GNU Radio Programming

TAs Winter 2021 : Michalis Raptakis , Eleftheria Plevridi
csdp1250@csd.uoc.gr, plevridi@csd.uoc.gr
Computer Science Department, University of Crete

Extending GNU Radio

- GNU Radio can be extended with additional functionality with two ways:
    1. In-tree development
    2. Out-of-tree (OOT) modules
- In the class we will develop a custom OOT module

GNU Radio OOT-modules

- OOT modules are GNU Radio components that do not belong to the GNU Radio source tree

Advantages:

- Easily maintained by individual developers
- Easy installation of multiple OOT modules
- Small and fast compilation units
- Seamless integration in the GRC

CGRAN (*http://cgran.org*) is a place with public available GNU Radio OOT modules.

Writing the first OOT module

- In order to create an OOT module some files and folders should be created
- The process is automated and the necessary files are produced by the **gr_modtool** tool
- To create a new OOT module with the name **ta_module** execute:

```
$ gr_modtool newmod ta_module
```

- This will create the module folder with name **gr-ta_module**

OOT module structure

- Each OOT module consists from a set of folders

- Necessary are the folders:
    - **include:** contains the public interfaces of the module classes and blocks
    - **lib:** contains the implementation files of the module classes and blocks. Can also contain private module classes
    - **grc:** includes **.yml** files that are used from GRC to provide a graphical representation of a block
    - **swig:** contains necessary files for the construction of the C++ to Python interface
    - **python:** contains blocks written in Python and/or files for the proper organization of the C++ to Python interface

GNU Radio block types

Depending the ratio between the items that a block consumes and produces, all possible blocks fall under 4 categories:

- Synchronous blocks (1:1)
- Interpolation blocks (1:N)
- Decimation blocks (N:1)
- Basic Blocks (M:N)

Synchronous Blocks (1:1)

- Blocks that consume and produce equal amount of items per port
- Easy to write, easy to understand
- In most cases, synchronous blocks are used
- If a synchronous block has zero inputs, is called **Source**
- If a synchronous block has zero outputs, is called **Sink**
- Developers should override the **work()** method

Interpolation Blocks (1:N)

- Similar with the synchronous blocks
- Fixed input-output ratio
- For every input item, N output items in each port are produced
- Developers should specify input-output ratio at the block constructor
- Developers should override the `work()` method

Decimation Blocks (N:1)

- Fixed input-output ratio
- For every N input item, 1 output item in each port is produced
- Developers should specify input-output ratio at the block constructor
- Developers should override the `work()` method

Basic Blocks (M:N)

- Arbitrary input-output ratio at any time instance of the program
- Developers should specify both the number of items consumed and produced manually
- Developers should override the general_work() method
- Great flexibility, hard to understand and develop
- All other block types are derived from the basic block

IO Signatures

Each GNU Radio block at the constructor should provide:

- The number of input ports
- The number of output ports
- The size of the item at the corresponding port

The above are also known as the IO signature of the block

IO Signatures

Example 0: Declare exactly 2 ports, with complex numbers as items

```
gr::io_signature::make(2, 2, sizeof(gr_complex));
```

Example 1: Declare exactly 2 ports, the first with float items and the second with items consisting form 64 complex numbers

```
gr::io_signature::make2(2, 2, sizeof(float),
                        64 * sizeof(gr_complex));
```

Items Processing

Until now, we saw how to create blocks. But:

- How items from input ports are processed?
- Who is responsible to feed the block with items?
- How the processed items are propagated at the output ports?

GNU Radio scheduler is responsible to activate each block, depending if the requirements of the input and output ports are satisfied. This means if the previous blocks have produced enough items and their is space to write the output items. If this holds, GNU Radio scheduler automatically executes the **work()** or **general_work()** method

The work() method

```
int
work(int noutput_items,
     gr_vector_const_void_star &input_items,
     gr_vector_void_star &output_items);
```

- **noutput_items:** The number of output items that this invocation can produce. Due to the fixed rate of input-output of synchronous, interpolation, decimation blocks the number of available input items can be easily retrieved
- **input_items:** Vector of input buffers, where each element corresponds to an input port
- **output_items:** Vector of output buffers, where each element corresponds to an output port
- **Returns** the number of items produced at each port

The general_work() method

```
int
general_work(int noutput_items,
       gr_vector_int &ninput_items,
       gr_vector_const_void_star &input_items,
       gr_vector_void_star &output_items);
```

- **noutput_items:** The number of output items that this invocation can produce
- **ninput_items:** Vector with the available input items at the corresponding input port
- **Returns** the number of items produced at each port
- Developers **MUST** call **consume()** or **consume_each()** to inform the scheduler the number of consumed items per port

Retrieve IO ports buffers

In order to retrieve the buffer of the corresponding port just perform a proper typecast from the *void ** pointer:

```
int
work(int noutput_items,
     gr_vector_const_void_star &input_items,
     gr_vector_void_star &output_items)
{
    /* Get the inputs declared at the constructor io signature */
    const float *in0 = (const float *) input_items[0];
    const gr_complex *in1 = (const gr_complex *) input_items[1];

    /* And the output port */
    gr_complex *out = (gr_complex *) output_items[0];
    ...
}
```

Access input and output items

To access items inside a buffer use standard C memory access methods

```
int
work(int noutput_items,
     gr_vector_const_void_star &input_items,
     gr_vector_void_star &output_items)
{
    /* Get the inputs declared at the constructor io signature */
    const float *in0 = (const float *) input_items[0];
    const gr_complex *in1 = (const gr_complex *) input_items[1];

    /* And the output port */
    gr_complex *out = (gr_complex *) output_items[0];

   /* We want to propagate the complex items only if the
    * the corresponding float input items is greater than 0.
    * Otherwise the output should be 0
    */
   memset(out, 0, noutput_items * sizeof(gr_complex));
   for(int i = 0; i < noutput_items; i++){
       if(in0[i] > 0){
           out[i] = in1[i];
       }
   }
   /* We processed all items */
   return noutput_items;
}
```

Import block to GRC

- To graphically import a block at the GRC the corresponding **.yml** file should be written

- The **.yml** file provides info about:
    - Block name
    - Parameter values
    - Number and type of IO ports
    - Public setter and getter methods

Build system

- GNU Radio and OOT modules use the CMake build system
- CMake is a tool that automatically produces Makefiles
- Keeps build files separately from the source code
- To build and install the OOT module:
  - *cd project_dir*
  - *mkdir build* (this is necessary only once)
  - *cd build* (this is necessary only once)
  - *cmake* .. (only if you change any of the CMakeLists.txt file)
  - *make*
  - *make install*
  - *make uninstall* (if you want to uninstall)
- After that the new OOT module should be available at the GRC

Creating a block from the beginning

- The following slides will guide you to create a new block inside the gr-ta_module
- The block takes as argument a threshold and a complex input. If the real or the imaginary part is greater than the threshold or less than the -threshold it outputs the threshold or the -threshold respectively. Otherwise the number itself.

Creating a block from the beginning

- Go inside the directory of the OOT module of the class (gr-ta_module)
- Create a block with name *complex_clamp* using the gr_modtool tool
- *gr_modtool add complex_clamp*
- When asked choose **sync** block as the block type and **cpp** for the implementation language
- Provide the a float parameter for the threshold
- For now, skip any QA related files

Creating a block from the beginning

- As this block does not provide any setter and getter, there is no need to change the *include/ta_module/complex_clamp.h* file
- In the *lib/complex_clamp_impl.h* make the appropriate private fields declarations

Creating a block from the beginning

```
class complex_clamp_impl : public complex_clamp
{
/*
 * Because work() is a method, after the end of its invocation all local variables
 * are lost. So we use private class variables to keep the necessary state
 */
private:
    const float d_threshold;

public:
    complex_clamp_impl(const float threshold);
    ~complex_clamp_impl();

    // Where all the action really happens
    int
    work(int noutput_items, gr_vector_const_void_star &input_items,
         gr_vector_void_star &output_items);
};
```

Creating a block from the beginning

In the *lib/complex_clamp_impl.cc* define the IO signatures and the constructor of the block

```
complex_clamp_impl::complex_clamp_impl(const float threshold)
  : gr::sync_block("complex_clamp",
          /* The block has exactly one complex input */
          gr::io_signature::make(1, 1, sizeof(gr_complex)),
          /* The block has exactly one complex output */
          gr::io_signature::make(1, 1, sizeof(gr_complex))),
          /* Initialize private members */
          d_threshold(threshold)
{}
```

Creating a block from the beginning

In the *lib/complex_clamp_impl.cc* provide the implementation of the **work()** method, where the real processing is performed

```
int
complex_clamp_impl::work(int noutput_items,
                         gr_vector_const_void_star &input_items,
                         gr_vector_void_star &output_items)
{
    int i;

    /*Get the input items. NOTE: No modification is allowed on them*/
    const gr_complex *in = (const gr_complex *) input_items[0];
    gr_complex *out = (gr_complex *) output_items[0];
    ...
}
```

Creating a block from the beginning

In the *lib/complex_clamp_impl.cc* provide the implementation of the **work()** method, where the real processing is performed

```
     for(i = 0; i < noutput_items; i++){
        out[i] = in[i];
        if(in[i].real() > d_threshold){
            out[i].real(d_threshold);
        }

        if (in[i].imag() > d_threshold) {
            out[i].imag(d_threshold);
        }

        if(in[i].real() < -d_threshold){
            out[i].real(-d_threshold);
        }

        if (in[i].imag() < -d_threshold) {
            out[i].imag(-d_threshold);
        }
    }
    // Tell runtime system how many output items we produced.
    return noutput_items;
}
```

Creating a block from the beginning

Now edit the **.yml** file for the GRC that *gr_modtool* created for you

```
id: tutorial_complex_clamp
label: complex_clamp
category: '[ta_module]'

templates:
  imports: import ta_module
  make: ta_module.complex_clamp(${threshold})

#  Make one 'parameters' list entry for every Parameter you want settable from the GUI.
#     Sub-entries of dictionary:
#     * id (makes the value accessible as \$keyname, e.g. in the make entry)
#     * label
#     * dtype
parameters:
- id: threshold
  label: Threshold
  dtype: float
.
.
.
```

Creating a block from the beginning

```
id: ta_module_complex_clamp
label: complex_clamp
category: '[ta_module]'
```

- The **id** should be unique
- **label** is the the string with the block name, displayed at GRC
- You can change the label of the block to something nicer. Eg:

```
id: ta_module_complex_clamp
label: Complex Clamp
category: '[ta_module]'
```

Creating a block from the beginning

- At the **parameters** section add an entry for every parameter of your block
- **id** is the unique identifier of the parameter
- **label** is the string with the parameter name that GRC will display to the user
- **dtype** specifies the type of the variable. Can be *complex*, *float*, *short*, *char*, *int* or *raw*

```
parameters:
- id: threshold
  label: Threshold
  dtype: float
```

Creating a block from the beginning

- Now we should specify the input and output ports of the block
- **domain** can be either *stream* or *message* for message passing ports
- **dtype** specifies the type of the port. Can be *complex*, *float*, *short*, *char* or *int*

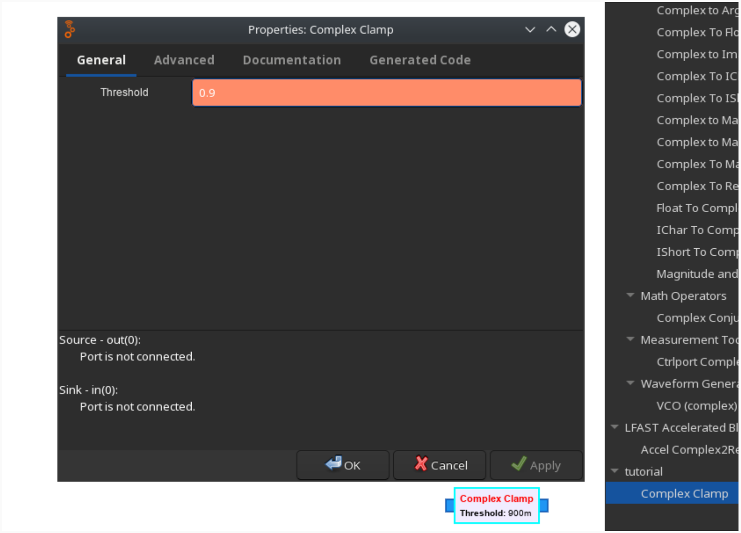
```
inputs:
- label: in
  domain: stream
  dtype: complex

outputs:
- label: out
  domain: stream
  dtype: complex
```

Creating a block from the beginning

- Now it is time to compile and install the module!
- Follow the build and install instructions located at the README of the **gr-tutorial** module
- Close and open GRC or use the **Reload** button in order the new changes to take effect

Creating a block from the beginning



Properties: Complex Clamp

General | Advanced | Documentation | Generated Code

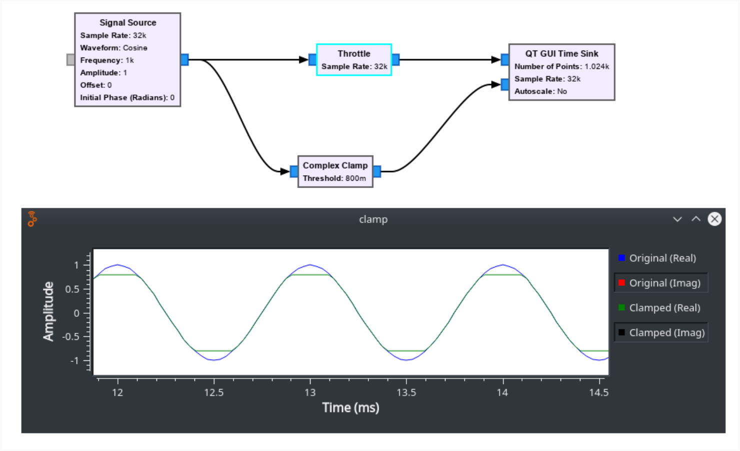
Threshold: 0.9

Source - out(0):
    Port is not connected.

Sink - in(0):
    Port is not connected.

Creating a block from the beginning

Use the *examples/clamp.grc* flowgraph to test the result!



Signal Source
Sample Rate: 32k
Waveform: Cosine
Frequency: 1k
Amplitude: 1
Offset: 0
Initial Phase (Radians): 0
Throttle
Sample Rate: 32k
QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 32k
Autoscale: No
Complex Clamp
Threshold: 800m
clamp
Amplitude
1
0.5
0
-0.5
-1
12
12.5
13
13.5
14
14.5
Time (ms)
Original (Real)
Original (Imag)
Clamped (Real)
Clamped (Imag)