Practical Tutorial for SDR using GNU Radio

Andr Silva[1]

Abstract— **This electronic document is a live template. The various components of your paper [title, text, heads, etc.] are already defined on the style sheet, as illustrated by the portions given in this document.**

I. INTRODUCTION

SOMETHING

II. GETTING STARTED

We need a Linux base for getting started, this installing guide was meant to Ubuntu 18.04 or the Ubuntu 19.04.

A. Installing VOLK

First of all we will install VOLK (Vector-Optimized Library of Kernels) because we will not use the internal VOLK of GNU Radio but an external one. VOLK it's a free library that contains kernels for mathematical operations, this means that for an operation 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
$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/bin/` folder and run `./volk_profile`. Note that the configuration file is written in the next path: `$HOME/.volk/volk_config`.

B. Installing GNU Radio

Now moving for GNU Radio, the first thing to do is install all the dependencies needed, with the next command:

```
sudo apt install cmake 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
```

After dependencies installed we will install GNU Radio via git. It is available the Version 3.8 however still has a lot of stability problems and bugs so we keep with the last commit of 3.7 version:

```
$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 GNU Radio, to see where was installed where it is a usefull command: "which gnuradio-companion".

C. Installing UHD Driver

To use the USRP's we need to install the UHD Driver, and for that we install three packages: `libuhd-dev, libuhd3.14.1` and `uhd-host`. This packages can be downloaded here: `https://launchpad.net/~ettusresearch/+archive/ubuntu/uhd/+packages`.

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

If everything works you can run the next command to see the devices connected: `$uhd_find_devices`, and this one to see information about that devices: `$uhd_usrp_probe"`.

D. Problems and Solutions

After installing GNU Radio when you open it if shows some warnings about `Camberra` just install it with:.

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

III. GUIDED TUTORIAL - MODULATION

BALBLABLA

A. Modulation - Tutorial_1.grc

In order to transmit data between SDR's it's necessary modulate the signal, so the modulation is the act of varying properties of a waveform, this wave is called carrier and keeps the information transmitted, so the the goal here is pick the data and modulate it so it can be transmitted in single carrier.

First we need to understand the working of some blocks and hence the GNU Radio itself.

*This work was supported by Instituto de Telecomunicaes

[1]Andr Silva is with Faculty of Sciences and Technology, University of Coimbra, Portugal

We can convert bytes in complex numbers using the "Chunks to Symbols" block, this block has a parameter called "Symbol Table" where we define the mapping between a byte (a chunk, this is our symbol) and a constellation point (point in the constellation to the symbol). But before we need to do two things: First we need to unpack the byte, this means split the 8 bits/byte in X significant bits/byte where X is the number of bits necessary to the symbol of the constellation type, this is, if we want modulate in BPSK X is 1, QPSK X is 2, 8PSK X is 4. This is possible using the "Repack Bits" block, this block has 3 important parameters, the "Bits per Input Byte" where we define how many bits we pick in an incoming byte, the "Bits per Output Byte" where we define how many significant bits will have the output byte (the remaining bits will be 0's), and finally the parameter "Endianness" to determinate if we want write in MSB or LSB (always read in LSB).

For example if we input the byte = "01000010":

With 8:2 in MSB we got:"000000**01**", "000000**00**", "000000**00**", "000000**10**".

With 8:2 in LSB we got:"000000**10**", "000000**00**", "000000**00**", "000000**01**".

Note that this block leads to an interpolation of "Bits per Output Byte/Bits per Input Byte, in this case 8/2=4, this is, for every 1 bytes outputs 4 bytes".

Second we need to map the resulting bits in the symbols that we really want, using the "Map" block that basically does: `output[i] = map[input[i]]`.

Lets suppose we want modulate using QPSK: First we create an "Constellation Object" where we insert the mapping of the symbols relative to the constellation points. We need to pick the 8 bits/byte and slicing in 2 significant bits per byte, so we use the "Repack bits" with 8 "Bits per Input Byte" and 2 "Bits per output Byte" with MSB endianness. Currently we have a byte with the first 6 bits with 0's and the next 2 bits with 2 bits of data, however it's not our symbol yet). Then we map this significant 2 bits into the symbol we want taking into account that we will map this symbols in constellation points next (using the "Constellation object" block). In this case the 2 significant bits coincide with the symbol, but it depends of what you want to do. Resulting in mapping into the constellation points in the 1

The last block is "Throttle", to understand this block we need understand how GR works. When we do data streaming, we are implicitly using a buffers to hold the data between blocks, this buffers are in finite size, so before the block do anything, it checks the amount of available space in input and output buffers, if there is none available space in that buffers the block don't do anything. This is called "Back-pressure" because we pressure back the data that is incoming, slowing things down. This is how GR controls the data flow, giving data to the block when it's necessary and holding it when block is busy working. We have source/sink blocks that produces/consumes data at a given fixed sample rate like Audio and USRP blocks, these blocks are called "clock" blocks, so typically a FG has only one source or one sink block, and not both because that leads

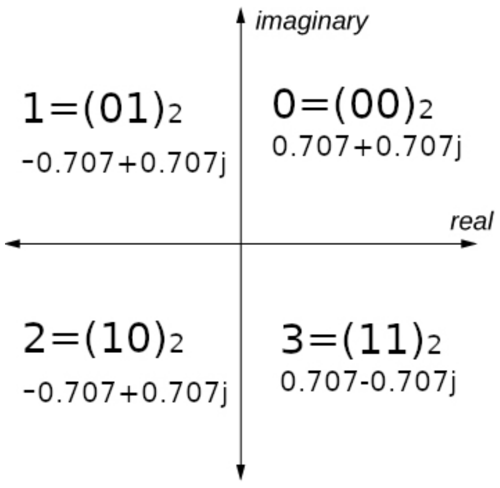

Fig. 1. QPSK: Mapping Symbols to Constellation Points

to an problem called as "2 clock problem" where there is 2 blocks with different sample rates leading a asynchronous clock sources causing "underruns"/"overflows" depending on the production/consumption rates are differentiating. As our FG doesn't has any of this "clock" blocks then we need to add 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. This means that while we are in simulating mode we need this block to control the flow however when we start using the "USRP" blocks we can take out this block.

To finalize the TX side we send the data to an virtual sink, that redirects the data to the virtual source with the same id, the only purpose of this block it's make the FG more clear, elegant and readable.

Now we move to the RX part of the FG where we use the "Constellation Decoder" to invert what we have done, this is, map the complex points to symbols (using the parameters of the "he "Constellation Object" block), and then we use "Map" block again to map symbols into bits of real data. And then pick the 2 significant bits and pack in a byte exactly the inverse as before.

B. Modulation - Tutorial_2.grc

What we have done is not enough because in real world we cant use all the bandwidth, so we need no narrow it, however if the channels are too narrow the symbols will be too wide and the tail of the last symbol will interfere with the present one inducing ISI (Inter-Symbol Interference). A pulse shaping filter we can use to work around is the RRC (Root Raised Cosine) filter because it's a feasible solution and it's implemented in GR.

If we don't put a shaping filter on the signal, the waves will have a lot of energy in adjacent channels. This filter uses taps to apply against our signal shaping it. This taps are calculated based in how many number of filters we want and how many samples we want per symbol. Theoretically has an infinite number of taps hence we can infinite attenuate the stop band, so really we can define the limit according with processing power we have. Remember that we have added ISI here that we will have to treat in RX side, as we will see.

Another thing to keep in mind is the roll-off factor, this is the measure of the excess bandwidth of the filter (bandwidth that is occupied beyond Nyquist bandwidth). This parameter is basically the inclination of the wave when rising and falling, when we set more roll-off, let's say 1, then will be easier for the receiver to decode however we will narrow less the bandwidth and that is not our objective, on the other hand, smaller roll-off will result in narrower bandwidth, however the inclination increases so attenuation in stop band is reduced. I provide an FG that made called ”roll_off.grc” inside of ”extra” folder where you can analyse the plot of 5 different roll-off factors. In this tutorial we will use the 0.22 value to this parameter.

Another thing included in modulation is the interpolation, so basically we can set a number of SPS (samples per symbol) that we want however, logically, more samples per symbol leads to less probability of errors hence inducting more overhead to our transmission. What this does is construct new points in the wave between the existing ones thus when we will reconstruct the signal we 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 4 for this parameter.

Note that at this point we have ISI in our signal, so first I have to get rid of this by using another RRC filter. When we combine 2 RRC filter together we get a RRC filter with Nyquist filter form. Knowing this property we can use another RRC filter resulting in a raised cosine pulse shaped filter with less ISI. If you observe the constellation plot ”RX Treated Constellation” there is a little noisy as a result of the ISI after the 32 filters however much less noisy than ”RX Constellation”.

Now that we have applied a RRC filter to our RX signal, in the same way that we have re-sampling by 4 we need to decimate to converting 4 samples per symbol into only 1 sample per symbol.

C. *Modulation - Tutorial_3.grc*

Now we are ready no get closer to reality, for that let's implement a ”Channel Model” block where simulates different properties of a channel that we will need to take care:

- 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.

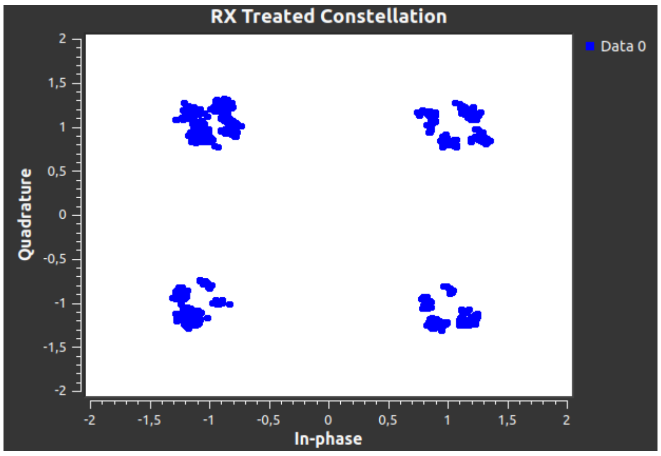

Fig. 2. Effects Multipath on Constellation Plot

- Multipath - We can emulate multipath delay by adding taps of a FIR (Finite Impulse Response) filter in the ”Taps” parameter.

The FG we have is already taken care of noise and timing offset using the ”Polyphase Clock Sync” block. If you play with the tweaks ”Time Offset” and ”Noise Amp” you can see in the plot of ”Rx Treated constellation” a nice and clear constellation, however if you tweak the ”Freq Offset” you can see that the constellation become a circle, that's because we are not treating that, it will be what we will do next tutorial.

D. *Modulation - Tutorial_4.grc*

Multipath is having different paths to our receptor like when we are using wireless the signal reflects on objects of the environment, for example walls, and go to receptor a little after the direct path, so the receptor receives the same signal various times with timing differences. I added some taps to simulate a multipath property, if you disable ”Costas Loop” and the ”CMA Equalizer” and set ”Output SPS” to 1 in the ”Polyphase Clock Sync” block you can see the effect of these taps because the constellation points are no longer clean and really converged: Fig: 2

This, obviously affects our transmission so to take out this problem we will need an equalizer, called ”CMA equalizer”. Here we define the number of taps where more taps means better final results however more overhead to the algorithm thus we want keep this number small but enough for correcting our channel (based on educated guess, I set 15 because works with other implementations and is enough for now). The ”Gain” I set 0.01 and the ”Samples per Symbol” of the input signal is set to ”2”, hence, the ”Output SPS” of ”Polyphase Clock Sync” block is also set to ”2”. If you take a look now for the constellation plot you will see a constellation a lot more converged and clean: Fig: 3

Note that the constellation is rotating a little, 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, thus we need to take care of the phase

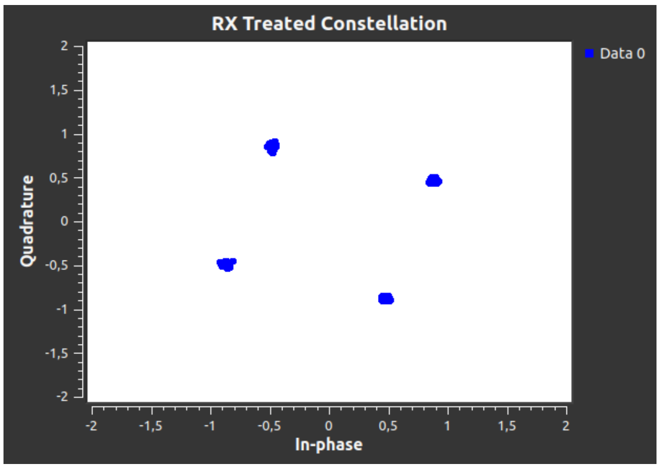

Fig. 3. Multipath corrected on Constellation Plot

offset. Besides that, if you tweak the ”Freq Offset” you will see an circle instead a constellation, that is why we need to take care of the last property mentioned before: ”Frequency offset”. We can track the offset of both of this things using an second order loop, however this type of recover is ”fine frequency correction” this means that we need to be close of the ideal frequency or our constellation won’t converge and will keep spinning. We have to set the ”Loop bandwidth” and the ”Order” parameters, the first one is relative to the second order loop (I will use 6.28/200.0), the second is about the order of the constellation type (2 to BPSK, 4 to QPSK and 8 to 8PSK, thus 4 for our example). Now you can enable this block you will note that the constellation is no more rotating and if you tweak the Frequency Offset note that it’s not getting a circle anymore, instead that, the constellation keeps without rotation, clear and converged.

E. Modulation - Tutorial_5.grc

If you take a closer look we have an problem writing the file, sometimes writes something and another times doesn’t. That’s because we might have an ambiguity of 90 degrees in our constellation. The solution for this is: Instead transmitting constellation itself, we will transmit the difference between symbols of the constellation, and then mapping the transmitted symbols into our bits of data. Basically the differential encoding depends not only of the transmitted symbol but also the previous one, removing ambiguity of 90 degrees. If it’s the case of BPSK we would have 180 degrees of ambiguity, if it’s 8PSK it would be 45 degrees of ambiguity. In our FG we can resolve by adding an ”Differential Encoder” Block right after ”Map” block resulting in first mapping the bits of data in symbols, and then convert these symbols in differential symbols dependent of the last one, and finally convert to constellation symbols. In RX is just do the inverse thing, right after we have the transmitted symbols we use ”Differential Decoder” to get the real symbols and then map to the bits of data.

F. Packet communications - Tutorial_6.grc

Now we have successfully modulated and transmitted our signal without ambiguity, but if you take a look in the output file, it will only be junk. This is happening because it’s not writing the bits in the supposed byte. Explaining better, if I send the byte ”01010101” and for some motive receives 4 bits in 0’s and then the right byte, this results in ”0000**0101**” and ”**0101**XXXX”, then all the bits forward are not aligned. This is the problem that is happening here because the ”Polyphase Clock Sync” block needs some time to adapt and in this time losses some bits, so we need a manner to always receive the bytes aligned. The solution for this is packetize our data and send the packet, consequently, we will lose all that packet but we will always get our bits aligned.

The first thing we need to know is how the ”Correlate Access Code - Tag Stream” block works. It expect unpacked data, this means that the input only has 1 significant bit per byte. Then exams the input finding an sequence called ”Access Code” of 64 bits taking into account that can be X bits different where X is the ”Threshold” parameter. After find this sequence the next 16 bits is the length of the payload (that will be outputted) 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 of payload length, being that the length of the payload is in bytes. Now that we have the composition of the header of the packet we need to create it.

Assuming that we want transmit packets with 112 bytes of payload size, hence 896 bits, then we need to create the header with that information. 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 (I will use the default that can be accessed here ”digital.packet_utils.default_access_code”), followed by ”00000000 01110000” = 112 and then ”00000000 01110000” = 112. With header ready we need to split the input in 1 bit/byte using ”Repack Bits” block mentioned before, then concatenate the 96 bits of the header with 896 bits of the payload using ”Stream Mux” block, and finally repack again to 2 bits/byte resulting in the packet ready to modulate and transmit. Of course that this can be optimized by instead repacking into 1 bits, uses directly 2 bits and creating the header with that in mind, but I am splitting in 1 bit because we will use FEC (Forward Error Correction) encoding ahead.

At the RX side we just need to repack to the 2 bits/byte into 1 bit/byte to use the ”Correlate Access Code - Tag Stream” block and extract the payload of the packet and finally repack to 8 bits/byte. Now if you take a look at the output file you see that the bits are aligned correctly in the byte needed.

G. Out-Of-Tree Modules - Tutorial_7.grc

We are getting the bits aligned, but we lose the first packet so we can solve this by transmitting an vector of bits right before we transmit our real data. There is just a big problem:

the GR doesn't has that implemented but has a thing called OOT (Out-Of-Tree Modules) where allow us extend GR with our own functions and blocks created in python or in C++. Now I will teach how to create a block in GR: The first thing is going to an folder where you want keep the OOT blocks and run the following commands:

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

Now we need to chose the type of block we want to write, where:

- 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 GR 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 selected `general`, and then we need to select the language (it can be in Python or C++), I have selected `cpp` as the language, then you need to input the arguments that your block will take, as I want only an vector of bytes I put `const std::vector<unsigned char> &vec`, and finally you select if you want QA (Quality Assurance) code, this is a template where you can do unit testing on your code (I selected no).

Here it's created 3 important things:

```
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 our source code file witch is the second file and where we will write our code. The last one is an XML file where we link our code to the GR.

Let's carefully analyse the source file:

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

The GR adds <+ and +> in places that you need to replace for values that you need. In the first line is where you select how many input connections you need, namely the minimum number, the maximum number, and the size of the type of data 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 like a byte, this means that we need to set an `unsigned char`. The second line is where you select how many output connections you need, namely the minimum number, the maximum number, and the size of the type of data that will be in each connection, so it's exactly the same as before.

Besides that we will need to use our vector in another function (general_work), an flag to know if all the vector was sent and finally an index to track how many bits of the vector we already sent. Resulting in changes in file source and header file, in file source just add this:

```
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 declare the variables:

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

And make getters and setters:

```
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;}
```

The next function to treat in file source is this one:

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

Here is where you define the production/consumption rate you want. We will want 1:1 so you need to add this in that function:

```
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 `general_work()` that will be performed by the block, the template show us this:

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

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 the first time or vector to send then we need to send it, taking into account that we can only send the maximum size available on the output buffer. In case there is not enough space we sent what we can and update the variable to keep track of the last byte sent and next time we know where we will start sending the vector. Here we don't consume anything, we are just producing items. In case there is enough space then we sent the rest of the vector and set the flag to know that we already sent all the vector and starts passing directly the next items. So in `general_work()` we will add this:

```
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 or vector not fully
    sent, so I need to insert the vector
     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's what I will
         output (produce, it can be all
         vector or all buffer)
     ii=0; //I don't 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 I can set the
         flag to get out. (Start passing
         directly)
          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 remaining file we have a XML where we need to set some things so we can link our written code to GR. The first tricky thing is remove the `\&` on the parameter in `<make>` tag. In `<param>` is where we define the parameter that will be parsed in our 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 `<souce>` has a parameter called `<type>` where we set what type of data will be inputted and outputted, in this case will be `byte`.

Now that we have all code written we need compile it and link to our GRC with these commands:

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

Finally we just need to click in "Reload Block" in the toolbar in GR and use the block like any other.

H. Forward Error Correction - LDPC - Tutorial_8.grc

I. Equations

J. Some Common Mistakes

IV. Using the Template

A. Headings, etc

B. Figures and Tables

TABLE I
An Example of a Table

One	Two
Three	Four

V. Conclusions

A conclusion section is not required. Although a conclusion may review the main points of the paper, do not replicate the abstract as the conclusion. A conclusion might elaborate on the importance of the work or suggest applications and extensions.

Appendix

Appendixes should appear before the acknowledgment.

ACKNOWLEDGMENT

The preferred spelling of the word acknowledgment in America is without an e after the g. Avoid the stilted expression, One of us (R. B. G.) thanks . . . Instead, try R. B. G. thanks. Put sponsor acknowledgments in the unnumbered footnote on the first page.

References are important to the reader; therefore, each citation must be complete and correct. If at all possible, references should be commonly available publications.

References

[1] G. O. Young, Synthetic structure of industrial plastics (Book style with paper title and editor), in Plastics, 2nd ed. vol. 3, J. Peters, Ed. New York: McGraw-Hill, 1964, pp. 1564.

[2] W.-K. Chen, Linear Networks and Systems (Book style). Belmont, CA: Wadsworth, 1993, pp. 123135.

[3] H. Poor, An Introduction to Signal Detection and Estimation. New York: Springer-Verlag, 1985, ch. 4.

[4] B. Smith, An approach to graphs of linear forms (Unpublished work style), unpublished.

[5] E. H. Miller, A note on reflector arrays (Periodical styleAccepted for publication), IEEE Trans. Antennas Propagat., to be publised.

[6] J. Wang, Fundamentals of erbium-doped fiber amplifiers arrays (Periodical styleSubmitted for publication), IEEE J. Quantum Electron., submitted for publication.

[7] C. J. Kaufman, Rocky Mountain Research Lab., Boulder, CO, private communication, May 1995.

[8] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, Electron spectroscopy studies on magneto-optical media and plastic substrate interfaces(Translation Journals style), IEEE Transl. J. Magn.Jpn., vol. 2, Aug. 1987, pp. 740741 [Dig. 9th Annu. Conf. Magnetics Japan, 1982, p. 301].

[9] M. Young, The Techincal Writers Handbook. Mill Valley, CA: University Science, 1989.

[10] J. U. Duncombe, Infrared navigationPart I: An assessment of feasibility (Periodical style), IEEE Trans. Electron Devices, vol. ED-11, pp. 3439, Jan. 1959.

[11] S. Chen, B. Mulgrew, and P. M. Grant, A clustering technique for digital communications channel equalization using radial basis function networks, IEEE Trans. Neural Networks, vol. 4, pp. 570578, July 1993.

[12] R. W. Lucky, Automatic equalization for digital communication, Bell Syst. Tech. J., vol. 44, no. 4, pp. 547588, Apr. 1965.

[13] S. P. Bingulac, On the compatibility of adaptive controllers (Published Conference Proceedings style), in Proc. 4th Annu. Allerton Conf. Circuits and Systems Theory, New York, 1994, pp. 816.

[14] G. R. Faulhaber, Design of service systems with priority reservation, in Conf. Rec. 1995 IEEE Int. Conf. Communications, pp. 38.

[15] W. D. Doyle, Magnetization reversal in films with biaxial anisotropy, in 1987 Proc. INTERMAG Conf., pp. 2.2-12.2-6.

[16] G. W. Juette and L. E. Zeffanella, Radio noise currents n short sections on bundle conductors (Presented Conference Paper style), presented at the IEEE Summer power Meeting, Dallas, TX, June 2227, 1990, Paper 90 SM 690-0 PWRS.

[17] J. G. Kreifeldt, An analysis of surface-detected EMG as an amplitude-modulated noise, presented at the 1989 Int. Conf. Medicine and Biological Engineering, Chicago, IL.

[18] J. Williams, Narrow-band analyzer (Thesis or Dissertation style), Ph.D. dissertation, Dept. Elect. Eng., Harvard Univ., Cambridge, MA, 1993.

[19] N. Kawasaki, Parametric study of thermal and chemical nonequilibrium nozzle flow, M.S. thesis, Dept. Electron. Eng., Osaka Univ., Osaka, Japan, 1993.

[20] J. P. Wilkinson, Nonlinear resonant circuit devices (Patent style), U.S. Patent 3 624 12, July 16, 1990.