CW 17/18 summary

Alexander Pastor

27.04.2017

Contents

1 Guided GNU Radio Tutorials

1.1 The GNU Radio Companion Interface

Fairly straight forward. At the top is the menu bar, which offers typical file administration services. In the middle on the left-hand-side there is the flowchart area. On the right side are the blocks that we can choose from. In the bottom-left-hand corner is the console output and right of it information about variables and imports.

1.2 Creating a Block Using Python or C++

1.2.1 Basic Terminology

module
A set of blocks.

Out-Of-Tree (OOT) modules
Custom-made, unofficial modules.

block
An abstract device with at least one input and/or output port. There are different types of blocks in GNU Radio. Upon creation of a block we choose a type. Depending on our choice a template is generated. Here's what we can choose from.

general	A basic block, with no fixed relation between input and output items.
source	A block with no input ports.
sink	A block with no output ports.
interpolator	A block with fewer input than output items. The number of output items is a fixed multiple of the input items.
decimator	A block with more input than output items. The number of input times is a fixed multiple of the output items.
hierarchical	A block consisting of blocks, which can be nested. **Note:** This type is sometimes referred to as *hier*

[question] Asking like a heretic: why should I ever use something else than general? Does telling the runtime system for instance the interpolating property of a block boost performance?

item
Abstract entity of arbitrary type and size. Can be data container, such as a vector or tuple. Can also be of "primitive type".

port
Input or output connector. Provides the block's input and output streams.

1.2.2 Python or C++?

Using Python is easier, yet less performant. Since Python is interpreted no compilation is required, therefore prototyping is faster.

SWIG
Another important feature of GNU Radio is the inclusion of SWIG. TL;DR: It glues together our C++ and Python stuff. This is done to allow C++ programmers to program most of the functionality in C++, but still make QA tests with Python for rapid prototyping.

From their website:

> SWIG is an interface compiler that connects programs written in C and C++ with scripting languages such as Perl, Python, Ruby, and Tcl. It works by taking the declarations found in C/C++ header files and using them to generate the wrapper code that scripting languages need to access the underlying C/C++ code. In addition, SWIG provides a variety of customization features that let you tailor the wrapping process to suit your application.

1.2.3 How to Create Modules and Blocks

Blocks are packaged in modules. Now here's how to create a block. For this purpose we use the handy console tool *gr_ modtool*.

```
# Let's keep things tidy.
$ mkdir my_modules
$ cd my_modules

# First we create a module.
# module_name is going to be the name of the module and directory name
# newmod | nm | create
$ gr_modtool create module_name
$ cd gr-module_name

# Then we add a block to the model
# Fill in the prompts accordingly
# BLOCK_TYPE: sync | hier (hierarchical) | decimator | interpolator | sink |
# source | general | tagged_stream | noblock
# PROG_LANG: cpp | python
# QA: none | python | cpp
module_name$ gr_modtool add block_name
```

1.2.4 Implement Block Functionality with Python

Implementing a block's functionality with Python is quite comfortable, because there's almost no clutter:

```
#!/usr/bin/env python2

import numpy
from gnuradio import gr

#notice how the block block_name inherits from gr.sync_block!
class block_name(gr.sync_block):
    def __init__(self, arg1):
        gr.sync_block.__init__(self,
            name="block_name",
            # defining input and output of the block
            in_sig=[numpy.float32],
            out_sig=[numpy.float32])
        self.arg1 = arg1

    # Here we implement the block's function
    def work(self, input_items, output_items):
        in0 = input_items[0]
        out = output_items[0]
        #signal processing here
        #here we multiply the signal with arg1 and add 7
        out[:] = in0*self.arg1 + 7
        return len(output_items[0])
```

1.2.5 Implement Block Functionality with C++

The OOT block constructor calls:

```
gr::block(const std::string &name,
          gr::io_signature::sptr input_signature,
          gr::io-signature::sptr output_signature)
```

In our case the block takes input arguments of type boolean and unsigned integer. The edited code template looks like the following:

```
block_name_impl::block_name_impl(bool arg1, unsigned int arg2) :
gr::block("block_name",
          gr::io_signature::make(MIN_IN, MAX_IN, sizeof(ITYPE)),
          gr::io_signature::make(MIN_OUT, MAX_OUT, sizeof(OTYPE))),
    d_arg1(arg1),
    d_arg2(arg2)
{}
```

Note: Programming in Python let's us get sloppy. Don't forget to add `d_arg1` and `d_arg2` as private class members to the header file.

Where MIN_IN, MAX_IN, MIN_OUT, MAX_OUT represent the minimum and maximum number of input and output ports respectively. These must be replaced with integers by the programmer. ITYPE and OTYPE represent the data type of the input and output respectively.

For a **sync** block MIN_IN, MAX_IN, MIN_OUT, MAX_OUT must have the same value.

For a **sink** MIN_OUT, MAX_OUT and OTYPE are 0. For a **source** MIN_IN, MAX_IN and ITYPE are 0, analogous to the sink implementation.

The `general_work()` function is purely virtually inherited from `gr::block`. The block's functionality is stuffed inside of it and because of its purely virtual nature it must be always implemented by the programmer.

[confirmation required] What is the difference between `work()` and `general_work()`?

I see

```
def work(self, input_items, output_items)
```

only in Python code and

```
virtual int general_work(int noutput_arguments,
                         gr_vector & ninput_items,
                         gr_vector_const_void_star & input_items,
                         gr_vector_void_star & output_items)
```

only in `C++` code. Thus, I assume they are functionally equivalent.

[question] What does the `d` in `d_gray_code(gray_code)` (Guided Tutorial GNU Radio in C++ p.3 @ bottom) stand for?

1.3 Testing and Installing Your Module

1.3.1 Code Quality Assurance (QA)

If we create the block with C++ we have the additional option to do the QA with C++.

The Python way (recommended by me):

In order to assure quality we can use built-in and auto-generated files located in `python/qa_block_name.py`. These contain so-called tests.

A test is conducted by passing a set of input values to the test environment and subsequent comparison of expected with actual results. Here's an exemplary Python script.

```
#!/usr/bin/env python2

from gnuradio import gr, gr_unittest
from gnuradio import blocks
import module_name_swig as module_name

class qa_block_name (gr_unittest.TestCase):

    def setUp (self):
        self.tb = gr.top_block ()

    def tearDown (self):
        self.tb = None

    def test_001_t (self):
        # testing whether block correctly calculates Fibonacci numbers
        # sample data
        src_data = (0, 1, 2, 3)
        expected_result = (1, 1, 2, 3)
        src = blocks.vector_source_i(src_data)
        block = block_name("fibs")
        snk = blocks.vector_sink_i()
        #connecting to a multiportblock has the following signature:
        #connect(src, port, dst, port)
        self.tb.connect(src, block)
        self.tb.connect(block, snk)
        self.tb.run()
        res = snk.data()
        self.assertFloatTuplesAlmostEqual(expected_result, res)

if __name__ == '__main__':
gr_unittest.run(qa_block_name, "qa_block_name.xml")
```

If our test went OK, we will get output similar to this:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.003s

OK
```

Otherwise something like this:

```
FAIL: test_001_t (__main__.qa_block_name)
----------------------------------------------------------------------
Traceback (most recent call last):
File "qa_block_name.py", line 44, in test_001_t
```

```
self.assertFloatTuplesAlmostEqual (expected_result, result_data, 6)
File "/usr/lib/python2.7/site-packages/gnuradio/gr_unittest.py",
line 90, in assertFloatTuplesAlmostEqual
self.assertAlmostEqual (a[i], b[i], places, msg)
AssertionError: 2 != 5.0 within 6 places

----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

Note: We can create as many tests as we want, but we need to keep the `test_` prefix.

Note: If we compare two float (tuples) we need to use the `assertFloatsAlmostEqual(a,b)` (`assertFloatTuplesAlmostEqual(a,b)`) function.

[question] What if I want to compare integer tuples? While the function comparing floats works it might be an overkill. I assume Python's "native" (you have to `import unittest`) assertions will suffice, however with my lowly Python skills I couldn't quickly find a way to make it work.

The `C++` way:

The `C++` QA-files are located under `lib/qa_block_name.cc`. The concept is similar to Python testing, but requires compilation.

Since there's no tutorial for QA with `C++` and testing with `C++` seemed to be clumsy I skipped this approach for now. I will come back to this later.

However, here are the skeletons:

qa_block_name.h

```
#ifndef _QA_BLOCK_NAME_H_
#define _QA_BLOCK_NAME_H_

#include <gnuradio/attributes.h>
#include <cppunit/TestSuite.h>

//! collect all the tests for the gr-filter directory

class __GR_ATTR_EXPORT qa_block_name
{
    public:
    //! return suite of tests for all of gr-filter directory
```

```
    static CppUnit::TestSuite *suite();
};

#endif /* _QA_BLOCK_NAME_H_ */
```

qa_block_name.cc

```
#include "qa_block_name.h"

CppUnit::TestSuite *
qa_tutorial_multiply::suite()
{
    CppUnit::TestSuite *s = new CppUnit::TestSuite("block_name");

    return s;
}
```

Let's recite the steps of setting up a test:

1. setting up some source data
2. specifying expected results
3. feeding the source data into a source
4. invoke block (if necessary with input parameters)
5. creating a sink for the block's output
6. connect the source and the block
7. connect the block and the sink
8. run the test
9. store the sink's data into a variable
10. assert that actual and expected result are the same

[question] Why is there a `lib/qa_module_name.cc` but no equivalent for Python QA? What's the point of this file and how to make use of it anyway? Is this just supposed to be a wrapper enabling you to encapsulate the tests of all the module's blocks into a single file?

1.3.2 Installing Block to GRC

To prepare for installation we first have to edit a XML file. In it the block's graphical interface is defined. Here's how the XML file of our block `block_name`. (The location of our file is `grc/module_name_block_name.xml`)

The following is achieved with this XML file: We will get a block `block_name` in the category `[module_name]` in GRC. The block will have a parameter `arg1 name string`, accessible through `$arg1` with the options `yep.` (value True) and `nope.` (value False). The block's input port has the name `in` and is of type byte. The output port has the name `out` and is of type complex.

```
<?xml version="1.0"?>
<block>
    <name>block_name</name>
    <key>module_name_block_name</key>
    <category>[module_name]</category>
    <import>import module_name</import>
    <make>module_name.block_name($arg1, $arg2)</make>
    <!-- Make one 'param' node for every Parameter you want settable from the GUI.
    Sub-nodes:
    * name
    * key (makes the value accessible as $keyname, e.g. in the make node)
    * type -->
    <param>
        <name>arg1 name string</name>
        <key>arg1</key>
        <type>bool</type>
        <option>
            <name>yep.</name>
            <value>True</value>
        </option>
        <option>
            <name>nope.</name>
            <value>False</value>
        </option>
    </param>

    <!-- Make one 'sink' node per input. Sub-nodes:
    * name (an identifier for the GUI)
    * type
    * vlen --- vector length
    * optional (set to 1 for optional inputs) -->
    <sink>
        <name>in</name>
        <type>byte</type>
    </sink>
```

```
    <!-- Make one 'source' node per output. Sub-nodes:
    * name (an identifier for the GUI)
    * type
    * vlen
    * optional (set to 1 for optional inputs) -->
    <source>
        <name>out</name>
        <type>complex</type>
    </source>
</block>
```

After we edited the XML file, we install it via the following commands:

```
# Getting to the parent directory of grc
cd ..
mkdir build
cd build
cmake ../
make
sudo make install
sudo ldconfig
```

To uninstall a block we simply type the following:

```
sudo make uninstall
```

1.4 Advanced Topics

1.4.1 `set_history()`

`void set_history(unsigned int N)` provides the current and the previous N-1 items to the input buffer, despite already having been consumed.

1.4.2 `set_output_multiple()`

`void set_output_multiple(unsigned int N)` allows us to make sure the output has a minimum block size of N. However, note that we can also get output blocks with the size of multiples of N, such as 2N or 3N.

1.4.3 Blocks, Blocks, Blocks!

Just a question here for brevity's sake...

[question] (Guided Tutorial GNU Radio in `C++`) The terminology of GNU Radio is a little bit confusing when it comes to rate changing blocks with a fixed rate. While this seems like a contradiction I see two ways of resolving it. The more likely one goes as follows: since such a block inherits from `gr_sync_decimator`

or `gr_sync_interpolator` the decimation/interpolation rate is variable, otherwise the sync prefix does not make any sense. The other possibility is to have the input and output rate change, but keeping their aspect ratio, specified as decimation or interpolation factor.

1.4.4 Polymorphic Types, Stream Tags and Message Passing

Data between blocks is passed in a stream, which makes sense if data is continuously produced and/or consumed. However, if you want to once-only pass specific information, e.g. to change variable values keeping up a continuous sparsely populated stream is wasteful. The solution to this problem are synchronous stream-tagging or asynchronous message passing.

Another problem is that a message can assume all kinds of data types. This is not a problem for the weakly typed Python, but problematic for the strongly typed `C++`. The solution to this are Polymorphic Types (PMT).

PMTs Basically making `C++` compatible to weak typing. We simply use the polymorphic (could rephrase with universal or weak) type `pmt::pmt_t`. Typecasting non-polymorphic types to PMT is easy, we simply use the commands of the following structure:

```
pmt::pmt_t Poly = pmt::from_complex(gr_complex(0,1));
```

Alternatively, we can use the following pattern in `C++` / Python:

```
pmt::pmt_t P = pmt::mp(5);
```

```
P_int = pmt.to_pmt(42)
```

Strings have a special role in PMTs. They are converted to the PMT symbol.

```
pmt::pmt_t string = pmt::string_to_symbol("content");
```

Useful PMT constants

```
pmt::pmt_t P_true = pmt::PMT_T;
pmt::pmt_t P_false = pmt::PMT_F;
pmt::pmt_t P_nil = pmt::PMT_NIL;
```

Stream Tags If two blocks are already connected through a stream, so-called tags can be passed along with the stream at fixed positions and consist of:

- a key, which serves as identifier
- a value, which can be any PMT

- (optionally) a source ID, which identifies the source of the tag.

Implementation in C++:

```
if(condition)
{
    add_item_tag(0, //port
                 nitems_written(0) + i, //offset - pos not fixed
                 pmt::mp("tag key"), //key
                 pmt::from_double(in[i]) // value
    );
    opt_var = opt_val;
}
```

Implementation in Python:

```
if condition:
    self.add_item_tag(0, #port
        self.nitems_written(0) + i, #offset
        pmt.intern("tag_key"), #key
        pmt.from_double(numpy.double(in0[i])) #value
    )
    self.opt_var = opt_val
```

Tag propagation policies:

- TPP_ALL_TO_ALL
- TPP_ONE_TO_ONE
- TPP_DONT

Msg Passing Works very different from stream tags. There is no offset and no key and a special message port designated for message passing.

Note: Multiple message output ports can be connected to a single input port. This is not possible with stream tags.

Add this to the block's constructor to create message ports and handler:

```
// Message port
message_port_register_in(pmt::mp("in_port_name"));
message_port_register_out(pmt::mp("out_port_name"));
// Message handler
set_msg_handler(
    // This is the port identifier
    pmt::mp("in_port_name"),
```

```
    // [FIXME class name] Bind the class method
    boost::bind(&block_class_name::msg_handler_method, this, _1)
);
```

The same for Python:

```
# Put this into the constructor to create message ports
self.message_port_register_in(pmt.intern("in_port_name"))
self.message_port_register_out(pmt.intern("out_port_name"))
# No bind necessary, we can pass the function directly
self.set_msg_handler(pmt.intern("in_port_name"), self.msg_handler_method)
```

Connecting message ports works analogous to the connecting other ports. The only difference: the function's name is `msg_connect`.

Example passing "message" every second to a debug block:

```
tb = gr.top_block()
src = blocks.message_strobe(pmt.to_pmt("message"), 1000)
dbg = blocks.message_debug()
tb.msg_connect(src, "pdus", dbg, "print")
```

2 Good to Know...

2.1 Python

- `[:]` means all entries of an array in Python.
- Python assert methods
- Numpy data types
- Python data types

2.2 LaTeX

I learned...

- ... how to create lists and tables.
- ... list types include *itemize*, *enumerate* and *description*
- ... how to highlight code using minted, stumbling upon the concept of shell-escaping.
- ... how to use the *hyperref* package to create internal and external links.
- ... that the set `\hyperlink{target}{text}`, `\hypertarget{target}{text}` is used to create internal links.

- `\href{target}{text}` is used to create external links.
- ... of a nice little tool to create tables online.
- ... that you need to compile twice before the TOC is updated.

2.3 Git - Executive Summary

Creating the repository:

```
#first open directory whose contents are to be controlled
#then
$ git init
$ echo "# Caption" >> README.md
$ git add .
$ git commit -m "initial commit"
$ git remote -v
$ git remote rm origin
$ git remote add origin https://github.com/tangboshi/bachelor_thesis.git
$ git push -u origin master
```

Pulling from the remote repository:

```
$ git remote -v
$ git fetch origin
```

Pushing to the remote repository:

```
$ git remote -v
# inspect remote branch
$ git remote show origin
$ git push -u origin master
```

Renaming and removing the remote repository:

```
$ git remote rename origin name
$ git remote remove name
```

2.4 Miscellaneous

- The difference between `.bashrc` and `.bash_profile`.
- `python` refers to Python 2 on virtually any Linux, except for my Arch Linux where it refers to Python 3. GNU Radio uses Python 2, thus one has to use `python2`.
- I was used to the file extension .cpp for `C++` files, but .cc is okay, too and is waaaay faster to type ;-) !