



# Using GNU Radio for Analog Communications

Hackspace Brussels - January 31, 2019



# Derek Kozel

- Radio Amateur since second year of university
  - UK Advanced license MW0LNA, US Extra K0ZEL
- Moved from the San Francisco Bay Area to Cardiff, Wales in April 2017
- Bachelors and Masters in ECE & Public Policy at Carnegie Mellon University
- Worked at Range Networks, SpaceX, Ettus Research (NI)
- Currently a PhD at Cardiff University in the Centre for High Frequency Engineering
- GNU Radio Project Officer

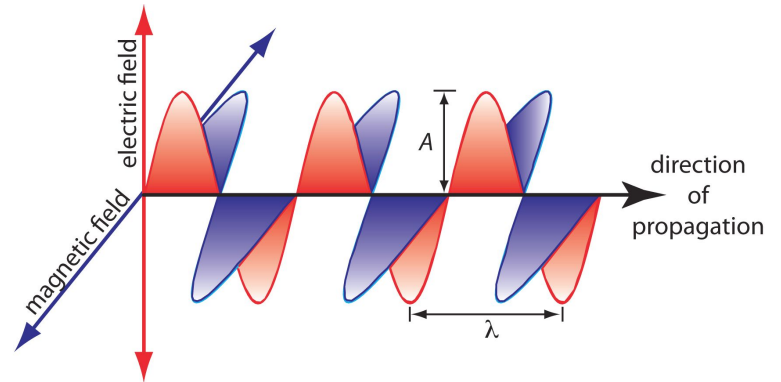




# Intro to Radio

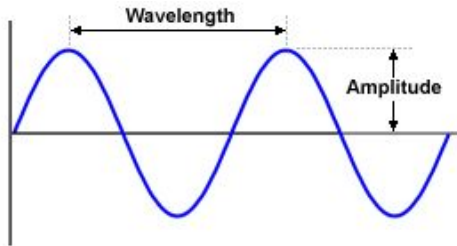
# Electromagnetic Waves

- Electric and Magnetic energy
- Can bounce, bend, and generally confuse



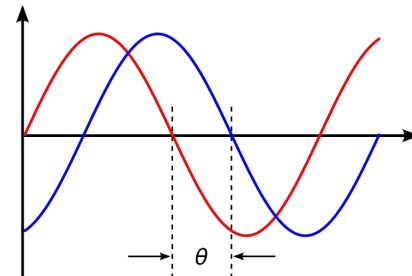
# Properties of a wave

- Frequency
  - Number of cycles per second (Hertz)
- Wavelength
  - Distance between start and end of a cycle



<https://en.wikipedia.org/wiki/Wavelength>

- Amplitude
  - The magnitude or strength of the wave
- Phase
  - The offset of the wave with respect to another wave



[https://en.wikipedia.org/wiki/File:Sine\\_voltage.svg](https://en.wikipedia.org/wiki/File:Sine_voltage.svg)

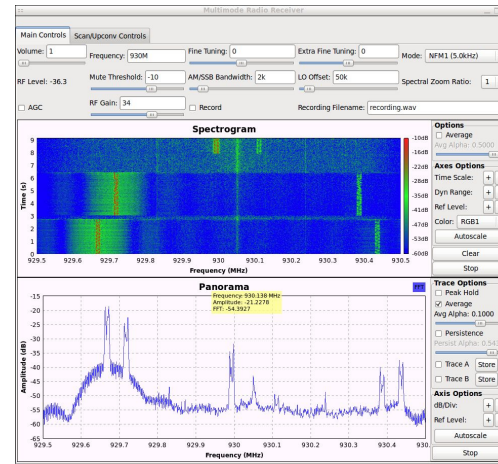
[https://en.wikipedia.org/wiki/Phase\\_\(waves\)](https://en.wikipedia.org/wiki/Phase_(waves))



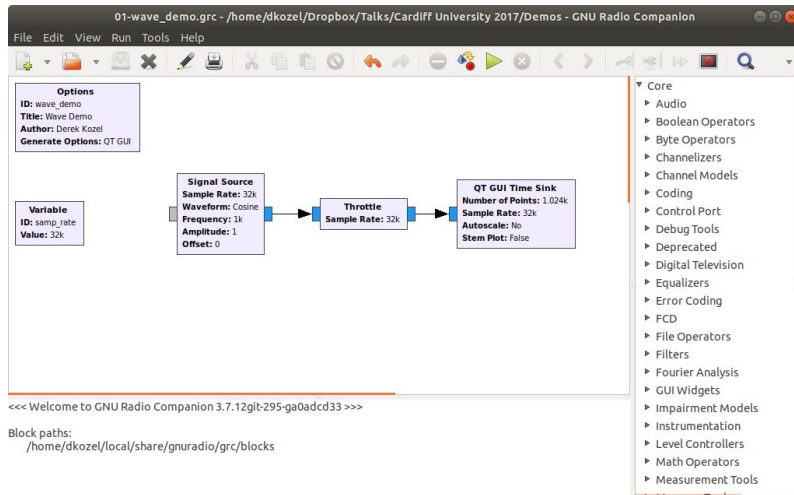
# Intro to GNU Radio



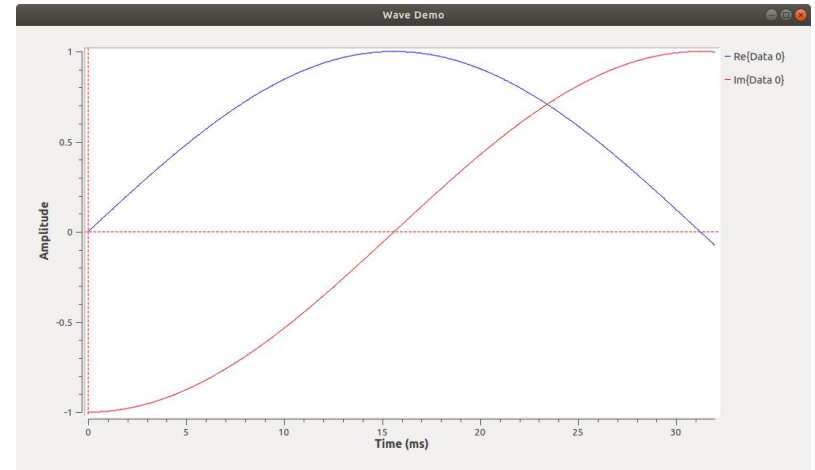
- A framework and set of libraries to build and run digital signal processing applications, primarily software defined radio ones
- Started in 2001
- Libre and open source
- Written in C++ and Python primarily
- Available on Linux, Windows, and Mac
- Used by a very wide variety of users
  - commerical, hobbyist, government



# GNU Radio Companion



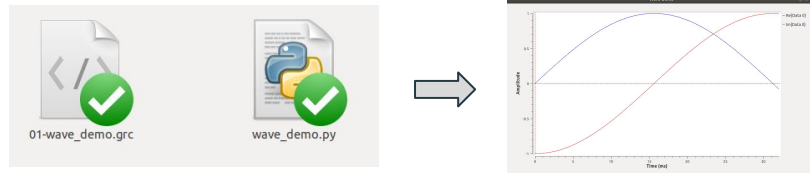
The screenshot shows the GNU Radio Companion (GRC) interface. The main window title is "01-wave\_demo.grc - /home/dkozel/Dropbox/Talks/Cardiff University 2017/Demos - GNU Radio Companion". The interface includes a menu bar (File, Edit, View, Run, Tools, Help), a toolbar with various icons, and a central block diagram area. On the left, there are "Options" and "Variable" panels. The block diagram consists of three blocks: "Signal Source" (Sample Rate: 32k, Waveform: Cosine, Frequency: 1k, Amplitude: 1, Offset: 0), "Throttle" (Sample Rate: 32k), and "QT GUI Time Sink" (Number of Points: 1.024k, Sample Rate: 32k, Autoscale: No, Stem Plot: False). On the right, a vertical list of component categories is visible, including Core, Audio, Boolean Operators, Byte Operators, Channelizers, Channel Models, Coding, Control Port, Debug Tools, Deprecated, Digital Television, Equalizers, Error Coding, FCD, File Operators, Filters, Fourier Analysis, GUI Widgets, Impairment Models, Instrumentation, Level Controllers, Math Operators, and Measurement Tools. At the bottom left, there is a welcome message: "<< Welcome to GNU Radio Companion 3.7.12git-295-ga0adcd33 >>>" and block paths: "Block paths: /home/dkozel/local/share/gnuradio/grc/blocks".





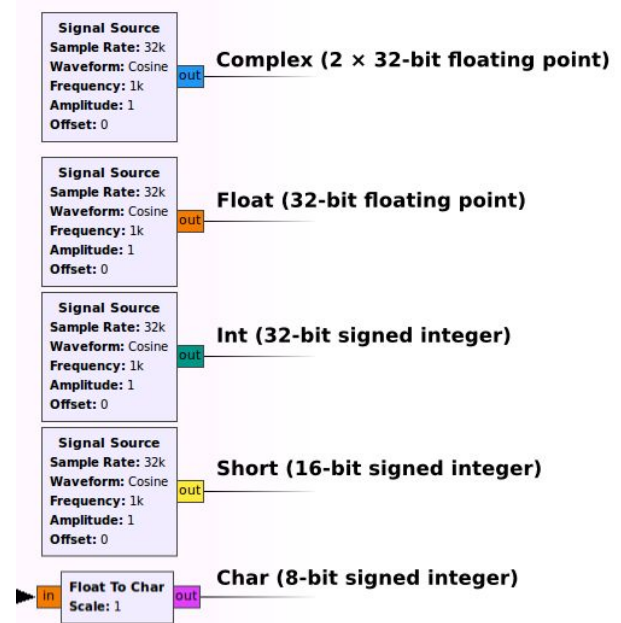
# Automatic Code Generation

- The graphical UI is generating Python code
  - Or C++ in the latest version
- We'll look quickly under the hood later



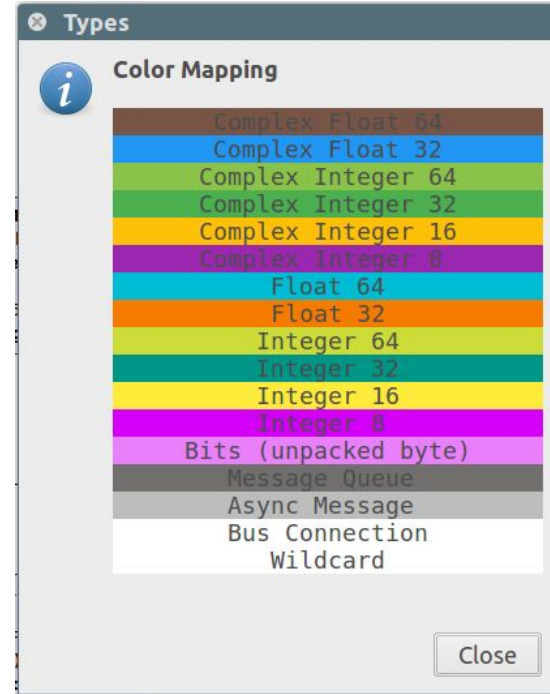
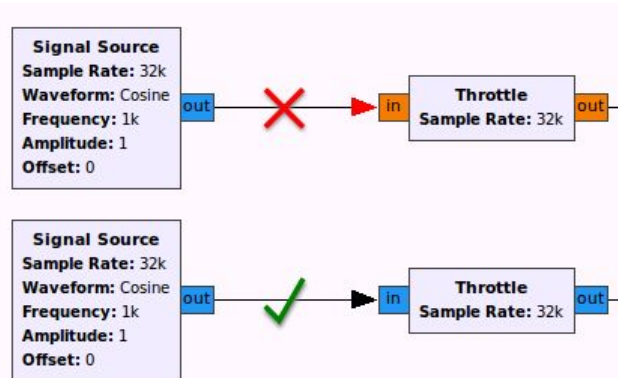
# Data Types

- Samples and data comes in different digital formats
- Semantic differences
  - Complex vs Real samples
  - Number vs Letters
- Size differences
  - 8 bits vs 32 bits



# Data Types

- Have to connect matching types
- GRC will warn you if there's a mismatch
- In the end, bits are bits
  - Computer will interpret them as you tell it to

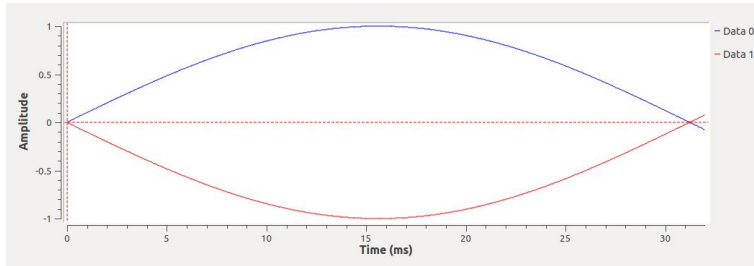




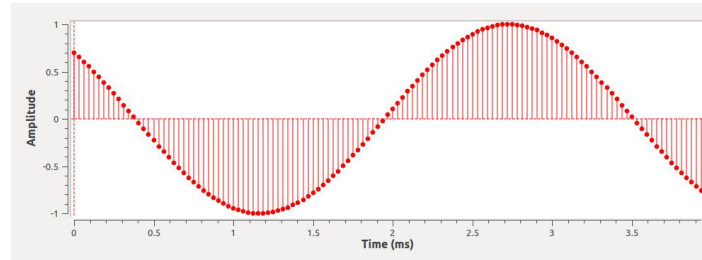
# Digital Signal Processing

# Time Domain

- Amplitude values over time
- Signals are continuous in the air or wire
- Signals are digitized by sampling the current value many times



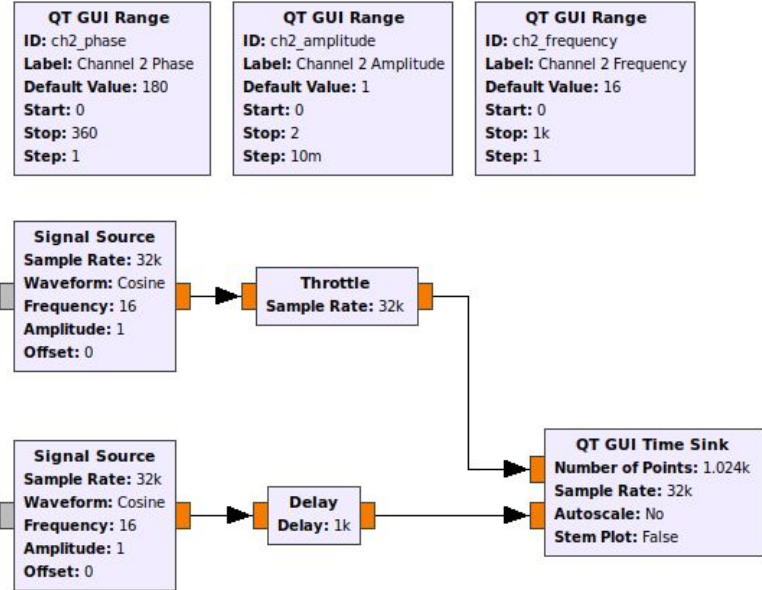
Continuous Signal (it's a lie!)



Discrete (sampled) Signal

# Exploring Waves

- Setup a flowgraph with controls for
  - Phase
  - Frequency
  - Amplitude



Delay value:

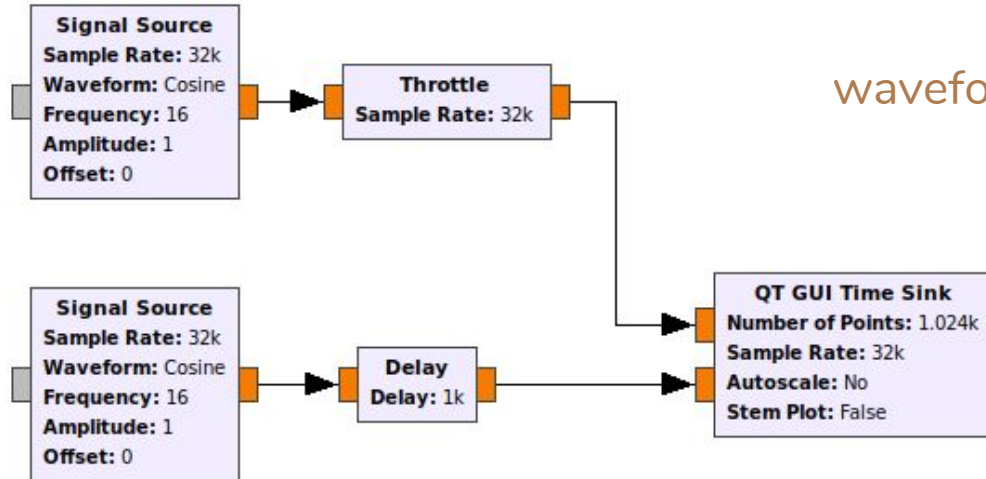
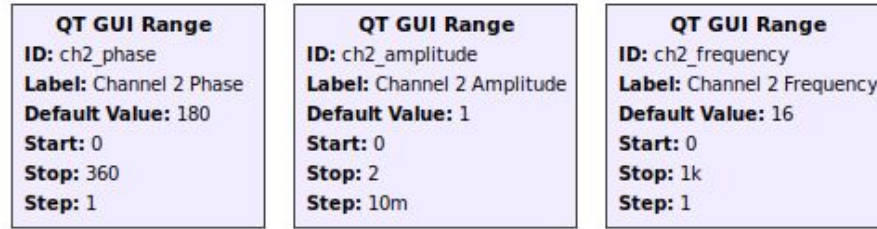
```
int((samp_rate/ch2_frequency) * (ch2_phase/360.0))
```

# Throttle Block

- GNU Radio will process data as fast as possible
- Hardware (Analog to Digital or Digital to Analog converters) will have a set sample rate
- Simulation only doesn't
- Add one (and only one!) throttle block to the flowgraph
  - Has a timer inside that tries to match the average throughput to the sample rate

Delay value:

```
int((samp_rate/ch2_frequency) * (ch2_phase/360.0))
```

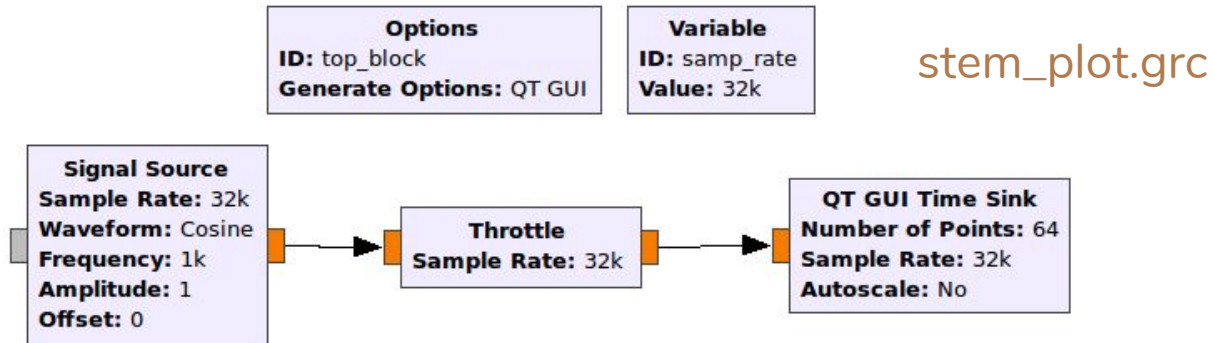
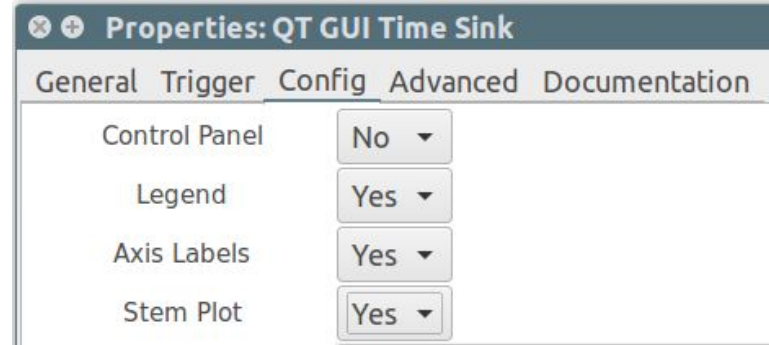


waveform\_demo.grc



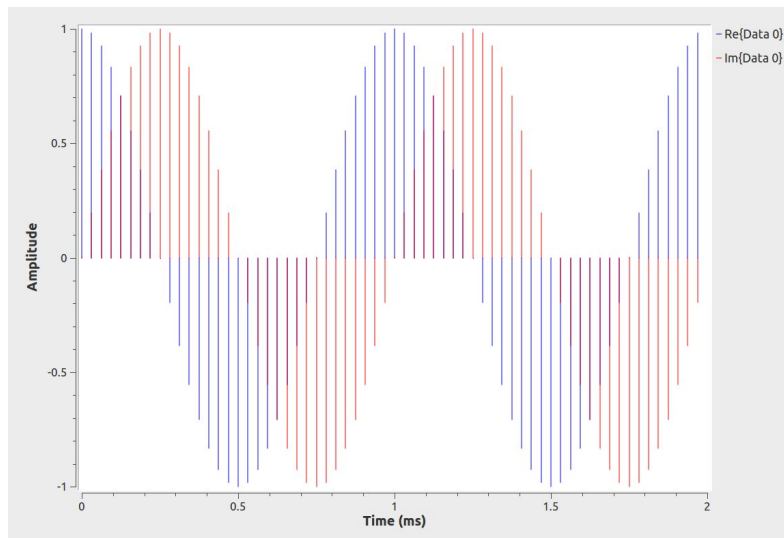
# Discrete Sampling

- Usually data is displayed as if it were continuous
  - Easier to visualize
  - Mostly accurate as long as you follow Nyquist's Sampling Law
- Can also display actual data points
  - Select Stem Plot under the Config menu in QT GUI Time Sink



# Complex Sampling

- Hard to make fast ADC/DACs
- Also ambiguities in frequency are real(ly painful)
- When mixing a signal with a sinewave crossing zero you lose all the information!
- Solution: Split the signal in two, mix with a sine and cosine, sample each result at the same time
  - Twice the information, all of it useful
  - Not cheating Nyquist
  - Bandwidth = Sample Rate

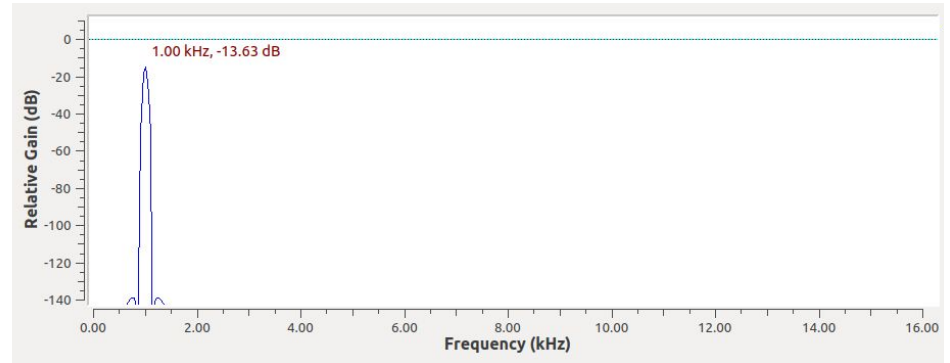
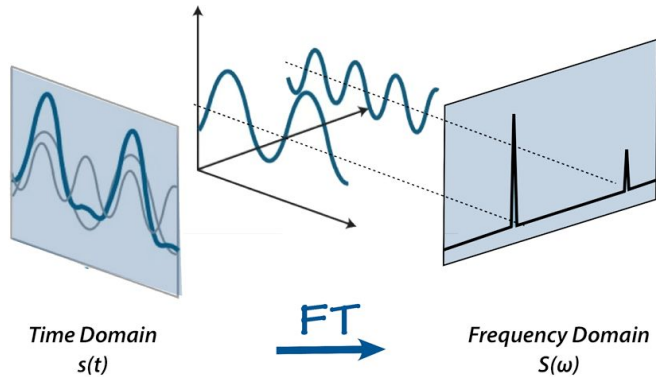


stem\_plot.grc



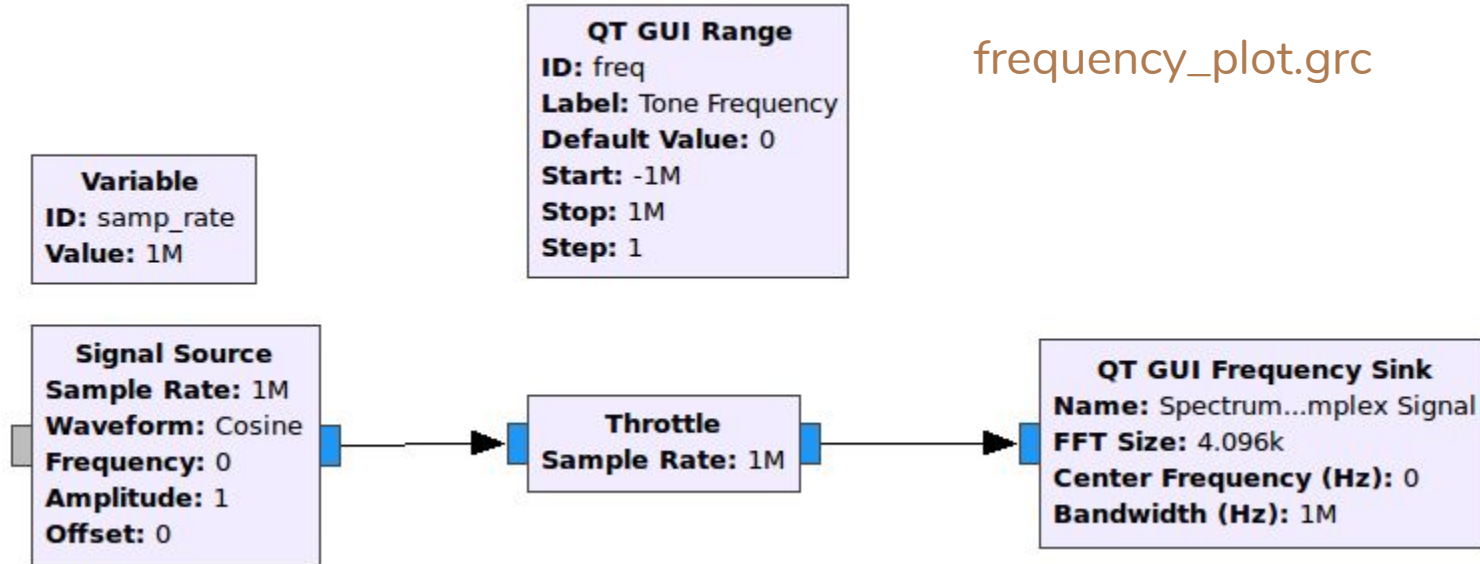
# Frequency Domain

- Time domain signals contain energy at certain frequencies
- They can be decomposed into the sum of many sine waves with different amplitudes



# Frequency Sink

frequency\_plot.grc



# Simple Receiver

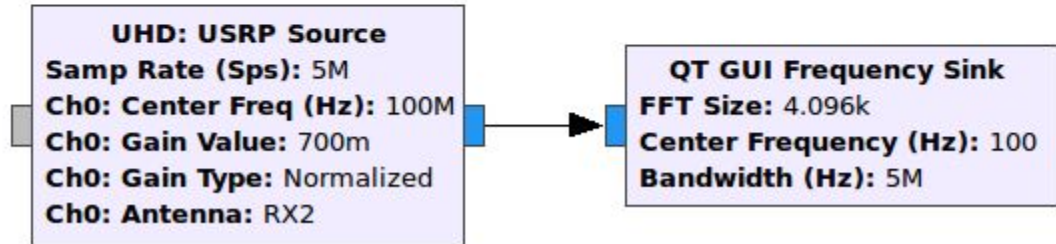
**Options**  
**ID:** top\_block  
**Generate Options:** QT GUI

**Variable**  
**ID:** samp\_rate  
**Value:** 5M

**QT GUI Range**  
**ID:** rx\_freq  
**Label:** Receive ...uency (MHz)  
**Default Value:** 100  
**Start:** 70  
**Stop:** 3k  
**Step:** 1

**QT GUI Range**  
**ID:** rx\_gain  
**Label:** Receive Analog Gain  
**Default Value:** 700m  
**Start:** 0  
**Stop:** 1  
**Step:** 100m

basic\_rx.grc

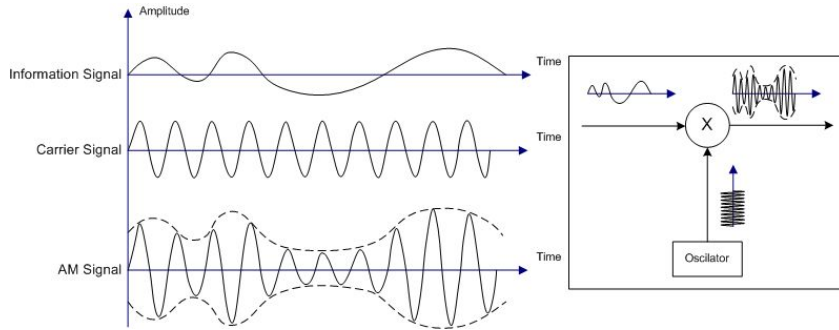


# Carrying Information

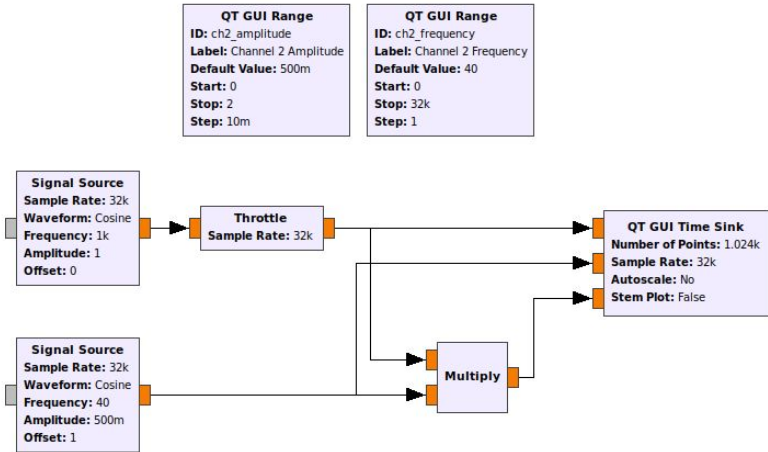
- Frequency, Amplitude, and Phase can all be changed over time
  - This change change of the signal is Modulation

# Amplitude Modulation

- Changing the amplitude of a “carrier” wave at a fixed frequency



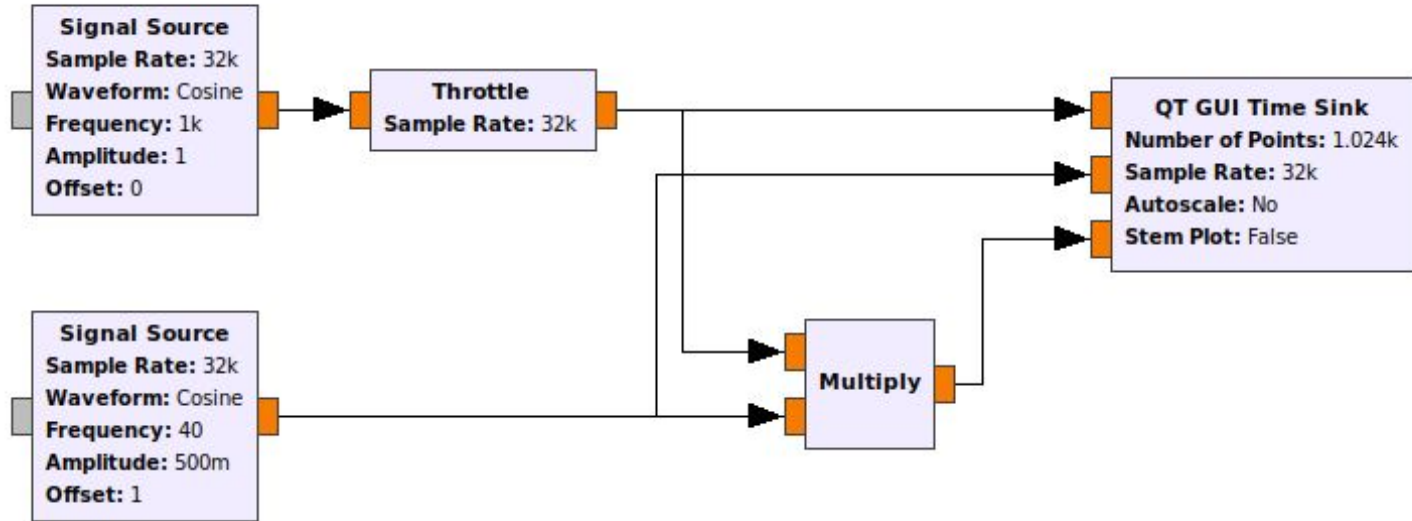
[https://en.wikipedia.org/wiki/Amplitude\\_modulation](https://en.wikipedia.org/wiki/Amplitude_modulation)



**QT GUI Range**  
ID: ch2\_amplitude  
Label: Channel 2 Amplitude  
Default Value: 500m  
Start: 0  
Stop: 2  
Step: 10m

**QT GUI Range**  
ID: ch2\_frequency  
Label: Channel 2 Frequency  
Default Value: 40  
Start: 0  
Stop: 32k  
Step: 1

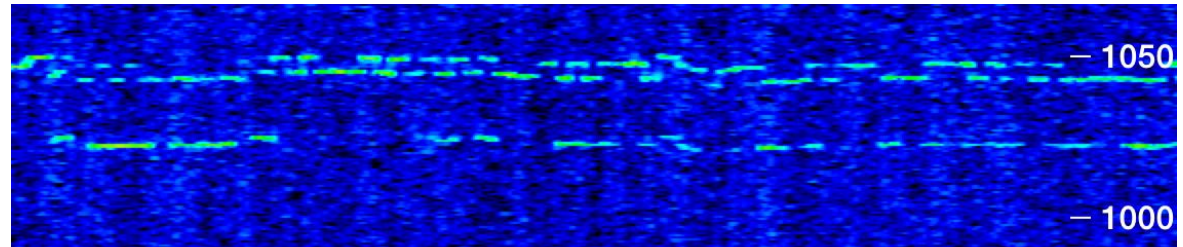
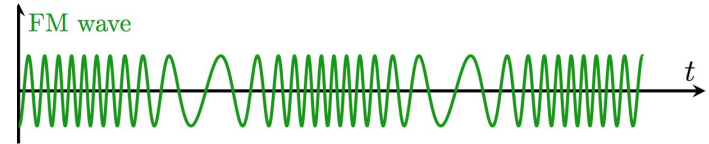
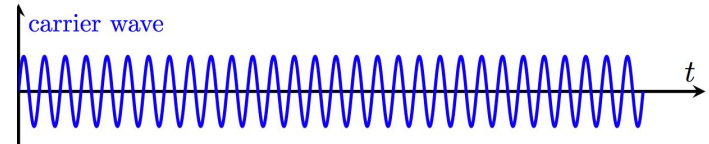
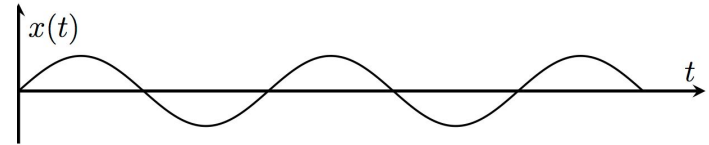
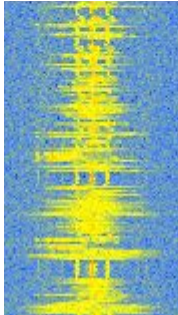
am\_demo.grc



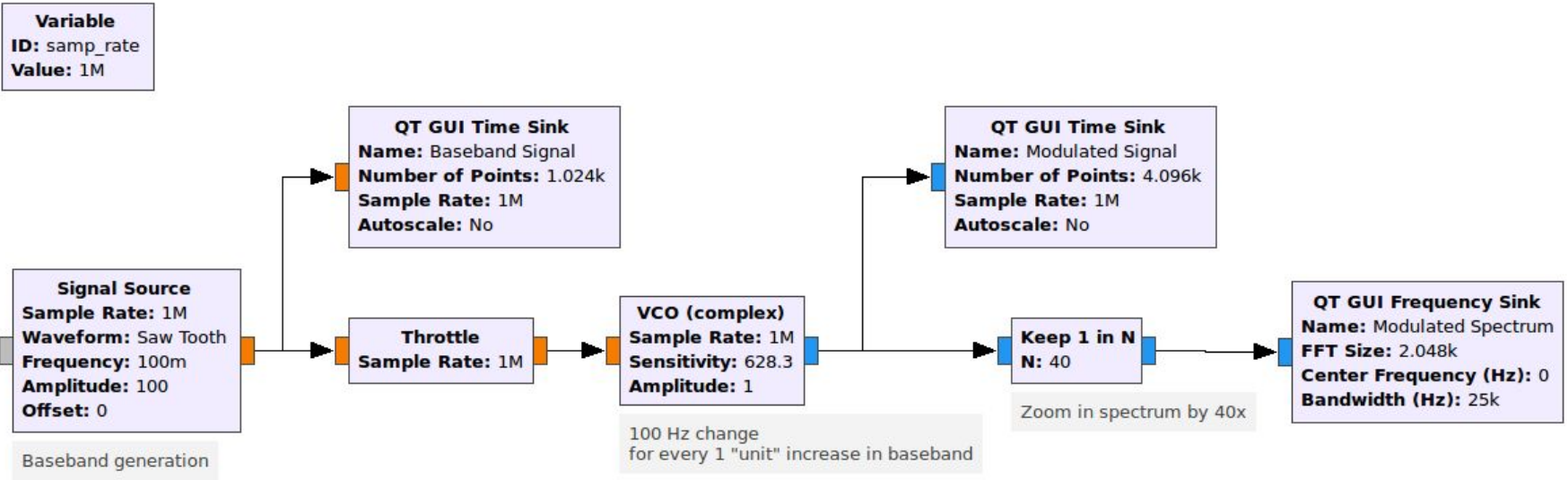


# Frequency Modulation

- Changing the frequency of a carrier wave
- Either discrete steps (Frequency Shift Keying)
- Or continuous (Broadcast FM)



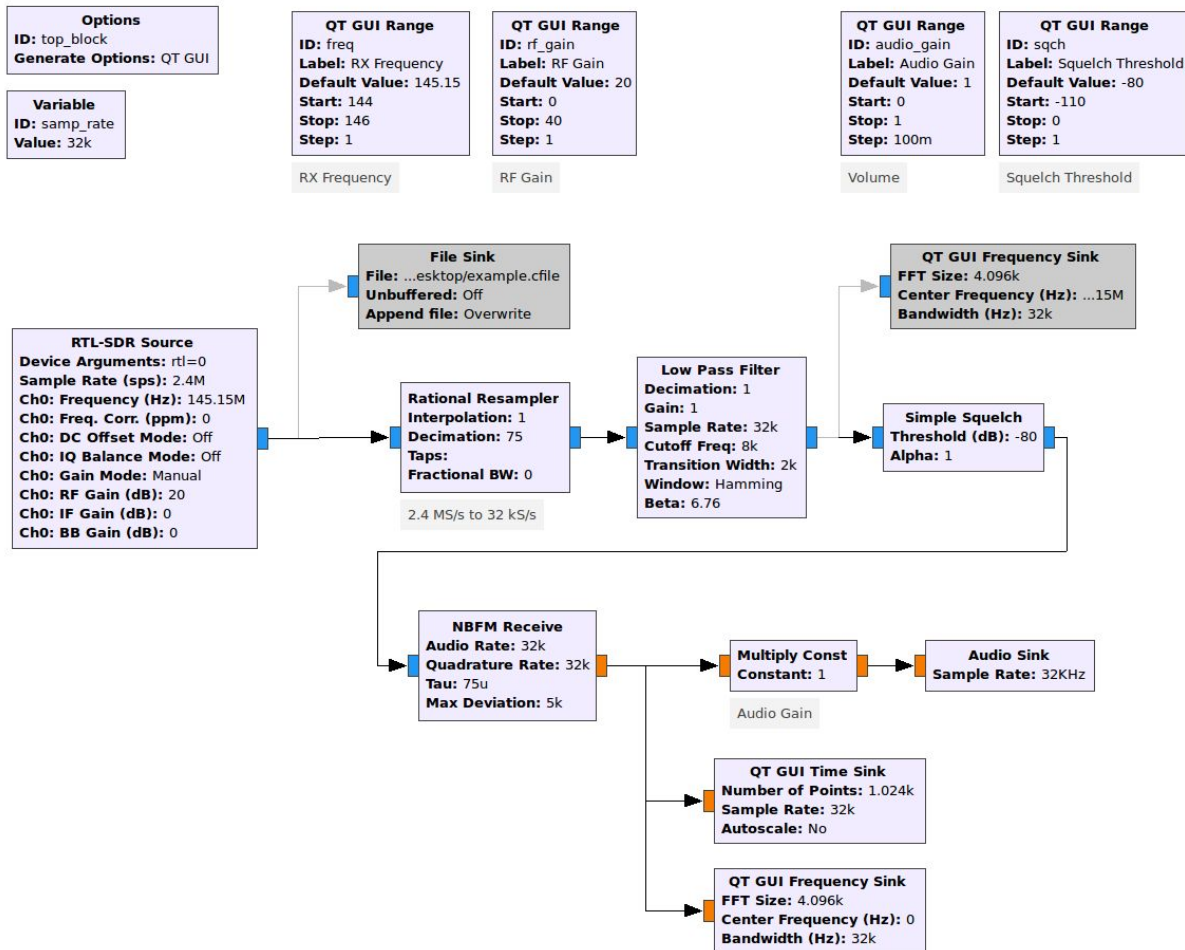
# FM Modulation Example



# Narrow Band FM

- Popular analog modulation scheme for voice transmission
  - Walkie Talkies, Land Mobile Radio
- Could implement each step of the modulation and demodulation
- GNU Radio already has it packaged

# Narrow Band FM Receiver



# NBFM Receiver - Notes

- Soundcards will support different rates, 32 and 44.1 kHz pretty universal
- Thoughtful selection of SDR sampling rate makes decimation simple (1/75)
  - Avoid large fractions (i.e. 1023/127) as they require LOTS of computation
- Squelch is in dB Full Scale, not dBm or dbW
  - GNU Radio has no way of knowing an absolute power level
- NBFM block
  - Can decimate, but usually set output and input sample rates to the same
  - Deviation and pre-emphasis ( $\tau$ ) are dependent on the transmitter, default values will work in most cases

# Underruns

- Soundcards and transmitters are hard-realtime systems, you must supply enough data to keep them always running
  - Failing to do so will cause an “underrun”
  - In RF will produce gaps in the transmission and splatter
  - In Audio will produce gaps and clicks
- GNU Radio will print “U” for underruns with USRPs and “aU” for soundcards (audio Underrun)

# Two Clock Problem

- SDR Transmitter or receiver has an internal reference oscillator, so does a soundcard
- If the two references are not EXACTLY the same there's a problem
  - Source (producer) frequency > Sink (consumer) means too many samples are available, will build up a backlog of data to handle
    - In to Out delay will increase (Audio will lag)
  - Source < Sink means not enough data is available, underruns will occur

# Mitigating the Two-Clock Problem

- Use the same reference oscillator for source and sink sample clocks (ADCs & DACs)
  - Great answer if using the same hardware for both, difficult (or impossible) with an SDR and soundcard
- Increase buffer sizes
  - Store more data before telling output to start
  - Reduces how often underruns occur
    - I.E. run out of data once a minute rather than 0.1 seconds



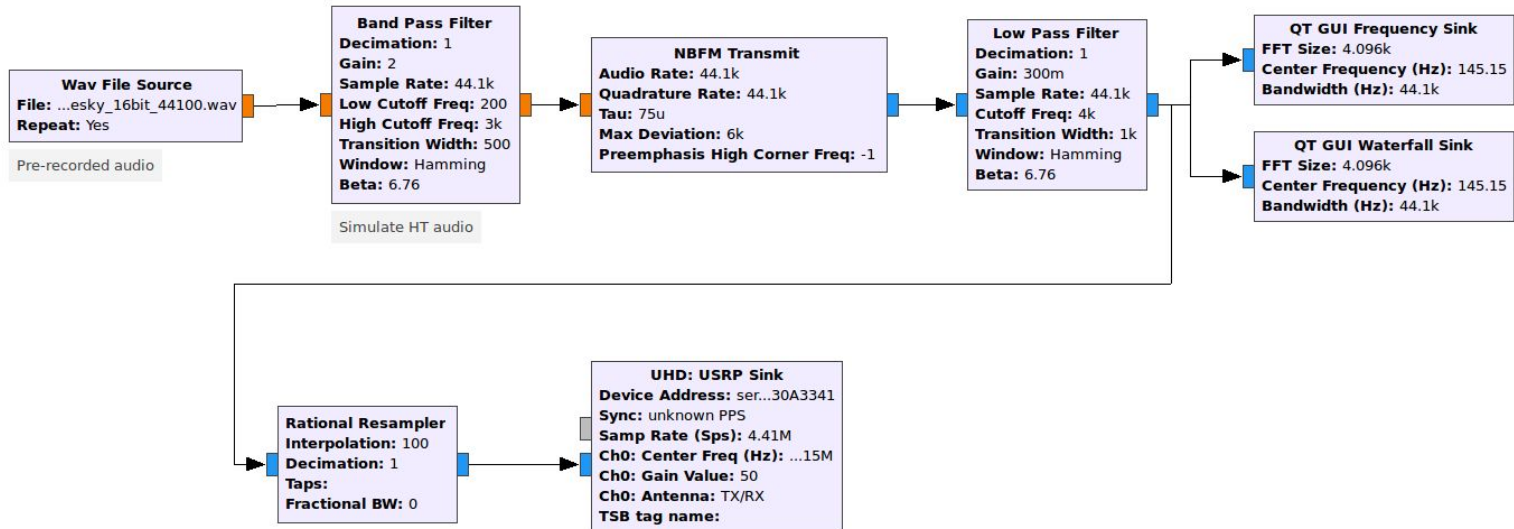
# NBFM Transmitter

**Options**  
ID: top\_block  
Generate Options: QT GUI

**Variable**  
ID: samp\_rate  
Value: 44.1k

**QT GUI Range**  
ID: rf\_gain  
Label: TX RF Gain  
Default Value: 50  
Start: 0  
Stop: 60  
Step: 1

**QT GUI Range**  
ID: freq  
Label: TX Frequency  
Default Value: 145.15  
Start: 144  
Stop: 146  
Step: 100m



# NBFM Transmitter - Notes

- USRP hardware sink sets transmit frequency, RF gain, and expected sample rate
  - USRP B200 (my demo hardware) is very flexible in sample rates, usually hardware will support specific rates
- Software interpolation/decimation will have sharper (better) filtering than FPGA or analog filters
  - This is a generalization but usually true
  - Interpolating by 100x means we have a clean signal but still very manageable sample rate (4.41 MS/s, easy for USB)
- Use the time and frequency sinks to plot signals at different points (think spectrum analyzer and oscilloscopes when debugging)
- Confirm functionality off the air before including hardware (simulation)
- FM is forgiving with filtering
  - Accidentally generated 6 kHz deviation, filtered to 4k Hz, received with 5 kHz, still works
  - Partially thanks to filter transition bandwidth

# Useful Tips

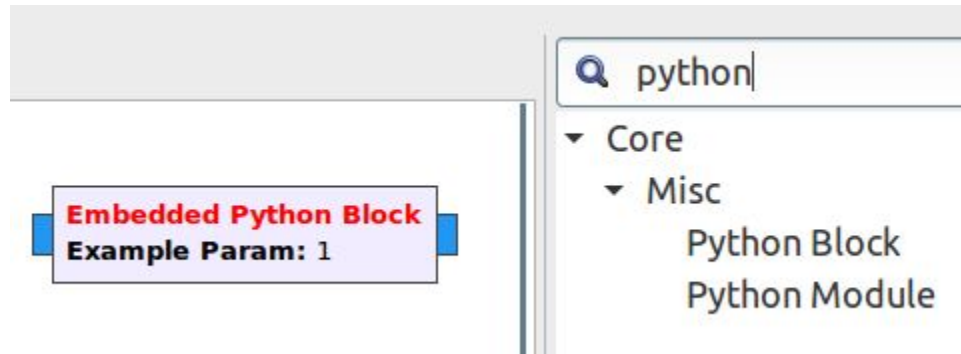
- Test/develop using a pre-recorded audio file
  - Expected format is 16 bit real valued samples
  - Sample rate chosen as 32 kHz to match what a soundcard (Mic in) would likely generate
- Add comments
  - Text box in the “Advanced” tab of each block
- Use variables and sliders (“Range” in QT)
  - Lets you experiment quickly with values to hand tune performance

# Programming Languages

- GNU Radio has a core written in C++
  - The main engine and all default blocks are C++
- Python is wrapped around the C++
  - Generally considered more experimenter friendly
  - Only small performance hit as main work is done in C++ land
- GRC is entirely written in Python
  - But again, the engine is C++, so best of both worlds

# Python Block

- Lets draw back the curtain and peek at the insides
- The “Embedded Python Block” lets you add custom code to a GRC flowgraph very easily
  - Code is stored in the .grc file
  - Default template supplies basic features



# Embedded Python Block

- Add a “Python Block” to the flowgraph, open it and click “Open in Editor” and use the Default
- The template has all the main features of a GNU Radio block setup already

```
"""
```

```
Embedded Python Blocks:
```

```
Each time this file is saved, GRC will instantiate the first class it finds  
to get ports and parameters of your block. The arguments to __init__ will  
be the parameters. All of them are required to have default values!  
"""
```

```
import numpy as np
```

```
from gnuradio import gr
```

```
class blk(gr.sync_block): # other base classes are basic_block, decim_block, interp_block  
    """Embedded Python Block example - a simple multiply const"""
```

```
    def __init__(self, example_param=1.0): # only default arguments here
```

```
        """arguments to this function show up as parameters in GRC"""
```

```
        gr.sync_block.__init__(
```

```
            self,
```

```
            name='Embedded Python Block', # will show up in GRC
```

```
            in_sig=[np.complex64],
```

```
            out_sig=[np.complex64]
```

```
        )
```

```
        # if an attribute with the same name as a parameter is found,
```

```
        # a callback is registered (properties work, too).
```

```
        self.example_param = example_param
```

```
    def work(self, input_items, output_items):
```

```
        """example: multiply with constant"""
```

```
        output_items[0][:] = input_items[0] * self.example_param
```

```
        return len(output_items[0])
```

# Headers and Includes

```
"""
```

```
Embedded Python Blocks:
```

```
Each time this file is saved, GRC will instantiate the first class it finds  
to get ports and parameters of your block. The arguments to __init__ will  
be the parameters. All of them are required to have default values!  
"""
```

```
import numpy as np  
from gnuradio import gr
```

- The red text surrounded by quotes is a comment explaining how the template works
- The import lines pull in code from gnuradio and numpy
  - numpy is a Python library of math functions that GNU Radio uses extensively
- You could add more imports to use other libraries



# Class and Initialization

```
class blk(gr.sync_block): # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, example_param=1.0): # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Embedded Python Block', # will show up in GRC
            in_sig=[np.complex64],
            out_sig=[np.complex64]
        )
```

- GNU Radio has several types (or “classes”) of blocks
  - We’re using a sync block since input and output rates are the same (synchronous)
- The next comment will appear in the block documentation tab
- The “\_\_init\_\_” function setups (initializes) our block
  - We have one parameter called example\_param with a default value of 1.0

# Block Initialization

```
class blk(gr.sync_block): # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, example_param=1.0): # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Embedded Python Block', # will show up in GRC
            in_sig=[np.complex64],
            out_sig=[np.complex64]
        )
```

- GNU Radio already knows a lot about blocks. We just have to fill in the specific details by calling `gr.sync_block.__init__(...)`
  - name is just for humans
- `in_sig/out_sig` is the “signature” of the input/output
  - How many channels, what type of data (1 channel of complex data)
  - The data types are numpy since this is Python

# Block Initialization - Continued

**Embedded Python Block**  
**Example Param: 1**

- `in_sig=[np.complex64, np.float32]` would be 1 channel complex and 1 channel real floats
- If you want to be able to change a value while the flowgraph is running (with a Range slider for instance) then create a “class attribute” like the following:

```
# if an attribute with the same name as a parameter is found,  
# a callback is registered (properties work, too).  
self.example_param = example_param
```

ID	epy_block_0
Code	Open in Editor
<u>Example Param</u>	1.0

- GRC will automatically add code to update the value correctly
  - Only values with an underline in GRC can be changed at runtime

# Doing Work on Samples

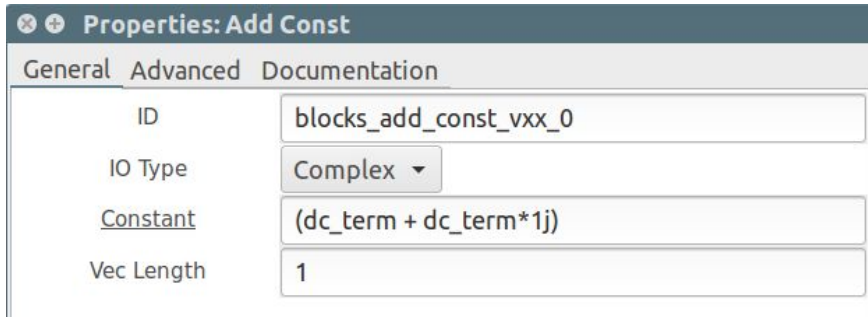
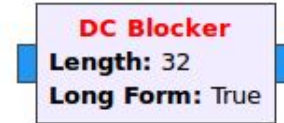
```
def work(self, input_items, output_items):  
    """example: multiply with constant"""  
    output_items[0][:] = input_items[0] * self.example_param  
    return len(output_items[0])
```

---

- The main purpose of most blocks is to do something with or to samples
  - GNU Radio will call the work function with a bunch of input samples and a place to put the output samples
- The default template multiplies each sample by a value (example\_param)
- We need to tell GNU Radio how many samples we've produced
  - In this case we've used all the input to make the same number of output samples
  - The len function gives the length of the output\_items array, so we return that number to GNU Radio's engine
- Clearly some Python knowledge is needed, but most of the heavy lifting already done

# DC Offset Example

- Same template but cleaned up
- Let's introduce a DC component to the signal
  - Usually a terrible idea
  - Could have used an Add Const block



**Properties: Add Const**

General Advanced Documentation

ID	blocks_add_const_vxx_0
IO Type	Complex ▾
Constant	(dc_term + dc_term*1j)
Vec Length	1

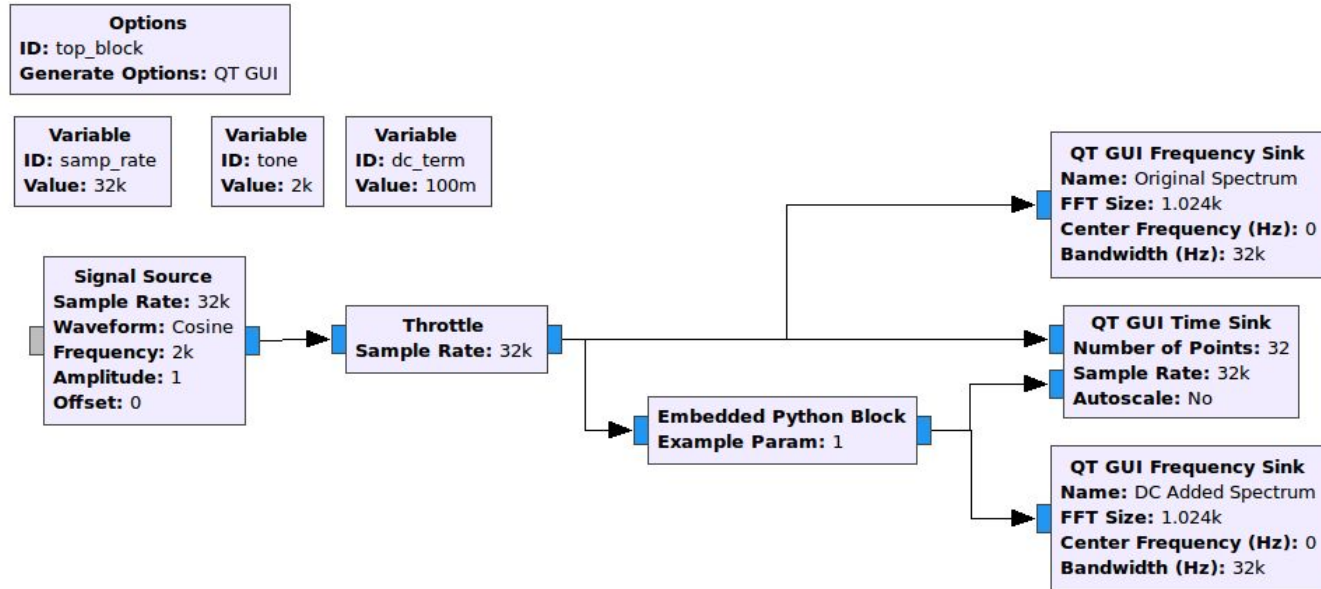
A screenshot of a software interface showing the properties of an 'Add Const' block. The title bar says 'Properties: Add Const'. There are three tabs: 'General', 'Advanced', and 'Documentation'. The 'General' tab is selected. Below the tabs is a table with four rows: 'ID' with value 'blocks\_add\_const\_vxx\_0', 'IO Type' with a dropdown menu showing 'Complex', 'Constant' with the expression '(dc\_term + dc\_term\*1j)', and 'Vec Length' with the value '1'.

Trivia:

Can remove a DC offset using the DC Blocker

# DC Offset Test Setup

- Basic testing setup with an Embedded Python Block



```
import numpy as np
from gnuradio import gr

class blk(gr.sync_block):
    # Block Documentation
    """DC Addition Block - Surely more is better!"""

    def __init__(self, dc_term=0.1): # One parameter

        gr.sync_block.__init__(
            self,
            name='DC Addition', # Will show up in GRC
            in_sig=[np.complex64], # Complex float 32 bit pairs
            out_sig=[np.complex64] # Complex float 32 bit pairs
        )

        self.dc_term = dc_term

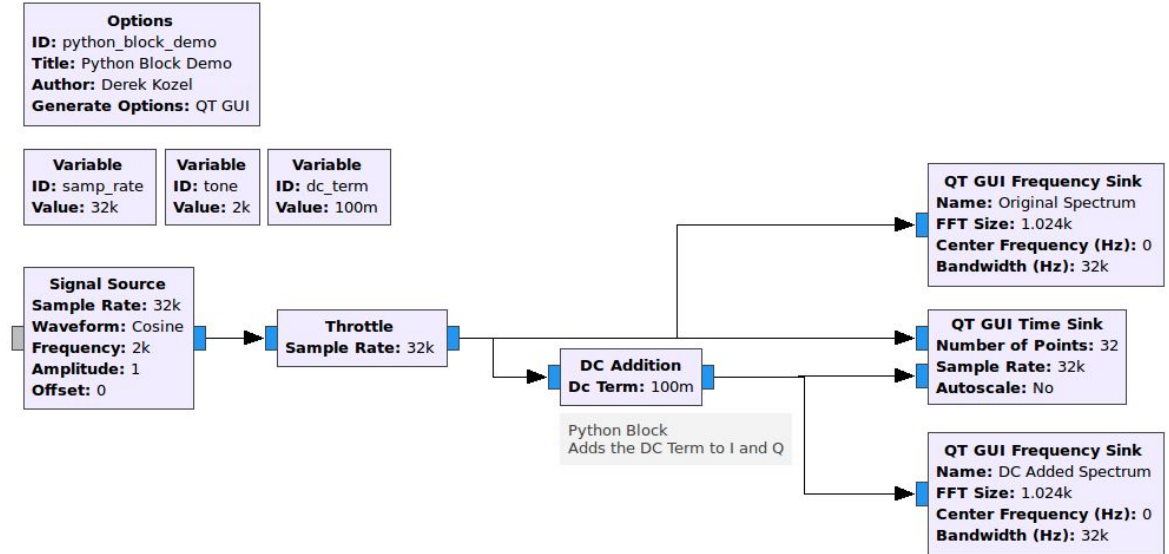
    def work(self, input_items, output_items):

        # Add the value of "dc term" to the I and Q parts of the signal
        # For example: output = input + (0.1 + j0.1)
        output_items[0][:] = input_items[0] + np.complex64(self.dc_term+self.dc_term*1j)

        # Tell GNU Radio's scheduler how many samples we are outputting
        return len(output_items[0])
```

# DC Offset Results

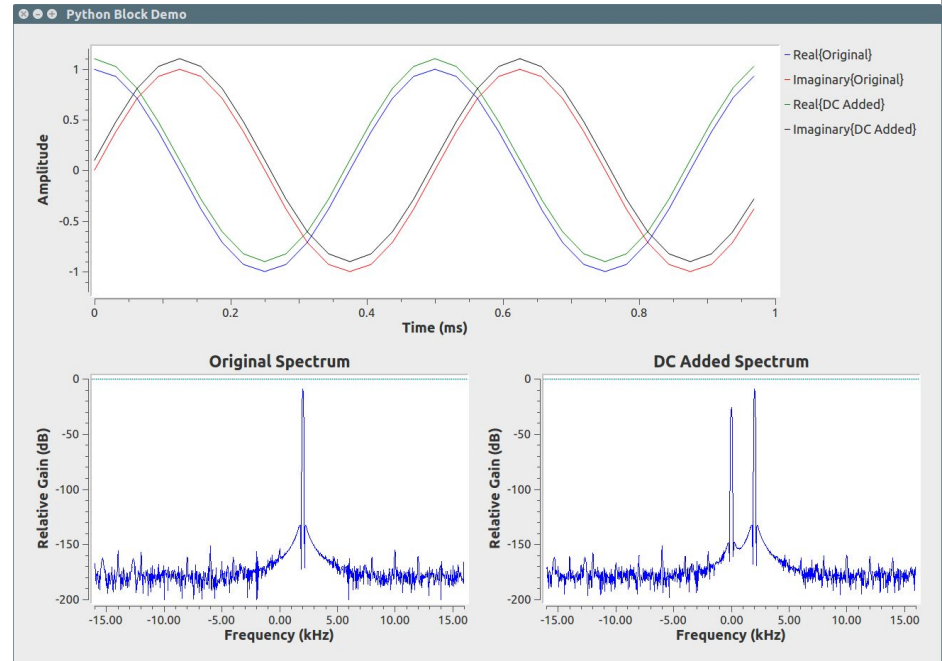
- Looks like a real block!





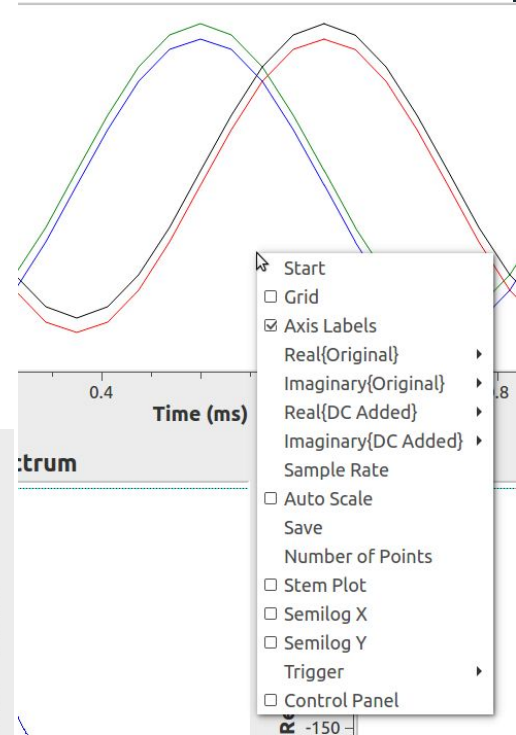
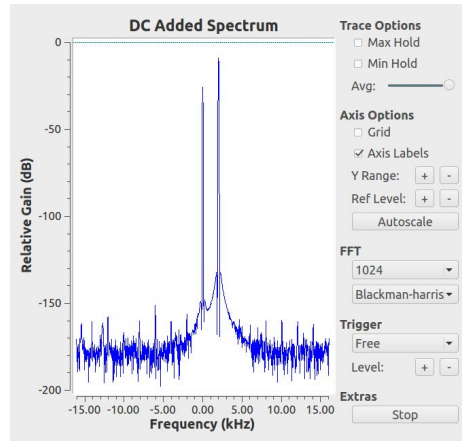
# DC Offset Results

- DC Offset clearly visible in time and frequency



# Quick Tips

- Click on the line labels in the Time plot to hide or show a particular line
  - Works on other visual sinks too
- Middle mouse click on a QT plot to bring up a menu of options.
- Enable a Control Panel in the Advanced Tab



# User Manual and Documentation

- A bit spread out and wanting in depth in spots
- User Manual: [www.gnuradio.org/doc/doxygen](http://www.gnuradio.org/doc/doxygen)
  - Generated from the C++
  - Useful for finding out more about blocks
  - Talks about the design of the core engine and code
- Python Manual: [www.gnuradio.org/doc/sphinx](http://www.gnuradio.org/doc/sphinx)
  - Generated from the Python
  - Does not cover many of the topics in main manual
  - Likely to be combined with the C++ in the next year

# User Manual and Documentation

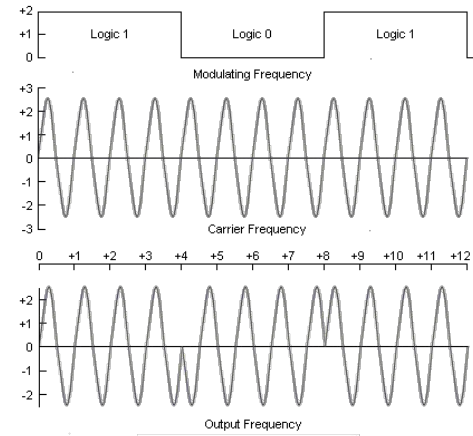
- Wiki: <http://wiki.gnuradio.org>
  - Several sets of tutorials
  - Presentations from other classes and events
  - Working groups and developer info
  - GNU Radio Conference info
    - Links to videos and slides from the talks
  - Lots of outdated pages, getting cleaner over time

# Main Website

- [www.gnuradio.org](http://www.gnuradio.org)
- Blog
  - Short and long posts about significant events
- Releases
  - Description of changes in new versions
- Links to everything on the previous page

# Phase Modulation

- Introduce changes in the carrier's phase to signal information





# Wrapping Up

# Thanks for Coming

- Questions?
- The latest version of these slides can always be found at

<http://www.derekkozel.com/talks>

- Twitter: @derekkozel
- Email: derek@bitstovolts.com
- Slides are licenced as Creative Commons Attribution-ShareAlike 4.0 International
  - <https://creativecommons.org/licenses/by-sa/4.0/>
- Examples are GNU General Public License v3.0 or later

