COE3DY4 Lab #3
From RF Data to Mono Audio

Objective

The primary objective of this lab is to learn how to use signal-flow graphs to model real-world radio applications in software. The specific case study focuses on 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 varying its frequency slightly, rather than its amplitude, as is the case for amplitude modulation (AM). While a rigorous mathematical treatment of FM theory is beyond the scope of this lab (and this course in general), the simple figure below illustrates the fundamental concept of FM modulation.

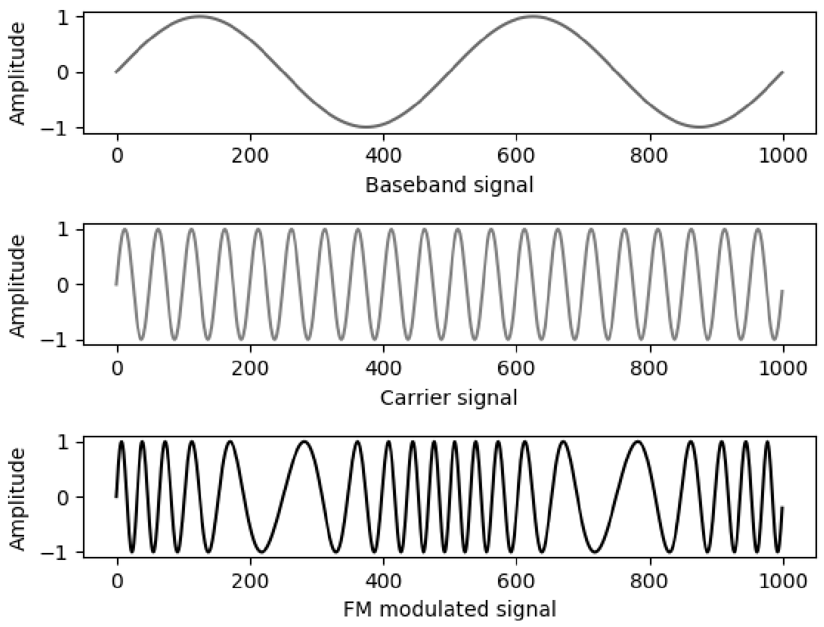

Figure 1: FM modulation in a nutshell

As the strength of the baseband signal increases (i.e., due to a change in the transmitted information), the carrier's frequency slightly increases, altering the shape of the frequency-modulated signal. Conversely, the carrier's frequency decreases as the strength of the baseband signal is reduced. By 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 the core concepts of frequency modulation 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, such as amplitude modulation. It was quickly adopted, and improvements, such as stereo FM, emerged in the late 1950s. Digital data began to be transmitted through FM channels in the 1990s using standards such as the Radio Data System (RDS) or, in North America, the Radio Broadcast Data System (RBDS). According to federal regulations in Canada[1], FM channels are allocated in the 88–108 MHz band with 200 kHz spacing. The center frequencies for each channel begin at 88.1 MHz and increase in 200 kHz steps up to 107.9 MHz. Typically, only alternate channels are used in the same geographic region to avoid interference. It is also worth noting that the reception range for FM stations is generally around 50 km, though this can vary based on transmission power, obstacles, interference, and other factors.

While early FM stations broadcast only mono audio, the spectrum of a modern (composite) FM channel is more complex. A good example of the various components that can appear in an FM channel is illustrated in the figure below (from the public domain[2]). The software-defined radio (SDR) project in this course will focus on mono audio, stereo audio, and RDS (used interchangeably with RBDS).

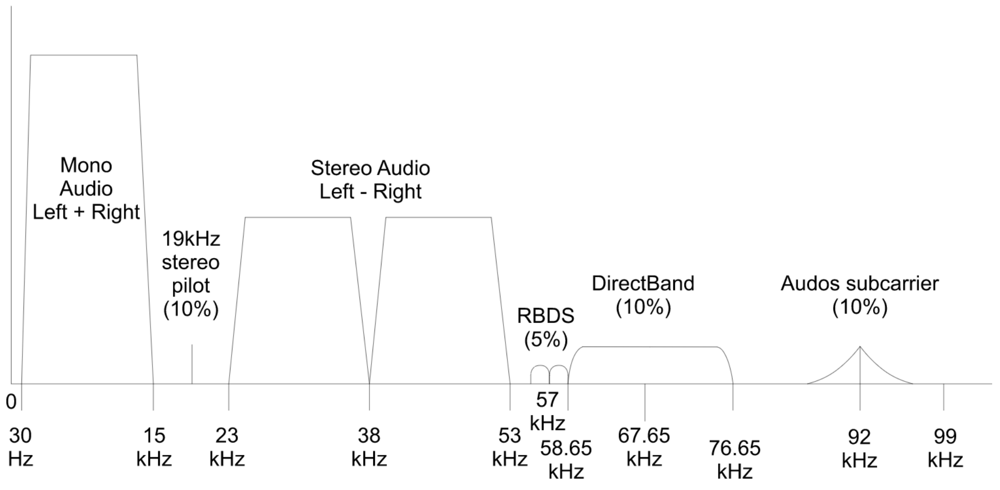

Figure 2: Spectrum of an FM channel

In addition to these core components (mono, stereo, RDS), modern FM channels often include multiplexed subcarriers that serve specialized purposes. The Subsidiary Communications Authorization (SCA) subcarrier was originally introduced to enable services such as background music distribution (e.g., music in retail stores without interruptions) and private data transmissions (e.g., paging services). SCA subcarriers operate within the FM channel but outside the main audio and RDS bandwidth. Over time, SCA evolved into Subsidiary Communications Multiplex Operation

[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

(SCMO), which focuses on digital applications, such as digital telemetry, enhanced digital reading services for the visually impaired, and encrypted data transmission for subscribers. These subcarriers are not standardized and, therefore, fall outside the scope of the project in this course. This lab will focus exclusively on mono audio, which occupies the base of each FM channel. As you progress through the project, you will gradually learn how to use the 19 kHz pilot tone for downconverting stereo audio (centered at 38 kHz) and how to extract digital data from the RDS subcarrier (centered at 57 kHz).

Before starting the specific tasks of this lab, it is important to clarify that only the positive frequencies from the FM channel are shown in the above figure. Another key point is that extracting information from a subcarrier in the digital domain is a challenging task. This includes processes such as pilot tone extraction using bandpass filtering, synchronization to the center frequency of subcarriers using phase-locked loops (PLLs), and digital demodulation of binary phase-shift keyed (BPSK) data. Although FM broadcasting is an old technology, the computational challenges involved provide a valuable opportunity to write <u>real-time</u> 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 focuses on circuit-based technologies. How these technologies can be digitized is not immediately obvious; nevertheless, the experiments in this lab will provide the necessary 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 processing can be performed in software on state-of-the-art processors. The most common input format for software-defined radio (SDR) systems is I/Q data, where I represents the in-phase component and Q represents the quadrature component of an RF signal. A newcomer to SDRs might find these terms overwhelming, especially because some references overcomplicate them with overly rigorous (and possibly counterintuitive) formalism. To keep it simple and intuitive, the RF signal can be described as a function of time (t) using its amplitude A, frequency f, and phase ϕ, as $A\cos(2\pi ft+\phi)$. The RF hardware receiver mixes the RF signal with a cosine wave at the tuning frequency f to produce the in-phase component I and with a sine wave at the same frequency f (derived by phase-shifting the cosine wave 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 retain the remaining information from the original RF signal. Specifically: $A=\sqrt{I^2+Q^2}$ and $\phi=\tan^{-1}\left(\frac{Q}{I}\right)$. While this paragraph summarizes the critical details concerning how I and Q sample values relate to the RF signal, you are encouraged to read one of the most accessible introductions to I/Q data at `https://pysdr.org/content/sampling.html`.

We are now at a point where we can begin building our first signal-flow graph, which is an abstract representation used for both modeling the FM receiver in `Python` and implementing it in real time using `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 the RF hardware, which produces I/Q data, is also included in the representation.

The case study in this lab assumes that the I/Q data has been sampled by the RF hardware at 2.4 Msamples/sec. Depending on the type of RF hardware, multiple sampling rates (or even ranges

[3]The internal structure of RF receivers, including 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.

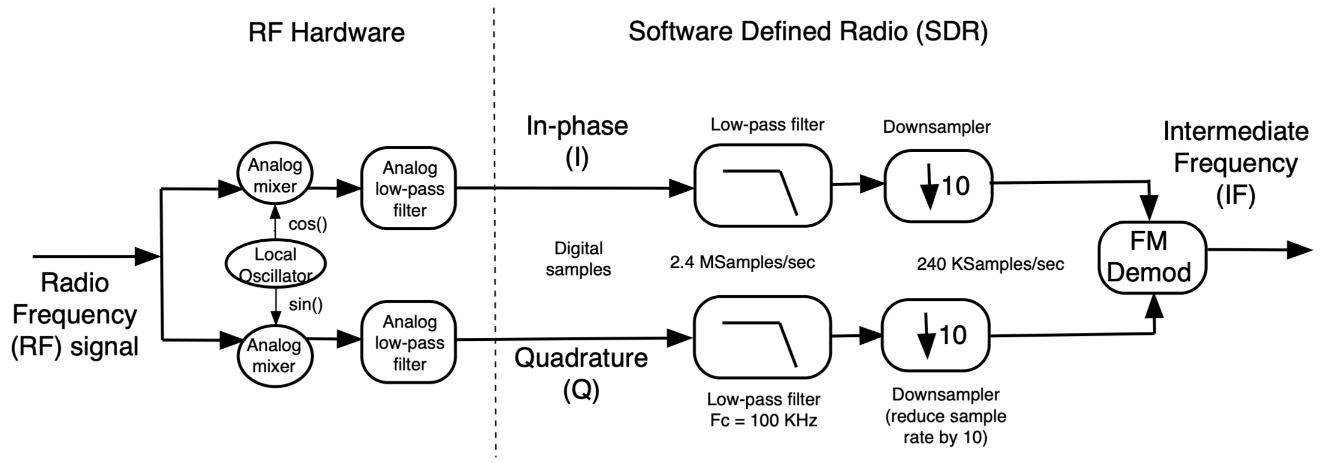

Figure 3: The signal-flow graph from RF samples in I/Q format to the demodulated FM channel

of sampling rates) can be supported. For the project, other sampling rates may be used; however, in the context of this lab, you will work exclusively with I/Q data sampled at 2.4 Msamples/sec at a user-configurable center frequency.

The I/Q data is interleaved and must be separated into two distinct data streams for processing in software. Before the FM signal is demodulated, the signal-flow graph will feature two separate processing channels: one for the I data and another for the Q data. Each of these channels applies a low-pass filter with a cutoff frequency of 100 kHz to extract the FM channel centered at the RF sampling frequency. Note that the bandwidth of the input RF signal can be much larger than that of an FM channel; according to the Nyquist criterion, it can extend up to half of the sampling rate, which, in this case study, is 2.4 Msamples/sec.

After the front-end low-pass filtering with a 100 kHz cutoff frequency, the data in each of the two processing channels (for I and Q, respectively) is downsampled by a factor of 10, resulting in a signal at 240 Ksamples/sec. This downsampling is achieved by retaining every tenth sample from the filtered data. Note that reducing the sample rate through decimation requires a low-pass filter for anti-aliasing, with a normalized cutoff frequency inversely proportional to the downsampling factor. However, the low-pass filter with a 100 kHz cutoff frequency (used to extract the FM channel) also serves as the anti-aliasing filter, making a separate filter unnecessary. Additionally, the computational cost of filtering can be reduced by calculating only the samples that will be retained after downsampling. While this optimization is not required in this lab, processing only the relevant samples will be one of many techniques introduced later in the project to ensure the real-time implementation is computationally efficient.

The I and Q data streams at 240 Ksamples/sec are input to the FM demodulator. As mentioned in the opening paragraph of this section, the vast amount of circuits-focused literature may obscure the simplicity of achieving FM demodulation in SDRs. Without going into the detailed mathematical analysis of FM demodulation, two simple observations made earlier provide the necessary insights for implementing FM demodulation in software: (i) as illustrated in Figure 1, when the carrier frequency is known, 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 be used to generate the phase information, i.e., $tan^{-1}(\frac{Q}{I})$, which can be implemented in software using the *arctan* function available in most scientific libraries. These two observations form the basis of the software FM demodulator, which is already provided to you as the `fmDemodArctan` function in `fmSupportLib.py` located in 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, which computes an estimate of the power spectral density (PSD). Formally, the PSD of a real and stationary signal is defined as the Fourier transform of its autocorrelation function. Intuitively, it measures a signal's power content as a function of frequency and serves as a useful visual tool to analyze the relative strength of frequency components. The discrete Fourier transform (DFT) is central to computing PSD estimates. The core idea for a fast yet reasonably accurate PSD estimate involves breaking a large block of data into smaller segments. Each segment is windowed before performing the Fourier transform, and the signal power is then computed (i.e., the magnitude squared of the complex value for each frequency bin). The power values from each segment are averaged across all segments to produce a smoothed estimate of the PSD. Finally, the averaged power values are translated to the decibel (dB) scale. For further details, comments are provided in the estimatePSD function from fmSupportLib.py.

The FM-demodulated data at the intermediate frequency (IF) is subsequently directed to three separate processing channels, as shown in the figure below. This lab focuses exclusively on a simplified mono audio channel. The stereo and RDS channels, as well as a revised version of the mono channel, will be introduced later as part of the project.

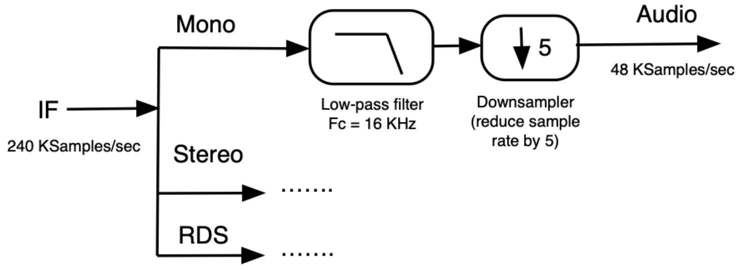

Figure 4: From the IF frequency to mono audio

In-lab experiments

With all the core concepts in place and leveraging the detailed start-up code provided in both Python and C++, you will perform several experiments as part of your in-lab work.

- The reference code in fmMonoBasic.py processes the input I/Q data, available at 2.4 Msamples/sec, down to the FM-demodulated signal at 240 Ksamples/sec using the signal-flow graph from Figure 3. The impulse response generation is handled using firwin, and filtering via convolution (in a single pass) is implemented using lfilter from SciPy. You will need to complete the code by implementing the signal-flow graph from Figure 4 to produce mono audio at 48 Ksamples/sec, applying **the same principles** as in the already-implemented code provided in this lab. Comments in the code indicate where to complete the in-lab work.

  Note that, due to its large size, the .raw RF data is not included in the GitHub repository. A web link to download the RF data will be provided via email and in Avenue.

- Based on the same principles as described above, the reference code in fmMonoBlock.py performs block processing down to the FM-demodulated signal using the signal-flow graph from Figure 3. Both lfilter from SciPy and the custom fmDemodArctan function from `fmSupportLib.py` handle the proper transitioning between consecutive blocks by saving the relevant state from the previous block.

As with the earlier task, you will need to extend the code to implement the signal-flow graph from Figure 4 to produce audio files. Note that there is a flag in both `fmMonoBasic.py` and `fmMonoBlock.py`, called `il_vs_th` (default value set to zero), which is used to control whether your code runs in in-lab mode (*il*) or take-home mode (*th*).

- The partial implementation in `fmMonoAnim.py` performs block processing similar to `fmMonoBlock.py`, but with animation enabled using the `animation` module from `matplotlib` (make sure you use `python3`). While you are not required to update this file, you are encouraged to explore it, especially if you find animation to be a useful tool for visualizing changing power spectra and understanding how the state of the FM channel evolves over time (from one processed block to the next).
- The final in-lab experiment focuses on familiarizing you with a `C++` project managed using a `Makefile` (for further details about using a `Makefile` to manage the compilation process, check the Appendix). The reference code is organized across multiple `C++` files and includes functionality to log data for `gnuplot`, enabling the visualization of both time-domain and frequency-domain data. As part of the in-lab work, you will need to complete the requested updates in `experiment.cpp` to generate frequency-domain data for plotting.

  An important note is that the data to be processed in `C++` is generated by `fmMonoBlock.py`. If you choose single precision (32-bit floats) in `Python`, you must use the same data type when reading the binary file in `C++`. The converse also applies: if 64-bit precision (double precision) is used in `Python`, it must also be used in `C++`.

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 `Python` and `C++`, you should perform the following:

- In `fmMonoBasic.py`, where all processing is performed in a single pass (a method that is not entirely practical but is easier to manage and learn), replace the methods from `SciPy` for impulse response generation (`firwin`) and digital filtering (`lfilter`) with 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 may become excessively high.

  The specific "tricks of the trade" that enable the entire signal-flow graph from Figures 3 and 4 to run in real time will become clearer as you progress through the project. For now, focus on validating that your own code can successfully produce audio samples from raw RF data.
- Repeat the same task as above for `fmMonoBlock.py`, where processing is performed in blocks. Again, you should reuse your own methods for impulse response generation and block filtering from the first lab.

  It is worth noting that using the *arctan* function for FM demodulation may pose a computational burden on some embedded platforms. Therefore, you are required to replace `fmDemodArctan` from `fmSupportLib.py` with a more "computationally-friendly" method, as discussed at the top of `fmSupportLib.py`.

In this "computationally-friendly" FM demodulator function, you must ensure proper transitioning between blocks by saving the necessary state, following the same principle used in `fmDemodArctan`. However, it is critical to understand that while the principle of state-saving is universally employed for processing real-time streams in blocks of manageable size, the specific implementation details are **unique** to each function in the signal-flow graph.

- The `estimatePSD` function from `fmSupportLib.py`, due to its importance as a visual tool for estimating power spectra, will be used as the basis for monitoring power spectra while working in `C++`. To further practice `C++`, you are required to implement the functionality of `estimatePSD` from `fmSupportLib.py` in the `C++` project located in the `src` sub-folder and visualize the resulting power spectra using a third-party tool like `gnuplot`. Visualizing the power spectra will be an integral part of the project experience.

The report must include three sections, one for each of the above tasks. Each section should be concise, with a maximum length of one page for the entire report unless there are extraordinary points to elaborate.

The source code must reside in the `model` and `src` sub-folders of your group's GitHub repository, and your report (in `.pdf`, `.txt`, or `.md` format) must be placed in the `doc` sub-folder.

Your submission, which should be pushed to the main branch of your group's GitHub repository for this lab, is due 16 hours before the start of your next lab session. Late submissions will incur penalties.

This lab contributes 6% to your total grade.

Appendix - Using a Makefile

The `Makefile` in the `src` sub-folder simplifies the process of compiling and linking multiple `C++` files in a project. To compile, simply type `make` in the folder where the `Makefile` is located.

- **Why use a Makefile?** Compiling source files individually and linking the object files manually can be slow and prone to errors. The `Makefile` automates this process by organizing the list of source files under a variable like `SRCS`:

  ```
  SRCS = experiment.cpp fourier.cpp genfunc.cpp iofunc.cpp logfunc.cpp filter.cpp
  ```

  If a new source file is added, simply update this list, and the `Makefile` will automatically include it during the next build.

- **Defining macros:** The `Makefile` supports passing macros to the `g++` compiler via variables. In this project, the `DEFINES` variable is used as an example for setting preprocessor macros. For example, to compile with the `DOUBLE` macro defined, run: `make DEFINES="-DDOUBLE"`.

- **Compilation and linking flags:** The `Makefile` uses various flags to control how files are compiled and linked:
  - **Compilation flags (`CXXFLAGS`):** Include optimizations at different levels, like `-O3` for performance or `-g -Og` for debugging, as well as warnings with `-Wall`.
  - **Linking flags (`LDFLAGS`):** Options like `-lm` for the math library or `-pthread` for threading support ensure that all necessary libraries are linked correctly.

- **Cleaning up object files:** `make clean` removes all object files and the executable, ensuring a fresh compilation when new macros or flags are applied.