GNU Radio

Introduction and Computational Capabilities of the Open Source GNU Radio Project

Thomas W. Rondeau

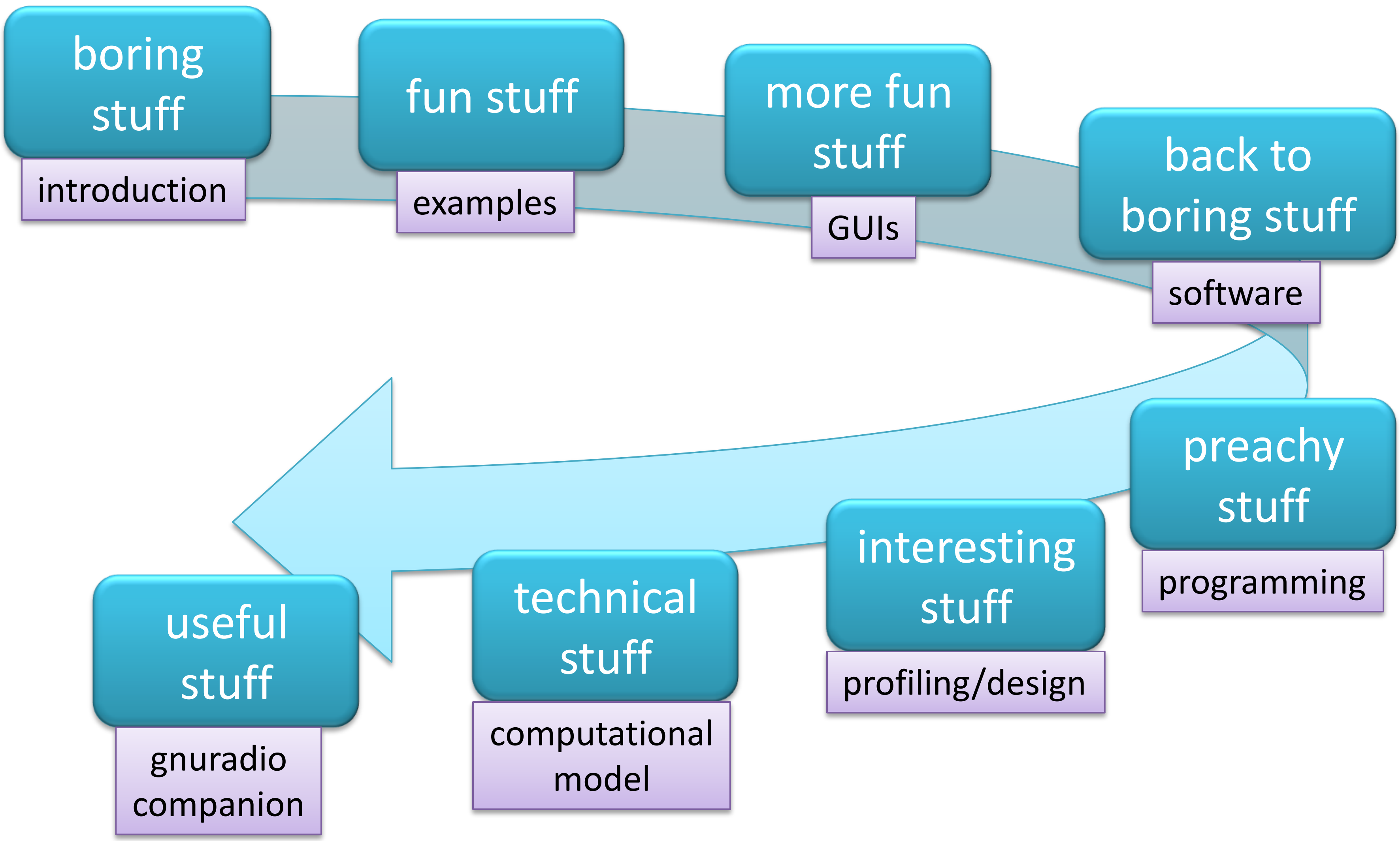
GNU Radio maintainer

SDR Technical Conference, 2010

Tutorial Scope

- An overview of GNU Radio and its purpose and capabilities
- A look inside to see how it works
- Understanding of the computational models, methods, and processes behind the software
- An appreciation for its multidisciplinary nature

Tutorial Outline

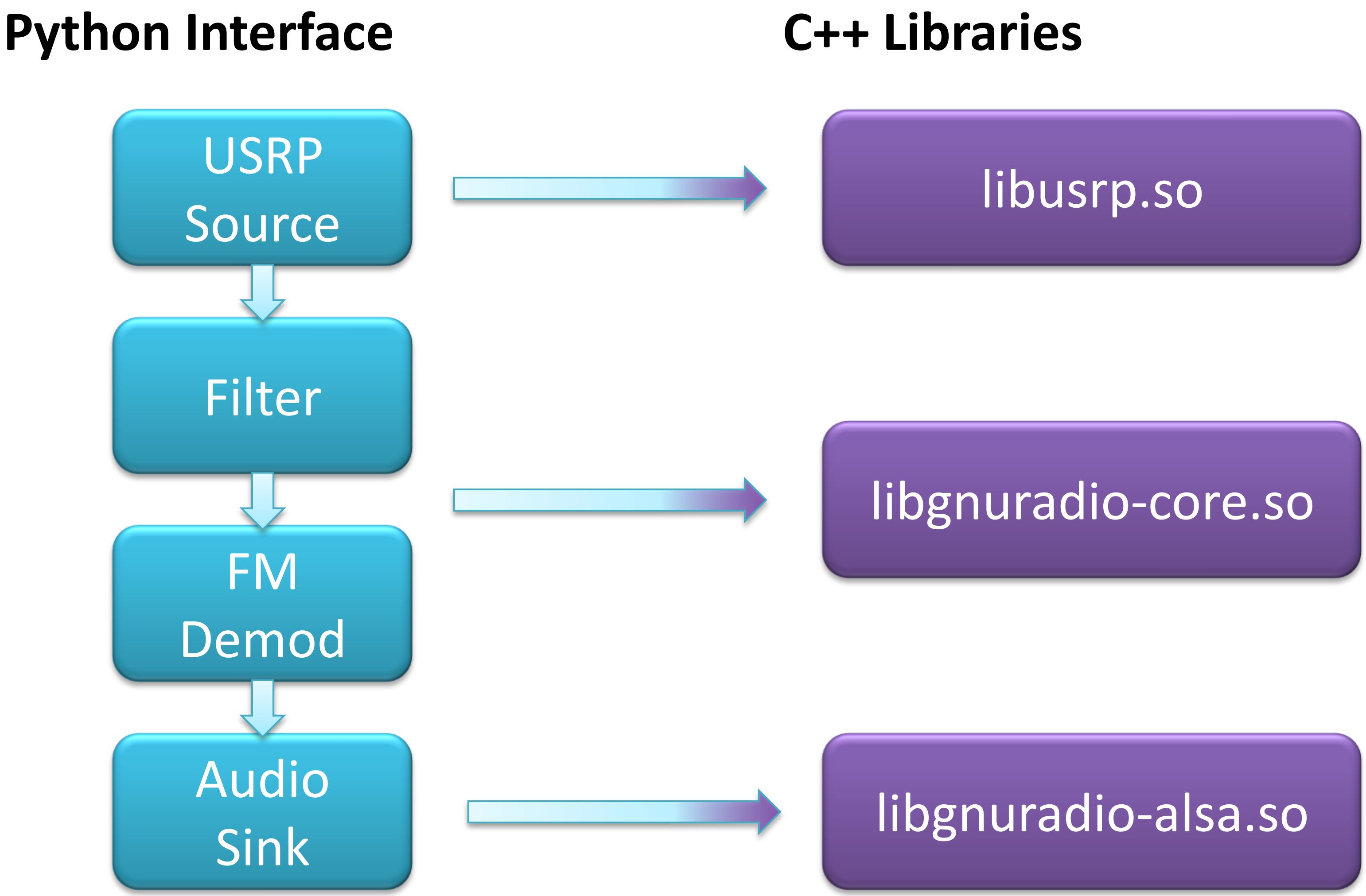


OPENING INTRODUCTION

GNU Radio
gnuradio.org

- Open source software radio
- Provides the scheduler for real-time operation
- Includes:
  - Many signal processing blocks
  - Interfaces to a few radio front ends
  - Graphical user interfaces (GUI)
  - Examples
- A platform to build and explore radios (or any other communications platform)

Python on top; C++ underneath



Structuring the Python

```
from gnuradio import gr
class myblock(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)
        self.block1 = gr.<block>
        self.block2 = gr.<block>

        self.connect(self.block1,
                     self.block2)
```

- Get the namespace
- Inherit from top_block
- Class constructor
- Call top_block constructor
- Create some GNU Radio blocks
- Connect blocks

Using the Python class

```
def main():
    tb = myblock()

    tb.start()

    tb.wait()
```

- Some function to use the block
- Instantiate a myblock object
- Start the flowgraph
- Block until it finishes

FM EXAMPLE WALKTHROUGH

ANALYSIS TOOLS

Visualization is an important part of analysis and debugging

Off-line tools:

Scipy: www.scipy.org

Matplotlib: matplotlib.sourceforge.net

On-line tools:

wxPython GUI: www.wxpython.org

QT GUI: qt.nokia.com/products

www.riverbankcomputing.co.uk

qwt.sourceforge.net

Basic Matplotlib Plotting

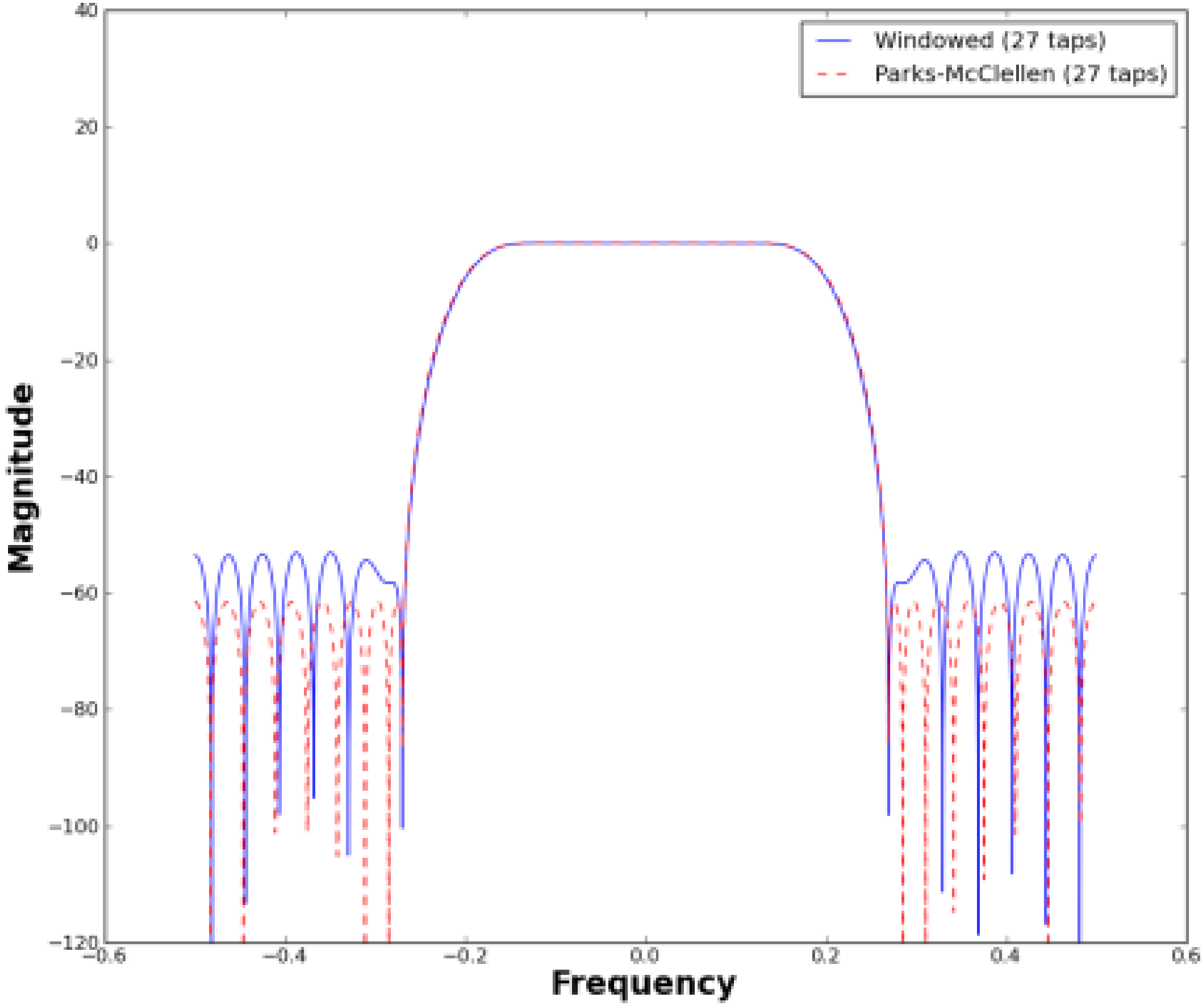
```
import scipy, pylab
t = scipy.arange(0, 1, 0.001)
x = scipy.cos(2*scipy.pi*(100)*t)
y = scipy.sin(2*scipy.pi*(100)*t)

fig = pylab.figure(1)
sp = fig.add_subplot(1,1,1)
p1 = sp.plot(t, x, “b-”, linewidth=2, label=“func1”)
p2 = sp.plot(t, y, “r-o”, linewidth=2, label=“func2”)
sp.legend()
pylab.show()
```

Using Matplotlib with GNU Radio

- Use gr.head to stop graph after N items
  - gr.head(gr.sizeof_gr_complex, N)
- Use gr.vector_sink_c to store data
  - self.vsink = gr.vector_sink_c()
  - After graph has run:
    - self.vsink.data() returns the data as a Python list
- We can now plot all *N* items of vsink

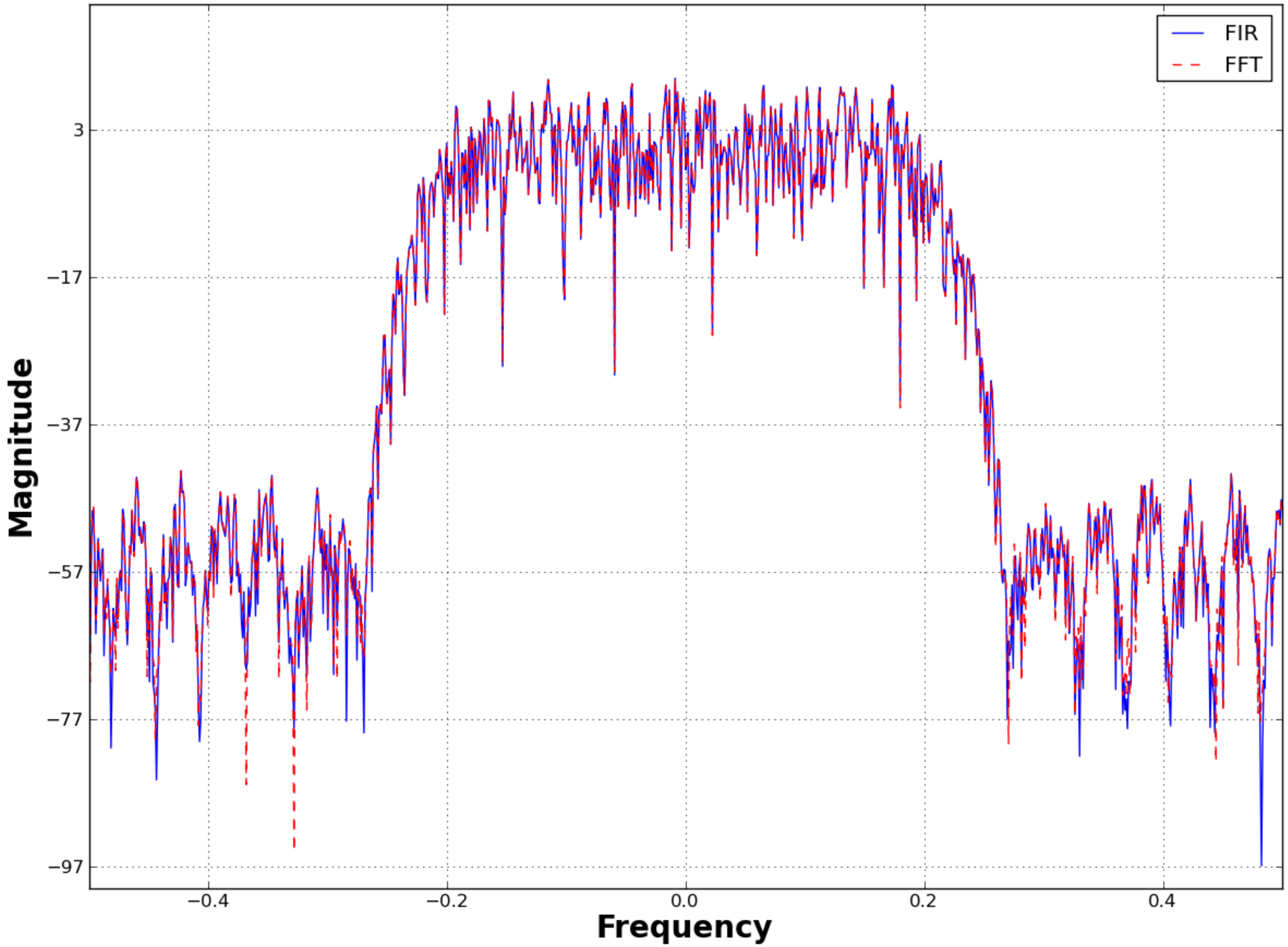
Matplotlib Output Examples:

Plotting filter impulse responses



Matplotlib Output Examples:

Filtering noise



USING MATPLOTLIB WITH FM EXAMPLE

The wx and QT GUI's add on-line support for visualization.

from gnuradio.qtgui import qtgui

```
self.qtsink = qtgui.sink_c(fftsize, window, fc, Rs, title,
                    fft, waterfall, waterfall3D, time, const, parent)
```

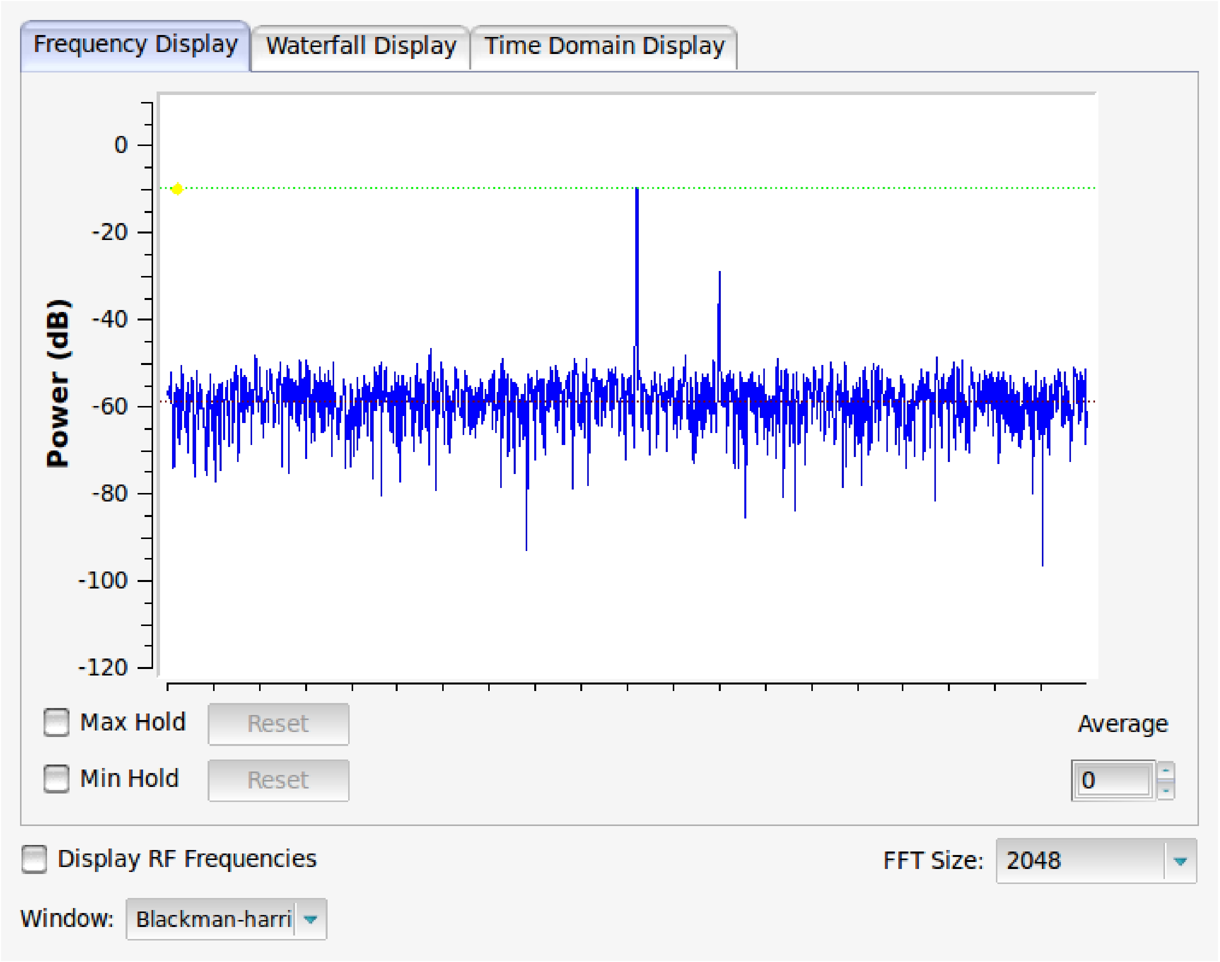
Set up with an initial FFT size, window function, center frequency, sample rate, and window title.

Remaining arguments turn on/off the different plots

Can also set a parent to work in with other QT apps

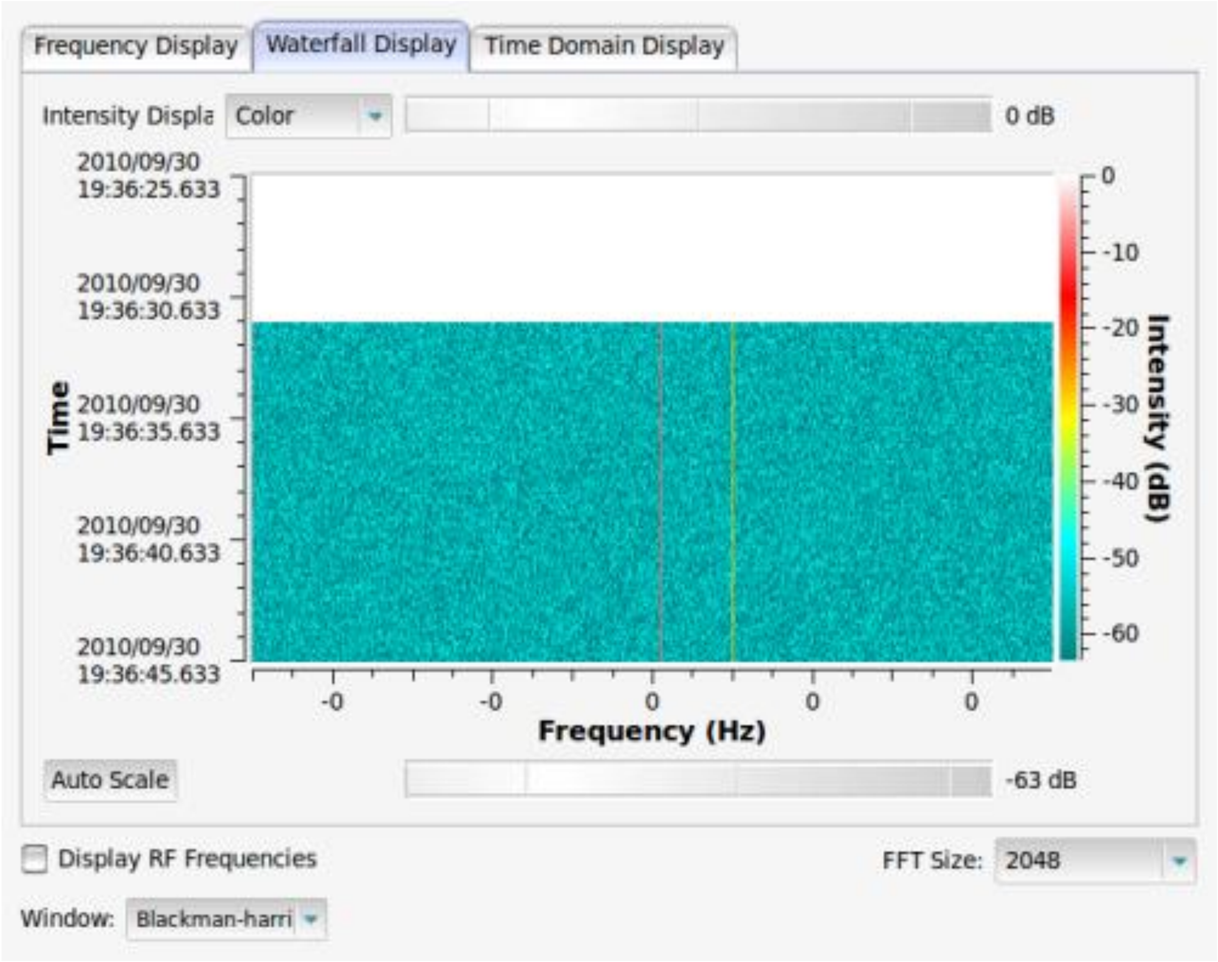
The QT GUI output offers multiple views:

FFT (or PSD)



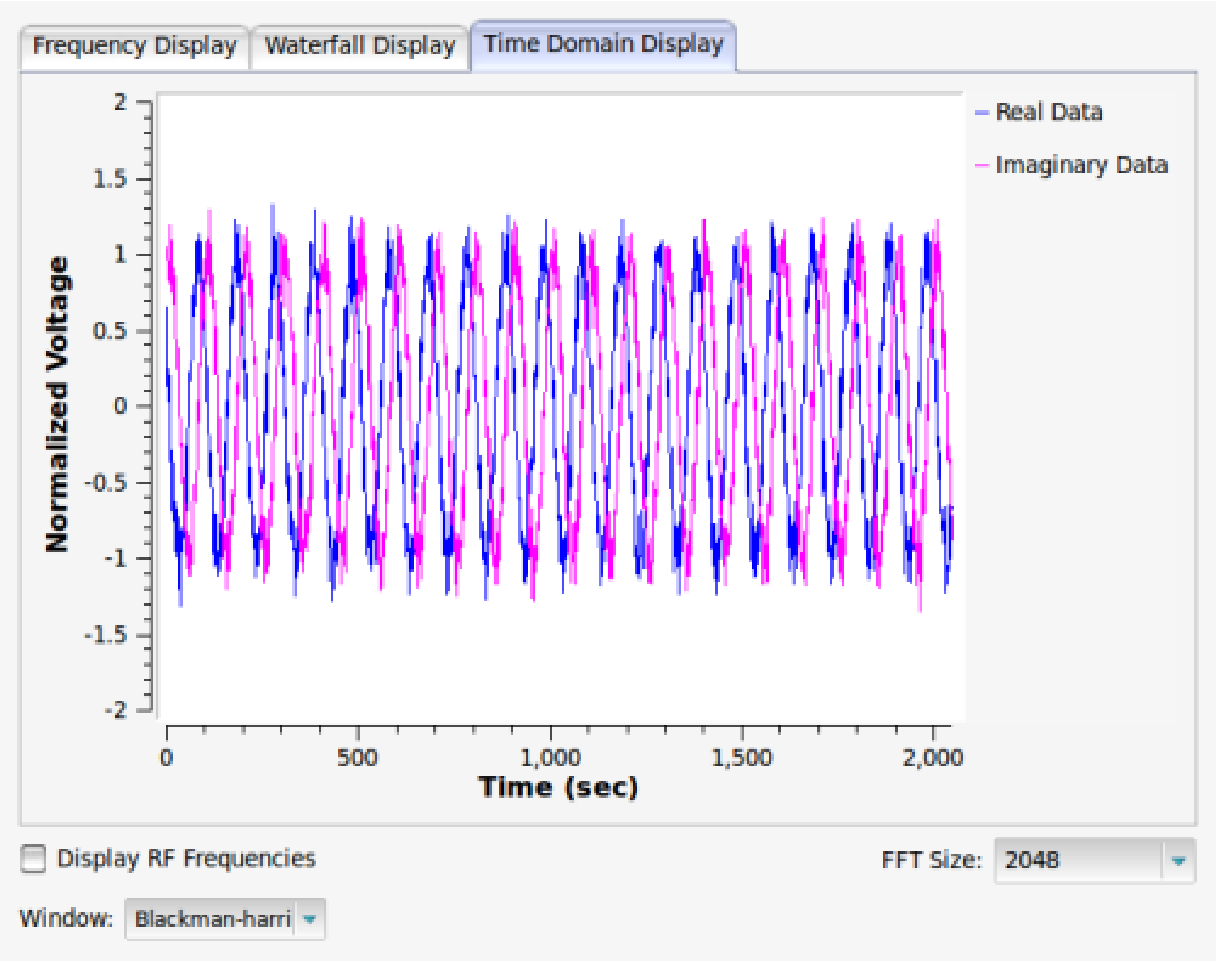
The QT GUI output offers multiple views:

Waterfall (or spectrogram)



The QT GUI output offers multiple views:

Time (with real and imaginary parts)



USING QT GUI WITH FM EXAMPLE

THINKING ABOUT SOFTWARE

SOFTWARE Radio

- More than just signal processing algorithms
- We worry about implementation as well
- OSS project has many objectives:
  - High quality, efficiency, and speed
  - Readable (and therefore editable)
  - Robust and reliable

Things we think about

Installation and operation on multiple OSes

Unit testing

Profiling and performance testing

Autotools: The worst build system aside from all the others…

- GNU’s Automake and Autconf
  - Well-understood build system in GNU community
- Test operating system support
- Ensure dependencies are met
- **make check** and **make distcheck** to test full build system

Unit Testing: make sure your code works and continues to work.

- For C++ code, we use the CppUnit test suite
- For blocks wrapped to Python, we use python.unittest
- Using Hudson Continuous Integration tool to monitor builds and tests
  - hudson-ci.org

Profiling Code

- First rule: “premature optimization is the root of all evil.”
- Code, test, get it right. Then optimize.
- Use profiling tools to find where your code needs work.
- Focus on measured performance problems and optimize.
- Things you think you know that just ain’t so…

Designing a FIR filter

i = 0

input

e	g	f	e	d	c	b	a

idx = 0

taps (len = 4)

c_0	c_1	c_2	c_3

buffer (2xlen)

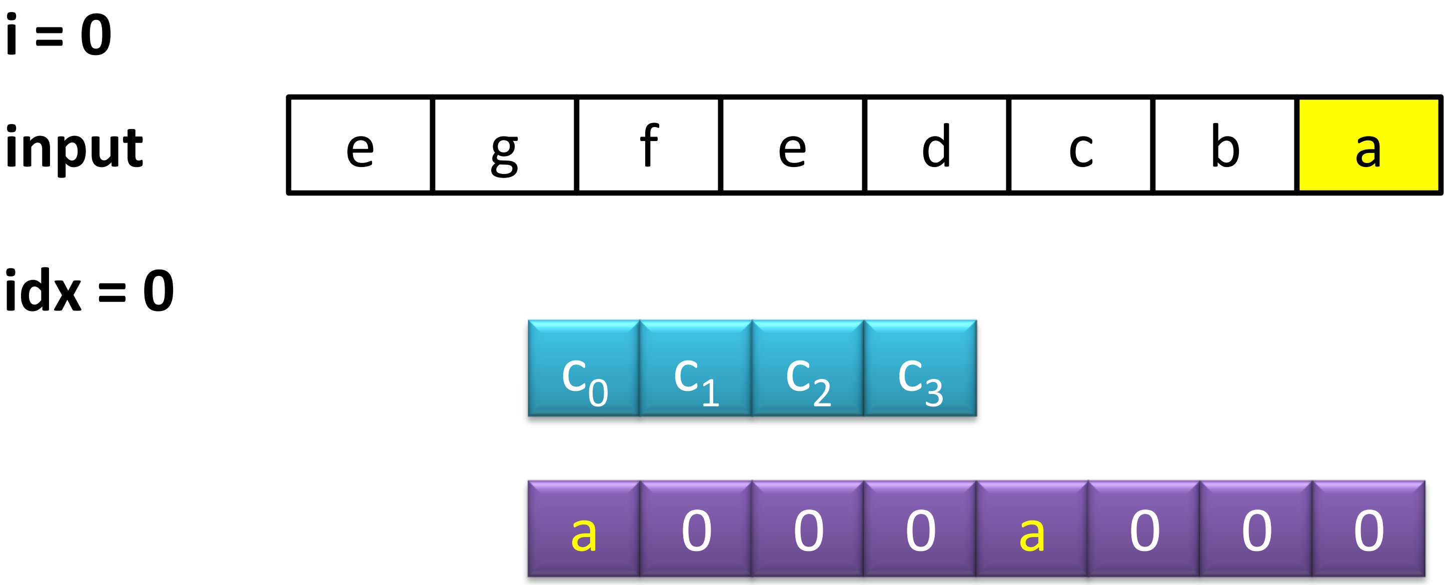
0	0	0	0	0	0	0	0

$$y_i = x_i c_0 + x_i c_1 + x_i c_2 + x_i c_3$$

Update:

1. Write next input to buffer at *idx*
2. Write same input to buffer at *idx+len*
3. increment *len* = *len* + 1
4. Perform filter calculation

Designing a FIR filter



Update:

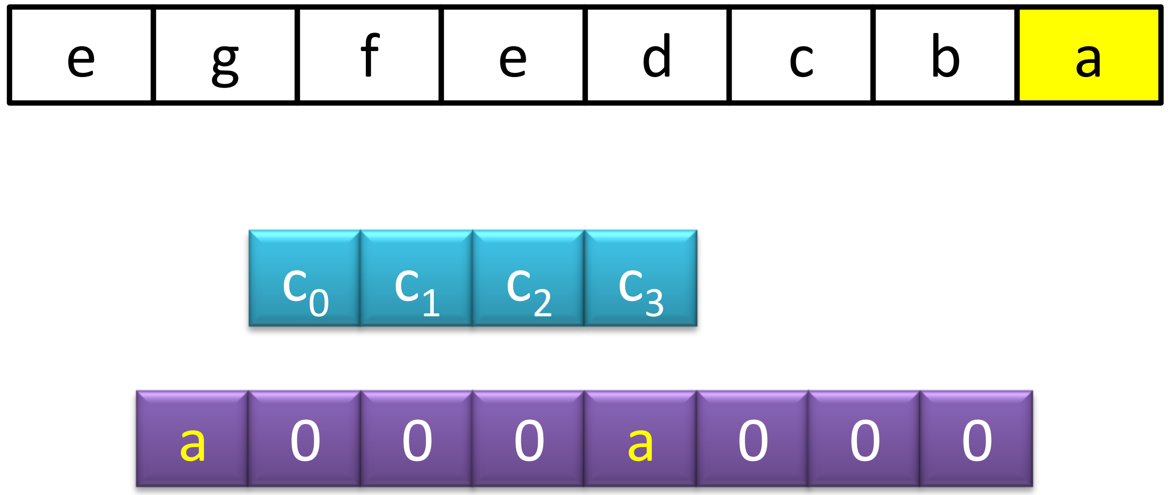
1. Write next input to buffer at *idx*
2. Write same input to buffer at *idx*+*len*
3. increment *len* = *len* + 1
4. Perform filter calculation

Designing a FIR filter

i = 0

input

idx = 1

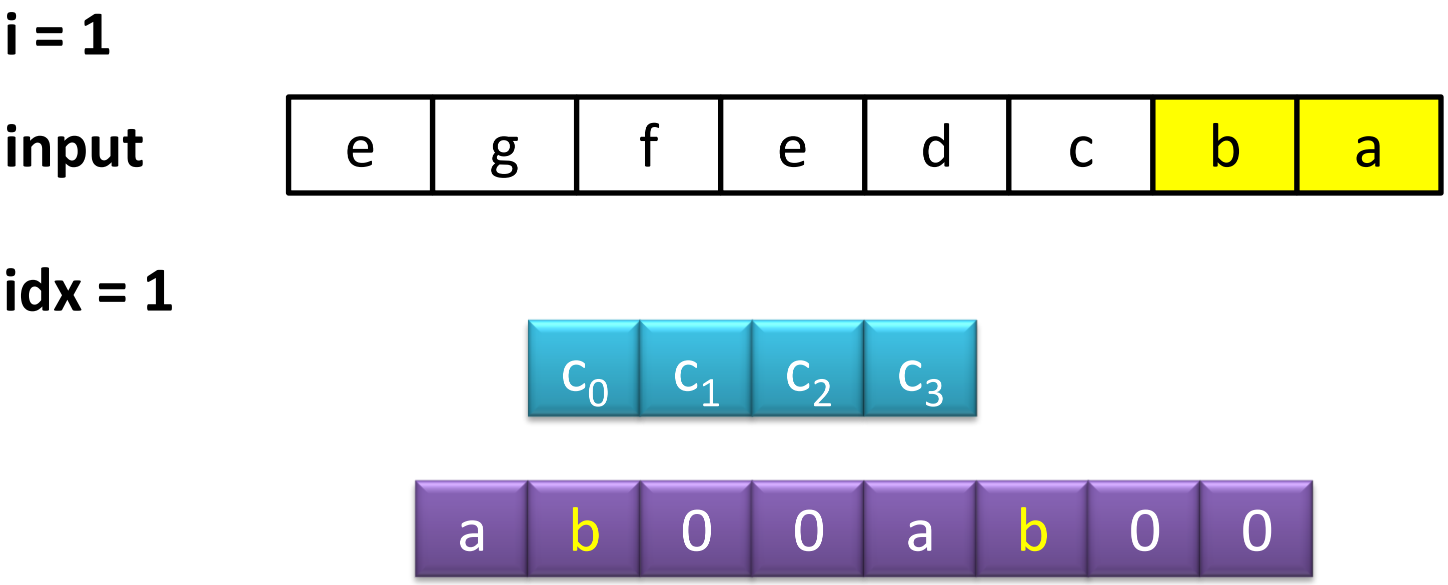


$y_i = 0 + 0 + 0 + ac_3$

Update:

1. Write next input to buffer at *idx*
2. Write same input to buffer at *idx*+*len*
3. increment *len* = *len* + 1
4. Perform filter calculation

Designing a FIR filter



Update:

1. Write next input to buffer at *idx*
2. Write same input to buffer at *idx+len*
3. increment *len* = *len* + 1
4. Perform filter calculation

Designing a FIR filter

i = 1

input

e	g	f	e	d	c	b	a

idx = 2

c_0	c_1	c_2	c_3

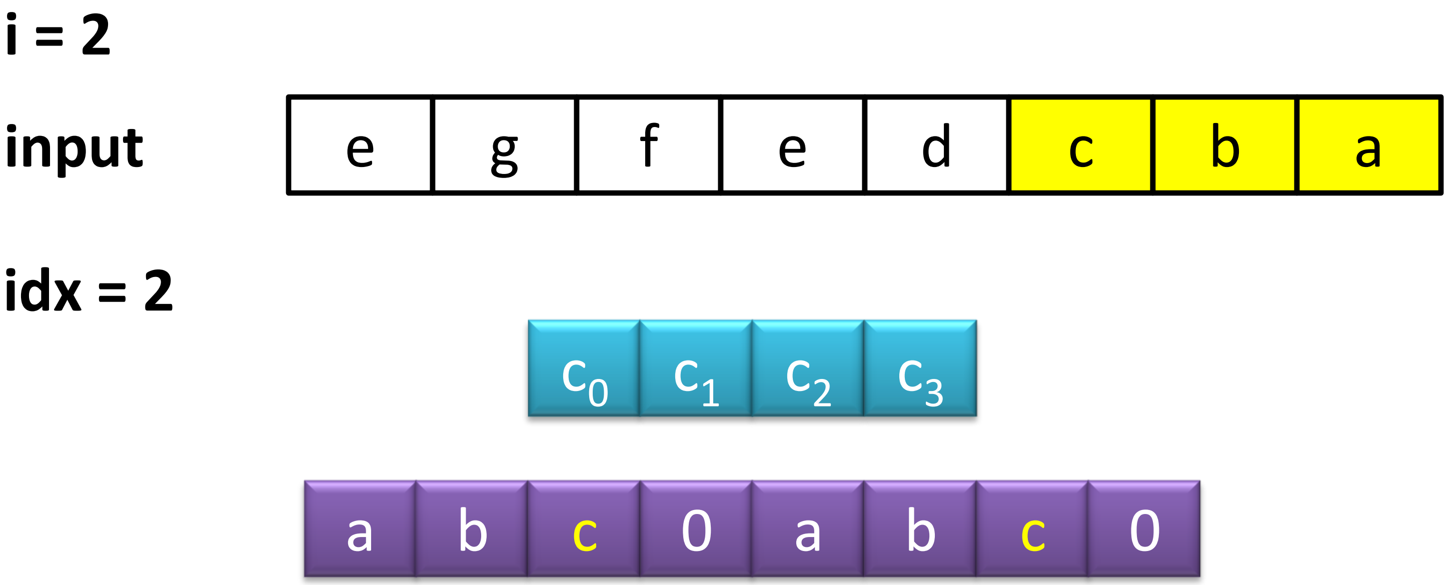
a	b	0	0	a	b	0	0

$$y_i = 0 + 0 + ac_2 + bc_3$$

Update:

1. Write next input to buffer at *idx*
2. Write same input to buffer at *idx*+*len*
3. increment *len* = *len* + 1
4. Perform filter calculation

Designing a FIR filter



Update:

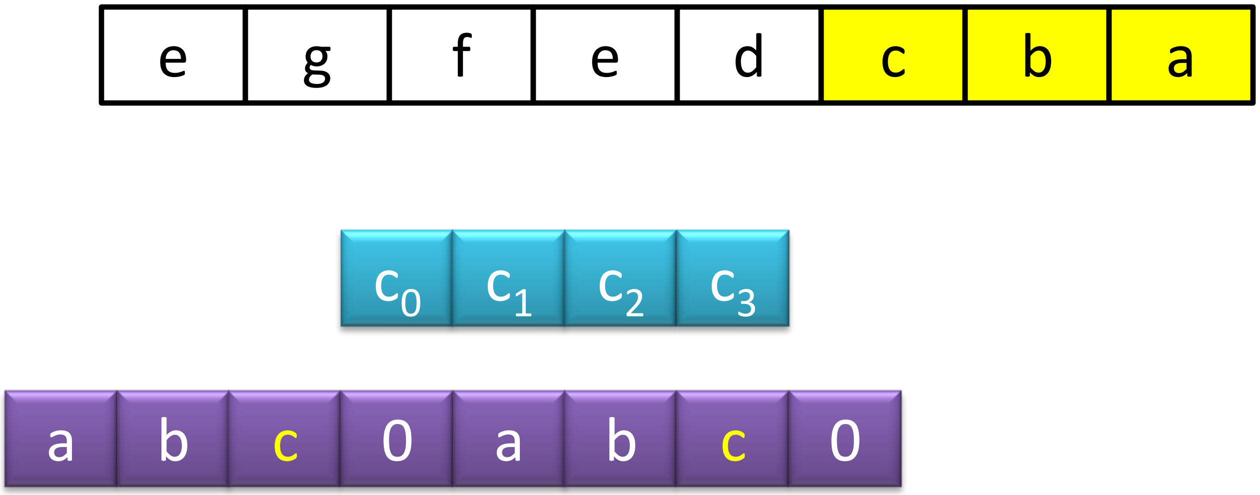
1. Write next input to buffer at *idx*
2. Write same input to buffer at *idx*+*len*
3. increment *len* = *len* + 1
4. Perform filter calculation

Designing a FIR filter

i = 2

input

idx = 3

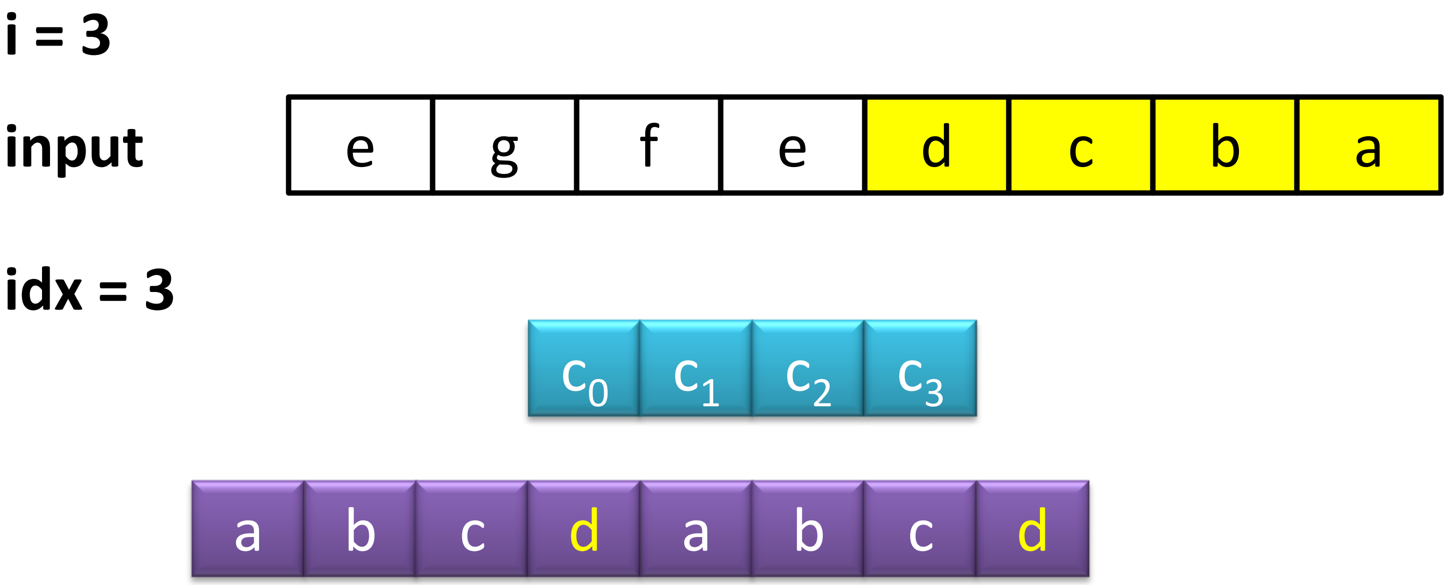


$y_i = 0 + ac_1 + bc_2 + cc_3$

Update:

1. Write next input to buffer at *idx*
2. Write same input to buffer at *idx+len*
3. increment *len* = *len* + 1
4. Perform filter calculation

Designing a FIR filter



Update:

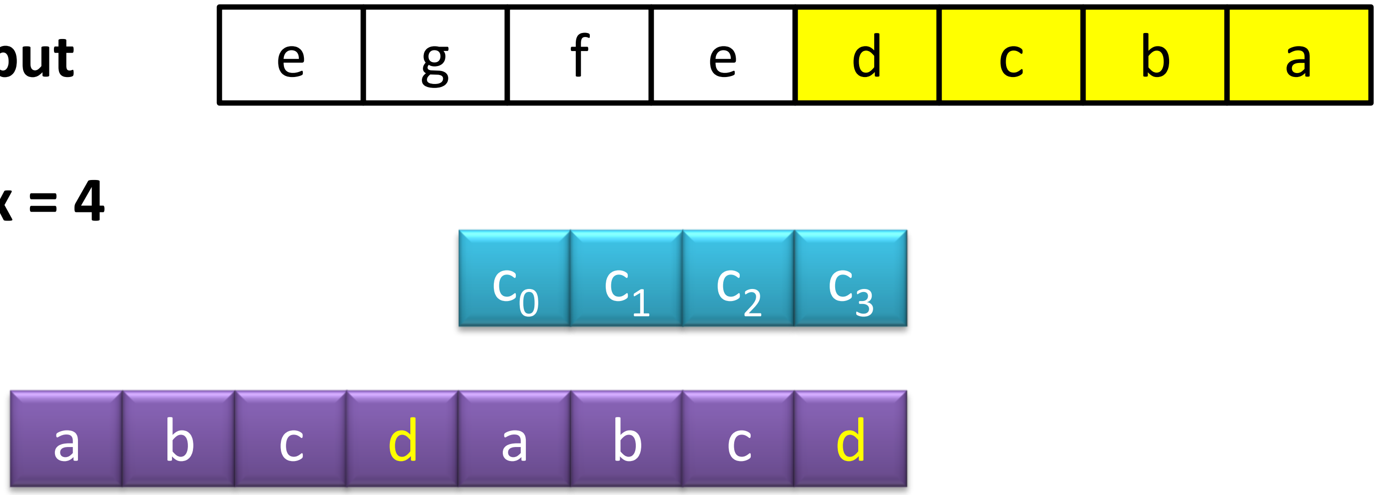
1. Write next input to buffer at *idx*
2. Write same input to buffer at *idx*+*len*
3. increment *len* = *len* + 1
4. Perform filter calculation

Designing a FIR filter

i = 3

input

idx = 4



$$y_i = ac_0 + bc_1 + cc_2 + dc_3$$

Update:

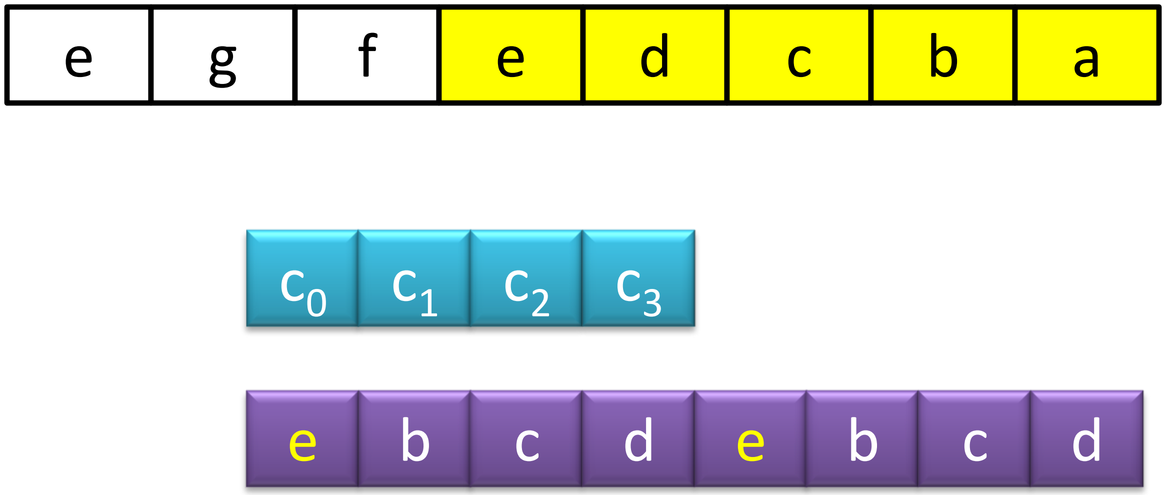
1. Write next input to buffer at *idx*
2. Write same input to buffer at *idx*+*len*
3. increment *len* = *len* + 1
4. Perform filter calculation

Designing a FIR filter

i = 4

input

idx = 0



Update:

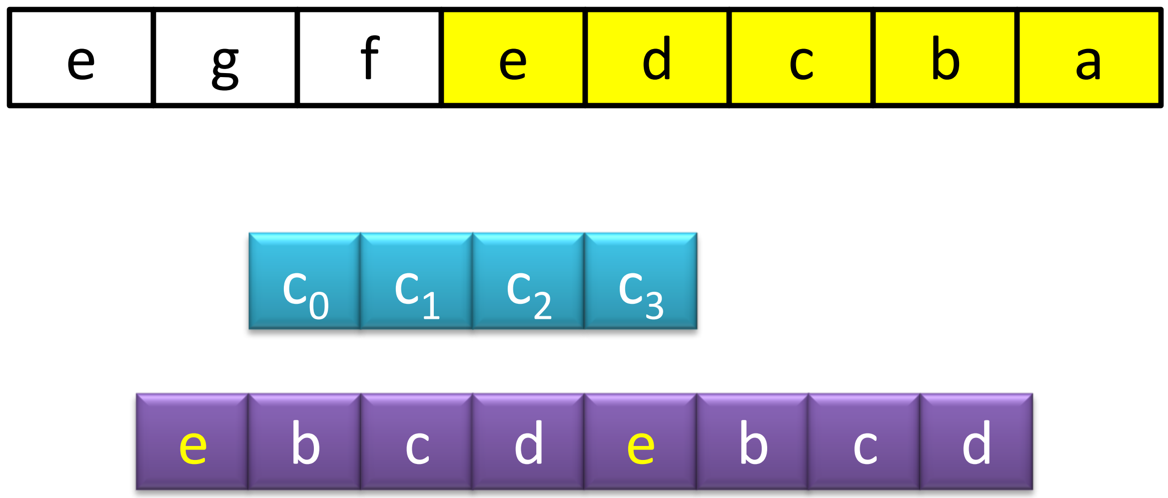
1. When idx == len, wrap around to 0

Designing a FIR filter

i = 4

input

idx = 1



$$y_i = bc_0 + cc_1 + dc_2 + ec_3$$

Update:

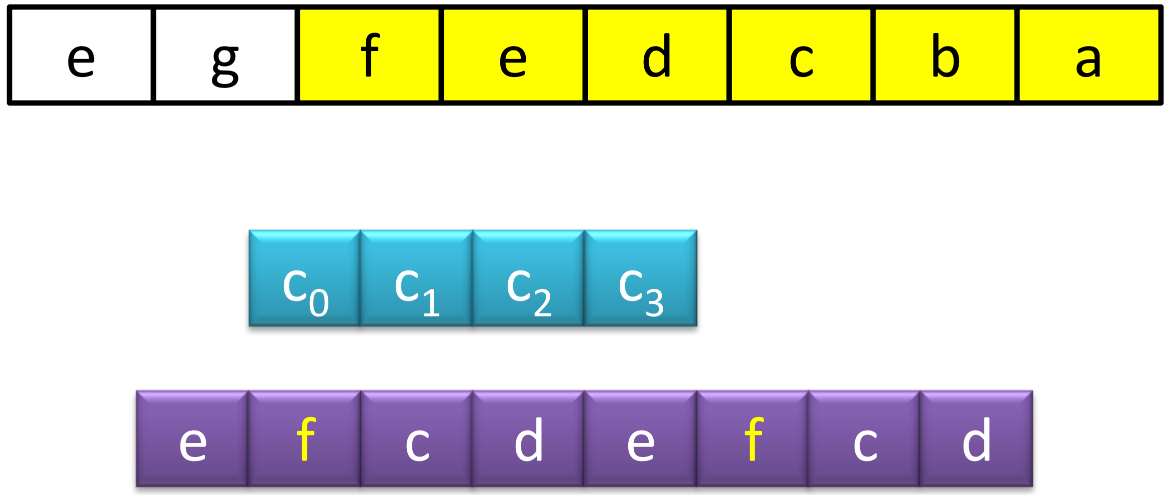
1. When idx == len, wrap around to 0

Designing a FIR filter

i = 5

input

idx = 1



Continue with this algorithm for all input items.

Designing a FIR filter

i = 5

input

e	g	f	e	d	c	b	a

idx = 2

c_0	c_1	c_2	c_3

e	f	c	d	e	f	c	d

$$y_i = cc_0 + dc_1 + ec_2 + fc_3$$

Continue with this algorithm for all input items.

Designing a FIR filter

- The only logic in this algorithm is to check when *idx* == *len* in order to reset *idx* = 0.
- How?

If statement

```
idx = idx + 1;
if(idx == len)
    idx = 0;
```

modulo len

```
idx = (idx+1) % len;
```

- Which is faster? Does it matter?

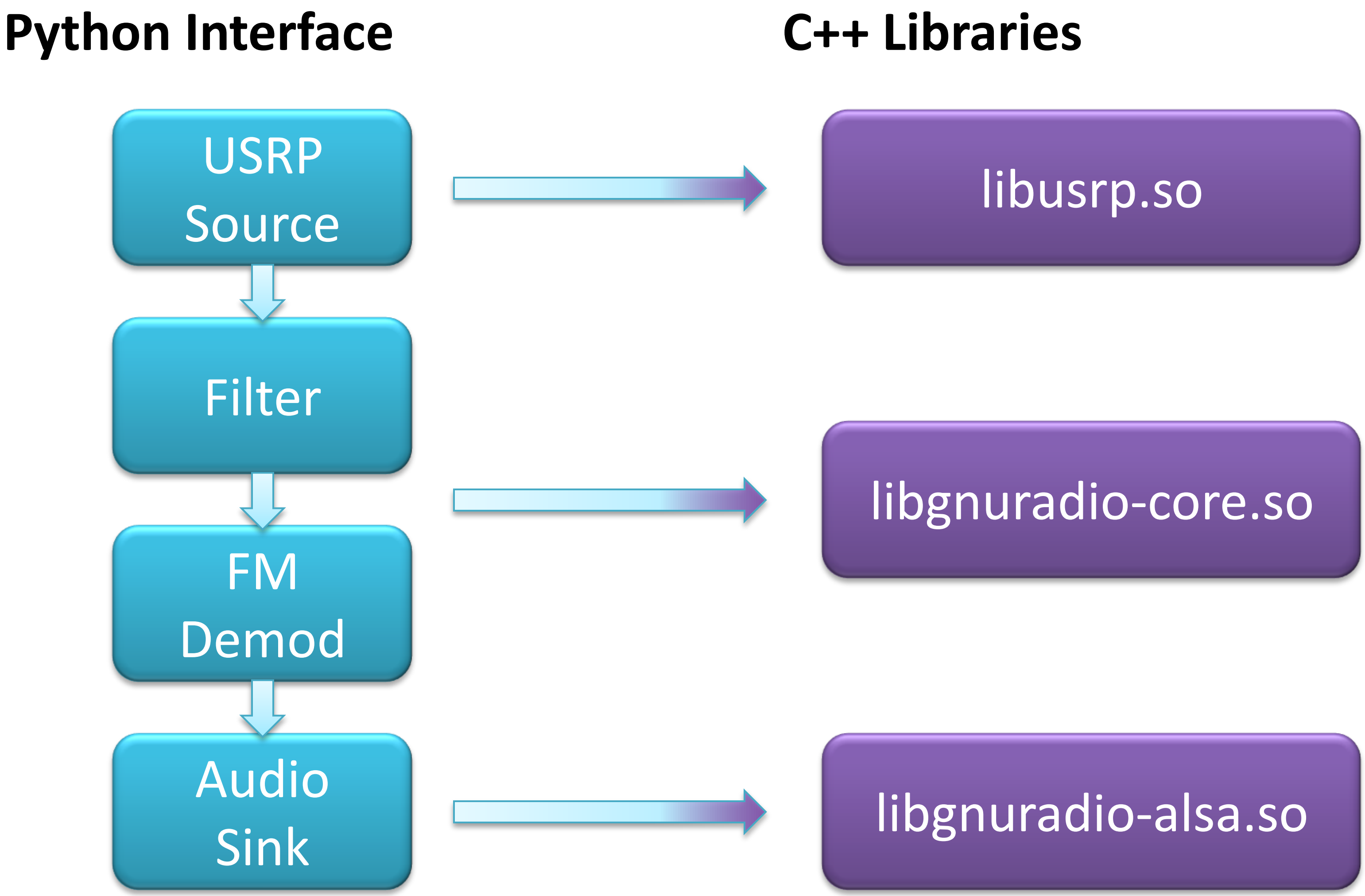
Profiling tools

- Walk through an example using:
  - Valgrind (http://valgrind.org)
  - Cachegrind (http://valgrind.org/info/tools.html)
  - KCachegrind (http://kcachegrind.sourceforge.net)

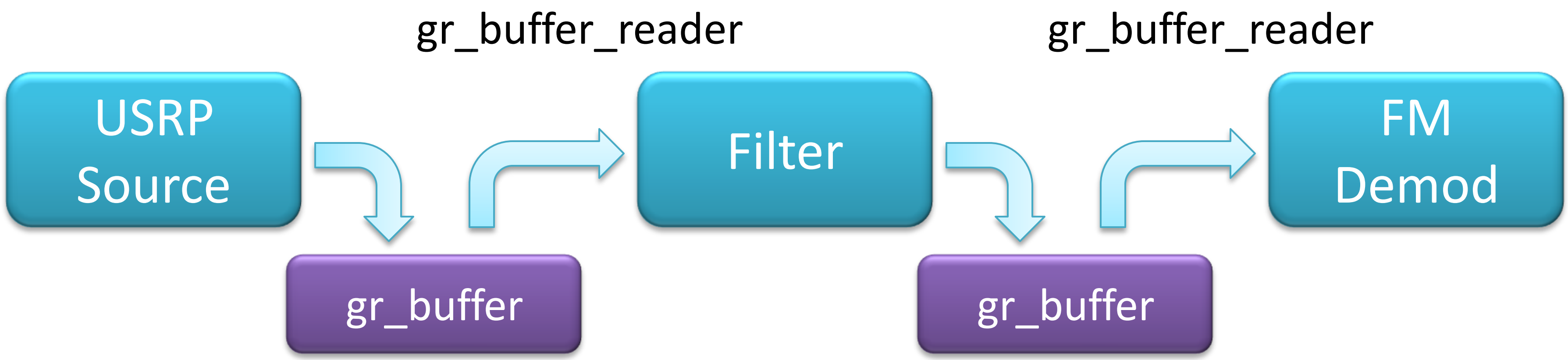
PROFILING EXAMPLE

PROGRAMMING MODEL

We started off with this concept:



Behind the scenes:



- Scheduler calls a block's **work** function and tells it how many items it can produce based on the number of items in the gr_buffer.

- Blocks read from their input buffer and write to an output buffer.

- Scheduler is optimized for throughput.

GNU Radio block work function

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

- There are *N* input streams
  - input_items[*n*] has ninput_items[*n*] items
- Can produce at most *noutput_items* number of items in any *output_items* output stream
- Tells scheduler
  - how many consumed from each input
  - how many produced (<= noutput_items)

Example:
multiply_const_ff

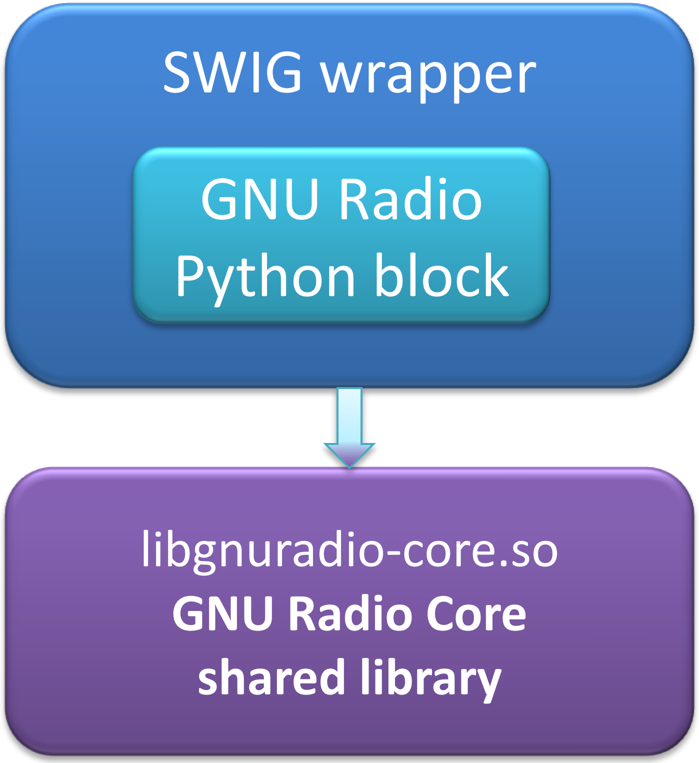

```
int general_work ( <see last slide> )
{
    const float *in = (const float*)input_items[0];
    float *out = (float*)ouput_items[0];

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

    // an equal number of items consumed and produced
    consume_each(noutput_items);
    return noutput_items;
}
```

Four basic types of blocks

- **Sync blocks**
  - number of items in equals the number of items out
  - like the multiply constant example
- **Decimation blocks**
  - number of items IN is *D* times the number of items OUT
- **Interpolation blocks**
  - number of items OUT is *I* times the number of items IN
- **Blocks**:
  - relationship between input and output items is not straight-forward

SWIG allows us to talk between the Python and C++ layers.

- Simple Wrapper Interface Generator (SWIG)
  - http://www.swig.org



Program GNU Radio in Python; computation handled in C++

- SWIG produces Python modules out of the C++ blocks
- Builds an interface based on an interface description file (.i)
  - The interface description file describes the API for talking between the two languages
  - Its content is very similar to the C++ .h header file

Advice:

If you want to write a new block, find a block that has similar properties and copy it.

CONSIDERING ALGORITHMS

Understanding GNU Radio's quantization

- What is the proper scope of a block?
- Try to use good software principles:
  - Increase usability
  - Reduce duplication
- Find the smallest level the algorithm can run
- Expand the scope only as needed
  - Only when the combination of other blocks cannot properly solve the problem

Programming the algorithm

- Follow good programming practices that we discussed earlier
- Make as much gain from the algorithm as possible
  - don’t just rely on super programming skills to overcome an inherently bad algorithm
- Takes a lot of multidisciplinary thinking

Example:

The FIR filter

- We know that filtering is convolution in time:

$$y[n] = t \otimes x[n]$$

- Which means, its multiplication in frequency:

$$Y[n] = \sum_{i=0}^{L-1} T[i] X[n-i]$$

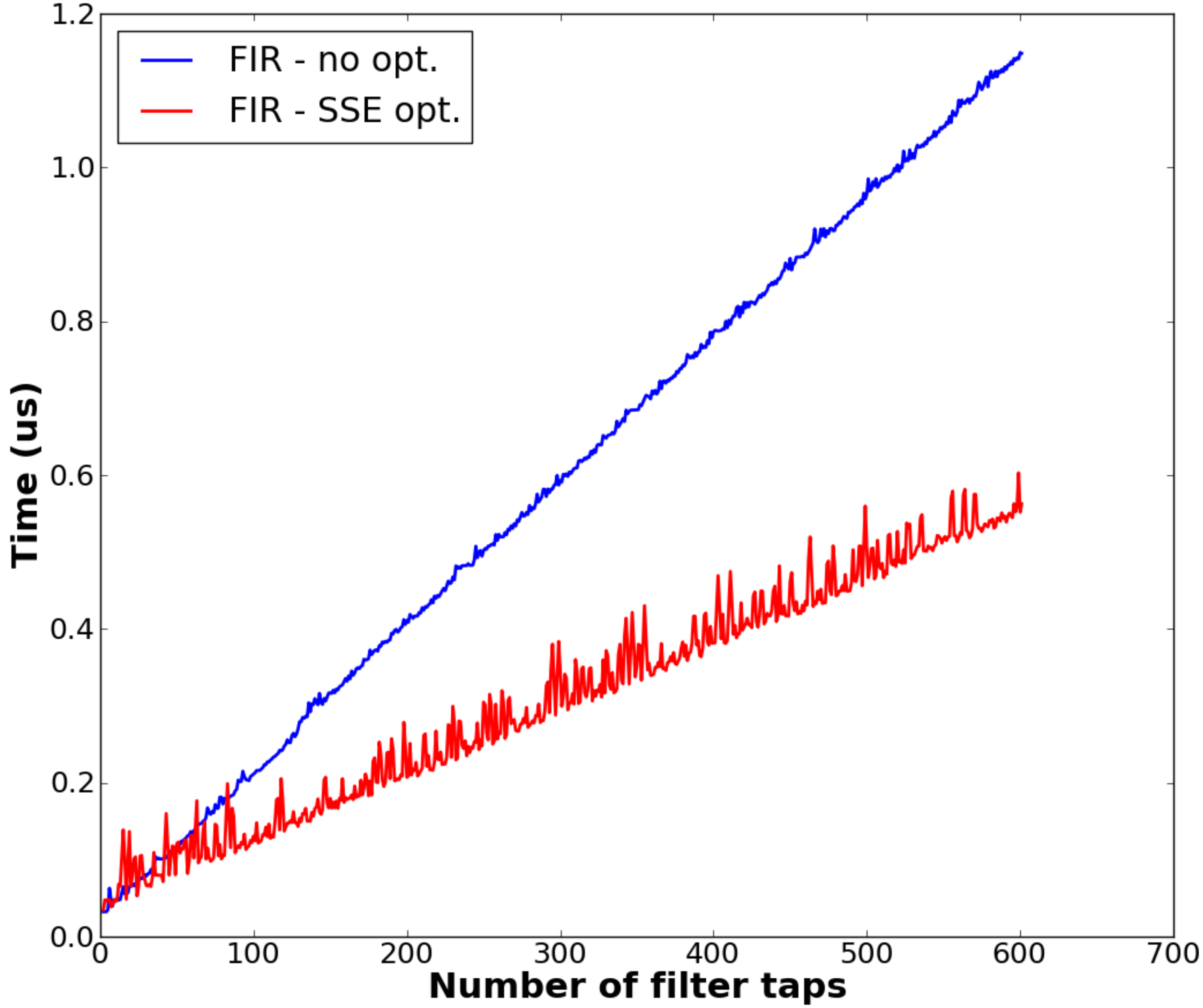
- With the efficiency of the FFT, convolution is faster in the frequency domain
  - "fast convolution"

GNU Radio implements both kinds of FIR filters

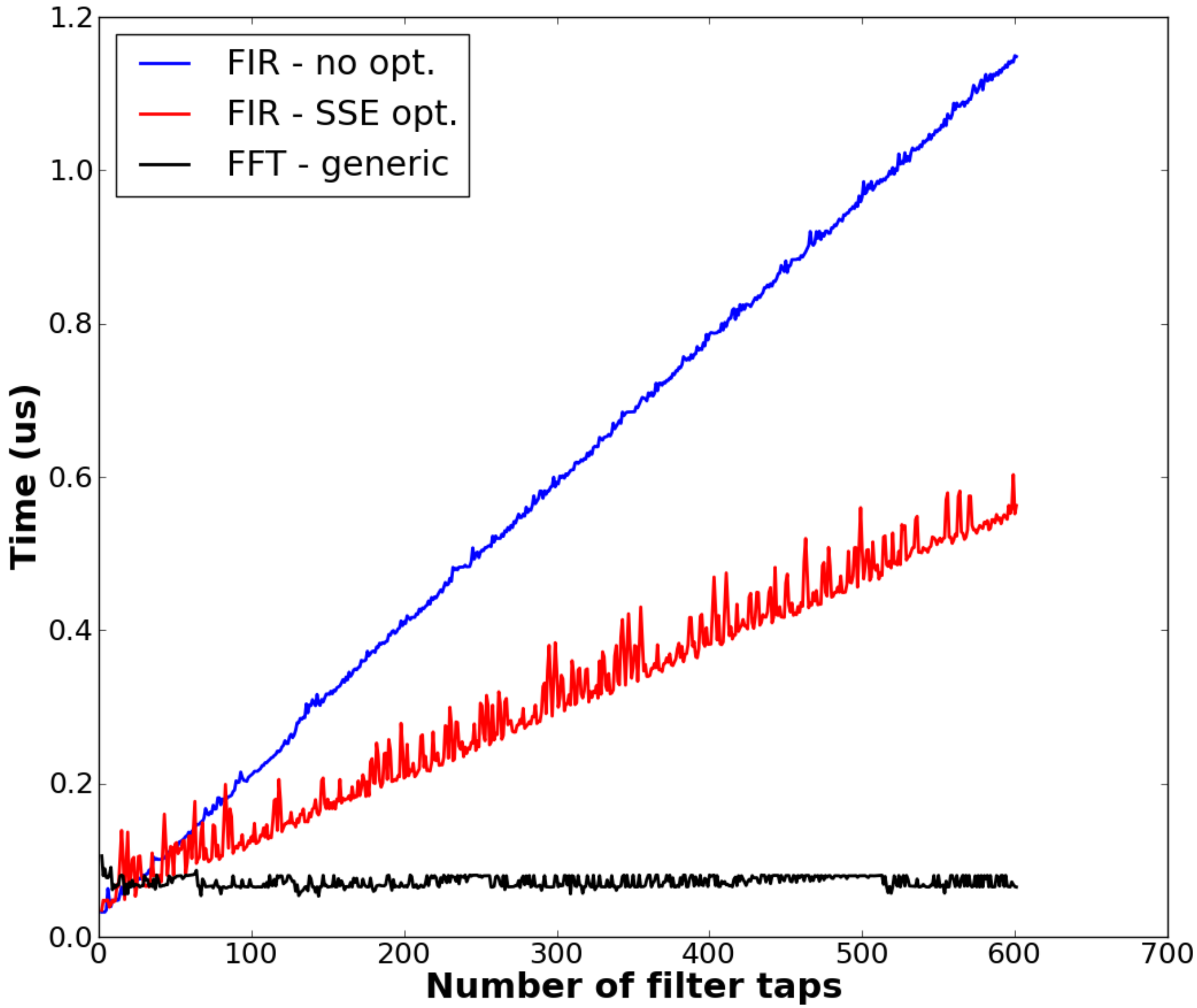
- FIR done as time convolution
  - gr_fir_filter_XXX
- FIR done in frequency domain
  - gr_fft_filter_XXX

- The time domain has been SIMD optimized
- How do they compare in speed?

Comparing the SIMD and non-SIMD time domain filters



Comparing the time domain to frequency domain filters

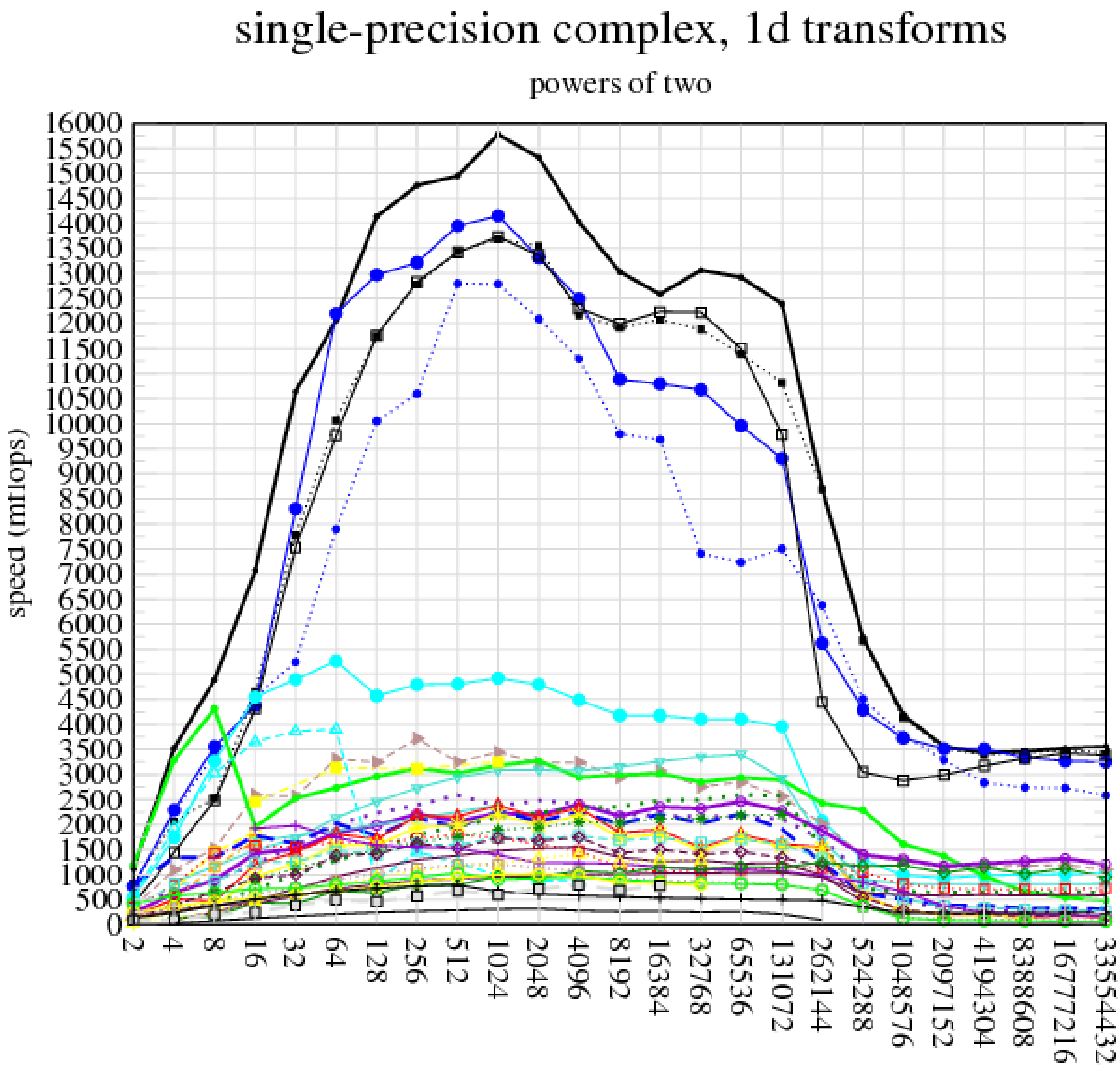


For all of our cleverness in the time domain

- Using the right algorithm produces a more efficient filter.
- FFT filter slower for small number of taps
  - around 22
- Not much slower at this point
- Some gains left to be made
  - SIMD optimize the multiplication loop
  - Some FFT sizes are faster than others; use them and pad with zeros

FFTW capabilities

(http://www.fftw.org/speed/CoreDuo-3.0GHz-icc64/)



- Intell IPPS
- Intel MKL in-place
- Intel MKL out-of-place
- FFTW3 out-of-place
- FFTW3 in-place

Other lines are from other FFT programs and are not important for this comparison

THE GNU RADIO COMPANION

Graphical tool for building GNU Radio flowgraphs

- Makes it easier to:
  - Visualize the data flow
  - Tie in with graphical sinks
  - Browse available library of blocks
  - Add live interactive capabilities through block callbacks
- **gnuradio-companion** is distributed with GNU Radio

GNU Radio Companion features:

- Variables
  - Set values of blocks
  - Dynamic variables add features such as sliders or edit boxes for on-line altering of parameters
- Python programming level:
  - many things can be altered by using Python programming such as calling other modules, functions, or creating lambda functions
  - Can even import new modules
- GUI interface is interactive and configurable
  - Add Notebooks for better on-screen organization

EXAMPLES OF USING THE GNU RADIO COMPANION

FIN