By Nathan Price* and Maciej Zawodniok

INTRODUCTION TO PROGRAMMING FOR GNU RADIO

* Former PhD student currently working at Southwest Research Institute

Useful links

- Video Series
- Tutorials:
  - https://wiki.gnuradio.org/index.php/Tutorials
  - https://wiki.gnuradio.org/index.php/OutOfTreeModules
  - https://wiki.gnuradio.org/index.php/Guided_Tutorial_PSK_Demodulation

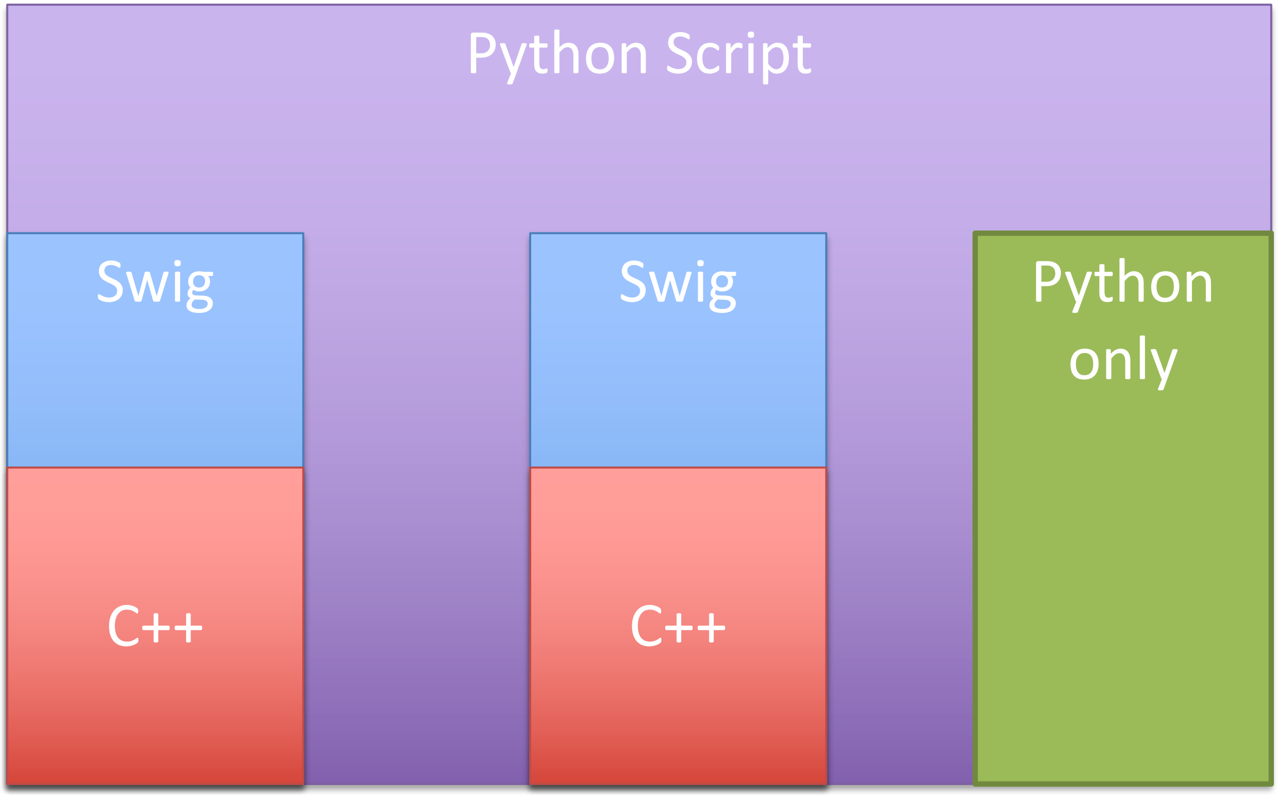
- https://www.gnuradio.org/doc/doxygen/

MISSOURI S&T

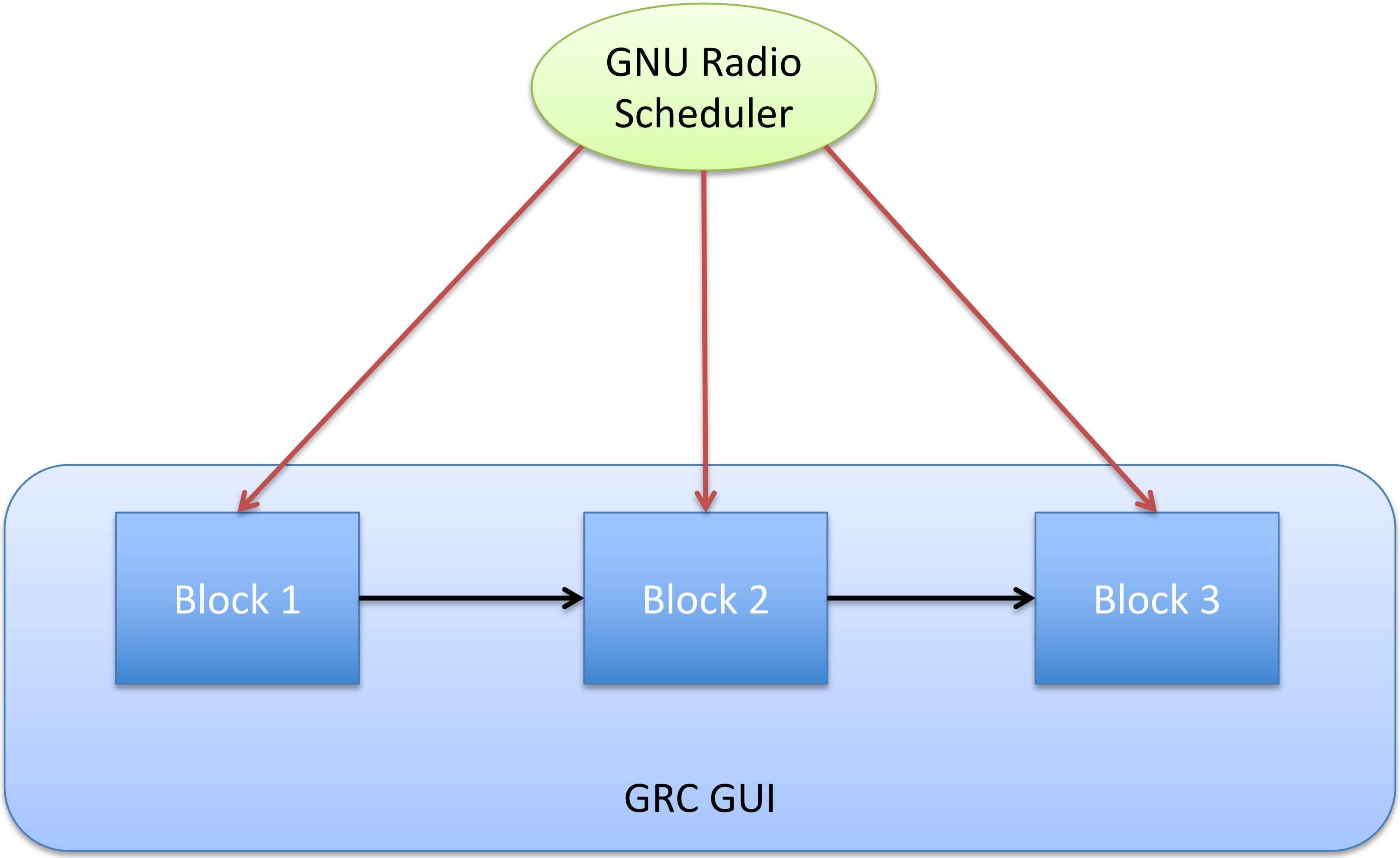
Important Notes

- GNU Radio should be installed from source!
- Save/preserve your progress
  - If needed restart fresh/from beginning and **carefully** copy source code
- Test each step (TX and RX separately/step-by-step)

Behind the scene of a flow graph



GNU Radio Runtime



Building a module

- GNU Radio is structured in modules
- A modules contains a set of blocks
  - Blocks may be of any type (e.g. source, function, sink, etc.)

Create new code: *gr_modtool*

- gr_modtool newmod *modulename*
  - Creates file structure
  - If you have existing directory structure use: “--skip-libs”, “--skip-swig” etc.
- cd gr-modulename
- gr_modtool add *blockname*
  - Adds necessary files to file structure

Add GRC code and compile

- **gr_modtool makexml my_block**
  - Creates XML file in “grc/” folder
  - May need editing if data types mixed
- Compile in subfolder
  - **mkdir build**
  - **cd build**
  - **cmake ..**
  - **make**
  - **sudo make install**

Structure of a Module

- apps: compiled code
- cmake: files necessary for compellation
- CMakeLists.txt: necessary for compellation
- docs: module documentation
- examples: examples written by author
- grc: .xml files required for module to work in GRC
- include: resource header files
- lib: implementation files and headers
- python: python code
- swig: magic

Files we care about

- lib/blockname_impl.cc
  - Constructor
  - Forecast
  - General work
- lib/blockname_impl.h
  - Partial class declaration
- grc/modulename_blockname.xml

Partial Class Declaration

```
#ifndef INCLUDED_DEMO_PKT_FRAMER_IMPL_H
#define INCLUDED_DEMO_PKT_FRAMER_IMPL_H

#include <demo/pkt_framer.h>

namespace gr {
  namespace demo {

    class pkt_framer_impl : public pkt_framer
    {
     private:
      // Nothing to declare in this block.

     public:
      pkt_framer_impl(unsigned int payload_size);
      ~pkt_framer_impl();
      unsigned int _payload_size;

      // Where all the action really happens
      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 demo
} // namespace gr

#endif /* INCLUDED_DEMO_PKT_FRAMER_IMPL_H */
```

Create a variable for each input parameter

Constructor

```
/*
 * The private constructor
 */
test_impl::test_impl()
  : gr::block("test",
          gr::io_signature::make(<+MIN_IN+>, <+MAX_IN+>, sizeof(<+ITYPE+>)),
          gr::io_signature::make(<+MIN_OUT+>, <+MAX_OUT+>, sizeof(<+OTYPE+>)))
{}
```

- How many inputs?
- What type?
- How many outputs?
- What type?
- First and only chance to read input arguments

Constructor specific options

- gr::io_signature::make()
- gr::io_signature::make2()
- gr::io_signature::make3()
- gr::io_signature::makev()

Warning

Forecast

```
void
test_impl::forecast (int noutput_items, gr_vector_int &ninput_items_required)
{
    /* <+forecast+> e.g. ninput_items_required[0] = noutput_items */
}
```

- Tell the scheduler how many input items you want.
- Consider *noutput_items* a work request
- *ninput_items_reqired[0]* is a vector because you may have more than one input. You may consume inputs at different rates (be careful).
- *noutput_items* is not a vector because you must return all outputs at the same rate.

Forecast considerations

- You are not required to produce exactly *noutput_items*; you are allowed to fall short.
- If your block does not have a synchronous IO relationship, it is better to overestimate *ninput_items_reqired*.

General Work

```
int
test_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 <+ITYPE*> *in = (const <+ITYPE*> *) input_items[0];
    <+OTYPE*> *out = (<+OTYPE*> *) output_items[0];

    // Do <+signal processing+>
    // Tell runtime system how many input items we consumed on
    // each input stream.
    consume_each (noutput_items);

    // Tell runtime system how many output items we produced.
    return noutput_items;
}
```

General Work

- Declare pointers to input and output buffers.
- Read item from input buffer -> process -> send to output buffer -> call consume
- You do not have to consume all items on the input buffer, nor do you have to fill the output buffer.
- Anything left on the input buffer will remain there.
- Correct *noutput_items* if you differed from forecast.

Contact Information

- Maciej Zawodniok
- Email: mjzx9c@mst.edu
- Cell: (573)-308-2319