Wireless Communication with GNU Radio

Zelin Yun and Shengli Zhou @ UConn

Abstract

May 20, 2022
Guidebook Version 0.1
Ubuntu 20.04 with GNU Radio 3.9
Ubuntu 22.04 with GNU Radio 3.10
C++ block, C++ block with parameter, Python block with parameter

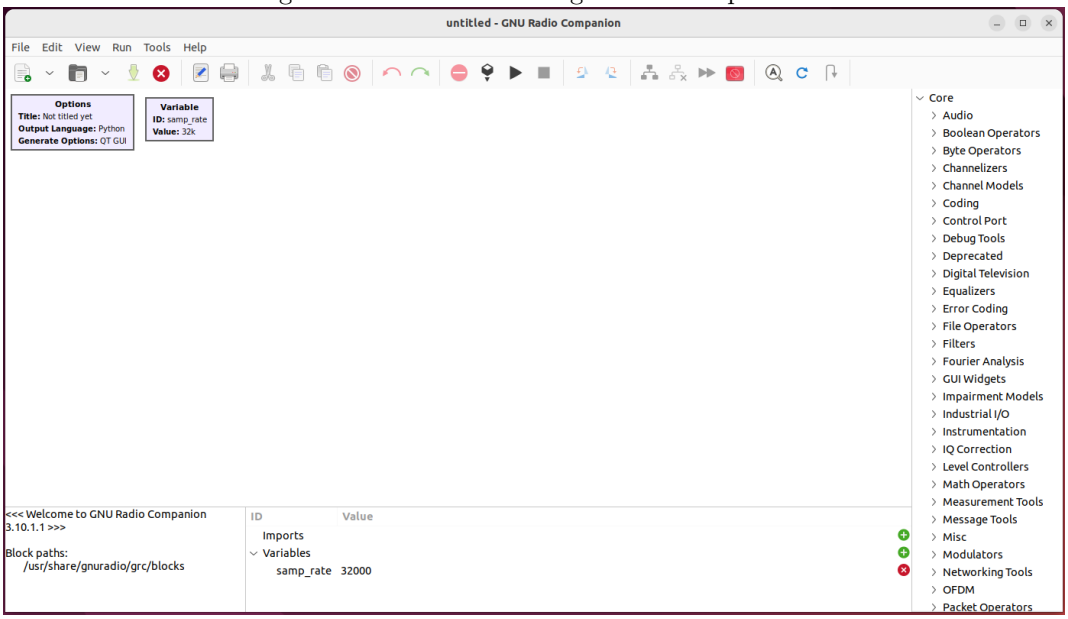
1 Introduction

This document gives a guide about using the GNU Radio and other equipment and tools to achieve the physical layer and even data link layer of a wireless communication stack. In this version, we provide two environments which are Ubuntu 20.04 with GNU Radio 3.9.5 and Ubuntu 22.04 with GNU Radio 3.10.

2 GNU Radio Installation

To use GNU Radio for wireless communication, there are two major components, one is the GNU Radio software, and the other one is the digital-to-analog converter (DAC) and analog-to-digital converter (ADC) which can be speaker and microphone for audio signal or Universal Software Radio Peripheral (USRP) for radio frequency signal. Here we use USRP as the example. For transmissions, GNU Radio generates the signal samples which are some float or complex number arrays and pushes the samples into the USRP which converts them into electromagnetic waves emitted from the antenna. For receptions, the USRP converts the electromagnetic waves into sample arrays which will be processed in GNU Radio and the transmitted information is recovered.

Figure 1: User interface of gnuradio-companion.



After install the Ubuntu, run the following commands in the terminal. The installation steps are from [GR].

For Ubuntu 20 with GR 3.9:

```
$ sudo apt-get update
$ sudo apt-get install git vim cmake build-essential
$ sudo apt-get install uhd-host
$ sudo cp /usr/lib/uhd/uhd-usrp.rules /etc/udev/rules.d/
$ sudo udevadm control --reload-rules
$ sudo udevadm trigger
$ sudo uhd_images_downloader
$ sudo add-apt-repository ppa:gnuradio/gnuradio-releases-3.9
$ sudo apt-get update
$ sudo apt-get install gnuradio python3-packaging
```

For Ubuntu 22 with GR 3.10:

```
$ sudo apt-get update
$ sudo apt-get install git vim cmake build-essential
$ sudo apt-get install uhd-host
$ sudo cp /usr/lib/uhd/utils/uhd-usrp.rules /etc/udev/rules.d/
$ sudo udevadm control --reload-rules
$ sudo udevadm trigger
$ sudo uhd_images_downloader
$ sudo apt-get install gnuradio-dev python3-packaging
```

The above commands first installed some tools and then install mainly two components, one is the UHD which is the driver of the USRP and the other one is GNU Radio. The commands from line 4 to 6 enable the USB permission for USRP when it is plugged in. The command **uhd_images_downloader** downloads the firmware for different types of USRP. After the installation, run this command to run the GNU Radio.

```
$ gnuradio-companion
```

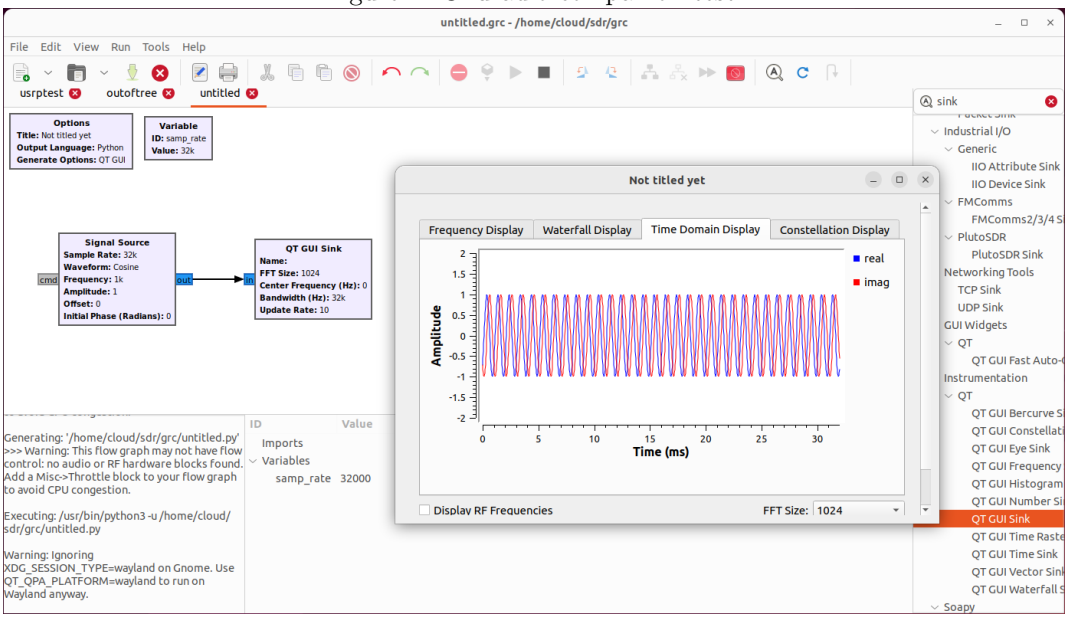
You should see GNU Radio is opened as shown in Figure 1. Next, we test the GNU Radio and USRP. First, build a folder to keep our files. Open another terminal and use the following commands:

```
$ cd
$ mkdir sdr
$ cd sdr/
$ mkdir grc
```

With the above commands, we first build a folder named **sdr** in the home path to keep all the GNU Radio related files. In the **sdr** folder, we build another **grc** folder to only keep the graphic programs of GNU Radio. To test the GNU Radio, in the **gnuradio-companion** we grab a block from the right side tool box named **Signal Source** (in the **Core/Waveform Generator**, you can also use the search function on the panel or press Ctrl+F), and the other block is **QT GUI Sink** (in the **Core/Instrumentation**). Now connect the output of **Signal Source** to the input of **QT GUI Sink**. To run the program, first save it into the **sdr/grc/** folder. And you could see this running in Figure 2:

To test the USRP, connect an antenna to the RX2 of the USRP and plug the USRP into the computer through the USB cable and run the example in Figure 3. We use a new program file, grab a **UHD: USRP Source** and connect it to a **QT GUI Sink**. We double click the UHD block and go to the **RF Options**, set Center Freq as 2412000000, set Gain Value as 0.7, set Gain Type as Normalized, set Antenna as RX2 and Bandwidth as 20000000. Also, in the program, there is a Variable block with ID: samp_rate, we change the Value as 20000000. Now we run it, you should see the USRP is receiving some noise as shown in Figure 3 or if you are lucky, you could see some WiFi signal.

Figure 2: Gnuradio-companion test.



3 Out-of-Tree Modules

This section introduces the steps to create a new module and a new block. First we go to the **sdr** folder.

```
$ cd
$ cd sdr/
$ gr_modtool newmod oot
```

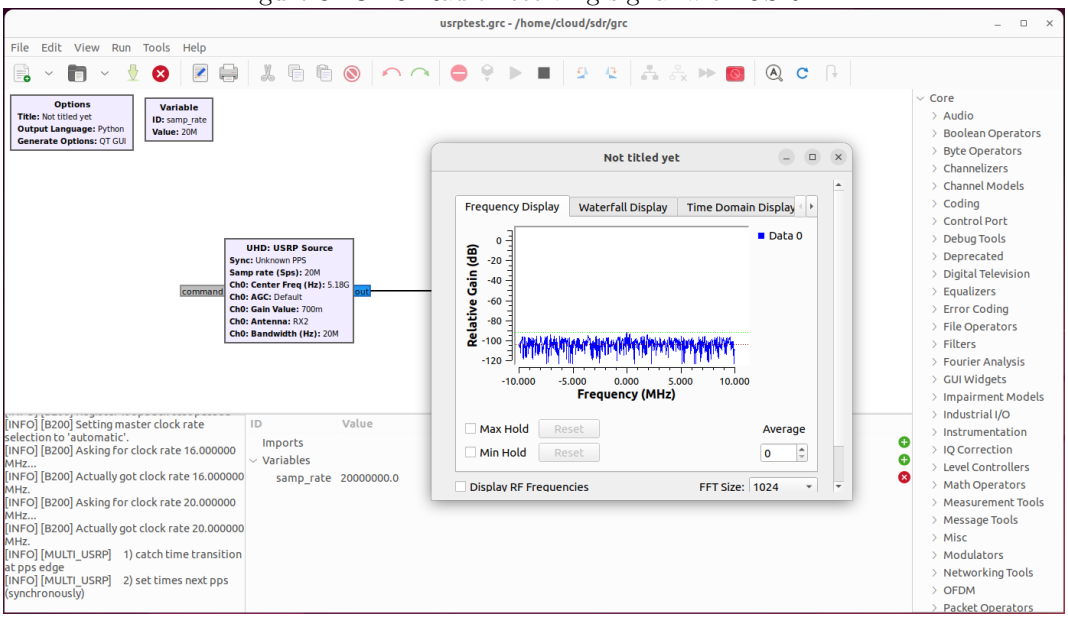
The module **oot** means the out-of-tree modules not from the GNU Radio itself. After that you can find that in **sdr** folder there is a new folder named **gr-oot**. This folder is generated by the gr_modtool and contains all files of the new module. But this module currently has no blocks to use, we now create a new block.

3.1 C++ Version

```
$ cd gr-oot/
$ gr_modtool add -t general -l cpp multiply
```

After the command, it requires you to enter the copyright holder where you can decide for your own codes. Following are parameters and QA codes, we can just press enter to skip. By these commands, we create a new block in the module **oot**. This block has the C++ codes to achieve the function of the block and the codes are kept in the **lib** folder, and the YAML codes for the graphic interface kept in **grc** folder. In this example, we make this block enlarge the amplitude if the input signal by 2 and outputs it. First, we go to the **lib** folder and can see two files **multiply_impl.cc** and **multiply_impl.h**. In this example we do not have to modify the header file, for the C++ file we modify it like this:

Figure 3: GNU Radio receiving signal with USRP .




```
#include <gnuradio/io_signature.h>
#include "multiply_impl.h"

namespace gr {
  namespace oot {
    multiply::sptr
    multiply::make()
    {
      return gnuradio::make_block_sptr<multiply_impl>(
        );
    }

    multiply_impl::multiply_impl()
      : gr::block("multiply",
              gr::io_signature::make(1 /* min inputs */, 1 /* max
                  inputs */, sizeof(float)),
              gr::io_signature::make(1 /* min outputs */, 1 /*max
                  outputs */, sizeof(float)))
    {}

    multiply_impl::~multiply_impl()
    {}

    void
    multiply_impl::forecast (int noutput_items, gr_vector_int &
        ninput_items_required)
    {
      gr_vector_int::size_type ninputs = ninput_items_required.size();
      for(int i=0; i < ninputs; i++)
      {
              ninput_items_required[i] = noutput_items;
      }
    }
```

```
31
32      int
33      multiply_impl::general_work (int noutput_items,
34                                   gr_vector_int &ninput_items,
35                                   gr_vector_const_void_star &input_items,
36                                   gr_vector_void_star &output_items)
37      {
38        const float *in = static_cast<const float*>(input_items[0]);
39        float *out = static_cast<float*>(output_items[0]);
40
41        for(int i=0; i<noutput_items; i++)
42        {
43          out[i] = in[i] * 2.0f;
44        }
45
46        consume_each (noutput_items);
47        return noutput_items;
48      }
49    }
50  }
```

After the C++ code, we further modify the **oot_multiply.block.yml** file in the **grc** folder like the following. Be careful that the YAML file requires extreme format alignment.

```
id: oot_multiply
label: multiply
category: '[oot]'

templates:
  imports: from gnuradio import oot
  make: oot.multiply()

inputs:
- label: inSig
  domain: stream
  dtype: float

outputs:
- label: outSig
  domain: stream
  dtype: float

file_format: 1
```

Now we compile the module and install it in the GNU Radio. Go the **gr-oot** folder and build a folder named **build** and in the **build** folder we compile and install the module.

```
$ cd
$ cd sdr/gr-oot/
$ mkdir build
$ cd build/
$ cmake ../
$ make
$ sudo make install
$ sudo ldconfig
```

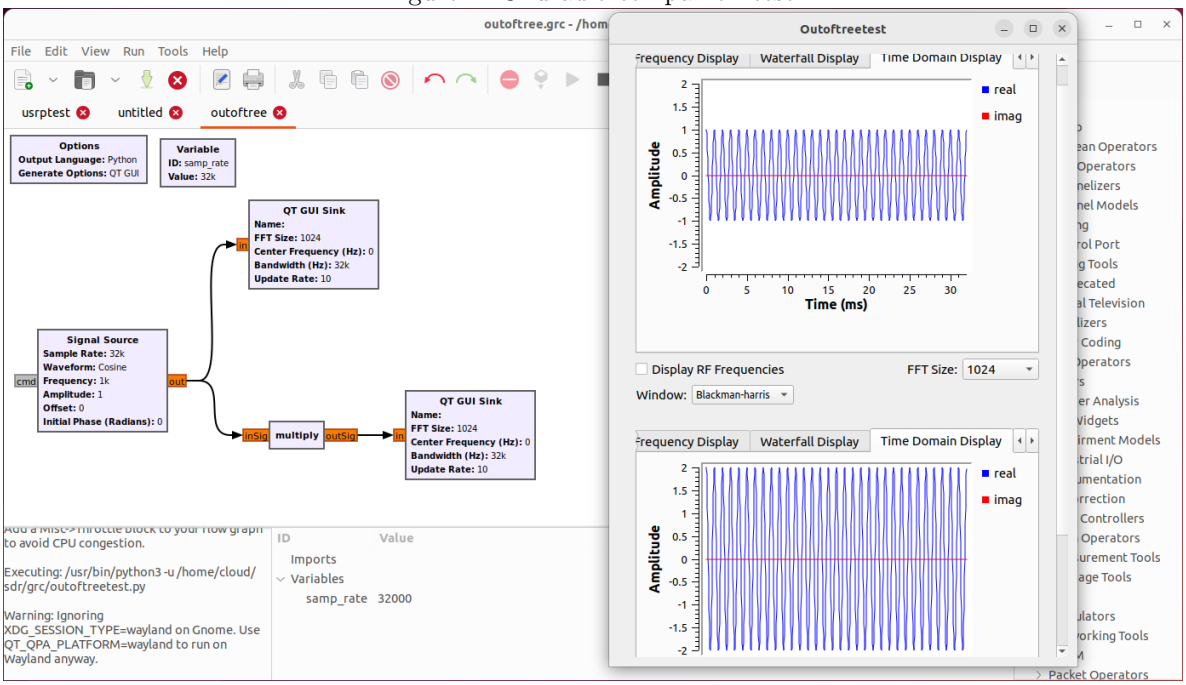
During the steps above, you may fail due to bugs in your code. After this, we open the **gnuradio-companion** and build a new program file to test it. First, in the toolbox right side you could find the new module and new block. Next, we grab a **Signal Source** block and change the output type from complex to float. Next, we one our newly built **multiply** block in the **oot** module and two **QT GUI Sink** and also change the input type of the two sinks to float. We connect the output of signal source directly to one sink and connect to the other sink through the **multiply** block. Finally we run it and see the original signal amplitude is multiplied by two.

Now we see although this block works well, it is not convenient that we need to recompile and reinstall the module every time we need to change the multiplier. In the following part we add a configurable parameter to the block. The main idea is that we add a variable in the C++ code and use it to do the multiplication. At the same time, the value is passed from the block of GNU Radio user interface through the YAML code.

First, we uninstall the previous block and use **gr_modtool** to create another block named **multiplyconf** using the following commands.

```
$ cd
$ cd sdr/
$ cd gr-oot/
$ cd build/
$ sudo make uninstall
$ make clean
$ cd ../
$ gr_modtool add -t general -l cpp multiplyconf
```

Figure 4: Gnuradio-companion test.



This time, besides the **multiplyconf_impl.cc**, we also need to modify the header file **multiplyconf_impl.h** and the another header file in **gr-oot/include/oot/** named **multiplyconf.h**. In the following, we give the codes of YAML:

```yaml
id: oot_multiplyconf
label: multiplyconf
category: '[oot]'

templates:
  imports: import oot
  make: oot.multiplyconf(${m})

parameters:
- id: m
  label: Multiplier
  dtype: float
  default: '1.0'

inputs:
- label: inSig
  domain: stream
  dtype: float

outputs:
- label: outSig
  domain: stream
  dtype: float

file_format: 1
```

You can see that there another part of codes describing the parameter and we have one parameter called **Multiplier** with the ID **m** and default value 1.0. Next we finish the **multiplyconf.h**.

```
#ifndef INCLUDED_OOT_MULTIPLYCONF_H
#define INCLUDED_OOT_MULTIPLYCONF_H

#include <oot/api.h>
#include <gnuradio/block.h>

namespace gr {
  namespace oot {

    class OOT_API multiplyconf : virtual public gr::block
    {
     public:
      typedef std::shared_ptr<multiplyconf> sptr;

      static sptr make(float m);
    };

  } // namespace oot
} // namespace gr

#endif
```

And the only difference of **multiplyconf.h** from before is that the **static sptr make();** changes to **static sptr make(float m);**. This **float m** takes the parameter from the YAML and passes it to the implemented work function in **multiplyconf_impl.cc**. Next, we check the codes of **multiplyconf_impl.h**.

```
#ifndef INCLUDED_OOT_MULTIPLYCONF_IMPL_H
#define INCLUDED_OOT_MULTIPLYCONF_IMPL_H

#include <oot/multiplyconf.h>

namespace gr {
  namespace oot {
    class multiplyconf_impl : public multiplyconf
    {
     private:
      float d_multiplier;

     public:
      multiplyconf_impl(float m);
      ~multiplyconf_impl();

      void forecast (int noutput_items, gr_vector_int &
          ninput_items_required);
      int general_work(int noutput_items,
           gr_vector_int &ninput_items,
           gr_vector_const_void_star &input_items,
           gr_vector_void_star &output_items);
    };
  } // namespace oot
} // namespace gr
#endif /* INCLUDED_OOT_MULTIPLYCONF_IMPL_H */
```

In the above code, you can also see that the constructor function also takes the parameter m. This is to initialize the member variable **d_multiplier** in the class to be used to multiply the signal. We finally check the codes of **multiplyconf_impl.cc**.

```cpp
#include <gnuradio/io_signature.h>
#include "multiplyconf_impl.h"
namespace gr {
  namespace oot {
    multiplyconf::sptr
    multiplyconf::make(float m)
    {
      return gnuradio::make_block_sptr<multiplyconf_impl>(m
        );
    }

    multiplyconf_impl::multiplyconf_impl(float m)
      : gr::block("multiplyconf",
              gr::io_signature::make(1, 1, sizeof(float)),
              gr::io_signature::make(1, 1, sizeof(float))),
              d_multiplier(m)
    {}

    multiplyconf_impl::~multiplyconf_impl()
    {}

    void
    multiplyconf_impl::forecast (int noutput_items, gr_vector_int &
        ninput_items_required)
    {
      gr_vector_int::size_type ninputs = ninput_items_required.size();
      for(int i=0; i < ninputs; i++)
      {
              ninput_items_required[i] = noutput_items;
      }
    }

    int
    multiplyconf_impl::general_work (int noutput_items,
                      gr_vector_int &ninput_items,
                      gr_vector_const_void_star &input_items,
                      gr_vector_void_star &output_items)
    {
      const float* in = static_cast<const float*>(input_items[0]);
      float* out = static_cast<float*>(output_items[0]);

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

      consume_each (noutput_items);
      return noutput_items;
    }

  } /* namespace oot */
} /* namespace gr */
```

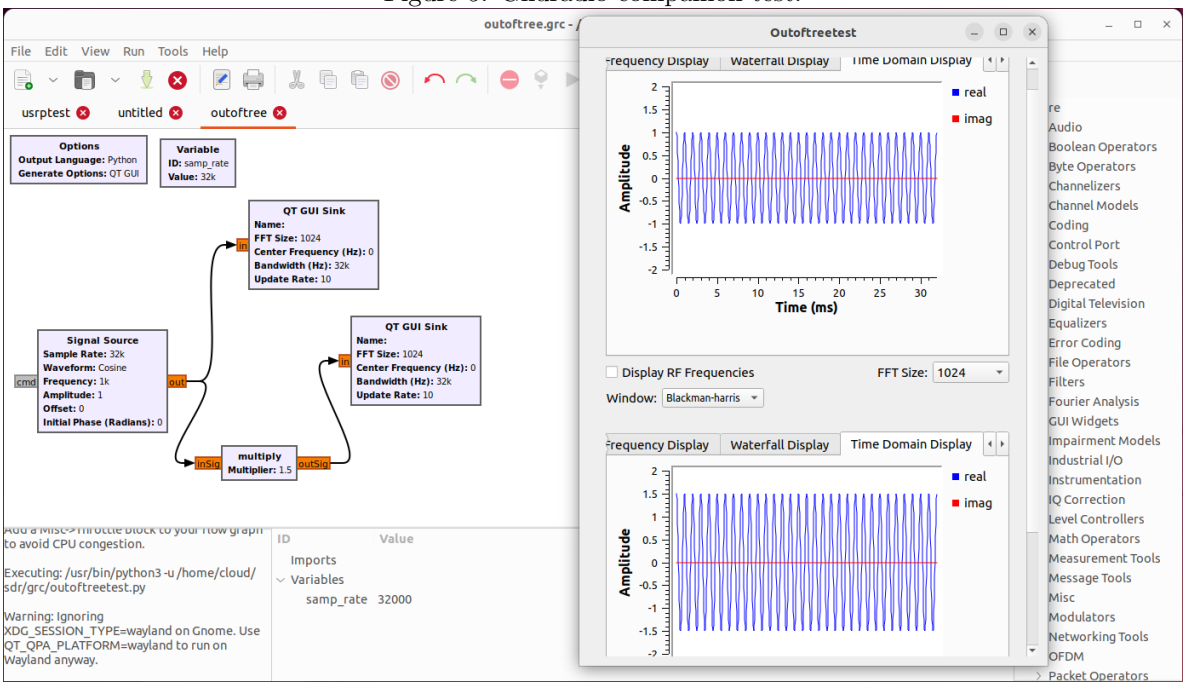
Now we have finished all the coding part, go back to the **gr-oot** folder. This time, before building the project, we need to first bind the C++ codes to python to be called in GNU Radio using **gr_modtool**. The binding is only needed when we create a C++ block with parameters. After that,

we build and install the project, the commands are as following. Notice that the word after the **bind** is the block name which should be exactly the same as your created block.

```
$ cd
$ cd sdr/gr-oot/
$ gr_modtool bind multiplyconf
$ cd build/
$ cmake ../
$ make
$ sudo make install
$ sudo ldconfig
```

Now we can see in Figure 5 the newly added block can be configured through the parameter in the user interface.

Figure 5: Gnuradio-companion test.



3.2 Python Version

GNU Radio provides also the option to use Python to program a block. The steps are similar to the C++ version.

```
$ cd
$ cd sdr/
$ cd gr-oot/
$ cd build/
$ sudo make uninstall
$ make clean
$ cd ../
$ gr_modtool add -t general -l python multiplypy
```

The python is pretty easy, for the coding part it only has a **multiplypy.py** in the **gr-oot/python/** folder. Now we check the code, this also has a configurable parameter.

```
import numpy
from gnuradio import gr
```

```
3
4  class multiplypy(gr.basic_block):
5      def __init__(self, m = 1.0):
6          self.multiplier = m
7          gr.basic_block.__init__(self,
8              name="multiplypy",
9              in_sig=[numpy.float32],
10             out_sig=[numpy.float32])
11
12     def forecast(self, noutput_items, ninputs):
13         ninput_items_required = [noutput_items] * ninputs
14         return ninput_items_required
15
16     def general_work(self, input_items, output_items):
17         ninput_items = min([len(items) for items in input_items])
18         noutput_items = min(len(output_items[0]), ninput_items)
19         output_items[0][:noutput_items] = [each * self.multiplier for
               each in input_items[0][:noutput_items]]
20         self.consume_each(noutput_items)
21         return noutput_items
```

And the YAML codes are also the same as C++ version.

```
1  id: oot_multiplypy
2  label: multiplypy
3  category: '[oot]'
4
5  templates:
6    imports: import oot
7    make: oot.multiplypy(${m})
8
9  parameters:
10 - id: m
11   label: Multiplier
12   dtype: float
13   default: '1.0'
14
15 inputs:
16 - label: inSig
17   domain: stream
18   dtype: float
19
20 outputs:
21 - label: outSig
22   domain: stream
23   dtype: float
24
25 file_format: 1
```

For all the above source codes, you can find them in [Yun].

4 Some features of GR

Here we attach some issues worth notification during the GNU Radio developments.

- For a general block, the work function will only be called when there are samples waiting at the input for the block to process. If there is no sample to be processed, means that the **ninput_item** is zero, the work function will not be called. This issue happens when I have processed all the inputs and return to generate the output in next round of work function in next state, however, the work function is not called again so there no chance to generate the output samples unit more input samples are provided.

References

[GR] GR. InstallingGR. https://wiki.gnuradio.org/index.php/InstallingGR.

[Yun] Zelin Yun. GR-OOT Source Codes. https://github.com/cloud9477/gr-oot.