COE3DY4 – Project Report

Wednesday Group 52
Jacob Luft - Allen Mei - Yazdan Rahman - Kevin Cheung
luftj1 - meia6 - rahmay1 - cheunk20
April 9 2021

Introduction

This project is regarding a software implementation of a software-defined radio (SDR) system for real-time reception of frequency modulated (FM) mono and stereo audio. The reception of frequencies is obtained using a combination of a RF dongle on the Realtek TRL 2832U chipset along with a Raspberry Pi 4. The SDR is created with the use of C++, with python parts for modeling and verification.

Project overview

To create the FM receiver in software, there are many parts that data is passed through to achieve the desired outcome. At the beginning of it all is the data itself, a radio frequency signal retrieved from a radio-frequency receiver. This data is translated into the digital domain, producing two 8-bit samples: one for the in-phase component, and another for the quadrature component. The translation of the data is done by the hardware, and only the output has bearing on the design itself. Following this is the built software defined radio (SDR) that manipulates the FM signal input into stereo audio.

The process starts off at the RF front-end block. This extracts the FM channel using a low-pass filter. As the FM channel is situated around 88.1 - 107.9 MHz in Canada, a 100 KHz low-pass filter is able to remove the unwanted data from the higher frequencies. There are two modes that the SDR operates in, mode 0 and mode 1. Mode 0 is sampled at a rate of 2.4 MSamples/sec, while mode 1 is sampled at a rate of 2.5 MSamples/sec. The extracted data is then decimated by 10 to downsample to a signal at the intermediate frequency (IF) which results in 240 KSamples and 250 KSamples, for mode 0 and mode 1 respectively. By default, the SDR operates in mode 0 unless specified by the user.

Following the RF front-end block, the data is split into two paths: the mono path and the stereo path. The mono path extracts the mono audio while the stereo path extracts the pilots tone for translation of the stereo channel. Starting with the mono path, the SDR obtains the mono audio channel which resides in the 0 - 15 KHz portion of the FM channel. This data is then downsampled to a rate of 48 KSamples/sec. The input data is the previous demodulated data at either 240 KSamples or 250 KSamples depending on the mode. This block outputs data in the form of 16-bit signed integers that can be read by an audio player.

In the stereo path there are two paths, the stereo carrier recovery and the stereo channel extraction. Both of these paths contain a band-pass filter to extract the necessary data. In the stereo carrier recovery, a 19 KHz pilot tone is recovered and synchronized with a phase-locked loop (PLL) to extract the stereo carrier. The stereo channel is simply extracted with the aforementioned band-pass filter. These two signals are mixed and further combined with the previously extracted mono channel to produce left and right audio.

Implementation details

Lab Work

One of the basic building blocks used throughout the labs and project is the discrete fourier transform which changes a signal from the time to frequency domain. Mathematically, this is done for a single frequency by integrating $x(t)e^{j2\pi ft}$through $\pm\ \infty$. In the code, this is achieved through a nested for-loop, multiplying each value in the input x vector by $e^{j2\pi ft}$, where the outer for-loop goes through each frequency bin, and the inner for-loop goes through each x value.

Next, to filter out high frequency signals we use a low-pass filter. This low-pass filter is created by multiplying the frequency response of the input with a rectangular window. The dimensions of the window is what determines the filter's specifications, the height determines the gain and the length determines the pass band. A sinc function is the inverse Fourier transform of the rectangular window, and a derivation of this function gives the impulse response of a low-pass filter. In our labs, we first implemented the impulse response using SciPy's lfilter method to generate filter coefficients. Afterwards, we created our own filter method which calculates the normalized frequency, sums the evaluations of the normalized sinc function for every tap to an output variable, then multiplies the sum at each index i by $sin^2(\frac{i\pi}{N_{Taps}})$to apply the hann window.

In reality, filters are finite in size, giving the term finite impulse response filters (FIR). These filters truncate the sinc function based on the number of filter taps. These taps are critical to the filter as it determines the signal quality. However as taps increase as does the number of multiplications, and as does the processing time. In our SDR we used 151 taps for both the RF front-end as well as the mono path. We found this number to be a good balance between quality and compute speed.

To process data in real time, we used the technique of dividing the input stream into blocks. In this fashion, it is possible to operate on data in real time. However, as convolution requires data from previous states, there would be discontinuities when traversing from block to block. To avoid this, save states are implemented, where data from the previous block is passed onto the next. In python, this was done by first performing convolution on the first block with the respective filter coefficients, and then saving the state into a separate variable. Then, for every consecutive block, the state variables are continuously re-passed as a state parameter into our function, and then overwritten with the output of the convolution function for the next block.

Translation of RF signal to IF signal

With these building blocks in place it is possible for us to move onto the first signal flow graph, the translation of RF signal to an IF signal. The RF hardware provides a signal in two components, the in-phase component (I) and the quadrature component (Q). To translate this signal into the desired intermediate frequency (IF) for mono processing, both components must first be passed through a low-pass filter as well as decimated by 10. After separating the I and Q components, we then passed them through a low-pass filter in order to extract the FM channel centered around 100 KHz. This data is then downsampled by a factor of 10 and can finally be demodulated to extract the original signal. We also replaced the given fmDemodArctan function with a computationally friendly version, and impulse response generation and digital filtering functions were replaced with our own. Mathematically, to demodulate the signal, partial derivatives are taken to determine the phase difference between the I and Q samples. Converting the arctan equation to a purely algebraic form, we get the equation $\frac{I\Delta Q - Q\Delta I}{I^2+Q^2}$. For the code implementation, we use the same equation using each pair of I and Q values. For each ΔI andΔQ, we simply take the current I and Q and subtract their previous respective values. The I and Q at the first index are set to zero to allow this functionality.

Using the outline from fmMonoBasic.py we then created the same process for block processing in fmMonoBlock.py. To allow for state saving we return an array in our block filter method that stores values of x. This array is then used in the next state convolution, and that state saves values for the next state. The general concept is implemented through FIRfiltering using convolution. The output of the function fmdemod() within our code will approximate the filtering coefficients to perform low-pass filtering of the signal. Generally, we implemented the convolution in a standard fashion where $y[n] = x[n]h[0] + x[n-1]h[1] + ... + x[n-h]$. Working in blocks presented the unique problem that if the $x[n-h]$term indexed beyond the beginning of our block, we needed to be able to access the data required. To perform this operation, we choose to store the last *h* number of samples from the previous block into a vector labeled *zi*. Note, the values are stored ***from the end of the previous block*** meaning the end sample of *x_prev* will be the beginning sample of *zi*. To continue the indexing until the for loop, using the length of *h* as it's control variable, we then index *zi* at using $zi[h - n - 1]$ using the conditional that while $i - k >= 0$ it would operate using *x* as input as at that point we would be operating in the region of our current, not previous, block.

As we began to work with real data, it became critical to be able to visualize the information that we were dealing with. Starting from the second lab, the gnuplot tool was used to check the correctness of our implementations. With our building blocks developed in labs, as we moved into block processing and other more complicated tasks, this enabled us to verify our design at each step with ease.

RF front-end to C++

The first task for the main project is to translate the previously implemented RF front-end functions into C++. The SDR operates in two different modes, 0 and 1. To select between the different modes, the bash command inputted would contain the mode number, 0 or a 1. By default the SDR operates in mode 0. Depending on the mode, variables would be initialized accordingly. As mode 0 sampled at a rate of 2.4 Msamples/sec, while mode 1 sampled at a rate of 2.5 Msamples/sec, we change our sampling rate variable Fs to either 2.4e6 or 2.5e6. With this we can have the same variables for both mode 0 and mode 1 for our various functions.

To implement RF front end, we first read in values from a fdin data stream file with the function readStdinBlockData. This function simply reads the RF data and splits the I and Q data into separate vectors for further processing. To now filter and decimate both the I and Q data, we run two instances of convolveBlockMode0, a function that performs convolution and downsampling. Decimation is performed with a for loop. By incrementing the loop by the decimation factor, we are able to perform calculations on only values that are multiples of the decimation. The state saving process is the same as the one found in python; the function stores the filtered data in one vector, and the saved state in another vector. These vectors would be filled and read as it moves from block to block. As returning multiple vectors in C++ is a cumbersome task, we decided to pass addresses of global arrays into the function to be changed. In this fashion, the data would be stored in real time outside of the function. With the data properly filtered and decimated, we now call the fmDemod function to demodulate the data.

Mono 0

We now move onto mono processing. Similar to RF front-end, we check which mode the SDR is operating in and initialize values accordingly. Values affected include the decimation value (mono_decim), upscale value (mono_upscale), and sampling rate (mono_Fs). We start with mono 0, as the process is identical to RF front-end processing. As such we reuse the convolveBlockMode0 function, which produces the desired outcome.

When porting over the Mode 0 code into Allen's VM, the program started crashing when calling the logVector function specifically after estimatePSD function calls. This was due to the block_size being too large to handle, thus the estimatePSD implementation at certain block sizes would generate NAN arrays. This was mainly because of the estimatePSD function and how it was built, if no_segments reached a size of 0 due to block_size being too big. This was fixed by decreasing the block_size by continuous factors of 2 till the estimatePSD function started running.

Mono 1

Mono mode 1 is similar to mode 0, but we artificially upsample, in addition to the downsampling functionality of mode 0. Initially, we created a separate function that would first upsample the data by itself. We then took the upsampled data and fed it into the convolveBlockMode0 function, which did the filtering and downsampling. This allowed us to clearly see what was going on when we upsampled, as we knew the filter and downsampling was correct. The current implementation is done all together, with the upsampling, filtering and downsampling in one function called convolveBlockMode1. This function foregoes padding the upsampled signal with 0's, instead working with the unaltered input data. We treat the coefficient vector used in convolution as though we were feeding it the upsampled signal at a rate of

250kH, then apply a gain factor equal to our upscaling factor to mitigate the decreased magnitudes resulting from upsampling. We then just “spoof” convolving using the upsampled signal and returning the downsampled signal, but we never actually pad 0’s or perform all calculations. Because we know we’d only keep the output samples of an upsampled vector *y* at intervals of 125, and we know that any input vector *x* index that isn’t a multiple of 24 will yield a 0 value, and therefore x[k]h[k-n] will be 0, we can just selectively perform calculations that we know will be reflected in our downsampled vector *y’* and save a ton of time on computations.

When we graphed the mono 1 output in the frequency domain, the first couple samples did not match the python at first. We found the issue to be with our upsampling. When we passed in the calculated coefficients, we assumed the fs and taps were multiplied by 24, but it was not the case. The fix was to simply multiply mono_fs by 24.

Stereo Processing

With mono processing completed we move onto stereo processing. We first modeled the process in python, to allow for easier bug fixing. We first started working with the framework of fmMono1.py as it already contained a working mono path. First we obtained the coefficients for the filter from both the stereo channel audio and the pilot tone audio using a function called manualBPIR. With these coefficients we passed them into our filter to extract the stereo channel audio as well as the pilot tone audio. As the data was passed in as blocks, we implement the same state saving techniques used in the earlier portion of our design. Following this we call the PLL function. This function synchronizes to the pilot tone to produce an output that is multiplied by the NCO to be used for mixing.

During testing we found that the left audio channel was outputting the audio we expect, but the right one had static. The error was that our for loop exit condition was too big, and we were indexing the audio block out of bounds. As C writes to other parts of memory when indexed out of bounds, we believed that the right audio block was being overwritten. By changing the exit condition in the for loop the issue was solved.

When looking at the NCO output graphs we found that the end and beginning of blocks were not continuous. This led to us outputting values going in and out of PLL. From this we noticed that ncoOut remained constant throughout different blocks. The problem turned out to be that the trigOffset value was not being updated throughout blocks; we had assumed it was being changed within the PLL function. To solve this we simply updated triOffset by the pilot length every iteration.

While working through the project we found that initializing all the different variables at the beginning of our code made troubleshooting a lot easier. By creating intuitive names for all numbers being passed through our code we could clearly see what the input of our functions were, and there was no confusion as to where a number came from. Changing values to test our code also became a trivial task as all of our variables were sorted and located at the top of our main class.

Analysis and measurements

Block size is 102400 for both mode 0 and mode 1. We desired our block passed to fwrite/aplay to be approximately 1,024 samples. This was chosen as it is small enough to be processed within time constraints, and large enough to capture a sufficient amount of information. We then work in reverse. Downsampling by 5 to 48kHz (or by downsample/upsample ~= 5) puts a *5 multiplier to 1025. Front-end downsampling decimates by 10, so a *10 multiplier is applied to 1024*5. And finally, the I/Q stream read-in is split into 2 separate paths for processing before consolidation. So apply a *2 multiplier to 1024*50 for 1024*100 = 102400.

Practically any vector that's size is not related to block size is related to the chosen number of num_taps. The larger num-taps is, the larger the vectors created using it as a reference are and this increases the number of multiplications performed to convolve each sample of our output vector from the convolution. High num_taps gives more accuracy in filtering but also increases run time needed for multiplications. We chose num_taps to be 151. Note that when creating the vector coefficient for the upsampled signal, we also upsample num_taps to num_taps * 24.

In mode 0 and 1, within the front-end filter for both Q and I samples of size 51200, num_taps is 151 and we have a decimation factor of 10. This means we have 5120 samples to compute with 151 multiplications per sample, or 773,120 multiplications required compared to the 7,731,200 required if we processed the entire input block and then decimated through sample selection. This convolution takes between 2.7 to 6 ms, though usually operating around 3ms. Each block will also require 151 accumulations as we store num_taps length of values for use for our next input block. We then perform fm demodulation which requires ***at most*** 5120 *2 multiplications and 5120 divisions and 5120*2 accumulations. This process takes between 0.05 to 0.1ms.

For Stereo, during extraction of both the pilot tone and the stereo block, the convolution takes an input of size 5120, num taps of 151, and downsampling ratio of 1 (thus no downsampling). This turns into 5120*151 = 773,120 multiplications. The speed of all these computations take around the same as the I/Q convolutions, coming in at around 2.7 to 6 ms, though usually operating around 3ms. This makes sense from a theoretical perspective since the number of multiplications is exactly the same.

In mode 0 within the mono path, convolution takes an input of size 5120, downsampled by 5, and uses 151 multiplications per sample computed. So 5120/5 = 1024, 1024*151 = 154,624 multiplications. Within the stereo path for mode 0 we do the same number of multiplications as the mono path. These convolutions individually run for between 0.5 to 0.9ms.

In mode 1, within the mono path, the input size is "upsampled" to 122,880, and num_taps is 3624. We then downsample by 125 so 122880/125 = 983. However, because we only need to do the convolution for the 983 samples created post-downsampling, and we know non-zero values of input only exist at every 24th *x* index, and the *h* value indexed will be changed to match accordingly, we still only perform 151 calculations per cycle such that 983*151 = 148,433 operations.We then must accumulate 3624 values from the input *x* vector for use in processing future blocks. This runs between 0.47ms to 0.7ms. The input sizes and output sizes are the same between mono and stereo. So they individually take approximately the same amount of time.

The PLL function, which includes the atan2() function, takes between 1.5 to 3ms, making it one of the more time consuming individual function calls. The input of the PLL includes an array of size 5120 and various other constant values. The PLL processes these elements by doing 8 multiplications as well as some accumulators and also an atan2() function call per 1 of the 5120 input elements. This amounts to 5120*8 = 40,960 multiplications which is typically not very large as compared to some of the convolve functions mentioned previously. The main overhead is due to the atan2() function call which happens 5120 times which tremendously slows down the function.

Cumulatively, the time to run 1 block of input data to output in either mode 0 and mode 1 takes between 14 to 29 ms. On average, it typically runs within the range of 14-15ms.

Proposal for Improvement

Threading should have been utilized also in order to create modularity for others who want to include more synchronous tasks such as Audos Subcarrier and Directband processing. The implementation of a synchronous queue system of simultaneous thread executions within the RF Thread would have greatly benefited modularity. A more systematic approach of error checking within each data processing function such as convolution would also enhance debugging for further implementation in the future. Another improvement would be an approximation of the arctan function, as it would have led to a great increase in performance because of the tremendous bottleneck of the PLL function. More specifically, using second and third order polynomials as well as rational functions, a more computational friendly approach could be satisfied with the drawback of decreased precision.

Within Mode 1 convolution operations, reduction of multiplications, divisions, and modulo would be a potential area of improvement, as division and modulo in particular may pose more of a resource threat to time sensitive systems. A more rigorous approach to the convolution algorithm and more practical in the real world would be the FFT approach. More specifically the Fourier Transform of the series is calculated, and pointwise multiplication would occur instead, then the Inverse Fourier Transform of the resulting series is calculated. This could theoretically bring down complexity from $O(N^2)$ to $O(N\log N)$ complexity.

Project activity

Week 1	**Everyone:** Read through the project document. **Jacob:** received the RaspPi/RF Dongle hardware.
Week 2	**Everyone:** Initial project changes (command line input, transferring lab work to project doc.) Reviewed lab and lecture material. **Yazdan, Kevin:** converted fm_demod python to computationally friendly C++ code. **Jacob:** finished setting up RaspPi/RF Dongle hardware, isolating strong frequency channel, documented calculation logic for code.

Week 3	**Yazdan, Kevin, Allen**: Finished implementing RF frontend and mono mode 0, verified against plots from python model. (Page 3, Paragraph 4) (Page 4, Paragraph 1 - 2) **Jacob**: Began live testing of mono mode 0, restructuring code. Worked on optimization of methodology and memory allocation methods. (Pages 3-6)
Week 4	**Yazdan, Kevin**: Python implementation of stereo (Page 5, Paragraph 3) **Everyone**: C++ stereo implementation + debugging. (Page 5, Paragraph 3 - 6) **Jacob**: Live testing of mono mode 0, 1. Debugging of fwrite and upsample/downsample convolution. (Page 6)
Week 5	**Everyone**: Debugged + optimized convolution methods for mode 0 and mode 1 for reasonable runtime (Page 4, Paragraph 3 - 5) (Page 5, Paragraph 1) **Yazdan, Kevin, Allen**: Implemented + debugged PLL python model, then converted to C++ (Page 5, Paragraph 3, 5) **Jacob**: Live testing of mono mode 0, 1 (Page 6). Experimented with variation in block size and num_taps to varying degrees of effectiveness. Defaulted to standard values from previous labs. (Page 5, Paragraph 6)
Week 6	**Everyone**: Finalized stereo, convolution optimization, and cleaned up code. (Page 5, Paragraph 1 - 2) **Jacob**: Live testing of mono mode 0, 1 (Page 6).

Conclusion

Over the course of the past 3 months, throughout our lab and project work, we were given the opportunity to consolidate our knowledge acquired throughout the past few years in a practical setting. Topics such as digital filtering, signal processing, and frequency modulation were incorporated, while technical skills such as coding, debugging, optimization, and runtime analysis were developed. By gaining experience on a realistic application, we are able to understand the complexity of performing tasks that may seem simple in nature, and appreciate the amount of work that goes into preparing designs to be interfaced into the real world.

References

N. Nicolici, *McMaster University Online Courses*, 22-Feb-2021. [Online]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/343598/viewContent/3090007/View. [Accessed: 9-Apr-2021].

MatplotLib Development Team, "matplotlib.pyplot.plot¶," *matplotlib.pyplot.plot - Matplotlib 3.4.1 documentation*, 2012. [Online]. Available: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html. [Accessed: 9-Apr-2021].

Cplusplus Development Team, "," *cplusplus.com*, 2000. [Online]. Available: http://www.cplusplus.com/reference/list/. [Accessed: 09-Apr-2021].