COE3DY4 Project Report
Group 47
Alexander Radikov | Muhammed Khan | Bilal Yusuf | Ebrahim Simmons
radikova@mcmaster.ca | khanm244@mcmaster.ca
yusufb1@mcmaster.ca | simmoe1@mcmaster.ca
4/4/2022

1: Introduction

This project is concerned with developing and testing a software implementation for a software-defined radio (SDR) system. This system receives real-time frequency modulated (FM) mono and stereo audio data and processes it based on different modes of operation which change our sampling frequency and parameters associated with those values. The FM data is received by the RF dongle on the Raspberry Pi 4, where it is then processed and outputted onto the Raspberry Pi itself. The SDR was created using C++, with python models for verification and testing of logic.

2: Project Overview

This project's goal was to implement a software-defined radio (SDR) system for real-time reception of frequency modulated (FM) audio in mono, stereo, and radio data system (RDS) protocol. Unfortunately, our group did not get to reach the RDS stage, but we did accomplish mono and attempted stereo. The front end was accomplished by using RF dongles for the real-time application of FM channels on Raspberry Pi 4 on which the values were outputted. Our SDR strives to retrieve data from FM channels using frequency demodulation to extract the broadcast signal. The mono subchannels have a frequency between 0-15 kHz with the stereo subchannels having a frequency between 23-53 kHz while the RDS has a subchannel frequency of 54-60 kHz. These three subchannels were the primary focus of the project.

The RF front-end block uses a low-pass filter to extract the FM channel, giving us only the frequencies we want and removing higher frequencies that are not needed. The low-pass filter is followed by a decimation of a value of 10 to get the intermediate frequency (IF). The input data that is fed into the RF depends on the mode. 2400 Ksamples/sec for mode 0, 1152 Ksamples/sec for mode 1, 2400 Ksamples/sec for mode 2, and 960 Ksamples/sec for mode 3. Unless specified by the user the default mode is set to mode 0.

The mono path uses the IF values from RF front-end and these values are sent through a low-pass filter then the values are expanded and then decimated by various values depending on the mode to give an Audio FS output of 48 Ksamples/sec. The block then outputs the data in 16-bit signed integers which are then read by an audio player.

The stereo path uses the IF values from the RF front-end and splits into two paths: stereo recovery extraction and stereo channel extraction. These paths have band-pass filters to extract the necessary details and are paired along with a phase-locked loop (PLL) that has a numerically controlled oscillator integrated into the function. The purpose of this is to clean the sinusoidal signal and reduce noise. These two signals are then mixed and combined with the mono audio FS output to produce left and right audio channels of 48 Ksamples/sec.

3: Implementation Details

Building Blocks

The work that had been done throughout the labs was essential to succeed in this project. One key building block was the discrete Fourier transformation. It transforms the signals from the time to frequency domain. The formula which is $x(t)^{-j*2*pi*f*t}$ is integrated from positive infinity to negative infinity. The way this was implemented was through two for loops which are nested. The inner for loops go through each x value while the outer for loop goes through each frequency bin.

Adding on another fundamental building block that was necessary to succeed in the project was the implementation of low-pass filters. A low-pass filter eliminates the frequency components of the input signal that are above a cut-off frequency. Multiplying the frequency response with a rectangular window with the height defining the filter gain and the width defining the filters pass band gives this result. Using sinc functions we can generate impulse responses and filter coefficients which help create these low-pass filters.

Finite Impulse Response (FIR) filters perform convolution between the signal being filtered and the filter coefficients. This is implemented by two nested for loops with the outer loop computing the phase and fix iterating from 0 to the size of y and the inner loop iterating from the phase to the size of h. Block processing is another technique used to process data in real time. In order to not run into problems we implement state saving, adding the previous state to the starting index of the current block.

RF front-end

In order to do any calculation first we need to obtain the data stream file, and in order to do that we use readstdinBlockData to read in the data. Next, we needed to split IQ data into two separate vectors which we did using the function splitIQ. splitIQ is a simple function, it sets the size of two vectors I and Q and then using a for loop statement iterates for size of inData assigning the I values to vector I, and Q values to vector Q. After the data is split we use impulseResponseLPF which calculates the impulse response h based on the sinc function. This function allocates memory for the impulse response, the formula is from python code that was adapted from the labs. Then we run two instances of convolution calling convolveOLD function twice, once for I samples and once for Q samples. We have state saving implemented in our convolveOLD function as well, storing the saved state in one vector and storing the filtered data in another. After convolution we run two instances of downsample function, one for I samples and one for Q samples. We decimate by a value of 10 to get the IF Fs values for whichever mode we are running. The downsample function utilizes a for loop which starts at index zero and is incremented by the decimation factor, performing the calculation on the values of multiples of 10 since that is what we are down sampling by. We then demodulate the data using fmDemodulation which was adapted from python.

When working on rf front end we had a lot of small issues like missing headers, sources, and such. We found that initializing our variables in a more organized manner was helpful, and it was easier to visualize why certain numbers were where they are. There was an issue we had that was helped solve by a TA in this block. When we were setting parameters based on our mode of operation we were redeclaring our value of Fs numerous times, which kept resetting the value to 0. Declaring Fs

once at the beginning solved it, and looking back the issue would have been a lot easier to determine had our code been more organized.

Mono

In the same manner as the RF Front End, depending on the mode that the Software Defined Radio is in, we set the variables as given in our constraints document. In modes 1 and 2, data is passed through the low pass filter by using functions impulseResponseLPF and convolveOLD. The next step is downsampling using our function downsample. The decimation values vary from mode to mode, with it being 10 in modes 0 and 2, 4 in mode 1, and 3 in mode 3. In modes 2 and 3, the main difference between the previous modes is in the fractional resampler. Using a calculation which is the ratio between the output sample rate and the GCD of the input and output sample rates we calculated the upscale factor for each of the modes. The upscale factor is shown in our code as the value of audio_up. For mode 0 and 1 it is 1 because there is no upsampling, whilst for mode 1 and 2 the values are 147 and 441 respectively. Because there needs to be upsampling and then downsampling we also implemented a new convolution method which does them both. This function is called convolveFIR, which downsamples and then upsamples based on the fast implementation. The mono output is stored in variable monoOut. When outputting for mono we initialize the vector with 0s and compute only the samples we need. After processing, you can see the vector padded with 0s, but they are inherent.

One issue we had to debug was when testing our mono audio output it did not sound anything close to what it should be, the output was scratching noise. We decided to use std::cerr command to see how many output values we had. We were outputting a lot less values than we should have been. We had to examine all of our functions in mono to determine the root of this problem. After going through each function we found out that in our convolveOLD we were indexing incorrectly in our outer for loop, instead of indexing from zero to size of x we were indexing to size of y instead. After fixing this bug we started to get the correct number of outputs and our audio started to match a lot more.

Stereo

The first step in stereo was reading the raw IQ data from the recorded file. Then we calculate the coefficients for the front-end low pass filter and the coefficients for the filter to extract mono audio using signal.firwin. We use signal.firwin because we know that the firwin implementation is correct and we are using the python model to ensure our implementation is correct. We then determine the impulse response for a bandpass filter for stereo carrier recovery and stereo extraction channel. This is done by using the pcoeffBPIR function which was adapted from project documents and lecture slides. The values of these coefficients are passed into our filter, where we extract the pilot tone audio and stereo channel audio. The bandpass filter with cutoff bands at 18.5kHz and 19.5kHz extracts the channel carrier, while the bandpass filter with cutoff bands at 22kHz and 54kHz extracts the stereo sub-channel. The next step we take is, we declare stereo_mixed an additional audio buffer that is to be later used to hold the output of stereo processing. Our fm demodulator in stereo uses a modified version of fm demod arctan (called secondaryFmDemodArctan). We then called the fmPll function. The PLL is used to give a clean output for the signal, adapted to the frequency/phase of the carrier. When we pass the data in blocks, state saving is used, so the values in the PLL function are carried over through each block. We complete stereo extraction and recovery by using lfilter with our audio

data collected previously that was stored in audio_coeffRec and audio_coeffEtr. In the mixer section of stereo processing, we use pointwise multiplication of the stereo channel extract and PLL output (ncoOut) in order to get our value stereo_mixed. Afterwards, using lfilter, we get filtered data for both the stereo and mono audio. We then downsample the audio data, with our decimation value being audio_decim = 5. The left audio block is calculated by the average of the sum of the mono and stereo audio blocks, while the right audio block is calculated by the difference. The final left and right audio data is then stacked using the function np.vstack() and transposed to give us our final audio output. We then write the output to the file fmStereo.wav

When we were implementing stereo processing in python, there was a notable issue when we took a look at our NCO output graphs. We noticed that the beginning and end of the blocks weren't continuous. When we tried to check this, we outputted the values going in and out of PLL, and saw that our NCO output value was the same even in different blocks. We ended up finding that the trigOffset value we have in our PLL function was not being updated, something we thought was done already. The solution to this was just to add the pilot length to trigOffset.

4: Analysis and Measurements

Multiplications and Accumulations for mode 0

For mode 0, our block size was set to 102,400. We wanted our block size that is passed to fWrite and the output function to be 1024 samples, this is because this allows us to cleanly decimate, downsample, and upsample without running into a non-integer sample number. The in-phase and quadrature components (I and Q samples) are then split into equal vectors of size 51,200. Each component is passed through a low-pass filter with 101 filter taps and a decimation value of 10. This gives us 2((102400 * 101)/(2 * 10)) = 1,034,240 total multiplications and accumulations. After demodulation is performed, the size of our vector is (102,400/2/10) = 5120.

For the mono path, the total number of multiplications and accumulations comes out to be (5120 * 101)/5 = 103,424. Meanwhile, in the stereo path, the demodulated data is passed through 2 bandpass filters of 101 taps each and a low-pass filter (101 taps, decim of 5) after mixing and moving onto mono path calculations. Therefore, the total number of multiplications and accumulations in the stereo audio path for this mode was 2*(5120 * 101) + (5120 * 101)/5 = 1,137,664. At the end of a block, a total of 102400/2/10/5 = 1024 audio samples are produced which matches our expected value.

The stereo path requires about 10 times as many total multiplications and accumulations as the mono path, which when compared to the timing analysis we can see that the combined runtimes for the stereo path are 10 times larger than the mono path. The same pattern is present in mode 1 as well.

Multiplications and Accumulations for Mode 1

For mode 1, our block size was set to be 49,152. Again basing this number upon the desired 1024 output samples as well as ensuring that the block size can be evenly decimated and upsampled without decimal values. The I and Q components are split from this original data stream into equal vectors of size 24,576. Each component is passed through a low-pass filter (101 taps, decim of 4). This

gives us 2((49152 * 101)/(2 * 4)) = 1,241,088 total multiplications and accumulations. After demodulation is performed, the size of our vector is (49152/2/4) = 6144.

For the mono path, the total number of multiplications and accumulations comes out as (6144 *101)/6 = 103,424, which is the same as mode 0. Meanwhile, for stereo, the total number of multiplications and accumulations for this mode was 2*(6144 * 101) + (6144 * 101)/6 = 1,344,512 using the same logic as for the previous mode as we are not upsampling yet for these modes. At the end of the block, a total of 49152/2/4/6 = 1024 audio samples are produced which again matches the expected output.

Multiplications and Accumulations for Mode 2

For mode 2, our block size was set to be 112,000. We base this number upon a desired output sample size of 1029 (the lowest common multiple of both our decimation factors) this ensures our upsampling and downsampling will always produce clean numbers. The I and Q samples are split from this data stream into equal vectors of size 56,000. Each component is passed through a low-pass filter (101 taps, decim of 10). This gives us a total of 2((112000 * 101)/(2 * 10)) = 1,131,200 total multiplications and accumulations. After demodulation is performed, the size of our vector is (112000/2/10) = 5600.

For the mono path, the FM demodulated data was sent through our low-pass filter that included a resampler, the parameters passed to this low-pass filter were 14847 taps (101 * 147), upsample factor of 147, and a decimation factor of 800. This means that for our mono path in this mode, the total number of multiplications and accumulations is (5600 * 14847)/800 = 103,929. Meanwhile, for stereo, the FM demodulated data is run through 2 bandpass filters (101 taps each) as well as through a low-pass filter with resampling (14847 taps, decim of 800, and a upsample factor of 147) after mixing the stereo samples. Therefore, the total number of multiplications and accumulations for this mode was 2*(5600 * 101) + (5600 * 14847)/800 = 1,235,129. At the end of the block, a total of 112,000/2/10/(800/147) = 1029 samples are computed.

The stereo path requires about 10 times as many total multiplications and accumulations as the mono path, however the runtimes for stereo are only found to be 2 times larger than mono. This is most likely due to the O3 flag performing optimizations to the compilation of the code.

Multiplications and Accumulations for Mode 3

For mode 3, our block size was set to be 57,600. We base this number upon a desired output sample size of 1323 (the lowest common multiple of both our decimation factors) this ensures our upsampling and downsampling will always produce clean numbers. The I and Q samples are split from this data stream into equal vectors of size 28,800. Each component is passed through a low-pass filter (101 taps, decim of 3). This gives us a total of 2((57, 600 * 101)/(2 * 3)) = 1,939,200 total multiplications and accumulations. After demodulation is performed, the size of our vector is (57600/2/3) = 9600.

For the mono path, the FM demodulated data was sent through our low-pass filter that included a resampler, the parameters passed to this low-pass filter were 44,541 taps (101 * 441), upsample factor of 441, and a decimation factor of 3200. This means that for our mono path in this mode, the

total number of multiplications and accumulations is (9600 * 44,541)/3200 = 133,623. Meanwhile, for stereo, the FM demodulated data is run through 2 bandpass filters (101 taps each) as well as through a low-pass filter with resampling (44,541 taps, decim of 3200, and a upsample factor of 441) after mixing the stereo samples. Therefore, the total number of multiplications and accumulations for this mode was 2*(9600 * 101) + (9600 * 44,541)/3200 = 2,072,823. At the end of the block, a total of 57600/2/3/(3200/441) = 1323 samples are computed.

Multiplications and Accumulations for PLL

The atan2, cos and sin functions are only called inside of the PLL function, which in turn is in the stereo path. The PLL function is run once for every block that is read, and inside the PLL we loop through by the size of the demodulated data vector size. In the case for mode 0, this would be 5120. Also for every loop of our PLL we use one atan2 function, one sin function, and 2 cos functions. So in total at the end of mode 0 the PLL will run the atan2 function 5120 times, the sin function 5120 times and the cos function 2*5120 = 10,240 times. These values vary based on mode but remain purely based on the size of our FM demodulated data vector size so for each other mode the number of times we run would be 5120, 6144, 5600, 9600 for modes 0-3 respectively. Also since these are all built in functions inherent to the c++ library we cannot directly measure the number of multiplications and accumulations each function performs.

	101 Taps				13 Taps	301 Taps
	Mode 0	Mode 1	Mode 2	Mode 3		
RF Front End Processing						
RF LPF (I and Q)	3129	3551	3792	5930	1241	8629
RF Decimation (I and Q)	208	218	255	152	159	290
RF Demodulation	54	53	61	226	51	68
Mono Processing						
Mono LPF	330	389	2907	6554	126	895
Stereo Processing						
Stereo Carrier BPF	1564	1486	1645	2864	538	4288
Stereo Extraction BPF	1491	1856	1638	2807	534	4287
Stereo PLL	1377	1167	1352	2348	1246	1192
Stereo Mixing	20	77	18	32	18	14
Stereo LPF	321	318	2905	6323	167	889
Stereo Combination	6	6	9	11	9	5

All runtimes are in microseconds.

Changes in Runtimes based on Number of Taps

Since the relationship between the number of taps and the number of operations is linear, it is expected that for the RF Front End the number of operations would decrease by a factor of ~10 when 13 taps are used, and increase by a factor of 3 when 301 taps are used. This can be observed in the runtimes in the table above, and the data reveals that the relationship does hold (with some small variance). The run times for 301 taps scale up by about 3, which is as expected. For 13 taps, however, the runtime is not lowered by a factor of 10, but rather a factor of roughly 3. This is most likely due to the heavy load on the output stream slowing down the overall program execution, since there is a lot of data being outputted into the stream for each block. Overall, however, the results change as expected.

A similar change is observed in the Mono Path filters as well. Since the relationship is linear between taps and the number of operations, the runtimes increase by a factor of 3 for 301 taps and shrink by a factor of 10 for 13 taps. However, similar to the issue stated earlier, the runtime for 13 taps isn't quite 10 times smaller, but it's close enough to be an acceptable result when output stream overload is taken into account. Similarly, for stereo, the runtimes change in the same manner for Stereo Carrier BPF and Stereo Extraction BPF.

In terms of audio quality, there is a noticeable loss in audio quality when switching from 101 taps to 13. There is a ripping/whirring noise in the background, and the audio sounds a bit choppy. Overall, the audio is listenable though it is far from clear. When switching from 101 taps to 301 there is a noticeable increase in audio quality. Everything sounds a lot clearer and there is no strange background noise. Music sounds a lot clearer and instruments are a bit more distinguishable. It is worth noting that the audio was heard through monitor speakers which have relatively poor quality and that the differences in audio clarity may be a lot more obvious through a pair of headphones.

5: Proposal for Improvement

Something that could greatly improve the user experience and enhance the learning process for this project would be to implement some sort of UI for using the radio. As of now, for a user to use the SDR they would have to type in a (to them) complex command into a linux terminal to tune into a radio station and hear the audio output. However, if we were to construct a UI for the system using OpenGL or any other graphics library to interface the radio, not only would it enhance the user experience but it would allow us to push our understanding of C++ and learn about its top-tier graphics and OOP capabilities. It would also make debugging a lot easier, which would help our productivity.

Another idea that we could've incorporated to our project to improve its current state would be to further decrease the runtime of our software. The first thing we could have done was to improve the performance of our PLL by using the arctan estimation function instead of the C++ default function. As we know from previous labs using the arctan estimation function reduces runtime substantially as the estimations take much less processing power to compute as opposed to calculating the exact values for each iteration. Another idea was to incorporate our convolution and downsampling functions for modes 0 and 1 into one function similar to how we did it for modes 2 and 3. As we have it now we perform the entire impulse response and convolution functions then pass those parameters to our downsampling function to be properly formatted for the next step in the software, if instead, we combined these functions into one to both convolve the data as well as downsample by our specified

rate we could've saved iterations over the same data set which would've helped our system perform more operations at once. These ideas would help our project run more efficiently and would allow the software to run on lower spec machines as our complexity for these functions would be greatly reduced. These changes could theoretically change our complexity for these functions from O(N^2) to O(N) since we are now only calculating the samples we need and hence running through the data only once.

Additionally, we could increase our efficiency working as a group would be to make use of git branches and merges. Many times throughout the project, we would have issues with local repositories not matching the main repository because they didn't pull before editing a block of code, or someone else pushed their code to the main while someone else was editing. This caused quite a backlog of pushes that needed to be done before the main was updated with everyone's work. From personal experience using branches and merges for a real-life project, these tools could've helped us greatly to streamline our project work as everyone could have their own local branch on their machine and push their work whenever they wanted and whenever we need to debug we can simply merge the branches and get to testing.

6: Project Activity

Alex Radikov	**Week One:** Had a group meeting on who will handle the RaspPi/RF dongle for the project. **Week Two:** Reviewed lectures and project doc to develop an understanding of the project. **Week Three:** Went over lab feedback to see what needed to be changed to the code for the project. **Week Four:** Worked with Muhammad, Bilal, and Ebrahim on establishing working baseline code. Implemented and helped adapt RF front-end block from python to C++. Added downsample and upsample function to C++. Fixed memory allocation issues in fmDemodulation function related to the output vector. **Week Five:** Fixed several memory allocation issues with RF Front End and got it running. Fixed a bug with the downsample function parameters. Added the splitIQ function. Fixed improper indexing issues with fmDemodulation. Began implementing mono functionality as per the project documentation, and began producing an audio output. Added functionality for modes. Got modes 0 and 1 working by fixing the improper block size and taps. Spent several hours trying to debug background noise in audio output (see the issue with convolution function next week). **Week Six:** Adapted PLL function from python to c++. Debugged error in PLL concerning wrong integrator values not accumulating correctly. Worked on resampling convolution function needed for modes 2 and 3. Debugged issue with convolution function state saving not starting at phase. Cleaning up code and adding comments Tested code for correctness on VM listening to audio data and using GDB. Began implementing stereo processing in C++ based on the Python model. **Week Seven:** Worked on project report and sections: Introduction, Analysis of measurements, Proposal for improvement.
Muhammad Khan	**Week One:** Had a group meeting to discuss project strategy and figure out logistics. **Week Two:** Began developing a strong understanding of the theory with the use of project documentation and lectures. **Week Three**: Implemented the key functions from labs 2 and 3 into the project. Fixed

	an issue with the Virtual Machine (performance issues related to processor and memory allocation for AMD). **Week Four:** Established a working baseline for the project code by fixing missing headers, missing sources, and other small issues. Implemented the RF Front end as outlined by the Python model (though it still had some bugs) Added downsample function in C++. Fixed memory allocation issues in fmDemodulation function related to the output vector. **Week Five:** Fixed several memory allocation issues with RF Front End and got it running. Fixed a bug with the downsample function parameters. Added the splitIQ function. Fixed improper indexing issues with fmDemodulation. Verified correctness of RF front end by comparing against plots from the python model. Began implementing mono functionality as per the project documentation, and began producing an audio output. Used sample .raw files from lab 2 and the wavio.py file to test the audio output. Spent several hours trying to locate the source of the background noise in audio output. Refactored the code completely to work with linux pipes and aplay rather than wavio.py. Got modes 0 and 1 working by fixing the improper block size and taps. Met with Dr. Nicolici over the weeked to discuss the noise issue. **Week Six:** Implemented resampler for the LPF for mono modes 2 and 3 as outlined by Dr. Nicolici. Fixed improper state saving issue for the convolution function. Setup RaspberryPi and began live testing for all mono modes. Began implementing stereo processing in C++ based on the Python model (added filters, mixing, combining, and main functionality). Fixed key issues with memory allocation for stereo output vector. Spent several hours trying to fix the noise issue with the stereo. Heavily refactored the code, added comments and cleaned up the project directory. **Week Seven:** Worked on project report sections: Measurement and Analysis, Proposal for Improvement.
Bilal Yusuf	**Week One:** Had a group meeting on who will handle the RaspPi/RF dongle for the project. **Week Two:** Read over project documents and went over labs 1-3 to get ready for project. Watched lecture recordings I had missed to catch up for beginning of project. **Week Three:** Converted fmDemodulation python to computationally friendly C++. Created notes on how to implement RF front-end and Mono, calculating values for expander and decimation values **Week Four:** Worked with Muhammad, Alex and Ebrahim on establishing working baseline code. Implemented and helped adapt RF front-end block from python to C++. helped with creation of downsample and upsample functions in C++. **Week Five:** Worked with Muhammad Alex and Ebrahim fixing several memory allocation issues with RF Front End and got it running. Added the splitIQ function with Muhammad and Alex. Verified correctness of RF front end by comparing against plots from the python model. Began implementing mono functionality as per the project documentation, and began producing an audio output. Worked on bugs with convoleOLD function where indexing was going until size of y instead of size of x (explained in detail on last paragraph of mono in implementation details) **Week Six:** Broke off into two groups, Ebrahim and I started and finished full implementation of stereo in python model. I added state saving to fmPLL.py file. Created a fmSupportLib. porting functions over from labs like estimatePSD. Implemented mixer and digital filtering along with downsampling. Added stereo combiner with Ebrahim. Helped with debugging in c++ stereo giving suggestions of bugs ebrahim and I encountered in python version of stereo. **Week Seven:** Worked on project report sections: Project Overview, Implementation Details (Building Blocks, RFfront-end, Mono, Stereo), Conclusion.

Ebrahim Simmons	**Week One:** Had group meeting on who will handle RaspPi/RF dongle for project. **Week Two:** Read the project document and reviewed labs 1-3 to get ready for project. **Week Three:** Converted fmDemodulation python to computationally friendly C++ code. Wrote code to switch between modes. **Week Four:** Worked with Muhammad, Alex and Bilal on establishing working baseline code. Implemented and helped adapt RF front-end block from python to C++. Worked on downsampling/upsampling functions during meetings. Tested pushed code **Week Five:** met with TAs for various issues and to help debug (last paragraph in rf-front end subsection for example). verified correctness of rf front end by comparing against plots from python model. began implementation of mono by reading over project doc again and doing calculations needed. Worked on convolution function. Met with TAs to find issues with incorrect audio output (gave advice on issues with cutArray and how to use backtrace to debug). **Week Six:** Started and finished full implementation of stereo model in python (worked primarily in fmStereo.py implementing stereo carrier recovery and channel extraction). Added stereo combiner with Bilal. Debugged issue in PLL with trigOffset not updating correctly. Found errors in coefficients for bandpass filter pcoeffBPIR. Using python debugging methods, helped with debugging in c++ stereo **Week Seven:** Worked on project report sections: Project Overview, Implementation Details (Building Blocks, RFfront-end, Stereo).

7: Conclusion

After working on the project over the course of the last three months we have gained valuable knowledge and experience both technical and non-technical. We got the opportunity to see how a project interfaced with the real world of this scale is implemented and worked on from beginning to end. The experience of working in a team environment is similar to what we will experience in the outside world in job environments. The technical skills in the process that we all developed/learned include digital filtering, signal processing, frequency modulation, coding, debugging, and optimization. Similar to 3DQ5 we once again realize the benefits of breaking down a big project into small milestones/parts. First focusing on RF front-end, moving on to mono, and then stereo. Breaking a project down makes it a lot easier and doable rather than just tackling the whole project head-on. Overall we are grateful to gain this experience from this project along with the skills that we developed.

8: References

The resources that we used for this project were the class notes, project specification along with the C++ and python standard libraries.