European Gnu Radio Days 2019:
Writing a custom Gnu Radio processing bloc: (*OOT block*)

Tanguy Risset, Leonardo Cardoso (Insa-Lyon/Inria)

12 juin 2019

Goal of the tutorial

The goal of the tutorial is to understand the methodology to develop new GNU radio blocks in C ++ (we do not explore the creation of Python blocks). These blocks, created by the user, are usually called " Out of tree modules " (OOT). This mechanism makes it possible to define complex blocks written in C ++, which is much more efficient than the equivalent obtained by assembling GRC blocks.

This tutorial is based on tutorials posted by GNU Radio (`https://wiki.gnuradio.org/index.php/Tutorials`). It assumes you have basic Python and C ++ knowledge (Python and C should suffice), and you are familiar with Linux and basic `bash` commands. It also assumes that you know the simple use of `gnuradio-companion`. This tutorial is intended to work with the Adalm Pluto platform, but any other SDR platform you know how to use in GNU Radio will do the trick.

The tutorial is organized as follows :

1. After setting up the development environment, we will create with the `gr_modtool` tool a first OOT simple block that simply computes the squared of a signal (section 4).
2. Then we will create a QPSK demodulation block (section 5).
3. Finally we will see the mechanism of *history*

1 Working environment

You work in the virtual machine offered by the trainers.

You can follow the tutorials on your own machine but in this case :

— Make sure you have a compatible version of GNU Radio, especially that you have installed the drivers for the pluto-SDR module.

— Depending on the linux distribution and especially the GNU Radio version installed, the commands may vary slightly.

2 Initiating working environment

1. Launch Virtual-Box, import the virtual machine if it is not already done (information given at the beginning of the session).

2. If the keyboard is not configured the way you want, the key combination to switch from the keypad to the keypad and vice versa is : `Ctrl-Alt-Shift`. The open account (`user`) is `sudo` and the root password is `root`. You may not be able to access the internet from the virtual machine, but you can set up a shared directory accessible in the virtual machine and on the host machine.
3. The 'user' account has been configured so that the environment variables needed by GNURadio are configured. The `OOT_env.sh` file that makes these settings is available on the training website. FYI, this file declares the directory `$HOME/.local` as a directory where `gnuradio-companion` can find blocks. here is its content :

```
OOT_MODULE_INSTALL_PREFIX=$HOME/.local
export LD_LIBRARY_PATH="$OOT_MODULE_INSTALL_PREFIX/lib:${LD_LIBRARY_PATH}"
export PATH="$OOT_MODULE_INSTALL_PREFIX/bin:${PATH}"
export GRC_BLOCKS_PATH="$OOT_MODULE_INSTALL_PREFIX/share/gnuradio/grc/blocks/:\
${GRC_BLOCKS_PATH}"
```

4. Launch a command window (xterm)
5. Launch Gnuradio companion (command : `gnuradio-companion`),
6. Create a very simple flow graph that generates a sinusoid and displays it with a QT GUI sink sink block.

We will propose the creation of a new simple block that simply raises the signal squared (each sample will be squared). In Gnu Radio, each block belongs to a module, so we will create a module called `gr-tutorial` and a block named `square` in this module.

3 Learning to use `gr_modtool`

Gnu radio offers a tool to help design new modules : `gr_modtool`, this tool will produce a number of files that you modify to fit your needs.

— Type `gr_modtool help` then `gr_modtool help newmod`

Once `gr_modtool` has produced the files for the module and you have specialized them, the module is compiled using the `cmake` tool. The `cmake` command produces a set of tools for compiling a project for different platforms, testing, and creating packages for different systems. It is used in many projects such as KDE and MySQL and is a good alternative to *autotools*.

The principle is that `cmake` proceeds in two steps : in the first step, the files used for building the project (eg Makefile files) are produced, and in the second step these files are executed to build the project. Each built project has a `CMakeLists.txt` file that has its own syntax for expressing executed commands to set up the generated build files.

4 Creation of a module "gr-tutorial" with a simple bloc

We will create a simple block that simply raises a signal squared. We will first create a directory `$HOME/.local` in which we will add our *OOT* block (Out of Tree).

— Create the directory `$HOME/.local`

— Create a working directory for the tutorial for example `$HOME/OOT`

— go to this `OOT` directory. The position of this directory does not matter, the files necessary for GNU Radio will be installed in the directory `$HOME/.local`.

4.1 Creating the module

- create a new module `tutorial` :
  `gr_modtool newmod tutorial`
- Go to the `gr-tutorial` directory, familiarize yourself with the hierarchy.

4.2 Add a block in the module

- Create a block of general type called `square` (command below). **Important :** answer `cpp` for the first question and answer nothing (i.e. type carriage return) for other questions (including : do not enter *anything* when asked to enter a valid argument list) :
  `gr_modtool add -t general square`

- The file `python/qa_square.py` contains the tests that will be done in Python on the new block. Complete this file by creating a test method for your block, for example (you can retrieve this code directory `OOT_lab_files` on the VM) :

```
def test_001_t (self):
    # set up fg
    src_data = (-3, 4, -5.5, 2, 3)
    expected_result = (9, 16, 30.2, 4, 9)
    src = blocks.vector_source_f(src_data)
    sqr = tutorial.square()
    dst = blocks.vector_sink_f()
    self.tb.connect(src, sqr)
    self.tb.connect(sqr, dst)
    self.tb.run()
    # check data
    result_data = dst.data()
    self.assertFloatTuplesAlmostEqual(expected_result, result_data, 6)
```

  Note : `assertFloatTuplesAlmostEqual` is a method from `gr_unittest` which is a standard python extension for testing the *approximative* equality of floating or complex number tuples.
- Since the `qa_square` block already existed, we do not need to modify the file `python/CMakeLists.tx` view the file `python/CMakeLists.txt`.
- create the directory `gr-tutorial/build`, go to this directory
  - The usual command is : `cmake ../`
  - but as we create OOT blocks, we will use the command :
    `cmake -D CMAKE_INSTALL_PREFIX=$HOME/.local ../`
- Once the `cmake` worked without error, do `ls`, just to view the files produced.

4.3 Block compilation

- Try to build it (command `make`), try to understand the error message to know which file to modify for the compilation to work.
- Complete the file `gr-tutorial/lib/square_impl.cc` between the chevrons $<$ and $>$:

— how many ports (1 input, 1 output) of which type (float).

— Complete the `forecast` method according to the proposed comment.

— In the `general_work` method, replace the types between chevron and add the processing performed on each sample (i.e. squared the signal) :

```
for (int i = 0; i <noutput_items; i ++) {
        out [i] = in [i] * in [i];
}
```

— Build the `make` block in the `build` directory

— Test the `make test`, the test result can be viewed : `less Testing/Temporary/LastTest.log`

— debug your block ...

— The `ctest -V` command replaces the make test command : it displays the `LastTest.log` file on the screen. One can thus use print in the python code and make a first debugging of note block.

4.4 Creation of a GRC block for "square"

— Create a `gnuradio-companion` block in the `gr-tutorial` directory :
`gr_modtool makexml square`

— compile it and install (`make` and `make install` in `build`), check that you see it appear in `gnuradio-companion` (it is possible that you have to restart `gnuradio-companion`).

— Create a simple flow graph `grc` with this new block and test that

5 Building a block with a parameter : QPSK receiver

This part is largely inspired by tutorial 5 of the GnuRadio Wiki. QPSK demodulation takes a complex symbol placed on a constellation of 2^n values and returns an integer value of 0 and $2^n - 1$. We are going to realize a block carrying out a QPSK demodulaxtion for $n = 2$ (thus 4 possible values for the constellation). The block will have a parameter that will indicate whether we will use a *gray* encoding (a single bit changing in symbols representing two successive values).

— Add a block of the general type named `my_qpsk_demod`, but this time enter `bool gray_code` when asked for the valid argument list :

```
>gr_modtool add -t general my_qpsk_demod
GNU Radio module name identified: gr-tutorial
[...]
Enter valid argument list, including default arguments: bool gray_code
[...]
```

— run `cmake` :
`cmake -D CMAKE_INSTALL_PREFIX=$HOME/.local ../`

— Complete the file `lib/my_qpsk_demod_impl.cc` in the same way as the `square` block (same function `forecast`), but this time the port type will not be `float`, but `gr_complex` input and `unsigned char` output. Note the presence of the `gray_code` parameter of the `my_qpsk_demod_impl` function.

— Add a class variable `private d_gray_code` in class `my_qpsk_demo_impl` (file `my_qpsk_demo_imp`
and add its initialization to `gray_code` when declaration of the function `my_qpsk_demod_impl()`.
Your constructor should now look like this :

```
my_qpsk_demod_impl :: my_qpsk_demod_impl (bool gray_code)
   : gr :: block ("my_qpsk_demod",
          gr :: io_signature :: make (1, 1, sizeof (gr_complex)),
          gr :: io_signature :: make (1, 1, sizeof (char))),
     d_gray_code (gray_code)
  {}
```

— The `//do signal processing` section will simply consist, for each sample, in a call to the function `qpsk_4_decode(in [i])` for each output sample.

— Add the private function `qpsk_4_decode()`. For example the one proposed in the GNU Radio tutorial below, you can do yours :

```
unsigned char my_qpsk_demod_impl::qpsk_4_decode(const gr_complex &sample)
  {
    if (d_gray_code) {
      unsigned char bit0 = 0;
      unsigned char bit1 = 0;
      // The two left quadrants (quadrature component < 0)
      // have this bit set to 1
      if (sample.real() < 0) {
        bit0 = 0x01;
      }
      // The two lower quadrants (in-phase component < 0)
      // have this bit set to 1
      if (sample.imag() < 0) {
        bit1 = 0x01 << 1;
      }
     return bit0 | bit1;
    } else {
      // For non-gray code, we can't simply decide on signs,
      //so we check every single quadrant.
      if (sample.imag() >= 0 and sample.real() >= 0) {
        return 0x00;
     }
      else if (sample.imag() >= 0 and sample.real() < 0) {
        return 0x01;
     }
      else if (sample.imag() < 0 and sample.real() < 0) {
        return 0x02;
      }
      else if (sample.imag() < 0 and sample.real() >= 0) {
        return 0x03;
      }
    }
  }
```

— Compile your block (and fix the errors) : `make` in the `buid` directory.

— Create the file `grc` associated with your block : in the directory `gr-tutorial` type :
`gr_modtool makexml my_qpsk_demo`

— Because the block has a parameter, automatic generation of the `grc` file is not enough : `gr_modtool` creates a parameter in the `xml` file of your GRC block, but he does not know how you want to use it and what will be his influence on the behavior of the block. Here, we will indicate that the parameter will give the value `True` or `False` to the variable `gray_code`. Edit the `grc` file generated for your block (`grc/tutorial_my_qpsk_demod.xml`), This information must be entered in the `<param>` field of the file `xml` Arrange for this `<param>` field to be :

```
<param>
  <name>Gray Code</name>
  <key>gray_code</key>
  <value>True</value>
  <type>bool</type>
  <option>
    <name>Yes</name>
    <key>True</key>
  </option>
  <option>
    <name>No</name>
    <key>False</key>
  </option>
</param>
```

You can also change the name of the block under `grc` and call it for example : `My QPSK Demodulator`. **Warning**, the `xml` syntax is not very complicated, but **syntax errors are not reported** by GNU Radio, so you have to be very precise for this step. The only way to detect an error is a message when launching `gnuradio-companion` without the name of the offending GRC being mentioned :

`XML parser: Found 1 erroneous XML file while loading the block tree`

— your block must be ready to use now, type `make`, then `make install`. Attention, in case of error in the file `xml grc`, there is a very laconic message and no way to have precisely the error, the only symptom is that your block does not appear from the library `gnuradio_companion` even after restarting `gnuradio_companion`

— Launch`gnuradio_companion` and make sure you see your block `my_qpsk_demod` and the module tutorial

— To test it, open the GRC `OOT_lab_files/test_my_qpsk.grc` (test it once), replace the `my_qpsk_demo_cb` block with your `my_qpsk_demod` and verify that you have the same behavior.

6 Creation of a pipelined block

We will now create an audio filter, `filter_noise` by repeating these same operations, except that at creation the filter will inherit the `gr_sync_block` block to have the method `set_history(nbdata)` which is used to have a sliding window on `nbdata` samples :

`gr_modtool add -t sync filter_noise`

The purpose of this section is to implement a FIR filter that filters out additive noise. The equation of a FIR filter is :

$$y(n) = \sum_{k=0}^{N-1} c_k x(n-k)$$

where y is the signal at the output of the filter and the c_k are the coefficients of the filter. N is the order or length of the filter.

We can first use a simple filter that we declare in the file `taps.h` with the following content for example :

```
const int SFD_size = 8;
const float B[SFD_size] = {
  1.0,
  1.0,
  1.0,
  1.0,
  1.0,
  1.0,
  1.0,
  1.0
};
```

The correction of the `filter_noise.cc` file will be done in the same way as for the `square` block, with the following differences :

- No function `forecast` : the `sync` block consumes as many samples as they produce.
- You have to add two things in the constructor :
  - In the constructor of the `filter_noise_impl` object, you must indicate that you want to use a " history " of `SFD_size`. This is done with the following call : `set_history (SFD_size)`
  - The `general_work` function is now called tt work
  - The first samples that are missing to make the first convolution will be set to 0 by GNURadio.
- Here is the proposed code for the "do_signal_processing" part :

```
      for (int i=0; i<noutput_items; i++)
         {
           float corr=0.;
           for (int j=0;j<SFD_size;j++)
             {
               // in[0][0] is the oldest sample (because of history(SFD_size
               corr+=B[j]*in[i+SFD_size-1-j];
             }
           out[i]=corr;
        }
```

- test your block with these values for the python test :

```
src_data=       (1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)
expected_result=(1,2,3,4,5,6,7,8,8,8,8,8,8,8,8,8)
```