Reducing Audio Bandwidth with an FFT-based Low-Pass Filter

Trent Geisler
Analytics and Data Science
Kennesaw State University
Kennesaw, Georgia 30144

Andrew Henshaw
Analytics and Data Science
Kennesaw State University
Kennesaw, Georgia 30144

Lauren Staples
Analytics and Data Science
Kennesaw State University
Kennesaw, Georgia 30144

***Abstract*—This project demonstrates the application of the Fast Fourier Transform (FFT) as a method for low-pass filtering of audio signals. We discuss the need for low-pass filtering to prevent undersampling and to reduce bandwidth requirements. We explain filter-design techniques and introduce GNU Radio as a platform for audio and radio-frequency exploration. We explain and demonstrate how an appropriately-chosen low-pass filter can reduce bandwidth without noticeably degrading audio signal quality.**

I. Introduction

When an appropriate filter is designed and used to remove unwanted frequency components, the bandwidth and power necessary for audio signal transmission may be reduced. In radio operations, the frequency bands are scarce resources and are governed by national and international agencies. Every transmitter that can interfere with others has to operate in a licensed band and is subject to bandwidth limitations, which is why multiple radio stations can operate in the same geographical area. If a radio station transmits beyond its assigned bandwidth, it will impact the signal transmission of other local radio stations. To keep transmitters from interfering with other transmitters in different geographic areas, power limits are imposed. By reducing the bandwidth of their signals, a broadcaster can maximize the impact of its power allocation. Radio broadcasters control and limit the bandwidth of their emissions by the use of appropriately designed filters [2].

Both bandwidth and power savings are the motivation behind Fast Fourier Transform (FFT)-based low-pass filters. This paper will demonstrate that a Finite Impulse Response (FIR), low-pass filter implemented with an FFT can remove unwanted frequency components in order to reduce the bandwidth and power required for transmission. We will also demonstrate that this bandwidth reduction can be applied to audio signals without noticeably degrading the perceived quality of the sound.

A. Objective

Our objective is to design and demonstrate an FFT-based FIR low-pass filter to reduce bandwidth of a recorded audio signal for the purposes of AM Radio transmission.

II. Methods

We collected data, designed a low pass filter, pushed the signal through the filter, and then evaluated the resulting signal.

A. Data

We collected an audio signal by recording the song 'Flight of the Bumblebee' at the full frequency spectrum that a compact disc (CD) is recorded, which is 44.1 kHz. The human ear does not even hear this full spectrum. This means that there are potential bandwidth 'savings' for transmission! This collected audio recording is referred to as the 'raw signal.'

B. Filter Design

The typical Human can only hear frequencies in the range of 20Hz through 20kHz, and this range only decreases as humans age [3]. Using the website http://onlinetonegenerator.com/hearingtest.html, we will test the hearing range of each member listening to our final presentation. This demonstration will provide a tangible example why appropriate filtering of an audio signal will have little to no impact on the quality of sound that a person hears after the filtering.

When our project group performed the hearing test, the highest audible frequency was 15kHz. Since our project group was not able to hear the frequencies above the 15kHz threshold, any low-pass filter that removed the high frequencies above 15kHz would have no discernable impact on the quality of the audio signal that we would hear. Recall, removing the unecessary frequencies from an audio signal transmission provides a few nice benefits such as the following:

1) It reduces the necessary power for transmission of the audio signal,
2) It reduces the necessary bandwidth for transmission.

Filtering is the processing of a time-domain signal resulting in some change in that signal's original spectral content. The change is usually the reduction or filtering of unwanted input spectral components [4]. Given the human threshold for hearing, an obvious opportunity is to apply low-pass filtering to only transmit audio signals that humans can actually hear with smaller bandwidths. The ideal low-pass filter would completely eliminate all frequencies above a certain cutoff point (in our hearing test example, the cutoff would be at 15kHz) while passing all frequencies below the cutoff point [5].

To filter out higher frequencies in an audio signal transmission, we will build a low-pass filter. Specifically, we will build a low-pass filter using a finite number of non-zero filter

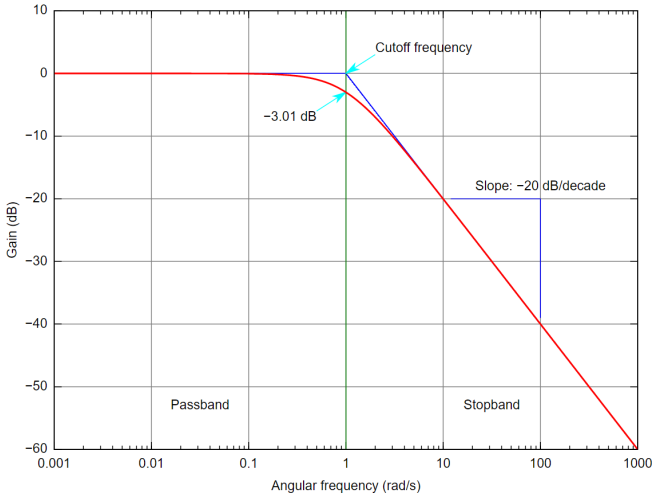

Fig. 1. Example of Low-Pass Filter: https://upload.wikimedia.org/wikipedia/commons/6/60/Butterworth_response.svg

coefficients which is called a *Finite Impulse Response* filter or FIR. Given an impulse response, we can find the coefficients of the filter, and vice versa [2]. For example, Figure 1 shows a graphical depiction of a low-pass filter that begins to cut out the frequencies above 1 rad/s. Frequencies below the cutoff frequency are considered to be in the "passband," or allowable frequency band. Frequencies above the cutoff frequency filtered out and are considered in the "stopband." Ideally, the slope between the passband and the stopband would be as steep as possible to limit the passing of unwanted high audio frequencies. The question still remains, how do you build a filter that eliminates these frequencies? What is an impulse response? How do we translate an impulse response to a frequency response and vice versa. We address these questions next.

1) Impulse Input, Impulse Response, and FFT: In our filtering example, we will be dealing with a linear time-invariant (LTI) system. A linear system is a class of systems where the system's outputs are the sum of the outputs of its parts. Additionally, time-invariant refers to a system where a time delay in the input sequence causes an equivalent delay in the output sequence. Now, if we are given a LTI system, we can calculate everything about the system if we know the *unit impulse response*. The *unit impulse response* refers to the system's time-domain output sequence when the input is a single unity-valued sample (unit impulse) surounded by zero-valued samples. Furthermore, knowing the impulse response of an LTI system, the *output sequence* is calculated by taking the *convolution* of the input sequence and the system's impulse response [4]. We will talk more about convolutions later, but we typically do not perform convolutions in the time domain since they are computationally expensive. Instead, multiplication is performed in the frequency domain. In order to transform the impuse response into a *frequency response*, we take the Fast Fourier Transform (FFT) of the impulse response [4].

Let's explain with a simple example. If we let $x(n)$ be a discrete-time sequence of individual signal amplitudes, then we can define a simple LTI *averager* system that takes the average of the last four inputs as follows:

$$y(n) = \frac{1}{4}\left[x(n) + x(n-1) + x(n-2) + x(n-3)\right] = \frac{1}{4}\sum_{k=n-3}^{n} x(k)$$

Given this simple averager, we can show the block diagram in Figure 3 (a), impulse input and impulse response in Figure 3 (b), and the frequency magnitude response created from the FFT of the impulse response in Figure 3 (c). The block diagram in Figure 3 (a) - also referred to as the *filter structure*, simply shows how an impulse input is transformed into a impulse response. The impulse response, $y(n)$, is created by passing the impulse input, $x(n)$, into the system and storing the four most recent values. Once there are four values, they are added together and multiplied by $\frac{1}{4}$ to calculate the average. The sequence proceeds one step, and then performs another average of the four most recent $x(n)$ values. The averaging continues as long as impulse input enters the filter. Since there are four separate input sample values to calculate an output value, the structure of this filter can be referred to as a *4-tap tapped-delay line FIR filter* using digital filter vernacular. The coefficients of this filter are all $\frac{1}{4}$ and we can use an FFT on the filter to provide all the frequency domain information.

For example, we can create a simple input impulse in Python by generating random values between -0.5 and 0.5 and then using the numpy package in Python to apply the FFT in order to get the frequency response, labeled X_m in Listing 1, of the impulse input. We can then take get the frequency response of the filter coefficients (labeled h_n in Listing 1) by performing another FFT. The frequency response of the filter coefficients is labeled as H_m in Listing 1. Finally, we can generate the new output spectrum, Y_m, by multiplying X_m and H_m since they are both in the frequency domain. See the entire code snipit below in Listing 1. All of the code for this example, to include the code for the plots, is listed in Listing 2 in the Appendix.

Listing 1. Simple Averager Filter

```
#generate impulse input
num_samples = 100
x_n = np.random.rand(num_samples)

#generate the frequency response for impulse input
X_m = np.fft.fft(x_n, N_fft)

#generate filter coefficients for simple averager
h_n = np.ones(4)/4

#generate the frequency response of the filter
# the second argument of fft is the number of fft bins

N_fft = 1024 #use a power of 2
H_m = np.fft.fft(h_n, N_fft)

#generate the output after filter
Y_m = H_m * abs(X_m)
```

Figure 2 shows the impact of the simple averager filter on the original X_m (input impulse after transformation into the frequency domain). The original X_m is ploted in blue in the bottom graph in Figure 2 and has a lot of frequency information across the entire domain. Once the simple average filter is applied, the output spectrum, Y_m in red, has greatly reduced frequency content as the frequency gets further away from 0. Additionally, you should notice that the output spectrum (red line) is created by simply multiplying the frequency response of the filter coefficients (top blue graph) by the input spectrum (bottom blue graph).

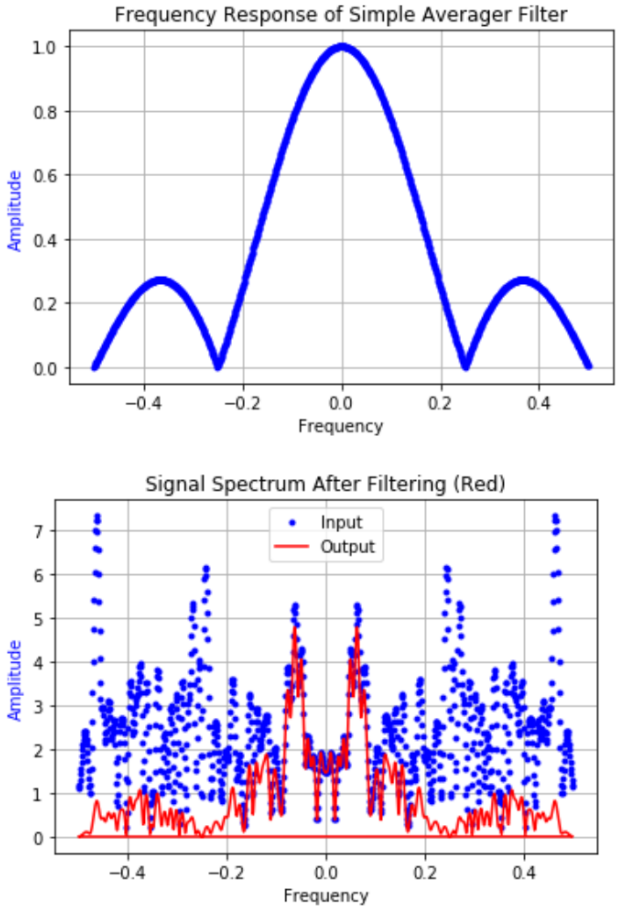

Fig. 2. Frequency Response from the Filter Coefficients (TOP) and the Spectrum for Original Spectrum (blue) and Output Spectrum (red) after filter is applied (BOTTOM)

The overview of this entire process is shown again in Figure 3. Looking at Figure 3 (c), the FFT transforms the impulse response $y(n)$ into the frequency information of $Y(m)$. $Y(m)$ provides the frequency magnitude response after the filter has been applied to the impulse input (also shown as the red line in the bottom graph of Figure 2). This is actually an example of a low-pass filter where the *averager* reduces the amplitude (attenuates) of the high-frequency signal. In our construction of a filter, we will have to use a different impulse response since we want to remove, not attenuate, the high frequency information content. The following section will explain how we design an FIR filter using the FFT and inverse FFT.

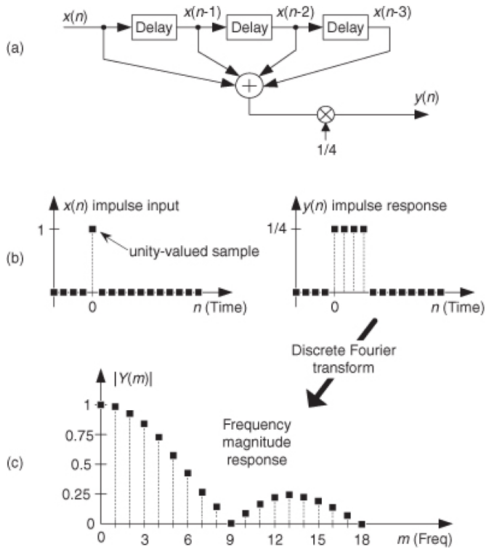

Fig. 3. Block Diagram (a), Impulse Input and Impulse Response (b), and Frequency Magnitude Response after FFT of Impulse Response (c). Taken from Reference [4] below and is Figure 1-12 from Chapter 1

2) Finite Impulse Response (FIR): Now that we have laid down the groundwork with the general overview and a simple example, we will discuss how to create an FIR filter. An FIR filter has a finite duation of nonzero output values given a finite duration of input values (this is how they were named!). Recall, in our example above, we used four "taps" that were all equal to $\frac{1}{4}$. The two factors that affect an FIR filter's frequency response are the number of taps and the coefficients [4]. If we were to calculate the impulse response, $y(n)$, from the impulse input, $x(n)$, in the time-domain, then we would have to perform the mathematical operation of a convolution. More specifically for an *M*-tap filter, we would calculate $y(n)$ as follows:

$$y(n) = \sum_{k=0}^{M-1} h(k)x(n-k)$$

Where $h(k)$ are the filter coefficients for each of the taps. **The terms *FIR filter coefficients* and *impulse response* mean the same thing** [4]. We can re-write the above equation using *convolution* notation as follows:

$$y(n) = h(k) \star x(n)$$

The major concept is that convolution in the time domain is equal to multiplication in the frequency domain. The process in which we can move from one domain to the other is by using an FFT. The FFT of $y(n)$ is equal to $Y(m) = H(m)X(m)$, which is the spectrum of the filter output. In our simple averager example above, we used multiplication to create the red line, $Y(m)$, in the bottom graph of Figure 2. In a similar way, we can determine

$y(n) = h(k) \star x(n)$ by taking the inverse FFT (denoted as I-FFT) of $Y(m)$ [4] The FFT and the inverse FFT allow us to move from the time domain to the frequency domain, and from the frequency domain back to the the time domain. To keep track of all the transformations, we listed the important relationships as follows:

1) $x(n) \overset{FFT}{\longrightarrow} X(m)$
2) $X(m) \overset{I-FFT}{\longrightarrow} x(n)$
3) $h(k) \overset{FFT}{\longrightarrow} H(m)$
4) $H(m) \overset{I-FFT}{\longrightarrow} h(k)$
5) $y(n) \overset{FFT}{\longrightarrow} Y(m) \Rightarrow h(k) \star x(n) \overset{FFT}{\longrightarrow} H(m)X(m)$
6) $Y(m) \overset{I-FFT}{\longrightarrow} y(n) \Rightarrow H(m)X(m) \overset{I-FFT}{\longrightarrow} h(k) \star x(n)$

Using these important relationships, a typical low-pass filter is designed by determining the desired frequency response of the filter, then using the relationship identified in (4) above to transform the filter information into the time domain so that it can be used in other applications. Since the process of calculating the desired frequency response is complicated, we will use a tool to provide the desired frequency response based on a set of input values. The following section will explain how we design an FIR filter using the GNU Radio Filter Design tool with the window method in order to create a low-pass filter that removes high frequency information content.

3) GNU Radio: Filter Design Tool: Now that we understand how to move between the frequency and time-domains using the FFT or inverse FFT, let's design our own low-pass filter by determining the desired frequency response and moving back to the time domain through an inverse FFT to calculate the filter coefficients that will provide the desired frequency response. Recall, in our hearing test example, our project group was unable to hear any frequency information content above 15 kHz. As such, we will attempt to design a low-pass filter that removes all of the frequency content above this threshold. Therefore, our *cutoff* frequency is 15kHz (see Figure 1 for a depiction of the cutoff frequency). To design the filter, we define the **frequency cut-off values** (end of passband and beginning of stopband), the **sample rate**, and the **stopband attenuation** (in decibels), and then use a Filter Design Tool within the GNU Radio application (which is written in Python) to calculate the frequency response and then apply the inverse FFT function to calculate the FIR filter coefficients.

We set the sample rate $=$ 44.1 kHz, the end of the passband at 15 kHz, the beginning of the stopband at 16 kHz, and the stopband attenuation at 60 dB. After hitting the "Design" button, the tool creates the frequency response for the desired filter which is displayed in Figure 4. As expected, the frequency response shows a drop-off at 15 kHz where the passband ends and the frequencies are cut. The tool will take the inverse FFT of the frequency response in order to generate the filter coefficients, or taps. Figure 5 shows the filter coefficients that provide the frequency response that removes frequency above 15 kHz. The filter coefficients are saved and then passed into the GNU Radio Tool that implements the filter created.

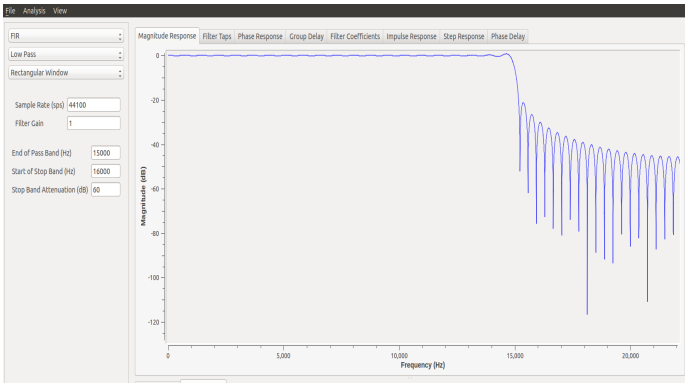

Fig. 4. Designing the 15 kHz Filter using the Filter Design Tool within GNU Radio

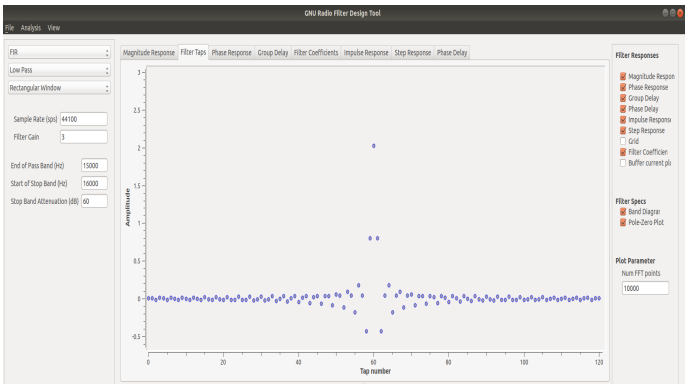

Fig. 5. Filter Coefficients for the 15 kHz Filter using the Filter Design Tool within GNU Radio

C. Nyquist-Shannon Sampling Theorem

When recording audio signals, one has to set a sampling rate (rate for analog signal, frequency for discrete signal). There are several reasons why we set a sampling rate:

1) Data storage conservation,
2) Bandwidth conservation,
3) Power conservation.

The formula for sampling is:

$$x[n] = x(t)|_{t=nT_S} = x(nT_S) \tag{1}$$

Where x is the signal or impulse input, t is the time, T_S is the sampling period and $f_S = \dfrac{1}{T_S}$ is the sampling frequency [2], and n is an index of the samples.

However, we have to be careful not to set this sampling rate too low, or else we run into a problem called aliasing. Low-frequency aliasing is caused by undersampling, and occurs when a different time function with a lower frequency

produces the same set of samples [7]. For example, the signal produced by the function:

$$x[t] = cos(2\pi 10t) \tag{2}$$

can be visualized in the time domain as:

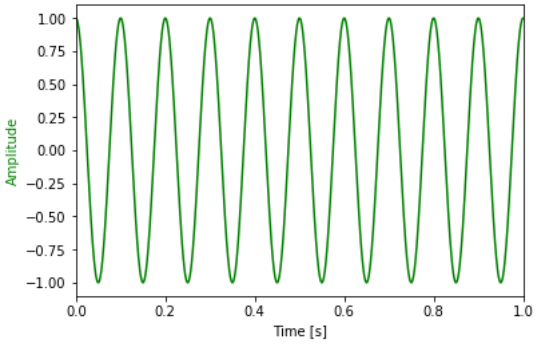

Fig. 6. Original signal, from equation 1 [8].

Converting to a discrete signal, by undersampling at 18 Hz rate, we produce the aliasing, as can be seen below.

$$x[n] = \cos(\frac{2\pi \cdot 10n}{18}) \tag{3}$$

Because the signal is periodic, adding or subtracting $2\pi n$ inside of the cos function does not change the equation.

$$x[n] = \cos(\frac{10\pi \cdot n}{9} - 2\pi n) \tag{4}$$

$$x[n] = \cos(\frac{10\pi \cdot n}{9} - \frac{18\pi n}{9}) \tag{5}$$

$$x[n] = \cos(-\frac{8\pi \cdot n}{9}) \tag{6}$$

$$x[n] = \cos(\frac{8\pi \cdot n}{9}) \tag{7}$$

By undersampling, we create an alias signal corresponding to $8\pi\frac{n}{9}$. Our sample rate of 18 Hz is too low. Below is our sampling overlaid on the original curve.

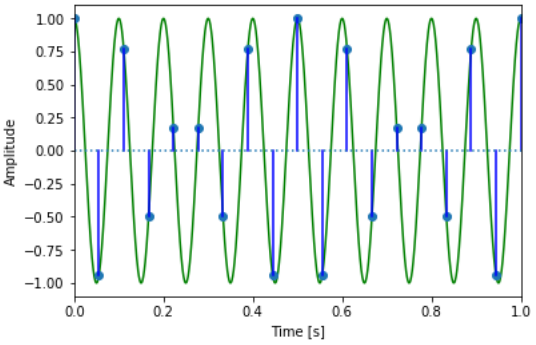

Fig. 7. Overlaid sampling (in blue) on the original curve (in green) 1 [8].

This shows that that there are additional functions that can cross through the sampling points of our original signal, creating alternative interpretations. For example, in the graph below, notice that the red curve matching the aliased result is a valid interpretation of the sampled points originally derived from the green curve.

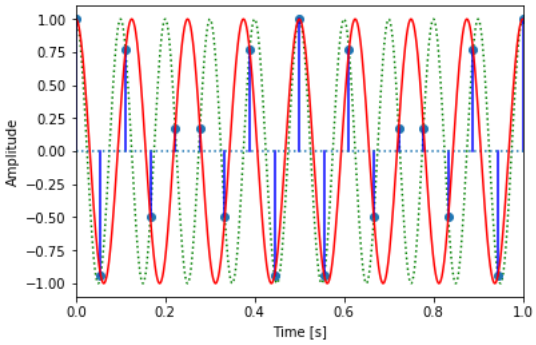

Fig. 8. The aliased signal (in red) intersects the original signal (in green) at the sampling points (in blue) 1 [8].

So, if our objective is to sample as little as possible, how do we determine the minimum rate necessary? The Nyquist-Shannon theorem states that if a function $x(t)$ contains no frequencies higher than B hertz, then it is completely determined by giving its ordinates at a series of points spaced $1/(2B)$ seconds apart [10]. It then follows that a sufficient sampling rate (nyquist rate) is therefore twice the maximum frequency B Hz, $2B$ [10]. Restated from a different angle: for a given sample rate f_s, perfect reconstruction of a signal is guaranteed for a bandlimit

$$B < f_s/2 \tag{8}$$

[10].

In our case, before processing with a filter, our raw signal had a sampling rate of 44.1 kHz. This is the standard for CD-quality audio. Since human hearing range is roughly 20 Hz to 20,000 Hz, the sampling rate has to at least be 40 kHz (from the Nyquist-Shannon theorem). Audio signals still have to be passed through a low-pass anti-aliasing filter, because even though humans cannot hear frequencies beyond that, these signals can still cause aliasing. A transition band of 4,100 Hz is added to the 40 kHz allows greater ease of anti-aliasing filtering [11]. Figure 1 shows this transition band. The larger this band, the less 'sharp' of a filter is required. This results in the standard frequency of 44.1 kHz for CDs.

We know from Equation 8 that the maximum frequency that can be represented at any given sampling rate is half the sampling rate; thus a 44.1 kHz CD can capture tones up to 22.05 kHz. This is often over-kill, since as previously mentioned, humans often cannot hear beyond 15 kHz. We have space to trim that down, but we must make sure to use filters to prevent aliasing.

III. Evaluation of Filtered Signal

To demonstrate the effects of low-pass filtering on a digitally-sampled audio file, we have built an application using the GNU Radio development framework. In addition to filtering and playing audio, the application demonstrates the importance of bandwidth reduction by transmission of the processed audio using a pair of software-defined radios operating at 903 MHz.

The GNU Radio project [1] is an open-source toolkit supporting the development of software-defined radios. Typically, application developed by building a flowgraph consisting of signal-processing components or "blocks" and their interconnects.

Once the flowgraph is designed, the layout is automatically converted into a Python program that manages the GNU Radio scheduler. The scheduler efficiently manages the routing of the work products to-and-from each block and is not limited by the speed of the Python interpreter. The signal-processing components are usually coded in C++ (for performance reasons).

A. Demo Flowgraph Overview

Our flowgraph is shown in figure 9 and the automatically-generated Python code is provided in the Appendix (Listing 4). Note that the interconnected blocks have multi-colored input and output tabs. The blue tabs show that the block produces or consumes a complex number, while the orange tabs indicate the use of simple floating-point values. Input tabs may only connect to one output tab, but output tabs may connect to one-or-more input tabs. Every tab must be connected.

Certain components have no connections. Primarily these blocks represent parameter blocks and are used to simplify the layout of the flowgraph. The blocks with the labels starting with "QT" represent graphical-user interface (GUI) elements that will be shown when the application is run. Our application uses the following GUI elements:

- Source selection (radio buttons)
- Filter selection (radio buttons)
- Volume (slider)
- Amplitude vs Time (oscilloscope widget)
- Amplitude vs Frequency (spectrograph widget)

B. Sources

Three sources are defined for the demo. The first source is "silence", which is generated by a constant stream of zeros.

The second source is simply a two-tone signal using 440 Hz (A above middle C), which is the reference frequency for the standard musical scale, and 3520 Hz, which is three octaves above the first tone. When selected in the demo, these two tones provide a very clear indication of the effects of the 1-kHz filter.

The third source is a digitally-sampled, CD-quality snippet from a recording of a performance of "Flight of the Bumblebee". With an effective bandwidth of 22.05 kHz, this source is affected by all of the filters. The block is set to "loop", so that it will run continuously.

A *QT GUI Chooser* drives the source selection block, allowing the chosen signal to pass to the rest of the flowgraph while blocking the other two sources.

C. Filter

Five filter options are defined:

- No filter
- 15-kHz filter
- 8-kHz filter
- 4-kHz filter
- 1-kHz filter

The filters are implemented using the *FFT Filter* component, which implements a low-pass filter by taking the Fourier Transform of the incoming signal, multiplying that by the transfer function representing the filter, and then performing an inverse-FFT to generate the output waveform. The filter parameters are given to the block as a sequence of filter taps, as defined above.

As in the source selection, the filter selection is managed by a QT GUI chooser block driving a selector block. The output of the selector block continues to the rest of the flowgraph.

D. Instrumentation

In order to visualize the operation of the filters, two GUI elements are used: an oscilloscope-like widget that charts amplitude versus time, and a spectrograph-like widget that shows amplitude versus frequency. When the different filters are chosen, the spectrograph clearly shows the change in bandwidth of the signal.

E. Output

Local audio output is provided by an *Audio Sink* block that drives the default audio device of the computer on which the program is run. The volume slider controls a multiplier block that directly affects the amplitude seen by both the and *Audio Sink* and the radio.

An amplitude-modulation (AM) radio is created in order to demonstrate a primary benefit of lowpass filtering: radio transmission bandwidth reduction. The operation of the AM-radio portion of the flowgraph is beyond the scope of this paper, but the output drives a software-defined-radio peripheral that transmits the audio at a center frequency of 903.5 MHz, which is 500 kHz above the base frequency of the radio.

F. Results

The GNU Radio toolkit provided a very effective platform for demonstrating the effects of lowpass filtering of different audio sources. Though not rigorously tested, we observed the behavior that was expected: the 15-kHz and 8-kHz filters reduced the bandwidth of the signal, but it was not readily discernible to our ears. It wasn't until we selected the 4-kHz filter that a difference was clearly noted.

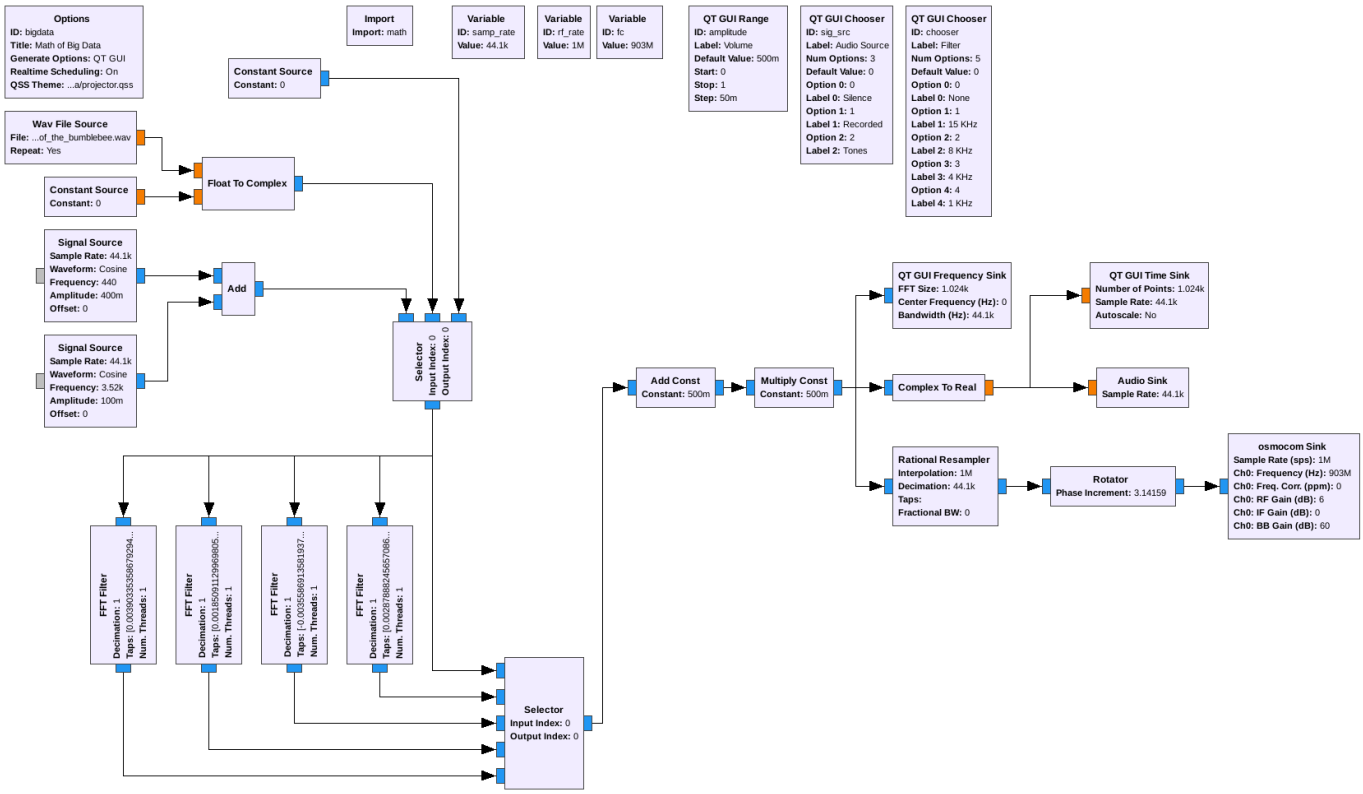

Fig. 9. Demonstration Application Flowgraph

Fig. 10. Demonstration Application: 15 kHz Filter

Fig. 11. Demonstration Application: 1 kHz Filter, Tones

IV. Conclusion

In radio operations, frequency bands are scarce resources that are governed by national and international agencies. Every transmitter must operate within its licensed band. Both bandwidth and power limitations are the motivation for more efficient use of bandwidth. A common way of achieving more efficient use of bandwidth is to filter frequencies occurring above the hearing range of humans (20 kHz). While humans cannot directly hear frequencies higher than this, these frequencies can still create tones through harmonics that are noticeable to humans. However, often times these tones are subtle and do not necessarily warrant the expense of the bandwidth they require. Our project designed several low-pass filters to find the point that optimizes the tradeoff in sound quality and bandwidth on a song called ”Flight of the Bumble Bees.” Sound quality deterioration was not discernable until the 4 kHz filter and below.

References

[1] *GNU Radio*, https://www.gnuradio.org/.

[2] Baxley, R., Henshaw, A., Nowlan, S., & Trewhitt, E. *Software-Defined Radio with GNU Radio: Theory and Application, Georgia Tech Professional Education course notes.* (2017)
[3] Rossing, Thomas (2007). Springer Handbook of Acoustics. Springer. pp. 747, 748. ISBN 978-0387304465.
[4] Lyons, Richard G. Author. "Understanding Digital Signal Processing." Ch. 1, 3, and 5. Web.
[5] https://en.wikipedia.org/wiki/Low-pass_filter. Accessed Nov. 7, 2019.
[6] Harris, Fred J. "Multirate Signal Processing for Communication Systems." Page 216, Equation 8.16.
[7] https://en.wikipedia.org/wiki/Nyquist_frequency. Accessed Nov. 15, 2019.
[8] Baxley, R., Henshaw, A., Nowlan, S., & Trewhitt, E. *Juyter Notebook for Sampling: Sampling.ipynb.* (2017)
[9] https://en.wikipedia.org/wiki/Nyquist_frequency. Accessed Nov. 16, 2019.
[10] https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem. Accessed Nov. 20, 2019.
[11] https://en.wikipedia.org/wiki/44,100_Hz. Accessed Nov. 20, 2019.

APPENDIX

A. Python Code for Filter Design [2]

Listing 2. Simple Averager Filter

```
num_samples = 100

# set seed value (optional)
np.random.seed(10)

# generate numpy array of random numbers
#this are the x(n) values
x = np.random.rand(num_samples)-0.5

# filter order
L = 4
h = np.ones(L)/L

plt.figure(11)
plt.stem(h)
plt.xlim([-1,4])
plt.title('Impulse Response - Coefficients of Simple Averager Filter')
N_fft = 1024 #use a power of two

# frequency response
# the second argument of fft is the number of fft bins to use.
freq_response = np.fft.fft(h,N_fft)

# Transform using DFT to get the frequency response
freq = np.fft.fftfreq(freq_response.size)


# plot filter frequency response
plt.figure(1)
plt.plot(freq, abs(freq_response), 'b.');
plt.ylabel('Amplitude', color='b');
plt.xlabel('Frequency');
plt.grid(True);
plt.title('Frequency Response of Simple Averager Filter');


# plot input frequency spectrum by
# using the FFT to convert time to frequency
x_spectrum = np.fft.fft(x,N_fft)

plt.figure(2)
plt.plot(freq, abs(x_spectrum), 'b.', label='Input');
plt.ylabel('Amplitude', color='b');
plt.xlabel('Frequency');
plt.grid(True);
plt.title('Input Signal Frequency Spectrum');

# Plot output spectrum
y = np.convolve(h,x)

#the following two lines provide the same thing
#1) output spectrum by convolution in the time domain and
#using FFT to convert to frequency domain

y_spectrum_convolve = np.fft.fft(y,N_fft)

#2) output spectrum by multiplying in the frequency domain

y_spectrum_multiply = abs(x_spectrum)*freq_response

#plt.plot(freq, abs(y_spectrum), 'r.', label='Output');
plt.plot(freq, abs(y_spectrum_convolve), 'r', label = 'Output')
plt.ylabel('Amplitude');
plt.xlabel('Frequency');
plt.grid(True);
plt.title('Signal Spectrum After Filtering (Red)');
plt.legend();
```

B. Demonstration Application

Listing 3. Demonstration Application

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
##################################################
# GNU Radio Python Flow Graph
# Title: Math of Big Data
# Generated: Mon Nov 18 11:11:20 2019
##################################################

from distutils.version import StrictVersion

if __name__ == '__main__':
    import ctypes
    import sys
    if sys.platform.startswith('linux'):
        try:
            x11 = ctypes.cdll.LoadLibrary('libX11.so')
            x11.XInitThreads()
        except:
            print "Warning: failed to XInitThreads()"

from PyQt5 import Qt
from PyQt5 import Qt, QtCore
from PyQt5.QtCore import QObject, pyqtSlot
from gnuradio import analog
from gnuradio import audio
from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio import filter
from gnuradio import gr
from gnuradio import qtgui
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from gnuradio.qtgui import Range, RangeWidget
from grc_gnuradio import blks2 as grc_blks2
from optparse import OptionParser
import math
import os
import osmosdr
import sip
import sys
import time
from gnuradio import qtgui


class bigdata(gr.top_block, Qt.QWidget):

    def __init__(self):
        gr.top_block.__init__(self, "Math of Big Data")
        Qt.QWidget.__init__(self)
        self.setWindowTitle("Math of Big Data")
        qtgui.util.check_set_qss()
        try:
            self.setWindowIcon(Qt.QIcon.fromTheme('gnuradio-grc'))
        except:
            pass
        self.top_scroll_layout = Qt.QVBoxLayout()
        self.setLayout(self.top_scroll_layout)
        self.top_scroll = Qt.QScrollArea()
        self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
        self.top_scroll_layout.addWidget(self.top_scroll)
        self.top_scroll.setWidgetResizable(True)
        self.top_widget = Qt.QWidget()
        self.top_scroll.setWidget(self.top_widget)
        self.top_layout = Qt.QVBoxLayout(self.top_widget)
        self.top_grid_layout = Qt.QGridLayout()
        self.top_layout.addLayout(self.top_grid_layout)

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

        if StrictVersion(Qt.qVersion()) < StrictVersion("5.0.0"):
```

```
        self.restoreGeometry(self.settings.value("geometry").toByteArray())
    else:
        self.restoreGeometry(self.settings.value("geometry", type=QtCore.QByteArray))

    ##################################################
    # Variables
    ##################################################
    self.sig_src = sig_src = 0
    self.samp_rate = samp_rate = 44100
    self.rf_rate = rf_rate = 1000000
    self.fc = fc = 903000000
    self.chooser = chooser = 0
    self.amplitude = amplitude = 0.5

    ##################################################
    # Blocks
    ##################################################
    self._sig_src_options = (0, 1, 2, )
    self._sig_src_labels = ('Silence', 'Recorded', 'Tones', )
    self._sig_src_group_box = Qt.QGroupBox('Audio Source')
    self._sig_src_box = Qt.QHBoxLayout()
    class variable_chooser_button_group(Qt.QButtonGroup):
        def __init__(self, parent=None):
            Qt.QButtonGroup.__init__(self, parent)
        @pyqtSlot(int)
        def updateButtonChecked(self, button_id):
            self.button(button_id).setChecked(True)
    self._sig_src_button_group = variable_chooser_button_group()
    self._sig_src_group_box.setLayout(self._sig_src_box)
    for i, label in enumerate(self._sig_src_labels):
            radio_button = Qt.QRadioButton(label)
            self._sig_src_box.addWidget(radio_button)
            self._sig_src_button_group.addButton(radio_button, i)
    self._sig_src_callback = lambda i: Qt.QMetaObject.invokeMethod(self._sig_src_button_group,
                                                                   "updateButtonChecked",
                                                                   Qt.Q_ARG("int",
                                                                            self._sig_src_options.index(i)))
    self._sig_src_callback(self.sig_src)
    self._sig_src_button_group.buttonClicked[int].connect(
            lambda i: self.set_sig_src(self._sig_src_options[i]))
    self.top_layout.addWidget(self._sig_src_group_box)
    self._chooser_options = (0, 1, 2, 3, 4, )
    self._chooser_labels = ('None', '15 KHz', '8 KHz', '4 KHz', '1 KHz', )
    self._chooser_group_box = Qt.QGroupBox('Filter')
    self._chooser_box = Qt.QHBoxLayout()
    class variable_chooser_button_group(Qt.QButtonGroup):
        def __init__(self, parent=None):
            Qt.QButtonGroup.__init__(self, parent)
        @pyqtSlot(int)
        def updateButtonChecked(self, button_id):
            self.button(button_id).setChecked(True)
    self._chooser_button_group = variable_chooser_button_group()
    self._chooser_group_box.setLayout(self._chooser_box)
    for i, label in enumerate(self._chooser_labels):
            radio_button = Qt.QRadioButton(label)
            self._chooser_box.addWidget(radio_button)
            self._chooser_button_group.addButton(radio_button, i)
    self._chooser_callback = lambda i: Qt.QMetaObject.invokeMethod(self._chooser_button_group,
                                                                   "updateButtonChecked",
                                                                   Qt.Q_ARG("int",
                                                                            self._chooser_options.index(i)))
    self._chooser_callback(self.chooser)
    self._chooser_button_group.buttonClicked[int].connect(
            lambda i: self.set_chooser(self._chooser_options[i]))
    self.top_layout.addWidget(self._chooser_group_box)
    self._amplitude_range = Range(0, 1, 0.05, 0.5, 200)
    self._amplitude_win = RangeWidget(self._amplitude_range,
                                      self.set_amplitude, 'Volume', "counter_slider", float)
    self.top_layout.addWidget(self._amplitude_win)
    self.rational_resampler_xxx_0 = filter.rational_resampler_ccc(
            interpolation=rf_rate,
            decimation=44100,
            taps=None,
            fractional_bw=None,
```

```
)
self.qtgui_time_sink_x_0 = qtgui.time_sink_f(
        1024, #size
        samp_rate, #samp_rate
        "", #name
        1 #number of inputs
)
self.qtgui_time_sink_x_0.set_update_time(0.010)
self.qtgui_time_sink_x_0.set_y_axis(0, 0.5)

self.qtgui_time_sink_x_0.set_y_label('Amplitude', "")

self.qtgui_time_sink_x_0.enable_tags(-1, True)
self.qtgui_time_sink_x_0.set_trigger_mode(qtgui.TRIG_MODE_FREE, qtgui.TRIG_SLOPE_POS, 0.1, 0, 0, "")
self.qtgui_time_sink_x_0.enable_autoscale(False)
self.qtgui_time_sink_x_0.enable_grid(True)
self.qtgui_time_sink_x_0.enable_axis_labels(True)
self.qtgui_time_sink_x_0.enable_control_panel(False)
self.qtgui_time_sink_x_0.enable_stem_plot(False)

if not False:
  self.qtgui_time_sink_x_0.disable_legend()

labels = ['', '', '', '', '',
          '', '', '', '', '']
widths = [1, 1, 1, 1, 1,
          1, 1, 1, 1, 1]
colors = ["blue", "red", "green", "black", "cyan",
          "magenta", "yellow", "dark red", "dark green", "blue"]
styles = [1, 1, 1, 1, 1,
          1, 1, 1, 1, 1]
markers = [-1, -1, -1, -1, -1,
           -1, -1, -1, -1, -1]
alphas = [1.0, 1.0, 1.0, 1.0, 1.0,
          1.0, 1.0, 1.0, 1.0, 1.0]

for i in xrange(1):
    if len(labels[i]) == 0:
        self.qtgui_time_sink_x_0.set_line_label(i, "Data {0}".format(i))
    else:
        self.qtgui_time_sink_x_0.set_line_label(i, labels[i])
    self.qtgui_time_sink_x_0.set_line_width(i, widths[i])
    self.qtgui_time_sink_x_0.set_line_color(i, colors[i])
    self.qtgui_time_sink_x_0.set_line_style(i, styles[i])
    self.qtgui_time_sink_x_0.set_line_marker(i, markers[i])
    self.qtgui_time_sink_x_0.set_line_alpha(i, alphas[i])

self._qtgui_time_sink_x_0_win = sip.wrapinstance(self.qtgui_time_sink_x_0.pyqwidget(), Qt.QWidget)
self.top_layout.addWidget(self._qtgui_time_sink_x_0_win)
self.qtgui_freq_sink_x_0 = qtgui.freq_sink_c(
        1024, #size
        firdes.WIN_BLACKMAN_hARRIS, #wintype
        0, #fc
        samp_rate, #bw
        "", #name
        1 #number of inputs
)
self.qtgui_freq_sink_x_0.set_update_time(0.01)
self.qtgui_freq_sink_x_0.set_y_axis(-140, 10)
self.qtgui_freq_sink_x_0.set_y_label('Relative Gain', 'dB')
self.qtgui_freq_sink_x_0.set_trigger_mode(qtgui.TRIG_MODE_FREE, 0.0, 0, "")
self.qtgui_freq_sink_x_0.enable_autoscale(False)
self.qtgui_freq_sink_x_0.enable_grid(True)
self.qtgui_freq_sink_x_0.set_fft_average(1.0)
self.qtgui_freq_sink_x_0.enable_axis_labels(True)
self.qtgui_freq_sink_x_0.enable_control_panel(False)

if not False:
  self.qtgui_freq_sink_x_0.disable_legend()

if "complex" == "float" or "complex" == "msg_float":
  self.qtgui_freq_sink_x_0.set_plot_pos_half(not True)

labels = ['', '', '', '', '',
```

```
            '', '', '', '', '']
        widths = [1, 1, 1, 1, 1,
            1, 1, 1, 1, 1]
        colors = ["blue", "red", "green", "black", "cyan",
            "magenta", "yellow", "dark red", "dark green", "dark blue"]
        alphas = [1.0, 1.0, 1.0, 1.0, 1.0,
            1.0, 1.0, 1.0, 1.0, 1.0]
        for i in xrange(1):
            if len(labels[i]) == 0:
                self.qtgui_freq_sink_x_0.set_line_label(i, "Data {0}".format(i))
            else:
                self.qtgui_freq_sink_x_0.set_line_label(i, labels[i])
            self.qtgui_freq_sink_x_0.set_line_width(i, widths[i])
            self.qtgui_freq_sink_x_0.set_line_color(i, colors[i])
            self.qtgui_freq_sink_x_0.set_line_alpha(i, alphas[i])

        self._qtgui_freq_sink_x_0_win = sip.wrapinstance(self.qtgui_freq_sink_x_0.pyqwidget(), Qt.QWidget)
        self.top_layout.addWidget(self._qtgui_freq_sink_x_0_win)
        self.osmosdr_sink_0 = osmosdr.sink( args="numchan=" + str(1) + " " + '' )
        self.osmosdr_sink_0.set_sample_rate(rf_rate)
        self.osmosdr_sink_0.set_center_freq(fc, 0)
        self.osmosdr_sink_0.set_freq_corr(0, 0)
        self.osmosdr_sink_0.set_gain(6, 0)
        self.osmosdr_sink_0.set_if_gain(0, 0)
        self.osmosdr_sink_0.set_bb_gain(60, 0)
        self.osmosdr_sink_0.set_antenna('', 0)
        self.osmosdr_sink_0.set_bandwidth(0, 0)

    #Below the output is cutoff since the list contains a all the filter coefficients.
        self.fft_filter_xxx_1 = filter.fft_filter_ccc(1, ([0.003903353586792946, 0.004398836754262447, 0.0048207263462
        self.fft_filter_xxx_1.declare_sample_delay(0)
        self.fft_filter_xxx_0_1 = filter.fft_filter_ccc(1, ([0.0028788824565708637, 0.0022246302105486393, -0.0054065
        self.fft_filter_xxx_0_1.declare_sample_delay(0)
        self.fft_filter_xxx_0_0 = filter.fft_filter_ccc(1, ([0.0018509112996980548, 0.00425605196505785, 0.00537562184
        self.fft_filter_xxx_0_0.declare_sample_delay(0)
        self.fft_filter_xxx_0 = filter.fft_filter_ccc(1, ([-0.003558691358193755, -0.005211095791310072, -0.0007477444
        self.fft_filter_xxx_0.declare_sample_delay(0)
        self.blocks_wavfile_source_0 = blocks.wavfile_source('/home/andy/Documents/flight_of_the_bumblebee.wav', True)
        self.blocks_rotator_cc_0 = blocks.rotator_cc(math.pi)
        self.blocks_multiply_const_vxx_0 = blocks.multiply_const_vcc((amplitude, ))
        self.blocks_float_to_complex_0 = blocks.float_to_complex(1)
        self.blocks_complex_to_real_0 = blocks.complex_to_real(1)
        self.blocks_add_xx_0 = blocks.add_vcc(1)
        self.blocks_add_const_vxx_0 = blocks.add_const_vcc((0.5, ))
        self.blks2_selector_0_0 = grc_blks2.selector(
                item_size=gr.sizeof_gr_complex*1,
                num_inputs=3,
                num_outputs=1,
                input_index=sig_src,
                output_index=0,
        )
        self.blks2_selector_0 = grc_blks2.selector(
                item_size=gr.sizeof_gr_complex*1,
                num_inputs=5,
                num_outputs=1,
                input_index=chooser,
                output_index=0,
        )
        self.audio_sink_0 = audio.sink(samp_rate, '', True)
        self.analog_sig_source_x_0_0 = analog.sig_source_c(samp_rate, analog.GR_COS_WAVE, 440, 0.4, 0)
        self.analog_sig_source_x_0 = analog.sig_source_c(samp_rate, analog.GR_COS_WAVE, 3520, 0.1, 0)
        self.analog_const_source_x_0_0 = analog.sig_source_c(0, analog.GR_CONST_WAVE, 0, 0, 0)
        self.analog_const_source_x_0 = analog.sig_source_f(0, analog.GR_CONST_WAVE, 0, 0, 0)

        ##################################################
        # Connections
        ##################################################
        self.connect((self.analog_const_source_x_0, 0), (self.blocks_float_to_complex_0, 1))
        self.connect((self.analog_const_source_x_0_0, 0), (self.blks2_selector_0_0, 0))
        self.connect((self.analog_sig_source_x_0, 0), (self.blocks_add_xx_0, 1))
        self.connect((self.analog_sig_source_x_0_0, 0), (self.blocks_add_xx_0, 0))
        self.connect((self.blks2_selector_0, 0), (self.blocks_add_const_vxx_0, 0))
        self.connect((self.blks2_selector_0_0, 0), (self.blks2_selector_0, 0))
        self.connect((self.blks2_selector_0_0, 0), (self.fft_filter_xxx_0, 0))
```

```
        self.connect((self.blks2_selector_0_0, 0), (self.fft_filter_xxx_0_0, 0))
        self.connect((self.blks2_selector_0_0, 0), (self.fft_filter_xxx_0_1, 0))
        self.connect((self.blks2_selector_0_0, 0), (self.fft_filter_xxx_1, 0))
        self.connect((self.blocks_add_const_vxx_0, 0), (self.blocks_multiply_const_vxx_0, 0))
        self.connect((self.blocks_add_xx_0, 0), (self.blks2_selector_0_0, 2))
        self.connect((self.blocks_complex_to_real_0, 0), (self.audio_sink_0, 0))
        self.connect((self.blocks_complex_to_real_0, 0), (self.qtgui_time_sink_x_0, 0))
        self.connect((self.blocks_float_to_complex_0, 0), (self.blks2_selector_0_0, 1))
        self.connect((self.blocks_multiply_const_vxx_0, 0), (self.blocks_complex_to_real_0, 0))
        self.connect((self.blocks_multiply_const_vxx_0, 0), (self.qtgui_freq_sink_x_0, 0))
        self.connect((self.blocks_multiply_const_vxx_0, 0), (self.rational_resampler_xxx_0, 0))
        self.connect((self.blocks_rotator_cc_0, 0), (self.osmosdr_sink_0, 0))
        self.connect((self.blocks_wavfile_source_0, 0), (self.blocks_float_to_complex_0, 0))
        self.connect((self.fft_filter_xxx_0, 0), (self.blks2_selector_0, 2))
        self.connect((self.fft_filter_xxx_0_0, 0), (self.blks2_selector_0, 3))
        self.connect((self.fft_filter_xxx_0_1, 0), (self.blks2_selector_0, 1))
        self.connect((self.fft_filter_xxx_1, 0), (self.blks2_selector_0, 4))
        self.connect((self.rational_resampler_xxx_0, 0), (self.blocks_rotator_cc_0, 0))

    def closeEvent(self, event):
        self.settings = Qt.QSettings("GNU Radio", "bigdata")
        self.settings.setValue("geometry", self.saveGeometry())
        event.accept()

    def setStyleSheetFromFile(self, filename):
        try:
            if not os.path.exists(filename):
                filename = os.path.join(
                    gr.prefix(), "share", "gnuradio", "themes", filename)
            with open(filename) as ss:
                self.setStyleSheet(ss.read())
        except Exception as e:
            print >> sys.stderr, e

    def get_sig_src(self):
        return self.sig_src

    def set_sig_src(self, sig_src):
        self.sig_src = sig_src
        self._sig_src_callback(self.sig_src)
        self.blks2_selector_0_0.set_input_index(int(self.sig_src))

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.qtgui_time_sink_x_0.set_samp_rate(self.samp_rate)
        self.qtgui_freq_sink_x_0.set_frequency_range(0, self.samp_rate)
        self.analog_sig_source_x_0_0.set_sampling_freq(self.samp_rate)
        self.analog_sig_source_x_0.set_sampling_freq(self.samp_rate)

    def get_rf_rate(self):
        return self.rf_rate

    def set_rf_rate(self, rf_rate):
        self.rf_rate = rf_rate
        self.osmosdr_sink_0.set_sample_rate(self.rf_rate)

    def get_fc(self):
        return self.fc

    def set_fc(self, fc):
        self.fc = fc
        self.osmosdr_sink_0.set_center_freq(self.fc, 0)

    def get_chooser(self):
        return self.chooser

    def set_chooser(self, chooser):
        self.chooser = chooser
        self._chooser_callback(self.chooser)
        self.blks2_selector_0.set_input_index(int(self.chooser))
```

```
    def get_amplitude(self):
        return self.amplitude

    def set_amplitude(self, amplitude):
        self.amplitude = amplitude
        self.blocks_multiply_const_vxx_0.set_k((self.amplitude, ))


def main(top_block_cls=bigdata, options=None):
    if gr.enable_realtime_scheduling() != gr.RT_OK:
        print "Error: failed to enable real-time scheduling."

    if StrictVersion("4.5.0") <= StrictVersion(Qt.qVersion()) < StrictVersion("5.0.0"):
        style = gr.prefs().get_string('qtgui', 'style', 'raster')
        Qt.QApplication.setGraphicsSystem(style)
    qapp = Qt.QApplication(sys.argv)

    tb = top_block_cls()
    tb.start()
    tb.setStyleSheetFromFile('/home/andy/math_of_big_data/projector.qss')
    tb.show()

    def quitting():
        tb.stop()
        tb.wait()
    qapp.aboutToQuit.connect(quitting)
    qapp.exec_()


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

C. Python Code for Sampling [2]

Listing 4. Sampling

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

# In[1]:


# Imports
get_ipython().run_line_magic('matplotlib', 'inline')
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
from pylab import rcParams
rcParams['figure.figsize'] = (9.0, 5.0)
import numpy as np
from ipywidgets import interactive
from IPython.display import display
from scipy import signal


# ### Sampling
# Let's explore the phenomenon of aliasing.
#
# ### Step 1: Create Signal
# Define a signal that is extremely oversampled:

# In[3]:


f_c = 10
sample_rate = 1000.0

# create time samples
t = np.arange(0,100,1.0/sample_rate)

# create signal values
x = np.cos(2*np.pi*f_c*t)


# plot time domain
plt.figure(1)
plt.plot(t, x, 'g');
plt.ylabel('Amplitude', color='g');
plt.xlabel('Time [s]');
plt.xlim([0,1]);


# ### Step 2: Subsample
# Now, let's overlay a subsampled discrete time version of this signal.  The carrier is 10Hz, which mean the Nyquist
# rate is 20Hz.  If we sample at a rate less than 20Hz, then there will be aliasing.
To explore this, start by
# sampling at 15Hz.

# In[4]:


sub_sample_rate = 18.0

# create time samples
t_sub_sample = np.arange(0,100,1.0/sub_sample_rate)

# create signal values
x_sub_sample = np.cos(2*np.pi*f_c*t_sub_sample)


# plot time domain
plt.figure(1)
plt.plot(t, x, 'g');
plt.stem(t_sub_sample, x_sub_sample, 'b', basefmt=':');
plt.ylabel('Amplitude');
plt.xlabel('Time [s]');
```

```
plt.xlim([0,1]);


# ### Step 3: Aliasing
# Nyquist tells us that this discrete-time signal is a subsampled version of the continous time signal.
# The implication is that the original signal cannot be perfectly recovered. Instead, when we try to recover
# the continous time signal from this sampled version, we get a different sinusoid.
The question is: which one?
#
# #### Math
# The continous-time signal is $x(t) = \cos(2\pi 10 t)$.  When we sample at 18Hz, we are left with
# \begin{eqnarray}
# x[n] &=& \cos(2\pi 10 n/18) \\&=& \cos(10 \pi n /9) \\&=& \cos(10 \pi n /9 - 2\pi n) \\&=& \cos(-8 \pi n /9) \\&=&
# \cos(8 \pi n /9).
# \end{eqnarray}
#
# To convert to continous time, "apply" an ideal DAC, we can substitute $n=18 t$, and are left with
# \begin{eqnarray}
# x[n] &=& \cos(8 \pi 18 t /9) \\&=& \cos(2\pi 8 t).
# \end{eqnarray}
#

# In[5]:


f_c_alias = 8
sample_rate = 1000.0

# create time samples
t = np.arange(0,100,1.0/sample_rate)

# create signal values
x_alias = np.cos(2*np.pi*f_c_alias*t)


# plot time domain
plt.figure(1)
plt.plot(t, x, 'g:');
plt.stem(t_sub_sample, x_sub_sample, 'b', basefmt=':');
plt.plot(t, x_alias, 'r');
plt.ylabel('Amplitude');
plt.xlabel('Time [s]');
plt.xlim([0,1]);


# ### Step 4: Example 2
# The continous-time signal is still $x(t) = \cos(2\pi 10 t)$.
#
# Now let's try to subsample even more.  For example, let's sample at 6Hz:
# \begin{eqnarray}
# x[n] &=& \cos(2\pi 10 n/6) \\&=& \cos(10 \pi n /3) \\&=& \cos(10 \pi n /3 - 2\pi n) \\&=& \cos(1 \pi n /3 + 3\pi n).
# \end{eqnarray}
#
# If we subtract $4\pi n$ from the argument, we are left the same signal.
# \begin{eqnarray}
# x[n] &=& \cos(1 \pi n /3 + 3\pi n - 4\pi n) \\&=& \cos(-2 \pi n /3) \\&=& \cos(2 \pi n /3).
# \end{eqnarray}
#
# To convert to continous time, "apply" an ideal DAC, we can substitute $n=18 t$, and are left with
# \begin{eqnarray}
# x[n] &=& \cos(2 \pi 6 t /3) \\&=& \cos(2\pi 2 t).
# \end{eqnarray}

# In[8]:


f_c_alias = 2
f_c = 10
sample_rate = 1000.0

# create time samples
t = np.arange(0,100,1.0/sample_rate)

# create signal values
```

```
x = np.cos(2*np.pi*f_c*t)

# create signal values
x_alias = np.cos(2*np.pi*f_c_alias*t)

sub_sample_rate = 6.0

# create time samples
t_sub_sample = np.arange(0,100,1.0/sub_sample_rate)

# create signal values
x_sub_sample = np.cos(2*np.pi*f_c*t_sub_sample)

# plot time domain
plt.figure(1)
plt.plot(t, x, 'g:');
plt.stem(t_sub_sample, x_sub_sample, 'b', basefmt=':', use_line_collection=True);
plt.plot(t, x_alias, 'r');
plt.ylabel('Amplitude');
plt.xlabel('Time [s]');
plt.xlim([0,1]);


# ### Step 5: Discrete-Time Spectrum
#
# The discrete time spectrum is indexed by $\hat{\omega} \in (-\pi, \pi]$.  For a signal that is a sinusoid,
# it will have the form $x[n] = \cos(\hat{\omega} n)$.  The value of $\hat{\omega}$ is where the discrete
# time spectrum will have a spectral line.  Going off of the previous example,
# $x[n] = \cos\left(\frac{2}{3}\pi n\right)$; that is $\hat{\omega} = \frac{2}{3}\pi$.
#
# To compute this spectrum, we simply take the DFT (FFT) of the discrete-time signal.
# In the previous notebook, we were actually "faking" the continuous-time spectrum to some extent.
#
#
# #### Side Note
# When solving aliasing problems, it is a handy rule of thumb to remember that
# $\hat{\omega} \in (-\pi, \pi]$.  For example, if you are trying to compute the discrete time signal
# $x[n] = \cos(\hat{\omega} n)$, and $\hat{\omega} \notin (-\pi, \pi]$, then you have simplified the
# argument of $\cos()$ enough.
#
#

# In[10]:


f_c_alias = 2
f_c = 10
sample_rate = 1000.0

# create time samples
t = np.arange(0,100,1.0/sample_rate)

# create signal values
x = np.cos(2*np.pi*f_c*t)

# create signal values
x_alias = np.cos(2*np.pi*f_c_alias*t)

sub_sample_rate = 6.0

# create time samples
t_sub_sample = np.arange(0,100,1.0/sub_sample_rate)

# create signal values
x_sub_sample = np.cos(2*np.pi*f_c*t_sub_sample)

# plot time domain
plt.figure(1)
plt.plot(t, x, 'g:');
plt.stem(t_sub_sample, x_sub_sample, 'b', basefmt=':', use_line_collection=True);
plt.plot(t, x_alias, 'r');
plt.ylabel('Amplitude');
plt.xlabel('Time [s]');
plt.xlim([0,1]);
```

```
total_signal_power = sum(abs(x_sub_sample)**2)

# Calculate DFT
spectrum = np.fft.fft(x_sub_sample)

# DFT frequencies
freq = np.fft.fftfreq(x_sub_sample.size)

plt.figure(2)
plt.plot(freq*2, abs(spectrum/total_signal_power), 'b');
plt.ylabel('Amplitude', color='b');
plt.xlabel('omega_hat/pi');
plt.ylim([0,1.2]);
plt.grid(True);


# ### Step 6: Compute Alias Frequency
#
# The alias frequency is $f_a = |f_s n - f_c|$, where $n$ is the integer that minimizes $f_a$.
# We can calculate this integer by $n = round(f_c/f_s)$.  Using these two formulas, we can create a
# function that returns the alias frequency.

# In[11]:


def alias_frequency(f_s,f_c):
    n = round(f_c/f_s)
    return abs(f_s*n-f_c)

# test values
print("alias_frequency(18,10) =",alias_frequency(18,10))
print("alias_frequency(6,10) =",alias_frequency(6,10))


# ### Step 7: Interactive Alias Demo
# Using the bits of code from this and previous excercises, create an interactive element that allows
# you to explore the effect of aliasing.  Specifically, create sliders for:
#
# #### Input
# - sample frequency
# - carrier frequency
#
# #### Output
# The output should be a time plot and a spectrum plot.  The time plot should contain:
# - the original "continous-time" signal (use a sample rate of 1000Hz to approximate continous time)
# - a stem plot of the samples of the discrete-time signal sampled at _sample frequency_
# - a plot of the continuous version of the discrete-time signal after it is passed through an ideal ADC
#
# You will probably need to utilize the `alias_frequency` function to accomplish this task.

# In[ ]:


##########  Insert Code in this cell  ###########
def plot_alias_demo(f_c=10,sub_sample_rate=6.0):  # add arguments and default values



interactive(plot_alias_demo,f_c=(1,10),sub_sample_rate=(2,16.0))  # add arguments and ranges


# In[ ]:
```