# USRP/GNU Radio Tutorial

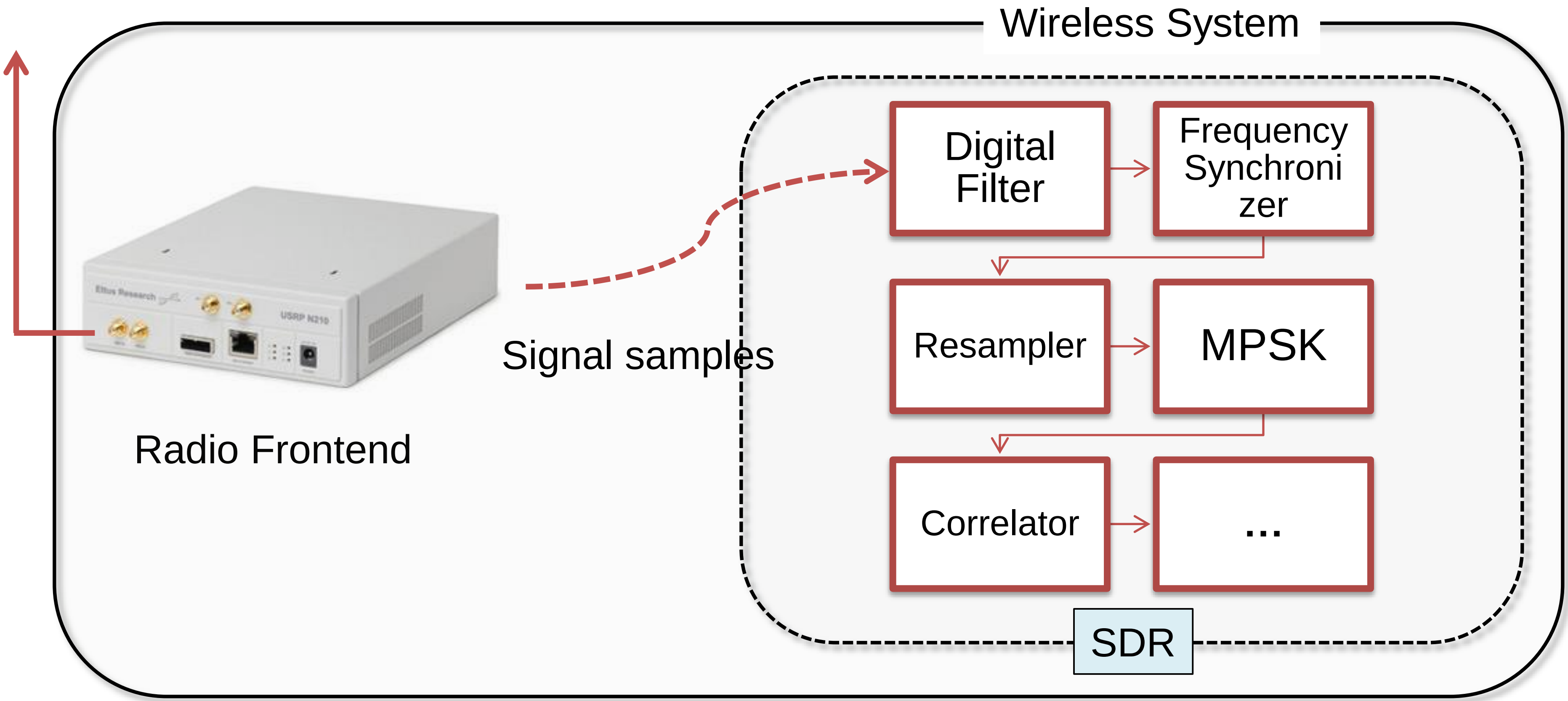
윤성로

NCSU, syoon4@ncsu.edu

http://www4.ncsu.edu/~syoon4/

# SDR (Software Defined Radio)

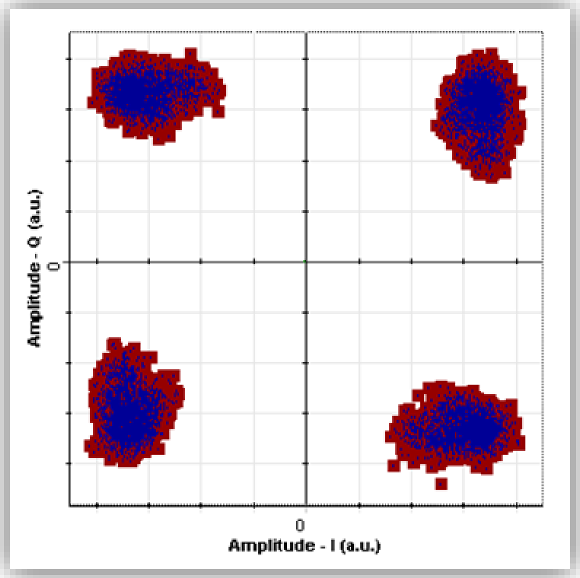
- Framework that implements radio functionalities in software

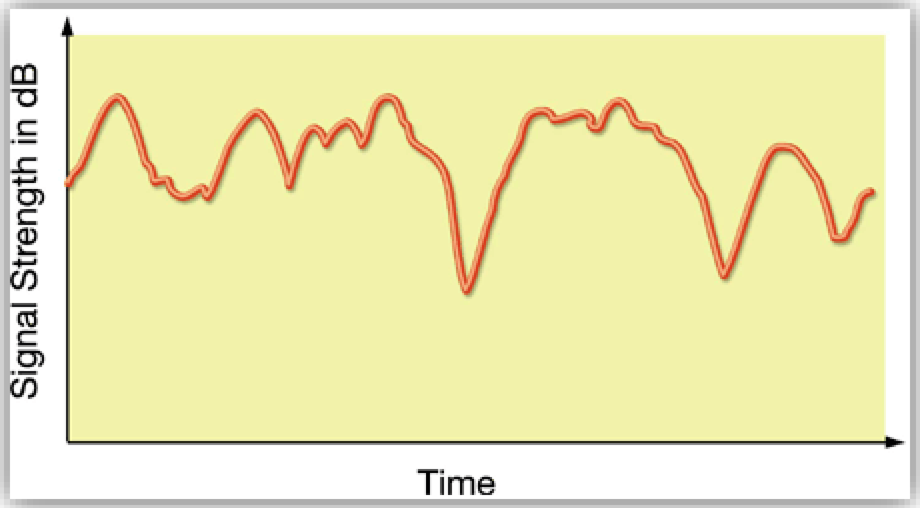


# SDR, Why use it?

* Easy & cheap to implement physical layer operations

- Physical layer information is very useful
  - Angle of arrival or channel impulse response → Indoor localization, Rate adaptation, MIMO communication
  - Channel coherence time → Mobility detection, Rate adaptation



# SDR, Why use it?

- In Protocol Design’s Perspective
  - Recent advances in physical layer technologies (e.g., MIMO, Interference Cancelation, OFDM, Frequency Domain Back-offs)
  - Existing MAC protocols doesn’t fully catch up with new physical layer technologies. Rather, physical layer is still dealt with as a black box
  - We can’t really separate between MAC and PHY when it comes to wireless networking
  - Understanding of physical layer is essential for MAC protocol innovation

# SDR, Why use it?

- But the physical layer has been EE's territory!
  - Physical layer was something magical for computer scientists
  - Understanding and developing signal processing logic was very hard
- SDR can help understanding of physical layer
  - We can learn DSP basics by reading C++ codes
  - Debug using real signal samples (I/Q, signal strengths, frequency offsets, … etc.)
- Further, SDR enables easy and quick prototyping & tests
  - Software-defined → very flexible
  - We can design whatever new protocols as we want!
  - Based on this, it's possible to design better MAC protocol

# SDR Usage Example: Research Project

- Network coding, Interference cancellation, Packet recovery…
  - MIT, using USRP / GNU Radio
- High performance MIMO w/ *N-antennas*
  - Clayton Shepard et al., using WARP (Rice University)
- Packet Recovery
  - Kun Tan et al., using SoRa (Microsoft Research Asia)

# SDR Usage Example: Research Project

- Sensor network testbed
  - WinLab, using WDR (Rutgers University)
- Rate adaptation, CSMA/CN, Indoor Localization
  - Sen Souvik et al., using USRP / GNU Radio
- Compressed sensing
  - Martin Braun et al., using USRP / GNU Radio

# SDR Usage Example: Practical Project

- 802.11 / 802.15.4 packet decoder
- FM RDS decoder
- 3GPP Tx/Rx
- RFID reader

# GNU Radio and USRP

- Common combination
  - Relatively cheap, easy to implement
  - But still powerful enough
  - Widely being used

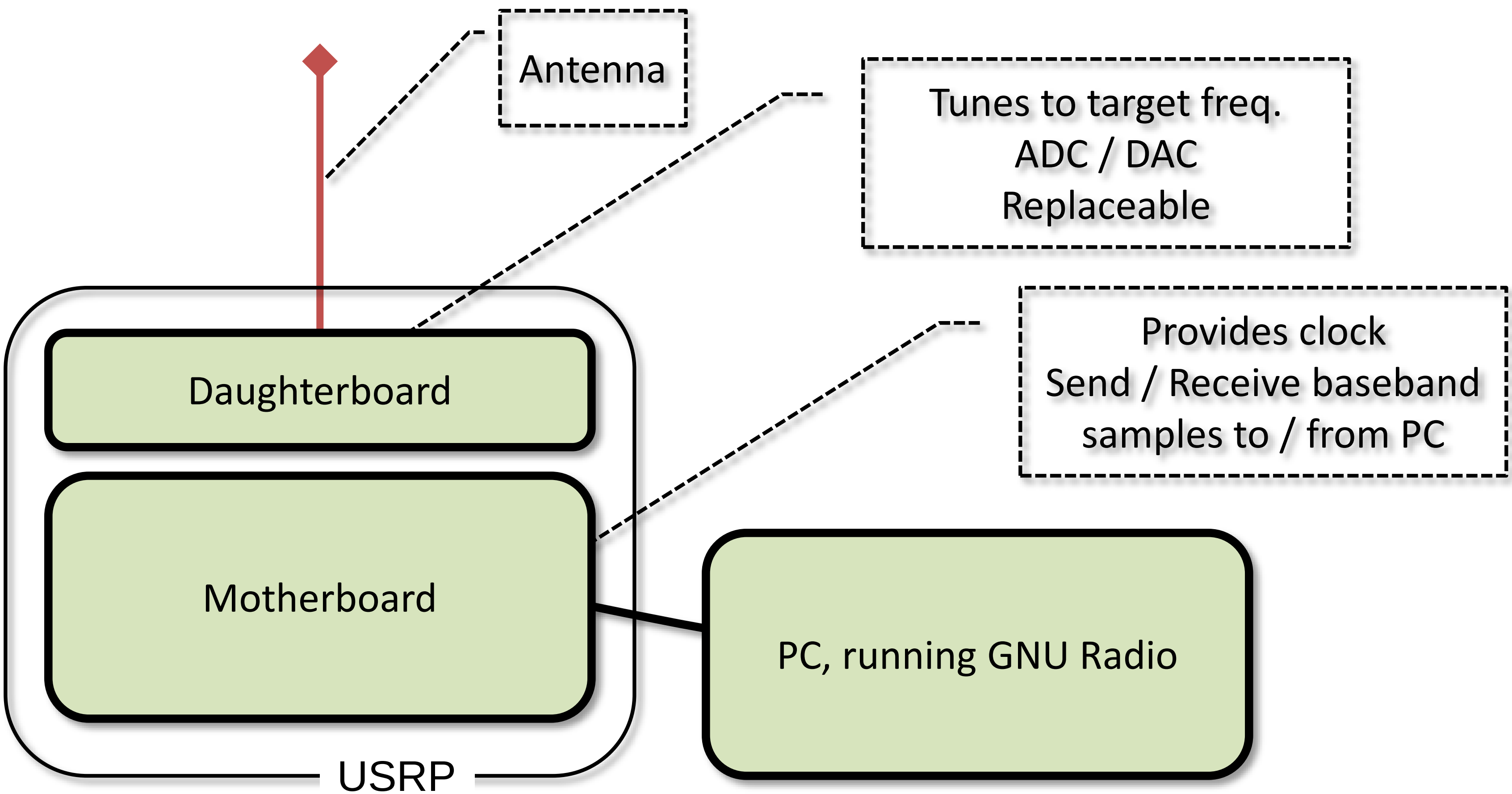
# USRP

# USRP

- Universal Software Radio Peripheral


- Operates with various software
  - GNU Radio
  - MATLAB/Simulink
  - LabVIEW (National Instruments)

# USRP Structure



# USRP Types

- USRP1 / B100
  - USB 2.0
  - 8 MSPS
- USRP N200 / N210
  - Gigabit Ethernet
  - 25 MSPS
- USRP E100 / E110
  - Own processing capability
  - 8 MSPS with ARM / Linux
  - Up to 64 MSPS with FPGA


# Antennas & Daughterboards

- Tune to a specific target frequency

- Daughterboards (DC ~ 5.9 GHz, SMA)
  - Basic Rx/Tx (DC ~ 250MHz)
  - LFTX / LFRX (DC ~ 30MHz)
  - TVRX (50 ~ 860MHz)
  - DBSRX (800 ~ 2300MHz)
  - WBX (50 ~ 2200MHz)
  - SBX (400 ~ 4400MHz)
  - XCVR2450 (2.4 ~ 2.5GHz, 4.9 ~ 5.9GHz)
  - RFX900 (750 ~ 1050 MHz)
  - …

# USRP Applications

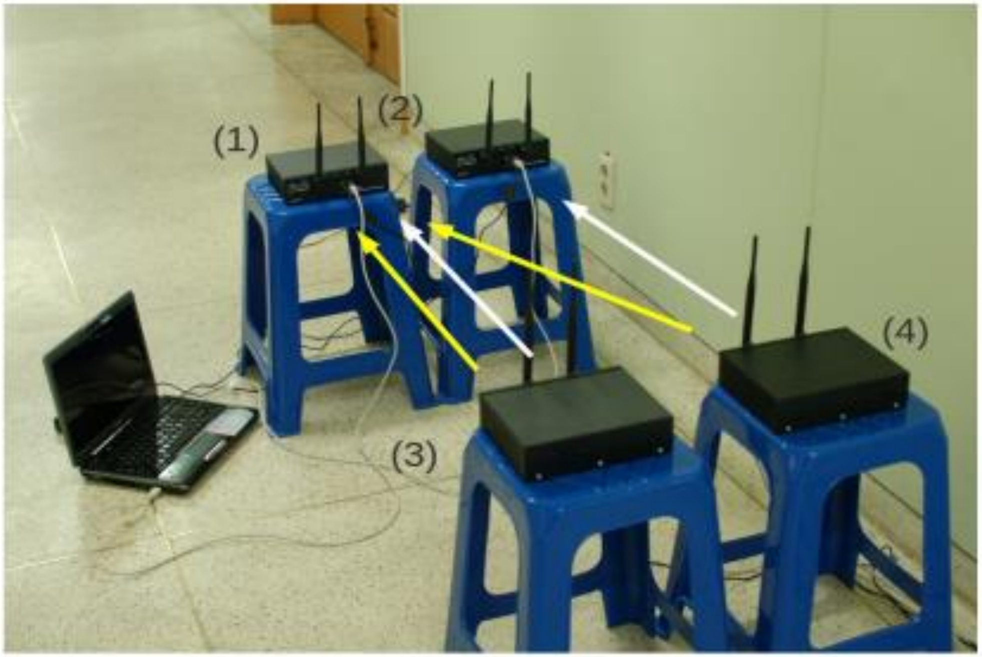
- FM Receiver
- MIMO Tx/Rx
- High-Performance MIMO Receiver
- RFID reader
- GSM base station
- Digital TV receiver
- Amateur radio
- WiFi receiver (802.11b / OFDM)

# USRP Applications

- FM Receiver

# USRP Applications

- MIMO Tx/Rx



# USRP Applications

- High-Performance MIMO Receiver

# USRP Limitations

- Communication bottleneck between PC and USRP
  - USB, Gigabit Ethernet
  - 25 MSPS
  - Limited sample rate → Hard to deal with broadband spectrum

- Software implementation runs slower than hardware
  - Tx / Rx turnaround time is very long
  - Hard to support real-time two-way communication (e.g., MAC)

# USRP Setup

1. Install UHD (universal hardware driver)
   - http://www.ettus.com/kb/category/software-documentation/installation
   - Source install
   - > *apt-get install uhd* (but not recommended)

2. Address setting
   - USRP-USB: uhd_find_devices --args="type=usrp1“
   - USRP-N: basically set to 192.168.10.2
     > *./usrp_burn_mb_eeprom* --key=ip-addr --val=XXX.XXX.XXX.XXX

# USRP Setup

3. ROM image & Firmware upload
   1. Download from http://files.ettus.com/binaries/master_images/
   2. usrp_n2xx_net_burner.py –addr=XXX.XXX.XXX.XXX –fw=<Path>
   3. usrp_n2xx_net_burner.py –addr=XXX.XXX.XXX.XXX –fpga=<Path>
   - USRP-USB: Automatic, doesn't need manual update

4. Calibration
   - **uhd_cal_rx_iq_balance:** mimimizes RX IQ imbalance
   - **uhd_cal_tx_dc_offset:** mimimizes TX DC offset
   - **uhd_cal_tx_iq_balance:** mimimizes TX IQ imbalance

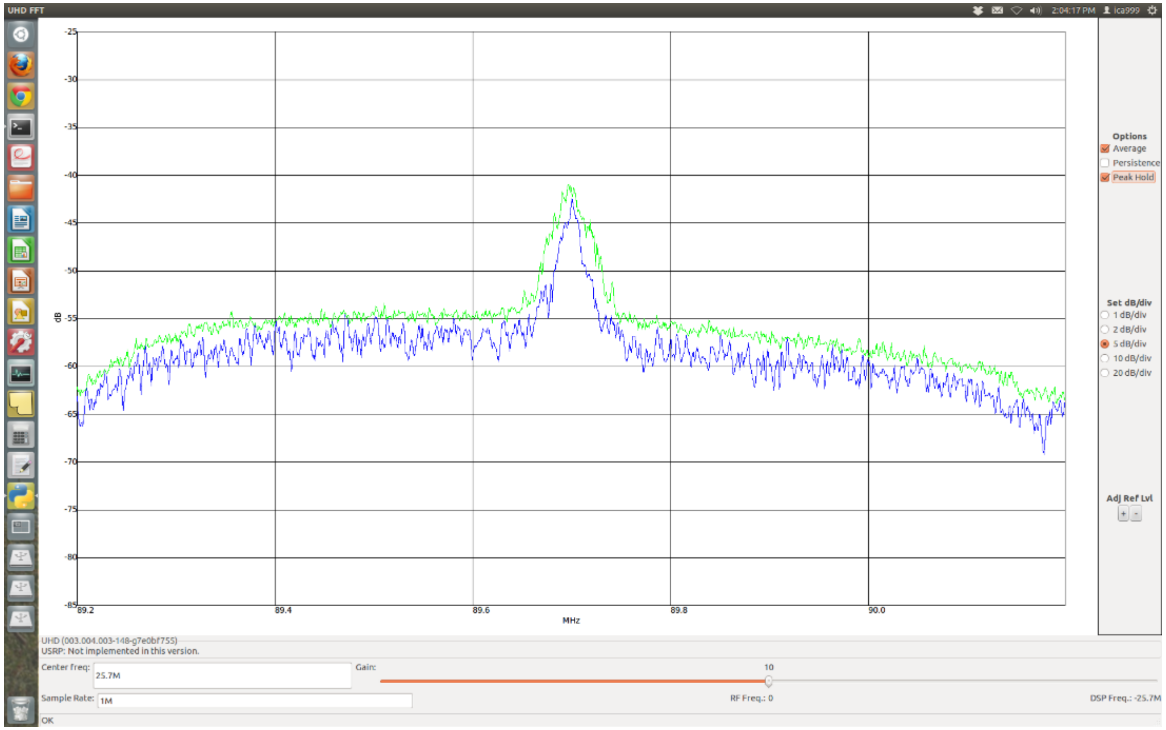
# USRP Setup

5. Test
   - At Tx: uhd_siggen –freq XX --sine
   - At Rx: uhd_fft -f XX –s YY

   - More applications: http://www.ettus.com/content/files/kb/application_note_uhd_examples.pdf

# USRP Setup

- Basic utility software, described in gnuradio.org
- uhd_fft
  - “A very simple spectrum analyzer tool which uses a connected UHD device (i.e., a USRP) to display the spectrum at a given frequency. This can be also used for a waterfall plot or as an oscilloscope.”



# USRP Setup

- uhd_rx_cfile
  - “Record an I/Q sample stream using a connected UHD device. Samples are written to a file and can be analysed off-line at a later time, using either GNU Radio or other tools such as Octave or Matlab.”
- uhd_rx_nogui
  - “Receive and listen to incoming signals on your audio device. This tool can demodulate AM and FM signals.”
- uhd_siggen{_gui}.py
  - “Simple signal generator, can create the most common signals (sine, sweep, square, noise).”
- gr_plot_XX
  - “This is an entire suite of apps which can display pre-recorded samples saved to a file. You can plot the spectra, PSD and time-domain representations of these signals.”

# GNU Radio

# GNU Radio

- GNU software library that operates USRPs
  - Official GNU projects
  - Started in 2001
  - Current version: 3.6.4
  - Current form in 2004 by MIT project

- Projects that used GNU Radio
  - https://www.cgran.org/
  - http://gnuradio.org/redmine/projects/gnuradio/wiki/OurUsers
  - http://gnuradio.org/redmine/projects/gnuradio/wiki/AcademicPapers

# GNU Radio

- Tutorials
  - http://gnuradio.org/redmine/projects/gnuradio/wiki/ExternalDocumentation
  - http://gnuradio.org/redmine/projects/gnuradio/wiki/HowToUse#Using-the-included-tools-and-utility-programs
  - http://gnuradio.org/redmine/projects/gnuradio/wiki#I-Getting-started
- Class list
  - C++: http://gnuradio.org/doc/doxygen/index.html
  - Python: http://gnuradio.org/doc/sphinx/index.html
  - Automatically generated documentations (from comments in source codes)
- Examples
  - /usr/local/share/gnuradio/examples/

# Where to start?

- Easier than normal software projects
  - Data & data structure is already there

- Difficulty for CS people
  - Lack of DSP basics
- Difficulty for EE people
  - Software structure, how to develop / build

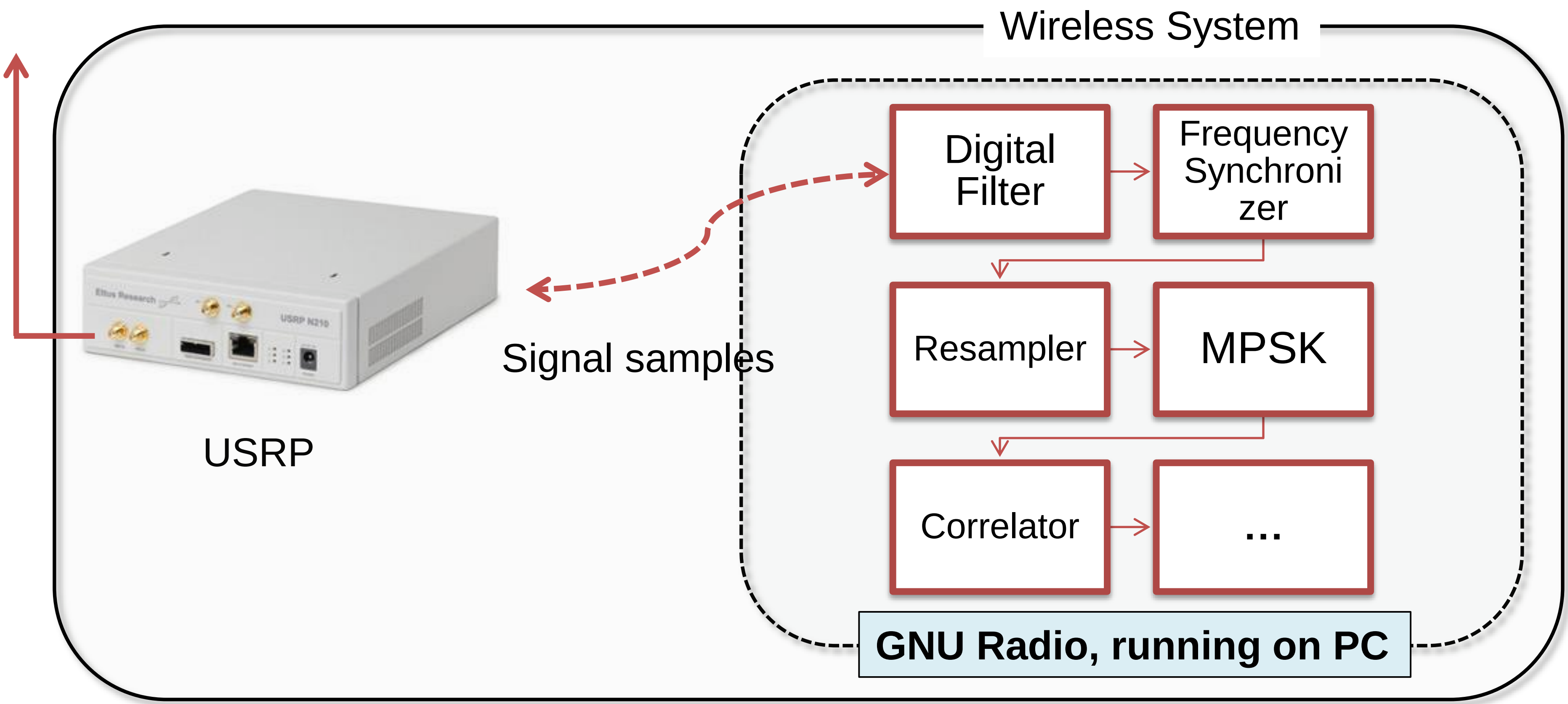
- Well, documentation isn't very complete…

# Lack of sound documentations…

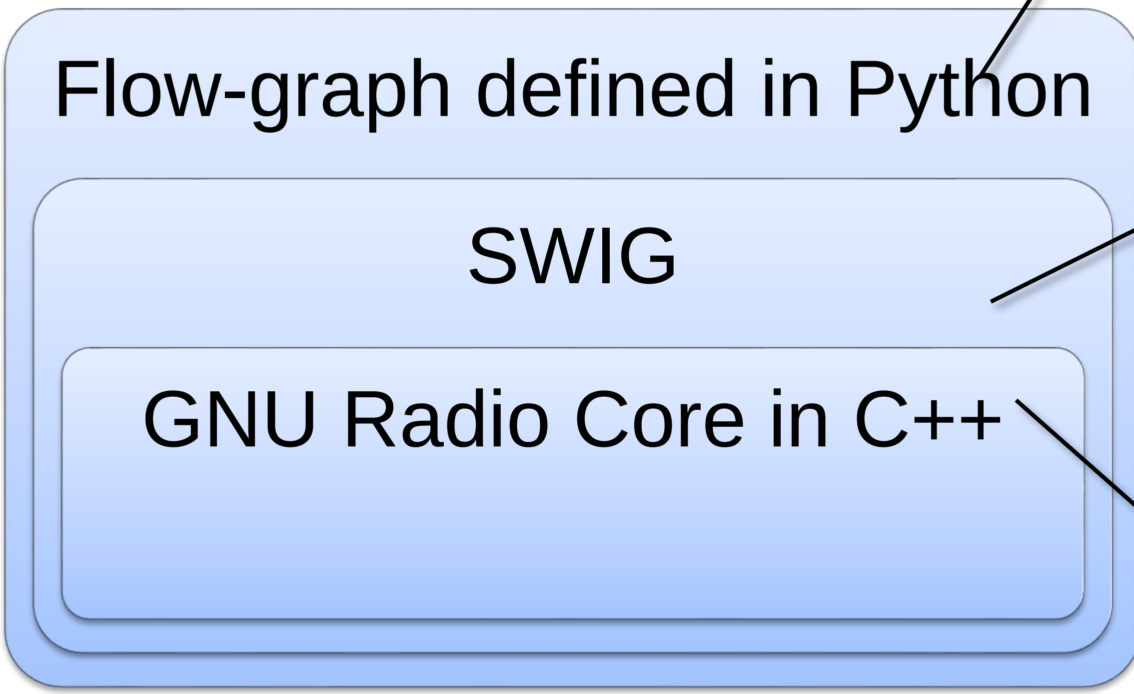
- "…GNU Radio code changes a lot, so creating a static documentation would not be very sensible. GNU Radio uses Doxygen and Sphinx to dynamically create documentation of the APIs."

- "…If you feel GNU Radio should really already have some functionality you want to use, either browse through the module directory Python uses or go through the source directory of GNU Radio. In particular, pay attention to the directories starting with gr- in the source directory, such as gr-trellis. These produce their own code and, consequently, their own modules…"

# The big picture (1/2)



# The big picture (2/2)




- High level programming
- Flexibility
- Interactivity
- Rapid prototyping

Automated wrapping of C++ interfaces for the use in Python

Low-level DSP
Good Performance
Shared library

# Scope

- Signal processing library
  - Mathematical operations
  - Filter (low-pass, band-pass)
  - Correlator
  - Modulator / demodulator
  - I/O
  - …
- Example applications
- Test tools

# gnuradio-core

- **AGC**
- **Filter**
- **Synchronizer (PLL/FLL)**
- **Interpolator / resampler**
- **FFT**
- **MPSK**
- **Packetizer**
- **Correlator / descrambler**
- Codec (RS, Viterbi)
- IO routines (disk, Ethernet, sound…)
- Runtime routines (message handler, scheduler…)
- Complex arithmetic operators (+, -, x, /, max, min)
- Random number generator
- Type converter

# GNU Radio -- How to install & run

# Installation

- Where to get?
  - “git clone git://gnuradio.org/gnuradio” → newest stable release
  - Archive → can choose from different versions

- Where to install?
  - Linux/Unix
    - ✓ For the best performance, install GNU Radio on console-based Linux
  - Cygwin
  - Windows

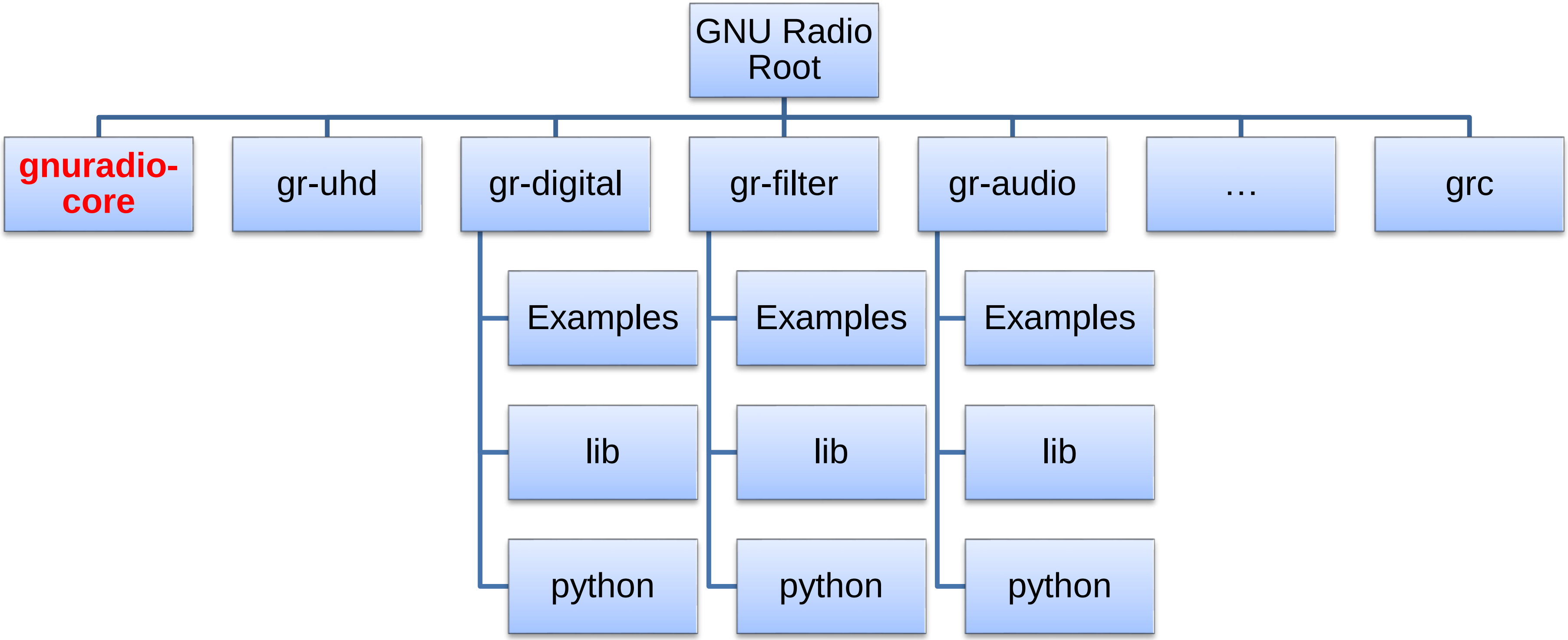
# Installation

- How to build?
  - cd $GNURadioRoot
  - cmake .
  - make
  - make test
  - sudo make install

- Dependencies
  - Boost
  - Numpy, scipy
  - …

# Scope

- Hierarchy



# How to write a program

# How to write a program

1. **Identify what I really want to implement**
2. Identify which DSP blocks are needed
3. Implement DSP function in C++
4. Create SWIG interface between python and C++
5. Compile and install the implemented library
6. Call the function from python

# How to write a program - Shortcut

Basic configuration using **open source projects**

Use USRP as a signal receiver

Dump whatever you've received into a file
- Wireless comm. → complex
- Sound → real number

Use MATLAB to process the samples → develop an algorithm

Integrate the algorithm into the C++ code

Configure Python Code

# How to add a module

- gr_modtool
  - > gr_modtool *newmod* module_name

```
apps   cmake   CMakeLists.txt   docs   grc   include   lib   python   swig
```

✓ Apps: application files

✓ cmake: configuration related files

✓ grc: gnuradio-companion codes

✓ Include: public header files

✓ Lib: c++ implementation and private header files

✓ Swig: SWIG interface (.i) files

✓ Docs: module description generated via doxygen

# How to add a module

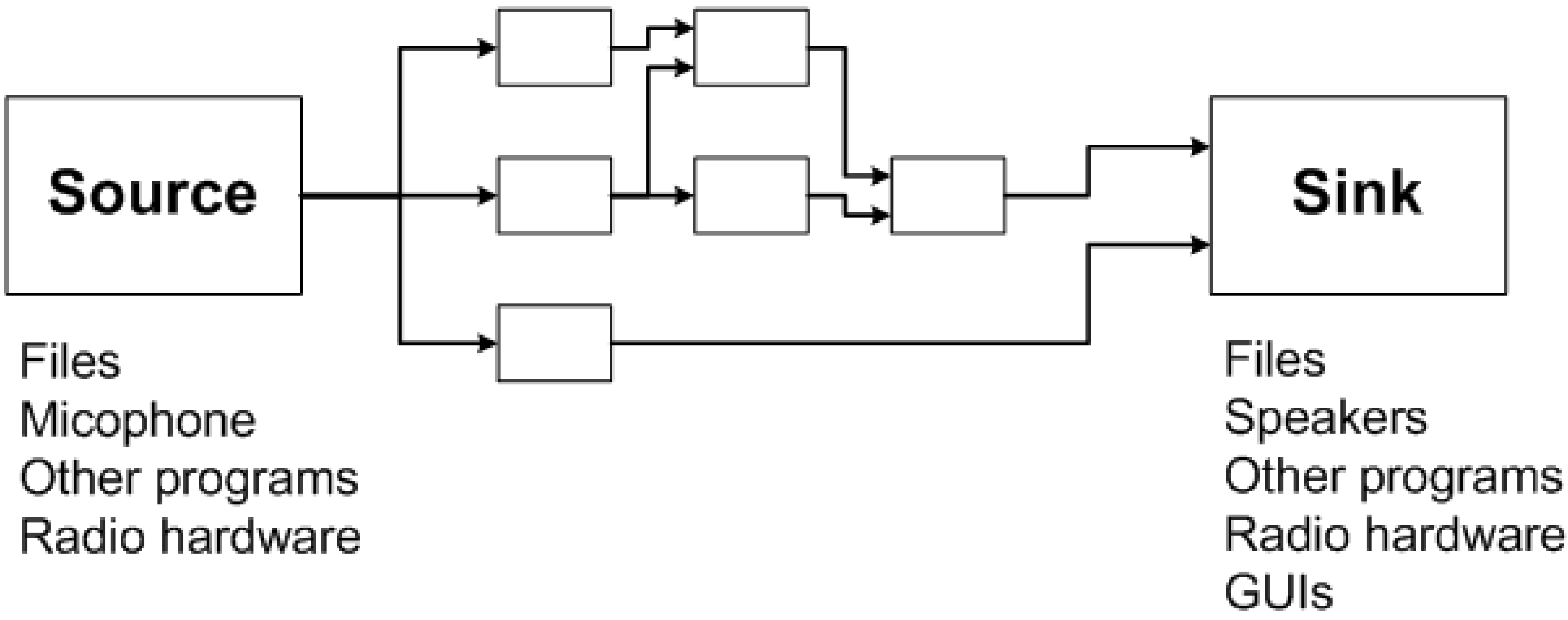
- > gr_modtool *add –t general* block_name
  - ✓ Automatically adds python / C++ / SWIG files and configures Makefile

```
GNU Radio module name identified: howto
Language: C++
Block/code identifier: square_ff
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n]
Add C++ QA code? [y/N]
Adding file 'square_ff_impl.h'...
Adding file 'square_ff_impl.cc'...
Adding file 'square_ff.h'...
Editing swig/howto_swig.i...
Adding file 'qa_square_ff.py'...
Editing python/CMakeLists.txt...
Adding file 'somework.xml'...
Editing grc/CMakeLists.txt...
```

- > gr_modtool *makexml* module_name
  - For gnuradio-companion

# Flow graph

- Defines how signal data is going to be processed
- Python part defines the flow graph



# Flow graph

- Block (node)
  - Nodes within the flow graph
  - Performs one (and only one) particular signal processing task
    - ✓ E.g., filtering, demodulating, correlating…
  - At least one source / sink block pair is needed

- Data path (edge)
  - Data comes in and goes out as a continuous stream
  - Cannot be changed during the runtime
  - The flow graph defines which function blocks the data stream would go through

# Flow graph

- Item
  - The data exchanged between two blocks
  - Usually complex signal samples but can be any number, bits, etc.
  - The type should be same between any output / input port pair
  - **Dynamic scheduler** redirects the output item stream from to the next block (circular buffer)

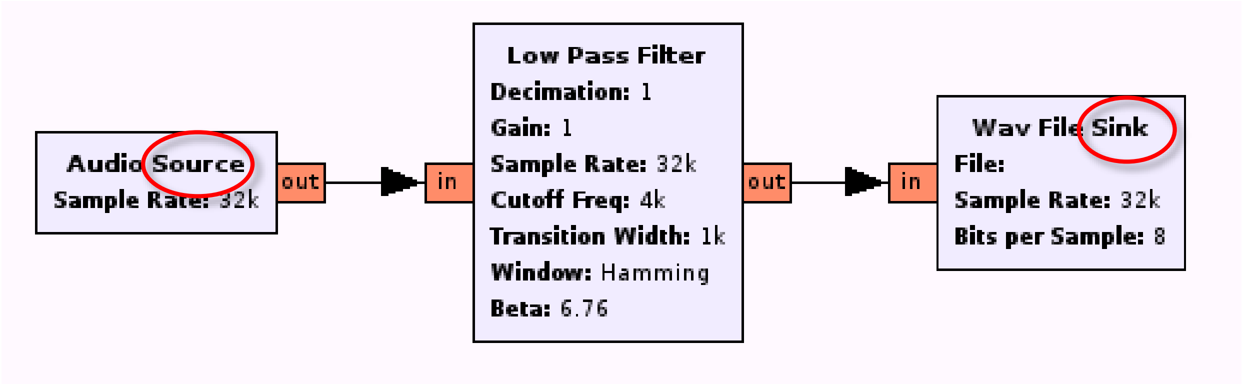
# Flow graph

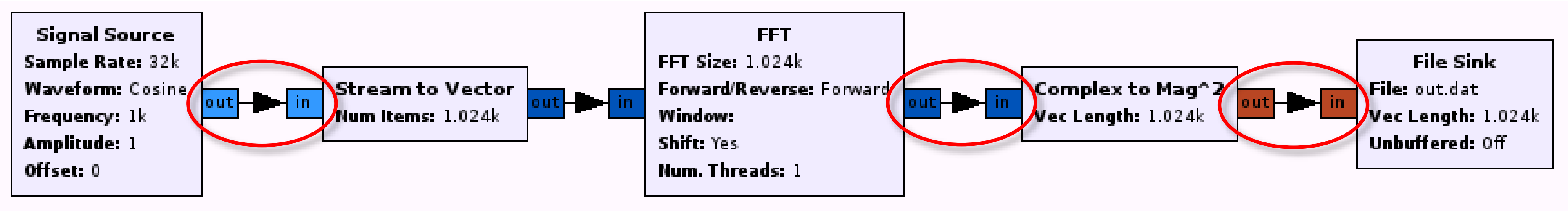
- Basic principle
  1. Identify required signal processing blocks
     - ✓ If needed, write a new block
  2. Define the data path between the blocks
  3. Set the program parameters (sample rate, filter bandwidth, etc.)

# GNU Radio Companion

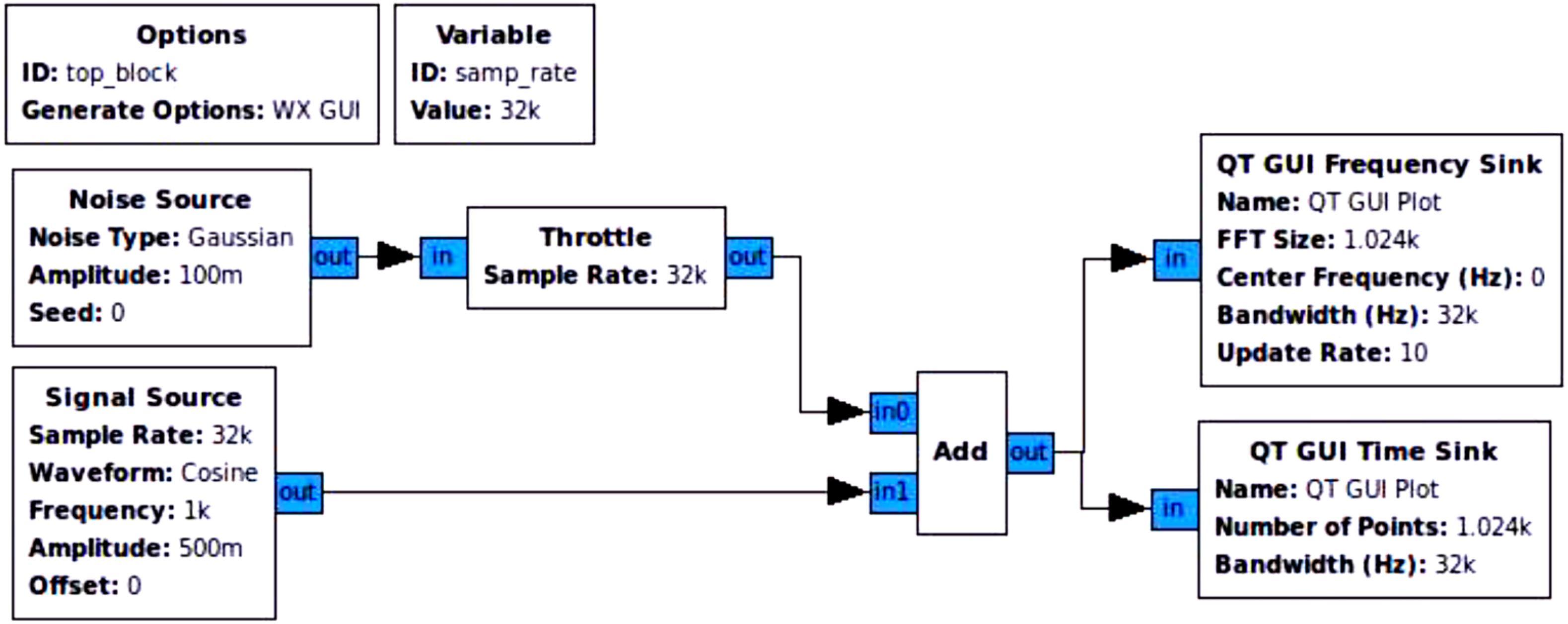
- Graphical interface to create the flow graph
- Easily assembles signal processing blocks

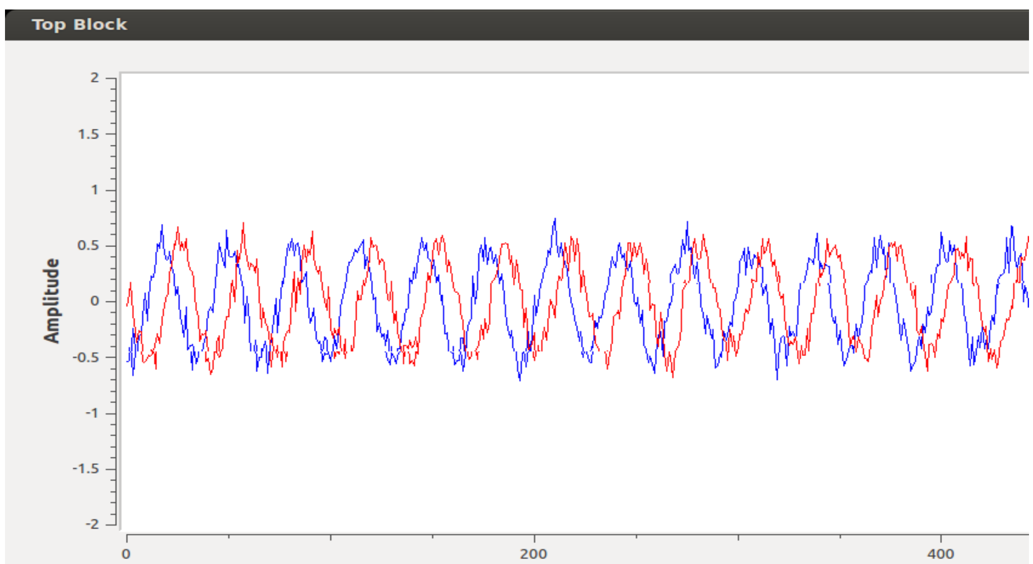
  > *gnuradio-companion* XX.grc



# GNU Radio Companion



# Manual Python Programming

- Import modules

```
#!/usr/bin/env python

from gnuradio import gr
from gnuradio import audio

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

        sample_rate = 32000
        ampl = 0.1

        src0 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 350, ampl)
        src1 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 440, ampl)
        dst = audio.sink (sample_rate, "")
        self.connect (src0, (dst, 0))
        self.connect (src1, (dst, 1))

if __name__ == '__main__':
    try:
        my_top_block().run()
    except [[KeyboardInterrupt]]:
        pass
```

# Manual Python Programming

- Import modules

| | |
|---|---|
| gr | The main GNU Radio library. You will nearly always need this. |
| audio | Soundcard controls (sources, sinks). You can use this to send or receive audio to the sound cards, but you can also use your sound card as a narrow band receiver with an external RF frontend. |
| blks2 | This module contains additional blocks written in Python which include often-used tasks like modulators and demodulators, some extra filter code, resamplers, squelch and so on. |
| digital | Anything related to digital modulation. |
| fft | Anything related to FFTs. |
| optfir | Routines for designing optimal FIR filters. |
| plot_data | Some functions to plot data with Matplotlib |
| wxgui | This is actually a submodule, containing utilities to quickly create graphical user interfaces to your flow graphs. Use `from gnuradio.wxgui import *` to import everything in the submodule or `from gnuradio.wxgui import stdgui2, fftsink2` to import specific components. See the section 'Graphical User Interfaces' for more information. |
| eng_notation | Adds some functions to deals with numbers in engineering notation such as @100M' for 100 * 10^6'. |
| eng_options | Use `from gnuradio.eng_options import eng_options` to import this feature. This module extends Pythons `optparse` module to understand engineering notation (see above). |
| gru | Miscellaneous utilities, mathematical and others. |

# Manual Python Programming

- Create a top block and initialize
  - Inherit from gr.top_block and call initializer __init__(self)

```
#!/usr/bin/env python

from gnuradio import gr
from gnuradio import audio

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

        sample_rate = 32000
        ampl = 0.1

        src0 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 350, ampl)
        src1 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 440, ampl)
        dst = audio.sink (sample_rate, "")
        self.connect (src0, (dst, 0))
        self.connect (src1, (dst, 1))

if __name__ == '__main__':
    try:
        my_top_block().run()
    except [[KeyboardInterrupt]]:
        pass
```

# Manual Python Programming

- gr.top_block
  - Top-level block that initiates the flow graph
  - Initializes signal processing blocks
  - Defines connection between the blocks

**Public Member Functions**

| | |
|---|---|
| | **~gr_top_block** () |
| void | **run** (int max_noutput_items=100000)<br>The simple interface to running a flowgraph. |
| void | **start** (int max_noutput_items=100000) |
| void | **stop** () |
| void | **wait** () |
| virtual void | **lock** () |
| virtual void | **unlock** () |
| void | **dump** () |
| int | **max_noutput_items** ()<br>Get the number of max noutput_items in the flowgraph. |
| void | **set_max_noutput_items** (int nmax)<br>Set the maximum number of noutput_items in the flowgraph. |
| **gr_top_block_sptr** | **to_top_block** () |

# Manual Python Programming

- How to control the flow graph
  - run() : “The simplest way to run a flow graph. Calls start(), then wait(). Used to run a flow graph that will stop on its own, or to run a flow graph indefinitely until SIGINT is received.”
  - start() : “Start the contained flow graph. Returns to the caller once the threads are created.”
  - stop() : “Stop the running flow graph. Notifies each thread created by the scheduler to shutdown, then returns to caller.”
  - wait() : “Wait for a flow graph to complete. Flowgraphs complete when either (1) all blocks indicate that they are done, or (2) after stop has been called to request shutdown.”
  - lock() : “Lock a flow graph in preparation for reconfiguration.”
  - unlock() : “Unlock a flow graph in preparation for reconfig\uration. When an equal number of calls to lock() and unlock() have occurred, the flow graph will be restarted automatically.”

# Manual Python Programming

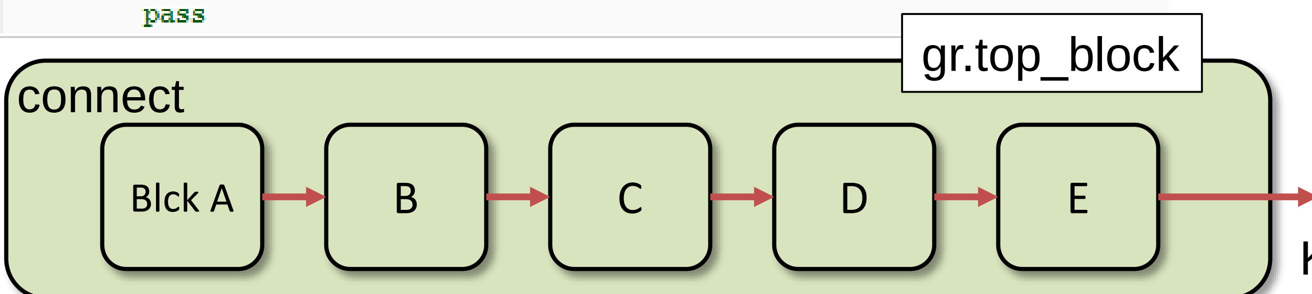
- Define blocks & configure parameters

```
#!/usr/bin/env python

from gnuradio import gr
from gnuradio import audio

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

        sample_rate = 32000
        ampl = 0.1

        src0 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 350, ampl)
        src1 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 440, ampl)
        dst = audio.sink (sample_rate, "")
        self.connect (src0, (dst, 0))
        self.connect (src1, (dst, 1))

if __name__ == '__main__':
    try:
        my_top_block().run()
    except [[KeyboardInterrupt]]:
        pass
```

# Manual Python Programming

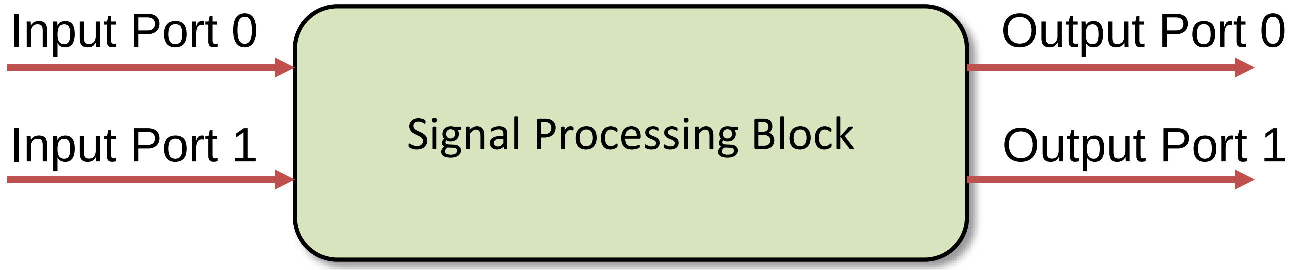
- Connect blocks

```python
#!/usr/bin/env python

from gnuradio import gr
from gnuradio import audio

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

        sample_rate = 32000
        ampl = 0.1

        src0 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 350, ampl)
        src1 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 440, ampl)
        dst = audio.sink (sample_rate, "")
        self.connect (src0, (dst, 0))
        self.connect (src1, (dst, 1))

if __name__ == '__main__':
    try:
        my_top_block().run()
    except [[KeyboardInterrupt]]:
        pass
```



# Manual Python Programming

- Connect
  - Connects output of a block to an input of a next block
  - One output can connect to multiple inputs
  - Can specify **target port** for both input and output
    - ✓ connect((A, 0), (B, 0))
      connect((A, 1), (B, 2))
      …
  - Port: entry point of items into a signal processing block
    - ✓ Just one unless explicitly defined
    - ✓ connect(A, B) is equal to connect((A, 0), (B, 0))



# Manual Python Programming

- More on block setting… (Filter)

```
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)
```

# Manual Python Programming

- More on block setting… (Filter)

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

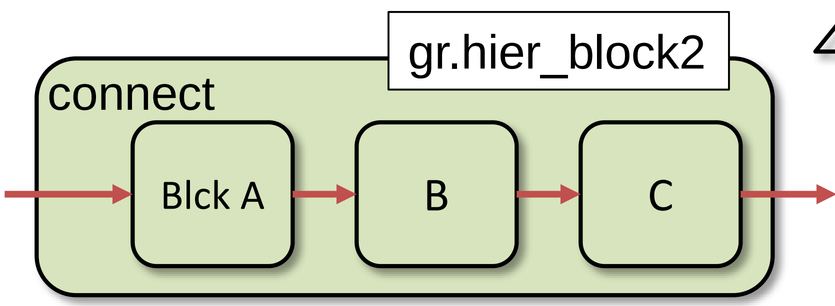
```
gr.firdes.low_pass_2(gain, sample rate,
                     bandwidth, transition band,
                     stopband attenuation (dB))
```

- **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

# Hier block

- From Filename import **HierBlock**

```python
class HierBlock(gr.hier_block2):
    def __init__(self, audio_rate, if_rate):
        gr.hier_block2.__init__(self, "HierBlock",
                                gr.io_signature(1, 1, gr.sizeof_float),
                                gr.io_signature(1, 2, gr.sizeof_gr_complex))

        B1 = gr.block1(...) # Put in proper code here!
        B2 = gr.block2(...)

        self.connect(self, B1, B2, self)
```




- Input / Output to block
- (Minimum, maximum number of ports, size of item)

# Hier block

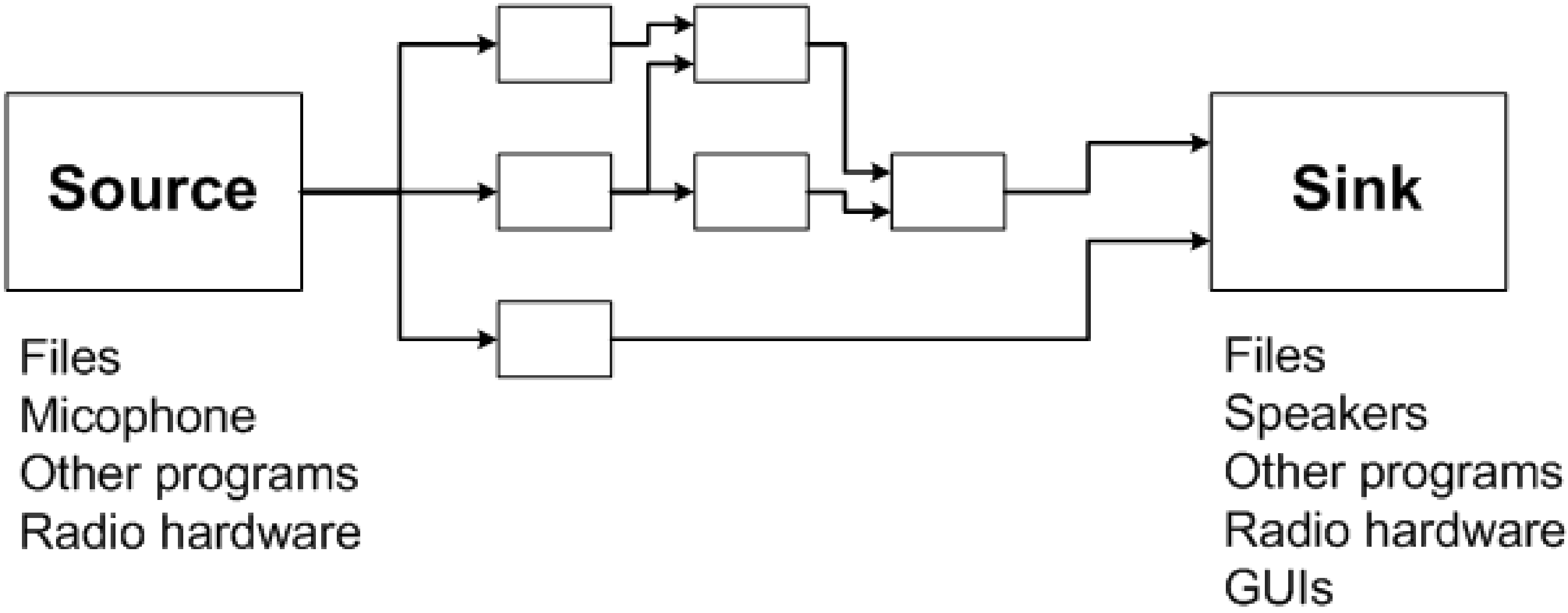
```
class transmit_path(gr.hier_block2):
    def __init__(self):
            gr.hier_block2.__init__(self, "transmit_path",
                    gr.io_signature(0, 0, 0), # Null signature
                    gr.io_signature(0, 0, 0))

        source_block = gr.source()
        signal_proc  = gr.other_block()
        sink_block   = gr.sink()

        self.connect(source_block, signal_proc, sink_block)

class receive_path(gr.hier_block2):
    def __init__(self):
            gr.hier_block2.__init__(self, "receive_path",
```

# Hier block

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

    tx_path = transmit_path()

    rx_path = receive_path()

    self.connect(tx_path)
    self.connect(rx_path)
```

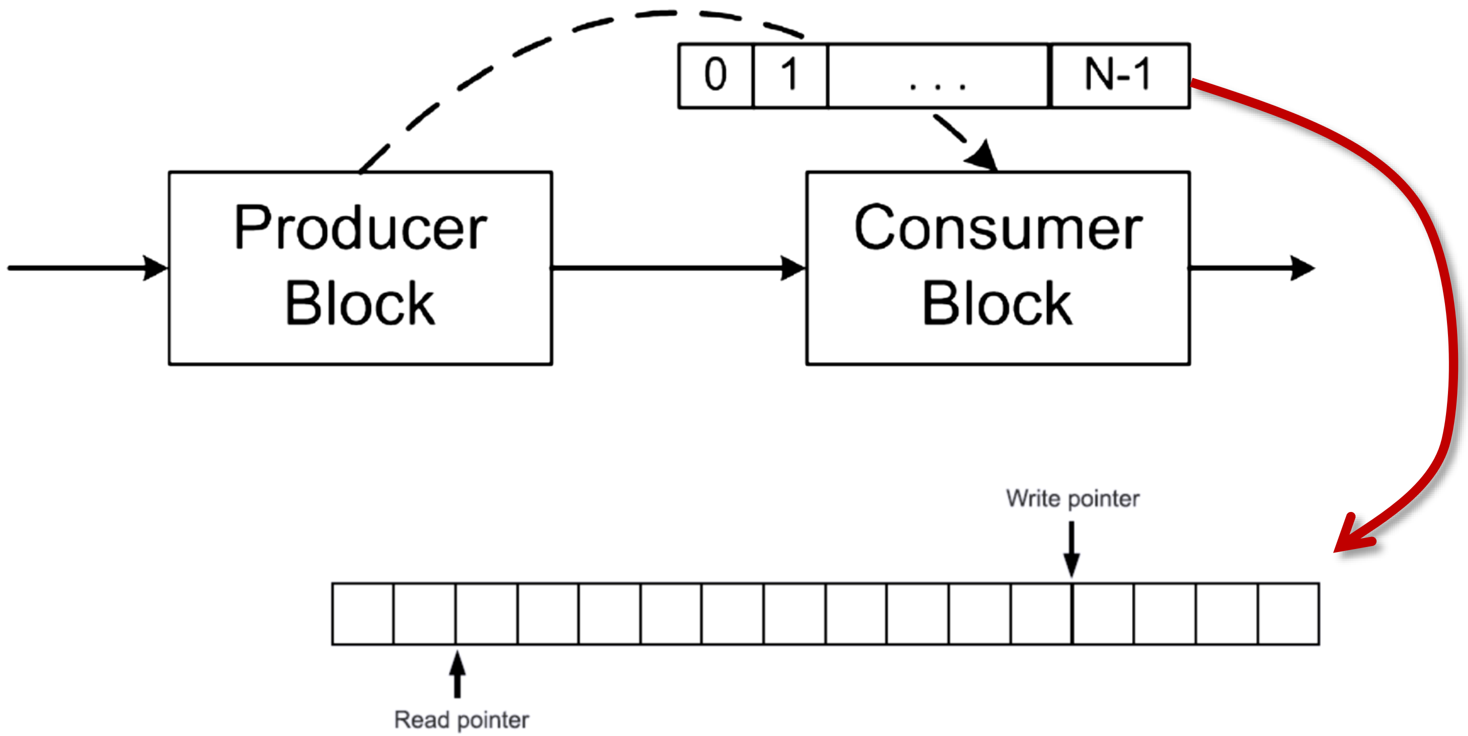
# Signal Processing Block Implementation

- Each block implements one signal processing functionality
  - A block ***consumes*** an input, processes it and ***produces*** an output
  - Source only produces & Sink only consumes



# Signal Processing Block Implementation

- Implemented as an infinite loop
  - ***dynamic scheduler*** delivers input and output between blocks
- Child class of gr_block




Rondeau, GNU Radio Tutorial, 2012

# Signal Processing Block Implementation

- Class Definition

```
typedef boost::shared_ptr<somework> somework_sptr;
somework_sptr somework_ff ();

class somework : public gr_block
{
private:
   friend somework_sptr somework_ff ();
   somework ();
   ...
}
```

# Signal Processing Block Implementation

- Block Initialization

```
somework:: somework() : gr_block("somework",
            gr_make_io_signature(1, 1, sizeof (float)), // input signature
            gr_make_io_signature(1, 1, sizeof (float))) // output signature
{
      …
}
```

  - gr_make_io_signature(minimum_port, maximum_port, item_size)

# Signal Processing Block Implementation

- virtual int gr_block::general_work(int noutput,
  gr_vector_int& ninput,
  gr_vector_const_void_star& input,
  gr_vector_void_star& output);

- Dynamic scheduler repeatedly calls general_work to perform signal processing
  - Each function call takes 4 parameters
  - Returns the number of items processed during the function call

# Signal Processing Block Implementation

- gr-digital/lib/digital_constellation_receiver_cb.cc
  - Second order PLL (Costas loop)

```
int digital_constellation_receiver_cb::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 = (const gr_complex *) input_items[0];
  unsigned char *out = (unsigned char *) output_items[0];
  int i=0;
  float phase_error;
  unsigned int sym_value;
  gr_complex sample, nco;

  float *out_err = 0, *out_phase = 0, *out_freq = 0;
  if(output_items.size() == 4) {
    out_err = (float *) output_items[1];
    out_phase = (float *) output_items[2];
    out_freq = (float *) output_items[3];
  }

```

Can define the number of input items per port stream

Input and output vector of pointer to each stream

Checks the number of output ports and process accordingly

# Signal Processing Block Implementation

```
  while((i < noutput_items) && (i < ninput_items[0])) {
    sample = in[i];
    nco = gr_expj(d_phase);    // get the NCO value for derotating the current sample
    sample = nco*sample;       // get the downconverted symbol

    sym_value = d_constellation->decision_maker_pe(&sample, &phase_error);
    phase_error_tracking(phase_error);  // corrects phase and frequency offsets

    out[i] = sym_value;

    if(output_items.size() == 4) {
      out_err[i] = phase_error;
      out_phase[i] = d_phase;
      out_freq[i] = d_freq;
    }
    i++;
  }
  consume_each(i);
  return i;
}
```

- Checks the number of output ports and process accordingly
- Assign values to the output circular buffer

- void consume (int which_input, int how_many_items);
- void consume_each (int how_many_items);

# Signal Processing Block Implementation

- In case (input : output) isn't 1:1
  - E.g., Down-sampling or interpolating
  - In case of 4x down-sampling:

```
virtual void forecast (int noutput_items,
        gr_vector_int   &ninput_items_required)
{
        ninput_items_required = noutput_items * 4;
}
```

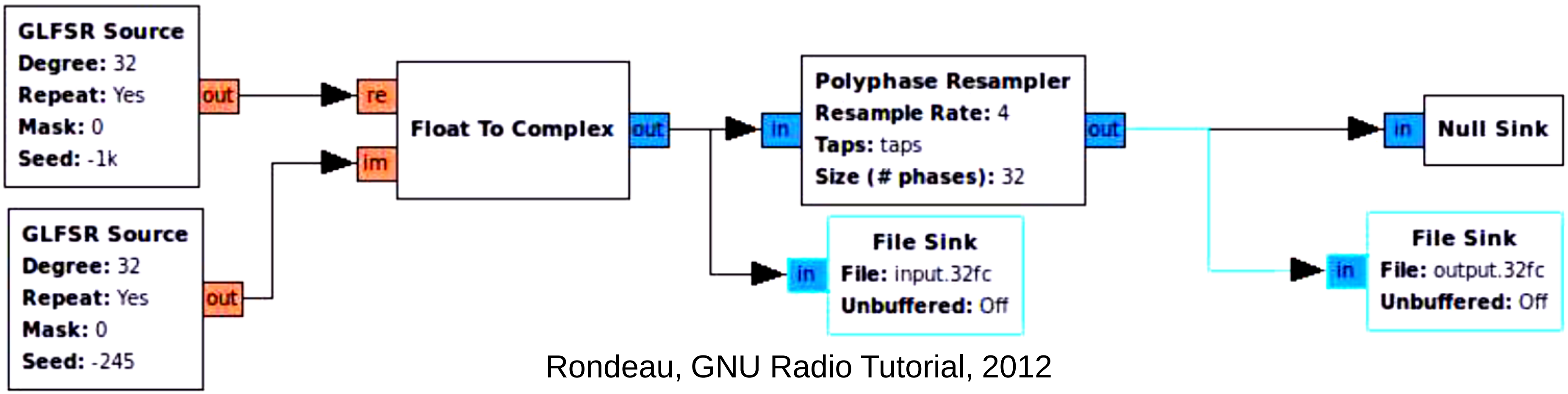
# SWIG

- gr_float_to_int.i
  - Automatically generated by gr_modtool *add –t general* block_name
  - Expose C++ functions for the use in Python

```
GR_SWIG_BLOCK_MAGIC(gr, float_to_int)

gr_float_to_int_sptr
gr_make_float_to_int (size_t vlen=1, float scale=1);

class gr_float_to_int : public gr_sync_block
{
public:
  float scale() const;
  void set_scale(float scale);
};
```

# How to Debug

- Use file sink to dump whatever intermediate products into files
  - Connect output to ***file sink***




Rondeau, GNU Radio Tutorial, 2012

- Then use MATLAB (or Octave) to load the file and analyze if anything is wrong
- Native utility: gt_plot_XXX.py

# How to Debug

- self.connect(self._bits, self._rrc, gr.file_sink(4, "bit.dat"))
  - Float or int
- self.connect(self._f2c, gr.file_sink(8, "wave.dat"))
  - Complex (i&q signal sample)

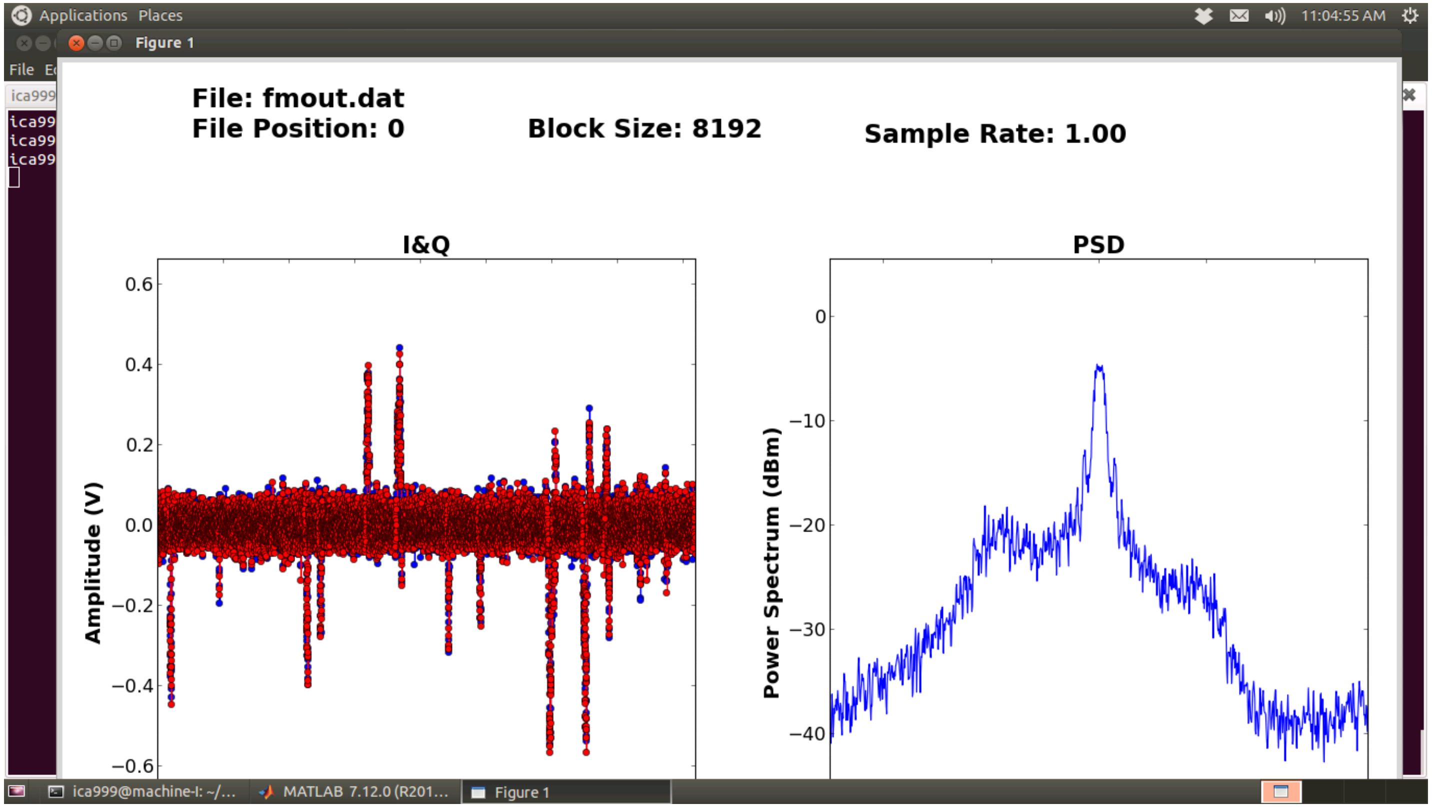
# How to Debug

- Matlab code example

```
f = fopen (filename, 'rb');
 if (f < 0)
   v = 0;
 else
   t = fread (f, [2, count], 'float');
   fclose (f);
   v = t(1, :) + t(2, :) * i;
   [r, c] = size (v);
   v = reshape (v, c, r);
 end

 F = fft(v);
 plot(abs(fftshift(F)));
```

# How to Debug

- GNU Radio native utility
  - gr_plot_psd
  - gr_plot_fft



# Test-driven programming

- GNU Radio supports test-driven programming:
  **gr_unittest**

```
> gr_modtool add -t general block_name
      Automatically adds python / C++ / SWIG files and
      configures Makefile
         GNU Radio module name identified: howto
         Language: C++
         Block/code identifier: square_ff
         Enter valid argument list, including default
         arguments:
         Add Python QA code? [Y/n]
```

  - qa_XXX.py is automatically generated

# Test-driven programming

- qa_float_to_short.py

```
from gnuradio import gr, gr_unittest
import ctypes

class test_float_to_short (gr_unittest.TestCase):
    def setUp (self):
        self.tb = gr.top_block ()

    def tearDown (self):
        self.tb = None

    def test_001(self):
        src_data = (0.0, 1.1, 2.2, 3.3, 4.4, 5.5,
             -1.1, -2.2, -3.3, -4.4, -5.5)
        expected_result = [0, 1, 2, 3, 4, 6,
             -1, -2, -3, -4, -6]
             ...
```

# Test-driven programming

```
            ...
src = gr.vector_source_f(src_data)
op = gr.float_to_short()
dst = gr.vector_sink_s()

self.tb.connect(src, op, dst)
self.tb.run()
result_data = list(dst.data())

self.assertEqual(expected_result, result_data)

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

# Test-driven programming

- test_float_to_short.xml

```
<testsuite errors="0" failures="0"
name="unittest.suite.TestSuite" tests="3" time="0.011">
  <testcase classname="__main__.test_float_to_short"
name="test_001" time="0.0029"></testcase>
  <testcase classname="__main__.test_float_to_short"
name="test_002" time="0.0017"></testcase>
  <testcase classname="__main__.test_float_to_short"
name="test_003" time="0.0059"></testcase>
  <system-out><![CDATA[]]></system-out>
  <system-err><![CDATA[]]></system-err>
</testsuite>
```

# Quick Summary

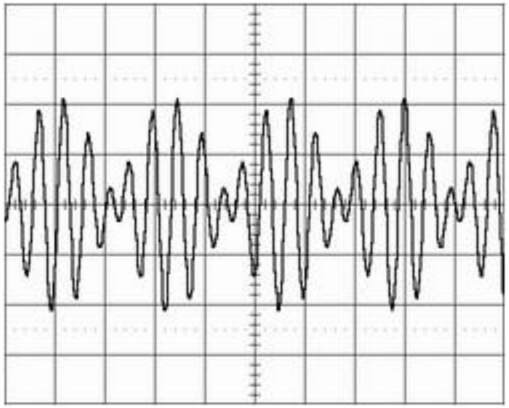
- Python
  - Configures flow graph (e.g., gnuradio-companion)
  - Inherits gr.top_block
  - Connect signal processing blocks

- SWIG
  - Glues Python with C++ (GNU Radio core library)

- C++
  - Implements operations of each signal processing block
  - While loop takes care of incoming samples
  - Dynamic scheduler delivers the items between connected blocks
  - Inherits gr_block

# Example Projects

# Example Project – Dial tone generator



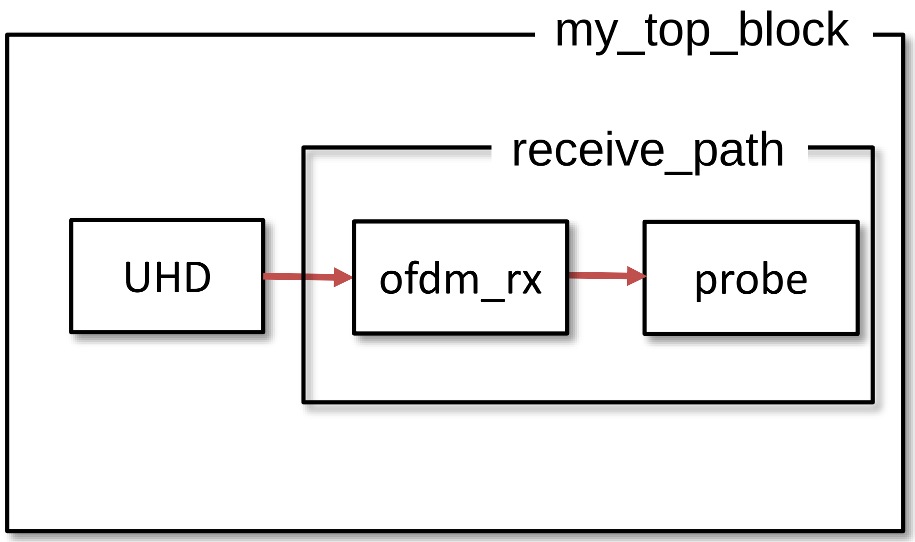
```python
from gnuradio import gr
from gnuradio.eng_option import eng_option
from optparse import OptionParser

class dial_tone_source(gr.top_block):
    def __init__(self, host, port, pkt_size, sample_rate, eof):
        gr.top_block.__init__(self, "dial_tone_source")

        amplitude = 0.3
        src0 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 350, amplitude)
        src1 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 440, amplitude)
        add = gr.add_ff()

        # Throttle needed here to account for the other side's audio card sampling rate
        thr = gr.throttle(gr.sizeof_float, sample_rate)
        sink = gr.udp_sink(gr.sizeof_float, host, port, pkt_size, eof=eof)
        self.connect(src0, (add, 0))
        self.connect(src1, (add, 1))
        self.connect(add, thr, sink)
```

# Example Project – Dial tone generator

```
if __name__ == '__main__':
    parser = OptionParser(option_class=eng_option)
    parser.add_option("", "--host", type="string", default="localhost",
                      help="Remote host name (domain name or IP address")
    parser.add_option("", "--port", type="int", default=65500,
                      help="port number to connect to")
    parser.add_option("", "--packet-size", type="int", default=1472,
                      help="packet size.")
    parser.add_option("-r", "--sample-rate", type="int", default=8000,
                      help="audio signal sample rate [default=%default]")
    parser.add_option("", "--no-eof", action="store_true", default=False,
                      help="don't send EOF on disconnect")
    (options, args) = parser.parse_args()
    if len(args) != 0:
        parser.print_help()
        raise SystemExit, 1

    # Create an instance of a hierarchical block
    top_block = dial_tone_source(options.host, options.port,
```

# Example Project – Packet Receiver

- Application structure



# Example Project – Packet Receiver

- benchmark_rx.py

```
class my_top_block(gr.top_block):
    def __init__(self, callback, options):
        gr.top_block.__init__(self)

        if(options.rx_freq is not None):
            self.source = uhd_receiver(options.args,
                                       options.bandwidth,
                                       options.rx_freq, options.rx_gain,
                                       options.spec, options.antenna,
                                       options.verbose)
        elif(options.from_file is not None):
            self.source = gr.file_source(gr.sizeof_gr_complex, options.from_file)
        else:
            self.source = gr.null_source(gr.sizeof_gr_complex)

        # Set up receive path
        # do this after for any adjustments to the options that may
        # occur in the sinks (specifically the UHD sink)
        self.rxpath = receive_path(callback, options)

        self.connect(self.source, self.rxpath)
```

# Example Project – Packet Receiver

```
def main():

    global n_rcvd, n_right

    n_rcvd = 0
    n_right = 0

    def rx_callback(ok, payload):
        global n_rcvd, n_right
        n_rcvd += 1
        (pktno,) = struct.unpack('!H', payload[0:2])
        if ok:
            n_right += 1
        print "ok: %r \t pktno: %d \t n_rcvd: %d \t n_right: %d" % (ok, pktno, n_rcvd, n_right)
```

```
    # build the graph
    tb = my_top_block(rx_callback, options)

    r = gr.enable_realtime_scheduling()
    if r != gr.RT_OK:
        print "Warning: failed to enable realtime scheduling"

    tb.start()                        # start flow graph
    tb.wait()                         # wait for it to finish

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
```

# Example Project – Packet Receiver

- receive_path.py

```
class receive_path(gr.hier_block2):
    def __init__(self, rx_callback, options):

        gr.hier_block2.__init__(self, "receive_path",
                                gr.io_signature(1, 1, gr.sizeof_gr_complex),
                                gr.io_signature(0, 0, 0))


        options = copy.copy(options)    # make a copy so we can destructively modify

        self._verbose     = options.verbose
        self._log         = options.log
        self._rx_callback = rx_callback      # this callback is fired when there's a packe

        # receiver
        self.ofdm_rx = digital.ofdm_demod(options,
                                          callback=self._rx_callback)

        # Carrier Sensing Blocks
        alpha = 0.001
        thresh = 30   # in dB, will have to adjust
        self.probe = gr.probe_avg_mag_sqrd_c(thresh,alpha)

        self.connect(self, self.ofdm_rx)
        self.connect(self.ofdm_rx, self.probe)
```

# Example Project – Packet Receiver

- gr_probe_avg_mag_sqrd_c.cc

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

  for (int i = 0; i < noutput_items; i++){
    double mag_sqrd = in[i].real()*in[i].real() + in[i].imag()*in[i].imag();
    d_iir.filter(mag_sqrd);     // computed for side effect: prev_output()
  }

  d_unmuted = d_iir.prev_output() >= d_threshold;
  d_level = d_iir.prev_output();
  return noutput_items;
}
```