3DY4 Project Report

Group 11

Abdelmoniem Hassan - 400248003 - hassaa73
Muaz Akhtar - 400249273 - akhtam17
Hakam Atassi - 400294803 - Atassih
Saarah Ahmed - 400304444 - ahmes98

Introduction

The objective of this project was to design and implement a software defined radio using basic DSP blocks. To meet the project's objectives, two primary challenges were addressed: the first was dealing with the streaming and real time processing of large amounts of data. The second being the design and verification of the DSP blocks used to convert collected IQ samples into audio data.

In terms of hardware, the project was restricted to a Realtek RTL2832U chipset to collect the IQ samples, and a Raspberry Pi 4 (4 cores, 4 threads @ 1.5GHz). Using this hardware, we were required to stream a real-time audio signal through by piping IQ measurements from the provided antenna.

Project Overview

The overall project can be divided into a few isolated segments, each imposing its own limitations on the performance and architecture of the SDR.

The first component of the project is the RF Front End. The responsibility of the RF Front End code is to convert the IQ samples into a radio channel signal through low pass filtering, demodulating, and decemating. This radio channel is then used by the remainder of the SDR to produce the various audio channels (Mono, stereo, etc…). This portion of the SDR introduced the first independent thread, as all following components of the system use the result of the computations performed in this stage.

The next component of the project is the mono audio channel extraction. This part of the project involved converting the FM demodulated channel output to an audible signal through low pass filtering at the baseband frequency [0-15Hz]. Due to the nature of the input signal being in blocks, this process required additional state saving logic to aid in the transition from one block to the next.

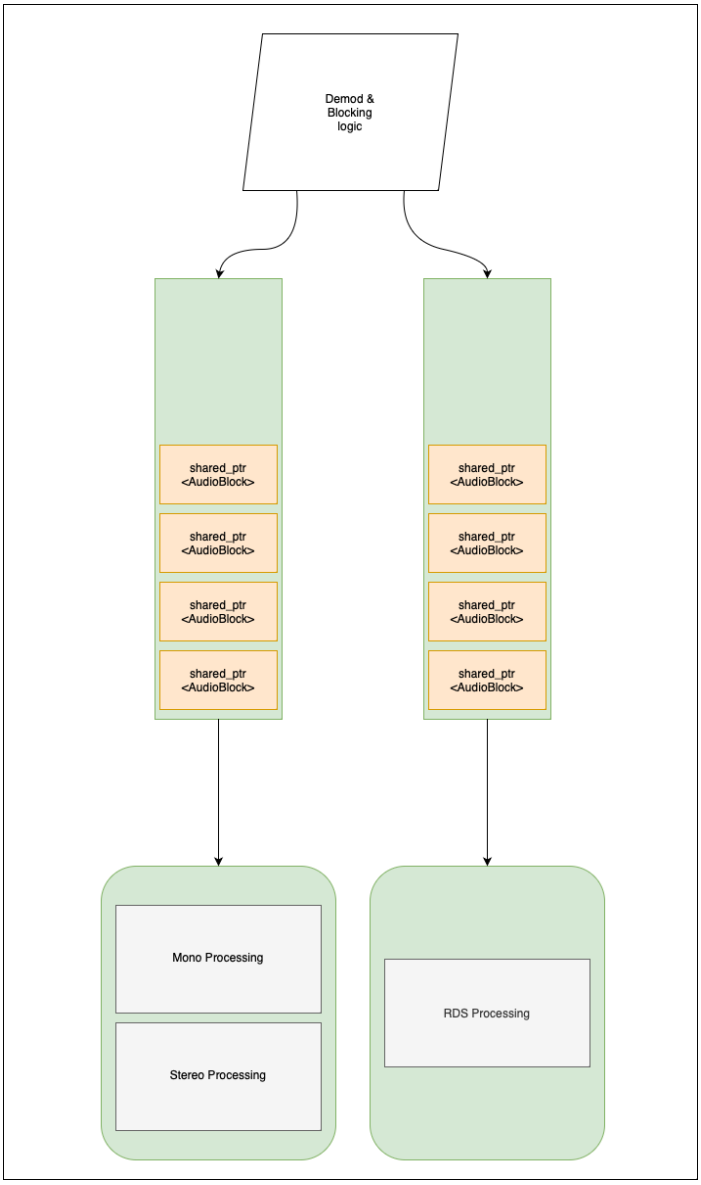
The stereo channel is used to add texture to the output audio by summing the audio signals of the mono and stereo channels. However, the key difference between stereo and mono is that stereo is not located in the baseband. As a result, a simple channel extraction through filtering is not sufficient to extract the audio signal. Instead, the isolated channel must be mixed with a phase locked carrier to bring the channel back to the baseband, where it is then filtered for a final time. The sum and difference between mono and stereo constitute the left and right channels respectively.

The RDS stage of the SDR is used to provide channel and song information to the listener. Since the information in this portion of the project is encoded digitally, the data extraction in this stage required extracting digital information from analog signals transmitted in through the radio channel. Since this channel, unlike stereo, was not supplemented with a carrier, it required a carrier recovery phase. Following the carrier recovery, an RPSK signal is extracted through a mix and filter of the original RDS signal. This RPSK signal is then sampled, differentially decoded, checked for errors, and finally, passed through a generator matrix to decode into human readable messages.

Implementation Details

To take advantage of the object oriented features provided by the C++ programming language, our code was separated into a few primary objects that encapsulated many of the functions, features, and requirements of the SDR. The first and most important of these objects is AudioBlock.

AudioBlock encapsulates the stream of data into discrete blocks of fixed size. Each AudioBlock object has the capability of undergoing RF, Mono Channel, Stereo Channel, and RDS processing. Each block is created within the front end thread. It then undergoes the RF processing which extracts the raw IQ data, then filters, resamples, and demodulates it. The blocks are then ready to be consumed by the audio and RDS threads. Since the Stereo mode requires each block to undergo Mono and Stereo processing, each AudioBlock undergoes these calculations on the same thread in sequence. RDS, however, has no dependance on the Mono and Stereo stream. As such, RDS undergoes processing in its own thread.



Since RDS and the Mono Stereo pair are processed on separate threads, their rate of consumption will vary. Since all AudioBlock objects depend on the same demodulated FM data, this variance in processing time introduces a critical issue with the allocation of data; if the Stereo Mono processing thread consumes blocks faster than RDS, RDS will never have any data to process . To overcome this issue, the FM demodulation thread creates a shared pointer of type AudioBlock which manages the lifetime of the object. The object remains in memory as long as there is a reference to the shared pointer in memory. This keeps it allocated until all of the Mono, Stereo, and RDS processing is complete.

Provided here is a structural diagram that depicts the processes of queueing and processing pointers to AudioBlock objects. Note that, although each queue may consume elements at different rates, each process can proceed at its own pace without the AudioBlock data being freed prior to undergoing all required Mono, Stereo, and RDS processing.

Front End

In Python

The model in Python begins by reading the raw data into an array and having it normalized between -1, and 1. The IQ data is then extracted in fixed block sizes, filtered using a low pass filter with a cutoff frequency of 100kHz and downsampled by a fixed factor of 10. The filtered IQ data is then demodulated.

Since the model's main objective was verification, All of the filtering, generation of coefficients was done using the Scipy data processing library. After verifying the overall process, the library functions were replaced by our custom implementations, which are verified against the library functions and that were ready to be ported over to C++.

C++ & Transition

The RF front end thread begins by generating the Low Pass Filter (LPF) coeffecients required for the RF processing, as well as the state vectors that need to be preserved between different blocks.

The thread then begins processing the raw IQ data passed into the standard input stream. The IQ data is then grouped into blocks, of which sizes are determined by the mode. An AudioBlock object is then created with the data set, the IQ samples are then separated, filtered and demodulated using an identical procedure to the python model. The demodulated data is stored as a private member of the object. Shared pointers to these objects are then placed onto separate queues for the audio, and RBDS paths to consume.

The C++ implementation was verified against the Python model. This was done by logging the data after each processing step. A script was then used to compare and plot the data from both implementations, in both the time and frequency domains.

Resampling & Mono

In Python

In order to optimize performance, a function was first developed in python that performs convolution along with the up and down sampling, as opposed to doing these two things separately. This is possible as only the points that are involved in the convolution are considered as not all points are needed. The function takes an input signal x, filter h, upsampling factor du, downsampling factor ds, and a state saving array called state. The function calculates the resampled output signal y by convolving x with h, taking into account the upsampling and downsampling factors, and using the phase variable to determine the starting point for each sample in y. The j variable is used to calculate the index of the input signal x that corresponds to the current sample being calculated and uses the phase value to do so.The function returns the resampled signal y and the updated state for use in subsequent calls to the function.

C++ & Transition

After confirming that the function was working by testing it with various input signals, and comparing the resampled output signals with expected results the function was converted from Python code to C++ code. The main difference in the C++ version of the code is that the input arrays x, h, and state are passed by reference, rather than by value. This allows the function to modify the input arrays directly, rather than creating copies of them in memory. This can improve the efficiency of the function, especially for large input signals and filters. The only other difference in the implementations is the syntax differences in C++.

Stereo

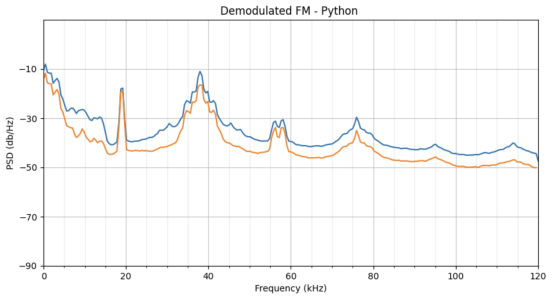
In Python

Although Stereo reuses many of the components that have been verified during the development of the Mono processing stream, Stereo introduces a few new components that have the potential of introducing new, unaddressed issues. The two primary components Stereo introduced are the PLL and the BPF filter generating function.

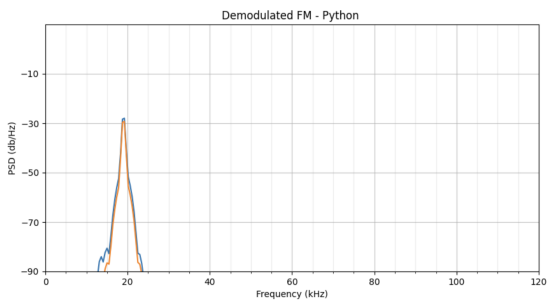
The purpose of the PLL is to lock an incoming sinusoid [19] kHz at a certain frequency and phase to correctly push the Stereo signal back into baseband. Additionally, since the initial Stereo channel is not in the baseband [23-53] kHz, a BPF filter must be designed to correctly extract the Stereo channel.

The initial development of Stereo was completed in a single pass, non-blocking implementation of the SDR. Since the single pass processing of Mono was completed as part of the Labs, much of the groundwork was already laid.

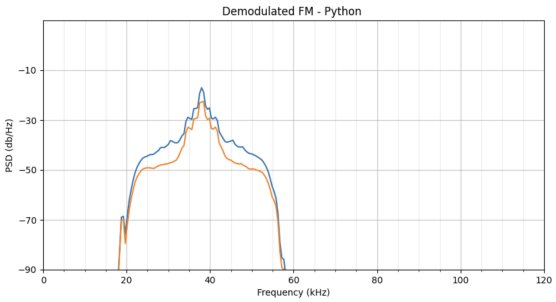
In order to more directly address the logic of how Stereo worked as opposed to being concerned with primitive implementation of filter generation, convolution, etc…, the first iteration of Stereo processing used purely library functions. The first step of this process involved using the Scipy.Firwin() function to generate a filter of 151 taps to extract the Stereo channel. A second filter of cutoffs 18.5-19.5 kHz to extract the Pilot Signal. These Filters were then applied to the FM demodulated signal to extract the Stereo and Pilot signals. The result was the following:



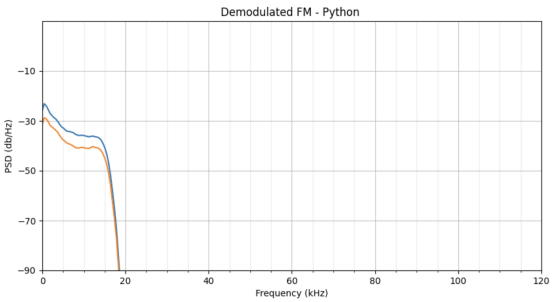
[FM demod]



[Extracted Pilot]

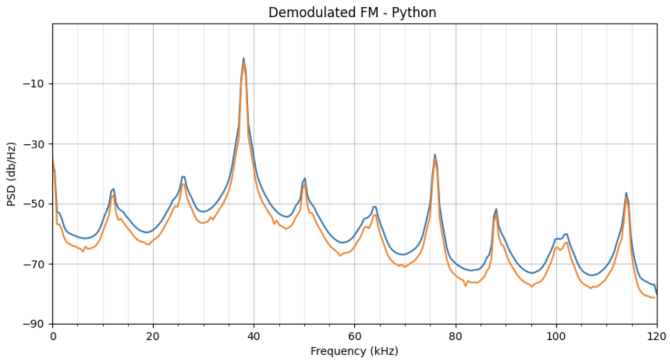


[Extracted Stereo]



[Mixed and Filtered Stereo]

With the signals extracted, the pilot was passed into the PLL with a frequency scaling of 2 such that its frequency is centered around 38kHz the Pilot was then mixed with the extracted stereo to place the Stereo channel in the baseband. To eliminate undesirable artifacts in the frequencies beyond the baseband as a result of the mixing, another convolution was performed using the same filter coefficients as the Mono channel extraction (baseband coeffections).



[Output of Pilot PLL]

The stereo signal is then mixed with the above PLL output. This placed the stereo channel output in the baseband. The baseband stereo is then filtered using the Mono coefficients. To account for the group delay (filter delay) introduced by this filtering process, the following code was added to delay the mono signal before L R channel extraction:

```
## Add delay to mono
delay_amount=int((N_taps-1)/2)
delayed_mono=[0.0]*delay_amount
delayed_mono=np.append(delayed_mono,audio_filt)
```

[Mono Code Delay Snippet]

```
left_channel = (delayed_mono-mixed_stereo_filtered)
right_channel = (delayed_mono+mixed_stereo_filtered)
```

[Channel Separation Snippet]

C++ & Transition

Besides some initial roadblocks encountered with the alignment of the Stereo and Mono channels as a result of the group delay, the transition from Python to C++ was fairly manageable.

The first step was to move any unimplemented functions in Python to C++. The first being the BPF coeffection generating function. This step was straightforward since the verification of the function involved nothing more than simply comparing a sequence of 150 coefficients from the Python code to the C++ code.

Next, a PLL was implemented in C++. Since the C++ code involved blocks but the Python code did not, the simplest way to manage this transition is to simply encapsulate the PLL into a class that calculated the output based on its own member variables. This way, the PLL was not exposed to the blocking of the input stream in any way.

Now, with the PLL and BPF filters implemented in C++, the same exact process was repeated for this part as the Mono processing. However, one very important detail is that Mono was now hardcoded to have a delay of (NTaps-1)/2, to account for the fact that some of the modes require Stereo, which inturn requires a delay. This delay was introduced using a Queue that is initialized with (NTaps-1)/2, zeros and is dequeued by the block size to fetch the (now delayed) Mono data prior to streaming the output.

```
std::queue<float> monoQ;    // QUEUE FOR THE MONO AUDIO DELAY
int delay_amount = 75; //optimal delay value
for(int i=0;i<delay_amount;i++){
    monoQ.push(0.0);    //add some delay to mono
}

//...

for (int i = 0; i < fm_demod.size(); i++)
{
    if (!monoQ.empty())
    {
        fmDemodDelayed[i] = monoQ.front();
        monoQ.pop();
    }
}
```

[Delay implementation in C++]

RDS

In Python

Using our experience transitioning from Stereo in Python to C++, it was evident that developing a single pass implementation of RDS in Python prior to a C++ implementation is key. Knowing this, we created a copy of the provided single pass Mono file provided in one of the labs and extended it to support RDS.

The first step in this process involved extracting the RDS channel using a filter generated by the scipi.Firwin() library [54-60] kHz. Since RDS does not transmit a carrier, the RDS signal itself is transformed to a carrier through a series of carrier extraction steps. These steps involve squaring the RDS signal doubling its frequency from ~57kHz to ~114kHz then passing the signal through a PLL with freq scaling factor of 0.5 to provide a carrier. This carrier signal is then mixed with the delayed (due to filter delay in carrier), extracted RDS signal to return it to the baseband. The mixed signal is then extracted from 0-3kHz using a low pass filter. The result is a raw RDS signal.

The next sequence of steps involve rational resampling, root raised cosine filtering, and clock/data recovery. Relational resampling used a library function and Root Raised Cosine was provided and therefore did not require implementation/verification.

The last phase of RDS involved extracting the data from the RDS signal. This step involves sampling the output signal at the SPS frequency to extract a series of 1s and 0s, based on polarity of the sampled value. This sequence of 1s and 0s is then xored (differentially decoded) and stored in a final resulting vector.

```
def decoding(toBeDecoded):
    i=0
    cdr = []
    while(i<len(toBeDecoded)-1):
        if((toBeDecoded[i]>0 and toBeDecoded[i+1]>0) or (toBeDecoded[i]<0 and toBeDecoded[i+1]<0)):
            cdr.append(0)
            i+=2
            continue
        if (i+1 >= len(toBeDecoded)):
            break
        if(toBeDecoded[i]<toBeDecoded[i+1]):
            cdr.append(0)
            i+=2
        else:
            cdr.append(1)
            i+=2
    post_cdr=[]
    for x in range(len(cdr)-1):
        post_cdr.append(cdr[x] ^ cdr[x+1])
    return post_cdr
```

The last step in decoding the signal involves converting this stream of 1s and 0s into blocks of synchronized data. This is done by computing a product of the last 26 bits of data in the decoded stream with an error correcting matrix **G.** Once each product is calculated, it is compared with the 5 possible block types (A, B, C, C', D). A frame is matched if a sequence of A, B, C || C' , D is detected. This frame match is recorded as the anchor for frames.

C++ & Transition

Similar to the Python model, we began implementing the RDS path by refactoring the already implemented Audio path in C++. First we created a new thread for the RDS processing, initialized all of the required coefficients and state-saving objects that need to be shared between blocks. The AudioBlock class was then extended with a new method called processRBDS that mimics the process described above in the python implementation. It begins with the demodulated FM data, extracts the rds channel, then the carrier, mixes them after taking care of all the required delays. After more

filtering and resampling the RDS demodulation is done, and data is ready for the clock and data recovery portion. All of the previous steps already had all the building blocks implemented in C++ for mono/stereo processing. However, for the CDR steps we had to port and verify all of the decoding and frame synchronization functions from the python model. All of these new functions were verified against the python implementation, using logging scripts and plots. The RDS thread consumes the same AudioBlocks as the Audio path, from a separate queue. This ensures every block is processed for both Audio and RDS.

Analysis and Measurements

Assumption: Block sizes vary for each mode

Path	Multiplications and accumulations per sample or bit	Non-linear operations per sample or bit
Mono Mode 0	1661	0
Mono Mode 1	1661	0
Mono Mode 2	1795	0
Mono Mode 3	2123	0
Stereo Mode 0	2265	20
Stereo Mode 1	2265	20
Stereo Mode 2	2465	21.77
Stereo Mode 3	2958	26.12
RDS Mode 0	4708	17.24
RDS Mode 2	22937	86.23

Mode	fmDemod Runtime
fmDemod mode 0	68 µs
fmDemod mode 1	88 µs
fmDemod mode 2	313 µs
fmDemod mode 3	125 µs

Path	Runtime (Average across all blocks in 4 sec sample) (151 Taps)
RF Processing Mode 0	10544 µs
RF Processing Mode 1	13477 µs
RF Processing Mode 2	78493 µs
RF Processing Mode 3	16517 µs
Mono Mode 0	522 µs
Mono Mode 1	607 µs
Mono Mode 2	4803 µs
Mono Mode 3	1573 µs
Stereo Mode 0	8867 µs
Stereo Mode 1	10962 µs
Stereo Mode 2	46097 µs
Stereo Mode 3	16574 µs
RDS Mode 0	9584 µs
RDS Mode 2	43280 µs

Filter Function	Runtime
MultiConvolveAndResample (RF) (Modes 0, 1, 2, 3)	7101, 9145, 35532, 13113 (µs)
MultiConvolveAndResample (Stereo)	7375, 8695, 34680, 12902 (µs)
ConvolveAndResample	453µs (decim = 5)

Multiply/Accum per Output Sample & Run Time Analysis

The measured results are in line with what we expected intuitively. Across all functions, mode 0 and mode 1 are almost identical to each other in terms of runtime. Mode 1 takes slightly longer for the mono and stereo paths, since it has to upsample before filtering and downsampling. Modes 2 and 3 take significantly longer to run as expected due to their larger block sizes. Modes 2 and 3 involve a large upsample ratio for the audio processing which required larger block sizes and number of taps to process properly in the convolveAndResample approach. Mode 2 has 5x the amount of samples per block as modes 0,1, and mode 3 has 1.5x. Comparing the run times across the RF, mono, and stereo processing we see that the runtimes increase linearly by approximately the same factors. Additionally, the multiplications/accumulations per output sample was computed for each mono, stereo and RDS (where applicable) mode. From the findings, the analysis agreed with the measurements taken and the run time per sample when computed, matched the trend for the multiplications/accumulations per sample plus the non-linear operations per sample. Moreover, the filter blocks in the code contribute the most to the run time of the SDR as the convolution proves to be the bottleneck of this project. Varying the number of taps across the the filters had a significant effect on the audio quality. When N_{taps} was set to 13 we saw very fast performance but at a huge cost to audio quality, the songs could still be heard, but with a huge amount of noise. When N_{taps} was set to 301 the audio quality was significantly better, this increased run time, however, with some small teaks we were still able to meet the constraint of live running. fmDemod increased linearly with the block sizes as expected with our analysis. The filter performance also as expected increases linearly as the block size changes across modes and linearly as the N_{taps} changed, This lines up with our Multiplication/Accumilation per sample analysis.

Proposal for Improvement

Looking back on the final SDR developed throughout the course, quite a few things would have been done differently knowing what we know now. Some of which are technical in nature, while others more general.

Starting with the more general improvements to the project and development processes, it would have been wise to develop a more robust logging and debugging mechanism into the system. This would have allowed us to spend more time tackling the underlying issues with the software as opposed to focusing on locating the issue in the code. Perhaps one way of doing this is to improve the file structure such that the Python SDR generates a log report of at each step in the SDR that can be compared to the C++ SDR. This would allow for a side-by-side comparison of both systems very easily.

With regards to technical improvements, our RBDS block would have benefited from a more robust data extraction flow. In particular, we believe that resyncing the symbols after every few blocks to ensure that the maxima/minima are being sampled with relative accuracy. Generally, signals may experience a phase shift over time, and as such, the sampling may lose sync with the data sampler.

Additionally, following some of the conversations we had with professors and teaching staff, we realized that the PLLs of the system may benefit from some manual tuning. However, given that RDS was still underway at the time, signal tuning was not a priority. Given more time, we would have tuned the PLLs to improve the quality of the signals in the system.

Lastly, in terms of the system's cosmetics, a user interface using something like the QT library would have been nice, but was obviously not beyond the project's time budget.

Project Activity

Hakam Atassi

Week 1	Research, looked into project documents, went over previous labs, etc…
Week 2	Received some lab grades. Fixed pre-existing issues and set up environment
Week 3	See Week 2
Week 4	Went over basic set up and boot strapping. Got piping into the code working. Set up GDB for debugging. Moved code from Python to C++ with teammates. Set up debugging scripts in Python.
Week 5	Worked on stereo (channel extraction, PLL, delay, mixing, filtering). Debugged delay issues by using L+R audio file. Did debugging with mono resampling team to figure out why L+R audio file was not separating.
Week 6	Worked on implementing and debugging RDS carrier extraction, Symbol Sampling, Manchester decoding, Frame Sync, ect in C++.
Week 7	Report

Abdelmoniem Hassan

Week 1	Port Lab 3 implementation into C++, Begin extending the mono implementation to support multiple modes.
Week 2	Read data from stdin vs file, fix aplay functionality. Refactor data processing in C++ to use AudioBlock class, Continue Debugging multiple modes in Mono processing.
Week 3	Refactor code to multithreading approach. Final touches to mono in c++. Help debug Stereo in C++
Week 4	More Stereo Debugging in C++. Help merge stereo branch into new refactored main branch.
Week 5	Began Implementing single pass RDS in python. Live performance testing for Mono/Stereo on pi.
Week 6	Convert single pass RDS to blocking, and port to C++. Small threading debugging, As well as stereo for multiple modes debugging.
Week 7	Report

Muaz Akhtar

Week 1	Going over project doc and reviewing previous labs in depth
Week 2	Began developing the convolution and up/downsampling function for optimization and setting up work environment
Week 3	See Week 2
Week 4	Continue to test convolution function and implement it in C++ and debug/test mono in C++
Week 5	Implemented multithreading by creating thread functions, setting up queues, threads and mutex locks

Week 6	Start to Implement python version of RDS in C++ and create RDS thread
Week 7	Report and Presentations

Saarah Ahmed

Week 1	Reviewed project doc and started to understand tasks and requirements
Week 2	Set up environment and troubleshooted initial personal errors
Week 3	Looked into mono and previous labs
Week 4	Mono implementation and testing
Week 5	Looked into RDS and implementation
Week 6	Implemented RRC filter in C++ and integrated with other RDS code
Week 7	Report and presentation

Conclusion

In conclusion, our software defined radio project provided us with a unique opportunity to learn more about the world of signal processing and filtering. Through our exploration, we gained a deeper understanding of how radio signals are processed and transmitted. We were able to implement these concepts using software, which allowed us to manipulate signals in real-time and achieve our desired outcomes for our SDR. Before 3DY4, our team had some basic knowledge about computing systems through previous courses and personal projects. Now, we can confidently say we have a solid understanding of real time signal processing theory and practice, particularly when limited by memory, codes, and processing speed.

Additionally, this project helped develop an understanding of how incremental improvements to performance are made. The flow is generally something that take the shape of

General Python/Matlab Model => General C++ program => Refined C++ program => Final program

Which happens to be the general flow our project followed throughout the development process for each individual component. It also introduced various optimization techniques to us, such as skipping unneeded processing, breaking up loops, utilizing data structures effectively, and configuring the compiler for better performance using compiler flags.

Looking forward, our team hopes to use the knowledge we developed in our future embedded and/or signal processing projects to develop high quality, high performance code

References

[1] P. Virtanen *et al.*, ‘SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python’, *Nature Methods*, vol. 17, pp. 261–272, 2020.

[2] K. Cheshmi, "Stereo Mode, and Multi-threading," Computer Systems Integration Project, Department of Electrical and Computer Engineering, McMaster University, Hamilton, March 13, 2023. Lecture.

[3] S. Chen and K. Cheshmi, in *COMP ENG 3DY4*, 2023.

[4] Nicolici, Nicola. “Real-Time SDR for Mono/Stereo FM and RDS.” April 10, 2023. Document.