```
# ==================== START OF FILE: gr-analog/examples/fm_demod.py
====================
#!/usr/bin/env python
#
# Copyright 2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import blocks
from gnuradio import filter
from gnuradio import analog
from gnuradio import audio
from gnuradio.filter import firdes
from gnuradio.fft import window
import sys
import math

# Create a top_block


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

        input_rate = 400e3     # rate of a broadcast FM station
        audio_rate = 44.1e3    # Rate we send the signal to the speaker

        # resample from the output of the demodulator to the rate of
        # the audio sink.
        resamp_rate = audio_rate / input_rate

        # use a file as a dummy source. Replace this with a real radio
        # receiver to capture signals over-the-air.
        src = blocks.file_source(gr.sizeof_gr_complex, "dummy.dat", True)

        # Set the demodulator using the same deviation as the receiver.
        max_dev = 75e3
        fm_demod_gain = input_rate / (2 * math.pi * max_dev)
        fm_demod = analog.quadrature_demod_cf(fm_demod_gain)
        # Create a filter for the resampler and filter the audio
        # signal to 15 kHz. The nfilts is the number of filters in the
        # arbitrary resampler. It logically operates at a rate of
        # nfilts*input_rate, so we make those adjustments when
        # building the filter.
        volume = 0.20
        nfilts = 32
        resamp_taps = firdes.low_pass_2(volume * nfilts,        # gain
                                        nfilts * input_rate,    # sampling rate
                                        15e3,                   # low pass cutoff freq
                                        1e3,                    # width of trans. band
                                        60,                             # stop band
attenuaton
                                        window.WIN_KAISER)

        # Build the resampler and filter
        resamp_filter = filter.pfb_arb_resampler_fff(resamp_rate,
                                                     resamp_taps, nfilts)

        # sound card as final sink You may have to add a specific
        # device name as a second argument here, something like
        # "pulse" if using pulse audio or "plughw:0,0".
        audio_sink = audio.sink(int(audio_rate))

        # now wire it all together
        self.connect(src, fm_demod)
        self.connect(fm_demod, resamp_filter)
        self.connect(resamp_filter, (audio_sink, 0))


def main(args):
    tb = build_graph()
    tb.start()        # fork thread and return
    input('Press Enter to quit: ')
    tb.stop()


if __name__ == '__main__':
    main(sys.argv[1:])

#   ====================   END   OF   FILE:   gr-analog/examples/fm_demod.py
====================
# ==================== START OF FILE: gr-analog/examples/fmtest.py
====================
#!/usr/bin/env python
#
# Copyright 2009,2012,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import blocks
from gnuradio import filter
from gnuradio.fft import window
from gnuradio import analog
from gnuradio import channels
import sys
import math
import time
import numpy

try:
    import pylab
except ImportError:
    print("Error: Program requires matplotlib (see: matplotlib.sourceforge.net).")
    sys.exit(1)


class fmtx(gr.hier_block2):
    def __init__(self, lo_freq, audio_rate, if_rate):

        gr.hier_block2.__init__(self, "build_fm",
                                gr.io_signature(1, 1, gr.sizeof_float),
                                gr.io_signature(1, 1, gr.sizeof_gr_complex))

        fmtx = analog.nbfm_tx(audio_rate, if_rate, max_dev=5e3,
                              tau=75e-6, fh=0.925 * if_rate / 2.0)

        # Local oscillator
        lo = analog.sig_source_c(if_rate,            # sample rate
                                 analog.GR_SIN_WAVE, # waveform type
```

```
                                        lo_freq,                # frequency
                                        1.0,                    # amplitude
                                        0)                      # DC Offset
        mixer = blocks.multiply_cc()

        self.connect(self, fmtx, (mixer, 0))
        self.connect(lo, (mixer, 1))
        self.connect(mixer, self)


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

        self._nsamples = 1000000
        self._audio_rate = 8000

        # Set up N channels with their own baseband and IF frequencies
        self._N = 5
        chspacing = 16000
        freq = [10, 20, 30, 40, 50]
        f_lo = [0, 1 * chspacing, -1 * chspacing,
                2 * chspacing, -2 * chspacing]

        self._if_rate = 4 * self._N * self._audio_rate

        # Create a signal source and frequency modulate it
        self.sum = blocks.add_cc()
        for n in range(self._N):
            sig = analog.sig_source_f(
                self._audio_rate, analog.GR_SIN_WAVE, freq[n], 0.5)
            fm = fmtx(f_lo[n], self._audio_rate, self._if_rate)
            self.connect(sig, fm)
            self.connect(fm, (self.sum, n))

        self.head = blocks.head(gr.sizeof_gr_complex, self._nsamples)
        self.snk_tx = blocks.vector_sink_c()
        self.channel = channels.channel_model(0.1)

        self.connect(self.sum, self.head, self.channel, self.snk_tx)

        # Design the channlizer
        self._M = 10
        bw = chspacing / 2.0
        t_bw = chspacing / 10.0
        self._chan_rate = self._if_rate / self._M
        self._taps = filter.firdes.low_pass_2(1, self._if_rate, bw, t_bw,
                                              attenuation_dB=100,

window=window.WIN_BLACKMAN_hARRIS)
        tpc = math.ceil(float(len(self._taps)) / float(self._M))

        print("Number of taps:     ", len(self._taps))
        print("Number of channels: ", self._M)
        print("Taps per channel:   ", tpc)

        self.pfb = filter.pfb.channelizer_ccf(self._M, self._taps)

        self.connect(self.channel, self.pfb)

        # Create a file sink for each of M output channels of the filter and connect it
        self.fmdet = list()
        self.squelch = list()
        self.snks = list()
        for i in range(self._M):
            self.fmdet.append(analog.nbfm_rx(
                self._audio_rate, self._chan_rate))
            self.squelch.append(analog.standard_squelch(self._audio_rate * 10))
            self.snks.append(blocks.vector_sink_f())
            self.connect(
                (self.pfb, i), self.fmdet[i], self.squelch[i], self.snks[i])

    def num_tx_channels(self):
        return self._N

    def num_rx_channels(self):
        return self._M


def main():

    fm = fmtest()

    tstart = time.time()
    fm.run()
    tend = time.time()

    if 1:
```

```
fig1 = pylab.figure(1, figsize=(12, 10), facecolor="w")
fig2 = pylab.figure(2, figsize=(12, 10), facecolor="w")
fig3 = pylab.figure(3, figsize=(12, 10), facecolor="w")

Ns = 10000
Ne = 100000

fftlen = 8192
winfunc = numpy.blackman

# Plot transmitted signal
fs = fm._if_rate

d = fm.snk_tx.data()[Ns:Ns + Ne]
sp1_f = fig1.add_subplot(2, 1, 1)

X, freq = sp1_f.psd(d, NFFT=fftlen, noverlap=fftlen // 4, Fs=fs,
                    window=lambda d: d * winfunc(fftlen),
                    visible=False)
X_in = 10.0 * numpy.log10(abs(numpy.fft.fftshift(X)))
f_in = numpy.arange(-fs / 2.0, fs / 2.0, fs / float(X_in.size))
p1_f = sp1_f.plot(f_in, X_in, "b")
sp1_f.set_xlim([min(f_in), max(f_in) + 1])
sp1_f.set_ylim([-120.0, 20.0])

sp1_f.set_title("Input Signal", weight="bold")
sp1_f.set_xlabel("Frequency (Hz)")
sp1_f.set_ylabel("Power (dBW)")

Ts = 1.0 / fs
Tmax = len(d) * Ts

t_in = numpy.arange(0, Tmax, Ts)
x_in = numpy.array(d)
sp1_t = fig1.add_subplot(2, 1, 2)
p1_t = sp1_t.plot(t_in, x_in.real, "b-o")
#p1_t = sp1_t.plot(t_in, x_in.imag, "r-o")
sp1_t.set_ylim([-5, 5])

# Set up the number of rows and columns for plotting the subfigures
Ncols = int(numpy.floor(numpy.sqrt(fm.num_rx_channels())))
Nrows = int(numpy.floor(fm.num_rx_channels() / Ncols))
if(fm.num_rx_channels() % Ncols != 0):
    Nrows += 1
```

```
# Plot each of the channels outputs. Frequencies on Figure 2 and
# time signals on Figure 3
fs_o = fm._audio_rate
for i in range(len(fm.snks)):
    # remove issues with the transients at the beginning
    # also remove some corruption at the end of the stream
    #       this is a bug, probably due to the corner cases
    d = fm.snks[i].data()[Ns:Ne]

    sp2_f = fig2.add_subplot(Nrows, Ncols, 1 + i)
    X, freq = sp2_f.psd(d, NFFT=fftlen, noverlap=fftlen // 4, Fs=fs_o,
                        window=lambda d: d * winfunc(fftlen),
                        visible=False)
    #X_o = 10.0*numpy.log10(abs(numpy.fft.fftshift(X)))
    X_o = 10.0 * numpy.log10(abs(X))
    #f_o = numpy.arange(-fs_o/2.0, fs_o/2.0, fs_o/float(X_o.size))
    f_o = numpy.arange(0, fs_o / 2.0, fs_o / 2.0 / float(X_o.size))
    p2_f = sp2_f.plot(f_o, X_o, "b")
    sp2_f.set_xlim([min(f_o), max(f_o) + 0.1])
    sp2_f.set_ylim([-120.0, 20.0])
    sp2_f.grid(True)

    sp2_f.set_title(("Channel %d" % i), weight="bold")
    sp2_f.set_xlabel("Frequency (kHz)")
    sp2_f.set_ylabel("Power (dBW)")

    Ts = 1.0 / fs_o
    Tmax = len(d) * Ts
    t_o = numpy.arange(0, Tmax, Ts)

    x_t = numpy.array(d)
    sp2_t = fig3.add_subplot(Nrows, Ncols, 1 + i)
    p2_t = sp2_t.plot(t_o, x_t.real, "b")
    p2_t = sp2_t.plot(t_o, x_t.imag, "r")
    sp2_t.set_xlim([min(t_o), max(t_o) + 1])
    sp2_t.set_ylim([-1, 1])

    sp2_t.set_xlabel("Time (s)")
    sp2_t.set_ylabel("Amplitude")

pylab.show()
```

```
if __name__ == "__main__":
        main()

#     ====================     END     OF     FILE:     gr-analog/examples/fmtest.py
====================


#            ====================            START            OF            FILE:            gr-
analog/examples/tags/uhd_burst_detector.py ====================
#!/usr/bin/env python
#
# Copyright 2012,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import eng_notation
from gnuradio import gr
from gnuradio import filter, analog, blocks
from gnuradio import uhd
from gnuradio.fft import window
from gnuradio.eng_arg import eng_float
from gnuradio.filter import firdes
from argparse import ArgumentParser


class uhd_burst_detector(gr.top_block):
        def __init__(self, uhd_address, options):

                gr.top_block.__init__(self)

                self.uhd_addr = uhd_address
                self.freq = options.freq
                self.samp_rate = options.samp_rate
                self.gain = options.gain
                self.threshold = options.threshold
                self.trigger = options.trigger

                self.uhd_src = uhd.single_usrp_source(
                        device_addr=self.uhd_addr,
                        stream_args=uhd.stream_args('fc32'))
```

```
self.uhd_src.set_samp_rate(self.samp_rate)
self.uhd_src.set_center_freq(self.freq, 0)
self.uhd_src.set_gain(self.gain, 0)

taps = firdes.low_pass_2(1, 1, 0.4, 0.1, 60)
self.chanfilt = filter.fir_filter_ccc(10, taps)
self.tagger = blocks.burst_tagger(gr.sizeof_gr_complex)

# Dummy signaler to collect a burst on known periods
data = 1000 * [0, ] + 1000 * [1, ]
self.signal = blocks.vector_source_s(data, True)

# Energy detector to get signal burst
# use squelch to detect energy
self.det = analog.simple_squelch_cc(self.threshold, 0.01)
# convert to mag squared (float)
self.c2m = blocks.complex_to_mag_squared()
# average to debounce
self.avg = filter.single_pole_iir_filter_ff(0.01)
# rescale signal for conversion to short
self.scale = blocks.multiply_const_ff(2**16)
# signal input uses shorts
self.f2s = blocks.float_to_short()

# Use file sink burst tagger to capture bursts
self.fsnk = blocks.tagged_file_sink(
    gr.sizeof_gr_complex, self.samp_rate)

##################################################
# Connections
##################################################
self.connect((self.uhd_src, 0), (self.tagger, 0))
self.connect((self.tagger, 0), (self.fsnk, 0))

if self.trigger:
    # Connect a dummy signaler to the burst tagger
    self.connect((self.signal, 0), (self.tagger, 1))

else:
    # Connect an energy detector signaler to the burst tagger
    self.connect(self.uhd_src, self.det)
    self.connect(self.det, self.c2m, self.avg, self.scale, self.f2s)
    self.connect(self.f2s, (self.tagger, 1))
    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.uhd_src_0.set_samp_rate(self.samp_rate)


if __name__ == '__main__':
    parser = ArgumentParser()
    parser.add_argument("-a", "--address", default="addr=192.168.10.2",
                        help="select address of the device [default=%(default)r]")
    # parser.add_argument("-A", "--antenna", default=None,
    #                    help="select Rx Antenna (only on RFX-series boards)")
    parser.add_argument("-f", "--freq", type=eng_float, default=450e6,
                        help="set frequency to FREQ", metavar="FREQ")
    parser.add_argument("-g", "--gain", type=eng_float, default=0,
                        help="set gain in dB [default=%(default)r]")
    parser.add_argument("-R", "--samp-rate", type=eng_float, default=200000,
                        help="set USRP sample rate [default=%(default)r]")
    parser.add_argument("-t", "--threshold", type=float, default=-60,
                        help="Set the detection power threshold (dBm)
[default=%(default)r")
    parser.add_argument("-T", "--trigger", action="store_true", default=False,
                        help="Use internal trigger instead of detector
[default=%(default)r]")
    args = parser.parse_args()

    uhd_addr = args.address

    tb = uhd_burst_detector(uhd_addr, args)
    tb.run()

# ==================== END OF FILE: gr-
analog/examples/tags/uhd_burst_detector.py ====================


# ==================== START OF FILE: gr-audio/examples/python/audio_copy.py
====================
#!/usr/bin/env python
#
# Copyright 2004,2005,2007 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
```

```
#
#

from gnuradio import gr
from gnuradio import audio
from gnuradio.eng_arg import eng_float
from argparse import ArgumentParser


class my_top_block(gr.top_block):

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

        parser = ArgumentParser()
        parser.add_argument("-I", "--audio-input", default="",
                            help="pcm input device name.  E.g., hw:0,0 or /dev/dsp")
        parser.add_argument("-O", "--audio-output", default="",
                            help="pcm output device name.  E.g., hw:0,0 or /dev/dsp")
        parser.add_argument("-r", "--sample-rate", type=eng_float, default=48000,
                            help="set sample rate, default=%(default)s")
        args = parser.parse_args()

        sample_rate = int(args.sample_rate)
        src = audio.source(sample_rate, args.audio_input)
        dst = audio.sink(sample_rate, args.audio_output)

        # Determine the maximum number of outputs on the source and
        # maximum number of inputs on the sink, then connect together
        # the most channels we can without overlap
        nchan = min(src.output_signature().max_streams(),
                    dst.input_signature().max_streams())

        for i in range(nchan):
            self.connect((src, i), (dst, i))


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

# ==================== END OF FILE: gr-audio/examples/python/audio_copy.py
```

```
====================


# ==================== START OF FILE: gr-audio/examples/python/audio_play.py
====================
#!/usr/bin/env python
#
# Copyright 2004,2005,2007 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import audio
from gnuradio import blocks
from gnuradio.eng_arg import eng_float
from argparse import ArgumentParser


class my_top_block(gr.top_block):

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

        parser = ArgumentParser()
        parser.add_argument("-F", "--filename", default="audio.dat",
                            help="read input from FILENAME default=%(default)r")
        parser.add_argument("-r", "--sample-rate", type=eng_float, default=48000,
                            help="set sample rate (default=%(default)r)")
        parser.add_argument("-R", "--repeat", action="store_true")
        parser.add_argument("-O", "--audio-output", default="",
                            help="pcm output device name.  E.g., hw:0,0 or /dev/dsp")
        args = parser.parse_args()
        sample_rate = int(args.sample_rate)
        src = blocks.file_source(gr.sizeof_float, args.filename, args.repeat)
        dst = audio.sink(sample_rate, args.audio_output)
        self.connect(src, dst)


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

# ==================== END OF FILE: gr-audio/examples/python/audio_play.py
====================


# ==================== START OF FILE: gr-audio/examples/python/audio_to_file.py
====================
#!/usr/bin/env python
#
# Copyright 2004,2007,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import audio
from gnuradio import blocks
from gnuradio.eng_arg import eng_float
from argparse import ArgumentParser


class my_top_block(gr.top_block):

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

        parser = ArgumentParser()
        parser.add_argument("-I", "--audio-input", default="",
                            help="pcm input device name.  E.g., hw:0,0 or /dev/dsp")
        parser.add_argument("-r", "--sample-rate", type=eng_float, default=48000,
                            help="set sample rate to RATE (%(default)r)")
        parser.add_argument("-N", "--nsamples", type=eng_float,
                            help="number of samples to collect [default=+inf]")
        parser.add_argument('file_name', metavar='FILE-NAME',
                            help="Output file path")

        args = parser.parse_args()
        sample_rate = int(args.sample_rate)
        src = audio.source(sample_rate, args.audio_input)
        dst = blocks.file_sink(gr.sizeof_float, args.file_name)

        if args.nsamples is None:
            self.connect((src, 0), dst)
        else:
            head = blocks.head(gr.sizeof_float, int(args.nsamples))
            self.connect((src, 0), head, dst)


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

# ==================== END OF FILE: gr-audio/examples/python/audio_to_file.py
====================


# ==================== START OF FILE: gr-audio/examples/python/dial_tone.py
====================
#!/usr/bin/env python
#
# Copyright 2004,2005,2007,2012 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import audio
from gnuradio.eng_arg import eng_float
from argparse import ArgumentParser

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)
```

```
class my_top_block(gr.top_block):

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

        parser = ArgumentParser()
        parser.add_argument("-O", "--audio-output", default="",
                            help="pcm output device name.   E.g., hw:0,0 or /dev/dsp")
        parser.add_argument("-r", "--sample-rate", type=eng_float, default=48000,
                            help="set sample rate, default=%(default)s")
        args = parser.parse_args()

        sample_rate = int(args.sample_rate)
        ampl = 0.1

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


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

# ==================== END OF FILE: gr-audio/examples/python/dial_tone.py
====================


# ==================== START OF FILE: gr-
audio/examples/python/dial_tone_wav.py ====================
#!/usr/bin/env python
#
# Copyright 2004,2005,2007,2008,2012,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#
```

```
# GNU Radio example program to record a dial tone to a WAV file

from gnuradio import gr
from gnuradio import blocks
from gnuradio.eng_arg import eng_float
from argparse import ArgumentParser

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)


class my_top_block(gr.top_block):

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

        parser = ArgumentParser()
        parser.add_argument("-r", "--sample-rate", type=eng_float, default=48000,
                            help="set sample rate to RATE (%(default)r)")
        parser.add_argument("-N", "--samples", type=eng_float, required=True,
                            help="number of samples to record")
        parser.add_argument('file_name', metavar='WAV-FILE',
                            help='Output WAV file name', nargs=1)
        args = parser.parse_args()

        sample_rate = int(args.sample_rate)
        ampl = 0.1

        src0 = analog.sig_source_f(sample_rate, analog.GR_SIN_WAVE, 350, ampl)
        src1 = analog.sig_source_f(sample_rate, analog.GR_SIN_WAVE, 440, ampl)
        head0 = blocks.head(gr.sizeof_float, int(args.samples))
        head1 = blocks.head(gr.sizeof_float, int(args.samples))
        dst = blocks.wavfile_sink(args.file_name[0], 2, int(args.sample_rate),
                                  blocks.FORMAT_WAV, blocks.FORMAT_PCM_16)

        self.connect(src0, head0, (dst, 0))
        self.connect(src1, head1, (dst, 1))


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

# ==================== END OF FILE: gr-audio/examples/python/dial_tone_wav.py
====================


# ==================== START OF FILE: gr-audio/examples/python/mono_tone.py
====================
#!/usr/bin/env python
#
# Copyright 2004,2005,2007,2012 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import audio
from gnuradio.eng_arg import eng_float
from argparse import ArgumentParser

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)

#import os
# print os.getpid()
#raw_input('Attach gdb and press Enter: ')


class my_top_block(gr.top_block):

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

        parser = ArgumentParser()
        parser.add_argument("-O", "--audio-output", default="",
                                        help="pcm output device name.    E.g., hw:0,0 or /dev/dsp")
        parser.add_argument("-r", "--sample-rate", type=eng_float, default=48000,
                                        help="set sample rate to RATE %(default)r)")
        parser.add_argument("-D", "--dont-block", action="store_false", default=True,
                                        dest="ok_to_block")

        args = parser.parse_args()
        sample_rate = int(args.sample_rate)
        ampl = 0.5

        src0 = analog.sig_source_f(sample_rate, analog.GR_SIN_WAVE, 650, ampl)

        dst = audio.sink(sample_rate,
                                  args.audio_output,
                                  args.ok_to_block)

        self.connect(src0, (dst, 0))


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

# ==================== END OF FILE: gr-audio/examples/python/mono_tone.py
====================


# ==================== START OF FILE: gr-audio/examples/python/multi_tone.py
====================
#!/usr/bin/env python
#
# Copyright 2004,2006,2007,2012 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import audio
from gnuradio.eng_arg import eng_float, intx
from argparse import ArgumentParser

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)

#import os
# print os.getpid()
#raw_input('Attach gdb and press Enter: ')


class my_top_block(gr.top_block):

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

        parser = ArgumentParser()
        parser.add_argument("-O", "--audio-output", default="",
                            help="pcm output device name.  E.g., hw:0,0 or /dev/dsp")
        parser.add_argument("-r", "--sample-rate", type=eng_float, default=48000,
                            help="set sample rate to RATE (%(default)r)")
        parser.add_argument("-m", "--max-channels", type=int, default=16,
                            help="set maximum channels to use")
        parser.add_argument("-D", "--dont-block", action="store_false",
                            dest="ok_to_block")
        args = parser.parse_args()
        sample_rate = int(args.sample_rate)
        limit_channels = args.max_channels

        ampl = 0.1

        # With a tip of the hat to Harry Partch, may he R.I.P.
        # See "Genesis of a Music".  He was into some very wild tunings...
        base = 392
        ratios = {1: 1.0,
                  3: 3.0 / 2,
                  5: 5.0 / 4,
                  7: 7.0 / 4,
                  9: 9.0 / 8,
                  11: 11.0 / 8}

        # progression = (1, 5, 3, 7)
        # progression = (1, 9, 3, 7)
        # progression = (3, 7, 9, 11)
        # progression = (7, 11, 1, 5)
        progression = (7, 11, 1, 5, 9)

        dst = audio.sink(sample_rate,
                         args.audio_output,
                         args.ok_to_block)

        max_chan = dst.input_signature().max_streams()
        if (max_chan == -1) or (max_chan > limit_channels):
            max_chan = limit_channels

        for i in range(max_chan):
            quo, rem = divmod(i, len(progression))
            freq = base * ratios[progression[rem]] * (quo + 1)
            src = analog.sig_source_f(
                sample_rate, analog.GR_SIN_WAVE, freq, ampl)
            self.connect(src, (dst, i))


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

# ==================== END OF FILE: gr-audio/examples/python/multi_tone.py
====================


# ==================== START OF FILE: gr-audio/examples/python/noise.py
====================
#!/usr/bin/env python
#
# Copyright 2007 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import audio
from gnuradio import digital
from gnuradio.eng_arg import eng_float
from argparse import ArgumentParser


class my_top_block(gr.top_block):

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

        parser = ArgumentParser()
        parser.add_argument("-O", "--audio-output", default="",
                            help="pcm output device name.   E.g., hw:0,0 or /dev/dsp")
        parser.add_argument("-r", "--sample-rate", type=eng_float, default=48000,
                            help="set sample rate to RATE (48000)")
        args = parser.parse_args()
        sample_rate = int(args.sample_rate)
        ampl = 0.1

        src = digital.glfsr_source_b(32)      # Pseudorandom noise source
        b2f = digital.chunks_to_symbols_bf([ampl, -ampl], 1)
        dst = audio.sink(sample_rate, args.audio_output)
        self.connect(src, b2f, dst)


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

# ==================== END OF FILE: gr-audio/examples/python/noise.py ====================


# ==================== START OF FILE: gr-audio/examples/python/spectrum_inversion.py ====================
#!/usr/bin/env python
#
# Copyright 2004,2005,2007,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
```

```
# SPDX-License-Identifier: GPL-3.0-or-later
#

#
# Gang - Here's a simple script that demonstrates spectrum inversion
# using the multiply by [1,-1] method (mixing with Nyquist frequency).
# Requires nothing but a sound card, and sounds just like listening
# to a SSB signal on the wrong sideband.
#

from gnuradio import gr
from gnuradio import audio
from gnuradio import blocks
from gnuradio.eng_arg import eng_float, intx
from argparse import ArgumentParser


class my_top_block(gr.top_block):

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

        parser = ArgumentParser()
        parser.add_argument("-I", "--audio-input", default="",
                            help="pcm input device name.  E.g., hw:0,0 or /dev/dsp")
        parser.add_argument("-O", "--audio-output", default="",
                            help="pcm output device name.  E.g., hw:0,0 or /dev/dsp")
        parser.add_argument("-r", "--sample-rate", type=eng_float, default=8000,
                            help="set sample rate to RATE (%(default)r)")
        args = parser.parse_args()
        sample_rate = int(args.sample_rate)
        src = audio.source(sample_rate, args.audio_input)
        dst = audio.sink(sample_rate, args.audio_output)

        vec1 = [1, -1]
        vsource = blocks.vector_source_f(vec1, True)
        multiply = blocks.multiply_ff()

        self.connect(src, (multiply, 0))
        self.connect(vsource, (multiply, 1))
        self.connect(multiply, dst)


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

# ==================== END OF FILE: gr-audio/examples/python/spectrum_inversion.py ====================


# ==================== START OF FILE: gr-audio/examples/python/test_resampler.py ====================
#!/usr/bin/env python
#
# Copyright 2004,2005,2007,2012 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

import math
from gnuradio import gr
from gnuradio import audio
from gnuradio import filter
from gnuradio.eng_arg import eng_float, intx
from argparse import ArgumentParser

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)

try:
    from gnuradio import blocks
except ImportError:
    sys.stderr.write("Error: Program requires gr-blocks.\n")
    sys.exit(1)


class my_top_block(gr.top_block):

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

        parser = ArgumentParser()
        parser.add_argument("-O", "--audio-output", default="",
                          help="pcm output device name.   E.g., hw:0,0 or /dev/dsp")
        parser.add_argument("-i", "--input-rate", type=eng_float, default=8000,
                          help="set input sample rate to RATE %(default)r")
        parser.add_argument("-o", "--output-rate", type=eng_float, default=48000,
                          help="set output sample rate to RATE %(default)r")
        args = parser.parse_args()
        input_rate = int(args.input_rate)
        output_rate = int(args.output_rate)

        interp = output_rate // math.gcd(input_rate, output_rate)
        decim = input_rate // math.gcd(input_rate, output_rate)

        print("interp =", interp)
        print("decim  =", decim)

        ampl = 0.1
        src0 = analog.sig_source_f(input_rate, analog.GR_SIN_WAVE, 650, ampl)
        rr = filter.rational_resampler_fff(interp, decim)
        dst = audio.sink(output_rate, args.audio_output)
        self.connect(src0, rr, (dst, 0))


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

# ==================== END OF FILE: gr-audio/examples/python/test_resampler.py
====================



# ==================== START OF FILE: gr-blocks/examples/tags/test_file_tags.py
====================
#!/usr/bin/env python
#
# Copyright 2011,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
```

```
# SPDX-License-Identifier: GPL-3.0-or-later
#
#


from gnuradio import gr
from gnuradio import blocks
import sys
import numpy


def main():
	data = numpy.arange(0, 32000, 1).tolist()
	trig = 100 * [0, ] + 100 * [1, ]

	src = blocks.vector_source_s(data, True)
	trigger = blocks.vector_source_s(trig, True)

	thr = blocks.throttle(gr.sizeof_short, 10e3)
	ann = blocks.annotator_alltoall(1000000, gr.sizeof_short)
	tagger = blocks.burst_tagger(gr.sizeof_short)

	fsnk = blocks.tagged_file_sink(gr.sizeof_short, 1)

	tb = gr.top_block()
	tb.connect(src, thr, (tagger, 0))
	tb.connect(trigger, (tagger, 1))
	tb.connect(tagger, fsnk)

	tb.run()


if __name__ == "__main__":
	main()

# ==================== END OF FILE: gr-blocks/examples/tags/test_file_tags.py ====================


# ==================== START OF FILE: gr-blocks/examples/ctrlport/simple_copy_controller.py ====================
#!/usr/bin/env python

import sys
import pmt
from gnuradio.ctrlport.GNURadioControlPortClient import GNURadioControlPortClient

args = sys.argv
if(len(args) < 4):
        sys.stderr.write(
                'Not enough arguments: simple_copy_controller.py <host> <port> [true|false]\n\n')
        sys.exit(1)

hostname = args[1]
portnum = int(args[2])
msg = args[3].lower()
radiosys = GNURadioControlPortClient(
        host=hostname, port=portnum, rpcmethod='thrift')
radio = radiosys.client

if(msg == 'true'):
        radio.postMessage('copy0', 'en', pmt.PMT_T)
elif(msg == 'false'):
        radio.postMessage('copy0', 'en', pmt.PMT_F)
else:
        sys.stderr.write('Unrecognized message: must be true or false.\n\n')
        sys.exit(1)

# ==================== END OF FILE: gr-blocks/examples/ctrlport/simple_copy_controller.py ====================


# ==================== START OF FILE: gr-blocks/examples/ctrlport/usrp_sink_controller.py ====================
#!/usr/bin/env python

import sys
import pmt
from gnuradio.ctrlport.GNURadioControlPortClient import GNURadioControlPortClient
from argparse import ArgumentParser

parser = ArgumentParser()
parser.add_argument("-H", "--host", default="localhost",
                    help="Hostname to connect to (default=%(default)r)")
parser.add_argument("-p", "--port", type=int, default=9090,
                    help="Port of Controlport instance on host (default=%(default)r)")
parser.add_argument("-a", "--alias", default="gr uhd usrp sink0",
                    help="The UHD block's alias to control (default=%(default)r)")
```

```
parser.add_argument("command", metavar="COMMAND")
parser.add_argument("value", metavar="VALUE")
args = parser.parse_args()

port = 'command'
cmd = args.command
val = args.value

if(cmd == "tune" or cmd == "time"):
        sys.stderr.write("This application currently does not support the 'tune' or 'time' UHD "
                                    "message commands.\n\n")
        sys.exit(1)
elif(cmd == "antenna"):
        val = pmt.intern(val)
else:
        val = pmt.from_double(float(val))

radiosys = GNURadioControlPortClient(
        host=args.host, port=args.port, rpcmethod='thrift')
radio = radiosys.client

radio.postMessage(args.alias, port, pmt.cons(pmt.intern(cmd), val))

# ==================== END OF FILE: gr-blocks/examples/ctrlport/usrp_sink_controller.py ====================


# ==================== START OF FILE: gr-blocks/examples/ctrlport/usrp_source_controller.py ====================
#!/usr/bin/env python

import sys
import pmt
from gnuradio.ctrlport.GNURadioControlPortClient import GNURadioControlPortClient
from argparse import ArgumentParser

parser = ArgumentParser()
parser.add_argument("-H", "--host", default="localhost",
                                help="Hostname to connect to (default=%(default)r)")
parser.add_argument("-p", "--port", type=int, default=9090,
                                help="Port of Controlport instance on host (default=%(default)r)")
parser.add_argument("-a", "--alias", default="gr uhd usrp source0",
                                help="The UHD block's alias to control (default=%(default)r)")
parser.add_argument("command", metavar="COMMAND")
parser.add_argument("value", metavar="VALUE")
args = parser.parse_args()

port = 'command'
cmd = args.command
val = args.value

if(cmd == "tune" or cmd == "time"):
        sys.stderr.write("This application currently does not support the 'tune' or 'time' UHD "
                                        "message commands.\n\n")
        sys.exit(1)
if(cmd == "antenna"):
        val = pmt.intern(val)
else:
        val = pmt.from_double(float(val))

radiosys = GNURadioControlPortClient(
        host=args.host, port=args.port, rpcmethod='thrift')
radio = radiosys.client

radio.postMessage(args.alias, port, pmt.cons(pmt.intern(cmd), val))

# ==================== END OF FILE: gr-blocks/examples/ctrlport/usrp_source_controller.py ====================


# ==================== START OF FILE: gr-digital/examples/berawgn.py ====================
#!/usr/bin/env python
#
# Copyright 2012,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

"""
BER simulation for QPSK signals, compare to theoretical values.
Change the N_BITS value to simulate more bits per Eb/N0 value,
thus allowing to check for lower BER values.

Lower values will work faster, higher values will use a lot of RAM.
```

```
Also, this app isn't highly optimized--the flow graph is completely
reinstantiated for every Eb/N0 value.
Of course, expect the maximum value for BER to be one order of
magnitude below what you chose for N_BITS.
"""



import math
import numpy
from gnuradio import gr, digital
from gnuradio import analog
from gnuradio import blocks
import sys

try:
	from scipy.special import erfc
except ImportError:
	print("Error: could not import scipy (http://www.scipy.org/)")
	sys.exit(1)

try:
	from matplotlib import pyplot
except ImportError:
	print("Error: could not from matplotlib import pyplot
(http://matplotlib.sourceforge.net/)")
	sys.exit(1)

# Best to choose powers of 10
N_BITS = 1e7
RAND_SEED = 42



def berawgn(EbN0):
	""" Calculates theoretical bit error rate in AWGN (for BPSK and given Eb/N0) """
	return 0.5 * erfc(math.sqrt(10**(float(EbN0) / 10)))



class BitErrors(gr.hier_block2):
	""" Two inputs: true and received bits. We compare them and
	add up the number of incorrect bits. Because integrate_ff()
	can only add up a certain number of values, the output is
	not a scalar, but a sequence of values, the sum of which is
	the BER. """
```

```
    def __init__(self, bits_per_byte):
        gr.hier_block2.__init__(self, "BitErrors",
                                gr.io_signature(2, 2, gr.sizeof_char),
                                gr.io_signature(1, 1, gr.sizeof_int))

        # Bit comparison
        comp = blocks.xor_bb()
        intdump_decim = 100000
        if N_BITS < intdump_decim:
            intdump_decim = int(N_BITS)
        self.connect(self,
                     comp,
                     blocks.unpack_k_bits_bb(bits_per_byte),
                     blocks.uchar_to_float(),
                     blocks.integrate_ff(intdump_decim),
                     blocks.multiply_const_ff(1.0 / N_BITS),
                     self)
        self.connect((self, 1), (comp, 1))


class BERAWGNSimu(gr.top_block):
    " This contains the simulation flow graph "

    def __init__(self, EbN0):
        gr.top_block.__init__(self)
        self.const = digital.qpsk_constellation()
        # Source is N_BITS bits, non-repeated
        data = list(map(int, numpy.random.randint(
            0, self.const.arity(), N_BITS / self.const.bits_per_symbol())))
        src = blocks.vector_source_b(data, False)
        mod = digital.chunks_to_symbols_bc((self.const.points()), 1)
        add = blocks.add_vcc()
        noise = analog.noise_source_c(analog.GR_GAUSSIAN,
                                      self.EbN0_to_noise_voltage(EbN0),
                                      RAND_SEED)
        demod = digital.constellation_decoder_cb(self.const.base())
        ber = BitErrors(self.const.bits_per_symbol())
        self.sink = blocks.vector_sink_f()
        self.connect(src, mod, add, demod, ber, self.sink)
        self.connect(noise, (add, 1))
        self.connect(src, (ber, 1))

    def EbN0_to_noise_voltage(self, EbN0):
        """ Converts Eb/N0 to a complex noise voltage (assuming unit symbol power) """
        return 1.0 / math.sqrt(self.const.bits_per_symbol(* 10**(float(EbN0) / 10)))


def simulate_ber(EbN0):
    """ All the work's done here: create flow graph, run, read out BER """
    print("Eb/N0 = %d dB" % EbN0)
    fg = BERAWGNSimu(EbN0)
    fg.run()
    return numpy.sum(fg.sink.data())


if __name__ == "__main__":
    EbN0_min = 0
    EbN0_max = 15
    EbN0_range = list(range(EbN0_min, EbN0_max + 1))
    ber_theory = [berawgn(x) for x in EbN0_range]
    print("Simulating...")
    ber_simu = [simulate_ber(x) for x in EbN0_range]

    f = pyplot.figure()
    s = f.add_subplot(1, 1, 1)
    s.semilogy(EbN0_range, ber_theory, 'g-.', label="Theoretical")
    s.semilogy(EbN0_range, ber_simu, 'b-o', label="Simulated")
    s.set_title('BER Simulation')
    s.set_xlabel('Eb/N0 (dB)')
    s.set_ylabel('BER')
    s.legend()
    s.grid()
    pyplot.show()

# ==================== END OF FILE: gr-digital/examples/berawgn.py
====================


# ==================== START OF FILE: gr-digital/examples/example_costas.py
====================
#!/usr/bin/env python
#
# Copyright 2011-2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
```

```
#


from gnuradio import gr, digital, filter
from gnuradio import blocks
from gnuradio import channels
from gnuradio import eng_notation
from gnuradio.eng_arg import eng_float, intx
from argparse import ArgumentParser
import sys
import numpy

try:
    from matplotlib import pyplot
except ImportError:
    print("Error: could not import pyplot (http://matplotlib.sourceforge.net/)")
    sys.exit(1)


class example_costas(gr.top_block):
    def __init__(self, N, sps, rolloff, ntaps, bw, noise, foffset, toffset, poffset):
        gr.top_block.__init__(self)

        rrc_taps = filter.firdes.root_raised_cosine(
            sps, sps, 1.0, rolloff, ntaps)

        data = 2.0 * numpy.random.randint(0, 2, N) - 1.0
        data = numpy.exp(1j * poffset) * data

        self.src = blocks.vector_source_c(data.tolist(), False)
        self.rrc = filter.interp_fir_filter_ccf(sps, rrc_taps)
        self.chn = channels.channel_model(noise, foffset, toffset)
        self.cst = digital.costas_loop_cc(bw, 2)

        self.vsnk_src = blocks.vector_sink_c()
        self.vsnk_cst = blocks.vector_sink_c()
        self.vsnk_frq = blocks.vector_sink_f()

        self.connect(self.src, self.rrc, self.chn, self.cst, self.vsnk_cst)
        self.connect(self.rrc, self.vsnk_src)
        self.connect((self.cst, 1), self.vsnk_frq)


def main():
    parser = ArgumentParser(conflict_handler="resolve")
    parser.add_argument("-N", "--nsamples", type=int, default=2000,
                        help="Set the number of samples to process
[default=%(default)r]")
    parser.add_argument("-S", "--sps", type=int, default=4,
                        help="Set the samples per symbol [default=%(default)r]")
    parser.add_argument("-r", "--rolloff", type=eng_float, default=0.35,
                        help="Set the rolloff factor [default=%(default)r]")
    parser.add_argument("-W", "--bandwidth", type=eng_float, default=2 * numpy.pi /
100.0,
                        help="Set the loop bandwidth [default=%(default)r]")
    parser.add_argument("-n", "--ntaps", type=int, default=45,
                        help="Set the number of taps in the filters [default=%(default)r]")
    parser.add_argument("--noise", type=eng_float, default=0.0,
                        help="Set the simulation noise voltage [default=%(default)r]")
    parser.add_argument("-f", "--foffset", type=eng_float, default=0.0,
                        help="Set the simulation's normalized frequency offset (in Hz)
[default=%(default)r]")
    parser.add_argument("-t", "--toffset", type=eng_float, default=1.0,
                        help="Set the simulation's timing offset [default=%(default)r]")
    parser.add_argument("-p", "--poffset", type=eng_float, default=0.707,
                        help="Set the simulation's phase offset [default=%(default)r]")
    args = parser.parse_args()

    # Adjust N for the interpolation by sps
    args.nsamples = args.nsamples // args.sps

    # Set up the program-under-test
    put = example_costas(args.nsamples, args.sps, args.rolloff,
                         args.ntaps, args.bandwidth, args.noise,
                         args.foffset, args.toffset, args.poffset)
    put.run()

    data_src = numpy.array(put.vsnk_src.data())

    # Convert the FLL's LO frequency from rads/sec to Hz
    data_frq = numpy.array(put.vsnk_frq.data()) / (2.0 * numpy.pi)

    # adjust this to align with the data.
    data_cst = numpy.array(3 * [0, ] + list(put.vsnk_cst.data()))

    # Plot the Costas loop's LO frequency
    f1 = pyplot.figure(1, figsize=(12, 10), facecolor='w')
    s1 = f1.add_subplot(2, 2, 1)
```

```
        s1.plot(data_frq)
        s1.set_title("Costas LO")
        s1.set_xlabel("Samples")
        s1.set_ylabel("Frequency (normalized Hz)")

        # Plot the IQ symbols
        s3 = f1.add_subplot(2, 2, 2)
        s3.plot(data_src.real, data_src.imag, "o")
        s3.plot(data_cst.real, data_cst.imag, "rx")
        s3.set_title("IQ")
        s3.set_xlabel("Real part")
        s3.set_ylabel("Imag part")
        s3.set_xlim([-2, 2])
        s3.set_ylim([-2, 2])

        # Plot the symbols in time
        s4 = f1.add_subplot(2, 2, 3)
        s4.set_position([0.125, 0.05, 0.775, 0.4])
        s4.plot(data_src.real, "o-")
        s4.plot(data_cst.real, "rx-")
        s4.set_title("Symbols")
        s4.set_xlabel("Samples")
        s4.set_ylabel("Real Part of Signals")

        pyplot.show()


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

# ==================== END OF FILE: gr-digital/examples/example_costas.py
====================


# ==================== START OF FILE: gr-digital/examples/example_fll.py
====================
#!/usr/bin/env python
#
# Copyright 2011-2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
```

```
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#


from gnuradio import gr, digital, filter
from gnuradio import blocks
from gnuradio import channels
from gnuradio import eng_notation
from gnuradio.eng_arg import eng_float, intx
from argparse import ArgumentParser
import sys
import numpy

try:
    from matplotlib import pyplot
except ImportError:
    print("Error:    could    not    from    matplotlib    import    pyplot
(http://matplotlib.sourceforge.net/)")
    sys.exit(1)


class example_fll(gr.top_block):
    def __init__(self, N, sps, rolloff, ntaps, bw, noise, foffset, toffset, poffset):
        gr.top_block.__init__(self)

        rrc_taps = filter.firdes.root_raised_cosine(
            sps, sps, 1.0, rolloff, ntaps)

        data = 2.0 * numpy.random.randint(0, 2, N) - 1.0
        data = numpy.exp(1j * poffset) * data

        self.src = blocks.vector_source_c(data.tolist(), False)
        self.rrc = filter.interp_fir_filter_ccf(sps, rrc_taps)
        self.chn = channels.channel_model(noise, foffset, toffset)
        self.fll = digital.fll_band_edge_cc(sps, rolloff, ntaps, bw)

        self.vsnk_src = blocks.vector_sink_c()
        self.vsnk_fll = blocks.vector_sink_c()
        self.vsnk_frq = blocks.vector_sink_f()
        self.vsnk_phs = blocks.vector_sink_f()
        self.vsnk_err = blocks.vector_sink_f()
        self.connect(self.src, self.rrc, self.chn, self.fll, self.vsnk_fll)
        self.connect(self.rrc, self.vsnk_src)
        self.connect((self.fll, 1), self.vsnk_frq)
        self.connect((self.fll, 2), self.vsnk_phs)
        self.connect((self.fll, 3), self.vsnk_err)


def main():
    parser = ArgumentParser(conflict_handler="resolve")
    parser.add_argument("-N", "--nsamples", type=int, default=2000,
                        help="Set the number of samples to process
[default=%(default)r]")
    parser.add_argument("-S", "--sps", type=int, default=4,
                        help="Set the samples per symbol [default=%(default)r]")
    parser.add_argument("-r", "--rolloff", type=eng_float, default=0.35,
                        help="Set the rolloff factor [default=%(default)r]")
    parser.add_argument("-W", "--bandwidth", type=eng_float, default=2 * numpy.pi /
100.0,
                        help="Set the loop bandwidth [default=%(default)r]")
    parser.add_argument("-n", "--ntaps", type=int, default=45,
                        help="Set the number of taps in the filters [default=%(default)r]")
    parser.add_argument("--noise", type=eng_float, default=0.0,
                        help="Set the simulation noise voltage [default=%(default)r]")
    parser.add_argument("-f", "--foffset", type=eng_float, default=0.2,
                        help="Set the simulation's normalized frequency offset (in Hz)
[default=%(default)r]")
    parser.add_argument("-t", "--toffset", type=eng_float, default=1.0,
                        help="Set the simulation's timing offset [default=%(default)r]")
    parser.add_argument("-p", "--poffset", type=eng_float, default=0.0,
                        help="Set the simulation's phase offset [default=%(default)r]")
    args = parser.parse_args()

    # Adjust N for the interpolation by sps
    args.nsamples = args.nsamples // args.sps

    # Set up the program-under-test
    put = example_fll(args.nsamples, args.sps, args.rolloff,
                      args.ntaps, args.bandwidth, args.noise,
                      args.foffset, args.toffset, args.poffset)
    put.run()

    data_src = numpy.array(put.vsnk_src.data())
    data_err = numpy.array(put.vsnk_err.data())
    # Convert the FLL's LO frequency from rads/sec to Hz
    data_frq = numpy.array(put.vsnk_frq.data()) / (2.0 * numpy.pi)

    # adjust this to align with the data. There are 2 filters of
    # ntaps long and the channel introduces another 4 sample delay.
    data_fll = numpy.array(put.vsnk_fll.data()[2 * args.ntaps - 4:])

    # Plot the FLL's LO frequency
    f1 = pyplot.figure(1, figsize=(12, 10))
    s1 = f1.add_subplot(2, 2, 1)
    s1.plot(data_frq)
    s1.set_title("FLL LO")
    s1.set_xlabel("Samples")
    s1.set_ylabel("Frequency (normalized Hz)")

    # Plot the FLL's error
    s2 = f1.add_subplot(2, 2, 2)
    s2.plot(data_err)
    s2.set_title("FLL Error")
    s2.set_xlabel("Samples")
    s2.set_ylabel("FLL Loop error")

    # Plot the IQ symbols
    s3 = f1.add_subplot(2, 2, 3)
    s3.plot(data_src.real, data_src.imag, "o")
    s3.plot(data_fll.real, data_fll.imag, "rx")
    s3.set_title("IQ")
    s3.set_xlabel("Real part")
    s3.set_ylabel("Imag part")

    # Plot the symbols in time
    s4 = f1.add_subplot(2, 2, 4)
    s4.plot(data_src.real, "o-")
    s4.plot(data_fll.real, "rx-")
    s4.set_title("Symbols")
    s4.set_xlabel("Samples")
    s4.set_ylabel("Real Part of Signals")

    pyplot.show()


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

# ==================== END OF FILE: gr-digital/examples/example_fll.py
====================


# ==================== START OF FILE: gr-digital/examples/example_timing.py
====================
#!/usr/bin/env python
#
# Copyright 2011-2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#


from gnuradio import gr, digital, filter
from gnuradio import blocks
from gnuradio import channels
from gnuradio import eng_notation
from gnuradio.eng_arg import eng_float, intx
from argparse import ArgumentParser
import sys
import numpy

try:
    from matplotlib import pyplot
except ImportError:
    print("Error: could not from matplotlib import pyplot
(http://matplotlib.sourceforge.net/)")
    sys.exit(1)


class example_timing(gr.top_block):
    def __init__(self, N, sps, rolloff, ntaps, bw, noise,
                 foffset, toffset, poffset, mode=0):
        gr.top_block.__init__(self)

        rrc_taps = filter.firdes.root_raised_cosine(
            sps, sps, 1.0, rolloff, ntaps)
```

```
gain = bw
nfilts = 32
rrc_taps_rx = filter.firdes.root_raised_cosine(
    nfilts, sps * nfilts, 1.0, rolloff, ntaps * nfilts)

data = 2.0 * numpy.random.randint(0, 2, N) - 1.0
data = numpy.exp(1j * poffset) * data

self.src = blocks.vector_source_c(data.tolist(), False)
self.rrc = filter.interp_fir_filter_ccf(sps, rrc_taps)
self.chn = channels.channel_model(noise, foffset, toffset)
self.off = filter.mmse_resampler_cc(0.20, 1.0)

if mode == 0:
    self.clk = digital.pfb_clock_sync_ccf(sps, gain, rrc_taps_rx,
                                          nfilts, nfilts // 2, 1)
    self.taps = self.clk.taps()
    self.dtaps = self.clk.diff_taps()

    self.delay = int(numpy.ceil(((len(rrc_taps) - 1) // 2 +
                                 (len(self.taps[0]) - 1) // 2) // float(sps))) + 1

    self.vsnk_err = blocks.vector_sink_f()
    self.vsnk_rat = blocks.vector_sink_f()
    self.vsnk_phs = blocks.vector_sink_f()

    self.connect((self.clk, 1), self.vsnk_err)
    self.connect((self.clk, 2), self.vsnk_rat)
    self.connect((self.clk, 3), self.vsnk_phs)

else:  # mode == 1
    mu = 0.5
    gain_mu = bw
    gain_omega = 0.25 * gain_mu * gain_mu
    omega_rel_lim = 0.02
    self.clk = digital.clock_recovery_mm_cc(sps, gain_omega,
                                            mu, gain_mu,
                                            omega_rel_lim)

    self.vsnk_err = blocks.vector_sink_f()

    self.connect((self.clk, 1), self.vsnk_err)
```

```
        self.vsnk_src = blocks.vector_sink_c()
        self.vsnk_clk = blocks.vector_sink_c()

        self.connect(self.src, self.rrc, self.chn,
                     self.off, self.clk, self.vsnk_clk)
        self.connect(self.src, self.vsnk_src)


def main():
    parser = ArgumentParser(conflict_handler="resolve")
    parser.add_argument("-N", "--nsamples", type=int, default=2000,
                        help="Set the number of samples to process
[default=%(default)r]")
    parser.add_argument("-S", "--sps", type=int, default=4,
                        help="Set the samples per symbol [default=%(default)r]")
    parser.add_argument("-r", "--rolloff", type=eng_float, default=0.35,
                        help="Set the rolloff factor [default=%(default)r]")
    parser.add_argument("-W", "--bandwidth", type=eng_float, default=2 * numpy.pi /
100.0,
                        help="Set the loop bandwidth (PFB) or gain (M&M)
[default=%(default)r]")
    parser.add_argument("-n", "--ntaps", type=int, default=45,
                        help="Set the number of taps in the filters [default=%(default)r]")
    parser.add_argument("--noise", type=eng_float, default=0.0,
                        help="Set the simulation noise voltage [default=%(default)r]")
    parser.add_argument("-f", "--foffset", type=eng_float, default=0.0,
                        help="Set the simulation's normalized frequency offset (in Hz)
[default=%(default)r]")
    parser.add_argument("-t", "--toffset", type=eng_float, default=1.0,
                        help="Set the simulation's timing offset [default=%(default)r]")
    parser.add_argument("-p", "--poffset", type=eng_float, default=0.0,
                        help="Set the simulation's phase offset [default=%(default)r]")
    parser.add_argument("-M", "--mode", type=int, default=0,
                        help="Set the recovery mode (0: polyphase, 1: M&M)
[default=%(default)r]")
    args = parser.parse_args()

    # Adjust N for the interpolation by sps
    args.nsamples = args.nsamples // args.sps

    # Set up the program-under-test
    put = example_timing(args.nsamples, args.sps, args.rolloff,
                         args.ntaps, args.bandwidth, args.noise,
                         args.foffset, args.toffset, args.poffset,
                                        args.mode)
put.run()

if args.mode == 0:
        data_src = numpy.array(put.vsnk_src.data()[20:])
        data_clk = numpy.array(put.vsnk_clk.data()[20:])

        data_err = numpy.array(put.vsnk_err.data()[20:])
        data_rat = numpy.array(put.vsnk_rat.data()[20:])
        data_phs = numpy.array(put.vsnk_phs.data()[20:])

        f1 = pyplot.figure(1, figsize=(12, 10), facecolor='w')

        # Plot the IQ symbols
        s1 = f1.add_subplot(2, 2, 1)
        s1.plot(data_src.real, data_src.imag, "bo")
        s1.plot(data_clk.real, data_clk.imag, "ro")
        s1.set_title("IQ")
        s1.set_xlabel("Real part")
        s1.set_ylabel("Imag part")
        s1.set_xlim([-2, 2])
        s1.set_ylim([-2, 2])

        # Plot the symbols in time
        delay = put.delay
        m = len(data_clk.real)
        s2 = f1.add_subplot(2, 2, 2)
        s2.plot(data_src.real, "bs", markersize=10, label="Input")
        s2.plot(data_clk.real[delay:], "ro", label="Recovered")
        s2.set_title("Symbols")
        s2.set_xlabel("Samples")
        s2.set_ylabel("Real Part of Signals")
        s2.legend()

        # Plot the clock recovery loop's error
        s3 = f1.add_subplot(2, 2, 3)
        s3.plot(data_err, label="Error")
        s3.plot(data_rat, 'r', label="Update rate")
        s3.set_title("Clock Recovery Loop Error")
        s3.set_xlabel("Samples")
        s3.set_ylabel("Error")
        s3.set_ylim([-0.5, 0.5])
        s3.legend()
        # Plot the clock recovery loop's error
        s4 = f1.add_subplot(2, 2, 4)
        s4.plot(data_phs)
        s4.set_title("Clock Recovery Loop Filter Phase")
        s4.set_xlabel("Samples")
        s4.set_ylabel("Filter Phase")

        diff_taps = put.dtaps
        ntaps = len(diff_taps[0])
        nfilts = len(diff_taps)
        t = numpy.arange(0, ntaps * nfilts)

        f3 = pyplot.figure(3, figsize=(12, 10), facecolor='w')
        s31 = f3.add_subplot(2, 1, 1)
        s32 = f3.add_subplot(2, 1, 2)
        s31.set_title("Differential Filters")
        s32.set_title("FFT of Differential Filters")

        for i, d in enumerate(diff_taps):
            D = 20.0 * \
                numpy.log10(
                    1e-20 + abs(numpy.fft.fftshift(numpy.fft.fft(d, 10000))))
            s31.plot(t[i::nfilts].real, d, "-o")
            s32.plot(D)
        s32.set_ylim([-120, 10])

    # If testing the M&M clock recovery loop
    else:
        data_src = numpy.array(put.vsnk_src.data()[20:])
        data_clk = numpy.array(put.vsnk_clk.data()[20:])

        data_err = numpy.array(put.vsnk_err.data()[20:])

        f1 = pyplot.figure(1, figsize=(12, 10), facecolor='w')

        # Plot the IQ symbols
        s1 = f1.add_subplot(2, 2, 1)
        s1.plot(data_src.real, data_src.imag, "o")
        s1.plot(data_clk.real, data_clk.imag, "ro")
        s1.set_title("IQ")
        s1.set_xlabel("Real part")
        s1.set_ylabel("Imag part")
        s1.set_xlim([-2, 2])
        s1.set_ylim([-2, 2])
        # Plot the symbols in time
        s2 = f1.add_subplot(2, 2, 2)
        s2.plot(data_src.real, "bs", markersize=10, label="Input")
        s2.plot(data_clk.real, "ro", label="Recovered")
        s2.set_title("Symbols")
        s2.set_xlabel("Samples")
        s2.set_ylabel("Real Part of Signals")
        s2.legend()

        # Plot the clock recovery loop's error
        s3 = f1.add_subplot(2, 2, 3)
        s3.plot(data_err)
        s3.set_title("Clock Recovery Loop Error")
        s3.set_xlabel("Samples")
        s3.set_ylabel("Error")

    pyplot.show()


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

# ==================== END OF FILE: gr-digital/examples/example_timing.py ====================



# ==================== START OF FILE: gr-digital/examples/gen_whitener.py ====================
#!/usr/bin/env python
#
# Copyright 2011,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import blocks
from argparse import ArgumentParser
import sys


class my_graph(gr.top_block):

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

        parser = ArgumentParser()
        args = parser.parse_args()
        src = blocks.lfsr_32k_source_s()
        head = blocks.head(gr.sizeof_short, 2048)
        self.dst = blocks.vector_sink_s()
        self.connect(src, head, self.dst)


if __name__ == '__main__':
    try:
        tb = my_graph()
        tb.run()
        f = sys.stdout
        i = 0
        for s in tb.dst.data():
            f.write("%3d, " % (s & 0xff,))
            f.write("%3d, " % ((s >> 8) & 0xff,))
            i = i + 2
            if i % 16 == 0:
                f.write('\n')

    except KeyboardInterrupt:
        pass

# ==================== END OF FILE: gr-digital/examples/gen_whitener.py ====================


# ==================== START OF FILE: gr-digital/examples/run_length.py ====================
#!/usr/bin/env python
#
# Copyright 2007 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
```

```
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#


from optparse import OptionParser
import sys


def main():
    parser = OptionParser()
    parser.add_option("-f", "--file", default=None,
                      help="Choose file to read data from.")
    (options, args) = parser.parse_args()

    if options.file == None:
        print("Must specify file to read from using '-f'.")
        sys.exit(1)
    print("Using", options.file, "for data.")

    f = open(options.file, 'r')
    runs = []
    count = 0
    current = 0
    bytes = 0
    bits = 0

    for ch in f.read():
        x = ord(ch)
        bytes = bytes + 1
        for i in range(7, -1, -1):
            bits = bits + 1
            t = (x >> i) & 0x1
            if t == current:
                count = count + 1
            else:
                if count > 0:
                    if len(runs) < count:
                        for j in range(count - len(runs)):
                            runs.append(0)
                    runs[count - 1] = runs[count - 1] + 1

                current = 1 - current
                count = 1

    # Deal with last run at EOF
    if len(runs) < count and count > 0:
        for j in range(count - len(runs)):
            runs.append(0)
    runs[count - 1] = runs[count - 1] + 1

    chk = 0
    print("Bytes read: ", bytes)
    print("Bits read:    ", bits)
    print()
    for i in range(len(runs)):
        chk = chk + runs[i] * (i + 1)
        print("Runs of length", i + 1, ":", runs[i])
    print()
    print("Sum of runs:", chk, "bits")
    print()
    print("Maximum run length is", len(runs), "bits")


if __name__ == "__main__":
    main()

# ==================== END OF FILE: gr-digital/examples/run_length.py ====================


# ==================== START OF FILE: gr-digital/examples/snr_estimators.py ====================
#!/usr/bin/env python
#
# Copyright 2011-2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#


import sys

try:
    import scipy
    from scipy import stats
except ImportError:
    print("Error: Program requires scipy (www.scipy.org).")
    sys.exit(1)

try:
    from matplotlib import pyplot
except ImportError:
    print("Error: Program requires Matplotlib (matplotlib.org).")
    sys.exit(1)

from gnuradio import gr, digital, filter
from gnuradio import blocks
from gnuradio import channels
from optparse import OptionParser
from gnuradio.eng_option import eng_option

'''
This example program uses Python and GNU Radio to calculate SNR of a
noise BPSK signal to compare them.

For an explanation of the online algorithms, see:
http://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Higher-order_statistics
'''


def online_skewness(data):
    n = 0
    mean = 0
    M2 = 0
    M3 = 0

    for n in range(len(data)):
        delta = data[n] - mean
        delta_n = delta / (n + 1)
        term1 = delta * delta_n * n
        mean = mean + delta_n
        M3 = M3 + term1 * delta_n * (n - 1) - 3 * delta_n * M2
        M2 = M2 + term1

    return scipy.sqrt(len(data)) * M3 / scipy.power(M2, 3.0 / 2.0)
```

```
def snr_est_simple(signal):
    s = scipy.mean(abs(signal)**2)
    n = 2 * scipy.var(abs(signal))
    snr_rat = s / n
    return 10.0 * scipy.log10(snr_rat), snr_rat


def snr_est_skew(signal):
    y1 = scipy.mean(abs(signal))
    y2 = scipy.mean(scipy.real(signal**2))
    y3 = (y1 * y1 - y2)
    y4 = online_skewness(signal.real)
    #y4 = stats.skew(abs(signal.real))

    skw = y4 * y4 / (y2 * y2 * y2)
    s = y1 * y1
    n = 2 * (y3 + skw * s)
    snr_rat = s / n
    return 10.0 * scipy.log10(snr_rat), snr_rat


def snr_est_m2m4(signal):
    M2 = scipy.mean(abs(signal)**2)
    M4 = scipy.mean(abs(signal)**4)
    snr_rat = scipy.sqrt(2 * M2 * M2 - M4) / \
        (M2 - scipy.sqrt(2 * M2 * M2 - M4))
    return 10.0 * scipy.log10(snr_rat), snr_rat


def snr_est_svr(signal):
    N = len(signal)
    ssum = 0
    msum = 0
    for i in range(1, N):
        ssum += (abs(signal[i])**2) * (abs(signal[i - 1])**2)
        msum += (abs(signal[i])**4)
    savg = (1.0 / (float(N - 1.0))) * ssum
    mavg = (1.0 / (float(N - 1.0))) * msum
    beta = savg / (mavg - savg)

    snr_rat = ((beta - 1) + scipy.sqrt(beta * (beta - 1)))
    return 10.0 * scipy.log10(snr_rat), snr_rat
```

```
def main():
    gr_estimators = {"simple": digital.SNR_EST_SIMPLE,
                     "skew": digital.SNR_EST_SKEW,
                     "m2m4": digital.SNR_EST_M2M4,
                     "svr": digital.SNR_EST_SVR}
    py_estimators = {"simple": snr_est_simple,
                     "skew": snr_est_skew,
                     "m2m4": snr_est_m2m4,
                     "svr": snr_est_svr}

    parser = OptionParser(option_class=eng_option, conflict_handler="resolve")
    parser.add_option("-N", "--nsamples", type="int", default=10000,
                      help="Set the number of samples to process [default=%default]")
    parser.add_option("", "--snr-min", type="float", default=-5,
                      help="Minimum SNR [default=%default]")
    parser.add_option("", "--snr-max", type="float", default=20,
                      help="Maximum SNR [default=%default]")
    parser.add_option("", "--snr-step", type="float", default=0.5,
                      help="SNR step amount [default=%default]")
    parser.add_option("-t", "--type", type="choice",
                      choices=list(gr_estimators.keys()), default="simple",
                      help="Estimator type {0} [default=%default]".format(
                          list(gr_estimators.keys())))
    (options, args) = parser.parse_args()

    N = options.nsamples
    xx = scipy.random.randn(N)
    xy = scipy.random.randn(N)
    bits = 2 * scipy.complex64(scipy.random.randint(0, 2, N)) - 1
    # bits =(2*scipy.complex64(scipy.random.randint(0, 2, N)) - 1) + \
    #       1j*(2*scipy.complex64(scipy.random.randint(0, 2, N)) - 1)

    snr_known = list()
    snr_python = list()
    snr_gr = list()

    # when to issue an SNR tag; can be ignored in this example.
    ntag = 10000

    n_cpx = xx + 1j * xy

    py_est = py_estimators[options.type]
    gr_est = gr_estimators[options.type]
```

```
SNR_min = options.snr_min
SNR_max = options.snr_max
SNR_step = options.snr_step
SNR_dB = scipy.arange(SNR_min, SNR_max + SNR_step, SNR_step)
for snr in SNR_dB:
        SNR = 10.0**(snr / 10.0)
        scale = scipy.sqrt(2 * SNR)
        yy = bits + n_cpx / scale
        print("SNR: ", snr)

        Sknown = scipy.mean(yy**2)
        Nknown = scipy.var(n_cpx / scale)
        snr0 = Sknown / Nknown
        snr0dB = 10.0 * scipy.log10(snr0)
        snr_known.append(float(snr0dB))

        snrdB, snr = py_est(yy)
        snr_python.append(snrdB)

        gr_src = blocks.vector_source_c(bits.tolist(), False)
        gr_snr = digital.mpsk_snr_est_cc(gr_est, ntag, 0.001)
        gr_chn = channels.channel_model(1.0 / scale)
        gr_snk = blocks.null_sink(gr.sizeof_gr_complex)
        tb = gr.top_block()
        tb.connect(gr_src, gr_chn, gr_snr, gr_snk)
        tb.run()

        snr_gr.append(gr_snr.snr())

f1 = pyplot.figure(1)
s1 = f1.add_subplot(1, 1, 1)
s1.plot(SNR_dB, snr_known, "k-o", linewidth=2, label="Known")
s1.plot(SNR_dB, snr_python, "b-o", linewidth=2, label="Python")
s1.plot(SNR_dB, snr_gr, "g-o", linewidth=2, label="GNU Radio")
s1.grid(True)
s1.set_title('SNR Estimators')
s1.set_xlabel('SNR (dB)')
s1.set_ylabel('Estimated SNR')
s1.legend()

f2 = pyplot.figure(2)
s2 = f2.add_subplot(1, 1, 1)
s2.plot(yy.real, yy.imag, 'o')
```

```
        pyplot.show()


if __name__ == "__main__":
        main()

# ==================== END OF FILE: gr-digital/examples/snr_estimators.py
====================


# ==================== START OF FILE: gr-
digital/examples/narrowband/benchmark_add_channel.py ====================
#!/usr/bin/env python
#
# Copyright 2010,2011 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#


from gnuradio import channels, gr
from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

import random
import math
import sys


class my_top_block(gr.top_block):
        def __init__(self, ifile, ofile, options):
                gr.top_block.__init__(self)

                SNR = 10.0**(options.snr / 10.0)
                frequency_offset = options.frequency_offset
                time_offset = options.time_offset
                phase_offset = options.phase_offset * (math.pi / 180.0)

                # calculate noise voltage from SNR
                power_in_signal = abs(options.tx_amplitude)**2
                noise_power = power_in_signal / SNR
                noise_voltage = math.sqrt(noise_power)

                self.src = blocks.file_source(gr.sizeof_gr_complex, ifile)
                #self.throttle = blocks.throttle(gr.sizeof_gr_complex, options.sample_rate)

                self.channel = channels.channel_model(noise_voltage, frequency_offset,
                                                      time_offset,          noise_seed=-
random.randint(0, 100000))
                self.phase = blocks.multiply_const_cc(complex(math.cos(phase_offset),
                                                               math.sin(phase_offset)))
                self.snk = blocks.file_sink(gr.sizeof_gr_complex, ofile)

                self.connect(self.src, self.channel, self.phase, self.snk)


# /////////////////////////////////////////////////////////////////////////////
#                                   main
# /////////////////////////////////////////////////////////////////////////////

def main():
        # Create Options Parser:
        usage = "benchmack_add_channel.py [options] <input file> <output file>"
        parser = OptionParser(
                usage=usage, option_class=eng_option, conflict_handler="resolve")
        parser.add_option("-n", "--snr", type="eng_float", default=30,
                          help="set the SNR of the channel in dB [default=%default]")
        parser.add_option("", "--seed", action="store_true", default=False,
                          help="use a random seed for AWGN noise [default=%default]")
        parser.add_option("-f", "--frequency-offset", type="eng_float", default=0,
                          help="set    frequency    offset    introduced    by    channel
[default=%default]")
        parser.add_option("-t", "--time-offset", type="eng_float", default=1.0,
                          help="set timing offset between Tx and Rx [default=%default]")
        parser.add_option("-p", "--phase-offset", type="eng_float", default=0,
                          help="set   phase   offset   (in   degrees)   between   Tx   and   Rx
[default=%default]")
        parser.add_option("-m", "--use-multipath", action="store_true", default=False,
                          help="Use a multipath channel [default=%default]")
        parser.add_option("", "--tx-amplitude", type="eng_float",
                          default=1.0,
                          help="tell the simulator the signal amplitude [default=%default]")
        (options, args) = parser.parse_args()

        if len(args) != 2:
                parser.print_help(sys.stderr)
                sys.exit(1)

        ifile = args[0]
        ofile = args[1]

        # build the graph
        tb = my_top_block(ifile, ofile, options)

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

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


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

# ==================== END OF FILE: gr-digital/examples/narrowband/benchmark_add_channel.py ====================


# ==================== START OF FILE: gr-digital/examples/narrowband/digital_bert_rx.py ====================
#!/usr/bin/env python
#
# Copyright 2008,2011,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#


from gnuradio import gr, eng_notation
from optparse import OptionParser
from gnuradio.eng_option import eng_option
import threading
import sys
import time
import math

from gnuradio import digital
from gnuradio import blocks

# from current dir
from uhd_interface import uhd_receiver

n2s = eng_notation.num_to_str


class status_thread(threading.Thread):
        def __init__(self, tb):
                threading.Thread.__init__(self)
                self.setDaemon(1)
                self.tb = tb
                self.done = False
                self.start()

        def run(self):
                while not self.done:
                        print("Freq. Offset: {0:5.0f} Hz   Timing Offset: {1:10.1f} ppm   Estimated SNR:
{2:4.1f} dB   BER: {3:g}".format(
                                tb.frequency_offset(), tb.timing_offset() * 1e6, tb.snr(), tb.ber()))
                        try:
                                time.sleep(1.0)
                        except KeyboardInterrupt:
                                self.done = True



class bert_receiver(gr.hier_block2):
        def __init__(self, bitrate,
                                 constellation, samples_per_symbol,
                                 differential, excess_bw, gray_coded,
                                 freq_bw, timing_bw, phase_bw,
                                 verbose, log):

                gr.hier_block2.__init__(self, "bert_receive",
                                                        # Input signature
```

```
                                        gr.io_signature(1, 1, gr.sizeof_gr_complex),
                                        gr.io_signature(0, 0, 0))                          # Output
signature

        self._bitrate = bitrate

        self._demod = digital.generic_demod(constellation, differential,
                                            samples_per_symbol,
                                            gray_coded, excess_bw,
                                            freq_bw, timing_bw, phase_bw,
                                            verbose, log)

        self._symbol_rate = self._bitrate / self._demod.bits_per_symbol()
        self._sample_rate = self._symbol_rate * samples_per_symbol

        # Add an SNR probe on the demodulated constellation
        self._snr_probe = digital.probe_mpsk_snr_est_c(digital.SNR_EST_M2M4, 1000,
                                                       alpha=10.0                /
self._symbol_rate)
        self.connect(self._demod.time_recov, self._snr_probe)

        # Descramble BERT sequence.  A channel error will create 3 incorrect bits
        self._descrambler = digital.descrambler_bb(
            0x8A, 0x7F, 7)  # CCSDS 7-bit descrambler

        # Measure BER by the density of 0s in the stream
        self._ber = digital.probe_density_b(1.0 / self._symbol_rate)

        self.connect(self, self._demod, self._descrambler, self._ber)

    def frequency_offset(self):
        return self._demod.freq_recov.get_frequency() * self._sample_rate / (2 * math.pi)

    def timing_offset(self):
        return self._demod.time_recov.clock_rate()

    def snr(self):
        return self._snr_probe.snr()

    def ber(self):
        return (1.0 - self._ber.density()) / 3.0


class rx_psk_block(gr.top_block):
```

```
def __init__(self, demod, options):

        gr.top_block.__init__(self, "rx_mpsk")

        self._demodulator_class = demod

        # Get demod_kwargs
        demod_kwargs = self._demodulator_class.extract_kwargs_from_options(
                options)

        # demodulator
        self._demodulator = self._demodulator_class(**demod_kwargs)

        if(options.rx_freq is not None):
                symbol_rate = options.bitrate / self._demodulator.bits_per_symbol()
                self._source = uhd_receiver(options.args, symbol_rate,
                                                        options.samples_per_symbol,
                                                        options.rx_freq, options.rx_gain,
                                                        options.spec,
                                                        options.antenna, options.verbose)
                options.samples_per_symbol = self._source._sps

        elif(options.from_file is not None):
                self._source = blocks.file_source(
                        gr.sizeof_gr_complex, options.from_file)
        else:
                self._source = blocks.null_source(gr.sizeof_gr_complex)

        # Create the BERT receiver
        self._receiver = bert_receiver(options.bitrate,
                                                       self._demodulator._constellation,
                                                       options.samples_per_symbol,
                                                       options.differential,
                                                       options.excess_bw,
                                                       gray_coded=True,
                                                       freq_bw=options.freq_bw,
                                                       timing_bw=options.timing_bw,
                                                       phase_bw=options.phase_bw,
                                                       verbose=options.verbose,
                                                       log=options.log)

        self.connect(self._source, self._receiver)

def snr(self):
        return self._receiver.snr()

    def mag(self):
        return self._receiver.signal_mean()

    def var(self):
        return self._receiver.noise_variance()

    def ber(self):
        return self._receiver.ber()

    def frequency_offset(self):
        return self._receiver.frequency_offset()

    def timing_offset(self):
        return self._receiver.timing_offset()


def get_options(demods):
    parser = OptionParser(option_class=eng_option, conflict_handler="resolve")
    parser.add_option("", "--from-file", default=None,
                      help="input file of samples to demod")
    parser.add_option("-m", "--modulation", type="choice", choices=list(demods.keys()),
                      default='psk',
                      help="Select modulation from: %s [default=%%default]"
                            % (', '.join(list(demods.keys())),))
    parser.add_option("-r", "--bitrate", type="eng_float", default=250e3,
                      help="Select modulation bit rate (default=%default)")
    parser.add_option("-S", "--samples-per-symbol", type="float", default=2,
                      help="set samples/symbol [default=%default]")
    if not parser.has_option("--verbose"):
        parser.add_option("-v", "--verbose",
                          action="store_true", default=False)
    if not parser.has_option("--log"):
        parser.add_option("", "--log", action="store_true", default=False,
                          help="Log all parts of flow graph to files (CAUTION: lots of
data)")

    uhd_receiver.add_options(parser)

    demods = digital.modulation_utils.type_1_demods()
    for mod in list(demods.values()):
        mod.add_options(parser)
    (options, args) = parser.parse_args()
    if len(args) != 0:
        parser.print_help()
        sys.exit(1)

    return (options, args)


if __name__ == "__main__":
    print("""Warning: this example in its current shape is deprecated and
            will be removed or fundamentally reworked in a coming GNU Radio
            release.""")
    demods = digital.modulation_utils.type_1_demods()

    (options, args) = get_options(demods)

    demod = demods[options.modulation]
    tb = rx_psk_block(demod, options)

    print("\n*** SNR estimator is inaccurate below about 7dB")
    print("*** BER estimator is inaccurate above about 10%\n")
    updater = status_thread(tb)

    try:
        tb.run()
    except KeyboardInterrupt:
        updater.done = True
        updater = None

# ==================== END OF FILE: gr-digital/examples/narrowband/digital_bert_rx.py ====================


# ==================== START OF FILE: gr-digital/examples/narrowband/digital_bert_tx.py ====================
#!/usr/bin/env python
#
# Copyright 2008,2011,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#
```

```
from gnuradio import gr, eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import sys

from gnuradio import blocks
from gnuradio import digital

# from current dir
from uhd_interface import uhd_transmitter

n2s = eng_notation.num_to_str


class bert_transmit(gr.hier_block2):
    def __init__(self, constellation, samples_per_symbol,
                 differential, excess_bw, gray_coded,
                 verbose, log):

        gr.hier_block2.__init__(self, "bert_transmit",
                                # Output signature
                                gr.io_signature(0, 0, 0),
                                gr.io_signature(1,  1,  gr.sizeof_gr_complex))     #  Input
signature

        # Create BERT data bit stream
        self._bits = blocks.vector_source_b(
            [1, ], True)          # Infinite stream of ones
        self._scrambler = digital.scrambler_bb(
            0x8A, 0x7F, 7)   # CCSDS 7-bit scrambler

        self._mod = digital.generic_mod(constellation, differential,
                                        samples_per_symbol,
                                        gray_coded, excess_bw,
                                        verbose, log)

        self._pack = blocks.unpacked_to_packed_bb(
            self._mod.bits_per_symbol(), gr.GR_MSB_FIRST)

        self.connect(self._bits, self._scrambler, self._pack, self._mod, self)


class tx_psk_block(gr.top_block):
    def __init__(self, mod, options):
        gr.top_block.__init__(self, "tx_mpsk")

        self._modulator_class = mod

        # Get mod_kwargs
        mod_kwargs = self._modulator_class.extract_kwargs_from_options(options)

        # transmitter
        self._modulator = self._modulator_class(**mod_kwargs)

        if(options.tx_freq is not None):
            symbol_rate = options.bitrate / self._modulator.bits_per_symbol()
            self._sink = uhd_transmitter(options.args, symbol_rate,
                                         options.samples_per_symbol,
                                         options.tx_freq, options.tx_gain,
                                         options.spec,
                                         options.antenna, options.verbose)
            options.samples_per_symbol = self._sink._sps

        elif(options.to_file is not None):
            self._sink = blocks.file_sink(
                gr.sizeof_gr_complex, options.to_file)
        else:
            self._sink = blocks.null_sink(gr.sizeof_gr_complex)

        self._transmitter = bert_transmit(self._modulator._constellation,
                                          options.samples_per_symbol,
                                          options.differential,
                                          options.excess_bw,
                                          gray_coded=True,
                                          verbose=options.verbose,
                                          log=options.log)

        self.amp = blocks.multiply_const_cc(options.amplitude)
        self.connect(self._transmitter, self.amp, self._sink)


def get_options(mods):
    parser = OptionParser(option_class=eng_option, conflict_handler="resolve")
    parser.add_option("-m", "--modulation", type="choice", choices=list(mods.keys()),
                      default='psk',
                      help="Select modulation from: %s [default=%%default]"
                            % (', '.join(list(mods.keys())),))
    parser.add_option("", "--amplitude", type="eng_float", default=0.2,
                      help="set Tx amplitude (0-1) (default=%default)")
    parser.add_option("-r", "--bitrate", type="eng_float", default=250e3,
                      help="Select modulation bit rate (default=%default)")
    parser.add_option("-S", "--samples-per-symbol", type="float", default=2,
                      help="set samples/symbol [default=%default]")
    parser.add_option("", "--to-file", default=None,
                      help="Output file for modulated samples")
    if not parser.has_option("--verbose"):
        parser.add_option("-v", "--verbose",
                          action="store_true", default=False)
    if not parser.has_option("--log"):
        parser.add_option("", "--log", action="store_true", default=False)

    uhd_transmitter.add_options(parser)

    for mod in list(mods.values()):
        mod.add_options(parser)

    (options, args) = parser.parse_args()
    if len(args) != 0:
        parser.print_help()
        sys.exit(1)

    return (options, args)


if __name__ == "__main__":
    print("""Warning: this example in its current shape is deprecated and
            will be removed or fundamentally reworked in a coming GNU Radio
            release.""")
    mods = digital.modulation_utils.type_1_mods()

    (options, args) = get_options(mods)

    mod = mods[options.modulation]
    tb = tx_psk_block(mod, options)

    try:
        tb.run()
    except KeyboardInterrupt:
        pass

#          ====================          END          OF          FILE:          gr-
```

digital/examples/narrowband/digital_bert_tx.py ====================

```
# ==================== START OF FILE: gr-digital/examples/narrowband/uhd_interface.py ====================
#!/usr/bin/env python
#
# Copyright 2010,2011 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#


from gnuradio import gr, uhd
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

import sys


def add_freq_option(parser):
    """
    Hackery that has the -f / --freq option set both tx_freq and rx_freq
    """
    def freq_callback(option, opt_str, value, parser):
        parser.values.rx_freq = value
        parser.values.tx_freq = value

    if not parser.has_option('--freq'):
        parser.add_option('-f', '--freq', type="eng_float",
                          action="callback", callback=freq_callback,
                          help="set Tx and/or Rx frequency to FREQ [default=%default]",
                          metavar="FREQ")


class uhd_interface(object):
    def __init__(self, istx, args, sym_rate, sps, freq=None, lo_offset=None,
                 gain=None, spec=None, antenna=None, clock_source=None):

        if(istx):
            self.u = uhd.usrp_sink(
                device_addr=args, stream_args=uhd.stream_args('fc32'))
        else:
            self.u = uhd.usrp_source(
                device_addr=args, stream_args=uhd.stream_args('fc32'))

        # Set clock source
        if(clock_source):
            self.u.set_clock_source(clock_source, 0)

        # Set the subdevice spec
        if(spec):
            self.u.set_subdev_spec(spec, 0)

        # Set the antenna
        if(antenna):
            self.u.set_antenna(antenna, 0)

        self._args = args
        self._ant = antenna
        self._spec = spec
        self._gain = self.set_gain(gain)
        self._lo_offset = lo_offset
        self._freq = self.set_freq(freq, lo_offset)
        self._rate, self._sps = self.set_sample_rate(sym_rate, sps)
        self._clock_source = clock_source

    def set_sample_rate(self, sym_rate, req_sps):
        start_sps = req_sps
        while(True):
            asked_samp_rate = sym_rate * req_sps
            self.u.set_samp_rate(asked_samp_rate)
            actual_samp_rate = self.u.get_samp_rate()

            sps = actual_samp_rate / sym_rate
            if(sps < 2):
                req_sps += 1
            else:
                actual_sps = sps
                break

        if(sps != req_sps):
            print("\nSymbol Rate:         %f" % (sym_rate))
            print("Requested sps:       %f" % (start_sps))
```

```
                print("Given sample rate:      %f" % (actual_samp_rate))
                print("Actual sps for rate: %f" % (actual_sps))

            if(actual_samp_rate != asked_samp_rate):
                print("\nRequested sample rate: %f" % (asked_samp_rate))
                print("Actual sample rate: %f" % (actual_samp_rate))

            return (actual_samp_rate, actual_sps)

        def get_sample_rate(self):
            return self.u.get_samp_rate()

        def set_gain(self, gain=None):
            if gain is None:
                # if no gain was specified, use the mid-point in dB
                g = self.u.get_gain_range()
                gain = float(g.start() + g.stop()) / 2
                print("\nNo gain specified.")
                print("Setting gain to %f (from [%f, %f])" %
                      (gain, g.start(), g.stop()))

            self.u.set_gain(gain, 0)
            return gain

        def set_freq(self, freq=None, lo_offset=None):
            if(freq is None):
                sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
                sys.exit(1)

            r = self.u.set_center_freq(uhd.tune_request(freq, lo_offset))
            if r:
                return freq
            else:
                frange = self.u.get_freq_range()
                sys.stderr.write(("\nRequested frequency (%f) out or range [%f, %f]\n") %
                                 (freq, frange.start(), frange.stop()))
                sys.exit(1)

#-------------------------------------------------------------------#
#     TRANSMITTER
#-------------------------------------------------------------------#


class uhd_transmitter(uhd_interface, gr.hier_block2):
```

```
def __init__(self, args, sym_rate, sps, freq=None, lo_offset=None, gain=None,
                spec=None, antenna=None, clock_source=None, verbose=False):
    gr.hier_block2.__init__(self, "uhd_transmitter",
                            gr.io_signature(1, 1, gr.sizeof_gr_complex),
                            gr.io_signature(0, 0, 0))

    # Set up the UHD interface as a transmitter
    uhd_interface.__init__(self, True, args, sym_rate, sps,
                           freq, lo_offset, gain, spec, antenna, clock_source)

    self.connect(self, self.u)

    if(verbose):
        self._print_verbage()

@staticmethod
def add_options(parser):
    add_freq_option(parser)
    parser.add_option("-a", "--args", type="string", default="",
                      help="UHD device address args [default=%default]")
    parser.add_option("", "--spec", type="string", default=None,
                      help="Subdevice of UHD device where appropriate")
    parser.add_option("-A", "--antenna", type="string", default=None,
                      help="select Rx Antenna where appropriate")
    parser.add_option("", "--tx-freq", type="eng_float", default=None,
                      help="set transmit frequency to FREQ [default=%default]",
                      metavar="FREQ")
    parser.add_option("", "--lo-offset", type="eng_float", default=0,
                      help="set local oscillator offset in Hz (default is 0)")
    parser.add_option("", "--tx-gain", type="eng_float", default=None,
                      help="set transmit gain in dB (default is midpoint)")
    parser.add_option("-C", "--clock-source", type="string", default=None,
                      help="select clock source (e.g. 'external') [default=%default]")
    parser.add_option("-v", "--verbose",
                      action="store_true", default=False)

def _print_verbage(self):
    """
    Prints information about the UHD transmitter
    """
    print("\nUHD Transmitter:")
    print("Args:      %s" % (self._args))
    print("Freq:         %sHz" % (eng_notation.num_to_str(self._freq)))
    print("LO Offset:    %sHz" %
                (eng_notation.num_to_str(self._lo_offset)))
        print("Gain:            %f dB" % (self._gain))
        print("Sample Rate: %ssps" % (eng_notation.num_to_str(self._rate)))
        print("Antenna:         %s" % (self._ant))
        print("Subdev Spec:     %s" % (self._spec))
        print("Clock Source: %s" % (self._clock_source))

#-----------------------------------------------------------------#
#    RECEIVER
#-----------------------------------------------------------------#


class uhd_receiver(uhd_interface, gr.hier_block2):
    def __init__(self, args, sym_rate, sps, freq=None, lo_offset=None, gain=None,
                 spec=None, antenna=None, clock_source=None, verbose=False):
        gr.hier_block2.__init__(self, "uhd_receiver",
                                gr.io_signature(0, 0, 0),
                                gr.io_signature(1, 1, gr.sizeof_gr_complex))

        # Set up the UHD interface as a receiver
        uhd_interface.__init__(self, False, args, sym_rate, sps,
                               freq, lo_offset, gain, spec, antenna, clock_source)

        self.connect(self.u, self)

        if(verbose):
            self._print_verbage()

    @staticmethod
    def add_options(parser):
        add_freq_option(parser)
        parser.add_option("-a", "--args", type="string", default="",
                          help="UHD device address args [default=%default]")
        parser.add_option("", "--spec", type="string", default=None,
                          help="Subdevice of UHD device where appropriate")
        parser.add_option("-A", "--antenna", type="string", default=None,
                          help="select Rx Antenna where appropriate")
        parser.add_option("", "--rx-freq", type="eng_float", default=None,
                          help="set receive frequency to FREQ [default=%default]",
                          metavar="FREQ")
        parser.add_option("", "--lo-offset", type="eng_float", default=0,
                          help="set local oscillator offset in Hz (default is 0)")
        parser.add_option("", "--rx-gain", type="eng_float", default=None,
                          help="set receive gain in dB (default is midpoint)")
        parser.add_option("-C", "--clock-source", type="string", default=None,
                          help="select clock source (e.g. 'external') [default=%default]")
        if not parser.has_option("--verbose"):
            parser.add_option("-v", "--verbose",
                              action="store_true", default=False)

    def _print_verbage(self):
        """
        Prints information about the UHD transmitter
        """
        print("\nUHD Receiver:")
        print("UHD Args:     %s" % (self._args))
        print("Freq:         %sHz" % (eng_notation.num_to_str(self._freq)))
        print("LO Offset:    %sHz" %
              (eng_notation.num_to_str(self._lo_offset)))
        print("Gain:         %f dB" % (self._gain))
        print("Sample Rate:  %ssps" % (eng_notation.num_to_str(self._rate)))
        print("Antenna:      %s" % (self._ant))
        print("Spec:         %s" % (self._spec))
        print("Clock Source: %s" % (self._clock_source))

# ==================== END OF FILE: gr-digital/examples/narrowband/uhd_interface.py ====================


# ==================== START OF FILE: gr-digital/examples/ofdm/benchmark_add_channel.py ====================
#!/usr/bin/env python
#
# Copyright 2010,2011 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#


from gnuradio import gr, channels
from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser
```

```
import random
import math
import sys


class my_top_block(gr.top_block):
    def __init__(self, ifile, ofile, options):
        gr.top_block.__init__(self)

        SNR = 10.0**(options.snr / 10.0)
        time_offset = options.time_offset
        phase_offset = options.phase_offset * (math.pi / 180.0)

        # calculate noise voltage from SNR
        power_in_signal = abs(options.tx_amplitude)**2
        noise_power = power_in_signal / SNR
        noise_voltage = math.sqrt(noise_power)
        print("Noise voltage: ", noise_voltage)

        frequency_offset = options.frequency_offset / options.fft_length

        self.src = blocks.file_source(gr.sizeof_gr_complex, ifile)
        #self.throttle = blocks.throttle(gr.sizeof_gr_complex, options.sample_rate)

        self.channel = channels.channel_model(noise_voltage, frequency_offset,
                                              time_offset,          noise_seed=-
random.randint(0, 100000))
        self.phase = blocks.multiply_const_cc(complex(math.cos(phase_offset),
                                                      math.sin(phase_offset)))
        self.snk = blocks.file_sink(gr.sizeof_gr_complex, ofile)

        self.connect(self.src, self.channel, self.phase, self.snk)


# /////////////////////////////////////////////////////////////////////////////
#                                   main
# /////////////////////////////////////////////////////////////////////////////

def main():
    # Create Options Parser:
    usage = "benchmack_add_channel.py [options] <input file> <output file>"
    parser = OptionParser(
        usage=usage, option_class=eng_option, conflict_handler="resolve")
    parser.add_option("-n", "--snr", type="eng_float", default=30,
                              help="set the SNR of the channel in dB [default=%default]")
    parser.add_option("", "--seed", action="store_true", default=False,
                              help="use a random seed for AWGN noise [default=%default]")
    parser.add_option("-f", "--frequency-offset", type="eng_float", default=0,
                              help="set   frequency   offset   introduced   by   channel
[default=%default]")
    parser.add_option("-t", "--time-offset", type="eng_float", default=1.0,
                              help="set timing offset between Tx and Rx [default=%default]")
    parser.add_option("-p", "--phase-offset", type="eng_float", default=0,
                              help="set  phase  offset  (in  degrees)  between  Tx  and  Rx
[default=%default]")
    parser.add_option("-m", "--use-multipath", action="store_true", default=False,
                              help="Use a multipath channel [default=%default]")
    parser.add_option("", "--fft-length", type="intx", default=None,
                              help="set the number of FFT bins [default=%default]")
    parser.add_option("", "--tx-amplitude", type="eng_float",
                              default=1.0,
                              help="tell the simulator the signal amplitude [default=%default]")

    (options, args) = parser.parse_args()

    if len(args) != 2:
        parser.print_help(sys.stderr)
        sys.exit(1)

    if options.fft_length is None:
        sys.stderr.write("Please enter the FFT length of the OFDM signal.\n")
        sys.exit(1)

    ifile = args[0]
    ofile = args[1]

    # build the graph
    tb = my_top_block(ifile, ofile, options)

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

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


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

# ==================== END OF FILE: gr-digital/examples/ofdm/benchmark_add_channel.py ====================


# ==================== START OF FILE: gr-digital/examples/ofdm/receive_path.py ====================
#
# Copyright 2005,2006,2011 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import eng_notation
from gnuradio import digital
from gnuradio import analog

import copy
import sys

# /////////////////////////////////////////////////////////////////////////////
#                              receive path
# /////////////////////////////////////////////////////////////////////////////


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

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

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

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

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

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

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

        # Display some information about the setup
        if self._verbose:
            self._print_verbage()

    def carrier_sensed(self):
        """
        Return True if we think carrier is present.
        """
        # return self.probe.level() > X
        return self.probe.unmuted()

    def carrier_threshold(self):
        """
        Return current setting in dB.
        """
        return self.probe.threshold()

    def set_carrier_threshold(self, threshold_in_db):
        """
        Set carrier threshold.

        Args:
            threshold_in_db: set detection threshold (float (dB))
        """
        self.probe.set_threshold(threshold_in_db)

    @staticmethod
    def add_options(normal, expert):
        """
        Adds receiver-specific options to the Options Parser
        """
        normal.add_option("-W", "--bandwidth", type="eng_float",
                          default=500e3,
                          help="set symbol bandwidth [default=%default]")
        normal.add_option("-v", "--verbose",
                          action="store_true", default=False)
        expert.add_option("", "--log", action="store_true", default=False,
                          help="Log all parts of flow graph to files (CAUTION: lots of
data)")

    def _print_verbage(self):
        """
        Prints information about the receive path
        """
        pass

# ==================== END OF FILE: gr-digital/examples/ofdm/receive_path.py
====================


# ==================== START OF FILE: gr-digital/examples/ofdm/transmit_path.py
====================
#
# Copyright 2005,2006,2011 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#



from gnuradio import gr
from gnuradio import eng_notation
from gnuradio import blocks
from gnuradio import digital

import copy
import sys

# /////////////////////////////////////////////////////////////////////////////
#                              transmit path
# ////////////////////////////////////////////////////////////////////////////


class transmit_path(gr.hier_block2):
    def __init__(self, options):
        '''
        See below for what options should hold
        '''

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

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

        self._verbose = options.verbose          # turn verbose mode on/off
        self._tx_amplitude = options.tx_amplitude    # digital amp sent to radio

        self.ofdm_tx = digital.ofdm_mod(options,
                                        msgq_limit=4,
                                        pad_for_usrp=False)

        self.amp = blocks.multiply_const_cc(1)
        self.set_tx_amplitude(self._tx_amplitude)

        # Display some information about the setup
        if self._verbose:
            self._print_verbage()

        # Create and setup transmit path flow graph
        self.connect(self.ofdm_tx, self.amp, self)

    def set_tx_amplitude(self, ampl):
        """
        Sets the transmit amplitude sent to the USRP

        Args:
            : ampl 0 <= ampl < 1.0.    Try 0.10
        """
        self._tx_amplitude = max(0.0, min(ampl, 1))
        self.amp.set_k(self._tx_amplitude)

    def send_pkt(self, payload='', eof=False):
        """
        Calls the transmitter method to send a packet
        """
        return self.ofdm_tx.send_pkt(payload, eof)

    @staticmethod
    def add_options(normal, expert):
        """
        Adds transmitter-specific options to the Options Parser
        """
        normal.add_option("", "--tx-amplitude", type="eng_float",
                          default=0.1, metavar="AMPL",
                          help="set transmitter digital amplitude: 0 <= AMPL < 1.0
[default=%default]")
        normal.add_option("-W", "--bandwidth", type="eng_float",
                          default=500e3,
                          help="set symbol bandwidth [default=%default]")
        normal.add_option("-v", "--verbose", action="store_true",
                          default=False)
        expert.add_option("", "--log", action="store_true",
                          default=False,
                          help="Log all parts of flow graph to file (CAUTION: lots of
data)")

    def _print_verbage(self):
        """
        Prints information about the transmit path
        """
        print("Tx amplitude     %s" % (self._tx_amplitude))

# ==================== END OF FILE: gr-digital/examples/ofdm/transmit_path.py
====================


# ==================== START OF FILE: gr-digital/examples/ofdm/uhd_interface.py
====================
#!/usr/bin/env python
#
# Copyright 2010,2011 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
```

```
#


from gnuradio import gr, uhd
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

import sys


def add_freq_option(parser):
    """
    Hackery that has the -f / --freq option set both tx_freq and rx_freq
    """
    def freq_callback(option, opt_str, value, parser):
        parser.values.rx_freq = value
        parser.values.tx_freq = value

    if not parser.has_option('--freq'):
        parser.add_option('-f', '--freq', type="eng_float",
                          action="callback", callback=freq_callback,
                          help="set Tx and/or Rx frequency to FREQ [default=%default]",
                          metavar="FREQ")


class uhd_interface(object):
    def __init__(self, istx, args, bandwidth, freq=None, lo_offset=None,
                 gain=None, spec=None, antenna=None, clock_source=None):

        if(istx):
            self.u = uhd.usrp_sink(
                device_addr=args, stream_args=uhd.stream_args('fc32'))
        else:
            self.u = uhd.usrp_source(
                device_addr=args, stream_args=uhd.stream_args('fc32'))

        # Set clock source to external.
        if(clock_source):
            self.u.set_clock_source(clock_source, 0)

        # Set the subdevice spec
        if(spec):
            self.u.set_subdev_spec(spec, 0)
        # Set the antenna
        if(antenna):
            self.u.set_antenna(antenna, 0)

        self._args = args
        self._ant = antenna
        self._spec = spec
        self._gain = self.set_gain(gain)
        self._lo_offset = lo_offset
        self._freq = self.set_freq(freq, lo_offset)
        self._rate = self.set_sample_rate(bandwidth)
        self._clock_source = clock_source

    def set_sample_rate(self, bandwidth):
        self.u.set_samp_rate(bandwidth)
        actual_bw = self.u.get_samp_rate()

        return actual_bw

    def get_sample_rate(self):
        return self.u.get_samp_rate()

    def set_gain(self, gain=None):
        if gain is None:
            # if no gain was specified, use the mid-point in dB
            g = self.u.get_gain_range()
            gain = float(g.start() + g.stop()) / 2
            print("\nNo gain specified.")
            print("Setting gain to %f (from [%f, %f])" %
                  (gain, g.start(), g.stop()))

        self.u.set_gain(gain, 0)
        return gain

    def set_freq(self, freq=None, lo_offset=None):
        if(freq is None):
            sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
            sys.exit(1)

        r = self.u.set_center_freq(uhd.tune_request(freq, lo_offset))

        if r:
            return freq
        else:
            frange = self.u.get_freq_range()
            sys.stderr.write(("\nRequested frequency (%f) out or range [%f, %f]\n") %
                             (freq, frange.start(), frange.stop()))
            sys.exit(1)

#--------------------------------------------------------------------#
#   TRANSMITTER
#--------------------------------------------------------------------#


class uhd_transmitter(uhd_interface, gr.hier_block2):
    def __init__(self, args, bandwidth, freq=None, lo_offset=None, gain=None,
                 spec=None, antenna=None, clock_source=None, verbose=False):
        gr.hier_block2.__init__(self, "uhd_transmitter",
                                gr.io_signature(1, 1, gr.sizeof_gr_complex),
                                gr.io_signature(0, 0, 0))

        # Set up the UHD interface as a transmitter
        uhd_interface.__init__(self, True, args, bandwidth,
                               freq, lo_offset, gain, spec, antenna, clock_source)

        self.connect(self, self.u)

        if(verbose):
            self._print_verbage()

    @staticmethod
    def add_options(parser):
        add_freq_option(parser)
        parser.add_option("-a", "--args", type="string", default="",
                          help="UHD device address args [default=%default]")
        parser.add_option("", "--spec", type="string", default=None,
                          help="Subdevice of UHD device where appropriate")
        parser.add_option("-A", "--antenna", type="string", default=None,
                          help="select Rx Antenna where appropriate")
        parser.add_option("", "--tx-freq", type="eng_float", default=None,
                          help="set transmit frequency to FREQ [default=%default]",
                          metavar="FREQ")
        parser.add_option("", "--lo-offset", type="eng_float", default=0,
                          help="set local oscillator offset in Hz (default is 0)")
        parser.add_option("", "--tx-gain", type="eng_float", default=None,
                          help="set transmit gain in dB (default is midpoint)")
        parser.add_option("-C", "--clock-source", type="string", default=None,
                              help="select clock source (e.g. 'external') [default=%default]")
        parser.add_option("-v", "--verbose",
                              action="store_true", default=False)

    def _print_verbage(self):
        """
        Prints information about the UHD transmitter
        """
        print("\nUHD Transmitter:")
        print("UHD Args:        %s" % (self._args))
        print("Freq:            %sHz" % (eng_notation.num_to_str(self._freq)))
        print("LO Offset:       %sHz" %
              (eng_notation.num_to_str(self._lo_offset)))
        print("Gain:            %f dB" % (self._gain))
        print("Sample Rate:   %ssps" % (eng_notation.num_to_str(self._rate)))
        print("Antenna:         %s" % (self._ant))
        print("Subdev Sec:      %s" % (self._spec))
        print("Clock Source: %s" % (self._clock_source))


#--------------------------------------------------------------------#
#   RECEIVER
#--------------------------------------------------------------------#


class uhd_receiver(uhd_interface, gr.hier_block2):
    def __init__(self, args, bandwidth, freq=None, lo_offset=None, gain=None,
                 spec=None, antenna=None, clock_source=None, verbose=False):
        gr.hier_block2.__init__(self, "uhd_receiver",
                                gr.io_signature(0, 0, 0),
                                gr.io_signature(1, 1, gr.sizeof_gr_complex))

        # Set up the UHD interface as a receiver
        uhd_interface.__init__(self, False, args, bandwidth,
                               freq, lo_offset, gain, spec, antenna, clock_source)

        self.connect(self.u, self)

        if(verbose):
            self._print_verbage()

    @staticmethod
    def add_options(parser):
        add_freq_option(parser)
        parser.add_option("-a", "--args", type="string", default="",
                          help="UHD device address args [default=%default]")
        parser.add_option("", "--spec", type="string", default=None,
                          help="Subdevice of UHD device where appropriate")
        parser.add_option("-A", "--antenna", type="string", default=None,
                          help="select Rx Antenna where appropriate")
        parser.add_option("", "--rx-freq", type="eng_float", default=None,
                          help="set receive frequency to FREQ [default=%default]",
                          metavar="FREQ")
        parser.add_option("", "--lo-offset", type="eng_float", default=0,
                          help="set local oscillator offset in Hz (default is 0)")
        parser.add_option("", "--rx-gain", type="eng_float", default=None,
                          help="set receive gain in dB (default is midpoint)")
        parser.add_option("-C", "--clock-source", type="string", default=None,
                          help="select clock source (e.g. 'external') [default=%default]")
        if not parser.has_option("--verbose"):
            parser.add_option("-v", "--verbose",
                              action="store_true", default=False)

    def _print_verbage(self):
        """
        Prints information about the UHD transmitter
        """
        print("\nUHD Receiver:")
        print("UHD Args:     %s" % (self._args))
        print("Freq:         %sHz" % (eng_notation.num_to_str(self._freq)))
        print("LO Offset:    %sHz" %
              (eng_notation.num_to_str(self._lo_offset)))
        print("Gain:         %f dB" % (self._gain))
        print("Sample Rate:  %ssps" % (eng_notation.num_to_str(self._rate)))
        print("Antenna:      %s" % (self._ant))
        print("Subdev Sec:   %s" % (self._spec))
        print("Clock Source: %s" % (self._clock_source))

# ==================== END OF FILE: gr-digital/examples/ofdm/uhd_interface.py ====================


# ==================== START OF FILE: gr-dtv/examples/atsc_ctrlport_monitor.py ====================
#!/usr/bin/env python
#
# Copyright 2015 Free Software Foundation
#
```

```
# SPDX-License-Identifier: GPL-3.0-or-later
#


import numpy
from gnuradio.ctrlport.GNURadioControlPortClient import (
    GNURadioControlPortClient, TTransportException,
)
import matplotlib.animation as animation
import matplotlib.pyplot as plt
import sys
import matplotlib
matplotlib.use("QT4Agg")

"""
If a host is running the ATSC receiver chain with ControlPort
turned on, this script will connect to the host using the hostname and
port pair of the ControlPort instance and display metrics of the
receiver. The ATSC publishes information about the success of the
Reed-Solomon decoder and Viterbi metrics for use here in displaying
the link quality. This also gets the equalizer taps of the receiver
and displays the frequency response.
"""


class atsc_ctrlport_monitor(object):
    def __init__(self, host, port):
        argv = [None, host, port]
        radiosys = GNURadioControlPortClient(argv=argv, rpcmethod='thrift')
        self.radio = radiosys.client
        print(self.radio)

        vt_init_key = 'dtv_atsc_viterbi_decoder0::decoder_metrics'
        data = self.radio.getKnobs([vt_init_key])[vt_init_key]
        init_metric = numpy.mean(data.value)
        self._viterbi_metric = 100 * [init_metric, ]

        table_col_labels = ('Num Packets', 'Error Rate', 'Packet Error Rate',
                            'Viterbi Metric', 'SNR')

        self._fig = plt.figure(1, figsize=(12, 12), facecolor='w')
        self._sp0 = self._fig.add_subplot(4, 1, 1)
        self._sp1 = self._fig.add_subplot(4, 1, 2)
        self._sp2 = self._fig.add_subplot(4, 1, 3)
        self._plot_taps = self._sp0.plot([], [], 'k', linewidth=2)
        self._plot_psd = self._sp1.plot([], [], 'k', linewidth=2)
        self._plot_data = self._sp2.plot(
            [], [], 'ok', linewidth=2, markersize=4, alpha=0.05)

        self._ax2 = self._fig.add_subplot(4, 1, 4)
        self._table = self._ax2.table(cellText=[len(table_col_labels) * ['0']],
                                      colLabels=table_col_labels,
                                      loc='center')
        self._ax2.axis('off')
        cells = self._table.properties()['child_artists']
        for c in cells:
            c.set_lw(0.1)    # set's line width
            c.set_ls('solid')
            c.set_height(0.2)

        ani = animation.FuncAnimation(self._fig, self.update_data, frames=200,
                                      fargs=(self._plot_taps[0], self._plot_psd[0],
                                             self._plot_data[0], self._table),
                                      init_func=self.init_function,
                                      blit=True)
        plt.show()

    def update_data(self, x, taps, psd, syms, table):
        try:
            eqdata_key = 'dtv_atsc_equalizer0::taps'
            symdata_key = 'dtv_atsc_equalizer0::data'
            rs_nump_key = 'dtv_atsc_rs_decoder0::num_packets'
            rs_numbp_key = 'dtv_atsc_rs_decoder0::num_bad_packets'
            rs_numerrs_key = 'dtv_atsc_rs_decoder0::num_errors_corrected'
            vt_metrics_key = 'dtv_atsc_viterbi_decoder0::decoder_metrics'
            snr_key = 'probe2_f0::SNR'

            data = self.radio.getKnobs([])
            eqdata = data[eqdata_key]
            symdata = data[symdata_key]
            rs_num_packets = data[rs_nump_key]
            rs_num_bad_packets = data[rs_numbp_key]
            rs_num_errors_corrected = data[rs_numerrs_key]
            vt_decoder_metrics = data[vt_metrics_key]
            snr_est = data[snr_key]

            vt_decoder_metrics = numpy.mean(vt_decoder_metrics.value)
            self._viterbi_metric.pop()
            self._viterbi_metric.insert(0, vt_decoder_metrics)

        except TTransportException:
            sys.stderr.write("Lost connection, exiting")
            sys.exit(1)

        ntaps = len(eqdata.value)
        taps.set_ydata(eqdata.value)
        taps.set_xdata(list(range(ntaps)))
        self._sp0.set_xlim(0, ntaps)
        self._sp0.set_ylim(min(eqdata.value), max(eqdata.value))

        fs = 6.25e6
        freq = numpy.linspace(-fs / 2, fs / 2, 10000)
        H = numpy.fft.fftshift(numpy.fft.fft(eqdata.value, 10000))
        HdB = 20.0 * numpy.log10(abs(H))
        psd.set_ydata(HdB)
        psd.set_xdata(freq)
        self._sp1.set_xlim(0, fs / 2)
        self._sp1.set_ylim([min(HdB), max(HdB)])
        self._sp1.set_yticks([min(HdB), max(HdB)])
        self._sp1.set_yticklabels(["min", "max"])

        nsyms = len(symdata.value)
        syms.set_ydata(symdata.value)
        syms.set_xdata(nsyms * [0, ])
        self._sp2.set_xlim([-1, 1])
        self._sp2.set_ylim([-10, 10])

        per = float(rs_num_bad_packets.value) / float(rs_num_packets.value)
        ber = float(rs_num_errors_corrected.value) / \
            float(187 * rs_num_packets.value)

        table._cells[(1, 0)]._text.set_text("{0}".format(rs_num_packets.value))
        table._cells[(1, 1)]._text.set_text("{0:.2g}".format(ber))
        table._cells[(1, 2)]._text.set_text("{0:.2g}".format(per))
        table._cells[(1, 3)]._text.set_text(
            "{0:.1f}".format(numpy.mean(self._viterbi_metric)))
        table._cells[(1, 4)]._text.set_text("{0:.4f}".format(snr_est.value[0]))

        return (taps, psd, syms, table)

    def init_function(self):
        return self._plot_taps + self._plot_psd + self._plot_data
```

```
if __name__ == "__main__":
    host = sys.argv[1]
    port = sys.argv[2]
    m = atsc_ctrlport_monitor(host, port)

# ==================== END OF FILE: gr-dtv/examples/atsc_ctrlport_monitor.py
====================


# ==================== START OF FILE: gr-filter/examples/benchmark_filters.py
====================
#!/usr/bin/env python
#
# Copyright 2005-2007,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

import time
import random
from argparse import ArgumentParser
from gnuradio import gr
from gnuradio import blocks, filter
from gnuradio.eng_arg import eng_float, intx


def make_random_complex_tuple(L):
    result = []
    for x in range(L):
        result.append(complex(random.uniform(-1000, 1000),
                              random.uniform(-1000, 1000)))
    return tuple(result)


def benchmark(name, creator, dec, ntaps, total_test_size, block_size):
    block_size = 32768

    tb = gr.top_block()
    taps = make_random_complex_tuple(ntaps)
```

```
        src = blocks.vector_source_c(make_random_complex_tuple(block_size), True)
        head = blocks.head(gr.sizeof_gr_complex, int(total_test_size))
        op = creator(dec, taps)
        dst = blocks.null_sink(gr.sizeof_gr_complex)
        tb.connect(src, head, op, dst)
        start = time.time()
        tb.run()
        stop = time.time()
        delta = stop - start
        print("%16s: taps: %4d    input: %4g, time: %6.3f    taps/sec: %10.4g" % (
                name, ntaps, total_test_size, delta, ntaps * total_test_size / delta))


def main():
        parser = ArgumentParser()
        parser.add_argument("-n", "--ntaps", type=int, default=256)
        parser.add_argument("-t", "--total-input-size",
                                        type=eng_float, default=40e6)
        parser.add_argument("-b", "--block-size", type=intx, default=50000)
        parser.add_argument("-d", "--decimation", type=int, default=1)
        args = parser.parse_args()

        benchmark("filter.fir_filter_ccc", filter.fir_filter_ccc, args.decimation,
                        args.ntaps, args.total_input_size, args.block_size)
        benchmark("filter.fft_filter_ccc", filter.fft_filter_ccc, args.decimation,
                        args.ntaps, args.total_input_size, args.block_size)


if __name__ == '__main__':
        main()

# ==================== END OF FILE: gr-filter/examples/benchmark_filters.py
====================


# ==================== START OF FILE: gr-filter/examples/channelize.py
====================
#!/usr/bin/env python
#
# Copyright 2009,2012,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
```

```
#
#

from gnuradio import gr
from gnuradio import blocks
from gnuradio import filter
from gnuradio.fft import window
import sys
import time
import numpy

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)

try:
    import pylab
    from pylab import mlab
except ImportError:
    sys.stderr.write(
        "Error: Program requires matplotlib (see: matplotlib.sourceforge.net).\n")
    sys.exit(1)


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

        self._N = 2000000             # number of samples to use
        self._fs = 1000               # initial sampling rate
        self._M = M = 9               # Number of channels to channelize
        self._ifs = M * self._fs      # initial sampling rate

        # Create a set of taps for the PFB channelizer
        self._taps = filter.firdes.low_pass_2(1, self._ifs, 475.50, 50,
                                              attenuation_dB=100,

window=window.WIN_BLACKMAN_hARRIS)

        # Calculate the number of taps per channel for our own information
        tpc = numpy.ceil(float(len(self._taps)) / float(self._M))
        print("Number of taps:        ", len(self._taps))
        print("Number of channels: ", self._M)
        print("Taps per channel:     ", tpc)

        # Create a set of signals at different frequencies
        #    freqs lists the frequencies of the signals that get stored
        #    in the list "signals", which then get summed together
        self.signals = list()
        self.add = blocks.add_cc()
        freqs = [-70, -50, -30, -10, 10, 20, 40, 60, 80]
        for i in range(len(freqs)):
            f = freqs[i] + (M / 2 - M + i + 1) * self._fs
            self.signals.append(analog.sig_source_c(
                self._ifs, analog.GR_SIN_WAVE, f, 1))
            self.connect(self.signals[i], (self.add, i))

        self.head = blocks.head(gr.sizeof_gr_complex, self._N)

        # Construct the channelizer filter
        self.pfb = filter.pfb.channelizer_ccf(self._M, self._taps, 1)

        # Construct a vector sink for the input signal to the channelizer
        self.snk_i = blocks.vector_sink_c()

        # Connect the blocks
        self.connect(self.add, self.head, self.pfb)
        self.connect(self.add, self.snk_i)

        # Use this to play with the channel mapping
        # self.pfb.set_channel_map([5,6,7,8,0,1,2,3,4])

        # Create a vector sink for each of M output channels of the filter and connect it
        self.snks = list()
        for i in range(self._M):
            self.snks.append(blocks.vector_sink_c())
            self.connect((self.pfb, i), self.snks[i])


def main():
    tstart = time.time()

    tb = pfb_top_block()
    tb.run()

    tend = time.time()
```

```
print("Run time: %f" % (tend - tstart))

if 1:
        fig_in = pylab.figure(1, figsize=(16, 9), facecolor="w")
        fig1 = pylab.figure(2, figsize=(16, 9), facecolor="w")
        fig2 = pylab.figure(3, figsize=(16, 9), facecolor="w")

        Ns = 1000
        Ne = 10000

        fftlen = 8192
        winfunc = numpy.blackman
        fs = tb._ifs

        # Plot the input signal on its own figure
        d = tb.snk_i.data()[Ns:Ne]
        spin_f = fig_in.add_subplot(2, 1, 1)

        X, freq = mlab.psd(d, NFFT=fftlen, noverlap=fftlen / 4, Fs=fs,
                                        window=lambda d: d * winfunc(fftlen),
                                        scale_by_freq=True)
        X_in = 10.0 * numpy.log10(abs(X))
        f_in = numpy.arange(-fs / 2.0, fs / 2.0, fs / float(X_in.size))
        pin_f = spin_f.plot(f_in, X_in, "b")
        spin_f.set_xlim([min(f_in), max(f_in) + 1])
        spin_f.set_ylim([-200.0, 50.0])

        spin_f.set_title("Input Signal", weight="bold")
        spin_f.set_xlabel("Frequency (Hz)")
        spin_f.set_ylabel("Power (dBW)")

        Ts = 1.0 / fs
        Tmax = len(d) * Ts

        t_in = numpy.arange(0, Tmax, Ts)
        x_in = numpy.array(d)
        spin_t = fig_in.add_subplot(2, 1, 2)
        pin_t = spin_t.plot(t_in, x_in.real, "b")
        pin_t = spin_t.plot(t_in, x_in.imag, "r")

        spin_t.set_xlabel("Time (s)")
        spin_t.set_ylabel("Amplitude")

        Ncols = int(numpy.floor(numpy.sqrt(tb._M)))
```

```
Nrows = int(numpy.floor(tb._M / Ncols))
if(tb._M % Ncols != 0):
        Nrows += 1

# Plot each of the channels outputs. Frequencies on Figure 2 and
# time signals on Figure 3
fs_o = tb._fs
Ts_o = 1.0 / fs_o
Tmax_o = len(d) * Ts_o
for i in range(len(tb.snks)):
        # remove issues with the transients at the beginning
        # also remove some corruption at the end of the stream
        #       this is a bug, probably due to the corner cases
        d = tb.snks[i].data()[Ns:Ne]

        sp1_f = fig1.add_subplot(Nrows, Ncols, 1 + i)
        X, freq = mlab.psd(d, NFFT=fftlen, noverlap=fftlen / 4, Fs=fs_o,
                                window=lambda d: d * winfunc(fftlen),
                                scale_by_freq=True)
        X_o = 10.0 * numpy.log10(abs(X))
        f_o = numpy.arange(-fs_o / 2.0, fs_o / 2.0, fs_o / float(X_o.size))
        p2_f = sp1_f.plot(f_o, X_o, "b")
        sp1_f.set_xlim([min(f_o), max(f_o) + 1])
        sp1_f.set_ylim([-200.0, 50.0])

        sp1_f.set_title(("Channel %d" % i), weight="bold")
        sp1_f.set_xlabel("Frequency (Hz)")
        sp1_f.set_ylabel("Power (dBW)")

        x_o = numpy.array(d)
        t_o = numpy.arange(0, Tmax_o, Ts_o)
        sp2_o = fig2.add_subplot(Nrows, Ncols, 1 + i)
        p2_o = sp2_o.plot(t_o, x_o.real, "b")
        p2_o = sp2_o.plot(t_o, x_o.imag, "r")
        sp2_o.set_xlim([min(t_o), max(t_o) + 1])
        sp2_o.set_ylim([-2, 2])

        sp2_o.set_title(("Channel %d" % i), weight="bold")
        sp2_o.set_xlabel("Time (s)")
        sp2_o.set_ylabel("Amplitude")

pylab.show()
```

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

# ==================== END OF FILE: gr-filter/examples/channelize.py ====================


# ==================== START OF FILE: gr-filter/examples/chirp_channelize.py ====================
#!/usr/bin/env python
#
# Copyright 2009,2012,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import blocks
from gnuradio import filter
from gnuradio.fft import window
import sys
import time
import numpy

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)

try:
    import pylab
    from pylab import mlab
except ImportError:
    sys.stderr.write(
        "Error: Program requires matplotlib (see: matplotlib.sourceforge.net).\n")
    sys.exit(1)
```

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

        self._N = 200000                # number of samples to use
        self._fs = 9000               # initial sampling rate
        self._M = 9                     # Number of channels to channelize

        # Create a set of taps for the PFB channelizer
        self._taps = filter.firdes.low_pass_2(1, self._fs, 500, 20,
                                                    attenuation_dB=10,

window=window.WIN_BLACKMAN_hARRIS)

        # Calculate the number of taps per channel for our own information
        tpc = numpy.ceil(float(len(self._taps)) / float(self._M))
        print("Number of taps:         ", len(self._taps))
        print("Number of channels: ", self._M)
        print("Taps per channel:      ", tpc)

        repeated = True
        if(repeated):
            self.vco_input = analog.sig_source_f(
                self._fs, analog.GR_SIN_WAVE, 0.25, 110)
        else:
            amp = 100
            data = numpy.arange(0, amp, amp / float(self._N))
            self.vco_input = blocks.vector_source_f(data, False)

        # Build a VCO controlled by either the sinusoid or single chirp tone
        # Then convert this to a complex signal
        self.vco = blocks.vco_f(self._fs, 225, 1)
        self.f2c = blocks.float_to_complex()

        self.head = blocks.head(gr.sizeof_gr_complex, self._N)

        # Construct the channelizer filter
        self.pfb = filter.pfb.channelizer_ccf(self._M, self._taps)

        # Construct a vector sink for the input signal to the channelizer
        self.snk_i = blocks.vector_sink_c()

        # Connect the blocks
        self.connect(self.vco_input, self.vco, self.f2c)
        self.connect(self.f2c, self.head, self.pfb)
        self.connect(self.f2c, self.snk_i)

        # Create a vector sink for each of M output channels of the filter and connect it
        self.snks = list()
        for i in range(self._M):
            self.snks.append(blocks.vector_sink_c())
            self.connect((self.pfb, i), self.snks[i])


def main():
    tstart = time.time()

    tb = pfb_top_block()
    tb.run()

    tend = time.time()
    print("Run time: %f" % (tend - tstart))

    if 1:
        fig_in = pylab.figure(1, figsize=(16, 9), facecolor="w")
        fig1 = pylab.figure(2, figsize=(16, 9), facecolor="w")
        fig2 = pylab.figure(3, figsize=(16, 9), facecolor="w")
        fig3 = pylab.figure(4, figsize=(16, 9), facecolor="w")

        Ns = 650
        Ne = 20000

        fftlen = 8192
        winfunc = numpy.blackman
        fs = tb._fs

        # Plot the input signal on its own figure
        d = tb.snk_i.data()[Ns:Ne]
        spin_f = fig_in.add_subplot(2, 1, 1)

        X, freq = mlab.psd(d, NFFT=fftlen, noverlap=fftlen / 4, Fs=fs,
                           window=lambda d: d * winfunc(fftlen),
                           scale_by_freq=True)
        X_in = 10.0 * numpy.log10(abs(numpy.fft.fftshift(X)))
        f_in = numpy.arange(-fs / 2.0, fs / 2.0, fs / float(X_in.size))
        pin_f = spin_f.plot(f_in, X_in, "b")
        spin_f.set_xlim([min(f_in), max(f_in) + 1])
        spin_f.set_ylim([-200.0, 50.0])

        spin_f.set_title("Input Signal", weight="bold")
        spin_f.set_xlabel("Frequency (Hz)")
        spin_f.set_ylabel("Power (dBW)")

        Ts = 1.0 / fs
        Tmax = len(d) * Ts

        t_in = numpy.arange(0, Tmax, Ts)
        x_in = numpy.array(d)
        spin_t = fig_in.add_subplot(2, 1, 2)
        pin_t = spin_t.plot(t_in, x_in.real, "b")
        pin_t = spin_t.plot(t_in, x_in.imag, "r")

        spin_t.set_xlabel("Time (s)")
        spin_t.set_ylabel("Amplitude")

        Ncols = int(numpy.floor(numpy.sqrt(tb._M)))
        Nrows = int(numpy.floor(tb._M / Ncols))
        if(tb._M % Ncols != 0):
            Nrows += 1

        # Plot each of the channels outputs. Frequencies on Figure 2 and
        # time signals on Figure 3
        fs_o = tb._fs / tb._M
        Ts_o = 1.0 / fs_o
        Tmax_o = len(d) * Ts_o
        for i in range(len(tb.snks)):
            # remove issues with the transients at the beginning
            # also remove some corruption at the end of the stream
            #       this is a bug, probably due to the corner cases
            d = tb.snks[i].data()[Ns:Ne]

            sp1_f = fig1.add_subplot(Nrows, Ncols, 1 + i)
            X, freq = mlab.psd(d, NFFT=fftlen, noverlap=fftlen / 4, Fs=fs_o,
                               window=lambda d: d * winfunc(fftlen),
                               scale_by_freq=True)
            X_o = 10.0 * numpy.log10(abs(X))
            f_o = freq
            p2_f = sp1_f.plot(f_o, X_o, "b")
            sp1_f.set_xlim([min(f_o), max(f_o) + 1])
            sp1_f.set_ylim([-200.0, 50.0])
```

```
            sp1_f.set_title(("Channel %d" % i), weight="bold")
            sp1_f.set_xlabel("Frequency (Hz)")
            sp1_f.set_ylabel("Power (dBW)")

            x_o = numpy.array(d)
            t_o = numpy.arange(0, Tmax_o, Ts_o)
            sp2_o = fig2.add_subplot(Nrows, Ncols, 1 + i)
            p2_o = sp2_o.plot(t_o, x_o.real, "b")
            p2_o = sp2_o.plot(t_o, x_o.imag, "r")
            sp2_o.set_xlim([min(t_o), max(t_o) + 1])
            sp2_o.set_ylim([-2, 2])

            sp2_o.set_title(("Channel %d" % i), weight="bold")
            sp2_o.set_xlabel("Time (s)")
            sp2_o.set_ylabel("Amplitude")

            sp3 = fig3.add_subplot(1, 1, 1)
            p3 = sp3.plot(t_o, x_o.real)
            sp3.set_xlim([min(t_o), max(t_o) + 1])
            sp3.set_ylim([-2, 2])

        sp3.set_title("All Channels")
        sp3.set_xlabel("Time (s)")
        sp3.set_ylabel("Amplitude")

        pylab.show()


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

# ==================== END OF FILE: gr-filter/examples/chirp_channelize.py ====================


# ==================== START OF FILE: gr-filter/examples/decimate.py ====================
#!/usr/bin/env python
#
# Copyright 2009,2012,2013 Free Software Foundation, Inc.
#
```

```
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import blocks
from gnuradio import filter
from gnuradio.fft import window
import sys
import time
import numpy

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)


try:
    from matplotlib import pyplot
    from matplotlib import pyplot as mlab
except ImportError:
    sys.stderr.write(
        "Error: Program requires matplotlib (see: matplotlib.sourceforge.net).\n")
    sys.exit(1)


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

        self._N = 10000000          # number of samples to use
        self._fs = 10000            # initial sampling rate
        self._decim = 20            # Decimation rate

        # Generate the prototype filter taps for the decimators with a 200 Hz bandwidth
        self._taps = filter.firdes.low_pass_2(1, self._fs,
                                              200, 150,
                                              attenuation_dB=120,

window=window.WIN_BLACKMAN_hARRIS)
```

```
        # Calculate the number of taps per channel for our own information
        tpc = numpy.ceil(float(len(self._taps)) / float(self._decim))
        print("Number of taps:          ", len(self._taps))
        print("Number of filters:    ", self._decim)
        print("Taps per channel:      ", tpc)

        # Build the input signal source
        # We create a list of freqs, and a sine wave is generated and added to the source
        # for each one of these frequencies.
        self.signals = list()
        self.add = blocks.add_cc()
        freqs = [10, 20, 2040]
        for i in range(len(freqs)):
            self.signals.append(analog.sig_source_c(
                self._fs, analog.GR_SIN_WAVE, freqs[i], 1))
            self.connect(self.signals[i], (self.add, i))

        self.head = blocks.head(gr.sizeof_gr_complex, self._N)

        # Construct a PFB decimator filter
        self.pfb = filter.pfb.decimator_ccf(self._decim, self._taps, 0)

        # Construct a standard FIR decimating filter
        self.dec = filter.fir_filter_ccf(self._decim, self._taps)

        self.snk_i = blocks.vector_sink_c()

        # Connect the blocks
        self.connect(self.add, self.head, self.pfb)
        self.connect(self.add, self.snk_i)

        # Create the sink for the decimated siganl
        self.snk = blocks.vector_sink_c()
        self.connect(self.pfb, self.snk)


def main():
    tb = pfb_top_block()

    tstart = time.time()
    tb.run()
    tend = time.time()
    print("Run time: %f" % (tend - tstart))
```

```
if 1:
    fig1 = pyplot.figure(1, figsize=(16, 9))
    fig2 = pyplot.figure(2, figsize=(16, 9))

    Ns = 10000
    Ne = 10000

    fftlen = 8192
    winfunc = numpy.blackman
    fs = tb._fs

    # Plot the input to the decimator

    d = tb.snk_i.data()[Ns:Ns + Ne]
    sp1_f = fig1.add_subplot(2, 1, 1)

    X, freq = mlab.psd(d, NFFT=fftlen, noverlap=fftlen / 4, Fs=fs,
                       window=lambda d: d * winfunc(fftlen),
                       scale_by_freq=True)
    X_in = 10.0 * numpy.log10(abs(numpy.fft.fftshift(X)))
    f_in = numpy.arange(-fs / 2.0, fs / 2.0, fs / float(X_in.size))
    p1_f = sp1_f.plot(f_in, X_in, "b")
    sp1_f.set_xlim([min(f_in), max(f_in) + 1])
    sp1_f.set_ylim([-200.0, 50.0])

    sp1_f.set_title("Input Signal", weight="bold")
    sp1_f.set_xlabel("Frequency (Hz)")
    sp1_f.set_ylabel("Power (dBW)")

    Ts = 1.0 / fs
    Tmax = len(d) * Ts

    t_in = numpy.arange(0, Tmax, Ts)
    x_in = numpy.array(d)
    sp1_t = fig1.add_subplot(2, 1, 2)
    p1_t = sp1_t.plot(t_in, x_in.real, "b")
    p1_t = sp1_t.plot(t_in, x_in.imag, "r")
    sp1_t.set_ylim([-tb._decim * 1.1, tb._decim * 1.1])

    sp1_t.set_xlabel("Time (s)")
    sp1_t.set_ylabel("Amplitude")

    # Plot the output of the decimator
```

```
        fs_o = tb._fs / tb._decim

        sp2_f = fig2.add_subplot(2, 1, 1)
        d = tb.snk.data()[Ns:Ns + Ne]
        X, freq = mlab.psd(d, NFFT=fftlen, noverlap=fftlen / 4, Fs=fs_o,
                           window=lambda d: d * winfunc(fftlen),
                           scale_by_freq=True)
        X_o = 10.0 * numpy.log10(abs(numpy.fft.fftshift(X)))
        f_o = numpy.arange(-fs_o / 2.0, fs_o / 2.0, fs_o / float(X_o.size))
        p2_f = sp2_f.plot(f_o, X_o, "b")
        sp2_f.set_xlim([min(f_o), max(f_o) + 1])
        sp2_f.set_ylim([-200.0, 50.0])

        sp2_f.set_title("PFB Decimated Signal", weight="bold")
        sp2_f.set_xlabel("Frequency (Hz)")
        sp2_f.set_ylabel("Power (dBW)")

        Ts_o = 1.0 / fs_o
        Tmax_o = len(d) * Ts_o

        x_o = numpy.array(d)
        t_o = numpy.arange(0, Tmax_o, Ts_o)
        sp2_t = fig2.add_subplot(2, 1, 2)
        p2_t = sp2_t.plot(t_o, x_o.real, "b-o")
        p2_t = sp2_t.plot(t_o, x_o.imag, "r-o")
        sp2_t.set_ylim([-2.5, 2.5])

        sp2_t.set_xlabel("Time (s)")
        sp2_t.set_ylabel("Amplitude")

        pyplot.show()


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

# ==================== END OF FILE: gr-filter/examples/decimate.py ====================


# ==================== START OF FILE: gr-filter/examples/fft_filter_ccc.py
```

====================

```
#!/usr/bin/env python
#
# Copyright 2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr, filter
from gnuradio import analog
from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio.eng_arg import eng_float, intx
from argparse import ArgumentParser
import sys
import numpy

try:
	from matplotlib import pyplot
except ImportError:
	print("Error:	could	not	from	matplotlib	import	pyplot
(http://matplotlib.sourceforge.net/)")
	sys.exit(1)


class example_fft_filter_ccc(gr.top_block):
	def __init__(self, N, fs, bw0, bw1, tw, atten, D):
		gr.top_block.__init__(self)

		self._nsamps = N
		self._fs = fs
		self._bw0 = bw0
		self._bw1 = bw1
		self._tw = tw
		self._at = atten
		self._decim = D
		taps = filter.firdes.complex_band_pass_2(1, self._fs,
										self._bw0, self._bw1,
										self._tw, self._at)
		print("Num. Taps: ", len(taps))
```

```
        self.src = analog.noise_source_c(analog.GR_GAUSSIAN, 1)
        self.head = blocks.head(gr.sizeof_gr_complex, self._nsamps)

        self.filt0 = filter.fft_filter_ccc(self._decim, taps)

        self.vsnk_src = blocks.vector_sink_c()
        self.vsnk_out = blocks.vector_sink_c()

        self.connect(self.src, self.head, self.vsnk_src)
        self.connect(self.head, self.filt0, self.vsnk_out)


def main():
    parser = ArgumentParser(conflict_handler="resolve")
    parser.add_argument("-N", "--nsamples", type=int, default=10000,
                      help="Number of samples to process [default=%(default)r]")
    parser.add_argument("-s", "--samplerate", type=eng_float, default=8000,
                      help="System sample rate [default=%(default)r]")
    parser.add_argument("-S", "--start-pass", type=eng_float, default=1000,
                      help="Start of Passband [default=%(default)r]")
    parser.add_argument("-E", "--end-pass", type=eng_float, default=2000,
                      help="End of Passband [default=%(default)r]")
    parser.add_argument("-T", "--transition", type=eng_float, default=100,
                      help="Transition band [default=%(default)r]")
    parser.add_argument("-A", "--attenuation", type=eng_float, default=80,
                      help="Stopband attenuation [default=%(default)r]")
    parser.add_argument("-D", "--decimation", type=int, default=1,
                      help="Decmation factor [default=%(default)r]")
    args = parser.parse_args()

    put = example_fft_filter_ccc(args.nsamples,
                                 args.samplerate,
                                 args.start_pass,
                                 args.end_pass,
                                 args.transition,
                                 args.attenuation,
                                 args.decimation)
    put.run()

    data_src = numpy.array(put.vsnk_src.data())
    data_snk = numpy.array(put.vsnk_out.data())

    # Plot the signals PSDs
    nfft = 1024
```

```
        f1 = pyplot.figure(1, figsize=(12, 10))
        s1 = f1.add_subplot(1, 1, 1)
        s1.psd(data_src, NFFT=nfft, noverlap=nfft / 4,
               Fs=args.samplerate)
        s1.psd(data_snk, NFFT=nfft, noverlap=nfft / 4,
               Fs=args.samplerate)

        f2 = pyplot.figure(2, figsize=(12, 10))
        s2 = f2.add_subplot(1, 1, 1)
        s2.plot(data_src)
        s2.plot(data_snk.real, 'g')

        pyplot.show()


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

# ==================== END OF FILE: gr-filter/examples/fft_filter_ccc.py ====================


# ==================== START OF FILE: gr-filter/examples/fir_filter_ccc.py ====================
#!/usr/bin/env python
#
# Copyright 2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr, filter
from gnuradio import analog
from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio.eng_arg import eng_float, intx
from argparse import ArgumentParser
import sys
import numpy

try:
    from matplotlib import pyplot
except ImportError:
    print("Error:        could        not        from        matplotlib        import        pyplot
(http://matplotlib.sourceforge.net/)")
    sys.exit(1)


class example_fir_filter_ccc(gr.top_block):
    def __init__(self, N, fs, bw, tw, atten, D):
        gr.top_block.__init__(self)

        self._nsamps = N
        self._fs = fs
        self._bw = bw
        self._tw = tw
        self._at = atten
        self._decim = D
        taps = filter.firdes.low_pass_2(
            1, self._fs, self._bw, self._tw, self._at)
        print("Num. Taps: ", len(taps))

        self.src = analog.noise_source_c(analog.GR_GAUSSIAN, 1)
        self.head = blocks.head(gr.sizeof_gr_complex, self._nsamps)

        self.filt0 = filter.fir_filter_ccc(self._decim, taps)

        self.vsnk_src = blocks.vector_sink_c()
        self.vsnk_out = blocks.vector_sink_c()

        self.connect(self.src, self.head, self.vsnk_src)
        self.connect(self.head, self.filt0, self.vsnk_out)


def main():
    parser = ArgumentParser(conflict_handler="resolve")
    parser.add_argument("-N", "--nsamples", type=int, default=10000,
                        help="Number of samples to process [default=%(default)r]")
    parser.add_argument("-s", "--samplerate", type=eng_float, default=8000,
                        help="System sample rate [default=%(default)r]")
    parser.add_argument("-B", "--bandwidth", type=eng_float, default=1000,
                        help="Filter bandwidth [default=%(default)r]")
    parser.add_argument("-T", "--transition", type=eng_float, default=100,
                        help="Transition band [default=%(default)r]")
    parser.add_argument("-A", "--attenuation", type=eng_float, default=80,
                        help="Stopband attenuation [default=%(default)r]")
    parser.add_argument("-D", "--decimation", type=int, default=1,
                        help="Decmation factor [default=%(default)r]")
    args = parser.parse_args()

    put = example_fir_filter_ccc(args.nsamples,
                                 args.samplerate,
                                 args.bandwidth,
                                 args.transition,
                                 args.attenuation,
                                 args.decimation)
    put.run()

    data_src = numpy.array(put.vsnk_src.data())
    data_snk = numpy.array(put.vsnk_out.data())

    # Plot the signals PSDs
    nfft = 1024
    f1 = pyplot.figure(1, figsize=(12, 10))
    s1 = f1.add_subplot(1, 1, 1)
    s1.psd(data_src, NFFT=nfft, noverlap=nfft / 4,
           Fs=args.samplerate)
    s1.psd(data_snk, NFFT=nfft, noverlap=nfft / 4,
           Fs=args.samplerate)

    f2 = pyplot.figure(2, figsize=(12, 10))
    s2 = f2.add_subplot(1, 1, 1)
    s2.plot(data_src)
    s2.plot(data_snk.real, 'g')

    pyplot.show()


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

# ==================== END OF FILE: gr-filter/examples/fir_filter_ccc.py ====================
# ==================== START OF FILE: gr-filter/examples/fir_filter_fff.py
====================
#!/usr/bin/env python
#
# Copyright 2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr, filter
from gnuradio import analog
from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio.eng_arg import eng_float, intx
from argparse import ArgumentParser
import sys
import numpy

try:
    from matplotlib import pyplot
except ImportError:
    print("Error: could not from matplotlib import pyplot
(http://matplotlib.sourceforge.net/)")
    sys.exit(1)


class example_fir_filter_fff(gr.top_block):
    def __init__(self, N, fs, bw, tw, atten, D):
        gr.top_block.__init__(self)

        self._nsamps = N
        self._fs = fs
        self._bw = bw
        self._tw = tw
        self._at = atten
        self._decim = D
        taps = filter.firdes.low_pass_2(
            1, self._fs, self._bw, self._tw, self._at)
        print("Num. Taps: ", len(taps))
        self.src = analog.noise_source_f(analog.GR_GAUSSIAN, 1)
        self.head = blocks.head(gr.sizeof_float, self._nsamps)

        self.filt0 = filter.fir_filter_fff(self._decim, taps)

        self.vsnk_src = blocks.vector_sink_f()
        self.vsnk_out = blocks.vector_sink_f()

        self.connect(self.src, self.head, self.vsnk_src)
        self.connect(self.head, self.filt0, self.vsnk_out)


def main():
    parser = ArgumentParser(conflict_handler="resolve")
    parser.add_argument("-N", "--nsamples", type=int, default=10000,
                        help="Number of samples to process [default=%(default)r]")
    parser.add_argument("-s", "--samplerate", type=eng_float, default=8000,
                        help="System sample rate [default=%(default)r]")
    parser.add_argument("-B", "--bandwidth", type=eng_float, default=1000,
                        help="Filter bandwidth [default=%(default)r]")
    parser.add_argument("-T", "--transition", type=eng_float, default=100,
                        help="Transition band [default=%(default)r]")
    parser.add_argument("-A", "--attenuation", type=eng_float, default=80,
                        help="Stopband attenuation [default=%(default)r]")
    parser.add_argument("-D", "--decimation", type=int, default=1,
                        help="Decmation factor [default=%(default)r]")
    args = parser.parse_args()

    put = example_fir_filter_fff(args.nsamples,
                                 args.samplerate,
                                 args.bandwidth,
                                 args.transition,
                                 args.attenuation,
                                 args.decimation)
    put.run()

    data_src = numpy.array(put.vsnk_src.data())
    data_snk = numpy.array(put.vsnk_out.data())

    # Plot the signals PSDs
    nfft = 1024
    f1 = pyplot.figure(1, figsize=(12, 10))
    s1 = f1.add_subplot(1, 1, 1)
    s1.psd(data_src, NFFT=nfft, noverlap=nfft / 4,
           Fs=args.samplerate)
    s1.psd(data_snk, NFFT=nfft, noverlap=nfft / 4,
           Fs=args.samplerate)

    f2 = pyplot.figure(2, figsize=(12, 10))
    s2 = f2.add_subplot(1, 1, 1)
    s2.plot(data_src)
    s2.plot(data_snk.real, 'g')

    pyplot.show()


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

# ==================== END OF FILE: gr-filter/examples/fir_filter_fff.py
====================


# ==================== START OF FILE: gr-filter/examples/gr_filtdes_api.py
====================
#!/usr/bin/env python
#
# Copyright 2012 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio.filter import filter_design
import sys

'''
API Blocking call
returns filter taps for FIR filter design
returns b,a for IIR filter design
'''
filtobj = filter_design.launch(sys.argv)
```

```
# Displaying all filter parameters
print("Filter Count:", filtobj.get_filtercount())
print("Filter type:", filtobj.get_restype())
print("Filter params", filtobj.get_params())
print("Filter Coefficients", filtobj.get_taps())

# ==================== END OF FILE: gr-filter/examples/gr_filtdes_api.py
====================


# ==================== START OF FILE: gr-filter/examples/gr_filtdes_callback.py
====================
#!/usr/bin/env python
#
# Copyright 2012 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio.filter import filter_design
import sys
try:
    from PyQt5 import Qt, QtCore, QtGui
except ImportError:
    print("Please install PyQt5 to run this script
(http://www.riverbankcomputing.co.uk/software/pyqt/download)")
    raise SystemExit(1)

'''
Callback example
Function called when "design" button is pressed
or pole-zero plot is changed
launch function returns gr_filter_design mainwindow
object when callback is not None
'''


def print_params(filtobj):
    print("Filter Count:", filtobj.get_filtercount())
    print("Filter type:", filtobj.get_restype())
```

```
        print("Filter params", filtobj.get_params())
        print("Filter Coefficients", filtobj.get_taps())


app = Qt.QApplication(sys.argv)
# launch function returns gr_filter_design mainwindow object
main_win = filter_design.launch(sys.argv, print_params)
main_win.show()
app.exec_()

# ==================== END OF FILE: gr-filter/examples/gr_filtdes_callback.py
====================


# ==================== START OF FILE: gr-filter/examples/gr_filtdes_live_upd.py
====================
#!/usr/bin/env python
#
# Copyright 2012,2020 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio.filter import filter_design
from gnuradio import gr, filter
from gnuradio.fft import window
from gnuradio import blocks
import sys

try:
        from gnuradio import qtgui
        from PyQt5 import QtGui, QtCore
        import sip
except ImportError:
        sys.stderr.write("Error: Program requires PyQt5 and gr-qtgui.\n")
        sys.exit(1)



try:
        from gnuradio import analog
except ImportError:
        sys.stderr.write("Error: Program requires gr-analog.\n")
        sys.exit(1)

try:
        from gnuradio import blocks
except ImportError:
        sys.stderr.write("Error: Program requires gr-blocks.\n")
        sys.exit(1)

try:
        from gnuradio import channels
except ImportError:
        sys.stderr.write("Error: Program requires gr-channels.\n")
        sys.exit(1)


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

            Rs = 8000
            f1 = 1000
            f2 = 2000

            npts = 2048

            self.qapp = QtGui.QApplication(sys.argv)

            self.filt_taps = [1, ]

            src1 = analog.sig_source_c(Rs, analog.GR_SIN_WAVE, f1, 0.1, 0)
            src2 = analog.sig_source_c(Rs, analog.GR_SIN_WAVE, f2, 0.1, 0)
            src = blocks.add_cc()
            channel = channels.channel_model(0.01)
            self.filt = filter.fft_filter_ccc(1, self.filt_taps)
            thr = blocks.throttle(gr.sizeof_gr_complex, 100 * npts)
            self.snk1 = qtgui.freq_sink_c(npts, window.WIN_BLACKMAN_hARRIS,
                                          0, Rs,
                                          "Complex Freq Example", 1)

            self.connect(src1, (src, 0))
            self.connect(src2, (src, 1))
            self.connect(src, channel, thr, self.filt, (self.snk1, 0))
            # Get the reference pointer to the SpectrumDisplayForm QWidget
            pyQt = self.snk1.qwidget()

            # Wrap the pointer as a PyQt SIP object
            # This can now be manipulated as a PyQt5.QtGui.QWidget
            pyWin = sip.wrapinstance(pyQt, QtGui.QWidget)
            pyWin.show()

        def update_filter(self, filtobj):
            print("Filter type:", filtobj.get_restype())
            print("Filter params", filtobj.get_params())
            self.filt.set_taps(filtobj.get_taps())


if __name__ == "__main__":
        tb = my_top_block()
        tb.start()
        mw = filter_design.launch(sys.argv, tb.update_filter)
        mw.show()
        tb.qapp.exec_()
        tb.stop()

# ==================== END OF FILE: gr-filter/examples/gr_filtdes_live_upd.py
====================


# ==================== START OF FILE: gr-filter/examples/gr_filtdes_restrict.py
====================
#!/usr/bin/env python
#
# Copyright 2012 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio.filter import filter_design
import sys
try:
        from PyQt5 import Qt, QtCore, QtGui
except ImportError:
        print("Please install PyQt5 to run this script
(http://www.riverbankcomputing.co.uk/software/pyqt/download)")
        raise SystemExit(1)


'''
Callback with restrict example
Function called when "design" button is pressed
or pole-zero plot is changed
'''


def print_params(filtobj):
        print("Filter Count:", filtobj.get_filtercount())
        print("Filter type:", filtobj.get_restype())
        print("Filter params", filtobj.get_params())
        print("Filter Coefficients", filtobj.get_taps())


app = Qt.QApplication(sys.argv)
main_win = filter_design.launch(sys.argv, callback=print_params, restype="iir")
main_win.show()
app.exec_()

# ==================== END OF FILE: gr-filter/examples/gr_filtdes_restrict.py
====================


# ==================== START OF FILE: gr-filter/examples/interpolate.py
====================
#!/usr/bin/env python
#
# Copyright 2009,2012,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import blocks
from gnuradio import filter
from gnuradio.fft import window
import sys
import time
import numpy

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)

try:
    import pylab
    from pylab import mlab
except ImportError:
    sys.stderr.write(
        "Error: Program requires matplotlib (see: matplotlib.sourceforge.net).\n")
    sys.exit(1)


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

        self._N = 100000          # number of samples to use
        self._fs = 2000           # initial sampling rate
        self._interp = 5          # Interpolation rate for PFB interpolator
        self._ainterp = 5.5       # Resampling rate for the PFB arbitrary resampler

        # Frequencies of the signals we construct
        freq1 = 100
        freq2 = 200

        # Create a set of taps for the PFB interpolator
        # This is based on the post-interpolation sample rate
        self._taps = filter.firdes.low_pass_2(self._interp,
                                              self._interp * self._fs,
                                              freq2 + 50, 50,
                                              attenuation_dB=120,

window=window.WIN_BLACKMAN_hARRIS)

        # Create a set of taps for the PFB arbitrary resampler
        # The filter size is the number of filters in the filterbank; 32 will give very low side-
lobes,
        # and larger numbers will reduce these even farther
        # The taps in this filter are based on a sampling rate of the filter size since it acts
        # internally as an interpolator.
        flt_size = 32
        self._taps2 = filter.firdes.low_pass_2(flt_size,
                                               flt_size * self._fs,
                                               freq2 + 50, 150,
                                               attenuation_dB=120,

window=window.WIN_BLACKMAN_hARRIS)

        # Calculate the number of taps per channel for our own information
        tpc = numpy.ceil(float(len(self._taps)) / float(self._interp))
        print("Number of taps:         ", len(self._taps))
        print("Number of filters:    ", self._interp)
        print("Taps per channel:      ", tpc)

        # Create a couple of signals at different frequencies
        self.signal1 = analog.sig_source_c(
            self._fs, analog.GR_SIN_WAVE, freq1, 0.5)
        self.signal2 = analog.sig_source_c(
            self._fs, analog.GR_SIN_WAVE, freq2, 0.5)
        self.signal = blocks.add_cc()

        self.head = blocks.head(gr.sizeof_gr_complex, self._N)

        # Construct the PFB interpolator filter
        self.pfb = filter.pfb.interpolator_ccf(self._interp, self._taps)

        # Construct the PFB arbitrary resampler filter
        self.pfb_ar = filter.pfb.arb_resampler_ccf(
            self._ainterp, self._taps2, flt_size)
        self.snk_i = blocks.vector_sink_c()

        # self.pfb_ar.pfb.print_taps()
        # self.pfb.pfb.print_taps()

        # Connect the blocks
        self.connect(self.signal1, self.head, (self.signal, 0))
        self.connect(self.signal2, (self.signal, 1))
        self.connect(self.signal, self.pfb)
        self.connect(self.signal, self.pfb_ar)
        self.connect(self.signal, self.snk_i)

        # Create the sink for the interpolated signals
        self.snk1 = blocks.vector_sink_c()
        self.snk2 = blocks.vector_sink_c()
        self.connect(self.pfb, self.snk1)
        self.connect(self.pfb_ar, self.snk2)


def main():
    tb = pfb_top_block()

    tstart = time.time()
    tb.run()
    tend = time.time()
    print("Run time: %f" % (tend - tstart))

    if 1:
        fig1 = pylab.figure(1, figsize=(12, 10), facecolor="w")
        fig2 = pylab.figure(2, figsize=(12, 10), facecolor="w")
        fig3 = pylab.figure(3, figsize=(12, 10), facecolor="w")

        Ns = 10000
        Ne = 10000

        fftlen = 8192
        winfunc = numpy.blackman

        # Plot input signal
        fs = tb._fs

        d = tb.snk_i.data()[Ns:Ns + Ne]
        sp1_f = fig1.add_subplot(2, 1, 1)

        X, freq = mlab.psd(d, NFFT=fftlen, noverlap=fftlen / 4, Fs=fs,
                           window=lambda d: d * winfunc(fftlen),
                           scale_by_freq=True)
        X_in = 10.0 * numpy.log10(abs(numpy.fft.fftshift(X)))
        f_in = numpy.arange(-fs / 2.0, fs / 2.0, fs / float(X_in.size))
        p1_f = sp1_f.plot(f_in, X_in, "b")
        sp1_f.set_xlim([min(f_in), max(f_in) + 1])
        sp1_f.set_ylim([-200.0, 50.0])

        sp1_f.set_title("Input Signal", weight="bold")
        sp1_f.set_xlabel("Frequency (Hz)")
        sp1_f.set_ylabel("Power (dBW)")
```

```
Ts = 1.0 / fs
Tmax = len(d) * Ts

t_in = numpy.arange(0, Tmax, Ts)
x_in = numpy.array(d)
sp1_t = fig1.add_subplot(2, 1, 2)
p1_t = sp1_t.plot(t_in, x_in.real, "b-o")
#p1_t = sp1_t.plot(t_in, x_in.imag, "r-o")
sp1_t.set_ylim([-2.5, 2.5])

sp1_t.set_title("Input Signal", weight="bold")
sp1_t.set_xlabel("Time (s)")
sp1_t.set_ylabel("Amplitude")

# Plot output of PFB interpolator
fs_int = tb._fs * tb._interp

sp2_f = fig2.add_subplot(2, 1, 1)
d = tb.snk1.data()[Ns:Ns + (tb._interp * Ne)]
X, freq = mlab.psd(d, NFFT=fftlen, noverlap=fftlen / 4, Fs=fs,
                   window=lambda d: d * winfunc(fftlen),
                   scale_by_freq=True)
X_o = 10.0 * numpy.log10(abs(numpy.fft.fftshift(X)))
f_o = numpy.arange(-fs_int / 2.0, fs_int / 2.0,
                   fs_int / float(X_o.size))
p2_f = sp2_f.plot(f_o, X_o, "b")
sp2_f.set_xlim([min(f_o), max(f_o) + 1])
sp2_f.set_ylim([-200.0, 50.0])

sp2_f.set_title("Output Signal from PFB Interpolator", weight="bold")
sp2_f.set_xlabel("Frequency (Hz)")
sp2_f.set_ylabel("Power (dBW)")

Ts_int = 1.0 / fs_int
Tmax = len(d) * Ts_int

t_o = numpy.arange(0, Tmax, Ts_int)
x_o1 = numpy.array(d)
sp2_t = fig2.add_subplot(2, 1, 2)
p2_t = sp2_t.plot(t_o, x_o1.real, "b-o")
#p2_t = sp2_t.plot(t_o, x_o.imag, "r-o")
sp2_t.set_ylim([-2.5, 2.5])

sp2_t.set_title("Output Signal from PFB Interpolator", weight="bold")
```

```
        sp2_t.set_xlabel("Time (s)")
        sp2_t.set_ylabel("Amplitude")

        # Plot output of PFB arbitrary resampler
        fs_aint = tb._fs * tb._ainterp

        sp3_f = fig3.add_subplot(2, 1, 1)
        d = tb.snk2.data()[Ns:Ns + (tb._interp * Ne)]
        X, freq = mlab.psd(d, NFFT=fftlen, noverlap=fftlen / 4, Fs=fs,
                           window=lambda d: d * winfunc(fftlen),
                           scale_by_freq=True)
        X_o = 10.0 * numpy.log10(abs(numpy.fft.fftshift(X)))
        f_o = numpy.arange(-fs_aint / 2.0, fs_aint / 2.0,
                           fs_aint / float(X_o.size))
        p3_f = sp3_f.plot(f_o, X_o, "b")
        sp3_f.set_xlim([min(f_o), max(f_o) + 1])
        sp3_f.set_ylim([-200.0, 50.0])

        sp3_f.set_title(
            "Output Signal from PFB Arbitrary Resampler", weight="bold")
        sp3_f.set_xlabel("Frequency (Hz)")
        sp3_f.set_ylabel("Power (dBW)")

        Ts_aint = 1.0 / fs_aint
        Tmax = len(d) * Ts_aint

        t_o = numpy.arange(0, Tmax, Ts_aint)
        x_o2 = numpy.array(d)
        sp3_f = fig3.add_subplot(2, 1, 2)
        p3_f = sp3_f.plot(t_o, x_o2.real, "b-o")
        p3_f = sp3_f.plot(t_o, x_o1.real, "m-o")
        #p3_f = sp3_f.plot(t_o, x_o2.imag, "r-o")
        sp3_f.set_ylim([-2.5, 2.5])

        sp3_f.set_title(
            "Output Signal from PFB Arbitrary Resampler", weight="bold")
        sp3_f.set_xlabel("Time (s)")
        sp3_f.set_ylabel("Amplitude")

        pylab.show()


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

# ==================== END OF FILE: gr-filter/examples/interpolate.py ====================


# ==================== START OF FILE: gr-filter/examples/reconstruction.py ====================
#!/usr/bin/env python
#
# Copyright 2010,2012,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr, digital
from gnuradio import filter
from gnuradio import blocks
from gnuradio.fft import window
import sys
import numpy

try:
    from gnuradio import channels
except ImportError:
    print("Error: Program requires gr-channels.")
    sys.exit(1)

try:
    from matplotlib import pyplot
except ImportError:
    print("Error: Program requires matplotlib (see: matplotlib.sourceforge.net).")
    sys.exit(1)

fftlen = 8192


def main():
    N = 10000
```

```
fs = 2000.0
Ts = 1.0 / fs
t = numpy.arange(0, N * Ts, Ts)

# When playing with the number of channels, be careful about the filter
# specs and the channel map of the synthesizer set below.
nchans = 10

# Build the filter(s)
bw = 1000
tb = 400
proto_taps = filter.firdes.low_pass_2(1, nchans * fs,
                                      bw, tb, 80,
                                      window.WIN_BLACKMAN_hARRIS)
print("Filter length: ", len(proto_taps))

# Create a modulated signal
npwr = 0.01
data = numpy.random.randint(0, 256, N)
rrc_taps = filter.firdes.root_raised_cosine(1, 2, 1, 0.35, 41)

src = blocks.vector_source_b(data.astype(numpy.uint8).tolist(), False)
mod = digital.psk_mod(samples_per_symbol=2)
chan = channels.channel_model(npwr)
rrc = filter.fft_filter_ccc(1, rrc_taps)

# Split it up into pieces
channelizer = filter.pfb.channelizer_ccf(nchans, proto_taps, 2)

# Put the pieces back together again
syn_taps = [nchans * t for t in proto_taps]
synthesizer = filter.pfb_synthesizer_ccf(nchans, syn_taps, True)
src_snk = blocks.vector_sink_c()
snk = blocks.vector_sink_c()

# Remap the location of the channels
# Can be done in synth or channelizer (watch out for rotattions in
# the channelizer)
synthesizer.set_channel_map([0, 1, 2, 3, 4,
                             15, 16, 17, 18, 19])

tb = gr.top_block()
tb.connect(src, mod, chan, rrc, channelizer)
tb.connect(rrc, src_snk)
```

```
vsnk = []
for i in range(nchans):
        tb.connect((channelizer, i), (synthesizer, i))

        vsnk.append(blocks.vector_sink_c())
        tb.connect((channelizer, i), vsnk[i])

tb.connect(synthesizer, snk)
tb.run()

sin = numpy.array(src_snk.data()[1000:])
sout = numpy.array(snk.data()[1000:])

# Plot original signal
fs_in = nchans * fs
f1 = pyplot.figure(1, figsize=(16, 12), facecolor='w')
s11 = f1.add_subplot(2, 2, 1)
s11.psd(sin, NFFT=fftlen, Fs=fs_in)
s11.set_title("PSD of Original Signal")
s11.set_ylim([-200, -20])

s12 = f1.add_subplot(2, 2, 2)
s12.plot(sin.real[1000:1500], "o-b")
s12.plot(sin.imag[1000:1500], "o-r")
s12.set_title("Original Signal in Time")

start = 1
skip = 2
s13 = f1.add_subplot(2, 2, 3)
s13.plot(sin.real[start::skip], sin.imag[start::skip], "o")
s13.set_title("Constellation")
s13.set_xlim([-2, 2])
s13.set_ylim([-2, 2])

# Plot channels
nrows = int(numpy.sqrt(nchans))
ncols = int(numpy.ceil(float(nchans) / float(nrows)))

f2 = pyplot.figure(2, figsize=(16, 12), facecolor='w')
for n in range(nchans):
        s = f2.add_subplot(nrows, ncols, n + 1)
        s.psd(vsnk[n].data(), NFFT=fftlen, Fs=fs_in)
        s.set_title("Channel {0}".format(n))
        s.set_ylim([-200, -20])

    # Plot reconstructed signal
    fs_out = 2 * nchans * fs
    f3 = pyplot.figure(3, figsize=(16, 12), facecolor='w')
    s31 = f3.add_subplot(2, 2, 1)
    s31.psd(sout, NFFT=fftlen, Fs=fs_out)
    s31.set_title("PSD of Reconstructed Signal")
    s31.set_ylim([-200, -20])

    s32 = f3.add_subplot(2, 2, 2)
    s32.plot(sout.real[1000:1500], "o-b")
    s32.plot(sout.imag[1000:1500], "o-r")
    s32.set_title("Reconstructed Signal in Time")

    start = 0
    skip = 4
    s33 = f3.add_subplot(2, 2, 3)
    s33.plot(sout.real[start::skip], sout.imag[start::skip], "o")
    s33.set_title("Constellation")
    s33.set_xlim([-2, 2])
    s33.set_ylim([-2, 2])

    pyplot.show()


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

# ==================== END OF FILE: gr-filter/examples/reconstruction.py ====================



# ==================== START OF FILE: gr-filter/examples/resampler.py ====================
#!/usr/bin/env python
#
# Copyright 2009,2012,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
```

```
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import filter
from gnuradio import blocks
import sys
import numpy

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)

try:
    from matplotlib import pyplot
except ImportError:
    sys.stderr.write(
        "Error: Program requires matplotlib (see: matplotlib.sourceforge.net).\n")
    sys.exit(1)


class mytb(gr.top_block):
    def __init__(self, fs_in, fs_out, fc, N=10000):
        gr.top_block.__init__(self)

        rerate = float(fs_out) / float(fs_in)
        print("Resampling from %f to %f by %f " % (fs_in, fs_out, rerate))

        # Creating our own taps
        taps = filter.firdes.low_pass_2(32, 32, 0.25, 0.1, 80)

        self.src = analog.sig_source_c(fs_in, analog.GR_SIN_WAVE, fc, 1)
        #self.src = analog.noise_source_c(analog.GR_GAUSSIAN, 1)
        self.head = blocks.head(gr.sizeof_gr_complex, N)

        # A resampler with our taps
        self.resamp_0 = filter.pfb.arb_resampler_ccf(rerate, taps,
                                                      flt_size=32)

        # A resampler that just needs a resampling rate.
        # Filter is created for us and designed to cover
        # entire bandwidth of the input signal.
        # An optional atten=XX rate can be used here to
        # specify the out-of-band rejection (default=80).
        self.resamp_1 = filter.pfb.arb_resampler_ccf(rerate)

        self.snk_in = blocks.vector_sink_c()
        self.snk_0 = blocks.vector_sink_c()
        self.snk_1 = blocks.vector_sink_c()

        self.connect(self.src, self.head, self.snk_in)
        self.connect(self.head, self.resamp_0, self.snk_0)
        self.connect(self.head, self.resamp_1, self.snk_1)


def main():
    fs_in = 8000
    fs_out = 20000
    fc = 1000
    N = 10000

    tb = mytb(fs_in, fs_out, fc, N)
    tb.run()

    # Plot PSD of signals
    nfftsize = 2048
    fig1 = pyplot.figure(1, figsize=(10, 10), facecolor="w")
    sp1 = fig1.add_subplot(2, 1, 1)
    sp1.psd(tb.snk_in.data(), NFFT=nfftsize,
            noverlap=nfftsize / 4, Fs=fs_in)
    sp1.set_title(("Input Signal at f_s=%.2f kHz" % (fs_in / 1000.0)))
    sp1.set_xlim([-fs_in / 2, fs_in / 2])

    sp2 = fig1.add_subplot(2, 1, 2)
    sp2.psd(tb.snk_0.data(), NFFT=nfftsize,
            noverlap=nfftsize / 4, Fs=fs_out,
            label="With our filter")
    sp2.psd(tb.snk_1.data(), NFFT=nfftsize,
            noverlap=nfftsize / 4, Fs=fs_out,
            label="With auto-generated filter")
    sp2.set_title(("Output Signals at f_s=%.2f kHz" % (fs_out / 1000.0)))
    sp2.set_xlim([-fs_out / 2, fs_out / 2])
    sp2.legend()

    # Plot signals in time
    Ts_in = 1.0 / fs_in
    Ts_out = 1.0 / fs_out
    t_in = numpy.arange(0, len(tb.snk_in.data()) * Ts_in, Ts_in)
    t_out = numpy.arange(0, len(tb.snk_0.data()) * Ts_out, Ts_out)

    fig2 = pyplot.figure(2, figsize=(10, 10), facecolor="w")
    sp21 = fig2.add_subplot(2, 1, 1)
    sp21.plot(t_in, tb.snk_in.data())
    sp21.set_title(("Input Signal at f_s=%.2f kHz" % (fs_in / 1000.0)))
    sp21.set_xlim([t_in[100], t_in[200]])

    sp22 = fig2.add_subplot(2, 1, 2)
    sp22.plot(t_out, tb.snk_0.data(),
              label="With our filter")
    sp22.plot(t_out, tb.snk_1.data(),
              label="With auto-generated filter")
    sp22.set_title(("Output Signals at f_s=%.2f kHz" % (fs_out / 1000.0)))
    r = float(fs_out) / float(fs_in)
    sp22.set_xlim([t_out[r * 100], t_out[r * 200]])
    sp22.legend()

    pyplot.show()


if __name__ == "__main__":
    main()

# ==================== END OF FILE: gr-filter/examples/resampler.py
====================


# ==================== START OF FILE: gr-filter/examples/synth_filter.py
====================
#!/usr/bin/env python
#
# Copyright 2010,2012,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import filter
from gnuradio import blocks
import sys
import numpy

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)

try:
    from matplotlib import pyplot
except ImportError:
    sys.stderr.write(
        "Error: Program requires matplotlib (see: matplotlib.sourceforge.net).\n")
    sys.exit(1)


def main():
    N = 1000000
    fs = 8000

    freqs = [100, 200, 300, 400, 500]
    nchans = 7

    sigs = list()
    for fi in freqs:
        s = analog.sig_source_c(fs, analog.GR_SIN_WAVE, fi, 1)
        sigs.append(s)

    taps = filter.firdes.low_pass_2(len(freqs), fs,
                                    fs / float(nchans) / 2, 100, 100)
    print("Num. Taps = %d (taps per filter = %d)" % (len(taps),
                                                   len(taps) / nchans))
    filtbank = filter.pfb_synthesizer_ccf(nchans, taps)

    head = blocks.head(gr.sizeof_gr_complex, N)
    snk = blocks.vector_sink_c()

    tb = gr.top_block()
    tb.connect(filtbank, head, snk)

    for i, si in enumerate(sigs):
        tb.connect(si, (filtbank, i))

    tb.run()

    if 1:
        f1 = pyplot.figure(1)
        s1 = f1.add_subplot(1, 1, 1)
        s1.plot(snk.data()[1000:])

        fftlen = 2048
        f2 = pyplot.figure(2)
        s2 = f2.add_subplot(1, 1, 1)
        winfunc = numpy.blackman
        s2.psd(snk.data()[10000:], NFFT=fftlen,
               Fs=nchans * fs,
               noverlap=fftlen / 4,
               window=lambda d: d * winfunc(fftlen))

        pyplot.show()


if __name__ == "__main__":
    main()

# ==================== END OF FILE: gr-filter/examples/synth_filter.py ====================


# ==================== START OF FILE: gr-filter/examples/synth_to_chan.py ====================
#!/usr/bin/env python
#
# Copyright 2010,2012,2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import blocks
from gnuradio import filter
import sys
import numpy

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)

try:
    from matplotlib import pyplot
except ImportError:
    sys.stderr.write(
            "Error: Program requires matplotlib (see: matplotlib.sourceforge.net).\n")
    sys.exit(1)


def main():
    N = 1000000
    fs = 8000

    freqs = [100, 200, 300, 400, 500]
    nchans = 7

    sigs = list()
    fmtx = list()
    for fi in freqs:
            s = analog.sig_source_f(fs, analog.GR_SIN_WAVE, fi, 1)
            fm = analog.nbfm_tx(fs, 4 * fs, max_dev=10000,
                                                tau=75e-6, fh=0.925 * (4 * fs) / 2.0)
            sigs.append(s)
            fmtx.append(fm)

    syntaps = filter.firdes.low_pass_2(
            len(freqs), fs, fs / float(nchans) / 2, 100, 100)
    print("Synthesis Num. Taps = %d (taps per filter = %d)" % (len(syntaps),
                                                                                                              len(syntaps)          /
nchans))
    chtaps = filter.firdes.low_pass_2(
            len(freqs), fs, fs / float(nchans) / 2, 100, 100)
    print("Channelizer Num. Taps = %d (taps per filter = %d)" % (len(chtaps),
                                                                                                                  len(chtaps)        /
nchans))
    filtbank = filter.pfb_synthesizer_ccf(nchans, syntaps)
    channelizer = filter.pfb.channelizer_ccf(nchans, chtaps)
```

```
noise_level = 0.01
head = blocks.head(gr.sizeof_gr_complex, N)
noise = analog.noise_source_c(analog.GR_GAUSSIAN, noise_level)
addnoise = blocks.add_cc()
snk_synth = blocks.vector_sink_c()

tb = gr.top_block()

tb.connect(noise, (addnoise, 0))
tb.connect(filtbank, head, (addnoise, 1))
tb.connect(addnoise, channelizer)
tb.connect(addnoise, snk_synth)

snk = list()
for i, si in enumerate(sigs):
    tb.connect(si, fmtx[i], (filtbank, i))

for i in range(nchans):
    snk.append(blocks.vector_sink_c())
    tb.connect((channelizer, i), snk[i])

tb.run()

if 1:
    channel = 1
    data = snk[channel].data()[1000:]

    f1 = pyplot.figure(1)
    s1 = f1.add_subplot(1, 1, 1)
    s1.plot(data[10000:10200])
    s1.set_title(("Output Signal from Channel %d" % channel))

    fftlen = 2048
    winfunc = numpy.blackman
    #winfunc = numpy.hamming

    f2 = pyplot.figure(2)
    s2 = f2.add_subplot(1, 1, 1)
    s2.psd(data, NFFT=fftlen,
           Fs=nchans * fs,
           noverlap=fftlen / 4,
           window=lambda d: d * winfunc(fftlen))
    s2.set_title(("Output PSD from Channel %d" % channel))
```

```
        f3 = pyplot.figure(3)
        s3 = f3.add_subplot(1, 1, 1)
        s3.psd(snk_synth.data()[1000:], NFFT=fftlen,
               Fs=nchans * fs,
               noverlap=fftlen / 4,
               window=lambda d: d * winfunc(fftlen))
        s3.set_title("Output of Synthesis Filter")

        pyplot.show()


if __name__ == "__main__":
    main()

# ==================== END OF FILE: gr-filter/examples/synth_to_chan.py
====================


# ==================== START OF FILE: gr-qtgui/examples/pyqt_const_c.py
====================
#!/usr/bin/env python
#
# Copyright 2011,2012,2015 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr, filter
from gnuradio import blocks
import sys

try:
    from gnuradio import qtgui
    from PyQt5 import QtWidgets, Qt
    import sip
except ImportError:
    sys.stderr.write("Error: Program requires PyQt5 and gr-qtgui.\n")
    sys.exit(1)

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)

try:
    from gnuradio import channels
except ImportError:
    sys.stderr.write("Error: Program requires gr-channels.\n")
    sys.exit(1)


class dialog_box(QtWidgets.QWidget):
    def __init__(self, display, control):
        QtWidgets.QWidget.__init__(self, None)
        self.setWindowTitle('PyQt Test GUI')

        self.boxlayout = QtWidgets.QBoxLayout(
            QtWidgets.QBoxLayout.LeftToRight, self)
        self.boxlayout.addWidget(display, 1)
        self.boxlayout.addWidget(control)

        self.resize(800, 500)


class control_box(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle('Control Panel')

        self.setToolTip('Control the signals')
        QtWidgets.QToolTip.setFont(Qt.QFont('OldEnglish', 10))

        self.layout = QtWidgets.QFormLayout(self)

        # Control the first signal
        self.freq1Edit = QtWidgets.QLineEdit(self)
        self.freq1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Frequency:", self.freq1Edit)
        self.freq1Edit.editingFinished.connect(self.freq1EditText)

        self.amp1Edit = QtWidgets.QLineEdit(self)
        self.amp1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Amplitude:", self.amp1Edit)
        self.amp1Edit.editingFinished.connect(self.amp1EditText)

        # Control the second signal
        self.freq2Edit = QtWidgets.QLineEdit(self)
        self.freq2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Frequency:", self.freq2Edit)
        self.freq2Edit.editingFinished.connect(self.freq2EditText)

        self.amp2Edit = QtWidgets.QLineEdit(self)
        self.amp2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Amplitude:", self.amp2Edit)
        self.amp2Edit.editingFinished.connect(self.amp2EditText)

        self.quit = QtWidgets.QPushButton('Close', self)
        self.quit.setMinimumWidth(100)
        self.quit.clicked.connect(QtWidgets.qApp.quit)
        self.layout.addWidget(self.quit)

    def attach_signal1(self, signal):
        self.signal1 = signal
        self.freq1Edit.setText("{0}".format(self.signal1.frequency()))
        self.amp1Edit.setText("{0}".format(self.signal1.amplitude()))

    def attach_signal2(self, signal):
        self.signal2 = signal
        self.freq2Edit.setText("{0}".format(self.signal2.frequency()))
        self.amp2Edit.setText("{0}".format(self.signal2.amplitude()))

    def freq1EditText(self):
        try:
            newfreq = float(self.freq1Edit.text())
            self.signal1.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")

    def amp1EditText(self):
        try:
            newamp = float(self.amp1Edit.text())
            self.signal1.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")

    def freq2EditText(self):
        try:
            newfreq = float(self.freq2Edit.text())
            self.signal2.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")

    def amp2EditText(self):
        try:
            newamp = float(self.amp2Edit.text())
            self.signal2.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")


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

        Rs = 8000
        f1 = 100
        f2 = 200

        npts = 2048

        self.qapp = QtWidgets.QApplication(sys.argv)

        src1 = analog.sig_source_c(Rs, analog.GR_SIN_WAVE, f1, 0.5, 0)
        src2 = analog.sig_source_c(Rs, analog.GR_SIN_WAVE, f2, 0.5, 0)
        src = blocks.add_cc()
        channel = channels.channel_model(0.001)
        thr = blocks.throttle(gr.sizeof_gr_complex, 100 * npts)
        self.snk1 = qtgui.const_sink_c(npts, "Constellation Example", 1, None)
        self.snk1.disable_legend()

        self.connect(src1, (src, 0))
        self.connect(src2, (src, 1))
        self.connect(src, channel, thr, (self.snk1, 0))

        self.ctrl_win = control_box()
        self.ctrl_win.attach_signal1(src1)
        self.ctrl_win.attach_signal2(src2)

        # Get the reference pointer to the SpectrumDisplayForm QWidget
        pyQt = self.snk1.qwidget()
        # Wrap the pointer as a PyQt SIP object
        # This can now be manipulated as a PyQt5.QtWidgets.QWidget
        pyWin = sip.wrapinstance(pyQt, QtWidgets.QWidget)

        self.main_box = dialog_box(pyWin, self.ctrl_win)
        self.main_box.show()


if __name__ == "__main__":
    tb = my_top_block()
    tb.start()
    tb.qapp.exec_()
    tb.stop()

#    ====================    END    OF    FILE:    gr-qtgui/examples/pyqt_const_c.py
====================


#   ====================   START   OF   FILE:   gr-qtgui/examples/pyqt_example_c.py
====================
#!/usr/bin/env python
#
# Copyright 2011,2012,2015 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr, filter
from gnuradio.fft import window
from gnuradio import blocks
import sys

try:
    from gnuradio import qtgui
    from PyQt5 import QtWidgets, Qt
    import sip
except ImportError:
    sys.stderr.write("Error: Program requires PyQt5 and gr-qtgui.\n")
    sys.exit(1)

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)

try:
    from gnuradio import channels
except ImportError:
    sys.stderr.write("Error: Program requires gr-channels.\n")
    sys.exit(1)


class dialog_box(QtWidgets.QWidget):
    def __init__(self, display, control):
        QtWidgets.QWidget.__init__(self, None)
        self.setWindowTitle('PyQt Test GUI')

        self.boxlayout = QtWidgets.QBoxLayout(
            QtWidgets.QBoxLayout.LeftToRight, self)
        self.boxlayout.addWidget(display, 1)
        self.boxlayout.addWidget(control)

        self.resize(800, 500)


class control_box(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle('Control Panel')

        self.setToolTip('Control the signals')
        QtWidgets.QToolTip.setFont(Qt.QFont('OldEnglish', 10))

        self.layout = QtWidgets.QFormLayout(self)

        # Control the first signal
        self.freq1Edit = QtWidgets.QLineEdit(self)
        self.freq1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Frequency:", self.freq1Edit)
        self.freq1Edit.editingFinished.connect(self.freq1EditText)

        self.amp1Edit = QtWidgets.QLineEdit(self)
        self.amp1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Amplitude:", self.amp1Edit)
        self.amp1Edit.editingFinished.connect(self.amp1EditText)

        # Control the second signal
        self.freq2Edit = QtWidgets.QLineEdit(self)
        self.freq2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Frequency:", self.freq2Edit)
        self.freq2Edit.editingFinished.connect(self.freq2EditText)

        self.amp2Edit = QtWidgets.QLineEdit(self)
        self.amp2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Amplitude:", self.amp2Edit)
        self.amp2Edit.editingFinished.connect(self.amp2EditText)

        self.quit = QtWidgets.QPushButton('Close', self)
        self.quit.setMinimumWidth(100)
        self.layout.addWidget(self.quit)

        self.quit.clicked.connect(QtWidgets.qApp.quit)

    def attach_signal1(self, signal):
        self.signal1 = signal
        self.freq1Edit.setText(("{0}").format(self.signal1.frequency()))
        self.amp1Edit.setText(("{0}").format(self.signal1.amplitude()))

    def attach_signal2(self, signal):
        self.signal2 = signal
        self.freq2Edit.setText(("{0}").format(self.signal2.frequency()))
        self.amp2Edit.setText(("{0}").format(self.signal2.amplitude()))

    def freq1EditText(self):
        try:
            newfreq = float(self.freq1Edit.text())
            self.signal1.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")

    def amp1EditText(self):
        try:
            newamp = float(self.amp1Edit.text())
            self.signal1.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")

    def freq2EditText(self):
        try:
            newfreq = float(self.freq2Edit.text())
            self.signal2.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")

    def amp2EditText(self):
        try:
            newamp = float(self.amp2Edit.text())
            self.signal2.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")


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

        Rs = 8000
        f1 = 1000
        f2 = 2000

        fftsize = 2048

        self.qapp = QtWidgets.QApplication(sys.argv)
        ss = open(gr.prefix() + '/share/gnuradio/themes/dark.qss')
        sstext = ss.read()
        ss.close()
        self.qapp.setStyleSheet(sstext)

        src1 = analog.sig_source_c(Rs, analog.GR_SIN_WAVE, f1, 0.1, 0)
        src2 = analog.sig_source_c(Rs, analog.GR_SIN_WAVE, f2, 0.1, 0)
        src = blocks.add_cc()
        channel = channels.channel_model(0.001)
        thr = blocks.throttle(gr.sizeof_gr_complex, 100 * fftsize)
        self.snk1 = qtgui.sink_c(fftsize, window.WIN_BLACKMAN_hARRIS,
                                 0, Rs,
                                 "Complex Signal Example",
                                 True, True, True, False, None)

        self.connect(src1, (src, 0))
        self.connect(src2, (src, 1))
        self.connect(src, channel, thr, self.snk1)
        self.ctrl_win = control_box()
        self.ctrl_win.attach_signal1(src1)
        self.ctrl_win.attach_signal2(src2)

        # Get the reference pointer to the SpectrumDisplayForm QWidget
        pyQt = self.snk1.qwidget()

        # Wrap the pointer as a PyQt SIP object
        # This can now be manipulated as a PyQt5.QtWidgets.QWidget
        pyWin = sip.wrapinstance(pyQt, QtWidgets.QWidget)

        self.main_box = dialog_box(pyWin, self.ctrl_win)

        self.main_box.show()


if __name__ == "__main__":
    tb = my_top_block()
    tb.start()
    tb.qapp.exec_()
    tb.stop()

# ==================== END OF FILE: gr-qtgui/examples/pyqt_example_c.py ====================



# ==================== START OF FILE: gr-qtgui/examples/pyqt_example_f.py ====================
#!/usr/bin/env python
#
# Copyright 2011,2012,2015 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr, filter
from gnuradio import blocks
from gnuradio.fft import window
import sys

try:
    from gnuradio import qtgui
    from PyQt5 import QtWidgets, Qt
    import sip
except ImportError:
    sys.stderr.write("Error: Program requires PyQt5 and gr-qtgui.\n")
    sys.exit(1)

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)


class dialog_box(QtWidgets.QWidget):
    def __init__(self, display, control):
        QtWidgets.QWidget.__init__(self, None)
        self.setWindowTitle('PyQt Test GUI')

        self.boxlayout = QtWidgets.QBoxLayout(
                QtWidgets.QBoxLayout.LeftToRight, self)
        self.boxlayout.addWidget(display, 1)
        self.boxlayout.addWidget(control)

        self.resize(800, 500)


class control_box(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle('Control Panel')

        self.setToolTip('Control the signals')
        QtWidgets.QToolTip.setFont(Qt.QFont('OldEnglish', 10))

        self.layout = QtWidgets.QFormLayout(self)

        # Control the first signal
        self.freq1Edit = QtWidgets.QLineEdit(self)
        self.freq1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Frequency:", self.freq1Edit)
        self.freq1Edit.editingFinished.connect(self.freq1EditText)

        self.amp1Edit = QtWidgets.QLineEdit(self)
        self.amp1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Amplitude:", self.amp1Edit)
        self.amp1Edit.editingFinished.connect(self.amp1EditText)

        # Control the second signal
        self.freq2Edit = QtWidgets.QLineEdit(self)
        self.freq2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Frequency:", self.freq2Edit)
        self.freq2Edit.editingFinished.connect(self.freq2EditText)

        self.amp2Edit = QtWidgets.QLineEdit(self)
        self.amp2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Amplitude:", self.amp2Edit)
        self.amp2Edit.editingFinished.connect(self.amp2EditText)

        self.quit = QtWidgets.QPushButton('Close', self)
        self.quit.setMinimumWidth(100)
        self.layout.addWidget(self.quit)

        self.quit.clicked.connect(QtWidgets.qApp.quit)

    def attach_signal1(self, signal):
        self.signal1 = signal
        self.freq1Edit.setText(("{0}").format(self.signal1.frequency()))
        self.amp1Edit.setText(("{0}").format(self.signal1.amplitude()))

    def attach_signal2(self, signal):
        self.signal2 = signal
        self.freq2Edit.setText(("{0}").format(self.signal2.frequency()))
        self.amp2Edit.setText(("{0}").format(self.signal2.amplitude()))

    def freq1EditText(self):
        try:
            newfreq = float(self.freq1Edit.text())
            self.signal1.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")

    def amp1EditText(self):
        try:
            newamp = float(self.amp1Edit.text())
            self.signal1.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")
    def freq2EditText(self):
        try:
            newfreq = float(self.freq2Edit.text())
            self.signal2.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")

    def amp2EditText(self):
        try:
            newamp = float(self.amp2Edit.text())
            self.signal2.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")


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

        Rs = 8000
        f1 = 1000
        f2 = 2000

        fftsize = 2048

        self.qapp = QtWidgets.QApplication(sys.argv)

        src1 = analog.sig_source_f(Rs, analog.GR_SIN_WAVE, f1, 0.1, 0)
        src2 = analog.sig_source_f(Rs, analog.GR_SIN_WAVE, f2, 0.1, 0)
        src = blocks.add_ff()
        thr = blocks.throttle(gr.sizeof_float, 100 * fftsize)
        noise = analog.noise_source_f(analog.GR_GAUSSIAN, 0.001)
        add = blocks.add_ff()
        self.snk1 = qtgui.sink_f(fftsize, window.WIN_BLACKMAN_hARRIS,
                                 0, Rs,
                                 "Float Signal Example",
                                 True, True, True, False, None)

        self.connect(src1, (src, 0))
        self.connect(src2, (src, 1))
        self.connect(src, thr, (add, 0))
        self.connect(noise, (add, 1))
        self.connect(add, self.snk1)
        self.ctrl_win = control_box()
        self.ctrl_win.attach_signal1(src1)
        self.ctrl_win.attach_signal2(src2)

        # Get the reference pointer to the SpectrumDisplayForm QWidget
        pyQt = self.snk1.qwidget()

        # Wrap the pointer as a PyQt SIP object
        # This can now be manipulated as a PyQt5.QtWidgets.QWidget
        pyWin = sip.wrapinstance(pyQt, QtWidgets.QWidget)

        self.main_box = dialog_box(pyWin, self.ctrl_win)

        self.main_box.show()


if __name__ == "__main__":
    tb = my_top_block()
    tb.start()
    tb.qapp.exec_()
    tb.stop()

# ==================== END OF FILE: gr-qtgui/examples/pyqt_example_f.py ====================


# ==================== START OF FILE: gr-qtgui/examples/pyqt_freq_c.py ====================
#!/usr/bin/env python
#
# Copyright 2012,2015 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr, filter
from gnuradio import blocks
from gnuradio.fft import window
import sys
```

```
try:
    from gnuradio import qtgui
    from PyQt5 import QtWidgets, Qt
    import sip
except ImportError:
    sys.stderr.write("Error: Program requires PyQt5 and gr-qtgui.\n")
    sys.exit(1)

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)

try:
    from gnuradio import channels
except ImportError:
    sys.stderr.write("Error: Program requires gr-channels.\n")
    sys.exit(1)


class dialog_box(QtWidgets.QWidget):
    def __init__(self, display, control):
        QtWidgets.QWidget.__init__(self, None)
        self.setWindowTitle('PyQt Test GUI')

        self.boxlayout = QtWidgets.QBoxLayout(
            QtWidgets.QBoxLayout.LeftToRight, self)
        self.boxlayout.addWidget(display, 1)
        self.boxlayout.addWidget(control)

        self.resize(800, 500)


class control_box(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle('Control Panel')

        self.setToolTip('Control the signals')
        QtWidgets.QToolTip.setFont(Qt.QFont('OldEnglish', 10))

        self.layout = QtWidgets.QFormLayout(self)
        # Control the first signal
        self.freq1Edit = QtWidgets.QLineEdit(self)
        self.freq1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Frequency:", self.freq1Edit)
        self.freq1Edit.editingFinished.connect(self.freq1EditText)

        self.amp1Edit = QtWidgets.QLineEdit(self)
        self.amp1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Amplitude:", self.amp1Edit)
        self.amp1Edit.editingFinished.connect(self.amp1EditText)

        # Control the second signal
        self.freq2Edit = QtWidgets.QLineEdit(self)
        self.freq2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Frequency:", self.freq2Edit)
        self.freq2Edit.editingFinished.connect(self.freq2EditText)

        self.amp2Edit = QtWidgets.QLineEdit(self)
        self.amp2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Amplitude:", self.amp2Edit)
        self.amp2Edit.editingFinished.connect(self.amp2EditText)

        self.quit = QtWidgets.QPushButton('Close', self)
        self.quit.setMinimumWidth(100)
        self.layout.addWidget(self.quit)

        self.quit.clicked.connect(QtWidgets.qApp.quit)

    def attach_signal1(self, signal):
        self.signal1 = signal
        self.freq1Edit.setText(("{0}").format(self.signal1.frequency()))
        self.amp1Edit.setText(("{0}").format(self.signal1.amplitude()))

    def attach_signal2(self, signal):
        self.signal2 = signal
        self.freq2Edit.setText(("{0}").format(self.signal2.frequency()))
        self.amp2Edit.setText(("{0}").format(self.signal2.amplitude()))

    def freq1EditText(self):
        try:
            newfreq = float(self.freq1Edit.text())
            self.signal1.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")
    def amp1EditText(self):
        try:
            newamp = float(self.amp1Edit.text())
            self.signal1.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")

    def freq2EditText(self):
        try:
            newfreq = float(self.freq2Edit.text())
            self.signal2.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")

    def amp2EditText(self):
        try:
            newamp = float(self.amp2Edit.text())
            self.signal2.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")


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

        Rs = 8000
        f1 = 100
        f2 = 200

        npts = 2048

        self.qapp = QtWidgets.QApplication(sys.argv)
        ss = open(gr.prefix() + '/share/gnuradio/themes/dark.qss')
        sstext = ss.read()
        ss.close()
        self.qapp.setStyleSheet(sstext)

        src1 = analog.sig_source_c(Rs, analog.GR_SIN_WAVE, f1, 0.1, 0)
        src2 = analog.sig_source_c(Rs, analog.GR_SIN_WAVE, f2, 0.1, 0)
        src = blocks.add_cc()
        channel = channels.channel_model(0.01)
        thr = blocks.throttle(gr.sizeof_gr_complex, 100 * npts)
        self.snk1 = qtgui.freq_sink_c(npts, window.WIN_BLACKMAN_hARRIS,
                                      0, Rs,
                                      "Complex Freq Example", 3, None)

        self.connect(src1, (src, 0))
        self.connect(src2, (src, 1))
        self.connect(src, channel, thr, (self.snk1, 0))
        self.connect(src1, (self.snk1, 1))
        self.connect(src2, (self.snk1, 2))

        self.ctrl_win = control_box()
        self.ctrl_win.attach_signal1(src1)
        self.ctrl_win.attach_signal2(src2)

        # Get the reference pointer to the SpectrumDisplayForm QWidget
        pyQt = self.snk1.qwidget()

        # Wrap the pointer as a PyQt SIP object
        # This can now be manipulated as a PyQt5.QtWidgets.QWidget
        pyWin = sip.wrapinstance(pyQt, QtWidgets.QWidget)

        # pyWin.show()
        self.main_box = dialog_box(pyWin, self.ctrl_win)
        self.main_box.show()


if __name__ == "__main__":
    tb = my_top_block()
    tb.start()
    tb.qapp.exec_()
    tb.stop()

# ==================== END OF FILE: gr-qtgui/examples/pyqt_freq_c.py ====================


# ==================== START OF FILE: gr-qtgui/examples/pyqt_freq_f.py ====================
#!/usr/bin/env python
#
# Copyright 2012,2015 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
```

```
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr, filter
from gnuradio import blocks
from gnuradio.fft import window
import sys

try:
    from gnuradio import qtgui
    from PyQt5 import QtWidgets, Qt
    import sip
except ImportError:
    sys.stderr.write("Error: Program requires PyQt5 and gr-qtgui.\n")
    sys.exit(1)

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)


class dialog_box(QtWidgets.QWidget):
    def __init__(self, display, control):
        QtWidgets.QWidget.__init__(self, None)
        self.setWindowTitle('PyQt Test GUI')

        self.boxlayout = QtWidgets.QBoxLayout(
            QtWidgets.QBoxLayout.LeftToRight, self)
        self.boxlayout.addWidget(display, 1)
        self.boxlayout.addWidget(control)

        self.resize(800, 500)


class control_box(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle('Control Panel')

        self.setToolTip('Control the signals')
        QtWidgets.QToolTip.setFont(Qt.QFont('OldEnglish', 10))
        self.layout = QtWidgets.QFormLayout(self)

        # Control the first signal
        self.freq1Edit = QtWidgets.QLineEdit(self)
        self.freq1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Frequency:", self.freq1Edit)
        self.freq1Edit.editingFinished.connect(self.freq1EditText)

        self.amp1Edit = QtWidgets.QLineEdit(self)
        self.amp1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Amplitude:", self.amp1Edit)
        self.amp1Edit.editingFinished.connect(self.amp1EditText)

        # Control the second signal
        self.freq2Edit = QtWidgets.QLineEdit(self)
        self.freq2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Frequency:", self.freq2Edit)
        self.freq2Edit.editingFinished.connect(self.freq2EditText)

        self.amp2Edit = QtWidgets.QLineEdit(self)
        self.amp2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Amplitude:", self.amp2Edit)
        self.amp2Edit.editingFinished.connect(self.amp2EditText)

        self.quit = QtWidgets.QPushButton('Close', self)
        self.quit.setMinimumWidth(100)
        self.layout.addWidget(self.quit)

        self.quit.clicked.connect(QtWidgets.qApp.quit)

    def attach_signal1(self, signal):
        self.signal1 = signal
        self.freq1Edit.setText(("{0}").format(self.signal1.frequency()))
        self.amp1Edit.setText(("{0}").format(self.signal1.amplitude()))

    def attach_signal2(self, signal):
        self.signal2 = signal
        self.freq2Edit.setText(("{0}").format(self.signal2.frequency()))
        self.amp2Edit.setText(("{0}").format(self.signal2.amplitude()))

    def freq1EditText(self):
        try:
            newfreq = float(self.freq1Edit.text())
```

```
                self.signal1.set_frequency(newfreq)
            except ValueError:
                print("Bad frequency value entered")

    def amp1EditText(self):
        try:
            newamp = float(self.amp1Edit.text())
            self.signal1.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")

    def freq2EditText(self):
        try:
            newfreq = float(self.freq2Edit.text())
            self.signal2.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")

    def amp2EditText(self):
        try:
            newamp = float(self.amp2Edit.text())
            self.signal2.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")


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

        Rs = 8000
        f1 = 100
        f2 = 200

        npts = 2048

        self.qapp = QtWidgets.QApplication(sys.argv)

        src1 = analog.sig_source_f(Rs, analog.GR_SIN_WAVE, f1, 0.1, 0)
        src2 = analog.sig_source_f(Rs, analog.GR_SIN_WAVE, f2, 0.1, 0)
        src = blocks.add_ff()
        thr = blocks.throttle(gr.sizeof_float, 100 * npts)
        self.snk1 = qtgui.freq_sink_f(npts, window.WIN_BLACKMAN_hARRIS,
                                      0, Rs,
                                                    "Real freq Example", 3, None)

        self.connect(src1, (src, 0))
        self.connect(src2, (src, 1))
        self.connect(src, thr, (self.snk1, 0))
        self.connect(src1, (self.snk1, 1))
        self.connect(src2, (self.snk1, 2))

        self.ctrl_win = control_box()
        self.ctrl_win.attach_signal1(src1)
        self.ctrl_win.attach_signal2(src2)

        # Get the reference pointer to the SpectrumDisplayForm QWidget
        pyQt = self.snk1.qwidget()

        # Wrap the pointer as a PyQt SIP object
        # This can now be manipulated as a PyQt5.QtWidgets.QWidget
        pyWin = sip.wrapinstance(pyQt, QtWidgets.QWidget)

        # pyWin.show()
        self.main_box = dialog_box(pyWin, self.ctrl_win)
        self.main_box.show()


if __name__ == "__main__":
    tb = my_top_block()
    tb.start()
    tb.qapp.exec_()
    tb.stop()

# ==================== END OF FILE: gr-qtgui/examples/pyqt_freq_f.py ====================


# ==================== START OF FILE: gr-qtgui/examples/pyqt_histogram_f.py ====================
#!/usr/bin/env python
#
# Copyright 2013,2015 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
```

```
#

from gnuradio import gr
from gnuradio import blocks
import sys

try:
    from gnuradio import qtgui
    from PyQt5 import QtWidgets, Qt
    import sip
except ImportError:
    sys.stderr.write("Error: Program requires PyQt5 and gr-qtgui.\n")
    sys.exit(1)

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)


class dialog_box(QtWidgets.QWidget):
    def __init__(self, display, control):
        QtWidgets.QWidget.__init__(self, None)
        self.setWindowTitle('PyQt Test GUI')

        self.boxlayout = QtWidgets.QBoxLayout(
            QtWidgets.QBoxLayout.LeftToRight, self)
        self.boxlayout.addWidget(display, 1)
        self.boxlayout.addWidget(control)

        self.resize(800, 500)


class control_box(QtWidgets.QWidget):
    def __init__(self, snk, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle('Control Panel')
        self.snk = snk

        self.setToolTip('Control the signals')
        QtWidgets.QToolTip.setFont(Qt.QFont('OldEnglish', 10))

        self.layout = QtWidgets.QFormLayout(self)
        # Control the first signal
        self.freq1Edit = QtWidgets.QLineEdit(self)
        self.freq1Edit.setMinimumWidth(100)
        self.layout.addRow("Sine Frequency:", self.freq1Edit)
        self.freq1Edit.editingFinished.connect(self.freq1EditText)

        self.amp1Edit = QtWidgets.QLineEdit(self)
        self.amp1Edit.setMinimumWidth(100)
        self.layout.addRow("Sine Amplitude:", self.amp1Edit)
        self.amp1Edit.editingFinished.connect(self.amp1EditText)

        # Control the second signal
        self.amp2Edit = QtWidgets.QLineEdit(self)
        self.amp2Edit.setMinimumWidth(100)
        self.layout.addRow("Noise Amplitude:", self.amp2Edit)
        self.amp2Edit.editingFinished.connect(self.amp2EditText)

        # Control the histogram
        self.hist_npts = QtWidgets.QLineEdit(self)
        self.hist_npts.setMinimumWidth(100)
        self.hist_npts.setValidator(Qt.QIntValidator(0, 8191))
        self.hist_npts.setText("{0}".format(self.snk.nsamps()))
        self.layout.addRow("Number of Points:", self.hist_npts)
        self.hist_npts.editingFinished.connect(self.set_nsamps)

        self.hist_bins = QtWidgets.QLineEdit(self)
        self.hist_bins.setMinimumWidth(100)
        self.hist_bins.setValidator(Qt.QIntValidator(0, 1000))
        self.hist_bins.setText("{0}".format(self.snk.bins()))
        self.layout.addRow("Number of Bins:", self.hist_bins)
        self.hist_bins.editingFinished.connect(self.set_bins)

        self.hist_auto = QtWidgets.QPushButton("scale", self)
        self.layout.addRow("Autoscale X:", self.hist_auto)
        self.hist_auto.pressed.connect(self.autoscalex)

        self.quit = QtWidgets.QPushButton('Close', self)
        self.quit.setMinimumWidth(100)
        self.layout.addWidget(self.quit)

        self.quit.clicked.connect(QtWidgets.qApp.quit)

    def attach_signal1(self, signal):
        self.signal1 = signal
        self.freq1Edit.setText(("{0}").format(self.signal1.frequency()))
        self.amp1Edit.setText(("{0}").format(self.signal1.amplitude()))

    def attach_signal2(self, signal):
        self.signal2 = signal
        self.amp2Edit.setText(("{0}").format(self.signal2.amplitude()))

    def freq1EditText(self):
        try:
            newfreq = float(self.freq1Edit.text())
            self.signal1.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")

    def amp1EditText(self):
        try:
            newamp = float(self.amp1Edit.text())
            self.signal1.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")

    def amp2EditText(self):
        try:
            newamp = float(self.amp2Edit.text())
            self.signal2.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")

    def set_nsamps(self):
        res = self.hist_npts.text().toInt()
        if(res[1]):
            self.snk.set_nsamps(res[0])

    def set_bins(self):
        res = self.hist_bins.text().toInt()
        if(res[1]):
            self.snk.set_bins(res[0])

    def autoscalex(self):
        self.snk.autoscalex()


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

        Rs = 8000
        f1 = 100

        npts = 2048

        self.qapp = QtWidgets.QApplication(sys.argv)

        src1 = analog.sig_source_f(Rs, analog.GR_SIN_WAVE, f1, 0, 0)
        src2 = analog.noise_source_f(analog.GR_GAUSSIAN, 1)
        src = blocks.add_ff()
        thr = blocks.throttle(gr.sizeof_float, 100 * npts)
        self.snk1 = qtgui.histogram_sink_f(npts, 200, -5, 5,
                                           "Histogram", 1, None)
        self.snk1.disable_legend()

        self.connect(src1, (src, 0))
        self.connect(src2, (src, 1))
        self.connect(src, thr, self.snk1)

        self.ctrl_win = control_box(self.snk1)
        self.ctrl_win.attach_signal1(src1)
        self.ctrl_win.attach_signal2(src2)

        # Get the reference pointer to the SpectrumDisplayForm QWidget
        pyQt = self.snk1.qwidget()

        # Wrap the pointer as a PyQt SIP object
        # This can now be manipulated as a PyQt5.QtWidgets.QWidget
        pyWin = sip.wrapinstance(pyQt, QtWidgets.QWidget)

        # pyWin.show()
        self.main_box = dialog_box(pyWin, self.ctrl_win)
        self.main_box.show()


if __name__ == "__main__":
    tb = my_top_block()
    tb.start()
    tb.qapp.exec_()
    tb.stop()
```

```
# ==================== END OF FILE: gr-qtgui/examples/pyqt_histogram_f.py
====================


# ==================== START OF FILE: gr-qtgui/examples/pyqt_time_c.py
====================
#!/usr/bin/env python
#
# Copyright 2011,2012,2015 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import blocks
import sys

try:
    from gnuradio import qtgui
    from PyQt5 import QtWidgets, Qt
    import sip
except ImportError:
    sys.stderr.write("Error: Program requires PyQt5 and gr-qtgui.\n")
    sys.exit(1)

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)

try:
    from gnuradio import channels
except ImportError:
    sys.stderr.write("Error: Program requires gr-channels.\n")
    sys.exit(1)


class dialog_box(QtWidgets.QWidget):
    def __init__(self, display, control):
        QtWidgets.QWidget.__init__(self, None)
        self.setWindowTitle('PyQt Test GUI')

        self.boxlayout = QtWidgets.QBoxLayout(
            QtWidgets.QBoxLayout.LeftToRight, self)
        self.boxlayout.addWidget(display, 1)
        self.boxlayout.addWidget(control)

        self.resize(800, 500)


class control_box(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle('Control Panel')

        self.setToolTip('Control the signals')
        QtWidgets.QToolTip.setFont(Qt.QFont('OldEnglish', 10))

        self.layout = QtWidgets.QFormLayout(self)

        # Control the first signal
        self.freq1Edit = QtWidgets.QLineEdit(self)
        self.freq1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Frequency:", self.freq1Edit)
        self.freq1Edit.editingFinished.connect(self.freq1EditText)

        self.amp1Edit = QtWidgets.QLineEdit(self)
        self.amp1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Amplitude:", self.amp1Edit)
        self.amp1Edit.editingFinished.connect(self.amp1EditText)

        # Control the second signal
        self.freq2Edit = QtWidgets.QLineEdit(self)
        self.freq2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Frequency:", self.freq2Edit)
        self.freq2Edit.editingFinished.connect(self.freq2EditText)

        self.amp2Edit = QtWidgets.QLineEdit(self)
        self.amp2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Amplitude:", self.amp2Edit)
        self.amp2Edit.editingFinished.connect(self.amp2EditText)

        self.quit = QtWidgets.QPushButton('Close', self)
        self.quit.setMinimumWidth(100)
        self.layout.addWidget(self.quit)

        self.quit.clicked.connect(QtWidgets.qApp.quit)

    def attach_signal1(self, signal):
        self.signal1 = signal
        self.freq1Edit.setText(("{0}").format(self.signal1.frequency()))
        self.amp1Edit.setText(("{0}").format(self.signal1.amplitude()))

    def attach_signal2(self, signal):
        self.signal2 = signal
        self.freq2Edit.setText(("{0}").format(self.signal2.frequency()))
        self.amp2Edit.setText(("{0}").format(self.signal2.amplitude()))

    def freq1EditText(self):
        try:
            newfreq = float(self.freq1Edit.text())
            self.signal1.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")

    def amp1EditText(self):
        try:
            newamp = float(self.amp1Edit.text())
            self.signal1.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")

    def freq2EditText(self):
        try:
            newfreq = float(self.freq2Edit.text())
            self.signal2.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")

    def amp2EditText(self):
        try:
            newamp = float(self.amp2Edit.text())
            self.signal2.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")


class my_top_block(gr.top_block):
```

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

        Rs = 8000
        f1 = 100
        f2 = 200

        npts = 2048

        self.qapp = QtWidgets.QApplication(sys.argv)
        ss = open(gr.prefix() + '/share/gnuradio/themes/dark.qss')
        sstext = ss.read()
        ss.close()
        self.qapp.setStyleSheet(sstext)

        src1 = analog.sig_source_c(Rs, analog.GR_SIN_WAVE, f1, 0.1, 0)
        src2 = analog.sig_source_c(Rs, analog.GR_SIN_WAVE, f2, 0.1, 0)
        src = blocks.add_cc()
        channel = channels.channel_model(0.01)
        thr = blocks.throttle(gr.sizeof_gr_complex, 100 * npts)
        self.snk1 = qtgui.time_sink_c(npts, Rs,
                                      "Complex Time Example", 1, None)

        self.connect(src1, (src, 0))
        self.connect(src2, (src, 1))
        self.connect(src, channel, thr, (self.snk1, 0))
        #self.connect(src1, (self.snk1, 1))
        #self.connect(src2, (self.snk1, 2))

        self.ctrl_win = control_box()
        self.ctrl_win.attach_signal1(src1)
        self.ctrl_win.attach_signal2(src2)

        # Get the reference pointer to the SpectrumDisplayForm QWidget
        pyQt = self.snk1.qwidget()

        # Wrap the pointer as a PyQt SIP object
        # This can now be manipulated as a PyQt5.QtWidgets.QWidget
        pyWin = sip.wrapinstance(pyQt, QtWidgets.QWidget)

        # Example of using signal/slot to set the title of a curve
        # FIXME: update for Qt5
        # pyWin.setLineLabel.connect(pyWin.setLineLabel)
        #pyWin.emit(QtCore.SIGNAL("setLineLabel(int, QString)"), 0, "Re{sum}")
        self.snk1.set_line_label(0, "Re{Sum}")
        self.snk1.set_line_label(1, "Im{Sum}")
        #self.snk1.set_line_label(2, "Re{src1}")
        #self.snk1.set_line_label(3, "Im{src1}")
        #self.snk1.set_line_label(4, "Re{src2}")
        #self.snk1.set_line_label(5, "Im{src2}")

        # Can also set the color of a curve
        #self.snk1.set_color(5, "blue")

        self.snk1.set_update_time(0.5)

        # pyWin.show()
        self.main_box = dialog_box(pyWin, self.ctrl_win)
        self.main_box.show()


if __name__ == "__main__":
    tb = my_top_block()
    tb.start()
    tb.qapp.exec_()
    tb.stop()

# ==================== END OF FILE: gr-qtgui/examples/pyqt_time_c.py ====================


# ==================== START OF FILE: gr-qtgui/examples/pyqt_time_f.py ====================
#!/usr/bin/env python
#
# Copyright 2011,2012,2015 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import blocks
import sys

try:
    from gnuradio import qtgui
    from PyQt5 import QtWidgets, Qt
    import sip
except ImportError:
    sys.stderr.write("Error: Program requires PyQt5 and gr-qtgui.\n")
    sys.exit(1)

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)


class dialog_box(QtWidgets.QWidget):
    def __init__(self, display, control):
        QtWidgets.QWidget.__init__(self, None)
        self.setWindowTitle('PyQt Test GUI')

        self.boxlayout = QtWidgets.QBoxLayout(
            QtWidgets.QBoxLayout.LeftToRight, self)
        self.boxlayout.addWidget(display, 1)
        self.boxlayout.addWidget(control)

        self.resize(800, 500)


class control_box(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle('Control Panel')

        self.setToolTip('Control the signals')
        QtWidgets.QToolTip.setFont(Qt.QFont('OldEnglish', 10))

        self.layout = QtWidgets.QFormLayout(self)

        # Control the first signal
        self.freq1Edit = QtWidgets.QLineEdit(self)
        self.freq1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Frequency:", self.freq1Edit)
        self.freq1Edit.editingFinished.connect(self.freq1EditText)

        self.amp1Edit = QtWidgets.QLineEdit(self)
        self.amp1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Amplitude:", self.amp1Edit)
        self.amp1Edit.editingFinished.connect(self.amp1EditText)

        # Control the second signal
        self.freq2Edit = QtWidgets.QLineEdit(self)
        self.freq2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Frequency:", self.freq2Edit)
        self.freq2Edit.editingFinished.connect(self.freq2EditText)

        self.amp2Edit = QtWidgets.QLineEdit(self)
        self.amp2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Amplitude:", self.amp2Edit)
        self.amp2Edit.editingFinished.connect(self.amp2EditText)

        self.quit = QtWidgets.QPushButton('Close', self)
        self.quit.setMinimumWidth(100)
        self.layout.addWidget(self.quit)

        self.quit.clicked.connect(QtWidgets.qApp.quit)

    def attach_signal1(self, signal):
        self.signal1 = signal
        self.freq1Edit.setText(("{0}").format(self.signal1.frequency()))
        self.amp1Edit.setText(("{0}").format(self.signal1.amplitude()))

    def attach_signal2(self, signal):
        self.signal2 = signal
        self.freq2Edit.setText(("{0}").format(self.signal2.frequency()))
        self.amp2Edit.setText(("{0}").format(self.signal2.amplitude()))

    def freq1EditText(self):
        try:
            newfreq = float(self.freq1Edit.text())
            self.signal1.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")

    def amp1EditText(self):
        try:
            newamp = float(self.amp1Edit.text())
            self.signal1.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")
    def freq2EditText(self):
        try:
            newfreq = float(self.freq2Edit.text())
            self.signal2.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")

    def amp2EditText(self):
        try:
            newamp = float(self.amp2Edit.text())
            self.signal2.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")


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

        Rs = 8000
        f1 = 100
        f2 = 200

        npts = 2048

        self.qapp = QtWidgets.QApplication(sys.argv)

        src1 = analog.sig_source_f(Rs, analog.GR_SIN_WAVE, f1, 0.1, 0)
        src2 = analog.sig_source_f(Rs, analog.GR_SIN_WAVE, f2, 0.1, 0)
        src = blocks.add_ff()
        thr = blocks.throttle(gr.sizeof_float, 100 * npts)
        noise = analog.noise_source_f(analog.GR_GAUSSIAN, 0.001)
        add = blocks.add_ff()
        self.snk1 = qtgui.time_sink_f(npts, Rs,
                                      "Complex Time Example", 3, None)

        self.connect(src1, (src, 0))
        self.connect(src2, (src, 1))
        self.connect(src, thr, (add, 0))
        self.connect(noise, (add, 1))
        self.connect(add, self.snk1)
        self.connect(src1, (self.snk1, 1))
        self.connect(src2, (self.snk1, 2))
        self.ctrl_win = control_box()
        self.ctrl_win.attach_signal1(src1)
        self.ctrl_win.attach_signal2(src2)

        # Get the reference pointer to the SpectrumDisplayForm QWidget
        pyQt = self.snk1.qwidget()

        # Wrap the pointer as a PyQt SIP object
        # This can now be manipulated as a PyQt5.QtWidgets.QWidget
        pyWin = sip.wrapinstance(pyQt, QtWidgets.QWidget)

        # Example of using signal/slot to set the title of a curve
        # FIXME: update for Qt5
        # pyWin.setLineLabel.connect(pyWin.setLineLabel)
        #pyWin.emit(QtCore.SIGNAL("setLineLabel(int, QString)"), 0, "Re{sum}")
        self.snk1.set_line_label(0, "Re{sum}")
        self.snk1.set_line_label(1, "src1")
        self.snk1.set_line_label(2, "src2")

        # Can also set the color of a curve
        #self.snk1.set_color(5, "blue")

        # pyWin.show()
        self.main_box = dialog_box(pyWin, self.ctrl_win)
        self.main_box.show()


if __name__ == "__main__":
    tb = my_top_block()
    tb.start()
    tb.qapp.exec_()
    tb.stop()

# ==================== END OF FILE: gr-qtgui/examples/pyqt_time_f.py ====================


# ==================== START OF FILE: gr-qtgui/examples/pyqt_time_raster_b.py ====================
#!/usr/bin/env python
#
# Copyright 2012,2013,2015 Free Software Foundation, Inc.
#
```

```
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import blocks
from gnuradio import blocks
import sys

try:
    from gnuradio import qtgui
    from PyQt5 import QtWidgets, Qt
    import sip
except ImportError:
    print("Error: Program requires PyQt5 and gr-qtgui.")
    sys.exit(1)


class dialog_box(QtWidgets.QWidget):
    def __init__(self, display):
        QtWidgets.QWidget.__init__(self, None)
        self.setWindowTitle('PyQt Test GUI')

        self.boxlayout = QtWidgets.QBoxLayout(
            QtWidgets.QBoxLayout.LeftToRight, self)
        self.boxlayout.addWidget(display, 1)

        self.resize(800, 500)


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

        self.qapp = QtWidgets.QApplication(sys.argv)

        data0 = 10 * [0, ] + 40 * [1, 0] + 10 * [0, ]
        data0 += 10 * [0, ] + 40 * [0, 1] + 10 * [0, ]
        data1 = 20 * [0, ] + [0, 0, 0, 1, 1, 1, 0, 0, 0, 0] + 70 * [0, ]

        # Adjust these to change the layout of the plot.
        # Can be set to fractions.
        ncols = 100.25
        nrows = 100

        fs = 200
        src0 = blocks.vector_source_b(data0, True)
        src1 = blocks.vector_source_b(data1, True)
        thr = blocks.throttle(gr.sizeof_char, 50000)
        head = blocks.head(gr.sizeof_char, 10000000)
        self.snk1 = qtgui.time_raster_sink_b(fs, nrows, ncols, [], [],
                                             "Time Raster Example", 2, None)

        self.connect(src0, thr, (self.snk1, 0))
        self.connect(src1, (self.snk1, 1))

        # Get the reference pointer to the SpectrumDisplayForm QWidget
        pyQt = self.snk1.qwidget()

        # Wrap the pointer as a PyQt SIP object
        # This can now be manipulated as a PyQt5.QtWidgets.QWidget
        pyWin = sip.wrapinstance(pyQt, QtWidgets.QWidget)

        self.main_box = dialog_box(pyWin)
        self.main_box.show()


if __name__ == "__main__":
    tb = my_top_block()
    tb.start()
    tb.qapp.exec_()
    tb.stop()

# ==================== END OF FILE: gr-qtgui/examples/pyqt_time_raster_b.py ====================



# ==================== START OF FILE: gr-qtgui/examples/pyqt_time_raster_f.py ====================
#!/usr/bin/env python
#
# Copyright 2012,2013,2015 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
```

```
#
#

from gnuradio import gr
from gnuradio import blocks
import sys

try:
        from gnuradio import qtgui
        from PyQt5 import QtWidgets, Qt
        import sip
except ImportError:
        print("Error: Program requires PyQt5 and gr-qtgui.")
        sys.exit(1)


class dialog_box(QtWidgets.QWidget):
        def __init__(self, display):
                QtWidgets.QWidget.__init__(self, None)
                self.setWindowTitle('PyQt Test GUI')

                self.boxlayout = QtWidgets.QBoxLayout(
                        QtWidgets.QBoxLayout.LeftToRight, self)
                self.boxlayout.addWidget(display, 1)

                self.resize(800, 500)


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

                self.qapp = QtWidgets.QApplication(sys.argv)

                data0 = 10 * [0, ] + 40 * [1, 0] + 10 * [0, ]
                data0 += 10 * [0, ] + 40 * [0, 1] + 10 * [0, ]
                data1 = 20 * [0, ] + [0, 0, 0, 1, 1, 1, 0, 0, 0, 0] + 70 * [0, ]

                # Adjust these to change the layout of the plot.
                # Can be set to fractions.
                ncols = 100.25
                nrows = 100

                fs = 200
                src0 = blocks.vector_source_f(data0, True)
                src1 = blocks.vector_source_f(data1, True)
                thr = blocks.throttle(gr.sizeof_float, 50000)
                hed = blocks.head(gr.sizeof_float, 10000000)
                self.snk1 = qtgui.time_raster_sink_f(fs, nrows, ncols, [], [],
                                                     "Float Time Raster Example", 2, None)

                self.connect(src0, thr, (self.snk1, 0))
                self.connect(src1, (self.snk1, 1))

                # Get the reference pointer to the SpectrumDisplayForm QWidget
                pyQt = self.snk1.qwidget()

                # Wrap the pointer as a PyQt SIP object
                # This can now be manipulated as a PyQt5.QtWidgets.QWidget
                pyWin = sip.wrapinstance(pyQt, QtWidgets.QWidget)

                self.main_box = dialog_box(pyWin)
                self.main_box.show()


if __name__ == "__main__":
        tb = my_top_block()
        tb.start()
        tb.qapp.exec_()
        tb.stop()

# ==================== END OF FILE: gr-qtgui/examples/pyqt_time_raster_f.py ====================


# ==================== START OF FILE: gr-qtgui/examples/pyqt_waterfall_c.py ====================
#!/usr/bin/env python
#
# Copyright 2012,2015 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr, filter
from gnuradio import blocks
from gnuradio.fft import window
import sys

try:
        from gnuradio import qtgui
        from PyQt5 import QtWidgets, Qt
        import sip
except ImportError:
        sys.stderr.write("Error: Program requires PyQt5 and gr-qtgui.\n")
        sys.exit(1)

try:
        from gnuradio import analog
except ImportError:
        sys.stderr.write("Error: Program requires gr-analog.\n")
        sys.exit(1)

try:
        from gnuradio import channels
except ImportError:
        sys.stderr.write("Error: Program requires gr-channels.\n")
        sys.exit(1)


class dialog_box(QtWidgets.QWidget):
        def __init__(self, display, control):
                QtWidgets.QWidget.__init__(self, None)
                self.setWindowTitle('PyQt Test GUI')

                self.boxlayout = QtWidgets.QBoxLayout(
                        QtWidgets.QBoxLayout.LeftToRight, self)
                self.boxlayout.addWidget(display, 1)
                self.boxlayout.addWidget(control)

                self.resize(800, 500)


class control_box(QtWidgets.QWidget):
        def __init__(self, parent=None):
                QtWidgets.QWidget.__init__(self, parent)
                self.setWindowTitle('Control Panel')

                self.setToolTip('Control the signals')
                QtWidgets.QToolTip.setFont(Qt.QFont('OldEnglish', 10))

                self.layout = QtWidgets.QFormLayout(self)

                # Control the first signal
                self.freq1Edit = QtWidgets.QLineEdit(self)
                self.freq1Edit.setMinimumWidth(100)
                self.layout.addRow("Signal 1 Frequency:", self.freq1Edit)
                self.freq1Edit.editingFinished.connect(self.freq1EditText)

                self.amp1Edit = QtWidgets.QLineEdit(self)
                self.amp1Edit.setMinimumWidth(100)
                self.layout.addRow("Signal 1 Amplitude:", self.amp1Edit)
                self.amp1Edit.editingFinished.connect(self.amp1EditText)

                # Control the second signal
                self.freq2Edit = QtWidgets.QLineEdit(self)
                self.freq2Edit.setMinimumWidth(100)
                self.layout.addRow("Signal 2 Frequency:", self.freq2Edit)
                self.freq2Edit.editingFinished.connect(self.freq2EditText)

                self.amp2Edit = QtWidgets.QLineEdit(self)
                self.amp2Edit.setMinimumWidth(100)
                self.layout.addRow("Signal 2 Amplitude:", self.amp2Edit)
                self.amp2Edit.editingFinished.connect(self.amp2EditText)

                self.quit = QtWidgets.QPushButton('Close', self)
                self.quit.setMinimumWidth(100)
                self.layout.addWidget(self.quit)

                self.quit.clicked.connect(QtWidgets.qApp.quit)

        def attach_signal1(self, signal):
                self.signal1 = signal
                self.freq1Edit.setText(("{0}").format(self.signal1.frequency()))
                self.amp1Edit.setText(("{0}").format(self.signal1.amplitude()))

        def attach_signal2(self, signal):
                self.signal2 = signal
                self.freq2Edit.setText(("{0}").format(self.signal2.frequency()))
                self.amp2Edit.setText(("{0}").format(self.signal2.amplitude()))

        def freq1EditText(self):
                try:
                    newfreq = float(self.freq1Edit.text())
                    self.signal1.set_frequency(newfreq)
                except ValueError:
                    print("Bad frequency value entered")

        def amp1EditText(self):
                try:
                    newamp = float(self.amp1Edit.text())
                    self.signal1.set_amplitude(newamp)
                except ValueError:
                    print("Bad amplitude value entered")

        def freq2EditText(self):
                try:
                    newfreq = float(self.freq2Edit.text())
                    self.signal2.set_frequency(newfreq)
                except ValueError:
                    print("Bad frequency value entered")

        def amp2EditText(self):
                try:
                    newamp = float(self.amp2Edit.text())
                    self.signal2.set_amplitude(newamp)
                except ValueError:
                    print("Bad amplitude value entered")


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

        Rs = 8000
        f1 = 100
        f2 = 2000

        npts = 2048

        taps = filter.firdes.complex_band_pass_2(1, Rs, 1500, 2500, 100, 60)

        self.qapp = QtWidgets.QApplication(sys.argv)
        ss = open(gr.prefix() + '/share/gnuradio/themes/dark.qss')
        sstext = ss.read()
        ss.close()
        self.qapp.setStyleSheet(sstext)
                src1 = analog.sig_source_c(Rs, analog.GR_SIN_WAVE, f1, 0.1, 0)
                src2 = analog.sig_source_c(Rs, analog.GR_SIN_WAVE, f2, 0.1, 0)
                src = blocks.add_cc()
                channel = channels.channel_model(0.01)
                thr = blocks.throttle(gr.sizeof_gr_complex, 100 * npts)
                filt = filter.fft_filter_ccc(1, taps)
                self.snk1 = qtgui.waterfall_sink_c(npts, window.WIN_BLACKMAN_hARRIS,
                                                   0, Rs,
                                                   "Complex Waterfall Example", 2, None)
                self.snk1.set_color_map(0, qtgui.INTENSITY_COLOR_MAP_TYPE_COOL)
                self.snk1.set_color_map(1, qtgui.INTENSITY_COLOR_MAP_TYPE_COOL)

                self.connect(src1, (src, 0))
                self.connect(src2, (src, 1))
                self.connect(src, channel, thr, (self.snk1, 0))
                self.connect(thr, filt, (self.snk1, 1))

                self.ctrl_win = control_box()
                self.ctrl_win.attach_signal1(src1)
                self.ctrl_win.attach_signal2(src2)

                # Get the reference pointer to the SpectrumDisplayForm QWidget
                pyQt = self.snk1.qwidget()

                # Wrap the pointer as a PyQt SIP object
                # This can now be manipulated as a PyQt5.QtWidgets.QWidget
                pyWin = sip.wrapinstance(pyQt, QtWidgets.QWidget)

                # pyWin.show()
                self.main_box = dialog_box(pyWin, self.ctrl_win)
                self.main_box.show()


if __name__ == "__main__":
    tb = my_top_block()
    tb.start()
    tb.qapp.exec_()
    tb.stop()

# ==================== END OF FILE: gr-qtgui/examples/pyqt_waterfall_c.py
====================
# ==================== START OF FILE: gr-qtgui/examples/pyqt_waterfall_f.py
====================
#!/usr/bin/env python
#
# Copyright 2012,2015 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr, filter
from gnuradio import blocks
from gnuradio.fft import window
import sys

try:
    from gnuradio import qtgui
    from PyQt5 import QtWidgets, Qt
    import sip
except ImportError:
    sys.stderr.write("Error: Program requires PyQt5 and gr-qtgui.\n")
    sys.exit(1)

try:
    from gnuradio import analog
except ImportError:
    sys.stderr.write("Error: Program requires gr-analog.\n")
    sys.exit(1)


class dialog_box(QtWidgets.QWidget):
    def __init__(self, display, control):
        QtWidgets.QWidget.__init__(self, None)
        self.setWindowTitle('PyQt Test GUI')

        self.boxlayout = QtWidgets.QBoxLayout(
            QtWidgets.QBoxLayout.LeftToRight, self)
        self.boxlayout.addWidget(display, 1)
        self.boxlayout.addWidget(control)

        self.resize(800, 500)
```

```
class control_box(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle('Control Panel')

        self.setToolTip('Control the signals')
        QtWidgets.QToolTip.setFont(Qt.QFont('OldEnglish', 10))

        self.layout = QtWidgets.QFormLayout(self)

        # Control the first signal
        self.freq1Edit = QtWidgets.QLineEdit(self)
        self.freq1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Frequency:", self.freq1Edit)
        self.freq1Edit.editingFinished.connect(self.freq1EditText)

        self.amp1Edit = QtWidgets.QLineEdit(self)
        self.amp1Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 1 Amplitude:", self.amp1Edit)
        self.amp1Edit.editingFinished.connect(self.amp1EditText)

        # Control the second signal
        self.freq2Edit = QtWidgets.QLineEdit(self)
        self.freq2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Frequency:", self.freq2Edit)
        self.freq2Edit.editingFinished.connect(self.freq2EditText)

        self.amp2Edit = QtWidgets.QLineEdit(self)
        self.amp2Edit.setMinimumWidth(100)
        self.layout.addRow("Signal 2 Amplitude:", self.amp2Edit)
        self.amp2Edit.editingFinished.connect(self.amp2EditText)

        self.quit = QtWidgets.QPushButton('Close', self)
        self.quit.setMinimumWidth(100)
        self.layout.addWidget(self.quit)

        self.quit.clicked.connect(QtWidgets.qApp.quit)

    def attach_signal1(self, signal):
        self.signal1 = signal
        self.freq1Edit.setText(("{0}").format(self.signal1.frequency()))
        self.amp1Edit.setText(("{0}").format(self.signal1.amplitude()))
    def attach_signal2(self, signal):
        self.signal2 = signal
        self.freq2Edit.setText(("{0}").format(self.signal2.frequency()))
        self.amp2Edit.setText(("{0}").format(self.signal2.amplitude()))

    def freq1EditText(self):
        try:
            newfreq = float(self.freq1Edit.text())
            self.signal1.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")

    def amp1EditText(self):
        try:
            newamp = float(self.amp1Edit.text())
            self.signal1.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")

    def freq2EditText(self):
        try:
            newfreq = float(self.freq2Edit.text())
            self.signal2.set_frequency(newfreq)
        except ValueError:
            print("Bad frequency value entered")

    def amp2EditText(self):
        try:
            newamp = float(self.amp2Edit.text())
            self.signal2.set_amplitude(newamp)
        except ValueError:
            print("Bad amplitude value entered")


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

        Rs = 8000
        f1 = 100
        f2 = 2000

        npts = 2048
        self.qapp = QtWidgets.QApplication(sys.argv)

        src1 = analog.sig_source_f(Rs, analog.GR_SIN_WAVE, f1, 0.1, 0)
        src2 = analog.sig_source_f(Rs, analog.GR_SIN_WAVE, f2, 0.1, 0)
        src = blocks.add_ff()
        thr = blocks.throttle(gr.sizeof_float, 100 * npts)
        self.snk1 = qtgui.waterfall_sink_f(npts, window.WIN_BLACKMAN_hARRIS,
                                           0, Rs,
                                           "Real Waterfall Example", 2, None)
        self.snk1.set_color_map(0, qtgui.INTENSITY_COLOR_MAP_TYPE_COOL)
        self.snk1.set_color_map(1, qtgui.INTENSITY_COLOR_MAP_TYPE_COOL)

        self.connect(src1, (src, 0))
        self.connect(src2, (src, 1))
        self.connect(src, thr, (self.snk1, 0))
        self.connect(src1, (self.snk1, 1))

        self.ctrl_win = control_box()
        self.ctrl_win.attach_signal1(src1)
        self.ctrl_win.attach_signal2(src2)

        # Get the reference pointer to the SpectrumDisplayForm QWidget
        pyQt = self.snk1.qwidget()

        # Wrap the pointer as a PyQt SIP object
        # This can now be manipulated as a PyQt5.QtWidgets.QWidget
        pyWin = sip.wrapinstance(pyQt, QtWidgets.QWidget)

        # pyWin.show()
        self.main_box = dialog_box(pyWin, self.ctrl_win)
        self.main_box.show()


if __name__ == "__main__":
    tb = my_top_block()
    tb.start()
    tb.qapp.exec_()
    tb.stop()

# ==================== END OF FILE: gr-qtgui/examples/pyqt_waterfall_f.py
====================


# ==================== START OF FILE: gr-trellis/examples/python/test_tcm.py
```

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

from gnuradio import gr
from gnuradio import trellis, digital, blocks
from gnuradio import eng_notation
import math
import sys
import random
from gnuradio.trellis import fsm_utils
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import numpy

try:
	from gnuradio import analog
except ImportError:
	sys.stderr.write("Error: Program requires gr-analog.\n")
	sys.exit(1)


def run_test(f, Kb, bitspersymbol, K, dimensionality, constellation, N0, seed):
	tb = gr.top_block()

	# TX
	numpy.random.seed(-seed)
	packet = numpy.random.randint(0, 2, Kb)   # create Kb random bits
	packet[Kb - 10:Kb] = 0
	packet[0:Kb] = 0
	src = blocks.vector_source_s(packet.tolist(), False)
	b2s = blocks.unpacked_to_packed_ss(
		1, gr.GR_MSB_FIRST)   # pack bits in shorts
	# unpack shorts to symbols compatible with the FSM input cardinality
	s2fsmi = blocks.packed_to_unpacked_ss(bitspersymbol, gr.GR_MSB_FIRST)
	enc = trellis.encoder_ss(f, 0)   # initial state = 0
	mod = digital.chunks_to_symbols_sf(constellation, dimensionality)

	# CHANNEL
	add = blocks.add_ff()
	noise = analog.noise_source_f(
		analog.GR_GAUSSIAN, math.sqrt(N0 / 2), int(seed))

	# RX
	# Put -1 if the Initial/Final states are not set.
```

```
    va = trellis.viterbi_combined_fs(
        f, K, 0, 0, dimensionality, constellation, digital.TRELLIS_EUCLIDEAN)
    fsmi2s = blocks.unpacked_to_packed_ss(
        bitspersymbol, gr.GR_MSB_FIRST)    # pack FSM input symbols to shorts
    s2b = blocks.packed_to_unpacked_ss(
        1, gr.GR_MSB_FIRST)    # unpack shorts to bits
    dst = blocks.vector_sink_s()

    tb.connect(src, b2s, s2fsmi, enc, mod)
    tb.connect(mod, (add, 0))
    tb.connect(noise, (add, 1))
    tb.connect(add, va, fsmi2s, s2b, dst)

    tb.run()

    # A bit of cheating: run the program once and print the
    # final encoder state..
    # Then put it as the last argument in the viterbi block
    # print "final state = " , enc.ST()

    if len(dst.data()) != len(packet):
        print("Error: not enough data:", len(dst.data()), len(packet))
    ntotal = len(packet)
    nwrong = sum(abs(packet - numpy.array(dst.data())))
    return (ntotal, nwrong, abs(packet - numpy.array(dst.data())))


def main():
    parser = OptionParser(option_class=eng_option)
    parser.add_option("-f", "--fsm_file", type="string", default="fsm_files/awgn1o2_4.fsm",
                      help="Filename containing the fsm specification, e.g. -f
fsm_files/awgn1o2_4.fsm (default=fsm_files/awgn1o2_4.fsm)")
    parser.add_option("-e", "--esn0", type="eng_float", default=10.0,
                      help="Symbol energy to noise PSD level ratio in dB, e.g., -e 10.0
(default=10.0)")
    parser.add_option("-r", "--repetitions", type="int", default=100,
                      help="Number of packets to be generated for the simulation, e.g.,
-r 100 (default=100)")

    (options, args) = parser.parse_args()
    if len(args) != 0:
        parser.print_help()
        raise SystemExit(1)
```

```
fname = options.fsm_file
esn0_db = float(options.esn0)
rep = int(options.repetitions)

# system parameters
f = trellis.fsm(fname)    # get the FSM specification from a file
# alternatively you can specify the fsm from its generator matrix
# f=trellis.fsm(1,2,[5,7])
# packet size in bits (make it multiple of 16 so it can be packed in a short)
Kb = 1024 * 16
# bits per FSM input symbol
bitspersymbol = int(round(math.log(f.I()) / math.log(2)))
K = Kb / bitspersymbol    # packet size in trellis steps
modulation = fsm_utils.psk4    # see fsm_utlis.py for available predefined modulations
dimensionality = modulation[0]
constellation = modulation[1]
if len(constellation) / dimensionality != f.O():
    sys.stderr.write(
        'Incompatible FSM output cardinality and modulation size.\n')
    sys.exit(1)
# calculate average symbol energy
Es = 0
for i in range(len(constellation)):
    Es = Es + constellation[i]**2
Es = Es / (len(constellation) // dimensionality)
N0 = Es / pow(10.0, esn0_db / 10.0)    # calculate noise variance

tot_b = 0    # total number of transmitted bits
terr_b = 0    # total number of bits in error
terr_p = 0    # total number of packets in error
for i in range(rep):
    # run experiment with different seed to get different noise realizations
    (b, e, pattern) = run_test(f, Kb, bitspersymbol,
                               K, dimensionality, constellation, N0, -(666 + i))
    tot_b = tot_b + b
    terr_b = terr_b + e
    terr_p = terr_p + (e != 0)
    if ((i + 1) % 100 == 0):    # display progress
        print(i + 1, terr_p, '%.2e' % ((1.0 * terr_p) / (i + 1)),
              tot_b, terr_b, '%.2e' % ((1.0 * terr_b) / tot_b))
    if e != 0:
        print("rep=", i, e)
        for k in range(Kb):
            if pattern[k] != 0:
```

```
                                print(k)
        # estimate of the bit error rate
        print(rep, terr_p, '%.2e' % ((1.0 * terr_p) / (i + 1)),
              tot_b, terr_b, '%.2e' % ((1.0 * terr_b) / tot_b))


if __name__ == '__main__':
    main()

# ==================== END OF FILE: gr-trellis/examples/python/test_tcm.py ====================


# ==================== START OF FILE: gr-uhd/examples/python/freq_hopping.py ====================
#!/usr/bin/env python
#
# Copyright 2014,2019 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

"""
TXs a waveform (either from a file, or a sinusoid) in a frequency-hopping manner.
"""

import argparse
import numpy
import pmt
from gnuradio import gr
from gnuradio import blocks
from gnuradio import uhd


def setup_parser():
    """ Setup the parser for the frequency hopper. """
    parser = argparse.ArgumentParser(
        description="Transmit a signal in a frequency-hopping manner, using tx_freq tags."
    )
    parser.add_argument(
        '-i', '--input-file',
        help="File with samples to transmit. If left out, will transmit a sinusoid.")
parser.add_argument(
        "-a", "--args", default="",
        help="UHD device address args.")
parser.add_argument(
        "--spec", default="",
        help="UHD subdev spec.")
parser.add_argument(
        "--antenna", default="",
        help="UHD antenna settings.")
parser.add_argument(
        "--gain", default=None, type=float,
        help="USRP gain (defaults to mid-point in dB).")
parser.add_argument(
        "-r", "--rate", type=float, default=1e6,
        help="Sampling rate")
parser.add_argument(
        "-N", "--samp-per-burst", type=int, default=10000,
        help="Samples per burst")
parser.add_argument(
        "-t", "--hop-time", type=float, default=1000,
        help="Time between hops in milliseconds. This must be larger than or "
             "equal to the burst duration as set by --samp-per-burst")
parser.add_argument(
        "-f", "--freq", type=float, default=2.45e9,
        help="Base frequency. This is the middle channel frequency at which "
             "the USRP will Tx.")
parser.add_argument(
        "--dsp", action='store_true',
        help="DSP tuning only.")
parser.add_argument(
        "-d", "--freq-delta", type=float, default=1e6,
        help="Channel spacing.")
parser.add_argument(
        "-c", "--num-channels", type=int, default=5,
        help="Number of channels.")
parser.add_argument(
        "-B", "--num-bursts", type=int, default=30,
        help="Number of bursts to transmit before terminating.")
parser.add_argument(
        "-p", "--post-tuning", action='count',
        help="Tune after transmitting. Default is to tune immediately before "
             "transmitting.")
parser.add_argument(
        "-v", "--verbose", action='count',
        help="Print more information. The morer the printier.")
    return parser


class FrequencyHopperSrc(gr.hier_block2):
    """ Provides tags for frequency hopping """

    def __init__(
        self,
        n_bursts, n_channels,
        freq_delta, base_freq, dsp_tuning,
        burst_length, base_time, hop_time,
        post_tuning=False,
        tx_gain=0,
        verbose=False
    ):
        gr.hier_block2.__init__(
            self, "FrequencyHopperSrc",
            gr.io_signature(1, 1, gr.sizeof_gr_complex),
            gr.io_signature(1, 1, gr.sizeof_gr_complex),
        )
        n_samples_total = n_bursts * burst_length
        lowest_frequency = base_freq - numpy.floor(n_channels / 2) * freq_delta
        self.hop_sequence = [lowest_frequency + n *
                             freq_delta for n in range(n_channels)]
        numpy.random.shuffle(self.hop_sequence)
        # Repeat that:
        self.hop_sequence = [self.hop_sequence[x % n_channels]
                             for x in range(n_bursts)]
        if verbose:
            print("Hop Frequencies  | Hop Pattern")

print("=================|================================")
            for f in self.hop_sequence:
                print("{:6.3f} MHz  |  ".format(f / 1e6), end='')
                if n_channels < 50:
                    print(" " * int((f - base_freq) / freq_delta) + "#")
                else:
                    print("\n")

print("=================|================================")
        # There's no real point in setting the gain via tag for this application,
        # but this is an example to show you how to do it.
```

```
gain_tag = gr.tag_t()
gain_tag.offset = 0
gain_tag.key = pmt.string_to_symbol('tx_command')
gain_tag.value = pmt.to_pmt({'gain': tx_gain})
tag_list = [gain_tag, ]
for i in range(len(self.hop_sequence)):
    time = pmt.cons(
        pmt.from_uint64(int(base_time + i * hop_time + 0.01)),
        pmt.from_double((base_time + i * hop_time + 0.01) % 1),
    )
    tune_tag = gr.tag_t()
    tune_tag.offset = i * burst_length
    # TODO dsp_tuning should also be able to do post_tuning
    if i > 0 and post_tuning and not dsp_tuning:
        tune_tag.offset -= 1    # Move it to last sample of previous burst
    if dsp_tuning:
        tune_tag.key = pmt.string_to_symbol('tx_command')
        tune_tag.value = pmt.to_pmt(
            {'lo_freq': base_freq, 'dsp_freq': base_freq - self.hop_sequence[i]})
        tune_tag.value = pmt.dict_add(
            tune_tag.value, pmt.intern("time"), time)
    else:
        tune_tag.key = pmt.string_to_symbol('tx_command')
        tune_tag.value = pmt.to_pmt({'freq': self.hop_sequence[i]})
        tune_tag.value = pmt.dict_add(
            tune_tag.value, pmt.intern('time'), time)
    tag_list.append(tune_tag)
    length_tag = gr.tag_t()
    length_tag.offset = i * burst_length
    length_tag.key = pmt.string_to_symbol('packet_len')
    length_tag.value = pmt.from_long(burst_length)
    tag_list.append(length_tag)
    time_tag = gr.tag_t()
    time_tag.offset = i * burst_length
    time_tag.key = pmt.string_to_symbol('tx_time')
    time_tag.value = pmt.make_tuple(
        pmt.car(time),
        pmt.cdr(time)
    )
    tag_list.append(time_tag)
tag_source = blocks.vector_source_c(
    (1.0,) * n_samples_total, repeat=False, tags=tag_list)
mult = blocks.multiply_cc()
self.connect(self, mult, self)
```

```
        self.connect(tag_source, (mult, 1))


class FlowGraph(gr.top_block):
    """ Flow graph that does the frequency hopping. """

    def __init__(self, args):
        gr.top_block.__init__(self)
        if args.input_file is not None:
            src = blocks.file_source(
                gr.sizeof_gr_complex, args.input_file, repeat=True)
        else:
            src = blocks.vector_source_c((.5,) * int(1e6) * 2, repeat=True)
        # Setup USRP
        self.usrp = uhd.usrp_sink(
            args.args,
            uhd.stream_args('fc32'),
            "packet_len"
        )
        if args.spec:
            self.usrp.set_subdev_spec(args.spec, 0)
        if args.antenna:
            self.usrp.set_antenna(args.antenna, 0)
        self.usrp.set_samp_rate(args.rate)
        # Gain is set in the hopper block
        if not args.gain:
            gain_range = self.usrp.get_gain_range()
            args.gain = float(gain_range.start() + gain_range.stop()) / 2.0
        print("-- Setting gain to {} dB".format(args.gain))
        if not self.usrp.set_center_freq(args.freq):
            print('[ERROR] Failed to set base frequency.')
            exit(1)
        hopper_block = FrequencyHopperSrc(
            args.num_bursts, args.num_channels,
            args.freq_delta, args.freq, args.dsp,
            args.samp_per_burst, 1.0, args.hop_time / 1000.,
            args.post_tuning,
            args.gain,
            args.verbose,
        )
        self.connect(src, hopper_block, self.usrp)


def print_hopper_stats(args):
    """ Nothing to do with Grace Hopper """
    print("""
Parameter              | Value
===================+=========================
Hop Interval           | {hop_time} ms
Burst duration         | {hop_duration} ms
Lowest Frequency       | {lowest_freq:6.3f} MHz
Highest Frequency      | {highest_freq:6.3f} MHz
Frequency spacing      | {freq_delta:6.4f} MHz
Number of channels | {num_channels}
Sampling rate          | {rate} Msps
Transmit Gain          | {gain} dB
===================+=========================
    """.format(
        hop_time=args.hop_time,
        hop_duration=1000.0 / args.rate * args.samp_per_burst,
        gain=args.gain if args.gain else "(midpoint)",
        lowest_freq=args.freq / 1e6,
        highest_freq=(args.freq + (args.num_channels - 1) *
                      args.freq_delta) / 1e6,
        freq_delta=args.freq_delta / 1e6,
        num_channels=args.num_channels,
        rate=args.rate / 1e6,
    ))


def main():
    """ Go, go, go! """
    args = setup_parser().parse_args()
    if (1.0 * args.samp_per_burst / args.rate) > args.hop_time * 1e-3:
        print("Burst duration must be smaller than hop time.")
        exit(1)
    if args.verbose:
        print_hopper_stats(args)
    top_block = FlowGraph(args)
    print("Starting to hop, skip and jump... press Ctrl+C to exit.")
    top_block.usrp.set_time_now(uhd.time_spec(0.0))
    top_block.run()


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

# ==================== END OF FILE: gr-uhd/examples/python/freq_hopping.py
====================


# ==================== START OF FILE: gr-
vocoder/examples/alaw_audio_loopback.py ====================
#!/usr/bin/env python
#
# Copyright 2011 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import audio
from gnuradio import blocks
from gnuradio import vocoder


def build_graph():
    tb = gr.top_block()
    src = audio.source(8000)
    src_scale = blocks.multiply_const_ff(32767)
    f2s = blocks.float_to_short()
    enc = vocoder.alaw_encode_sb()
    dec = vocoder.alaw_decode_bs()
    s2f = blocks.short_to_float()
    sink_scale = blocks.multiply_const_ff(1.0 / 32767.)
    sink = audio.sink(8000)
    tb.connect(src, src_scale, f2s, enc, dec, s2f, sink_scale, sink)
    return tb


if __name__ == '__main__':
    tb = build_graph()
    tb.start()
    input('Press Enter to exit: ')
    tb.stop()
    tb.wait()
```

```
# ==================== END OF FILE: gr-vocoder/examples/alaw_audio_loopback.py ====================


# ==================== START OF FILE: gr-vocoder/examples/codec2_audio_loopback.py ====================
#!/usr/bin/env python
#
# Copyright 2005,2007,2011 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import audio
from gnuradio import blocks
from gnuradio import vocoder
from gnuradio.vocoder import codec2


def build_graph():
    tb = gr.top_block()
    src = audio.source(8000)
    src_scale = blocks.multiply_const_ff(32767)
    f2s = blocks.float_to_short()
    enc = vocoder.codec2_encode_sp(codec2.MODE_2400)
    dec = vocoder.codec2_decode_ps(codec2.MODE_2400)
    s2f = blocks.short_to_float()
    sink_scale = blocks.multiply_const_ff(1.0 / 32767.)
    sink = audio.sink(8000)
    tb.connect(src, src_scale, f2s, enc, dec, s2f, sink_scale, sink)
    return tb


if __name__ == '__main__':
    tb = build_graph()
    tb.start()
    input('Press Enter to exit: ')
    tb.stop()
    tb.wait()
```

```
# ==================== END OF FILE: gr-vocoder/examples/codec2_audio_loopback.py ====================


# ==================== START OF FILE: gr-vocoder/examples/cvsd_audio_loopback.py ====================
#!/usr/bin/env python
#
# Copyright 2007,2011 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import audio
from gnuradio import blocks
from gnuradio import filter
from gnuradio import vocoder


def build_graph():
    sample_rate = 8000
    scale_factor = 32000

    tb = gr.top_block()
    src = audio.source(sample_rate, "plughw:0,0")
    src_scale = blocks.multiply_const_ff(scale_factor)

    interp = filter.rational_resampler_fff(8, 1)
    f2s = blocks.float_to_short()

    enc = vocoder.cvsd_encode_sb()
    dec = vocoder.cvsd_decode_bs()

    s2f = blocks.short_to_float()
    decim = filter.rational_resampler_fff(1, 8)

    sink_scale = blocks.multiply_const_ff(1.0 / scale_factor)
    sink = audio.sink(sample_rate, "plughw:0,0")
    tb.connect(src, src_scale, interp, f2s, enc)
    tb.connect(enc, dec, s2f, decim, sink_scale, sink)

    if 0:   # debug
        tb.connect(src, blocks.file_sink(gr.sizeof_float, "source.dat"))
        tb.connect(src_scale, blocks.file_sink(
            gr.sizeof_float, "src_scale.dat"))
        tb.connect(interp, blocks.file_sink(gr.sizeof_float, "interp.dat"))
        tb.connect(f2s, blocks.file_sink(gr.sizeof_short, "f2s.dat"))
        tb.connect(enc, blocks.file_sink(gr.sizeof_char, "enc.dat"))
        tb.connect(dec, blocks.file_sink(gr.sizeof_short, "dec.dat"))
        tb.connect(s2f, blocks.file_sink(gr.sizeof_float, "s2f.dat"))
        tb.connect(decim, blocks.file_sink(gr.sizeof_float, "decim.dat"))
        tb.connect(sink_scale, blocks.file_sink(
            gr.sizeof_float, "sink_scale.dat"))

    return tb


if __name__ == '__main__':
    tb = build_graph()
    tb.start()
    input('Press Enter to exit: ')
    tb.stop()
    tb.wait()

# ==================== END OF FILE: gr-vocoder/examples/cvsd_audio_loopback.py ====================


# ==================== START OF FILE: gr-vocoder/examples/g721_audio_loopback.py ====================
#!/usr/bin/env python
#
# Copyright 2011 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import audio
from gnuradio import blocks
from gnuradio import vocoder


def build_graph():
    tb = gr.top_block()
    src = audio.source(8000)
    src_scale = blocks.multiply_const_ff(32767)
    f2s = blocks.float_to_short()
    enc = vocoder.g721_encode_sb()
    dec = vocoder.g721_decode_bs()
    s2f = blocks.short_to_float()
    sink_scale = blocks.multiply_const_ff(1.0 / 32767.)
    sink = audio.sink(8000)
    tb.connect(src, src_scale, f2s, enc, dec, s2f, sink_scale, sink)
    return tb


if __name__ == '__main__':
    tb = build_graph()
    tb.start()
    input('Press Enter to exit: ')
    tb.stop()
    tb.wait()

# ==================== END OF FILE: gr-vocoder/examples/g721_audio_loopback.py ====================


# ==================== START OF FILE: gr-vocoder/examples/g723_24_audio_loopback.py ====================
#!/usr/bin/env python
#
# Copyright 2011 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import audio
from gnuradio import blocks
from gnuradio import vocoder


def build_graph():
    tb = gr.top_block()
    src = audio.source(8000)
    src_scale = blocks.multiply_const_ff(32767)
    f2s = blocks.float_to_short()
    enc = vocoder.g723_24_encode_sb()
    dec = vocoder.g723_24_decode_bs()
    s2f = blocks.short_to_float()
    sink_scale = blocks.multiply_const_ff(1.0 / 32767.)
    sink = audio.sink(8000)
    tb.connect(src, src_scale, f2s, enc, dec, s2f, sink_scale, sink)
    return tb


if __name__ == '__main__':
    tb = build_graph()
    tb.start()
    input('Press Enter to exit: ')
    tb.stop()
    tb.wait()

# ==================== END OF FILE: gr-vocoder/examples/g723_24_audio_loopback.py ====================


# ==================== START OF FILE: gr-vocoder/examples/g723_40_audio_loopback.py ====================
#!/usr/bin/env python
#
# Copyright 2011 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import audio
from gnuradio import blocks
from gnuradio import vocoder
```

```
def build_graph():
        tb = gr.top_block()
        src = audio.source(8000)
        src_scale = blocks.multiply_const_ff(32767)
        f2s = blocks.float_to_short()
        enc = vocoder.g723_40_encode_sb()
        dec = vocoder.g723_40_decode_bs()
        s2f = blocks.short_to_float()
        sink_scale = blocks.multiply_const_ff(1.0 / 32767.)
        sink = audio.sink(8000)
        tb.connect(src, src_scale, f2s, enc, dec, s2f, sink_scale, sink)
        return tb


if __name__ == '__main__':
        tb = build_graph()
        tb.start()
        input('Press Enter to exit: ')
        tb.stop()
        tb.wait()

# ==================== END OF FILE: gr-vocoder/examples/g723_40_audio_loopback.py ====================


# ==================== START OF FILE: gr-vocoder/examples/gsm_audio_loopback.py ====================
#!/usr/bin/env python
#
# Copyright 2005,2007,2011 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import audio
from gnuradio import blocks
from gnuradio import vocoder
```

```
def build_graph():
        tb = gr.top_block()
        src = audio.source(8000)
        src_scale = blocks.multiply_const_ff(32767)
        f2s = blocks.float_to_short()
        enc = vocoder.gsm_fr_encode_sp()
        dec = vocoder.gsm_fr_decode_ps()
        s2f = blocks.short_to_float()
        sink_scale = blocks.multiply_const_ff(1.0 / 32767.)
        sink = audio.sink(8000)
        tb.connect(src, src_scale, f2s, enc, dec, s2f, sink_scale, sink)
        return tb


if __name__ == '__main__':
        tb = build_graph()
        tb.start()
        input('Press Enter to exit: ')
        tb.stop()
        tb.wait()

# ==================== END OF FILE: gr-vocoder/examples/gsm_audio_loopback.py ====================


# ==================== START OF FILE: gr-vocoder/examples/ulaw_audio_loopback.py ====================
#!/usr/bin/env python
#
# Copyright 2011 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
#

from gnuradio import gr
from gnuradio import audio
from gnuradio import blocks
from gnuradio import vocoder
```

```
def build_graph():
        tb = gr.top_block()
        src = audio.source(8000)
        src_scale = blocks.multiply_const_ff(32767)
        f2s = blocks.float_to_short()
        enc = vocoder.ulaw_encode_sb()
        dec = vocoder.ulaw_decode_bs()
        s2f = blocks.short_to_float()
        sink_scale = blocks.multiply_const_ff(1.0 / 32767.)
        sink = audio.sink(8000)
        tb.connect(src, src_scale, f2s, enc, dec, s2f, sink_scale, sink)
        return tb


if __name__ == '__main__':
        tb = build_graph()
        tb.start()
        input('Press Enter to exit: ')
        tb.stop()
        tb.wait()

#           ====================           END           OF           FILE:           gr-
vocoder/examples/ulaw_audio_loopback.py ====================


#  ====================  START  OF  FILE:  gr-zeromq/examples/python/client.py
====================
#
# Copyright 2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio.
#
# SPDX-License-Identifier: GPL-3.0-or-later
#

######################################################################
#############
# Imports
######################################################################
#############
from gnuradio import zeromq
#import zeromq
from gnuradio import gr
from gnuradio import blocks
from gnuradio import analog
from gnuradio import eng_notation
from gnuradio.eng_arg import eng_float, intx
from argparse import ArgumentParser
import numpy
import sys
from threading import Thread
import time

##################################################################
#############
# GNU Radio top_block
##################################################################
#############


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

		self.options = options

		# socket addresses
		rpc_adr = "tcp://*:6667"
		probe_adr = "tcp://*:5557"
		source_adr = "tcp://" + self.options.servername + ":5555"

		# blocks
		#self.zmq_source = zeromq.req_source(gr.sizeof_float, 1, source_adr)
		#self.zmq_source = zeromq.pull_source(gr.sizeof_float, 1, source_adr)
		self.zmq_source = zeromq.sub_source(gr.sizeof_float, 1, source_adr)
		#self.zmq_probe = zeromq.push_sink(gr.sizeof_float, 1, probe_adr)
		self.zmq_probe = zeromq.pub_sink(gr.sizeof_float, 1, probe_adr)

		# connects
		self.connect(self.zmq_source, self.zmq_probe)

		# ZeroMQ
		self.rpc_manager = zeromq.rpc_manager()
		self.rpc_manager.set_reply_socket(rpc_adr)
		self.rpc_manager.add_interface("start_fg", self.start_fg)
		self.rpc_manager.add_interface("stop_fg", self.stop_fg)
		self.rpc_manager.start_watcher()
```

```
    def start_fg(self):
        print("Start Flowgraph")
        try:
            self.start()
        except RuntimeError:
            print("Can't start, flowgraph already running!")

    def stop_fg(self):
        print("Stop Flowgraph")
        self.stop()
        self.wait()

##################################################################
#############
# Options Parser
##################################################################
#############


def parse_args():
    """Argument parser."""
    parser = ArgumentParser()
    parser.add_argument("-s", "--servername", default="localhost",
                        help="Server hostname")
    args = parser.parse_args()
    return args

##################################################################
#############
# Waiter Thread
##################################################################
#############


class waiter(Thread):
    """ To keep the program alive when flowgraph is stopped. """

    def run(self):
        while keep_running:
            time.sleep(1)


##################################################################
#############
```

```
# Main
##############################################################################
if __name__ == "__main__":
	args = parse_args()
	tb = top_block(args)
	try:
		# keep the program running when flowgraph is stopped
		while True:
			time.sleep(1)
	except KeyboardInterrupt:
		pass
	print("Shutting down flowgraph.")
	tb.rpc_manager.stop_watcher()
	tb.stop()
	tb.wait()
	tb = None

# ==================== END OF FILE: gr-zeromq/examples/python/client.py ====================


# ==================== START OF FILE: gr-zeromq/examples/python/gui.py ====================
#
# Copyright 2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio.
#
# SPDX-License-Identifier: GPL-3.0-or-later
#

##############################################################################
# Imports
##############################################################################
from argparse import ArgumentParser
from gnuradio.eng_arg import eng_float, intx
import gui
import sys
import os
from PyQt5 import Qt, QtGui, QtCore, uic
import PyQt5.Qwt5 as Qwt
```

```
from gnuradio import zeromq
import signal


class gui(QtGui.QMainWindow):
    def __init__(self, window_name, options, parent=None):
        QtGui.QMainWindow.__init__(self, parent)

        # give Ctrl+C back to system
        signal.signal(signal.SIGINT, signal.SIG_DFL)

        self.gui = uic.loadUi(os.path.join(
            os.path.dirname(__file__), 'main_window.ui'), self)

        self.update_timer = Qt.QTimer()

        # socket addresses
        rpc_adr_server = "tcp://" + options.servername + ":6666"
        rpc_adr_client = "tcp://" + options.clientname + ":6667"
        probe_adr_server = "tcp://" + options.servername + ":5556"
        probe_adr_client = "tcp://" + options.clientname + ":5557"

        # ZeroMQ
        self.probe_manager = zeromq.probe_manager()
        self.probe_manager.add_socket(
            probe_adr_server, 'float32', self.plot_data_server)
        self.probe_manager.add_socket(
            probe_adr_client, 'float32', self.plot_data_client)

        self.rpc_mgr_server = zeromq.rpc_manager()
        self.rpc_mgr_server.set_request_socket(rpc_adr_server)
        self.rpc_mgr_client = zeromq.rpc_manager()
        self.rpc_mgr_client.set_request_socket(rpc_adr_client)

        self.gui.setWindowTitle(window_name)
        self.gui.qwtPlotServer.setTitle("Signal Scope")
        self.gui.qwtPlotServer.setAxisTitle(Qwt.QwtPlot.xBottom, "Samples")
        self.gui.qwtPlotServer.setAxisTitle(Qwt.QwtPlot.yLeft, "Amplitude")
        self.gui.qwtPlotServer.setAxisScale(Qwt.QwtPlot.xBottom, 0, 100)
        self.gui.qwtPlotServer.setAxisScale(Qwt.QwtPlot.yLeft, -2, 2)
        self.gui.qwtPlotClient.setTitle("Signal Scope")
        self.gui.qwtPlotClient.setAxisTitle(Qwt.QwtPlot.xBottom, "Samples")
        self.gui.qwtPlotClient.setAxisTitle(Qwt.QwtPlot.yLeft, "Amplitude")
        self.gui.qwtPlotClient.setAxisScale(Qwt.QwtPlot.xBottom, 0, 100)
        self.gui.qwtPlotClient.setAxisScale(Qwt.QwtPlot.yLeft, -2, 2)

        # Grid
        pen = Qt.QPen(Qt.Qt.DotLine)
        pen.setColor(Qt.Qt.black)
        pen.setWidth(0)
        grid_server = Qwt.QwtPlotGrid()
        grid_client = Qwt.QwtPlotGrid()
        grid_server.setPen(pen)
        grid_client.setPen(pen)
        grid_server.attach(self.gui.qwtPlotServer)
        grid_client.attach(self.gui.qwtPlotClient)

        # Signals
        self.connect(self.update_timer, QtCore.SIGNAL(
            "timeout()"), self.probe_manager.watcher)
        self.connect(self.gui.pushButtonRunServer, QtCore.SIGNAL(
            "clicked()"), self.start_fg_server)
        self.connect(self.gui.pushButtonStopServer,
                             QtCore.SIGNAL("clicked()"), self.stop_fg_server)
        self.connect(self.gui.pushButtonRunClient, QtCore.SIGNAL(
            "clicked()"), self.start_fg_client)
        self.connect(self.gui.pushButtonStopClient,
                             QtCore.SIGNAL("clicked()"), self.stop_fg_client)
        self.connect(self.gui.comboBox, QtCore.SIGNAL(
            "currentIndexChanged(QString)"), self.set_waveform)
        self.connect(self.gui.spinBox, QtCore.SIGNAL(
            "valueChanged(int)"), self.set_gain)
        self.shortcut_start = QtGui.QShortcut(
            Qt.QKeySequence("Ctrl+S"), self.gui)
        self.shortcut_stop = QtGui.QShortcut(
            Qt.QKeySequence("Ctrl+C"), self.gui)
        self.shortcut_exit = QtGui.QShortcut(
            Qt.QKeySequence("Ctrl+D"), self.gui)
        self.connect(self.shortcut_exit, QtCore.SIGNAL(
            "activated()"), self.gui.close)

        # start update timer
        self.update_timer.start(30)

    def start_fg_server(self):
        self.rpc_mgr_server.request("start_fg")

    def stop_fg_server(self):
        self.rpc_mgr_server.request("stop_fg")

    def start_fg_client(self):
        self.rpc_mgr_client.request("start_fg")

    def stop_fg_client(self):
        self.rpc_mgr_client.request("stop_fg")

    # plot the data from the queues
    def plot_data(self, plot, samples):
        self.x = list(range(0, len(samples), 1))
        self.y = samples
        # clear the previous points from the plot
        plot.clear()
        # draw curve with new points and plot
        curve = Qwt.QwtPlotCurve()
        curve.setPen(Qt.QPen(Qt.Qt.blue, 2))
        curve.attach(plot)
        curve.setData(self.x, self.y)
        plot.replot()

    def plot_data_server(self, samples):
        self.plot_data(self.gui.qwtPlotServer, samples)

    def plot_data_client(self, samples):
        self.plot_data(self.gui.qwtPlotClient, samples)

    def set_waveform(self, waveform_str):
        self.rpc_mgr_server.request("set_waveform", [str(waveform_str)])

    def set_gain(self, gain):
        self.rpc_set_gain(gain)

    def rpc_set_gain(self, gain):
        self.rpc_mgr_server.request("set_k", [gain])

##################################################################
#############
# Options Parser
##################################################################
#############


def parse_args():
    """Options parser."""
    parser = ArgumentParser()
    parser.add_argument("-s", "--servername", default="localhost",
                        help="Server hostname")
    parser.add_argument("-c", "--clientname", default="localhost",
                        help="Server hostname")
    args = parser.parse_args()
    return args


##############################################################################
# Main
##############################################################################
if __name__ == "__main__":
    args = parse_args()
    qapp = Qt.QApplication(sys.argv)
    qapp.main_window = gui("Remote GNU Radio GUI", args)
    qapp.main_window.show()
    qapp.exec_()

# ==================== END OF FILE: gr-zeromq/examples/python/gui.py ====================


# ==================== START OF FILE: gr-zeromq/examples/python/server.py ====================
#
# Copyright 2013 Free Software Foundation, Inc.
#
# This file is part of GNU Radio.
#
# SPDX-License-Identifier: GPL-3.0-or-later
#

##############################################################################
# Imports
##############################################################################
from gnuradio import zeromq
from gnuradio import gr
from gnuradio import blocks
from gnuradio import analog
from gnuradio import eng_notation
from gnuradio.eng_arg import eng_float, intx
from argparse import ArgumentParser
import numpy
import sys
from threading import Thread
import time


######################################################################
#############
# GNU Radio top_block
######################################################################
#############
class top_block(gr.top_block):
    def __init__(self, options):
        gr.top_block.__init__(self)

        self.options = options

        # socket addresses
        rpc_adr = "tcp://*:6666"
        probe_adr = "tcp://*:5556"
        sink_adr = "tcp://*:5555"

        # the strange sampling rate gives a nice movement in the plot :P
        self.samp_rate = samp_rate = 48200

        # blocks
        self.gr_sig_source = analog.sig_source_f(
            samp_rate, analog.GR_SIN_WAVE, 1000, 1, 0)
        self.throttle = blocks.throttle(gr.sizeof_float, samp_rate)
        self.mult = blocks.multiply_const_ff(1)
        #self.zmq_sink = zeromq.rep_sink(gr.sizeof_float, 1, sink_adr)
        self.zmq_sink = zeromq.pub_sink(gr.sizeof_float, 1, sink_adr)
        #self.zmq_sink = zeromq.push_sink(gr.sizeof_float, 1, sink_adr)
        #self.zmq_probe = zeromq.push_sink(gr.sizeof_float, 1, probe_adr)
        self.zmq_probe = zeromq.pub_sink(gr.sizeof_float, 1, probe_adr)
        #self.null_sink = blocks.null_sink(gr.sizeof_float)

        # connects
        self.connect(self.gr_sig_source, self.mult,
                     self.throttle, self.zmq_sink)
        self.connect(self.throttle, self.zmq_probe)

        # ZeroMQ
        self.rpc_manager = zeromq.rpc_manager()
        self.rpc_manager.set_reply_socket(rpc_adr)
        self.rpc_manager.add_interface("start_fg", self.start_fg)
        self.rpc_manager.add_interface("stop_fg", self.stop_fg)
        self.rpc_manager.add_interface("set_waveform", self.set_waveform)
        self.rpc_manager.add_interface("set_k", self.mult.set_k)
        self.rpc_manager.add_interface(
            "get_sample_rate", self.throttle.sample_rate)
        self.rpc_manager.start_watcher()

    def start_fg(self):
        print("Start Flowgraph")
        try:
            self.start()
        except RuntimeError:
            print("Can't start, flowgraph already running!")

    def stop_fg(self):
        print("Stop Flowgraph")
        self.stop()
        self.wait()

    def set_waveform(self, waveform_str):
        waveform = {'Constant': analog.GR_CONST_WAVE,
                    'Sine': analog.GR_SIN_WAVE,
                    'Cosine': analog.GR_COS_WAVE,
                    'Square': analog.GR_SQR_WAVE,
                    'Triangle': analog.GR_TRI_WAVE,
                    'Saw Tooth': analog.GR_SAW_WAVE}[waveform_str]
        self.gr_sig_source.set_waveform(waveform)

############################################################################
#############
# Options Parser
############################################################################
#############


def parse_args():
    """Argument parser."""
    parser = ArgumentParser()
    args = parser.parse_args()
    return args


########################################################################
##############
# Main
########################################################################
##############
if __name__ == "__main__":
    args = parse_args()
    tb = top_block(args)
    try:
        # keep the program running when flowgraph is stopped
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        pass
    print("Shutting down flowgraph.")
    tb.rpc_manager.stop_watcher()
    tb.stop()
    tb.wait()
    tb = None

# ==================== END OF FILE: gr-zeromq/examples/python/server.py
====================
```