Spectrum Sensing

June 15, 2024

Concepts: CR – SDR – SS

- *Cognitive Radio*: intelligent processing and flow controlling of wireless communication for effective use of radio resources, and to enable the coexistence between primary and secondary users (cognitive users).

1. Primary User (PU) is a user of the spectrum band, having a licensed access at any time in a geographical area. Multiple PUs can use the same spectrum simultaneously – MPU user cognitive radio network.
2. Secondary User (SU or CR) is licensed to access bands of the spectrum when the corresponding PU or multiple PUs are not accessing them.

The following three features set cognitive radio apart from conventional radio:

- *Cognition*: CR is aware of its physical and administrative surroundings.
- *Reconfiguration*: This cognitive understanding allows CR to select whether to dynamically and independently change its parameters.
- *Learning*: CR can benefit from the experience and test out new setups in fresh circumstances.

- *Software Defined Radio* is "**a radio in which some or all of the physical-layer functions are software defined**". Therefore, SDR replaces traditional hardware components with software algorithms, providing enhanced flexibility and adaptability in radio systems.

  SDR allows to experiment and develop prototypes and solutions for CR implementation.
- *Spectrum Sensing* is the process of gaining an understanding of the state of channel occupancy and identifying the existence of spectrum holes in a geographical area before a transmission is initiated.

  Theoretically, the spectrum sensing problem is defined as a hypotheses test based on the presence of the primary user signal:
  - H_o: No primary user signal is present (absence)
  - H_1: primary user signal is present (presence)

Spectrum Sensing

- ✓ Signal Detection
- ✓ Signal Classification
- ✓ Channel-State Estimation
- ✓ Decision Making
- ✓ Monitoring and System Management

Signal Detection

- Data Collection and Assembly:
  - Signal Reception Arrangement
  - Sampling and ADC Conversion
  - Data Storage
- Signal Conditioning
  - Data cleaning and anomaly detection
  - Narrowband Filtering and demodulation
  - Wideband Signal Decimation

→ Measurement Framework

- Energy calculation
  - Power Estimation and frequency scanning
  - Noise (SNR) Estimation
  - Thresholding

→ Machine-Learning Framework

Literature:

1. Spectrum Sensing Using Software Defined Radio for Cognitive Radio Networks: A Survey •
2. Spectrum sensing in cognitive radio: A deep learning based model •
3. A review of spectrum sensing in modern cognitive radio networks •

Data Collection and Assembly

- Enhanced signal reception
  - Diversity Antennas for reducing the impact of fading and shadowing: spatial diversity (using multiple antennas at different locations), polarization diversity (antennas with different polarization orientations), frequency diversity, and pattern diversity (antennas with different radiation patterns).
  - Enhanced Detection Sensitivity: By combining signals from multiple antennas, diversity combining techniques such as selection diversity, maximal ratio combining (MRC), or equal gain combining (EGC) can be employed to improve the detection sensitivity of spectrum sensing.
  - Multisensor Sensor Synchronization. Implementing diversity antennas adds complexity to the cognitive radio system, including hardware requirements and signal processing algorithms.

    Multiple antenna spectrum sensing • Multi-antenna receiver based on maximum-likelihood • SS Scheme for a Multi-Antenna Receiver • Deep Learning-Based Spectrum Sensing Scheme for a Multi-Antenna Receiver • SS Scheme for a Multi-Antenna Receiver •

- Sampling and ADC conversion
  - Narrow-band sampling:
    - Nyquist-based, wavelet-based,
    - Oversampling with enhanced resolution. Oversampling and undersampling. •
  - Wide-band sampling:
    - Compressive sampling (decimation), multicoset sampling
    - Data Compression and Undersampling to feed NN estimators: sigma-delta modulation

- Data Storage
  - Real-time analysis
  - Off-line training and validation
  - Data Retention Policies dictated by regulatory requirements, operational needs, or privacy considerations.
  - Data anonymization or encryption to protect sensitive information and ensure compliance with privacy regulations.
  - Data storage infrastructure for spectrum sensing applications may include local storage on cognitive radio devices, networked storage systems, or cloud-based storage solutions.

Signal Conditioning

- Narrow-band Filtering and demodulation
  - Narrow-band band pass and adaptive filtering to isolate specific signals or frequency bands of interest while suppressing noise and interference, improving the signal-to-noise ratio (SNR).
  - Demodulation involves extracting the original information carried by modulated signals, such as voice, data, or multimedia content.
- Wideband Signal Decimation
  - Channelization, filter banks, and multichannel sensing
  - Software-defined filters: Filtering implementation on software-defined radio (SDR) platforms (processing power, flexibility, cost, and power consumption).

Energy Calculation

- Narrowband Estimation •
  - Energy Detection (L_2-based estimation) and Entropy-based detection
  - Shape signal Detection : correlation, cyclostationarity, covariance, waveform-based
  - Matched Filter (L_2-based filtering)
  - Matrix decomposition-based (eigenvalue detection)

    Detection tutorial • Energy detection-based spectrum sensing machine • Maximizing Eigenvalue Using Machine Learning • FlashFFTConv • Energy detection under noise power • • Multiscale Wavelet Transform Extremum Detection With the Spectrum Energy Detection • Complex Rao Detector • LSTM detector •

    DL-based Signal detection • Exploring DL for Adaptive Energy Detection Threshold Determination: A Multistage Approach •Deep Learning for Adaptive Energy Detection Threshold •
- Wideband Estimation
  - Nyquist-based. Wavelet-based,
  - Filter-bank, and Multiple narrow bands

    Sub-Nyquist Sampling-Based Wideband Spectrum Sensing •
- Detection Performance Metrics:

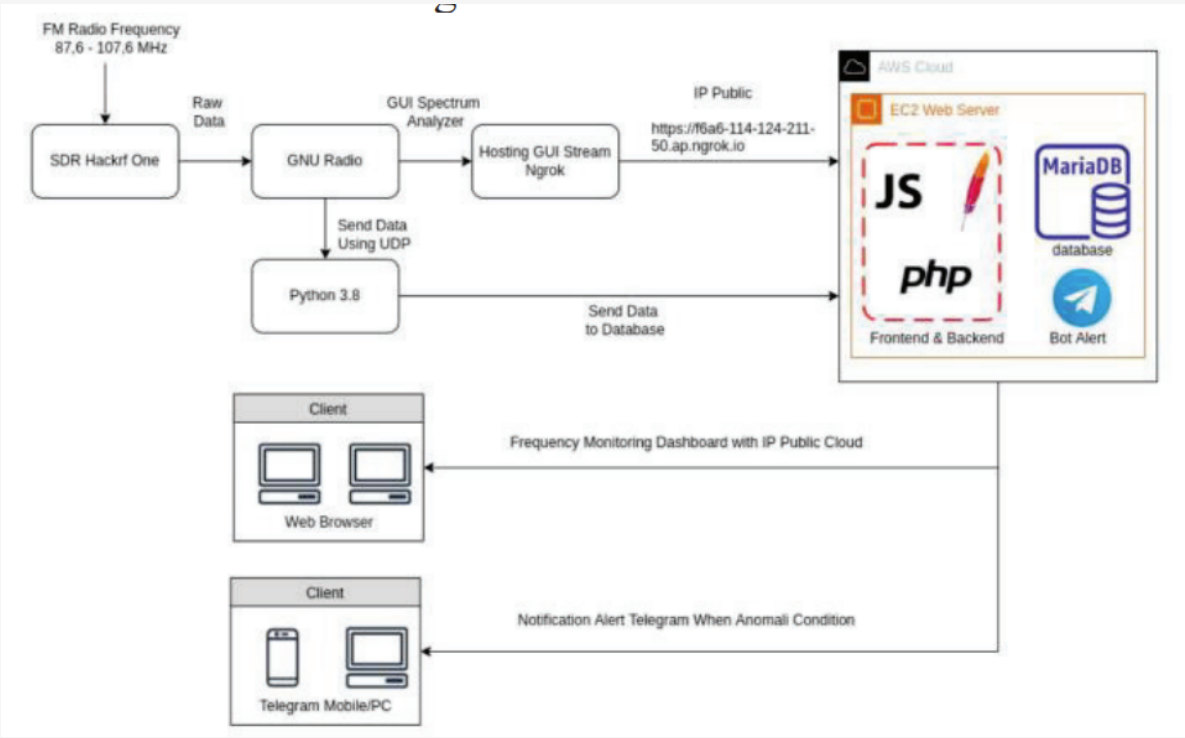
  Detection Probability, missed detection probability, False Alarm Probability, and Receiver operating characteristics curves. Signal to Noise ratio (SNR). Evaluating the practical performance of energy detector based spectrum sensing for cognitive radio •
- Cooperative cognitive radio networks

  Unwanted interference because of multipath fading and shadowing effects makes undetectable actual PU transmission (hidden-node problem).

Measurement Framework

SDR Design: Trade-offs between performance parameters (including cost, power consumption, and size) evaluated based on the specific requirements and constraints of the cognitive radio system.

Frequency Monitoring

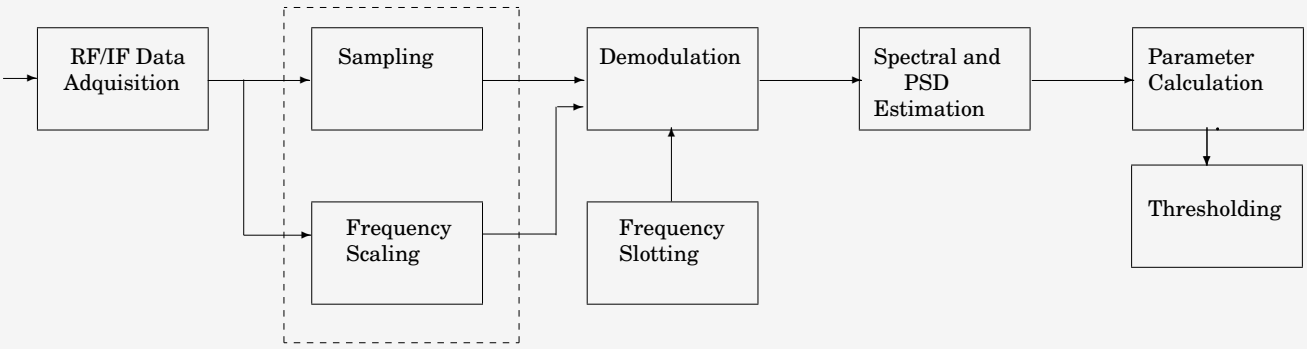


Frequency Monitoring Interface Design and System •

GNU[1] Radio is a free & open-source software development toolkit that provides signal processing blocks to implement SDR designs.

[1]GNU is a free-software operating system

SDR tasks for Narrowband Frequency Scanning



Frequency Scanning Pipeline

- IF Data Acquisition:
  1. Signal Source captures the signal.
  2. Filtering applies a low-pass filter to isolate the IF signal.
  3. Data Storage the filtered IF data.

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

class if_data_acquisition(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self, "IF Data Acquisition")

        # Parameters
        center_freq = 100e6  # center radio frequency
        samp_rate   = 2e6    # sampling rate

        # Block 1
        self.src = osmosdr.source(args="numchan=1")
        self.src.set_sample_rate(samp_rate)
        self.src.set_center_freq(center_freq)
        self.src.set_gain(30)
        # Block 2
        # Low-pass filter to isolate the IF signal
        self.lp_filter = filter.fir_filter_ccf(1, filter.firdes.low_pass(
            1, samp_rate, 200e3, 10e3, filter.firdes.WIN_HAMMING, 6.76))
        # Block 3
        # File sink to store the IF data
        self.file_sink = blocks.file_sink(gr.sizeof_gr_complex*1, "if_data.bin")

        # Connections
        self.connect(self.src, self.lp_filter, self.file_sink)
        ######
if __name__ == '__main__':
    tb = if_data_acquisition()
    tb.start()
    tb.wait()
```

- Sampling

.- Sampling Criterion: The sample rate must be at least twice the highest frequency component of the signal to accurately reconstruct the signal without aliasing. For FM radio, the highest frequency component typically includes the carrier frequency plus the deviation caused by the audio signal.

.- Bandwidth of FM Signal: FM signals have a wider bandwidth compared to AM signals. The bandwidth is determined by the carrier frequency and the maximum frequency deviation. For commercial FM radio, the bandwidth can be as wide as 200 kHz.

.- Decimation and Interpolation: In SDR systems, the initial sample rate may be very high (e.g., several MHz) to capture a wide range of frequencies. Decimation is used to reduce the sample rate for more manageable processing while preserving the signal of interest. Interpolation can be used to increase the sample rate if necessary for specific processing stages.

Typical Sample Rates for FM Demodulation:

.- RTL-SDR: Commonly used SDR hardware like the RTL-SDR typically uses sample rates in the range of 1.024 MSPS (mega samples per second) to 2.048 MSPS. These rates are sufficient to capture the full bandwidth of a commercial FM broadcast.

.- Higher-End SDRs: More advanced SDR hardware can support higher sample rates, providing greater flexibility and the ability to capture wider bandwidths or multiple channels simultaneously.

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

# Create a flow graph
fg = gr.top_block()

# Sample rate set high enough to capture the entire FM signal bandwidth (e.g., 2.048 MHz).
sample_rate = 2.048e6  # 2.048 MHz

# Signal source (e.g., from an RTL-SDR or file)
src = blocks.file_source(gr.sizeof_gr_complex, "fm_signal.dat", False)

# Throttle block to control the sample rate
throttle = blocks.throttle(gr.sizeof_gr_complex, sample_rate, True)

# Frequency translating FIR filter to shift the signal to baseband
freq_trans = filter.freq_xlating_fir_filter_ccc(1, [1], -250e3, sample_rate)

# Low-pass filter to remove high-frequency components
lpf = filter.fir_filter_ccf(1, filter.firdes.low_pass(1, sample_rate, 100e3, 10e3))

# FM demodulator
fm_demod = analog.wfm_rcv(
quad_rate=sample_rate,
audio_decimation=10,)

# Audio sink to play the demodulated signal
audio_sink = audio.sink(int(sample_rate / 10), "", True)
###### The decimation factor is chosen  to a manageable level for processing
## (e.g., reducing from 2.048 MHz to 204.8 kHz with a decimation factor of 10).

# Connect the blocks
fg.connect(src, throttle, freq_trans, lpf, fm_demod, audio_sink)

# Run the flow graph
fg.run()
```

.- *Initial Sample Rate*: Should be high enough to cover the entire FM signal bandwidth. 2 MHz is a common choice, but it can be adjusted based on the specific requirements of your SDR hardware and the signal environment.

.- *Audio Sample Rate*: Typically 44.1 kHz or 48 kHz for high-quality audio output.

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

# Create a flow graph
fg = gr.top_block()

# Define the initial sample rate (e.g., 2 MHz for capturing the FM signal)
initial_sample_rate = 2e6  # 2 MHz

# Define the audio sample rate (e.g., 48 kHz for audio output)
audio_sample_rate = 48e3  # 48 kHz

# Source block (e.g., from a file or RTL-SDR)
src = blocks.wavfile_source("fm_signal.wav", True)

# Throttle block to control the initial sample rate
throttle = blocks.throttle(gr.sizeof_gr_complex, initial_sample_rate, True)

# Frequency translating FIR filter to shift the signal to baseband
freq_trans = filter.freq_xlating_fir_filter_ccc(1, [1], -250e3, initial_sample_rate)

# Low-pass filter to remove high-frequency components
lpf = filter.fir_filter_ccf(1, filter.firdes.low_pass(1, initial_sample_rate, 100e3, 10e3))

# FM demodulator
fm_demod = analog.wfm_rcv(
quad_rate=initial_sample_rate,
audio_decimation=int(initial_sample_rate / audio_sample_rate),
)

# Audio sink to play the demodulated signal
audio_sink = audio.sink(int(audio_sample_rate), "", True)

# Connect the blocks
fg.connect(src, throttle, freq_trans, lpf, fm_demod, audio_sink)

# Run the flow graph
fg.run()
```

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

class if_data_sampling(gr.top_block):
def __init__(self):
gr.top_block.__init__(self, "IF Data Sampling")

# Parameters
center_freq = 100e6    # center radio frequency
if_freq     = 10e6     # center IFrequency
samp_rate   = 2e6      # sampling rate

# SDR Source
self.src = osmosdr.source(args="numchan=1")
self.src.set_sample_rate(samp_rate)
self.src.set_center_freq(center_freq)
self.src.set_gain(30)

# Downconvert to IF
self.downconvert = filter.freq_xlating_fir_filter_ccf(1, (1,), if_freq, samp_rate)

# File Sink to store IF data
self.sink = blocks.file_sink(gr.sizeof_gr_complex, "if_data.bin")

# Connections
self.connect(self.src, self.downconvert, self.sink)

if __name__ == '__main__':
tb = if_data_sampling()
tb.start()
tb.wait()
```

The `Throttling block` is added for controling the flow of data to match the processing rate.

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

class if_data_acquisition(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self, "IF Data Acquisition")

        # Parameters
        center_freq = 100e6  # Center Radiofrequency
        if_freq     = 10e6   # Intermediate frequency
        samp_rate   = 2e6

        # Blocks
        self.src = osmosdr.source(args="numchan=1")
        self.src.set_sample_rate(samp_rate)
        self.src.set_center_freq(center_freq)
        self.src.set_gain(30)
        self.chan_filt = filter.freq_xlating_fir_filter_ccf(1, (1,), if_freq - center_freq,
                                                            samp_rate)
        ############ Throttling block
        self.throttle = blocks.throttle(gr.sizeof_gr_complex*1, samp_rate, True)
        self.sink = blocks.file_sink(gr.sizeof_gr_complex*1, "if_data.bin")

        # Connections
        self.connect(self.src, self.chan_filt, self.throttle, self.sink)

if __name__ == '__main__':
    tb = if_data_acquisition()
    tb.start()
    tb.wait()
```

Throttling is used to control the rate at which operations are performed, ensuring that the system doesn't get overwhelmed by too many requests in a short period. Specific aspects of throttling in SDR:

.- *Sample Rate Control*: SDR systems often need to process signals at specific sample rates. A throttling block can ensure that samples are produced or consumed at a constant rate, matching the desired sample rate of the system.

.- *Buffer Management*: In SDR, data is often processed in chunks or buffers. Throttling helps manage these buffers to ensure that they do not overflow (too much data too quickly) or underflow (too little data too slowly).

.- *Synchronization*: Ensures that different components of the SDR system, which may operate at different rates, remain in sync. For instance, the rate at which data is received from an antenna must be synchronized with the rate at which it is processed and possibly transmitted.

.- *Flow Graph Control*: In many SDR platforms like GNU Radio, flow graphs (which represent the data flow through various processing blocks) include throttling blocks to control the overall flow of data through the graph. This ensures that each block processes data at the correct rate.

.- *CPU Load Management*: Throttling can also be used to manage CPU load. By controlling the rate of data processing, the system can avoid overloading the CPU, ensuring stable operation.

```
from gnuradio import gr
from gnuradio import blocks

# Create a flow graph
fg = gr.top_block()

# Source block generating random data
src = blocks.vector_source_f([1, 0, 1, 0], True)

# Throttle block to limit data flow to 1 sample per second
throttle = blocks.throttle(gr.sizeof_float, 1, True)

# Sink block to print the output
sink = blocks.vector_sink_f()

# Connect the blocks
fg.connect(src, throttle)
fg.connect(throttle, sink)

# Run the flow graph
fg.run()
```

- Decimation

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

class if_decimation(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self, "IF Decimation")

        # Parameters

        center_freq       = 100e6
        if_freq           = 10e6
        samp_rate         = 2e6
        decimation_factor = 10

        # Blocks
        self.src = osmosdr.source(args="numchan=1")
        self.src.set_sample_rate(samp_rate)
        self.src.set_center_freq(center_freq)
        self.src.set_gain(30)

        # Downconvert to IF
        self.downconvert = filter.freq_xlating_fir_filter_ccf(1, (1,), if_freq, samp_rate)

        # Decimate the signal
        self.decimate = filter.fir_filter_ccf(decimation_factor, (1,))
        decimated_samp_rate = samp_rate // decimation_factor

        # Capture decimated IF data
        self.sink = blocks.file_sink(gr.sizeof_gr_complex, "decimated_if_data.bin")

        # Connections
        self.connect(self.src, self.downconvert, self.decimate, self.sink)

if __name__ == '__main__':
    tb = if_decimation()
    tb.start()
    tb.wait()
```

Wavelet Decimation

```
from gnuradio import gr, blocks, osmosdr
import pywt
import numpy as np

class wavelet_decimation(gr.sync_block):
    def __init__(self, wavelet, level):
        gr.sync_block.__init__(self,
            name="wavelet_decimation",
            in_sig=[np.complex64],
            out_sig=[np.complex64])
        self.wavelet = wavelet
        self.level = level

    def work(self, input_items, output_items):
        data = input_items[0]
        coeffs = pywt.wavedec(data, self.wavelet, level=self.level)
        # Decimate (keep only the approximation coefficients)
        output_items[0][:] = pywt.waverec(coeffs[:-1] + [None]*self.level, self.wavelet)
        return len(output_items[0])

class if_wavelet_decimation(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self, "IF Wavelet Decimation")

        # Parameters
        samp_rate = 2e6
        center_freq = 100e6
        wavelet = 'db4'
        level = 2

        # Blocks
        self.src = osmosdr.source(args="numchan=1")
        self.src.set_sample_rate(samp_rate)
        self.src.set_center_freq(center_freq)
        self.src.set_gain(30)

        self.wavelet_dec = wavelet_decimation(wavelet, level)

        self.sink = blocks.file_sink(gr.sizeof_gr_complex, "wavelet_decimated_if_data.bin")

        # Connections
        self.connect(self.src, self.wavelet_dec, self.sink)

if __name__ == '__main__':
    tb = if_wavelet_decimation()
    tb.start()
    tb.wait()
```

Frequency scaling in SDR involves adjusting the sampling rate to match the frequency of interest for effectively capturing and processing signals, especially when dealing with different frequency bands.

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

class frequency_scaling(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self, "Frequency Scaling")

        # Parameters
        samp_rate = 2e6
        center_freq = 100e6
        scaled_samp_rate = 500e3

        # Blocks
        self.src = osmosdr.source(args="numchan=1")
        self.src.set_sample_rate(samp_rate)
        self.src.set_center_freq(center_freq)
        self.src.set_gain(30)

        # Frequency scaling
        self.resampler = filter.rational_resampler_ccf(
            interpolation=int(scaled_samp_rate),
            decimation=int(samp_rate)
        )

        # Sink
        self.sink = blocks.null_sink(gr.sizeof_gr_complex)

        # Connections
        self.connect(self.src, self.resampler, self.sink)

if __name__ == '__main__':
    tb = frequency_scaling()
    tb.start()
    tb.wait()
```

- FM Demodulation

  FM Demodulator Block processes the incoming modulated FM signal and converts it into an audio signal.

```
from gnuradio import gr
from gnuradio import analog
from gnuradio import blocks
from gnuradio import audio

# Create a flow graph
fg = gr.top_block()

# Source block: A hardware source like RTL-SDR or a simulated source
# For this example, we use a signal source to simulate an FM signal
sample_rate = 250e3  # 250 kHz
signal_freq = 100e3  # 100 kHz
fm_deviation = 75e3  # 75 kHz
audio_rate = 48e3    # 48 kHz (typical audio rate)
fm_signal = analog.sig_source_c(sample_rate, analog.GR_COS_WAVE, signal_freq, 1, 0)

# Throttle block to limit data flow rate
throttle = blocks.throttle(gr.sizeof_gr_complex, sample_rate, True)

# FM Demodulator block
fm_demod = analog.wfm_rcv(
quad_rate=sample_rate,
audio_decimation=int(sample_rate / audio_rate)
)

# Low-pass filter

# Audio sink block
audio_sink = audio.sink(audio_rate, "", True)

# Connect the blocks
fg.connect(fm_signal, throttle)
fg.connect(throttle, fm_demod)
fg.connect(fm_demod, audio_sink)

# Run the flow graph
fg.run()
```

  Low-Pass Filter for removing high-frequency noise and unwanted components.

```
from gnuradio import filter

# Low-pass filter to remove high-frequency components
lpf = filter.fir_filter_ccf(1, filter.firdes.low_pass(1, sample_rate, 100e3, 10e3))
```

Quadrature Demodulator

.-Source Block: Typically a hardware source like an RTL-SDR or another SDR device that captures the RF signal.

.-Throttling Block: Ensures that the data rate through the system is controlled. This is especially important when using file sources or simulated data sources.

.-Low Pass Filter: Filters the signal to remove unwanted frequencies.

.-Quadrature Demodulator: Converts the frequency modulated signal into an amplitude modulated signal.

.-Audio Sink: Outputs the demodulated audio signal.

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

# Create a flow graph
fg = gr.top_block()

# Source block (simulated signal for this example)
sample_rate = 1e6
src = analog.sig_source_c(sample_rate, analog.GR_SIN_WAVE, 100e3, 1, 0)

# Throttle block to control the data rate
throttle = blocks.throttle(gr.sizeof_gr_complex, sample_rate, True)

# Low Pass Filter block to remove high-frequency components
lpf_taps = filter.firdes.low_pass(1, sample_rate, 100e3, 10e3, filter.firdes.WIN_HAMMING)
lpf = filter.fir_filter_ccf(1, lpf_taps)

# Quadrature Demodulator block for FM demodulation
demod = analog.quadrature_demod_cf(1.0)

# Audio Sink block to output demodulated audio
sink = audio.sink(int(sample_rate/10), "", True)

# Connect the blocks
fg.connect(src, throttle)
fg.connect(throttle, lpf)
fg.connect(lpf, demod)
fg.connect(demod, sink)

# Run the flow graph
fg.run()
```

- Spectrum Analyzer

  Osmocom (open source mobile communications) is an open-source software project that implements multiple mobile communication standards

  .- Device Arguments: rtl=0

```
  self.osmosdr_source = osmosdr.source(args="numchan=1 rtl=0")
```

  .- Sample Rate: 2M (or as per hardware capabilities)
  .- Center Frequency (Match the center frequency of the Osmocom Source): 100M (or as per interest)
  .- Gain: 10

```
 # FFT computation

from gnuradio import gr, blocks, analog, filter, fft
import osmosdr

class spectral_calculation(gr.top_block):
     def __init__(self):
            gr.top_block.__init__(self, "Spectral Computation")

            # Parameters
            samp_rate = 2e6
            center_freq = 100e6

            # Source block
            self.src = osmosdr.source(args="numchan=1")
            self.src.set_sample_rate(samp_rate)
            self.src.set_center_freq(center_freq)
            self.src.set_gain(30)

            # FFT block
            self.fft_size = 1024
            self.fft = fft.fft_vcc(self.fft_size, True, (), True)

            # Stream to Vector for FFT
            self.stream_to_vector = blocks.stream_to_vector(gr.sizeof_gr_complex*1, self.
                                                fft_size)

            # Probe Signal for Output
            self.probe = blocks.probe_signal_vf()

            # Connections
            self.connect(self.src, self.stream_to_vector, self.fft, self.probe)

  if __name__ == '__main__':
        tb = spectral_calculation()
        tb.start()
        tb.wait()
        # Example to read FFT output
        fft_output = tb.probe.level()
        print("FFT Output:", fft_output)
```

.- Source Block: Set the frequency, sample rate, and other parameters specific to your SDR.
.- Throttle Block: Set the sample rate to match your source.
.- FFT Sink: Configure the FFT parameters (e.g., FFT size, averaging, etc.).

```
# FFT Spectrum Analyzer

from gnuradio import gr, blocks, fft, analog
from gnuradio import eng_notation
from gnuradio.fft import window
from gnuradio.filter import firdes
import osmosdr
import time

class top_block(gr.top_block):

       def __init__(self):
              gr.top_block.__init__(self, "FFT Spectrum Analyzer")

              # Variables
              ##################################################
              self.samp_rate = samp_rate = 1e6
              self.freq = freq = 100e6

              # Blocks
              ##################################################
              self.rtlsdr_source_0 = osmosdr.source(args="numchan=1")
              self.rtlsdr_source_0.set_sample_rate(samp_rate)
              self.rtlsdr_source_0.set_center_freq(freq, 0)
              self.rtlsdr_source_0.set_freq_corr(0, 0)
              self.rtlsdr_source_0.set_dc_offset_mode(0, 0)
              self.rtlsdr_source_0.set_iq_balance_mode(0, 0)
              self.rtlsdr_source_0.set_gain_mode(False, 0)
              self.rtlsdr_source_0.set_gain(10, 0)
              self.rtlsdr_source_0.set_if_gain(20, 0)
              self.rtlsdr_source_0.set_bb_gain(20, 0)
              self.rtlsdr_source_0.set_antenna("", 0)
              self.rtlsdr_source_0.set_bandwidth(0, 0)

              self.throttle_0 = blocks.throttle(gr.sizeof_gr_complex*1, samp_rate,True)
              self.fft_vxx_0 = fft.fft_vcc(1024, True, (window.blackmanharris(1024)), True, 1)
              self.stream_to_vector_0 = blocks.stream_to_vector(gr.sizeof_gr_complex*1, 1024)
              self.complex_to_mag_squared_0 = blocks.complex_to_mag_squared(1024)
              self.sink = blocks.vector_sink_f()

              # Connections
              ##################################################
              self.connect((self.rtlsdr_source_0, 0), (self.throttle_0, 0))
              self.connect((self.throttle_0, 0), (self.stream_to_vector_0, 0))
              self.connect((self.stream_to_vector_0, 0), (self.fft_vxx_0, 0))
              self.connect((self.fft_vxx_0, 0), (self.complex_to_mag_squared_0, 0))
              self.connect((self.complex_to_mag_squared_0, 0), (self.sink, 0))

  if __name__ == '__main__':
       tb = top_block()
       tb.start()
       try:
                input('Press Enter to stop:')
       except EOFError:
                pass
       tb.stop()
       tb.wait()
```

QT GUI Frequency Sink (for spectrum visualization):
.- Center Frequency: Match the center frequency of the Osmocom Source
.- Bandwidth: Match the sample rate of the Osmocom Source

```
#  GNU FFT analyzer with  spectrum visualization

import sys
from gnuradio import gr, blocks, filter, analog
from gnuradio import qtgui
from gnuradio import osmosdr
from gnuradio.qtgui import Range, RangeWidget
from PyQt5 import Qt
import sip

class spectrum_analyzer(gr.top_block, Qt.QWidget):
       def __init__(self):
              #  visualization
              ##################################################
              gr.top_block.__init__(self, "Spectrum Analyzer")
              Qt.QWidget.__init__(self)
              self.setWindowTitle("Spectrum Analyzer")
              qtgui.util.check_set_qss()
              self.top_scroll_layout = Qt.QVBoxLayout()
              self.setLayout(self.top_scroll_layout)
              self.top_scroll = Qt.QScrollArea()
              self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
              self.top_scroll_layout.addWidget(self.top_scroll)
              self.top_scroll.setWidgetResizable(True)
              self.top_widget = Qt.QWidget()
              self.top_scroll.setWidget(self.top_widget)
              self.top_layout = Qt.QVBoxLayout(self.top_widget)
              self.top_grid_layout = Qt.QGridLayout()
              self.top_layout.addLayout(self.top_grid_layout)

              self.settings = Qt.QSettings("GNU Radio", "spectrum_analyzer")

              # Variables
              ##################################################
              self.samp_rate = 2e6
              self.center_freq = 100e6

              # Blocks
              ##################################################
              self.osmosdr_source = osmosdr.source(args="numchan=1 rtl=0")
              self.osmosdr_source.set_sample_rate(self.samp_rate)
              self.osmosdr_source.set_center_freq(self.center_freq, 0)
              self.osmosdr_source.set_freq_corr(0, 0)
              self.osmosdr_source.set_dc_offset_mode(0, 0)
              self.osmosdr_source.set_iq_balance_mode(0, 0)
              self.osmosdr_source.set_gain_mode(False, 0)
              self.osmosdr_source.set_gain(10, 0)
              self.osmosdr_source.set_if_gain(20, 0)
              self.osmosdr_source.set_bb_gain(20, 0)
              self.osmosdr_source.set_antenna("", 0)
              self.osmosdr_source.set_bandwidth(0, 0)

              self.qtgui_freq_sink_x = qtgui.freq_sink_c(
                     1024, #size
                      filter.firdes.WIN_BLACKMAN_hARRIS, #wintype
                      self.center_freq, #fc
                      self.samp_rate, #bw
                      "", #name
                      1 #number of inputs
              )
              self.qtgui_freq_sink_x.set_update_time(0.10)
```

```
            self.qtgui_freq_sink_x.set_y_axis(-140, 10)
            self.qtgui_freq_sink_x.set_y_label('Relative Gain', 'dB')
            self.qtgui_freq_sink_x.set_trigger_mode(qtgui.TRIG_MODE_FREE, 0.0, 0, "")
            self.qtgui_freq_sink_x.enable_autoscale(False)
            self.qtgui_freq_sink_x.enable_grid(False)
            self.qtgui_freq_sink_x.set_fft_average(1.0)
            self.qtgui_freq_sink_x.enable_axis_labels(True)
            self.qtgui_freq_sink_x.enable_control_panel(False)

            self._qtgui_freq_sink_x_win = sip.wrapinstance(self.qtgui_freq_sink_x.qwidget(),
                                                           Qt.QWidget)
            self.top_layout.addWidget(self._qtgui_freq_sink_x_win)

            self.connect((self.osmosdr_source, 0), (self.qtgui_freq_sink_x, 0))

if __name__ == '__main__':
    qapp = Qt.QApplication(sys.argv)
    tb = spectrum_analyzer()
    tb.start()
    tb.show()
    def quitting():
        tb.stop()
        tb.wait()
    qapp.connect(qapp, Qt.SIGNAL("aboutToQuit()"), quitting)
    qapp.exec_()
```

[✓] Spectrum-Analyzer

Git-hub spectrum-analyzer •
PyQtGraph based GUI for soapy power •
Fast Power Spectrum Sensing •

PSD Estimation

Multi-taper Method of PSD: Uses multiple orthogonal tapers to reduce spectral leakage and variance.

```
import numpy as np
from spectrum import pmtm
import matplotlib.pyplot as plt

# Generate a sample signal
samp_rate = 2e6
center_freq = 100e6
duration = 1  # 1 second
t = np.arange(0, duration, 1/samp_rate)
signal = np.sin(2 * np.pi * 100e3 * t) + 0.5 * np.sin(2 * np.pi * 200e3 * t)

# Apply Multitaper Method
[eigenvalues, eigenvectors] = pmtm(signal, NW=4, show=False)
psd = np.abs(eigenvectors) ** 2

# Average the PSD estimates from all tapers
avg_psd = np.mean(psd, axis=1)
freqs = np.fft.fftfreq(len(avg_psd), 1/samp_rate)

# Plot the results
plt.figure()
plt.plot(freqs, 10 * np.log10(avg_psd))
plt.xlabel('Frequency (Hz)')
plt.ylabel('Power/Frequency (dB/Hz)')
plt.title('Multitaper Power Spectral Density')
plt.show()
```

Welch's Method:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import welch
from gnuradio import gr, blocks, analog, filter
import osmosdr

class enhanced_spectral_estimation(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self, "Enhanced Spectral Estimation")

        # Parameters
        self.samp_rate = 2e6
        self.center_freq = 100e6

        # Blocks
        self.src = osmosdr.source(args="numchan=1")
        self.src.set_sample_rate(self.samp_rate)
        self.src.set_center_freq(self.center_freq)
        self.src.set_gain(30)

        # Capture data
        self.probe = blocks.probe_signal_vc()

        # Connections
        self.connect(self.src, self.probe)

    def capture_data(self, duration=5):
        self.start()
        time.sleep(duration)
        self.stop()
        self.wait()
```

```
        captured_data = self.probe.level()
        return np.array(captured_data)

    def estimate_spectrum(self, data):
        # Welch's method
        freqs, psd = welch(data, fs=self.samp_rate, window='hamming', nperseg=1024, noverlap=
                                                                    512, scaling='density')
        return freqs, psd

    def plot_spectrum(self, freqs, psd):
        plt.figure()
        plt.semilogy(freqs, psd)
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Power Spectral Density (dB/Hz)')
        plt.title('Enhanced Power Spectral Density using Welch\'s Method')
        plt.grid(True)
        plt.show()

if __name__ == '__main__':
    tb = enhanced_spectral_estimation()
    data = tb.capture_data()
    freqs, psd = tb.estimate_spectrum(data)
    tb.plot_spectrum(freqs, psd)
```

Wavelet-based PSD estimation with denoising:

```
import numpy as np
import pywt
from gnuradio import gr, blocks, analog, filter, osmosdr
import matplotlib.pyplot as plt

class wavelet_spectral_estimation(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self, "Wavelet Spectral Estimation in Noisy Channel")

        # Parameters
        samp_rate = 2e6
        center_freq = 100e6

        # SDR source block
        self.src = osmosdr.source(args="numchan=1")
        self.src.set_sample_rate(samp_rate)
        self.src.set_center_freq(center_freq)
        self.src.set_gain(30)

        # Probe to capture data
        self.probe = blocks.probe_signal_vc()

        # Connections
        self.connect(self.src, self.probe)

    def capture_data(self, duration=5):
        self.start()
        time.sleep(duration)
        self.stop()
        self.wait()
        captured_data = self.probe.level()
        return captured_data

    def wavelet_transform(self, data, wavelet='db1', level=4):
        coeffs = pywt.wavedec(data, wavelet, level=level)
        return coeffs

    def thresholding(self, coeffs, threshold=0.2):
```

```
        return [pywt.threshold(c, threshold * max(c)) for c in coeffs]

    def reconstruct_signal(self, coeffs, wavelet='db1'):
        return pywt.waverec(coeffs, wavelet)

    def estimate_spectrum(self, data):
        freqs, psd = plt.psd(data, NFFT=1024, Fs=self.samp_rate)
        return freqs, psd

    def plot_spectrum(self, freqs, psd):
        plt.figure()
        plt.plot(freqs, 10 * np.log10(psd))
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Power/Frequency (dB/Hz)')
        plt.title('Wavelet Spectral Estimation in Noisy Channel')
        plt.show()

if __name__ == '__main__':
    tb = wavelet_spectral_estimation()
    data = tb.capture_data()
    coeffs = tb.wavelet_transform(data);   thresholded_coeffs = tb.thresholding(coeffs)
    reconstructed_signal = tb.reconstruct_signal(thresholded_coeffs)
    freqs, psd = tb.estimate_spectrum(reconstructed_signal)
    tb.plot_spectrum(freqs, psd)
```

1. VHF Frequency Scanning

    1. Osmocom Source: To interface with your RTL-SDR hardware.
    2. Frequency Sink: To visualize the frequency spectrum.
    3. Frequency Selector: To scan through different frequencies.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sys
from gnuradio import gr, blocks, analog, qtgui, osmosdr
from PyQt5 import Qt
import sip
import numpy as np
import time

class vhf_scanner(gr.top_block, Qt.QWidget):
    def __init__(self):
        gr.top_block.__init__(self, "VHF Scanner")
        Qt.QWidget.__init__(self)
        self.setWindowTitle("VHF Scanner")
        qtgui.util.check_set_qss()
        self.top_scroll_layout = Qt.QVBoxLayout()
        self.setLayout(self.top_scroll_layout)
        self.top_scroll = Qt.QScrollArea()
        self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
        self.top_scroll_layout.addWidget(self.top_scroll)
        self.top_scroll.setWidgetResizable(True)
        self.top_widget = Qt.QWidget()
        self.top_scroll.setWidget(self.top_widget)
        self.top_layout = Qt.QVBoxLayout(self.top_widget)
        self.top_grid_layout = Qt.QGridLayout()
        self.top_layout.addLayout(self.top_grid_layout)

        self.settings = Qt.QSettings("GNU Radio", "vhf_scanner")

        self.samp_rate = 2e6
        self.start_freq = 88e6
        self.stop_freq = 108e6
        self.step_freq = 1e6
        self.dwell_time = 1
        self.center_freq = self.start_freq

        # Blocks
        ####################
        # Osmocom Source
        ####################
        self.osmosdr_source = osmosdr.source(args="numchan=1 rtl=0")
         self.osmosdr_source.set_sample_rate(self.samp_rate)
        self.osmosdr_source.set_center_freq(self.center_freq, 0)
        self.osmosdr_source.set_freq_corr(0, 0)
        self.osmosdr_source.set_dc_offset_mode(0, 0)
        self.osmosdr_source.set_iq_balance_mode(0, 0)
        self.osmosdr_source.set_gain_mode(False, 0)
        self.osmosdr_source.set_gain(10, 0)
        self.osmosdr_source.set_if_gain(20, 0)
        self.osmosdr_source.set_bb_gain(20, 0)
        self.osmosdr_source.set_antenna("", 0)
        self.osmosdr_source.set_bandwidth(0, 0)

        self.throttle = blocks.throttle(gr.sizeof_gr_complex*1, self.samp_rate,True)


        # Frequency Sink
        ####################
```

```
        self.qtgui_freq_sink_x = qtgui.freq_sink_c(
            1024,  # FFT size
            qtgui.WIN_BLACKMAN_hARRIS,  # Window function
            self.center_freq,  # Center frequency
            self.samp_rate,  # Sample rate
            "Frequency Spectrum",  # Name
            1  # Number of inputs
        )
        self.qtgui_freq_sink_x.set_update_time(0.10)
        self.qtgui_freq_sink_x.set_y_axis(-140, 10)
        self.qtgui_freq_sink_x.set_y_label('Relative Gain', 'dB')
        self.qtgui_freq_sink_x.enable_autoscale(False)
        self.qtgui_freq_sink_x.enable_grid(False)
        self.qtgui_freq_sink_x.set_fft_average(1.0)
        self.qtgui_freq_sink_x.enable_axis_labels(True)
        self.qtgui_freq_sink_x.enable_control_panel(False)

        self._qtgui_freq_sink_x_win = sip.wrapinstance(self.qtgui_freq_sink_x.qwidget(), Qt.
                                                       QWidget)
        self.top_layout.addWidget(self._qtgui_freq_sink_x_win)

        # Connections
        self.connect((self.osmosdr_source, 0), (self.throttle, 0), (self.qtgui_freq_sink_x, 0))

        # Frequency Scanner Logic
        self.freq_scanner()

        def freq_scanner(self):
            import threading
            def scan():
                while True:
                    for freq in np.arange(self.start_freq, self.stop_freq, self.step_freq):
                        self.center_freq = freq
                        self.osmosdr_source.set_center_freq(self.center_freq, 0)
                        time.sleep(self.dwell_time)
        thread = threading.Thread(target=scan)
        thread.daemon = True
        thread.start()

if __name__ == '__main__':
    qapp = Qt.QApplication(sys.argv)
    tb = vhf_scanner()
    tb.start()
    tb.show()
    def quitting():
        tb.stop()
        tb.wait()
    qapp.connect(qapp, Qt.SIGNAL("aboutToQuit()"), quitting)
    qapp.exec_()
```

[✓] VHF scanning

AM and NFM scanner with multiple equally spaced channels •
GNU Radio Schematic Marine VHF Channel Scanner •

Two-antenna diversity by selection diversity

1. Osmocom Source: To interface with your dual-channel SDR hardware.
2. Complex to Magnitude Squared: To calculate the power of the signals from each antenna.
3. Max Selector: To select the signal with the highest power.
4. Frequency Sink: For visualizing the frequency spectrum of the selected signal.
5. Stream Selector: To select the appropriate signal stream.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sys
from gnuradio import gr, blocks, analog, qtgui, osmosdr
from PyQt5 import Qt
import sip

class max_selector(gr.sync_block):
def __init__(self):
gr.sync_block.__init__(self,
name="max_selector",
in_sig=[(gr.sizeof_gr_complex, 2)],
out_sig=[gr.sizeof_gr_complex])

def work(self, input_items, output_items):
in0 = input_items[0][:, 0]
in1 = input_items[0][:, 1]
out = output_items[0]

power0 = abs(in0) ** 2
power1 = abs(in1) ** 2

if power0.mean() > power1.mean():
out[:] = in0
else:
out[:] = in1

return len(output_items[0])

class dual_channel_sdr(gr.top_block, Qt.QWidget):
def __init__(self):
gr.top_block.__init__(self, "Dual Channel SDR")
Qt.QWidget.__init__(self)
self.setWindowTitle("Dual Channel SDR")
qtgui.util.check_set_qss()
self.top_scroll_layout = Qt.QVBoxLayout()
self.setLayout(self.top_scroll_layout)
self.top_scroll = Qt.QScrollArea()
self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
self.top_scroll_layout.addWidget(self.top_scroll)
self.top_scroll.setWidgetResizable(True)
self.top_widget = Qt.QWidget()
self.top_scroll.setWidget(self.top_widget)
self.top_layout = Qt.QVBoxLayout(self.top_widget)
self.top_grid_layout = Qt.QGridLayout()
self.top_layout.addLayout(self.top_grid_layout)

self.settings = Qt.QSettings("GNU Radio", "dual_channel_sdr")

self.samp_rate = 2e6
self.center_freq = 100e6

# Blocks
self.osmosdr_source = osmosdr.source(args="numchan=2,rtl=0")
self.osmosdr_source.set_sample_rate(self.samp_rate)
self.osmosdr_source.set_center_freq(self.center_freq, 0)
```

```
self.osmosdr_source.set_center_freq(self.center_freq, 1)
self.osmosdr_source.set_freq_corr(0, 0)
self.osmosdr_source.set_freq_corr(0, 1)
self.osmosdr_source.set_dc_offset_mode(0, 0)
self.osmosdr_source.set_dc_offset_mode(0, 1)
self.osmosdr_source.set_iq_balance_mode(0, 0)
self.osmosdr_source.set_iq_balance_mode(0, 1)
self.osmosdr_source.set_gain_mode(False, 0)
self.osmosdr_source.set_gain_mode(False, 1)
self.osmosdr_source.set_gain(10, 0)
self.osmosdr_source.set_gain(10, 1)
self.osmosdr_source.set_if_gain(20, 0)
self.osmosdr_source.set_if_gain(20, 1)
self.osmosdr_source.set_bb_gain(20, 0)
self.osmosdr_source.set_bb_gain(20, 1)
self.osmosdr_source.set_antenna("", 0)
self.osmosdr_source.set_antenna("", 1)
self.osmosdr_source.set_bandwidth(0, 0)
self.osmosdr_source.set_bandwidth(0, 1)

self.complex_to_mag_sq_0 = blocks.complex_to_mag_squared(1)
self.complex_to_mag_sq_1 = blocks.complex_to_mag_squared(1)

self.max_selector = max_selector()

self.qtgui_freq_sink_x_0 = qtgui.freq_sink_c(
1024,  # FFT size
qtgui.WIN_BLACKMAN_hARRIS,  # Window function
self.center_freq,  # Center frequency
self.samp_rate,  # Sample rate
"Frequency Spectrum",  # Name
1  # Number of inputs
)
self.qtgui_freq_sink_x_0.set_update_time(0.10)
self.qtgui_freq_sink_x_0.set_y_axis(-140, 10)
self.qtgui_freq_sink_x_0.set_y_label('Relative Gain', 'dB')
self.qtgui_freq_sink_x_0.enable_autoscale(False)
self.qtgui_freq_sink_x_0.enable_grid(False)
self.qtgui_freq_sink_x_0.set_fft_average(1.0)
self.qtgui_freq_sink_x_0.enable_axis_labels(True)
self.qtgui_freq_sink_x_0.enable_control_panel(False)

self._qtgui_freq_sink_x_0_win = sip.wrapinstance(self.qtgui_freq_sink_x_0.qwidget(), Qt.QWidget
                                                 )
self.top_layout.addWidget(self._qtgui_freq_sink_x_0_win)

# Connections
self.connect((self.osmosdr_source, 0), (self.complex_to_mag_sq_0, 0))
self.connect((self.osmosdr_source, 1), (self.complex_to_mag_sq_1, 0
```

Two-antenna diversity by equal gain combining,

.- Two SDR Source blocks (e.g., RTL-SDR Source or USRP Source) for the two antennas.
.- Two Complex to MagPhase blocks to extract magnitude and phase.
.- Add, Multiply, and Complex Multiply blocks to align phases and combine signals.
.- A Sink block (e.g., FFT Sink, QT GUI Sink) to visualize the combined signal.

```
from gnuradio import gr, blocks, analog, filter
from gnuradio import uhd
import numpy as np

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

        # SDR Source blocks
        self.src0 = uhd.usrp_source(
            ",".join(("addr0", "")),
            uhd.stream_args(
                cpu_format="fc32",
                channels=range(1),
            ),
        )
        self.src1 = uhd.usrp_source(
            ",".join(("addr1", "")),
            uhd.stream_args(
                cpu_format="fc32",
                channels=range(1),
            ),
        )

        # Complex to MagPhase blocks
        self.mag_phase0 = blocks.complex_to_magphase()
        self.mag_phase1 = blocks.complex_to_magphase()

        # Phase adjustment
        self.phase_diff = blocks.sub_ff()
        self.adjusted_phase1 = blocks.multiply_cc()

        # Combine signals with equal gain
        self.combined_signal = blocks.add_cc()

       # Connect blocks
        self.connect(self.src0, self.mag_phase0)
        self.connect(self.src1, self.mag_phase1)
        self.connect((self.mag_phase0, 1), (self.phase_diff, 0))
        self.connect((self.mag_phase1, 1), (self.phase_diff, 1))
        self.connect(self.src1, self.adjusted_phase1)
        self.connect((self.phase_diff, 0), (self.adjusted_phase1, 1))
        self.connect(self.src0, (self.combined_signal, 0))
        self.connect(self.adjusted_phase1, (self.combined_signal, 1))

        # Output Sink
        self.sink = blocks.null_sink(gr.sizeof_gr_complex)
        self.connect(self.combined_signal, self.sink)

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

Two-antenna diversity by maximal ratio combining.

.- Setup SDR Receivers: Configure two SDR devices to receive signals simultaneously.
.- Calculate Weights: Calculate the optimal weights based on the received signal strengths.
.- Maximal Ratio Combining: Combine the signals using the calculated weights.

```
from gnuradio import gr, blocks, analog, filter
from gnuradio import uhd
import numpy as np

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

        # SDR Source blocks
        self.src0 = uhd.usrp_source(
            ",".join(("addr0", "")),
            uhd.stream_args(
                cpu_format="fc32",
                channels=range(1),
            ),
        )
        self.src1 = uhd.usrp_source(
            ",".join(("addr1", "")),
            uhd.stream_args(
                cpu_format="fc32",
               channels=range(1),
            ),
        )

        # Complex to MagPhase blocks
        self.mag_phase0 = blocks.complex_to_magphase()
        self.mag_phase1 = blocks.complex_to_magphase()

        # Magnitude Squared for SNR weighting
        self.mag_squared0 = blocks.multiply_ff()
        self.mag_squared1 = blocks.multiply_ff()

        # Multiplier blocks for weighting
        self.weighted0 = blocks.multiply_cc()
        self.weighted1 = blocks.multiply_cc()

        # Combine signals with maximal ratio combining
        self.combined_signal = blocks.add_cc()

        # Connections
        self.connect(self.src0, self.mag_phase0)
        self.connect(self.src1, self.mag_phase1)

        self.connect((self.mag_phase0, 0), (self.mag_squared0, 0))
        self.connect((self.mag_phase0, 0), (self.mag_squared0, 1))
        self.connect((self.mag_phase1, 0), (self.mag_squared1, 0))
        self.connect((self.mag_phase1, 0), (self.mag_squared1, 1))

        self.connect(self.src0, (self.weighted0, 0))
        self.connect(self.src1, (self.weighted1, 0))
        self.connect((self.mag_squared0, 0), (self.weighted0, 1))
        self.connect((self.mag_squared1, 0), (self.weighted1, 1))

        self.connect(self.weighted0, (self.combined_signal, 0))
        self.connect(self.weighted1, (self.combined_signal, 1))

        # Output Sink
        self.sink = blocks.null_sink(gr.sizeof_gr_complex)
        self.connect(self.combined_signal, self.sink)
```

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

2. Spectrum Parameter Estimation

   FM power strength

   - Source Block: *'osmosdr.source'* or equivalent.
   - Frequency Translation: Use a frequency translating filter to focus on the desired signal.
   - Demodulation: *'analog.wfm_rcv'* for wideband FM demodulation.
   - Power Calculation: Use a probe signal block to measure power.

```
from gnuradio import gr, blocks, analog, filter, audio, osmosdr
import numpy as np

class fm_power_example(gr.top_block):
     def __init__(self):
         gr.top_block.__init__(self, "FM Power Example")

         # Parameters
         samp_rate = 2e6
         center_freq = 100e6 # Example frequency
         audio_rate = 48e3

         # Blocks
         self.src = osmosdr.source(args="numchan=1")
         self.src.set_sample_rate(samp_rate)
         self.src.set_center_freq(center_freq)
         self.src.set_gain(30)

         self.chan_filt = filter.freq_xlating_fir_filter_ccf(1, (1,), 0, samp_rate)
         self.fm_demod = analog.wfm_rcv(quad_rate=samp_rate, audio_decimation=10)
         self.audio_sink = audio.sink(int(audio_rate), '', True)
         self.throttle = blocks.throttle(gr.sizeof_gr_complex*1, samp_rate, True)
         self.probe_signal = blocks.probe_signal_c()

         # Connections
         self.connect(self.src, self.chan_filt, self.fm_demod, self.probe_signal)
         self.connect(self.fm_demod, self.audio_sink)

         def get_power(self):
             signal_level = self.probe_signal.level()
             power_dbm = 10 * np.log10(signal_level)
             return power_dbm

  if __name__ == '__main__':
         tb = fm_power_example()
         tb.start()
         tb.wait()
         print("Power (dBm):", tb.get_power())
```

Signal-to-Noise ratio – S/N:

```
from gnuradio import gr
from gnuradio import blocks
from gnuradio import analog
from gnuradio import filter
from gnuradio import audio
from gnuradio import qtgui

class fm_snr_example(gr.top_block):

  def __init__(self):
    gr.top_block.__init__(self, "FM SNR Example")

    # Parameters
    samp_rate = 2e6
    center_freq = 100e6 # Example FM station frequency
    audio_rate = 48e3

    # Blocks
    self.src = osmosdr.source(args="numchan=1")
    self.src.set_sample_rate(samp_rate)
    self.src.set_center_freq(center_freq)
    self.src.set_gain(30)

    self.chan_filt = filter.freq_xlating_fir_filter_ccf(1, (1,), 0, samp_rate)

    self.fm_demod = analog.wfm_rcv(quad_rate=samp_rate, audio_decimation=10)

    self.audio_sink = audio.sink(int(audio_rate), '', True)

    self.throttle = blocks.throttle(gr.sizeof_gr_complex*1, samp_rate, True)

    self.probe_signal = blocks.probe_signal_f()

    # Connections
    self.connect(self.src, self.chan_filt, self.fm_demod, self.audio_sink)
    self.connect(self.fm_demod, self.probe_signal)

  def get_snr(self):
    signal_power = self.probe_signal.level()
    noise_power = 1 # Placeholder, actual noise calculation needed
    snr = 10 * log10(signal_power / noise_power)
    return snr

if __name__ == '__main__':
  tb = fm_snr_example()
  tb.start()
  tb.wait()
  print("SNR:", tb.get_snr())
```

3. Thresholding

Detección de canal: Valor reportado: presencia – 1 , ausencia – 0 por canal

```
from gnuradio import gr, blocks, analog, filter
import numpy as np

class threshold_energy_detection(gr.top_block):
  def __init__(self):
    gr.top_block.__init__(self, "Threshold Energy Detection")

    # Parameters
    samp_rate = 2e6
    center_freq = 100e6
    threshold = 0.5 # Example threshold

    # Blocks
    self.src = osmosdr.source(args="numchan=1")
    self.src.set_sample_rate(samp_rate)
    self.src.set_center_freq(center_freq)
    self.src.set_gain(30)

    self.chan_filt = filter.freq_xlating_fir_filter_ccf(1, (1,), 0, samp_rate)
    self.fm_demod = analog.wfm_rcv(quad_rate=samp_rate, audio_decimation=10)
    self.probe_signal = blocks.probe_signal_f()

    # Connections
    self.connect(self.src, self.chan_filt, self.fm_demod, self.probe_signal)

  def detect_signal(self):
    signal_energy = self.probe_signal.level()
    if signal_energy > threshold:
      print("Signal Detected")
    else:
      print("No Signal Detected")

if __name__ == '__main__':
  tb = threshold_energy_detection()
  tb.start()
  tb.wait()
  tb.detect_signal()
```

- ✓ Energy Detection (L_2-based estimation) and Entropy-based detection

  Energy detection-based spectrum sensing machine •
- ✓ Shape signal Detection : correlation, cyclostationarity, covariance, waveform-based
- ✓ Matched Filter (L_2-based filtering)
- ✓ Matrix decomposition-based (eigenvalue detection)

  Maximizing Eigenvalue Using Machine Learning •

  FlashFFTConv •

  Energy detection under noise power •

  Multiscale Wavelet Transform Extremum Detection With the Spectrum Energy Detection •

  Exploring DL for Adaptive Energy Detection Threshold Determination: A Multistage Approach •Deep Learning for Adaptive Energy Detection Threshold •
- ✓ rtlsdr-scanner, software defined radio frequency scanner que utiliza un chip Realtek RTL2832u para convertir las senales de radio analógicas en digitales ✓ Jason • Raspberry Radio Scanner •