COE-3DY4 Project Report

Group #18

Joshua Chan - chanj94@mcmaster.ca

Sara Armanazi - armanazs@mcmaster.ca

Clara Abadir – abadirc@mcmaster.ca

Jashanjyot Randhawa - randhj13@mcmaster.ca

April 10, 2023

Table of Contents

Introduction

The objective of this project is to use front-end radio frequency hardware such as the Realtek RTL2832U chipset, which is a RF dongle, and the Raspberry Pi 4 to implement a software-defined radio (SDR) system for the real-time reception of frequency modulated (FM) mono and stereo audio. Additionally, this includes the real-time reception of digital data sent through FM broadcast using the radio data system (RDS) protocol [1]. This entails writing software for audio data processing and output after radio signals are received. The scope of this project involves the radio frequency front-end, mono audio, stereo audio, and RDS data processing. Due to time constraints, the project was only completed for reception of FM mono and stereo audio. The RDS block was started but not fully implemented.

Project Overview

Specifications

To implement this project, many concepts and techniques are used such as frequency modulation (FM), software defined radio (SDR), and basic building blocks including finite impulse response filters, FM demodulators and more. The specifications for the four modes of operation are shown in Tables 1 and 2.

Table 1: Group specifications for 4 modes of operation [2].

Settings for group 18	Mode 0	Mode 1	Mode 2	Mode 3
RF Fs (Ksamples/sec)	2400	1152	2400	1920
IF Fs (Ksamples/sec)	240	288	240	320
Audio Fs (Ksamples/sec)	48	48	44.1	44.1

Building Blocks Foundation

Frequency modulation (FM) is a technique used to encode a message signal in a carrier wave by varying the frequency of the wave while keeping the amplitude and phase constant. On the other hand, **frequency demodulation** is used to extract the original message from the carrier. **Software-defined radio systems (SDRs)** use software to modulate and demodulate radio signals for radio communication [3]. The software performs signal processing [3] that allows reception of mono/stereo audio as well as digital data. The data that feeds the SDR system implemented in this project is driven by FM stations. **A finite impulse response (FIR)** filter takes the inverse Fourier transform of a frequency domain input to obtain a discretized time-domain continuous filter response. The FIR filter used is based on the sinc function. An ideal filter assumes an infinite response, whereas an FIR filter has a finite response, and thus, truncates the sinc function based on the number of selected sampling points. The number of sampling points is known as the number of filter taps, Ntaps. A **phase-locked loop (PLL)** is a circuit with an oscillator that creates an output signal that is adjusted to match the frequency of the input signal [4]. In the Stereo Carrier Recovery section, the PLL was used to synchronize the extracted 19KHz pilot tone from the band-pass-filter. The output of the PLL was later multiplied by a factor of 2 (NCO = 2) to produce the recovered stereo carrier to be used for mixing. **Re-sampling** involves up-sampling, low-pass filtering, then down-sampling to match the input rate to the required output rate. The purpose of re-sampling is to reduce computation load. Also, if the incoming sampling rate is not an integer multiple of the output rate, then up-sampling and down-sampling is required. This is because you cannot have fractions in discrete signal processing [5]. The purpose of the **radio data system (RDS)** is to display information such as the station name and time.

Overview

The overall process of this SDR is shown in Figure 1. It starts with the RF hardware which is used to acquire the RF signal and translate it into the digital domain [1]. Then the RF processing uses low-pass filtering, down-sampling, and FM demodulation to extract the FM channel. The FM demodulated signal is taken as inputs to the mono and stereo paths as well as RDS. The mono path goes through re-sampling and outputs the sum of the left and right audio channels. The stereo path takes the FM demodulated data and goes through two paths, one is band-pass filtering and a PLL to obtain the recovered stereo carrier. The other goes through only a band-pass filter passing a different frequency range to obtain the stereo channel. The stereo channel and carrier are mixed, filtered, and combined with the mono channel to separate and output the left and right audio channels. Finally, the RDS path goes through re-sampling, clock and data recovery and frame synchronization to output the radio text.

Implementation Details

The sequence of completing the project started with RF processing, then completing mono audio with all the modes working, then stereo audio, then getting multithreading to work with it, and then RDS. The project was developed by starting with the code and concepts from the labs.

Labs

In lab 1, we developed an understanding of basic digital signal processing primitives used in SDRs and implemented them in Python. This included discrete Fourier transform (DFT), deriving impulse response coefficients and digital filter design using convolution. Fourier transform is changing a signal from time domain to frequency domain for simpler and more efficient computational processing [6]. In the project we use Discrete Fourier Transform which discretizes both time and frequency variables. This concept is essential for digital filtering since the inverse DFT is used to obtain the impulse response. Digital filtering is done through convolution of an input signal with the impulse response coefficients. We used a FIR low-pass filter by deriving its impulse response based on the sinc function. The sinc function (time domain) is the inverse Fourier transform of the rectangular window (frequency domain). The 2nd lab was translating the same functions and code from lab 1 in python to C++. The main challenge with that was the different programming language syntax. The objective of the 3rd lab was to use signal-flow graphs to process samples from a frequency-modulated (FM) channel to produce mono audio [7]. We implemented a low-pass filter and down-sampler to process the in-phase and quadrature digital samples to a demodulated FM channel. Then using the FM demodulated channel, we performed low-pass filtering and down-sampling again to obtain the mono audio. This process helped set up the start of the project which was processing RF data and mono audio using the same concepts.

RF and Mono Processing

The RF front-end processing extracts the FM channel for 4 different modes of operation based on Table 1. Each mode has a different input RF sampling rate, intermediate frequency sampling rate and audio output sampling rate. To implement this in the code, the project is set up to take in an integer argument to represent the desired mode. The RF input is the in-phase, I, and quadrature, Q, samples from the RF dongle [1]. The process to retrieve the FM demodulated signal from these samples is the same as from lab 3. The input to the mono audio processing is the FM demodulated signal. The processing is implemented the same way as in lab 3 to retrieve the mono audio channel. The difference in this project and lab 3 at this point is how the convolution is implemented. The implementation of the convolution

function in lab 3 is efficient enough for real-time processing. For the project, the convolution function is revised to also implement up-sampling and down-sampling at the same time. The re-sampling is first done in RF processing to reduce the sampling rate from RF to the intermediate frequency (IF). Depending on the mode, the up and down-sample factors are determined by

$$\frac{\left(\frac{IF}{GCD}\right)}{\left(\frac{RF}{GCD}\right)} = \frac{U}{D},$$

where the GCD is the greatest common divisor of IF (desired frequency) and RF (input frequency), U is the up-sample factor and D is the down-sampling factor. The function only calculates the convolution of the samples being kept, while the removed samples are skipped to improve speed and efficiency. Then, re-sampling is done again in mono processing to reduce the sampling rate from IF to the audio output frequency. This is done using the same logic as before except with audio and IF sampling rates.

$$\frac{\left(\frac{audio}{GCD}\right)}{\left(\frac{IF}{GCD}\right)} = \frac{U}{D}$$

A challenge faced when getting all modes to work for the mono processing was to understand that the up-sampling factor does not only affect the indexing when calculating convolution, but that it also introduces gain. After a signal is up-sampled, the output sampling frequency increases and the output amplitude decreases. The gain is equal to the up-sampling factor. The number of taps must be multiplied by the factor of gain, and the output signal must also be by the gain.

Another challenge faced was realizing how much the –O3 optimization flag improves compilation time. Before using this flag, the audio kept lagging and aplay kept making output saying that the audio was receiving input too slowly. This was thought that the convolution algorithm was wrong and that it needed to be optimized further in C++. This helped further revise the algorithm.

When implementing the mono and RF processing part, the main function seemed to have a lot of content including initialization of variables and the actual functions. Thus, a design decision was taken to create classes in hopes of simplifying the code. All constants were initialized in their class rather than in the main function. Although it made code look simpler, it was soon discovered during the stereo processing section that it was a little hard to see the whole picture and debug the program. Next time, the places where classes are implemented would be more selective. Instead, functions may only be necessary but not an entire class.

Stereo Processing

The stereo path takes the FM demodulated signal output from the RF block as an input to two sub-blocks: stereo carrier recovery and stereo channel extraction. To demodulate and extract the message signal, we need to first understand how the stereo signal was modulated. The stereo signal was modulated using double side band suppressed carrier (DSB-SC) modulation. This means, only the sidebands are transmitted while the carrier is suppressed. To retrieve the message signal, we use coherent detection which takes the DSB-SC wave and is multiplied by the carrier [8]. Then, it is filtered by a low-pass filter. Hence, there are two sub-blocks, one for carrier recovery and the other, the actual channel signal. The stereo carrier recovery block takes the FM demodulated signal through a band-pass filter allowing frequencies 18.5kHz-19.5kHz to extract the 19kHz pilot tone (carrier). Also, it goes

through a PLL to synchronize the phase. The stereo channel extraction takes the FM demodulated signal through a band-pass filter allowing frequencies 22kHz-54kHz which is the range for the stereo audio. The recovered carrier and the stereo channel are multiplied together and are processed through digital filtering and re-sampling. Finally, they are combined with the mono audio to separate the left and right audio channels. The stereo is the difference of the left and right channels while the mono audio is the sum. Thus, the left channel is the sum of stereo and mono data, and the right channel is stereo subtract mono data.

The main challenge of the stereo processing was during debugging. Initially, we thought the left and right channels were supposed to be written and output separately as seen in the C++ code in lab two. However, at first, we noticed the audio wasn't separated as both types of music were evident in both ears. We realized we needed to write the left and right data into a single vector with each side being every other index. The left channel was the even indices while the right was odd. To discover this, we had to write the code in python for easier debugging. Initially, we had written the stereo block in C++ which was harder to debug. After translating the code to Python, we realized that our code logic was correct because the python representation was working and outputting clear audio. After this, we re-translated the same code to C++, but it still did not work. This led us to believe that something was wrong with writing the output since this was a different process in C++ than Python. In python, we wrote the audio to a wav file while in C++ it piped into standard output which is read by aplay. After careful inspection of the current output audio from C++, we realized that the audio was only playing in the left ear and noise in the right ear. We could not have the left and right written separately. Due to the delays of debugging, and the assumption that it was because of the incorrect logic and implementation in C++, much time was lost and RDS could not be fully completed. Too much time was spent to realize the issue was with writing the output in the I/O function.

While developing the stereo audio, there was still noise, clashing audio between the right and left audio, and a bit of popping noises. This was solved by adding the zero padding/all-pass filter to the mono audio before it was combined with the stereo audio. The audio needed to be padded in proportion to the number of taps that the stereo audio was phase delayed by. After experimenting with the taps to delay the mono audio, it was found that U/D *(Ntaps-1)/2 completely separated the audio without any noise. A new vector was added to store and reapply the state of the mono audio that was shifted out because of the zero padding. This solved the popping noise. This was probably because the mono audio was missing since the state was not saved.

Multi-threading

Once stereo processing was completed, we started implementing multi-threading for the RF processing and mono/stereo audio. Multi-threading allows for more efficiency in the program since separate audio processing functions can be concurrently occurring. The RF processing produces and outputs the FM demodulated data that is required as input for mono and stereo audio. Thus, the RF thread was the producer thread which copies the FM demodulated data into a queue to be used by the consumer audio thread (including mono and stereo). RDS processing would have been a second consumer thread. However, since it was not finished, it could not be added as another thread. Since mono and stereo were both in the audio thread, this led to another command line input argument to determine which output was desired. The second argument was the "audio path". An input of 0 ran the mono audio path and an input of 1 ran the stereo path.

RDS Processing

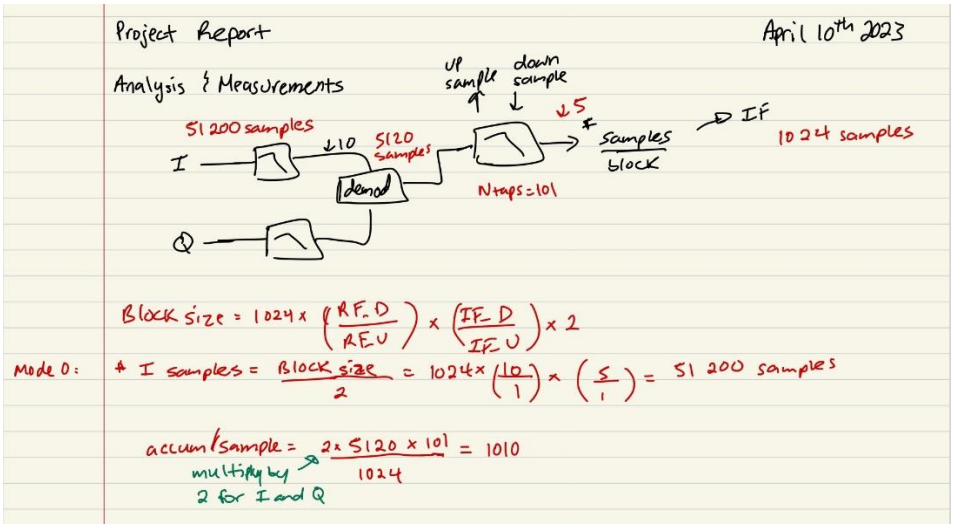
The RDS block was not completed, however, since many functions were like the mono and stereo path, it was started to some degree in python and C++. Again, the input was the FM demodulated wave which was fed into the RDS channel extraction sub-block. This block goes through a band-pass filter allowing frequencies 54kHz to 60kHz. The RDS carrier recovery contains two paths, one for the RDS channel to go through an all-pass filter for delay and the other for carrier recovery. The carrier recovery process was the same as in stereo carrier recovery, but with a minor addition before the band-pass filter. The RDS channel goes through squaring nonlinearity which multiplies the RDS channel by itself at each index. Next, the recovered RDS carrier and the RDS channel are multiplied through a mixer function and goes through low-pass filtering. Then the output of the low-pass filter goes through a root-raised cosine filter which was given in python. The RDS processing path was only completed up until this part (in Python). However, the next steps would have been clock and data recovery to complete RDS demodulation then RDS data processing.

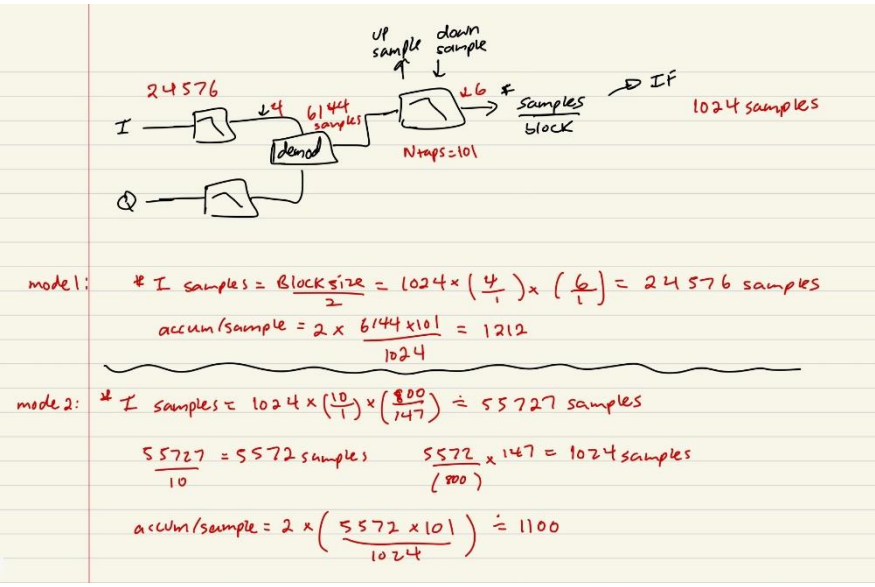
Analysis and Measurements

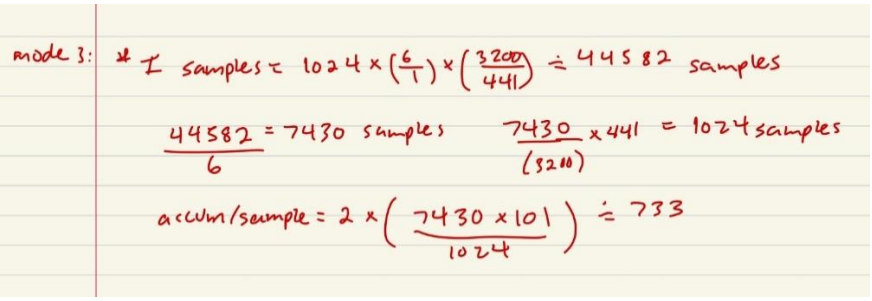
For our implementation, each block of samples are first read from the RF block which reads from the block size: Block size = 1024 * int(rf_D/rf_U) * int(if_D/if_U) * 2 where rf_D, rf_U, if_D, and if_U where rf is for the RF block and if is the signal processed by the mono block. U and D tell the up-sample and down-sample factor.

Mode	RF_U	RF_D	IF_U	IF_D
0	1	10	1	5
1	1	4	1	6
2	1	10	147	800
3	1	6	441	3200

Sample calculations for modes 0-3 with 101 N_taps are shown below:







Mono:

# of taps	Modes/ Analyzation:	Accumulations & multiplications per audio sample	nonlinear operations atan2/sin/cos per audio sample	Run-time measurements	
				Raspberry pi	localhost
101	Mode 0	1010	0		
	Mode 1	1212	0		
	Mode 2	1100	0		
	Mode 3	733	0		
13	Mode 0	130	0		
	Mode 1	156	0		
	Mode 2	141	0		
	Mode 3	188	0		
301	Mode 0	3010	0		
	Mode 1	3612	0		
	Mode 2	3275	0		
	Mode 3	4368	0		

The number of accumulations per audio sample for stereo modes are the same as mono since it uses the same block size and the same number of taps.

Due to time constraints, although the runtimes were unable to be measured, inferences can be made about the runtimes. As the number of multiplications per sample is increased, the longer the runtimes. The non-linear operations are more complex than multiplication and will therefore take more time than multiplication. The raspberry pi is simpler in terms of computer complexity compared to a local host desktop or laptop and the runtime for a raspberry pi would therefore be slower than the local host. As the number of taps increases, the runtime should increase and vice versa and the number of taps decreases.

Proposal for Improvement

Extra Features

A feature that would benefit the users of our system is adding a user interface. A GUI could be created with buttons, sliders, and a file selection feature. Button would eliminate the possibility of the users incorrectly entering commands. It would also make it easier to select a mode and which audio path to choose. Sliders would make it easier for the user to choose which radio frequency to listen to. A file selection feature would easily allow users to drag and drop an audio file to listen to.

Another feature to add to better our productivity when validating/improving it could be to write test cases for each function or block of code from the signal flow graph. Furthermore, we can automate the testing process to run tests frequently since running the code takes a long time to process. This would make testing more efficient and easier to maintain, so if any issue or bug arises it is discovered more efficiently and in a timely manner.

Both features are technically feasible to add because GUIs are common to almost every user application and there are software tools and libraries that help create this. Also, having test cases and tools are common in software industries to test their own software modules in units.

Better Performance

The runtime performance of this system could be improved by researching how C++ processes data and how to optimize the way data should be passed and how code should be written. An example of this was found while developing the faster convolution function. It was found that converting integers to float is faster than converting unsigned integers to float. This was experimentally seen to help improve computation time.

Project Activity

Table 3: The detailed work breakdown items of each member.

Members	**Mono**	**Stereo**	**RDS**	**Multi-threading**
Sara Armanazi	-Added in lab3 files (low-pass-filter + convolve)	- Added the fmPll.py function and converted it to C++ - Wrote the BPF function in py and cpp	- Initialized all-pass filter function	- Initialized multi-threading functions
Joshua Chan	-debugging and testing -wrote, improved and completed the convolution algorithm	-rigorous debugging and testing in python and cpp -added zero padding	-edited/ changed all-pass filter function	-debugging and testing

	-wrote the overall input/output functions			
Jashanjyot Randhawa	-help understand the theory behind the mono path	-wrote mixer function -help convert the code from python to c++ -debugging and testing	-some research behind the all-pass filter	-simple peer editing to find any errors in code
Clara Abadir	- brainstormed up and down sample values for each mode -Translated efficient convolution function (fastConvBlock) from python to C++	-wrote StereoCombiner function -edited some stereo functions -debugging	-initialized RDS file with the required functions in order	-implemented multi-threading into code in the main project file -debugging

Table 4: The progress of Sara Armanazi over the project duration.

Progress	**Sara Armanazi (400324304)**
Week 1 (Feb. 14)	Reviewed labs 1-3 codes, read project document.
Week 2 (Feb. 21)	Added in lab 3 model files, and the LPF and Convolve functions.
Week 3 (Feb. 28)	Implemented the BPF function in python and C++.
Week 4 (March 7)	Added the fmPll.py and converted it to C++.
Week 5 (March 14)	Helped with troubleshooting stereo, converted from C++ to python.
Week 6 (March 21)	Initialized all-pass filter function.
Week 7 (March 28)	Initialized multi-threading.
Week 8 (April 4)	Project report, cross examination, presentation

Table 5: The progress of Clara Abadir over the project duration.

Progress	**Clara Abadir (400326366)**
Week 1 (Feb. 14)	Project released midway through the week.
Week 2 (Feb. 21)	Midterm recess.
Week 3 (Feb. 28)	Started reading the project document and going over previous labs code.
Week 4 (March 7)	Reviewing concepts for efficient convolution function and up and down sampling.
Week 5 (March 14)	Translated fastConvBlock to C++ for mono processing.
Week 6 (March 21)	Worked on stereo processing functions and implementing signal-flow graph (combiner function, and correct steps for processing). Roughly started RDS processing in C++ and Python.
Week 7 (March 28)	Debugging stereo and implemented multi-threading to work with RF, mono, and stereo. Minor updates to RDS processing.
Week 8 (April 4)	Project report, cross examination, presentation

Table 6: The progress of Jashanjyot Randhawa over the project duration.

Progress	**Jashanjyot Randhawa**
Week 1 (Feb. 14)	Met with 2 new group members

Week 2 (Feb. 21)	Went over labs 1-3
Week 3 (Feb. 28)	Read over project documentation
Week 4 (March 7)	Helped understand theory behind mono audio
Week 5 (March 14)	Wrote mixer function and help convert the stereo code from python to c++
Week 6 (March 21)	Debugging and testing. Did some research on what an all-pass filter is
Week 7 (March 28)	Revision of code that was already completed
Week 8 (April 4)	Project report, cross examination, presentation

Table 7: The progress of Joshua Chan over the project duration.

Progress	**Joshua Chan**
Week 1 (Feb. 14)	
Week 2 (Feb. 21)	
Week 3 (Feb. 28)	
Week 4 (March 7)	Started to read and understand the project documentation for mono processing. Set up the raspberry pi at home to read radio waves from the dongle.
Week 5 (March 14)	Started to work on the more efficient convolution algorithm. Partially completed the convolution algorithm. Wrote the algorithm in python. Converted the algorithm to block processing in python. Created a class in C++ which held all the group specific constraints and to automate the up-sampling and down-sampling factor. Also implemented classes for mono and the RF processing part in hopes of simplifying the code.
Week 6 (March 21)	Completed converting the more efficient convolution function in C++. Had problems with the other modes but figured it out. Created the input and output audio functions. Lots of debugging.
Week 7 (March 28)	Completed the stereo processing in python. Rigourously debugged the C++ stereo function by going through and comparing every single value on the C++ and python implementation using gdb and pdb. Eventually, figured it out. Helped with multi-threading debugging and implementation.
Week 8 (April 4)	Project report, cross examination, presentation

Conclusion

Overall, this project taught us a lot about digital signal processing concepts and gave us hands-on experience of creating it in actual software. The lectures were directly related to the labs and projects and did a good job in helping us understand the concepts required to complete the project. We learned how to use signal-flow graphs to model radio applications in software and how useful they are to organize the required steps and blocks to completing a block of processing. Furthermore, something new some of us learned is multi-threading which allows different parts of the program to run simultaneously.

References

[1] S. Chen and K. Cheshmi, “COE3DY4 Project Real-time SDR for mono/stereo FM and RDS.” Hamilton, 2023.

[2] S. Chen and K. Cheshmi, “3DY4 Project - Custom Settings for Group 18.” Hamilton, 2023.

[3] “Software defined radio,” *Software Defined Radio - an overview | ScienceDirect Topics*. [Online]. Available: https://www.sciencedirect.com/topics/engineering/software-defined-radio. [Accessed: 09-Apr-2023].

[4] R. Awati and J. Burke, “What is a phase-locked loop (PLL)?,” *Networking*, 14-Jun-2021. [Online]. Available: https://www.techtarget.com/searchnetworking/definition/phase-locked-loop. [Accessed: 09-Apr-2023].

[5] S. Chen, “DSP Resampling,” in *3DY4 Lecture, 27-Feb-2023.*

[6] S. Chen and K. Cheshmi, “COE3DY4 Lab #1 DSP Primitives in Python.” Hamilton, 2023.

[7] S. Chen and K. Cheshmi, “COE3DY4 Lab #3 From RF Data to Mono Audio.” Hamilton, 2023.

[8] “DSBSC demodulators,” *Tutorials Point*. [Online]. Available: https://www.tutorialspoint.com/analog_communication/analog_communication_dsbsc_demodulators.htm. [Accessed: 09-Apr-2023].