Introduction to GNU Radio

Using GRC and Python for Software Defined Radio

Wylie Standage-Beier

GRCon 2025

September 8, 2025

Agenda

Resources

- GNU Radio Documentation: www.gnuradio.org/doc/doxygen/
- Tutorials: wiki.gnuradio.org/index.php?title=Tutorials
- Community: chat.gnuradio.org
- Workshop materials: github.com/thewyliestcoyote/grcon25-workshop

Introduction to SDR

- What is Software Defined Radio?
- IQ Data Fundamentals
- Key SDR Concepts

Software Defined Radio

- Radio systems implemented in software
- Flexibility to change modulation and protocols
- Faster implementation
- Key components will always be hardware: RF frontend, ADC/DAC, Filter, Amplitude, Antennas
- Applications: Communications, Research, Education

IQ Data

- Complex representation: $s(t) = I(t) + jQ(t)$
- Magnitude: $|s(t)| = \sqrt{I^2 + Q^2}$
- Phase: $\phi(t) = \text{atan2}(Q, I)$
- Benefits: Negative frequencies, Simplified math

GNU Radio Companion

- Visual Flow Graph Design
- Block Types and Categories
- Building Your First Flowgraph
- Best Practices and Tips

GRC Interface

- Block Library: Available processing blocks
- Canvas: Design area for flowgraphs
- Properties: Block configuration
- Console: Messages and errors
- Variables: Global parameters

Working with GRC - Getting Started

- **Creating a New Flowgraph**: File → New
- **Options Block**: Auto-generated, configures runtime
- **Sample Rate**: Set globally via variable block
- **Generate Options**: GUI QT, No GUI, or Hier Block
- **Run to Completion**: Enable for batch processing

Data Types in GRC

Stream Types

- Complex Float 32 (blue)
- Float 32 (orange)
- Int 32 (green)
- Short (yellow)
- Byte (magenta)

Port Colors

- Blue: Complex data
- Orange: Real data
- Green: Integer data
- Red: Message ports
- Gray: Disabled/Error

PDU vs Stream Processing

Stream Processing:

- Continuous sample flow
- Fixed-rate processing
- Back-pressure control
- Real-time operation

Stream Characteristics:

- Buffered data transfer
- Scheduler managed
- Deterministic timing*
- Fixed rate

PDU (Message) Processing:

- Discrete packets
- Asynchronous handling
- Variable length data
- Metadata support

PDU Use Cases:

- MAC layer protocols
- Control messages
- Bursty communications
- Network packets
- Stream to PDU conversion

Connecting Blocks

- **Click and Drag**: From output to input port
- **Type Matching**: Ports must have compatible types
- **Rate Matching**: Consider sample rate changes
- **Multiple Outputs**: One output can feed multiple inputs
- **Throttle Block**: Required for non-hardware flows

Common Pitfalls:

- Missing throttle in simulation
- Type mismatches (use type converters)
- Sample rate inconsistencies

GRC Tips and Tricks

- **Ctrl+F**: Find blocks quickly
- **Ctrl+D**: Duplicate selected blocks
- **Ctrl+R**: Rotate block
- **Ctrl+E**: Enable/Disable block
- **Middle Click**: Delete connection
- **Shift+Click**: Select multiple blocks

Workflow Tips:

- Save frequently (Ctrl+S)
- Use descriptive variable names
- Document with Note blocks
- Version control .grc files

Block Taxonomy

Sources & Sinks

Data I/O:

- Signal/Noise Source
- File Source/Sink
- Audio Source/Sink
- Network (UDP/TCP)

Hardware:

- USRP Source/Sink
- RTL-SDR Source
- PlutoSDR

Signal Processing

Rate Change:

- Decimating FIR
- Interpolating FIR
- Rational Resampler

Transforms:

- FFT/IFFT
- Add/Multiply
- Mag/Phase

Modulation

Modulators:

- AM/FM/PM
- PSK/QAM
- OFDM/GFSK

Synchronization:

- Costas Loop
- Clock Recovery
- AGC

Stream Control:

- Mux/Demux
- Throttle
- Head/Skip

Sync Blocks

What are Sync Blocks?

- Fixed 1:1 input/output ratio
- Process samples synchronously
- Most common block type
- Predictable data flow

Examples:

- Add/Multiply blocks
- Complex to Mag/Phase
- Type converters
- Math operations

Characteristics:

- `work()` processes N samples
- Returns number processed
- No rate change
- Simple to implement

Use Cases:

- Signal arithmetic
- Format conversion
- Simple filtering
- Measurement blocks

Source and Sink Blocks

Source Blocks:

- No input ports
- Generate or acquire data
- Drive the flowgraph

Source Examples:

- Signal Source
- Noise Source
- File Source
- Audio Source
- USRP/RTL-SDR Source
- Random Source

Sink Blocks:

- No output ports
- Consume data
- End points of flowgraph

Sink Examples:

- File Sink
- Audio Sink
- GUI displays (Time, FFT)
- USRP/RTL-SDR Sink
- Null Sink
- Vector Sink

Decimation and Interpolation Blocks

Decimation Blocks:

- Reduce sample rate
- N inputs $\rightarrow$ 1 output
- Anti-aliasing filtering
- Factor must be integer

Decimation Examples:

- Decimating FIR Filter
- Keep 1 in N
- Low Pass + Decimation
- Polyphase Decimator

Interpolation Blocks:

- Increase sample rate
- 1 input $\rightarrow$ N outputs
- Anti-imaging filtering
- Factor must be integer

Interpolation Examples:

- Interpolating FIR Filter
- Repeat
- Low Pass + Interpolation
- Polyphase Interpolator

General and Basic Blocks

General Blocks:

- Variable input/output ratio
- Call `consume()` manually
- More complex than sync
- Flexible processing

General Examples:

- Correlate Access Code
- Packet Encoder/Decoder
- Stream CRC32
- Protocol formatters

Basic Blocks:

- Most flexible type
- Handle any I/O pattern
- Manual forecast/consume
- Complex implementations

Tagged Stream Blocks:

- Length-tagged packets
- Process complete bursts
- Preserve packet boundaries
- Used in packet radio

Tagged Streams

What are Tagged Streams?

- Metadata attached to samples
- Key-value pairs (PMT format)
- Synchronous with data flow
- Precise sample timing

Common Tags:

- `packet_len`: Burst length
- `rx_time`: Timestamp
- `rx_freq`: Center frequency
- `rx_rate`: Sample rate

Use Cases:

- Burst transmission
- Time synchronization
- Dynamic parameters
- Packet boundaries
- Channel info

Key Blocks:

- Stream to Tagged Stream
- Tagged Stream Align
- Tagged Stream Mux
- Burst Tagger
- Tag Debug

Variables and Parameters

- **Variable Block**: Define reusable values
- **QT GUI Range**: Runtime adjustable sliders
- **QT GUI Entry**: Text input for parameters
- **QT GUI Chooser**: Dropdown/button selection
- **Parameter Block**: Command-line arguments

Example Variables:

- samp_rate = 2e6
- center_freq = 100e6
- gain = 10

GUI Widgets and Visualization

Display Widgets

- Time Sink
- Frequency Sink
- Waterfall Sink
- Constellation Sink
- Histogram

Tips:

- Use tabs to organize complex GUIs
- Set update rates appropriately
- Enable grid and autoscale

Control Widgets

- Range Slider
- Entry Box
- Chooser
- Push Button
- Check Box

Hierarchical Blocks in GRC

- **Create**: Right-click $\rightarrow$ Create Hier Block
- **Purpose**: Reusable sub-flowgraphs
- **Pad Blocks**: Define inputs/outputs
- **Parameters**: Pass values into hier block
- **Generate**: Creates .grc and .py files

Use Cases:

- Custom modulator/demodulator pairs
- Reusable filter chains
- Complex signal processing pipelines

Debugging in GRC

Debug Tools:

- **Print/Debug Block**: Console output
- **Tag Debug**: Monitor stream tags
- **Message Debug**: View message contents
- **Number Sink**: Display numeric values
- **Vector Sink**: Capture data for analysis

Console Messages:

- Build errors (red)
- Runtime warnings (yellow)
- Debug output (white)
- Performance metrics (green)

Performance Optimization

Buffer Settings:

- **Output Buffer Size**: Increase for bursty data
- **CPU Affinity**: Pin blocks to specific cores
- **Max Output Items**: Control processing chunk size

Best Practices:

- Use decimation early in the chain
- Minimize type conversions
- Avoid unnecessary GUI updates
- Profile with Performance Monitor
- Consider using compiled Python blocks

Common GRC Patterns

Signal Generation:

- Signal Source → Throttle → Sink

File Processing:

- File Source → Processing → File Sink

Real-time SDR:

- USRP/RTL-SDR Source → Processing → GUI

Modulation:

- Source → Modulator → Channel → Demodulator

Python Integration

- GNU Radio Python API
- Custom Python Blocks
- Embedding in Applications
- Advanced Python Techniques

GRC to Python

- GRC generates Python code automatically
- File: top_block.py
- Fully functional application
- Can be modified and extended
- Class inherits from gr.top_block

Custom Python Blocks

- sync_block: 1:1 input/output ratio
- decimator: N:1 ratio (downsampling)
- interpolator: 1:N ratio (upsampling)
- general_block: Variable ratios
- basic_block: base class, used for multi-rate and ...

Sync Block Example

1:1 Input/Output Ratio - Signal Processing

```
import numpy as np
from gnuradio import gr

class amplitude_scale(gr.sync_block):
    def __init__(self, scale=2.0, offset=0.0):
        gr.sync_block.__init__(self, name="Amplitude Scale",
            in_sig=[np.complex64], out_sig=[np.complex64])
        self.scale, self.offset = scale, offset

    def work(self, input_items, output_items):
        output_items[0][:] = input_items[0] * self.scale + self.offset
        return len(output_items[0])
```

Decimator Block Example

N:1 Downsampling - Averaging

```
from gnuradio import gr
import numpy as np

class averaging_decimator(gr.decim_block):
    def __init__(self, decimation=4):
        gr.decim_block.__init__(self, name="Averaging Decimator",
            in_sig=[np.float32], out_sig=[np.float32], decim=decimation)
        self.decimation = decimation

    def work(self, input_items, output_items):
        in0, out = input_items[0], output_items[0]
        for i in range(len(out)):
            out[i] = np.mean(in0[i*self.decimation:(i+1)*self.decimation])
        return len(out)
```

Interpolator Block Example

1:N Upsampling - Zero Padding

```
from gnuradio import gr

class zero_pad_interpolator(gr.interp_block):
    def __init__(self, interpolation=4):
        gr.interp_block.__init__(self, name="Zero Pad Interpolator",
            in_sig=[np.complex64], out_sig=[np.complex64], interp=interpolation)
        self.interp = interpolation

    def work(self, input_items, output_items):
        out = output_items[0]
        out[::self.interp] = input_items[0]
        out.flat[1::self.interp] = 0
        return len(out)
```

General Block Example (1/2)

Variable Rate - Packet Detector Setup

```
from gnuradio import gr
import numpy as np

class packet_detector(gr.basic_block):
    def __init__(self, threshold=0.5):
        gr.basic_block.__init__(self, name="Packet Detector",
            in_sig=[np.float32], out_sig=[np.float32])
        self.threshold = threshold

    def forecast(self, noutput_items, ninput_items_required):
        ninput_items_required[0] = noutput_items
```

General Block Example (2/2)

Variable Rate - Work Function

```
def general_work(self, input_items, output_items):
    in0, out = input_items[0], output_items[0]
    consumed = produced = 0

    for i, sample in enumerate(in0):
        if abs(sample) > self.threshold:
            out[produced] = sample
            produced += 1
        consumed = i + 1
        if produced >= len(out):
            break

    self.consume(0, consumed)
    return produced
```

Message Passing Block

Asynchronous Message Processing

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

class message_statistics(gr.basic_block):
    def __init__(self):
        gr.basic_block.__init__(self, name="Message Statistics",
            in_sig=None, out_sig=None)
        self.message_port_register_in(pmt.intern("data_in"))
        self.message_port_register_out(pmt.intern("stats_out"))
        self.set_msg_handler(pmt.intern("data_in"), self.handle_message)

    def handle_message(self, msg):
        data = pmt.to_python(msg)
        stats = {'mean': float(np.mean(data)), 'std': float(np.std(data)),
                 'max': float(np.max(data)), 'min': float(np.min(data))}
        self.message_port_pub(pmt.intern("stats_out"), pmt.to_pmt(stats))
```

Tagged Stream Block Example (1/2)

Packet Encoder with Stream Tags

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

class packet_encoder(gr.tagged_stream_block):
    def __init__(self, packet_len=100, preamble=[1,-1,1,-1]):
        gr.tagged_stream_block.__init__(self,
            name="Packet Encoder",
            in_sig=[np.float32],
            out_sig=[np.complex64],
            length_tag_name="packet_len")
        self.packet_len = packet_len
        self.preamble = np.array(preamble, dtype=np.complex64)
        self.set_tag_propagation_policy(gr.TPP_DONT)

    def work(self, input_items, output_items):
        in0 = input_items[0]
        out = output_items[0]

        # Get packet length from tag
        tags = self.get_tags_in_window(0, 0, len(in0))
        packet_len = len(in0)  # Default to input length

        for tag in tags:
            if pmt.equal(tag.key, pmt.intern("packet_len")):
                packet_len = pmt.to_long(tag.value)
                break
```

Tagged Stream Block Example (2/2)

Packet Encoder - Processing and Tagging

```
# Add preamble
preamble_len = len(self.preamble)
out[:preamble_len] = self.preamble

# Encode data (simple BPSK for example)
data_len = min(packet_len, len(in0))
encoded = np.array([1+0j if x > 0 else -1+0j for x in in0[:data_len]])
out[preamble_len:preamble_len+data_len] = encoded

# Add tags for packet boundaries
self.add_item_tag(0,  # output port
    self.nitems_written(0),  # absolute offset
    pmt.intern("packet_start"),  # tag key
    pmt.from_long(preamble_len + data_len),  # tag value
    pmt.intern(self.name()))  # source

# Add tag for modulation type
self.add_item_tag(0,
    self.nitems_written(0) + preamble_len,
    pmt.intern("modulation"),
    pmt.intern("BPSK"),
    pmt.intern(self.name()))

return preamble_len + data_len
```

Reading Tag Values and Updating State

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

class simple_tag_reader(gr.sync_block):
    def __init__(self):
        gr.sync_block.__init__(self, name="Simple Tag Reader",
            in_sig=[np.complex64], out_sig=[np.complex64])
        self.gain = 1.0
        self.frequency = 0.0

    def work(self, input_items, output_items):
        in0 = input_items[0]
        out = output_items[0]

        # Get all tags in this work call's window
        tags = self.get_tags_in_window(0, 0, len(in0))

        # Process each tag and update member variables
        for tag in tags:
            key = pmt.to_python(tag.key)
            value = pmt.to_python(tag.value)

            if key == "gain":
                self.gain = value
            elif key == "freq":
                self.frequency = value

        # Copy input to output (pass-through)
        out[:] = in0
```

Composite Block - FM Demodulator Setup

```
from gnuradio import gr, blocks, analog, filter

class fm_demod_cf(gr.hier_block2):
    def __init__(self, audio_rate=48000, quad_rate=480000):
        gr.hier_block2.__init__(self, "FM Demodulator",
            gr.io_signature(1, 1, gr.sizeof_gr_complex),
            gr.io_signature(1, 1, gr.sizeof_float))

        # Create internal blocks
        self.quad_demod = analog.quadrature_demod_cf(quad_rate/(2*3.14159*75e3))
        audio_decim = int(quad_rate / audio_rate)
        audio_taps = filter.firdes.low_pass(1.0, quad_rate, 15e3, 1e3)
        self.audio_filter = filter.fir_filter_fff(audio_decim, audio_taps)
```

Hierarchical Block Example (2/2)

Composite Block - Connections

```
        # Connect internal blocks to form the processing chain
        self.connect(self, self.quad_demod)
        self.connect(self.quad_demod, self.audio_filter)
        self.connect(self.audio_filter, self)

# Usage example:
if __name__ == '__main__':
    tb = gr.top_block()
    src = analog.sig_source_c(480000, analog.GR_COS_WAVE, 1000, 1, 0)
    demod = fm_demod_cf()
    sink = audio.sink(48000, '')
    tb.connect(src, demod, sink)
    tb.run()
```

Tagged Stream Example

Stream Tagging for Burst Processing

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

class burst_tagger(gr.sync_block):
    def __init__(self, burst_len=100):
        gr.sync_block.__init__(self, "Burst Tagger",
            in_sig=[np.complex64], out_sig=[np.complex64])
        self.burst_len, self.sample_count = burst_len, 0

    def work(self, input_items, output_items):
        output_items[0][:] = input_items[0]  # Pass through data

        for i in range(len(output_items[0])):
            if self.sample_count % self.burst_len == 0:
                tag = gr.tag_t()
                tag.offset = self.nitems_written(0) + i
                tag.key = pmt.string_to_symbol("burst_start")
                tag.value = pmt.from_long(self.sample_count // self.burst_len)
                self.add_item_tag(0, tag)
            self.sample_count += 1
        return len(output_items[0])
```

Signal Processing

- Digital Filters
- Modulation Techniques
- Spectral Analysis
- Synchronization

Digital Filters

- Low Pass Filter (LPF)
- High Pass Filter (HPF)
- Band Pass Filter (BPF)
- FIR and IIR implementations
- Key parameters: Cutoff frequency, Transition width

Modulation Techniques Overview

Analog Modulation

Amplitude (AM):

- DSB-FC (Full Carrier)
- DSB-SC (Suppressed)
- SSB (Single Sideband)
- VSB (Vestigial)

Frequency (FM):

- NBFM (Narrowband)
- WBFM (Wideband)
- PM (Phase Mod)

Applications:

- AM/FM Radio
- TV Broadcasting
- Amateur Radio

Digital Modulation

Amplitude Shift:

- OOK (On-Off Keying)
- ASK (Amplitude Shift)
- PAM (Pulse Amplitude)

Frequency/Phase:

- FSK (2-FSK, 4-FSK)
- MSK (Minimum Shift)
- GMSK (Gaussian MSK)
- BPSK/QPSK/8PSK
- DPSK (Differential)

Quadrature:

- QAM (16, 64, 256)
- OFDM (Orthogonal)

GNU Radio Blocks

Modulators:

- AM/FM/NBFM Mod
- PSK Mod/Demod
- QAM Mod/Demod
- GFSK Mod/Demod
- OFDM Tx/Rx

Supporting Blocks:

- Constellation Object
- Symbol Sync
- Costas Loop
- Symbol Sync
- AGC
- FLL Band-Edge

Implementing Modulation in GNU Radio

AM Transmitter Example:

- Signal Source →
- Add Const (DC offset) →
- Multiply (with carrier) →
- Audio/File Sink

FM Transmitter Example:

- Audio Source →
- FM Modulator →
- Rational Resampler →
- Low Pass Filter →
- USRP/File Sink

Digital PSK Example:

- Random Source →
- Constellation Modulator →
- Root Raised Cosine Filter →
- Resample to ADC rate →
- Transmit

Common Issues:

- Frequency offset
- Timing synchronization
- Phase ambiguity
- ISI (Inter-Symbol Interference)

PSK - Phase Shift Keying

What is PSK?

- Digital data encoded in signal phase
- Constant amplitude (power efficient)
- Common in satellite and wireless

Common Types:

- BPSK: 2 phases, 1 bit/symbol
- QPSK: 4 phases, 2 bits/symbol
- 8-PSK: 8 phases, 3 bits/symbol

Reference:

`https://wiki.gnuradio.org/index.php?title=Guided_Tutorial_PSK_Demodulation`

GNU Radio Blocks:

- PSK Mod/Demod
- Constellation Object
- Costas Loop (carrier sync)
- Symbol Sync

Key Parameters:

- Samples per symbol: 2-8
- Excess bandwidth: 0.35
- Gray coding for BER

QAM - Quadrature Amplitude Modulation

What is QAM?

- Combines amplitude and phase
- Higher data rates than PSK
- Used in WiFi, LTE, cable modems

Common Types:

- 16-QAM: 4 bits/symbol
- 64-QAM: 6 bits/symbol
- 256-QAM: 8 bits/symbol

GNU Radio Blocks:

- QAM Mod/Demod
- Constellation Object
- Adaptive Equalizer
- AGC

Trade-offs:

- High data rate vs SNR requirement
- More complex than PSK
- Sensitive to amplifier linearity

FSK - Frequency Shift Keying

What is FSK?

- Digital data encoded in frequency
- Constant amplitude signal
- Simple and robust

Common Types:

- Binary FSK (2 frequencies)
- GFSK (Gaussian - Bluetooth)
- GMSK (GSM cellular)

GNU Radio Blocks:

- GFSK Mod/Demod
- Frequency Mod/Demod
- Quadrature Demod
- Clock Recovery MM

Key Parameters:

- Modulation index: 0.5-5
- Deviation: 2.4-75 kHz
- BT product: 0.3-0.5

Hardware Integration

- Supported SDR Hardware
- Configuration and Setup
- Performance Considerations

Network Interfaces

TCP/UDP

- Stream IQ over network
- Remote SDR operation
- Simple but higher latency

ZeroMQ

- High-performance messaging
- Multiple patterns
- Better for real-time

Out-of-Tree Modules

- Creating Custom Modules
- Module Structure
- Distribution and Sharing

Creating OOT Modules

What are OOT Modules?

- Custom GNU Radio components
- Developed independently from core
- Packaged for easy distribution
- Can include:
  - Processing blocks
  - Hierarchical blocks
  - Python/C++ code
  - GRC block definitions

Why Create OOT Modules?

- Share custom functionality
- Maintain proprietary code
- Organize project-specific blocks
- Contribute to community

gr_modtool Workflow

1. `gr_modtool newmod mymodule`
2. `cd gr-mymodule`
3. `gr_modtool add myblock`
4. Edit block code (Python/C++)
5. `gr_modtool bind myblock`
6. `mkdir build && cd build`
7. `cmake .. && make`
8. `sudo make install`

Popular OOT Modules

- **gr-ieee802-11**: WiFi implementation
- **gr-lora**: LoRa transceiver
- **gr-satellites**: Satellite decoders
- **gr-fosphor**: GPU spectrum display

OOT Module Structure

Directory Layout:

```
gr-mymodule/
|-- CMakeLists.txt
|-- python/
|   |-- __init__.py
|   |-- myblock.py
|   `-- qa_myblock.py
|-- lib/
|   |-- myblock_impl.cc
|   `-- myblock_impl.h
|-- include/mymodule/
|   `-- myblock.h
|-- grc/
|   `-- mymodule_myblock.block.yml
|-- examples/
|   `-- example_flowgraph.grc
`-- docs/
```

Key Components:

- **CMakeLists.txt**: Build configuration
- **python/**: Python implementations
- **lib/**: C++ implementations
- **include/**: Public headers
- **grc/**: GRC block definitions (YAML)
- **examples/**: Demo flowgraphs

Conclusion

- Key Takeaways
- Next Steps
- Resources

Summary

- Learned SDR fundamentals
- Built flowgraphs in GRC
- Created Python applications
- Explored hardware options
- Ready for advanced topics!

Resources

- GNU Radio Documentation: www.gnuradio.org/doc/doxygen/
- Tutorials: wiki.gnuradio.org/index.php?title=Tutorials
- Community: chat.gnuradio.org
- Workshop materials: github.com/thewyliestcoyote/grcon25-workshop

Questions?

Thank you for attending!

Welcome to GRCon25

Contact: thewyliestcoyote@gmail.com
Matrix Chat: thewyliestcoyote