COE3DY4 Lab #3
From RF Data to Mono Audio

Objective

The main objective of this lab is to understand how to use signal-flow graphs to model a real-life radio application in software. The particular case study concerns processing samples from a frequency-modulated (FM) channel to produce mono audio.

Preparation

- Revise the material from labs 1 and 2

Frequency Modulation

Frequency modulation (FM) uses the message information from the baseband signal to modulate, i.e., modify, the carrier frequency by slightly varying its frequency rather than its strength, as is the case for amplitude modulation (AM). While a rigorous mathematical treatment of FM theory is beyond the scope of this lab (or this course in general), the simple figure below captures the essence of FM modulation.

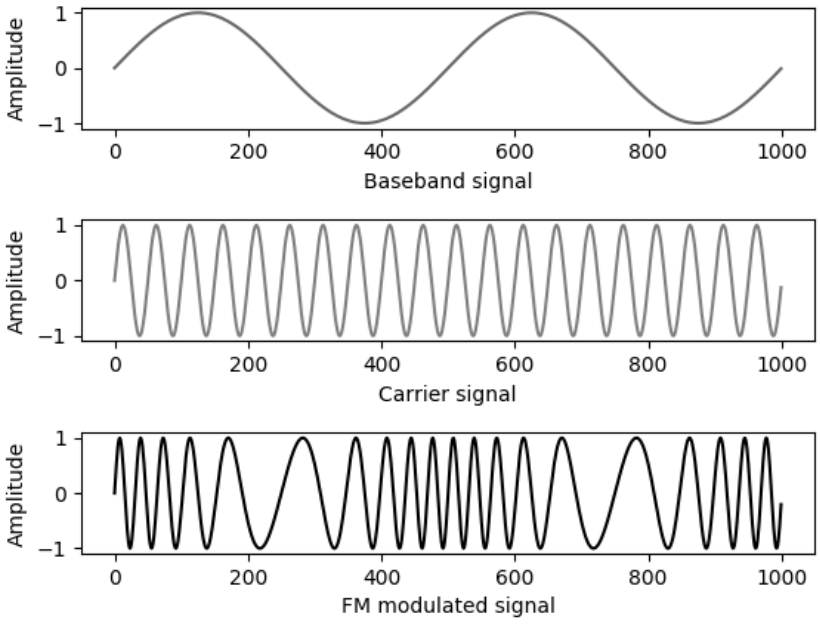

Figure 1: FM modulation in a nutshell

As the strength of the baseband signal grows (i.e., a change in the information that is transmitted), there is a slight increase in the carrier's frequency, which changes the shape of the FM-modulated signal. Conversely, the carrier's frequency decreases as the baseband's strength is reduced. Knowing the carrier's center frequency, a demodulator can measure the **rate** of change in the phase of the received signal to recover the baseband message.

Although core concepts of FM can have broad applications in data communication, they are most commonly associated with FM broadcasting, which occurs within the electromagnetic spectrum's very high frequency (VHF) range. FM broadcasting was invented nearly a century ago in the 1930s as an alternative to early radio technologies, like AM. It gained quickly in adoption, and improvements, such as stereo FM, emerged in the late 1950s. Digital data started to be transmitted through FM channels in the 1990s using standards such as Radio Data System (RDS) or Radio Broadcast Data System (RBDS) in North America. According to federal regulations in Canada [1], FM channels are allotted in the band 88-108 MHz with 200 kHz spacing. The center frequencies for each channel begin at 88.1 MHz and continue to increase in 200 KHz steps up to 107.9 MHz. Usually, only alternate channels are used in the same geographic region. It is also worth mentioning that the reception distances for FM stations are generally in the range of 50 km, though they can vary based on the transmit power, obstacles, interference, etc.

While the early FM stations would broadcast only mono audio, the spectrum of a modern (composite) FM channel is complex. A good example of different components that can appear in an FM channel is illustrated using the figure below (from the public domain [2]). The software-defined radio (SDR) project from this course will be concerned with mono audio, stereo audio and RDS (which will be used interchangeably with RBDS). Note that the components from the subcarriers above RDS, either the phased-out standards like DirectBand or other types of multiplexed subcarriers (also called SCA or SCMO subcarriers used for paging, closed-circuit programming, radio reading service, ...), are not standardized and hence beyond the scope of our work. While the focus of this lab is solely on mono audio, which is carried at the base of each FM channel, how to use the 19KHz pilot tone for downconverting the stereo audio (that is centred at 38 KHz) or how to extract digital data from the RDS subcarrier (centred at 57 KHz) will be gradually introduced as you progress through the project.

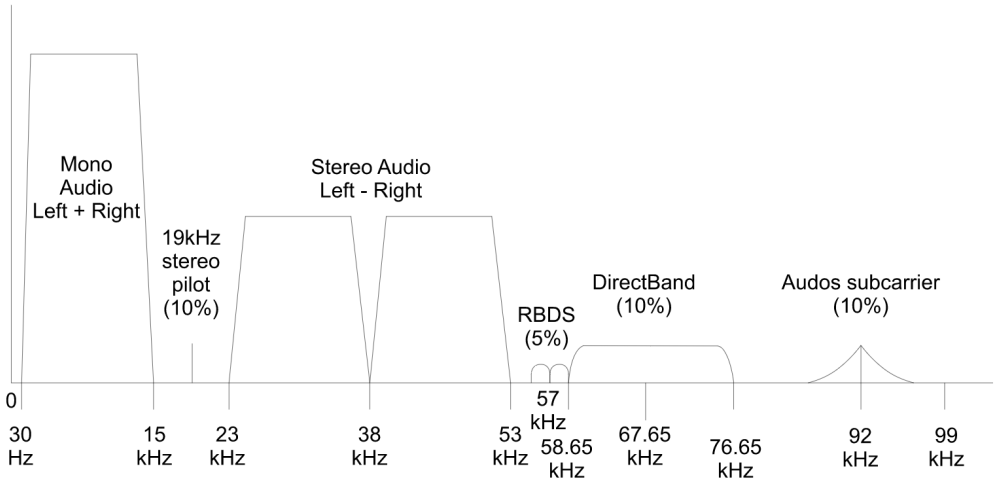

Figure 2: Spectrum of an FM channel

[1] https://www.ic.gc.ca/eic/site/smt-gst.nsf/eng/sf01153.html

[2] https://upload.wikimedia.org/wikipedia/commons/c/cd/RDS_vs_DirectBand_FM-spectrum2.svg

Before focusing on the specific work for this lab, it is worth clarifying that only the positive frequencies from the FM channel are shown in the above figure. Another important point worth mentioning is how the information from a subcarrier is extracted is a non-trivial task in the digital domain; this includes pilot tone extraction using bandpass filtering, synchronization to the center frequency of the sub-carriers using phase-locked loops (PLLs), digital demodulation of binary phase shift keyed (BPSK) data, etc. Hence, although FM broadcasting is an old technology, the computing challenges provide a great learning opportunity to write real-time software on **real** data for a *real-life* application.

FM Demodulation in SDRs

Many types of FM receivers have been deployed in practice. Due to the state of technology in the early days of FM broadcasting, which was fundamentally analog, most of the literature is focused on circuit-type technologies. How these types of technologies can be digitized is not self-evident; nevertheless, the experiments from this lab will provide the much-needed introductions.

As a starting point, if radio-frequency (RF) data can be sampled at the center frequency of an FM channel and downconverted by RF hardware to streams of digital data, all the processing can be done in software on state-of-the-art processors. The most common input format to SDR systems is I/Q data, where I stands for the in-phase component and Q stands for the quadrature component of an RF signal. A newcomer to SDRs might find these simple terms overwhelming, especially because, in some references, they can be overcomplicated with overly rigorous (and possibly even counter-intuitive) formalism. To keep it simple and at the more intuitive level, the RF signal can change in time (t) using its amplitude A, frequency f and phase ϕ as $Acos(2\pi ft+\phi)$. The RF hardware receiver will mix the RF signal with a cosine at the tuning frequency f to produce the in-phase component I and a sine at the same frequency f (using the cosine from the same local oscillator phase shifted by $\frac{\pi}{2}$) to produce the quadrature component Q [3]. Since the center frequency f used for RF sampling is already known, the digitized I and Q values carry the remaining information from the original RF signal. More precisely: $A = \sqrt{I^2 + Q^2}$ and $\phi = tan^{-1}(\frac{Q}{I})$. While this paragraph has summarized the most critical details concerning how I and Q sample values relate to the RF signal, you are encouraged to read further one of the most accessible introductions to I/Q data at `https://pysdr.org/content/sampling.html`.

We are at a point where we can start building our first signal-flow graph, which is an abstract representation that will be used both for modelling the FM receiver in `Python` and for its real-time implementation in `C++`. The signal-flow graph for the front end of the FM receiver is shown in the figure below (note that an abstract interface to RF hardware that produces I/Q data is also shown).

The case study in this lab assumes that the I/Q data has been sampled by the RF hardware at 2.4 Msamples/sec. Multiple sampling rates (or even ranges of sampling rates) can be supported depending on the type of RF hardware. For the project, other sampling rates might be used; nonetheless, in the context of this lab, you will work only with I/Q data that has been sampled at 2.4 Msamples/sec at a user-configurable center frequency.

The I/Q data is interleaved, and therefore, it is separated into two different data streams for processing in software. Before the FM data is demodulated, the signal-flow graph will have two separate processing channels: one for the I data and another for the Q data. In each of these processing channels, a low-pass filter with a cutoff frequency of 100KHz is used to extract the FM

[3]The internal structure of RF receivers, including the local oscillators, mixers, analog low-pass filters, etc, is beyond the scope of this course. Further details can be found in textbooks on analog/RF circuits.

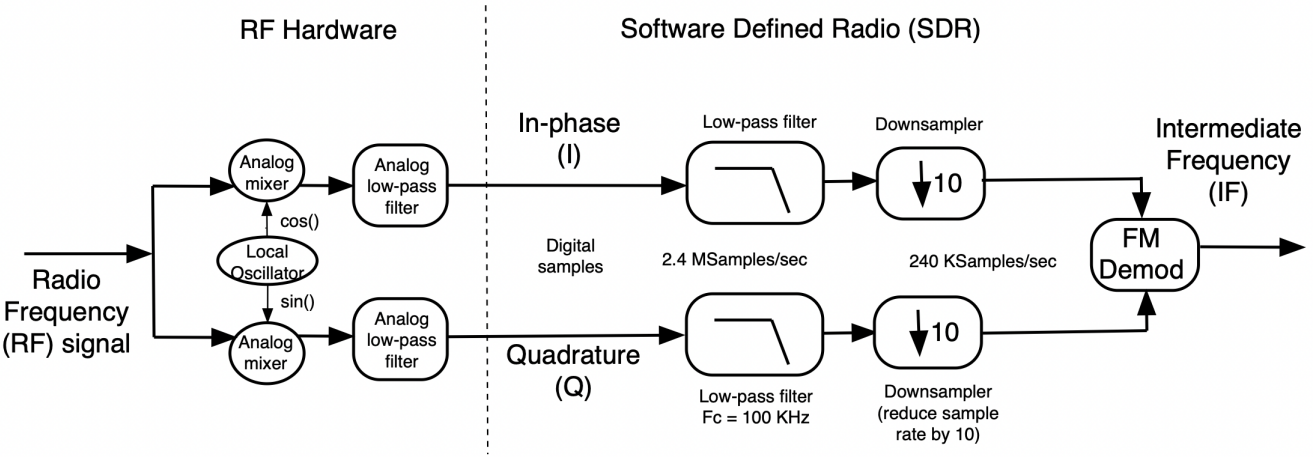

Figure 3: The signal-flow graph from RF samples in I/Q format to the demodulated FM channel

channel at the center frequency used for RF sampling. Note that the bandwidth of the input RF signal can be much larger than that of an FM channel; as per the Nyquist criterion, it can be up to half of the sampling rate, which is 2.4 Msamples/sec for this case study.

After the front-end low-pass filtering with a 100KHz cutoff frequency, the data in each of the two processing channels (for I and Q respectively) is downsampled by a scale factor of 10, which produces a signal at 240 Ksamples/sec. This is achieved by keeping every tenth sample from the filtered data. Note that sample rate reduction through decimation would also require a low-pass filter for anti-aliasing (with a normalized cutoff inverse proportional to the downsampling rate). However, the low-pass filter with a cutoff frequency at 100KHz (used to extract the FM channel) serves the purpose of this anti-aliasing filter, which becomes redundant. Also, the computational cost of filtering can be reduced by computing only the samples that will not be thrown away. Although this is not required in this lab, processing only for the relevant samples will be one of the many optimization techniques that will eventually be used throughout the project to make the real-time implementation practically feasible.

The I and Q data streams at 240 Ksamples/sec are input to the FM demodulator. As alluded to in the opening paragraph of this section, the vast amount of circuits-focused literature might obfuscate the simplicity of achieving FM demodulation in SDRs. Without the need to go through the detailed mathematical analysis of FM demodulation, two simple observations already made earlier provide the necessary insights for software FM demodulation: (i) as illustrated in Figure 1, when we know the carrier frequency, the message from the baseband signal can be extracted by monitoring the **rate** of change in the phase of the FM-modulated signal; (ii) the I and Q samples can generate the phase information, i.e., $tan^{-1}(\frac{Q}{I})$, which can be easily implemented in software through the *arctan* function available in most scientific libraries. These two simple observations provide the basis of the software FM demodulator that is already provided to you through `fmDemodArctan` function from `fmSupportLib.py` from the `model` sub-folder.

With all the `Python` code provided up to the FM-demodulated signal, you can monitor the FM channel using the `psd` method from `matplotlib` that computes an estimate of the power spectral density (PSD). Formally, the power spectral density of a real and stationary signal is defined as the Fourier transform of its autocorrelation function. Intuitively, it is a measure of a signal's power content versus its frequency, and it is a very useful visual tool that tells us the strength of the frequency components relative to each other. The discrete Fourier transform is at the center of computing PSD estimates. The core idea for a fast yet reasonably accurate PSD estimate is to

break a large block into segments; then the data from each segment is windowed before performing the Fourier transform on it; then the signal power is computed (i.e., the magnitude of the complex value of each frequency bin); then the signal power is translated to the decibel (dB) scale; and finally the results from each frequency bin from each segment are averaged. For further details, comments are provided in the `estimatePSD` function from `fmSupportLib.py`.

The FM-demodulated data at the intermediate frequency (IF) is subsequently broadcast to three separate processing channels, as shown in the figure below. This lab is only concerned with a simplified mono audio channel version. The stereo and the RDS channels and a revised version of the mono channel will be a part of the project.

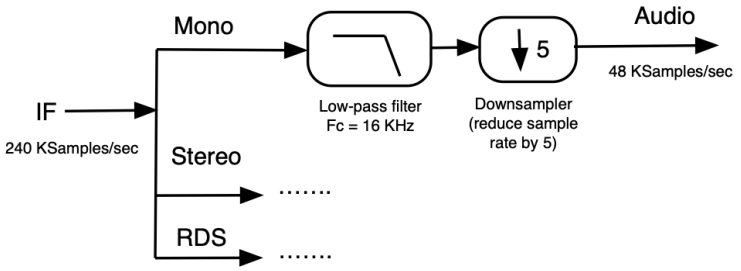

Figure 4: From the IF frequency to mono audio

In-lab experiments

With all the core concepts in place, using the detailed start-up code in both `Python` and `C++` you are asked to perform several experiments as part of your in-lab work.

With all the core concepts in place, using the detailed start-up code in both `Python` and `C++`, you are asked to perform several experiments as part of your in-lab work.

- The reference code from `fmMonoBasic.py` processes the input I/Q data available at 2.4 Msamples/sec down to the FM-demodulated signal at 240Ksamples/sec using the signal-flow graph from Figure 3. The impulse response generation is done using `firwin`, and filtering via convolution (in a single pass) is done using `lfilter` from `SciPy`. You will have to complete the code by implementing the signal-flow graph from Figure 4 to produce mono audio at 48 Ksamples/sec using **the same principles** as from the first lab and from the already-implemented code from this lab (comments, where to do the in-lab work, are provided in the code). As a side note, the code for writing the `.wav` file is already there; you must fill in the audio buffers and enjoy! Note also, due to its excessive size, the `.raw` RF data is not placed in the GitHub repo; rather, they are placed in a special folder in the virtual machine in the lab (see comments from `fmMonoBasic.py`); a separate web link will be provided via email (and in Avenue) to download it for your personal machine.

- Based on the same principle as above, the reference code from `fmMonoBlock.py` does block processing down to the FM-demodulated signal using the first signal-flow graph from Figure 3. Both `lfilter` from `SciPy` and the custom `fmDemodArctan` from `fmSupportLib.py` are already handling the proper transitioning between consecutive blocks (by saving the relevant state from the previous block). As requested above, you will need to extend the code to implement the second signal-flow graph from Figure 4 and produce the audio files. Note,

there is a flag in both `fmMonoBasic.py` and `fmMonoBlock.py` (called `il_vs_th` and assigned by default to zero) that should be used to control if your code will run in the in-lab mode (*il*) or the take-home mode (*th*).

- The partial implementation from `fmMonoAnim.py` does block processing (as `fmMonoBlock.py`), however the animation is enabled using the `animation` library from `matplotlib`. While it is not requested to update this file, you are encouraged to explore it, especially if you find animation to be a useful tool to visualize changing power spectra and understand how the state of the FM channel is changing over time (from one processed block to another). Note, you can also overlay the graph generated by the custom-built `estimatePSD` from `fmSupportLib.py` on top of the graph produced by the built-in `psd` method from `matplotlib`, and assess their resemblance (recall both of them are **estimates**).

- The final in-lab experiment is concerned with getting you familiarized with a `C++` project managed through a `Makefile` available in the `src` sub-folder. The reference code is partitioned across many `C++` files. It can log data for `gnuplot` for drawing both time-domain and frequency-domain data. As part of the in-lab, complete the requested updates in `experiment.cpp` to produce the frequency domain data for plotting.

Take-home exercises

In addition to **completing** *and* **submitting** your in-lab experiments, to further improve your understanding of DSP and FM, as well as to become more proficient with `C++`, you should perform the following:

- In `fmMonoBasic.py`, where all the processing is done in a single-pass (not quite practical but easier to manage and learn), replace the methods from `SciPy` for impulse response generation (`firwin`) and digital filtering (`lfilter`) using your **own** methods from lab 1. It is critical to note that **ONLY** the filter from Figure 4 should be replaced with your own single-pass convolution in `fmMonoBasic.py` (otherwise the computational time might be excessively high). The same requirement applies to the changes in `fmMonoBlock.py` (see below), i.e., only the audio filter operating at 240 Ksamples/sec should be replaced with your own block convolution. What "tricks of the trade" can be used to make the entire signal-flow graph from Figures 3 and 4 run in real-time will become self-evident as you progress through the project (for now, it is better to move one step at a time).

- Repeat the same task as above for `fmMonoBlock.py`, where the processing is done in blocks. Again, you should reuse your own methods for impulse response generation and block filtering from lab 1. It is also important to understand that using the *arctan* function for FM demodulation can pose a computational burden on some embedded platforms. Hence, you are asked to also replace `fmDemodArctan` from `fmSupportLib.py` with a more "computationally-friendly" method discussed at the top of `fmSupportLib.py`. Take note that in this "computationally-friendly" FM demodulator function, you must account for a proper transitioning between blocks based on the same principle of state-saving implemented in `fmDemodArctan`; nonetheless, it is essential to understand that, while the principle of state-saving is universally employed when real-time streams are processed in blocks of manageable size, the specific details are **unique** to each function in the signal-flow graph.

- The `estimatePSD` function from `fmSupportLib.py`, while not as accurate as the built-in `psd` method from `matplotlib`, it is computationally less expensive and, considering its visual relevance as an estimate, it will be used as the basis for monitoring power spectra while working in `C++`. To also further practice `C++`, it is mandatory that you implement the code from `estimatePSD` from `fmSupportLib.py` in the `C++` project from the `src` sub-folder and visualize the power spectra produced by the `C++` code in a third party tool like `gnuplot`. While the theory behind producing PSD estimates is not of critical importance, both `C++` coding and visualizing the power spectra will be part of the project experience. Therefore, it is essential to get you comfortable with both of them.

Your report should have three sections, one for each of the above items. One page and a half is sufficient, and it should not be exceeded unless there are out-of-ordinary points to be made.

Your sources should stay in the `model` and `src` sub-folders of the GitHub repo; your report (in .pdf, .txt or .md format) should be included in the `doc` sub-folder.

Your submission is due one week plus one extra day after you have started the in-lab experiments. For example, the Tuesday groups who start the lab on Tuesday, January 30, at 2:30 p.m. have this lab due before Wednesday, February 7, at 2:30 p.m.; the same principle applies to Wednesday and Thursday groups.

This lab is worth 8 % of your total grade.