Exposing GNU Radio: Developing and Debugging

Tom Rondeau
tom@trondeau.com

GNU Radio Maintainer

May 27, 2012

Download Materials

- http://www.trondeau.com/gr-tutorial
- Presentation PDF
- Case Study materials
  - GNU Radio apps to run examples.
  - Links to source code for analysis.
  - Data file for first case study.
  - Images of expected output.
  - Exercises.

Prologue

"The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point."

\- Claude Shannon, "A Mathematical Theory of Communication"

THE PHYSICAL LAYER IS DEAD!

Long live the physical layer...

- Innovation these days comes from the *use* of the PHY layer.
- Flexibility and reconfigurability are key.
- Rapidly prototype and experiment to prove new ideas.
- Communications is still very hard.

GNU Radio helps us realize this

Communications Building Blocks

1. Rapid prototyping of PHY layer systems.
2. Analysis and development of signals.
3. Educational material to teach the workings of various comms.
4. Expose engineering techniques and tricks.

How GNU Radio Helps

Provides...

1. Basic data structure, the flow graph, to build streaming signal processing systems.
2. Connections to and from hardware and software.
3. A set of I/O and signal processing blocks.
4. A framework of programming tools and examples.
5. A community of experts and enthusiasts.
6. Open source licensing.

Information Sources for GNU Radio

Tools and Manuals

- Project website: gnuradio.org
- Download source: gnuradio.org/redmine/projects/gnuradio/wiki/Download
- Online C++ Manual: gnuradio.org/doc/doxygen/
- Online Python Manual: gnuradio.org/doc/sphinx/
- My webite for news and analysis: www.trondeau.com

Community

- Active developer community producing examples.
- Large participation on our mailing list.
- The Complimentary GNU Radio Archive Network (CGRAN).
- Growing list of projects on github.
- Large participation at conferenes like the Wireless Innovation Forum's WinnComm'11.
- Impressive turnout and participation at the 2011 GNU Radio Conference.
- GNU Radio Conference 2012 comming soon.

Basics of Python Programming

./example_basics.py

```
from gnuradio import gr, filter

class my_topblock(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)

        amp = 1
        taps = filter.firdes.low_pass(1, 1, 0.1, 0.01)

        self.src = gr.noise_source_c(gr.GR_GAUSSIAN, amp)
        self.flt = filter.fir_filter_ccf(1, taps)
        self.snk = gr.null_sink(gr.sizeof_gr_complex)

        self.connect(self.src, self.flt, self.snk)

if __name__ == "__main__":
    tb = my_topblock()
    tb.start()
    tb.wait()
```

Chapter the First: Scheduling

"Oh dear! Oh dear! I shall be late!"

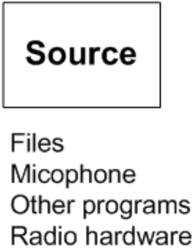
- Lewis Carroll, *Alice's Adventures in Wonderland*

GNU Radio Fundamentals

A real-time, streaming signal processing platform.

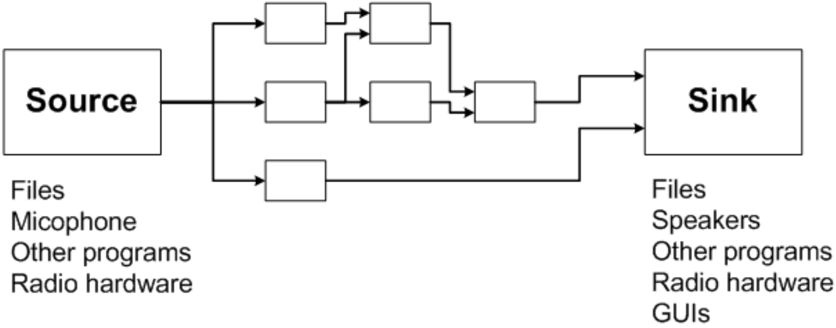
Fundamentals: The Flowgraph

Fundamentals: The Flowgraph



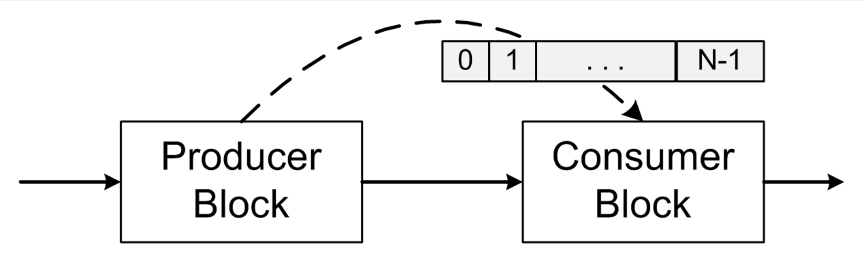
Fundamentals: The Flowgraph

Fundamentals: The Flowgraph

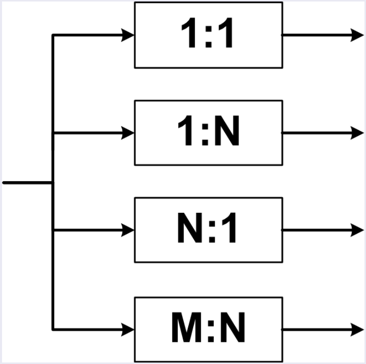


Fundamentals: Dynamic Scheduling

The dynamic scheduler passes chunks of *items* between signal processing blocks.



Fundamentals: I/O Signatures



Programming Model

- All low level work and signal processing is done in C++.
- Wrapped into Python for use as a scripting language.
- GNU Radio Companion: a graphical interface to build GNU Radio applications that sits on top of Python.

GNU Radio Processing Blocks

- Basic mathematical and logical operations.
- Large library of filter design and processing algorithms.
- I/O support for many domains.
- Type conversions.
- Analog (AM/FM) processing techniques.
- Synchronization algorithms (PLL, Costas loop, etc.).
- Data and flow graph management blocks.
- Narrowband and OFDM digital modulation capabilities.
- Various audio vocoders.
- Trellis, convolutional coding, and similar algorithm support.
- Graphical visualization tools (oscilloscopes, PSD, and waterfall viewers).
- Many examples for all different areas of signal processing.

GNU Radio Top-Level Components (as of v3.7)

Fundamentals

- gr-analog
- gr-block
- gr-digital
- gr-fec
- gr-fft
- gr-filter
- gr-runtime
- gr-trellis
- gr-vocoder
- gr-wavelet

Graphical Interfaces

- gr-qtgui
- gr-wxgui

Hardware Interfaces

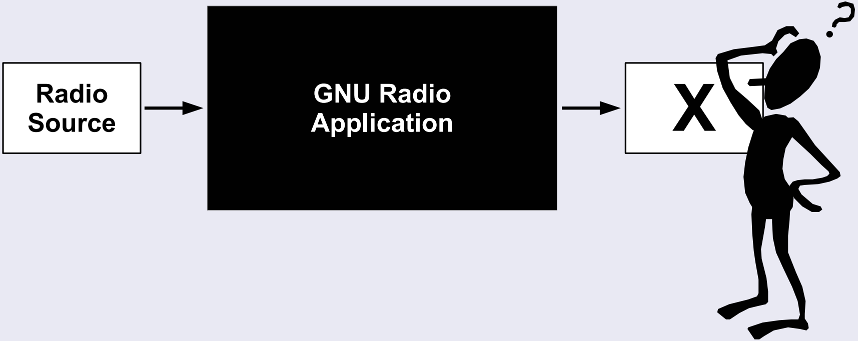
- gr-audio
- gr-comedi
- gr-fcd
- gr-shd
- gr-uhd

Chapter the Second: Graphical User Interfaces

"Here, then, we have, in the very beginning, the groundwork for something more than a mere guess."
- Edgar Allan Poe, *The Gold Bug*

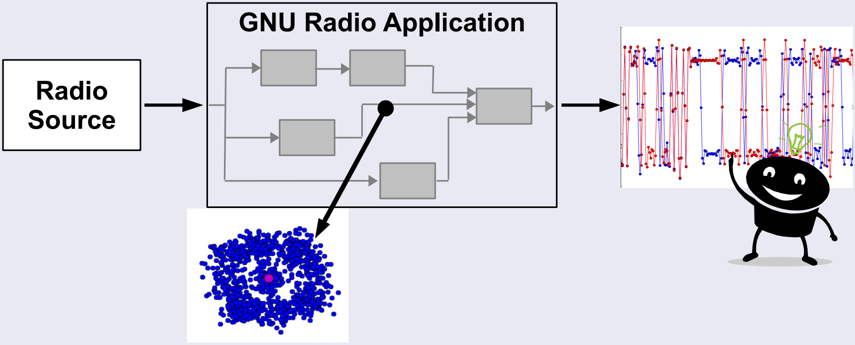
Classic Problem

I have an app. It doesn't work. Why?



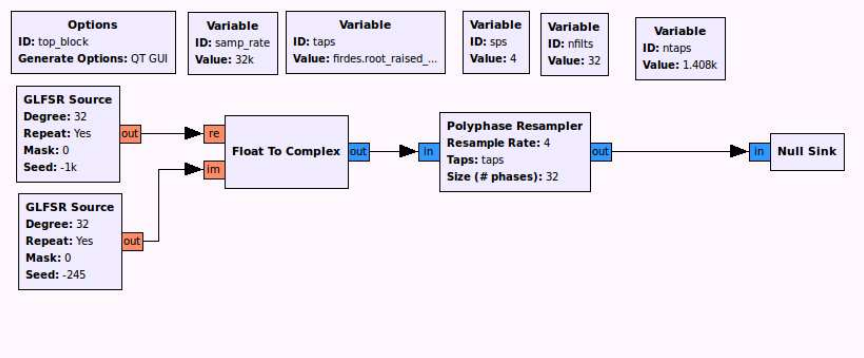
Classic Problem

Visualizing the signal at different points can illuminate the problem.



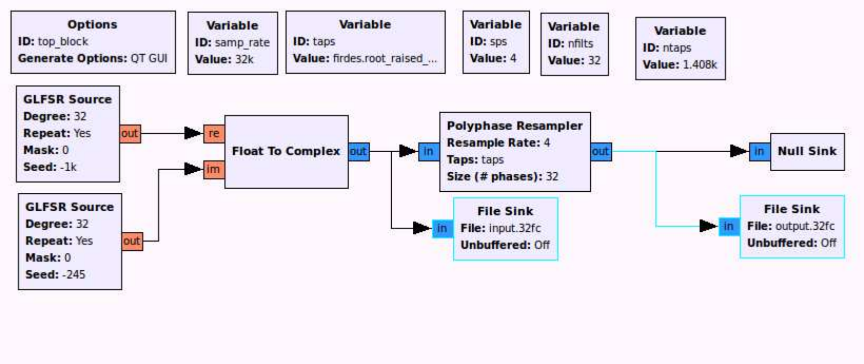
Example: example_gui.grc

A graph that does something



Example: example_gui.grc

Visualizing the state with files



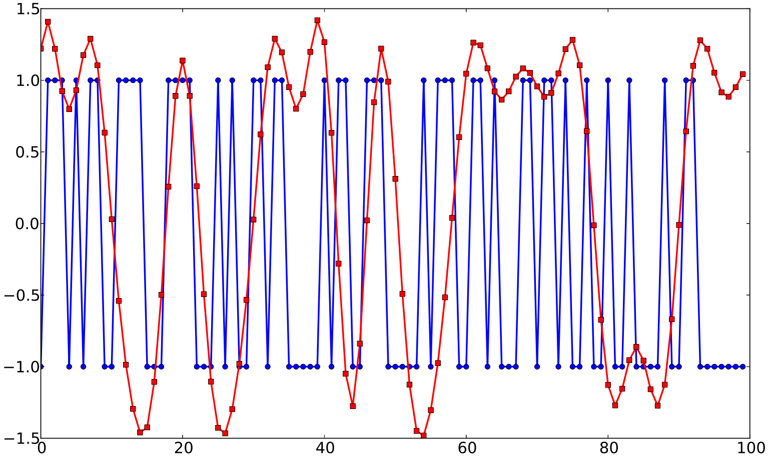
Example: example_gui.grc

Read in the data with your favorite language/program

```
import scipy, pylab

# Read in the data into two variables
in_data  = scipy.fromfile(open('input.32fc', 'r'),
                 dtype=scipy.complex64, count=10000)
out_data = scipy.fromfile(open('output.32fc', 'r'),
                 dtype=scipy.complex64, count=10000)

# Do some data manipulations here

# Plot the data
fig = pylab.figure(1, figsize=(14,8), facecolor='w')
sp = fig.add_subplot(1,1,1)
sp.plot(in_data.real[1200:1300], 'b-o', linewidth=2)
sp.plot(out_data.real[1200:1300], 'r-s', linewidth=2)
pylab.show()
```

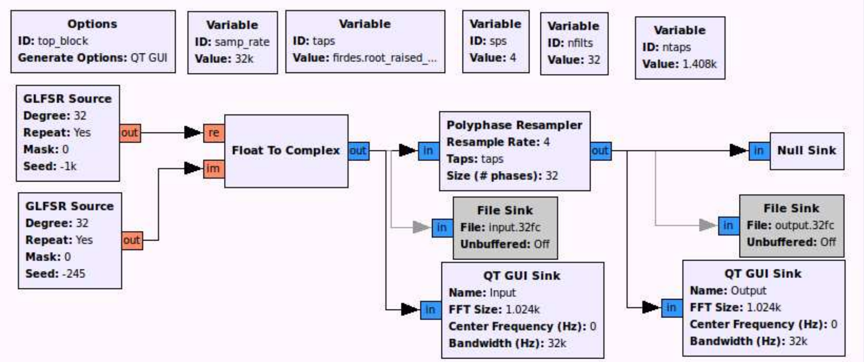
Example: example_gui.grc

Python's Pylab output



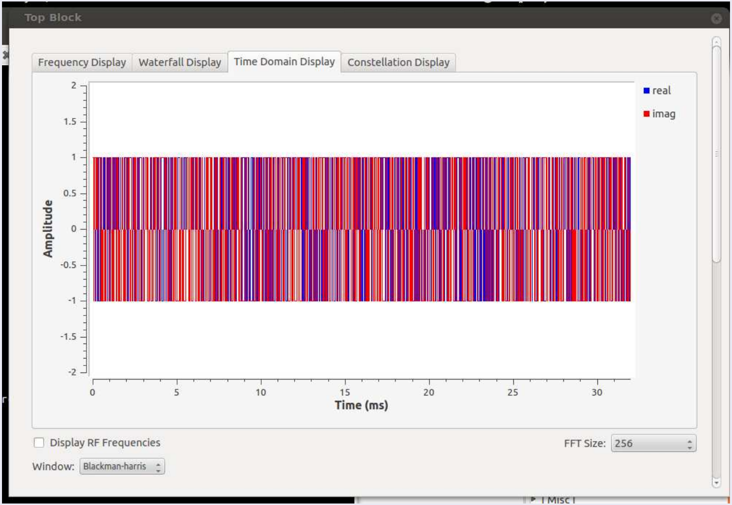
Example: example_gui.grc

Realtime Visualization: QT-GUI



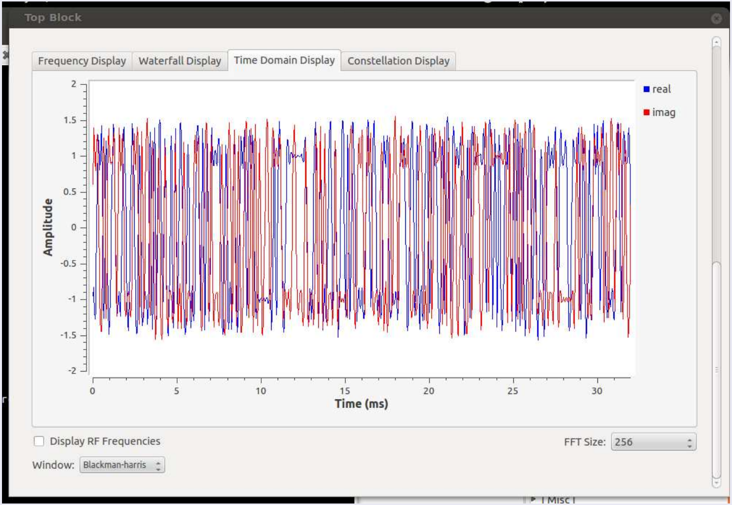
Example: example_gui.grc

QT-GUI output from first ('Input') sink



Example: example_gui.grc

QT-GUI output from second ('Output') sink



Chapter the Third: Filtering

"The need for filters intrudes on any thought experiment about the wonders of abundant information."

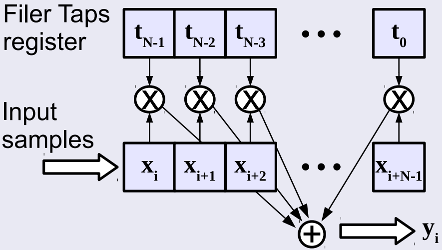
- James Gleik, *The Information*

FIR Filters = Convolution

Python Example

- ./convolution.py
- ./convolve_filter.py

Convolution: $y[i] = \sum t_{N-n-1} x[i+n]$



Common Filters

Standard Types

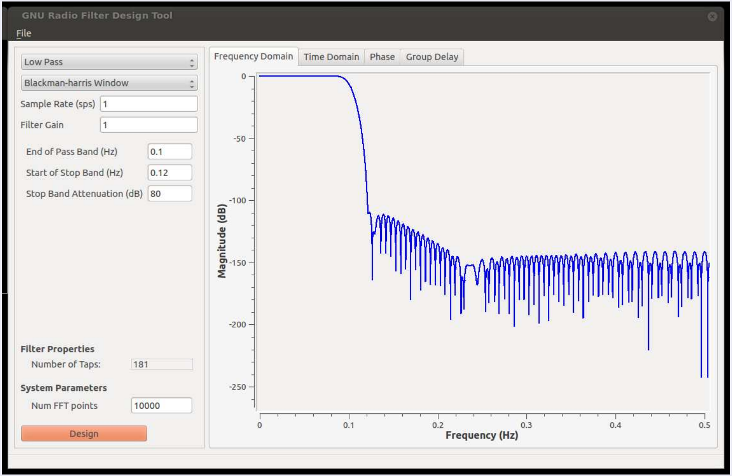
- Low Pass
- High Pass
- Band Pass
- Band Reject (Notch)
- Nyquist / Gaussian / pulse shapping

Standard Implementations

- Tapered Windowing of Sinc (Hamming, Hann, Blackman-harris, etc.)
- Equiripple (via Parks-McClellen algorithm)

Demonstration

gr_filter_design



Time vs. Frequency Domain

Convolution in Time $\Longleftrightarrow$ Multiplication in Frequency

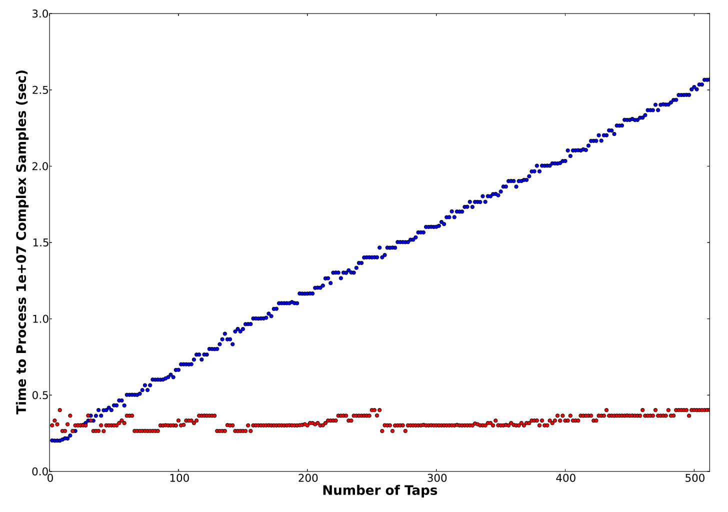
$$\mathcal{F}\left(t * x\right) = \mathcal{F}(t) \cdot \mathcal{F}(x)$$

$$t * x = \mathcal{F}^{-1}\left(\mathcal{F}(t) \cdot \mathcal{F}(x)\right)$$

$\rightarrow$ $\mathcal{F}$ is the Fourier transform operator.
And we know an FFT can be done with complexity $O(N \log(N))$

How Complexity Helps

The FFT method quickly takes over



Using firdes to create a filter

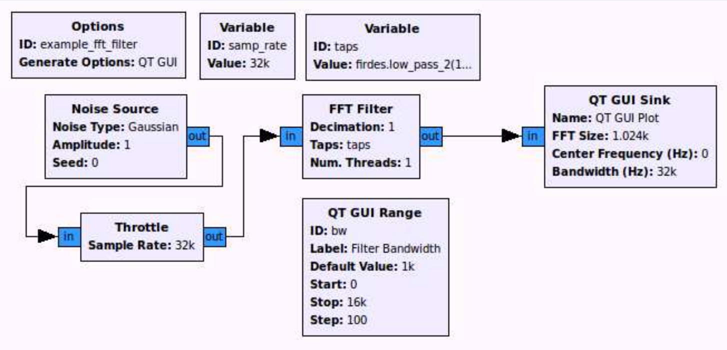
gr.firdes.low_pass_2 (after v3.7: filter.firdes.low_pass_2)

```
gr.firdes.low_pass_2(gain, sample rate,
                     bandwidth, transition band,
                     stopband attenuation (dB))
```

- **gain**: constant multiplication coefficient to all taps
- **sample rate**: sample rate of filter in samples/second
- **bandwidth**: end of passband (3 dB point); units relative to sample rate
- **transition band**: distance between end of passband and start of stopband; units relative to sample rate
- **stopband attenuation**: attenuation (in dB) in stopband

GRC Example Environment (example_fft_filter.grc)

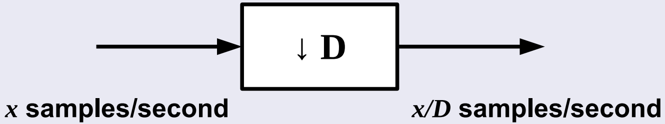
Update filter taps to adjust bandwidth



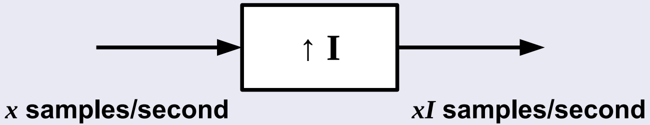
- Generate white noise
- Uses FFT filter with taps in variable *taps*
- Taps defined using *bw* variable adjustable at runtime

Rate Change is Fundamental to Software Radio

Downsampling (decimation)

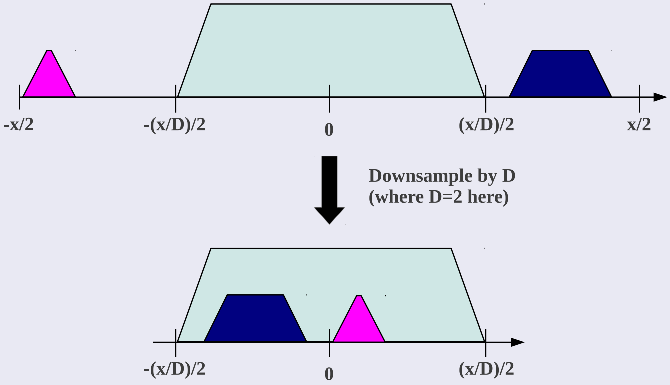


Upsampling (interpolation)

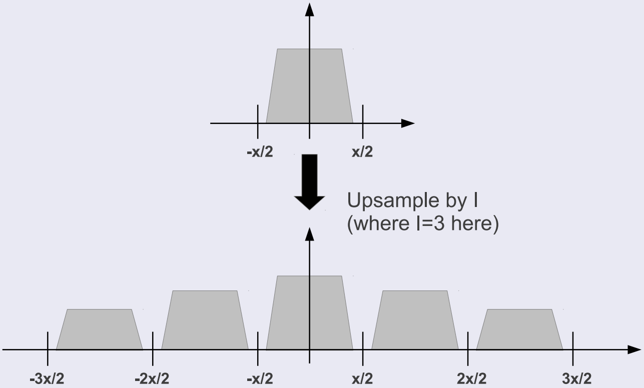


Downsampling aliases outside bands

The 'Nyquist Zones' now fall into the new passband

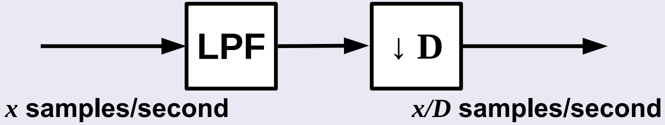


Upsampling creates images

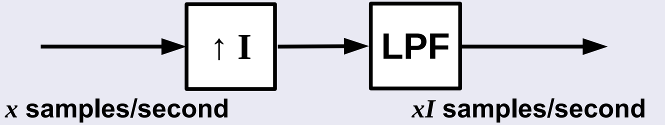


Always Filter when Changing Rates

Filter Signals Before Downsampling



Upsample and then Remove Images



Rate Changing GNU Radio Blocks

Downsampling

- fir_filter_ccc, ccf, fcc, fff, fsf, scc
- fft_filter_ccc, fff
- pfb_decimator_ccf
- pfb_channelizer_ccf

Upsampling

- interp_fir_filter_ccc, ccf, fcc, fff, fsf, scc
- pfb_interpolator_ccf
- pfb_synthesize_ccf

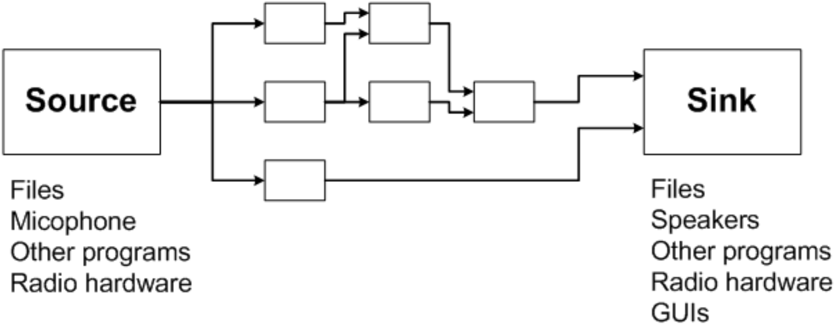
Resampling

- pfb_arb_resampler_ccf, fff
- fractional_interpolator_cc, ff

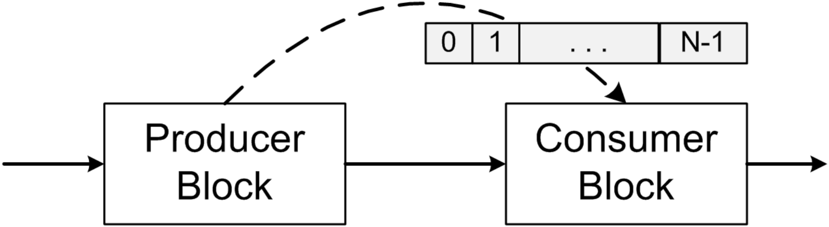
Chapter the Fourth: Programming Blocks

"Computers are good at following instructions, but not at reading your mind."
- Donald Knuth

The Flowgraph Structure, Revisited



Input and Output Buffers of a Streaming System



Python Programming

How to construct a basic application

- Create a **top block**.
- Instantiate the blocks for the app.
- Connect blocks to for a graph.
- Start and run the graph.

The Full Program

./example_basics.py

```
from gnuradio import gr, filter

class my_topblock(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)

        amp = 1
        taps = filter.firdes.low_pass(1, 1, 0.1, 0.01)

        self.src = gr.noise_source_c(gr.GR_GAUSSIAN, amp)
        self.flt = filter.fir_filter_ccf(1, taps)
        self.snk = gr.null_sink(gr.sizeof_gr_complex)

        self.connect(self.src, self.flt, self.snk)

if __name__ == "__main__":
    tb = my_topblock()
    tb.start()
    tb.wait()
```

Create a **top block**

A class that inherits from **gr.top_block**

```
from gnuradio import gr, filter

class my_topblock(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)
```

Instantiate the blocks for the app

Using GR block constructors

```
self.src = gr.noise_source_c(gr.GR_GAUSSIAN, amp)
self.flt = filter.fir_filter_ccf(1, taps)
self.snk = gr.null_sink(gr.sizeof_gr_complex)
```

Connect blocks to for a graph

self is a top_block with a **connect** member

```
self.connect(self.src, self.flt)
self.connect(self.flt, self.snk)
```

Start and run the graph

Use **start()** or **run()** from a function

```
if __name__ == "__main__":
    tb = my_topblock()
    tb.start()
    tb.wait()
    #tb.run() # run both start() and wait()
```

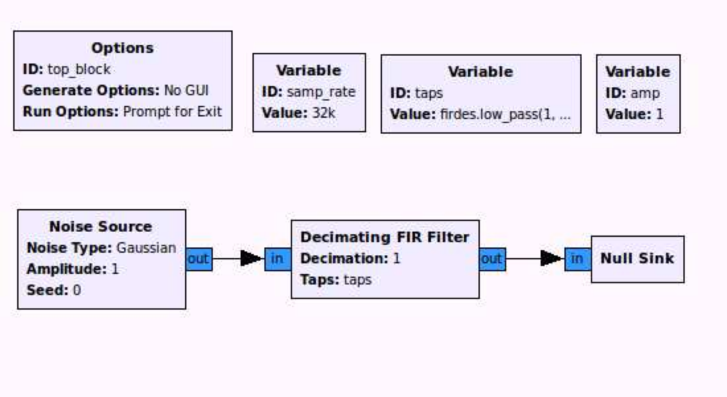
Another Look at the Full Program

./example_basics.py

```
from gnuradio import gr, filter

class my_topblock(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)

        amp = 1
        taps = filter.firdes.low_pass(1, 1, 0.1, 0.01)

        self.src = gr.noise_source_c(gr.GR_GAUSSIAN, amp)
        self.flt = filter.fir_filter_ccf(1, taps)
        self.snk = gr.null_sink(gr.sizeof_gr_complex)

        self.connect(self.src, self.flt, self.snk)

if __name__ == "__main__":
    tb = my_topblock()
    tb.start()
    tb.wait()
```

Visualizing the Program

example_python_prog.grc



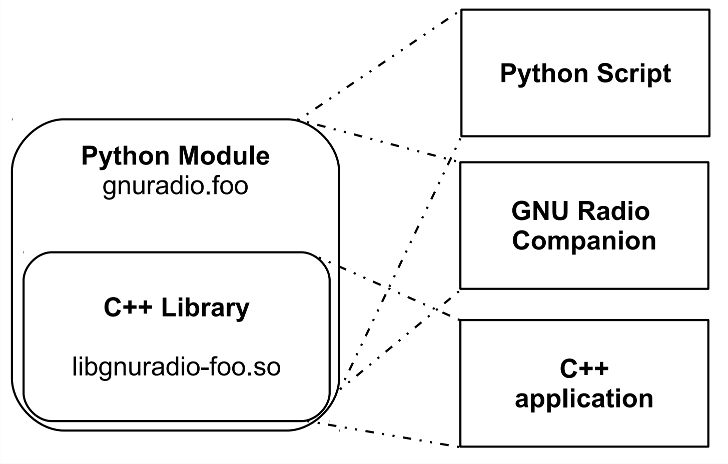
Programming Practices

New model as of GNU Radio v3.7

- We will discuss building component **foo**.
- We will develop the block **bar**.
- These will be accessible under the namespace **gr::foo**.

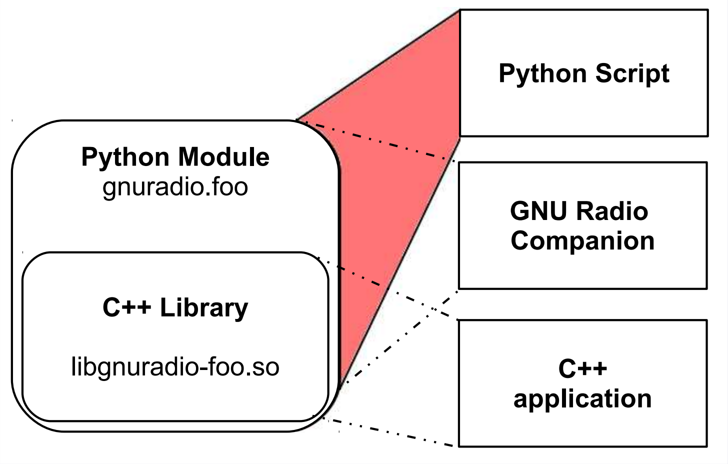
Programming Practices

Layered API



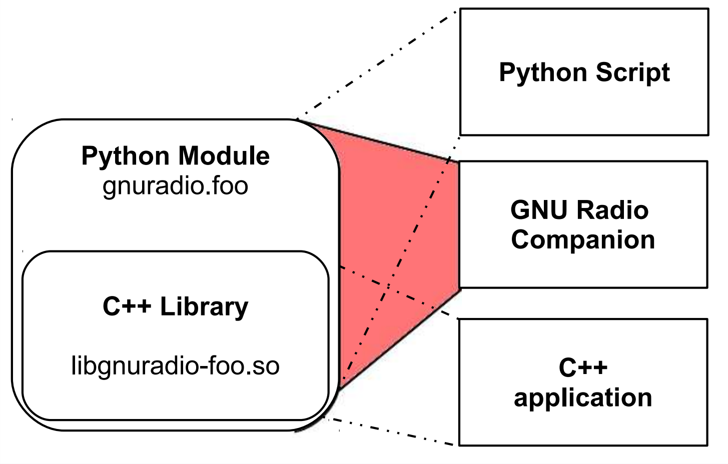
Programming Practices

Layered API: Python Wrapper through SWIG



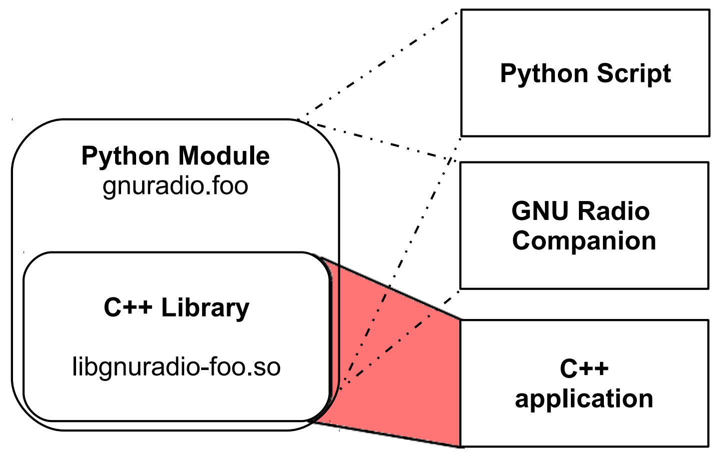
Programming Practices

Layered API: GNU Radio Companion GUI



Programming Practices

Layered API: Direct C++ Applications



Component Structure

Each component **foo** contains:

apps any full-fledged applications specific to the component

doc place for extra documentation, including Doxygen **.dox** files to describe the component.

examples example code to demonstrate usage of component blocks and algorithms.

grc GNU Radio Companion files and block tree.

include public header files

include/foo actual location of public header files. Headers in included using **#include** <**foo/bar.h**>.

lib location of the implementation source and private header files (**bar_impl.h** and **bar_impl.cc**, generally).

swig location of SWIG interface file. We use a simplified structure that only includes the public header file for SWIGing.

Component Structure

Private Implementation, Public Header

- The API of the block is defined in a public header file in **include/foo/bar.h**.
- Only methods defined in the public header file are accessible through the library and through the Python interface to the block.
- The factory **"make"** function is a member of the public class. It will instantiate a private implementation of the block.
- A block can have more than one **"make"** function.
- Multiple private implementations can be defined for a block base on architecture, platform, experiments, etc.

Component Structure

Private Implementation Specifics

- Implementation files contain a header and a source as **lib/bar_impl.h** and **lib/bar_impl.cc**
- Creates a private class that inherits from the public class.
- Implements the public **"make"** function.
- When multiple implementations, replace the **_impl** with a more appropriate suffix. The blocks that implement FFTs (like fft_vcc_fftw) are good examples by using FFTW.

Component Structure

Public Header File: foo.h

```
#ifndef INCLUDED_FOO_BAR_H
#define INCLUDED_FOO_BAR_H

#include <foo/api.h>
#include <gr_sync_block.h>

namespace gr {
  namespace foo {
    class FOO_API bar : virtual public gr_sync_block
    {
    public:
      typedef boost::shared_ptr<bar> sptr;

      /*!
       * Manual documentation.
       * \param var explanation of argument var.
       */
      static FOO_API sptr make(dtype var);
      virtual void set_var(dtype var) = 0;
      virtual dtype var() = 0;
    };

  } /* namespace foo */
} /* namespace gr */

#endif /* INCLUDED_FOO_BAR_H */
```

Component Structure

Private Header File: **foo_impl.h**

```
#ifndef INCLUDED_FOO_BAR_IMPL_H
#define INCLUDED_FOO_BAR_IMPL_H

#include <foo/bar.h>

namespace gr {
  namespace foo {
    class FOO_API bar_impl : public bar
    {
    private:
      dtype d_var;

    public:
      bar_impl(dtype var);
      ~bar_impl();
      void set_var(dtype var);
      dtype var();
      int work(int noutput_items,
               gr_vector_const_void_star &input_items,
               gr_vector_void_star &output_items);
    };
  } /* namespace foo */
} /* namespace gr */

#endif /* INCLUDED_FOO_BAR_H */
```

Component Structure

Private Source File: **foo_impl.cc**

```
#ifdef HAVE_CONFIG_H
#include ``config.h''
#endif

#include ``bar_impl.h''
#include <gr_io_signature.h>

namespace gr {
  namespace foo {
    bar::sptr bar::make(dtype var)
    {
      return gnuradio::get_initial_sptr(new bar_impl(var));
    }

    bar_impl::bar_impl(dtype var)
      : gr_sync_block("bar",
              gr_make_io_signature(1, 1, sizeof(in_type)),
              gr_make_io_signature(1, 1, sizeof(out_type)))
    {
      set_var(var);
    }

    bar_impl::~bar_impl()
    {
      // any cleanup code here
    }
```

Component Structure

Private Source File (cont.): **foo_impl.cc**

```
dtype
bar_impl::var()
{
  return d_var;
}

void
bar_impl::set_var(dtype var)
{
  d_var = var;
}
```

Component Structure

Private Source File (cont.): **foo_impl.cc**

```
    int
    bar_impl::work(int noutput_items,
                   gr_vector_const_void_star &input_items,
                   gr_vector_void_star &output_items)
    {
      const in_type *in = (const in_type*)input_items[0];
      out_type *out = (out_type*)output_items[0];

      // Perform work; read from in, write to out.

      return noutput_items;
    }

  } /* namespace foo */
} /* namespace gr */
```

Component Structure

Exporting to Python through SWIG: **swig/foo_swig.i**

```
#define FOO_API

%include ''gnuradio.i''

//load generated python docstrings
%include ''foo_swig_doc.i''

%{
#include ''foo/bar.h''
%}

%include ''foo/bar.h''

GR_SWIG_BLOCK_MAGIC2(foo, bar);
```

NOTE

We are using “GR_SWIG_BLOCK_MAGIC2” for the definitions now. When we are completely converted over, this will be replaced by “GR_SWIG_BLOCK_MAGIC”.

Block Methods

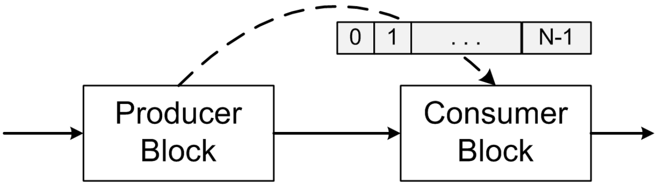
Various Important Block Properties

noutput_items The number of items the output buffer can handle.

consume The number of input items processed.

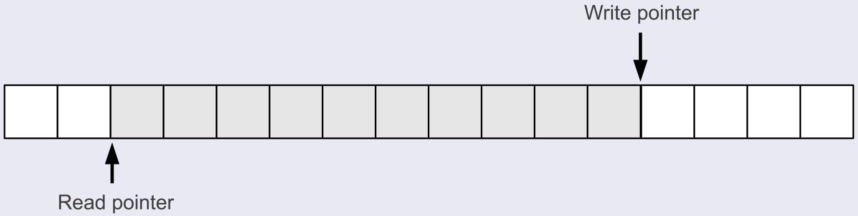
produce The number of output items produced.

rates Determines the input/output ratio (synch, decimation, interpolation, or other).



Buffer Pointers: Readers and Writers

Ciruclar Buffer Structure



Read and write to buffers. Blocks are told how much they can read and write.

Work Functions

gr_sync_block (see: digital_costas_loop)

- noutput_items: number available on input and output.
- return call tells scheduler to produce and consume that amount.
  - consume_all(noutput_items);
  - produce_all(noutput_items);
  - return WORK_CALLED_PRODUCE;

```
int
foo_bar::work(int noutput_items,
              gr_vector_const_void_star &input_items,
              gr_vector_void_star &output_items)
{
  const gr_complex *in = (gr_complex *)input_items[0];
  gr_complex *out = (gr_complex *)output_items[0];

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

Work Functions

gr_sync_decimator (see: goertzel_fc_impl.cc)

- noutput_items: number available to output.
- return call tells scheduler to produce noutput_items/decimation.
  - consume_all(noutput_items*decimation);
  - produce_all(noutput_items);

```
int
foo_bar::work(int noutput_items,
              gr_vector_const_void_star &input_items,
              gr_vector_void_star &output_items)
{
  const gr_complex *in = (gr_complex *)input_items[0];
  gr_complex *out = (gr_complex *)output_items[0];

  int j = 0;
  for(int i = 0; i < noutput_items; i++) {
    out[i] = function(in[j --> (j + decimation())]);
    j += decimation();
  }
  return noutput_items;
}
```

Work Functions

gr_sync_interpolator (see: gr_pfb_synthesizer_ccf)

- noutput_items: number available to output.
- return call tells scheduler to consume noutput_items/interpolation.
  - consume_all(noutput_items/interpolation);
  - produce_all(noutput_items);

```
int
foo_bar::work(int noutput_items,
              gr_vector_const_void_star &input_items,
              gr_vector_void_star &output_items)
{
  const gr_complex *in = (gr_complex *)input_items[0];
  gr_complex *out = (gr_complex *)output_items[0];

  int j = 0;
  for(int i = 0; i < noutput_items/interpolation; i++) {
    out[j --> (j + interpolation())] = function(in[i]);
    j += interpopation();
  }
  return noutput_items;
}
```

Work Functions

gr_block (see: gr_pfb_arb_resampler_ccf)

- No set relationship of input to output
  - consume(i, M);
  - produce(o, N);
  - return WORK_CALLED_PRODUCE;

```
int
foo_bar::general_work(int noutput_items,
                      gr_vector_int &ninput_items,
                      gr_vector_const_void_star &input_items,
                      gr_vector_void_star &output_items)
{
  const gr_complex *in = (gr_complex *)input_items[0];
  gr_complex *out = (gr_complex *)output_items[0];

  for(int i = 0; i < [condition]; i++) {
    out[x] = function(in[y]);
  }
  for(int i = 0; i < input_items.size(); i++) {
    consume(i, nitems[i]);
  }
  return noutput_items; // or produce(o, M) for each output 'o'
}
```

Case Study 1

gr_quadrature_demod_cf

- $<$ v3.7: **gnuradio-core/src/lib/general/**
- $\geq$ v3.7: **gr-analog/**
- Good example of a gr_sync_block.
- Uses **set_history** to ensure we can look behind us.
- $y[i] = g \cdot \tan^{-1}(x[i]x^*[i-1])$
- Always produces and consumes **noutput_items** during work.

Case Study 2

digital_costas_loop_cc

- Found in: **gr-digital/**
- A sync block with a loop.
- Inherits from **gri_control_loop**; implements:
  - advance_loop ← from current error estimate.
  - sets and gets for all control values (including: damping factor, loop bandwidth, alpha and beta gains, current frequency and phase estimates).
- Can be used with BPSK, QPSK, 8PSK.
- Two loops if second output of frequency estimate is used.
  - Done for performance reasons: reduce branches in inner loop.
- Example usage: **gr-digital/examples/example_costas.py**

Chapter the Last: Conclusions

"I never am really satisfied that I understand anything; because, understand it well as I may, my comprehension can only be an infinitesimal fraction of all I want to understand."

- Ada Lovelace

What We Covered

What We Covered

- Basic understanding of what GNU Radio is and is for.
- What the GNU Radio software package consists of.
- Some fundamental components and blocks.
- How to deal with filtering and sample rates.
- Building GNU Radio apps in Python.
- Basics of programming new blocks.

GRCon12

GNU Radio Conference 2012

- September 24 - 27
- Sheraton Gateway Hotel Atlanta Airport
- More info: www.trondeau.com/gnu-radio-conference-2012

Follows on the success of GRCon11

- Hosted by UPenn's Computer and Information Systems dept.
- Over 50 participants.
- Three days of talks and discussions.
- Huge amount of energy and excitement.
- More info: www.trondeau.com/gnu-radio-conference-2011