a message declaring if the received bit is a 0 or a 1, then the check node replies a message to each connected bit node (through edges) declaring what value it should be, taking in account the check node restriction and the information it has received from the other bit nodes connected to it. Basically the check node determines the respective parity-check equation and verify if it is satisfied (by doing the modulo-2 sum of that bit, then the result must be 0). If it is not satisfied than the bit is changed (flipped), otherwise goes no next bit. This process is repeated until it recovers all the message correctly or until ends a previously defined maximum number of iterations.

In the RX side the "FEC Extended Decoder" can use either the "LDPC Decoder Definition" object or the "LDPC Bit Flip Decoder Definition". The first one uses the *belief propagation decoding*, thus, is a soft-decision decoder and assumes a noise (designed for a memoryless AWGN channel) variance entered in "Sigma" parameter. It is necessary to provide the alist file name of the H matrix. The second one is a hard decision decoding scheme (uses *bit-flipping decoding*) where the decoder seeks to find the codeword that was most likely sent.This object does not take an alist file name but instead a predefined matrix object. Also it can take either a H or a G matrix constructed by the "LDPC Parity Check Matrix" or "LDPC Generator Matrix," respectively.

Finally, in order to inverse the padding operation previously done, it is used the "Keep M in N" block where we only keep the first 440 bits of data. The rest of the FG is analogue to the CC's FG with just some changes about the values of header and the "Stream Mux" blocks.

Furthermore, I have added an example (found in: [21]) in `Extra` folder called `ber_curve_gen_ldpc` where you can analyse the BER curve in different LDPC types, namely, Parity check and Gen. matrix. I advise you to run directly from console using python to avoid some problems.

VI. Guided Tutorial - Fully Wireless

A. USRP - Tutorial_8.grc

All the necessary code to get a transmission in simulation work is placed, so it is time to get fully wireless. For this, we will remove all the part of the channel model and add the "USRP Sink"/"USRP Source" blocks and remove the "Throttle" block (As it was already discussed in Section IV). Note that I also have said that we can not have 2 "clock" blocks in the same FG due to the "2 Clock Problem" , however, we can have two "USRP" blocks because the GR will consider two totally independent flows (not back-pressuring data, thus not changing sample rates). Also, you need to change the "Device Address" parameter to your USRP's serial in order to get the provided FG working. This address can be found running the next command:

```
$uhd_usrp_probe
```

When we change from simulation to real-world there is introduced a problem called "Saturation Limits", this is, the "USRP Sink" needs to send float values in $[-1, 1]$ range, anything below or above is bad because causes the signal to be saturated and nonlinear which we do not desire. The values feed into the USRP block can be seen by adding a "Time Sink GUI" where it is possible to analyse values above the limit. That is exactly why there is the need to add the "Multiply Const" block with a constant value experimentally found in a way to get the "Time Sink GUI" values on the range mentioned. However, this block does not solve for the RX side where it is required to play around with the TX/RX gains. Once you find good values you can define these values in the correspondent variable blocks. If you do not change the USRP's positions this method should work well (an automatic method to the same effect is add an Automatic Gain Controller (AGC) block).

Now that we are providing good values to the USRP's we need to maximizing the throughput by tweaking the sample rate in a form that can be the maximum possible (without dropping data) which can cause mainly two problems: "Underrun" (prints U's on the console) and "Overflow" (print O's on the console).

The "Underrun" happens when the TX side is not giving samples quick enough to the "USRP Sink" block to send them, this can be caused by lack of processing power (note that

the encoder block requires a lot of power) so there are two solutions: Or it is changed the CPU or it is slow down the sample rate of both USRP's. Another problem that may happen is "Overrun", this is, in RX side we are not able to consume samples quick enough, probably because of the back-pressure caused by the decoder block.

Taking this into account the goal is find the maximum value for the sample rate where if happens some of these two problems there must be only on the beginning of the transmission because it is rampaging up. Note that, this is another reason to use the vector preamble previously made, this is, if occurs any of this problems, it happens while data is not being transmitted yet. Another thing to consider is that the chosen sample rate value must be where **Clock Rate/Sample Rate** results in an integer value and for better performance that resulted value should be divisible by 4. The **Clock Rate** in the USRP's can be controlled in the Master Clock Rate (MCR) argument of the respective USRP blocks. Taking this into account we improved the USRP's MCR for $60e6$ and we have created a vector with sample rates that are best for performance, thus, we just need to choose the index on the vector to change the sample rate.

B. USRP - Tutorial_9_Differential_Encoder.grc

Another interesting problem is that taking a look to the RX data that is written on the file, sometimes it writes and other times it does not. The Costas Loop previously used is a blind synchronizer of the transmitted signal thus, not having the true orientation arising the problem of phase ambiguity. This problem consists in the possibility of the constellation points being rotated some amount of degrees depending the type of the modulation, generically, for MPSK it is possible to have $\frac{360}{M} * n, n \in [1, ..., M-1]$) degrees of rotation comparing with the generated constellation on TX side. In our case we have of $90/180/270$ degrees of rottarion in our received constellation.

There is two solutions for this problem, one of them is using differential encoding, where in TX side, instead transmitting the direct mapping of symbols into the constellation points, it is encoded the difference between these symbols and then mapped the resulting symbol into the constellation point. Here, a symbol depends not only of the transmitted symbol but also the previous one, being represented as $y[0] = (x[0] - x[-1])\%M$, where M is modulus of code's alphabet) [26]. At the RX side it is necessary do the reverse operation removing this way the phase ambiguity. The other solution is by using a sync word to train the receptor which will be addressed in the next Section.

As everything else we have pros and cons in the use of one or another. The advantage of differential encoder is that there is no need to train the receiver, hence it is faster because we are not introducing overhead, however, has two main disadvantages:

- If occurs one error in the transmission can cause two symbols wrongly recovered (because the dependency of the last one).
- It is not possible to use gray code mapping (if one bit is wrongly received the decoded symbol may have more than 1 bit error).

In our FG we will remove the "Map" block, since it does nothing now (because we cant use other map than $[0, 1, 2, 3]$ or we will take out the difference of symbols), and we will add a "Differential Encoder" Block right before "Chunks to Symbols" block. In RX is just do the inverse thing, right after the decoding block it is applied the "Differential Decoder" to retrieve the original symbols.

C. USRP - Tutorial_9_Correlation_Estimator.grc

The other solution to correct the phase ambiguity is by using a codeword known as a sync word, which is a set of bits known by both by TX and RX sides. When the TX sent this the RX is capable of identify this sequence, and hence correct the phase, therefore receiving the remaining information without any ambiguity. The identification method used is called correlation where we correlate the signal received against this word. When the signal contains the sync word the correlation procedure results in a peak of energy that can be used for phase estimate, thus, being possible to set the proper phase removing ambiguity and even estimate other properties of the signal. The sequence has a low auto-correlation in all the points except in the central one, being in this manner strong against noise (it can change some bits). There is already good known sequences for this sync word, for instance, the "Barker Codes" [27] (has 13 bit lengh, however bigger sequences means higher peaks). With this method it is possible using the gray code map, so if 1 bit is wrongly received than the decoded symbol is the neighbor of the real constellation point, thus, being at maximum, one-bit error.

With the "Correlation Estimator" block we can estimate some properties of the signal. At the input of this block is set the signal containing the sync word which is correlated with the defined sync word. The output signal will be exactly the same added some tags on the stream such as:

- phase_est - Estimation of the phase offset, used by the "Costas Loop" to reset its internal phase value and speeds up the synchronization, removing here the phase ambiguity.
- time_est - Estimation of the timing offset, used by the "Polyphase Clock Sync" to synchronization.
- corr_est - The values results from the correlation operation.
- amp_est - Amplitude estimation which can be used to multiply against the signal before it get in the "Polyphase Clock Sync" block.
- corr_start - The first sample of the correlation and its correspondent value.

To the sync word we use the default access code, where the result of auto-correlation it is showed in the Figure 7, besides that, as it is the same known word for the header of the packet we do not increase the amount of data transmitted.

Now we need to fill the "Correlation Estimator" block parameters. The "Symbols" parameter is the "Sync word"

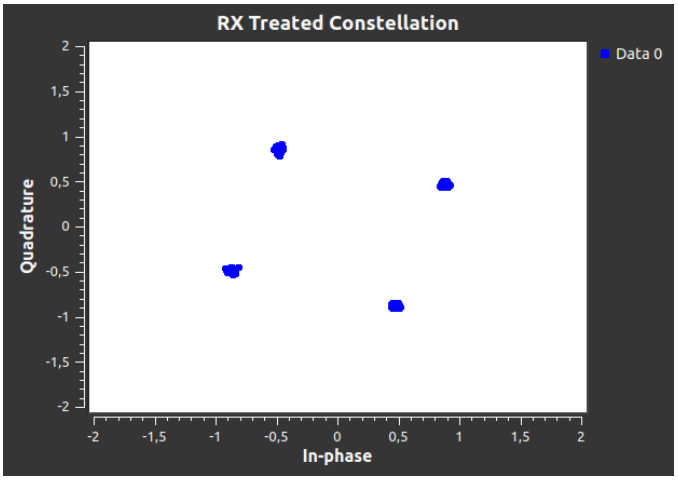


Fig. 7. Auto Correlation

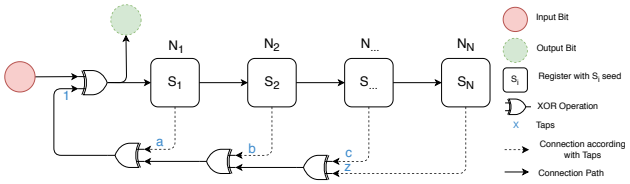
previously mentioned already modulated, so, in order to do that, it will be used a block called "Modulate Vector" where is input the vector containing that word and modulates it (also applies the filter taps parameter). The modulator we will use will be the generic one with the parameters: (1) the used constellation object; (2) "false" to differential encoder; (3) 4 to the SPS; (4) "true" for using the constellation object's map; (5) 0.22 to the roll-off, (6) "false" for verbose and finally, (7) "false" for logs, which results in: `(digital.generic_mod(pld_const, False, sps, True, eb, False, False))`. Note that this generic modulator take packed bytes, so we need to create a vector of packed bytes of the 64 bits access code, and applies the RRC so we do not need to do it in the "Filter taps" parameter. There is tricky thing here, because there exists a delay caused by the correlation detection algorithm (affected by the size of the sequence) resulting in the need to adjust the place where the tags are actually positioned on the stream. The "Tag markinig delay" parameter specifies that delay and its influenced by the SPS and sequence length. This value is found empirically by tweaking the values and analysing the second output of this block.

Finally, the "Threshold" parameter is where we define how much the stream must match to the sync word (in the correlation algorithm). There is set values between $[0-1]$, I have set 0.999 because it worked well, however if it is not working for you, try to decrease this value. Note that you can see the correlation working in "Correlation" and "Correlation ˆ 2" plots.

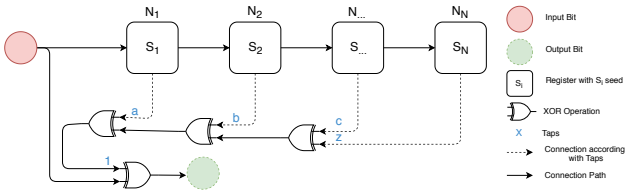
D. USRP - Tutorial_10.grc

As discussed, a received signal goes through many algorithms to successful recover the transmitted signal, for the purpose of both TX and RX sides being synchronized. This is possible because it is receiving different bits, however, if it is sent many zeros in a row, then, the receptor loses the symbol synchronization. For instance, by testing using the `videofhd.mpeg` file provided you can see that the zeros are not transmitted well. To solve this problem, instead of transmit all this zeros in a row, transmit the information with some bits at 1 without adding significant overload to the transmission. The scramble is often used to improve security in a transmission, however, at this view only but aims to eliminate long sequence of same bits, called as whitening sequences, consequently reducing problems with synchronization at the receiver [28].

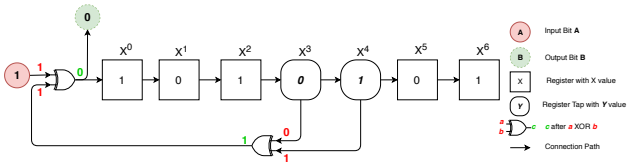
There are two types of scrambler, the multiplicative scrambler and the addictive scrambler which differs mainly in the self-synchronizing capability, this means that even with different seed after a some amount of input bits the original content will be descrambled correctly (which is good for a normal transmission) while the additive one will not be able to recover. Another property to keep in mind is, when descrambling, if one error bit is input, than will cause the output of K wrongly descrambled bits, being K the number of taps.



(a) Multiplicative Scrambler architecture.



(b) Multiplicative Descrambler architecture.



(c) Scrambler example for an input 1, with 1010101 as seed value and the $x^4 + x^3 + 1$ polynom.

Fig. 8. Multiplicative scrambler architecture represented in 8(a)) and its respective descrambler in 8(b)). A specific example is shown in 8(c)).

A multiplicative scrambler [29] is recommended in V.34 by International Telecommunication Union-Telecommunication Standardization Sector (ITU-T) [30]. It works by using a , i.e. a shift register whose is a linear function of its previous state (because is dependent of the content of the previous state of those registers) and new received bit. It is necessary to define three initial parameters: the number of registers N; the seed which is the initial value placed in theses inputs $s_1, ..., s_N$; and a polynomial of type $x^0 + x^a + x^b + ... + x^z$ where the register number $0, a, b, ..., z$ are the taps. These taps are the registers that influences the output bit used for each iteration;

the other records are not considered since it does not influence the output bit. Note that x^0 is always 1 (always does one XOR ($\oplus$) operation between the input bit and the tap's resulted bit).

For better understanding, consider the example of the architecture presented in Fig. 8(a)). An iteration is when a data bit is input and XOR'd with the bit that results from XOR operation of all of the tap's content (equivalent to $sum(modulo2)$ of all bits) on the current state resulting in the output bit already scrambled. After this all the registers are updated shifting to the right register and the output bit is set in the first register discarding the last one.

In the descrambler operation it is just do the reverse operations mentioned above. For each iteration on descrambler is input the bit. Then the output bit is the XOR of that input bit with all taps XOR'd between them and finally. On that iteration the input bit is added on the shift register moving all values to the right. Note that it is required to start with the same seed value as in the scrambler or it will be lost some bits. The descrambler architecture is shown in Fig. 8(b)).

The Fig. 8(c)) presents a practical example where $N = 7$, as registers containing as initial seed 1010101 and the scrambler polynom is $(0X19)_{16} = (00011001)_2 = x^4 + x^3 + 1$, (hence only the content of 4^{th} and the 5^{th} register influences the output). Let's say that these registers have as contents bit X and bit Y respectively. For a given input bit B, the result that will add to the first register is: $B \oplus X \oplus Y$, and finally all bits are shifted to the next register and B is set in first register (so X and Y is now in the 5^{th} and 6^{th} register respectively). In concrete, if the bit 1 is input, then $1 \oplus 0$ (4^{th} register) $\oplus$ 1 (5^{th} register) results B as 0. The content of the registers will be 0101010 and the output bit scrambled that will be transmitted is 0. Always like this in successively.

We already have a native scrambler in GR that we can use we just drag and drop, however, the input as a continuous stream, thus, the output bit of a Linear-Feedback Shift Register (LFSR) is dependent of the last input bit as previously discussed. With this in mind, if a packet is lost, then there will also be lost some initial bits on the next packet. Thus, the idea is create our own scrambler that takes a pre-established seed and scrambles a piece of bits (a packet), then resets the seed to the pre-established one and scrambles another packet, and so on, using always the same seed to initiate the registers. The descrambler is similar work. A problem that we face is that on scrambler, at first the registers contain the seed, thus, the first output byte after the descrambler would be junk. Also, it not outputs the last byte because it stays in the registers (it is not flushed). To solve the first problem we just need to drop the first byte after we scramble the data in such way only the real information is transmitted. To solve the second problem we just need to flush the scrambler by input one more byte, so the last byte of the packet is transmitted as well.

Going to the code itself, the idea is create frames to enable us reset the LFSR's registers. Go to GR's installation folder, which if you followed my instructions should be here:

```
$/usr/local/include/gnuradio/digital
```

In the `lfsr.h` file there is a function called `reset()` where the registers are reset to a given seed. We will use it for resetting the seed before scramble/descramble every packet.

First we need to create an OOT like it was discussed before, we used `scrambler_packets_same_seed` to the module's name and `scramble_packetize/descramble_packetize` to the scrambler/descrambler blocks names, respectively. In our scrambler source code we have 3 branches possible:

- If it is the first bit of each frame then we scramble then but we do not send them (first 8 bits are junk as discusses), here we just consume samples we do not produce them.
- If it is the case of the last bit of the frame then is is required to produce 8 more bits in order to flush the LFSR, in this case we just produce samples.
- Finally, if we are in the middle of the frame, basically we just check the output buffer space and we scramble whatever is possible (the size of the buffer or the size of the frame), consuming and producing samples.

To direct the incoming bit to each branch we have to create two flags, one for each of the this first two branches. The code in the work function of the source file is:

```
const unsigned char *in = (const unsigned char *) input_items[0];
unsigned char *out = (unsigned char *) output_items[0];
int ii=0; //Track how many input bits we consume
int oo=0; //Track how many output bit we produce
if(flag_last==1){ //It's the last bits of the frame: -> We need to flush it
 max_n_produce=(std::min(noutput_items, track_n_bits_added)); //Check buffer to the amount of bits that we can scramble
 for(int i=0; i<max_n_produce; i++){
   out[i]=d_lfsr.next_bit_scramble(0);
   oo++;
 }
 if(max_n_produce==track_n_bits_added){
 //If we sent all bits: ->Reset variables ->Go to the first bits branch
   flag_last=0;
   flag_first=1;
   track_n_bits_added=8;
 }else{ //If we didn't sent all bits, then go again to send what is possible.
   track_n_bits_added= track_n_bits_added-max_n_produce;
```

```
    }
}
else if(flag_first==1){//First bits of the frame: ->Trash so we DROP
  d_lfsr.reset(); //Reset the registers using the seed
  for(int i=0;i<8;i++){  //DROP 8 bits
    d_lfsr.next_bit_scramble(in[i]);
    ii++;
    remaining_bits--;
  }
  flag_first=0;
}
else{ //Normal behaviour - Inside the frame
  max_n_produce=(std::min(noutput_items, remaining_bits)); //Check buffer to the amount of bits that we can scramble
  for(int i=0; i<max_n_produce; i++){
      out[i]=d_lfsr.next_bit_scramble(in[i]);
      ii++;
      oo++;
    }
    if(max_n_produce==remaining_bits){ //If we sent all bits:  ->Go to last branch to flush ->Reset variables
      flag_last=1;
      remaining_bits=n_frame;
    }else{ //If we didn't sent all bits, then go again to send what is possible.
    remaining_bits=remaining_bits-max_n_produce;
  }
}
consume_each (ii);
return oo;
```

In the descrambler source code file we just use the function `next_bit_descramble()` in `lfsr.h` file to descramble the incoming bits taking into consideration the separation into frames and resetting the seed between them: The source code will look like this:

```
const unsigned char *in = (const unsigned char *) input_items[0];
unsigned char *out = (unsigned char *) output_items[0];
int ii=0; //Track how many input bits we consume
int oo=0; //Track how many output bits we produce
max_n_produce=(std::min(noutput_items, remaining_bits)); //Check buffer to the amount of bits that we can descramble
for(int i=0; i<max_n_produce; i++){
  out[i]=d_lfsr.next_bit_descramble(in[i]);
  ii++;
  oo++;
}
if(max_n_produce==remaining_bits){ //All bits of the frame was sent: ->Reset registers with seed ->Reset variables
  d_lfsr.reset();
  remaining_bits=n_frame;
}else{ //If we didn't sent all bits, then go again to send what is possible.
  remaining_bits=remaining_bits-max_n_produce;
}
consume_each (ii);
return oo;
```

If you prefer, instead doing manually all this code, I let you the all OOT ready to use in `Extra/OOT/Scramble` file. To install it is necessary to unzip it and run the `./script.sh` file inside `build` folder. To uninstall the OOT you need to do it manually by running `sudo make uninstall` in the same folder.

APPENDIX

E. Appendix A - Installation

Going to a specific installation, we need a Linux base for getting started, the next installing guide was meant to Ubuntu 18.04 or the Ubuntu 19.04 but it may work in other Linux OS's. First we will need to install all dependencies needed, with this commands:

```
$sudo apt install cmake
$sudo apt-get install build-essential
$sudo apt install git g++ libboost-all-dev python-dev python-mako python-numpy python-wxgtk3.0 python-sphinx python-cheetah swig libzmq3-dev libfftw3-dev libgsl-dev libcppunit-dev doxygen libcomedi-dev libqt4-opengl-dev python-qt4 libqwt-dev libsdl1.2-dev libusb-1.0-0-dev python-gtk2 python-lxml pkg-config python-sip-dev
```

F. Installing VOLK

We need to install VOLK (Vector-Optimized Library of Kernels) because we will not use the internal VOLK of GNU Radio but an external one since sometimes problems arise

from there. VOLK it is a free library that contains kernels for mathematical operations, this means that for an operation it is created a proto-kernel and added to VOLK for the architecture that we wish, hence faster operations. To install this follow the next commands:

```
$git clone https://github.com/gnuradio/volk
$cd volk && mkdir build && cd build
$sudo apt install cmake
$cmake ../ && make && make test
$sudo make install && sudo ldconfig
```

To complete this, we need to create a profile for our architecture, for this go to `/usr/local/bin/` folder and run `./volk_profile`. Note that the configuration file is written in the next path: `/home/$USER/.volk/volk_config`.

G. Installing UHD Driver

To use the USRP's we need to install the UHD Driver, here we have 2 options depending on your Ubuntu's version, first try to use the Personal Package Archive (PPA) provided by Ettus using this next command lines:

```
$sudo add-apt-repository ppa:ettusresearch/uhd
$sudo apt-get update
$sudo apt-get install libuhd-dev libuhd003 uhd-host
```

If for some reason you could not get automatically, then install the next packages `libuhd-dev, libuhd3.14.1` and `uhd-host` downloaded from Ettus website [31].

Finally, to download the image of your USRP just run: `./uhd_images_downloader.py` inside the folder `/usr/lib/uhd/utils/`.

If everything worked out you can run the next command to see the devices connected:

```
$uhd_find_devices
```

This command is used to see information about these devices:

```
$uhd_usrp_probe
```

H. Installing GNU Radio

Gnu Radio is available in the following repository: [32]. The most up-to-date version is 3.8, however still has several stability problems and bugs, therefore we will resort to version 3.7. The following steps are used to install this version, although the process should be similar for other versions as well:

```
$git clone https://github.com/gnuradio/gnuradio.git
$cd gnuradio && git checkout maint-3.7
$mkdir build && cd build
$cmake -DENABLE_INTERNAL_VOLK=OFF ../
$make && make test && sudo make install && sudo ldconfig
```

You can now use the GR and to see where was installed use this command:

```
$which gnuradio-companion
```

I. Problems and Solutions

After installing GR when you open, if it shows some warnings about `Canberra`, the solution is install it with:

```
$sudo apt install libcanberra-gtk-module libcanberra-gtk3-module
```


ACKNOWLEDGMENT

FALTA ACK

REFERENCES

[1] Wireless Innovation Forum. https://www.wirelessinnovation.org. [Online; Accessed: 04/02/2020].

[2] Wireless Innovation Forum. What is sdr? https://www.wirelessinnovation.org/Introduction_to_SDR. [Online; Accessed: 04/02/2020].

[3] MatWorks. MATLAB. https://www.mathworks.com/products/matlab.html?s_tid=hp_products_matlab. [Online; Accessed: 04/02/2020].

[4] MatWorks. Simulink. https://www.mathworks.com/products/simulink.html. [Online; Accessed: 04/02/2020].

[5] National Instruments. LabView. https://www.ni.com/en-us/shop/labview.html. [Online; Accessed: 16/10/2019].

[6] GNURadio Companion. GNURadio. https://www.gnuradio.org/about/. [Online; Accessed: 04/02/2020].

[7] Ettus Research. USRP B210 Kit. https://www.ettus.com/all-products/ub210-kit/. [Online; Accessed: 04/02/2020].

[8] Ettus Research. VERT 2450 Antenna. https://www.ettus.com/all-products/vert2450/. [Online; Accessed: 04/02/2020].

[9] GNURadio Companion. Install Dependencies. https://wiki.gnuradio.org/index.php/UbuntuInstall#Install_Dependencies. [Online; Accessed: 04/02/2020].

[10] GNURadio Companion. Vector-Optimized Library of Kernels. http://libvolk.org/. [Online; Accessed: 04/02/2020].

[11] Ettus. USRP HD. https://files.ettus.com/manual/. [Online; Accessed: 04/02/2020].

[12] Ettus Hardware. USRP Hardware Driver. http://www.ettus.com/sdr-software/uhd-usrp-hardware-driver/. [Online; Accessed: 04/02/2020].

[13] GNURadio Companion. Install GNU Radio. https://wiki.gnuradio.org/index.php/UbuntuInstall. [Online; Accessed: 04/02/2020].

[14] Travis F. Collins, Robin Getz, Di Pu, and Alexander M. Wyglinski. *Software-Defined Radio for Engineers*. Artech House Publishers, 2018.

[15] NASA. Root Raised Cosine Filters & Pulse Shaping in Communication Systems. https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20120008631.pdf. [Online; Accessed: 04/02/2020].

[16] GNU Radio Wiki. Polyphase clock sync. https://wiki.gnuradio.org/index.php/Polyphase_Clock_Sync, 2019. [Online; Accessed: 2/11/2019].

[17] Trondeau. Control Loop Gain Values. http://www.trondeau.com/blog/2011/8/13/control-loop-gain-values.html. [Online; Accessed: 05/02/2020].

[18] Jr C. Richard Johnson, William A. Sethares, and Andrew G. Klein. *Software Receiver Design - Build Your Own Communication System in Five Easy Steps*. Cambridge, 2011.

[19] Fred Harris. Band edge filters: Characteristics and performance in carrier and symbol synchronization. *The 13 th International Symposium on Wireless Personal Multimedia Communications*, 2010.

[20] Massachusetts Institute of Technology. Convolutional codes. http://web.mit.edu/6.02/www/s2009/handouts/labs/lab5.shtml, 2018. [Online; Accessed: 16/10/2019].

[21] GNU Radio. FEC Examples. https://github.com/gnuradio/gnuradio/tree/master/gr-fec/examples. [Online; Accessed: 05/02/2020].

[22] Sarah Johnson. Introducing low-density parity-check codes. *School of Electrical Engineering and Computer Science - University of Newcastle, Australia*, 05 2010.

[23] R. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, pages 21–28, 1962.

[24] D. J. C. MacKay. Good error-correcting codes based on very sparse matrices. *IEEE Transactions on Information Theory*, 45(2):399–431, March 1999.

[25] GNU Radio. FEC API. https://www.gnuradio.org/doc/doxygen/page_fec.html. [Online; Accessed: 06/02/2020].

[26] GNU Radio Wiki. Differential encoder. https://wiki.gnuradio.org/index.php/Differential_Encoder, 2019. [Online; Accessed: 16/10/2019].

[27] Peter Borwein and Michael J. Mossinghoff. *Barker sequences and flat polynomials*, page 7188. London Mathematical Society Lecture Note Series. Cambridge University Press, 2008.

[28] A. Bruce Carlson and Paul B. Crilly. *Communications Systems - An Introduction to Signals and Noise in Electrical Communication*. McGraw-Hill, 5 edition, 2010.

[29] University of British Columbia Course ELEX: Data Communications Lesson 13. Pn sequences and scramblers. http://www.ece.ubc.ca/~edc/3525.jan2014/lectures/lec13.pdf, 2014. [Online; Accessed: 16/10/2019].

[30] International Telecommunication Union Telecommunication Standardization Sector. *Series V: Data Communication Over the Telephone Network*. International Telecommunication Union, 1998.

[31] GNU Radio. FEC API. https://launchpad.net/~ettusresearch/+archive/ubuntu/uhd/+packages. [Online; Accessed: 07/02/2020].

[32] GNU Radio. Repositorio GitHub. https://github.com/gnuradio/gnuradio. [Online; Accessed: 07/02/2020].