COMPENG 3DY4 Final Report

Winter 2024

Group 09

April 5, 2024

Member Name	MacID	Student Number
Ginoth Kumarakulasingam	kumarakg	400258323
Ty Greenwood	greent14	400311174
Oscar Lu	lus70	400252202
Cass Smith	smithc85	400255184

Introduction

The objective of this project is to implement a software-defined radio (SDR) system in real-time. The system receives frequency-modulated (FM) mono/stereo audio, as well as digital data sent using the radio data system (RDS) protocol, performs signal processing and plays the audio or displays the data.

Overview

The SDR consists of the following components: RF hardware, RF front-end, mono, stereo and RDS. The RF hardware is used to receive analog RF signals and produce digital I and Q samples. The RF front-end extracts the FM channel and performs FM demodulation. The mono component extracts the mono audio sub-channel, performs filtering and resampling to an output of 48 kilosamples/sec. The stereo component extracts the 19 kHz pilot tone, performs the downconversion of the stereo channel, which is then filtered and combined with the mono audio data to produce the left and right audio channels. Finally, the RDS component extracts digital data according to the RDS protocol standard and displays the information words of data. In order to have the SDR system working in real-time, multithreading was also implemented. The computations required for the RF front-end were done in a thread, and the output FM demodulated data was fed into two queues, one to be used by the audio thread and the other to be used by the RDS thread. This allows the RF front-end, audio data, and RDS data to be computed simultaneously.

Implementation Details

Labs

In lab, we created a Python function that generates filter coefficients for a low pass filter, given a cutoff frequency, sampling frequency, and number of taps. We translated the pseudocode given in the Lab 1 instructions to Python and encountered no issues in getting it to work properly. Also in Lab 1, we were tasked with creating a convolution function to replace the lfilter function in Python, also adding state saving in order to be able to perform block convolution. State saving was done by storing the last h-1 values of the input block being processed, where h is the number of filter coefficients, into a separate array so they can be used when the next block is being processed.

In Lab 2 we focused on converting our Python code from lab 1 to C++. We wrote C++ functions for impulse response generation and convolution. We had no issues converting the impulse response function to C++ and had no issues with its functionality. When converting the function for single pass convolution, we did run into an issue where we were getting a segmentation fault. This was because we were resizing the output vector to the incorrect size, which was fixed by correcting the size to that of the input signal plus the size of the filter minus one (we were previously setting it to the same size as the input signal). The function was implemented by taking an output vector, input vector, and a filter coefficient vector as inputs, and looping through output vector, at each step looping through filter coefficient vector and adding the result of the multiplication between a value from the filter coefficient vector and the input vector (as long as the index at which we are accessing the input vector is within the bounds of the vector). To implement block

convolution in C++ we kept the same basic structure as single pass convolution, except that if we were trying to access the input vector at a negative index, we instead used a value from the state vector (an addition vector passed to the function). After computing all the elements in the output vector, we then saved a number of elements from the end of the input vector equal to the size of the filter coefficients vector minus in the state vector, to be used for computing the next block.

The goal in Lab 3 was to implement the mono audio path in Python. We were given a Python file in which the front-end filtering and downsampling were already complete. The file read IQ data from a separate file, normalized the data between one and negative one, then split into I data and Q data while being put through a 100kHz low pass filter at the same time. The filtered data is then downsampled by a factor of ten, then finally demodulated using the built in fmDemodArctan function. From there we filtered the demodulated data through a low pass filter with a cutoff of 16kHz, first using the built in lfilter function, then using our own single pass convolution function, then finally with our own block processing convolution function. The resulting signal was then downsampled by a factor of five, at which point a .wav file was generated to be able to listen to the audio produced. We then replaced the built in fmDemodArctan function with a demodulation function of our own. Our demodulation function first declares an output array the same size as the I data, then loops through the output array finding the derivative of I and Q at each index using the element at the current index and the previous element (passed into the function). Those derivatives are then plugged into a function given in lecture:

$$\boldsymbol{m(t) = \frac{1}{I^2(t) + Q^2(t)}\left[I(t)\frac{dQ(t)}{dt} - Q(t)\frac{dI(t)}{dt}\right]}$$

Figure 1: Fast demodulation formula.

Mono Path

The mono path is implemented in the audio thread, which produces both mono and stereo audio. Float values of FM demodulated data at the IF (intermediate frequency) sampling rate are taken as input and the output is the mono audio consisting of 16-bit signed integer values at the audio sample rate. This is done by first low-pass filtering the FM demodulated data, and then performing downsampling to reduce the sample rate from the IF Fs to the audio Fs. However, the IF Fs and audio Fs vary between the 4 modes of operation, and for modes 2 and 3 upsampling must happen first to resample the data from the IF Fs to the audio Fs. In our implementation of the mono path, filtering and downsampling/resampling are done at the same time so the run time of the system is optimized.

First, the upsampling factor U and the downsampling factor D are set such that $\frac{U}{D} \cdot IF = Audio\ Fs$. Next, the low-pass filter coefficients are calculated using the number of taps and the cutoff frequency equal to $\frac{U}{D} \cdot \frac{IF}{2}$. Block convolution and downsampling, for modes 0 and 1, or block convolution and resampling for modes 2 and 3, are then performed. Both the convolution with downsampling function and the resampler function were made significantly easier by the fact that we already had both Python and C++ implementations convolution, since we only had to make small adjustments to those functions instead of starting from scratch.

We did not model the block convolution with downsampling in Python because the pseudocode for the function was given in lecture, so we decided to instead jump right into the C++ implementation of the function. When comparing this function to standard convolution, it is clear

to see that most values generated by normal convolution are thrown out in the case of convolution with downsampling, so to speed up the function, we only compute the convolution for every D values in the input signal, skipping all the other calculations and saving a lot of time in doing so. Even though we skipped modelling this function, it worked as expected immediately.

In the case of block convolution and resampling, we did decide to model the function in Python first, to figure out the proper indexing we would need, to make sure that any fast-resampling function we create is faster than performing the full slow implementation of resampling, and to check for equivalence between the fast and slow resamplers. We found that we should loop through the input signal starting from index zero, incrementing the index by D after each iteration. In each iteration of that first for loop we use another for loop to iterate through the filter coefficients, but we found that the starting index for this loop should be n modulo U where n is the index of the outer for loop (the one iterating through the input signal), and this index should increment by U each iteration. This is done because the process of upsampling is done by zero padding the input vector so that it only contains a non-zero element every U elements. Using the indexing discussed above skips all calculations involving those zero elements, since the result of those calculations will always be zero. During the modelling of this function, we ran into an issue where the fast resampler was taking the same amount of time to run as the slow resampler, which was obviously a big problem. This was fixed when we realized that we should not loop through every single filter coefficient each time, as that meant we were still performing a lot of useless calculations.

Stereo Path

The stereo path is implemented in a separate thread from the frontend and RDS paths. The thread first pre-calculates filter coefficients and declares vectors to hold data and state at various stages in the path. Coefficients for bandpass filters are computed by elementwise subtraction of coefficients of two lowpass filters.

IF blocks are passed into the thread through a mutex-synchronized queue of float vectors, the setup outlined in lecture. IF data is bandpass filtered to extract the stereo subchannel and the pilot tone, the latter of which is passed to the PLL. The PLL, which is a simple refactoring of the provided Python code into C++, produces a 38 kHz sinusoid that is mixed with the stereo subchannel, and low-pass filtered to extract the audio-frequency stereo difference signal. The mixer consists of a simple elementwise multiplication of two data blocks. Meanwhile, the stereo sum signal is extracted the same way as it was in the mono path, with the addition of a fast all-pass filter to ensure the sum and difference signals are phase-aligned. This all-pass filter simply copies input blocks into its output but keeps a state vector of the same size as a filter does, introducing the same delay as a filter but requiring far fewer computations. Once both sum and difference signals are prepared, both are downsampled to the audio sample rate and fed to the combiner, which computes the elementwise sum and difference and applies appropriate gain to produce interleaved left and right channels. This interleaved data is fed to the same output routine used by the mono path, which converts floats to signed 16-bit integer format and prints to the standard output.

RDS Path

The only new component required for the RDS path is the Root-Raised Cosine Filter (RRCF). This was implemented using the formulae provided in the specification. Squaring nonlinearity was

implemented using the mixer module developed for the stereo path, by simply mixing the signal with itself.

RDS processing is done in a separate thread from the frontend and audio paths, with data sent to it through an identical vector queue arrangement to the one used by the audio path. The incoming IF data is bandpass filtered to extract the RDS channel from 54 to 60 kHz. This channel is then squared, bandpass filtered again from 113.5 to 114.5 kHz, and passed into the PLL, which has an NCO factor of 0.5. This produces the 57 kHz RDS carrier, which is mixed with the filtered RDS channel from before, which has passed through a fast all-pass filter to ensure phase alignment. This mixed signal is then lowpass filtered to 3 kHz and resampled to the requisite RDS sample rate as defined in the specification. This signal is then filtered using the RRCF, at which time it is ready for data recovery.

RDS Data Processing

Manchester and peak detection

Manchester decoding was first implemented by obtaining a vector that stores all the peaks and troughs from the Pre CDR samples vector as “1s” or “0s”. This is done by finding the 1st zero crossing in the Pre CDR samples output vector “rds_cos_filt”, this is done by looping through each sample of said vector while comparing 2 subsequent values. If these 2 values differ in sign a zero crossing is detected, and the index of the latter value is taken as the index for the “zero”. Then the first peak or trough is found by seeing if the zero index is more than 1.5 symbols per sample (SPS) away from index 0 in the vector, if so the first peak index will be set to the zero-crossing index minus 1.5 SPS. (This case would occur if the first 2 symbols are both peaks or troughs. If the zero-crossing index is less than 1.5 SPS from the start and more than 0.5 SPS from the start, then the first peak index is the zero-crossing index minus 0.5 SPS, this case is when the Pre CDR samples block contains a first peak or trough that is cut-off at the start, meaning the first symbol does not start at zero. The last case is when the Pre CDR samples block contains the tail end of a symbol (blk 2 in the case 3 diagram), since the symbol would have been used in the previous block, we do not use this in this block which means the first peak index is set to the zero-crossing index plus 0.5 SPS.

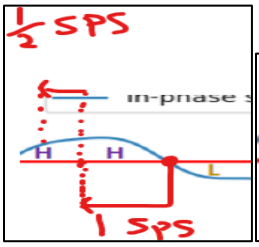

Figure 2. Case 1

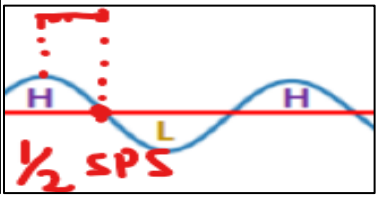

Figure 3. Case 2

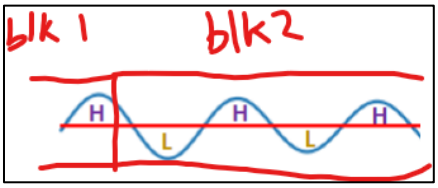

Figure 4. Case 3

After the first peak index is found, a vector named “peaks” is created with the size (rds_cos_filt.size() / SPS), this is equal to the number of symbols in the current block of rds_cos_filt (Pre CDR samples vector). Then the empty peaks vector is iterated through, in each iteration the Pre CDR vector at index [first_peak_index + iteration*rds_sps] (this is the max or min point at each symbol) is checked to be positive or negative where positive values are a peak thus we set the peaks vector at index of the for loop iterator to 1 and 0 if negative. Now we have a vector “peak” that contains the highs or lows of the Pre CDR samples, so we can compute the Manchester decoded output bits, but first we must compare 2 possible paths for the Manchester decoding, one which

starts pairing from the first peak, and the 2nd path starts pairing from the 2nd peak. This is done by creating both paths and storing the output “decoded bits” in 2 separate vectors “bits1, bits2” and taking the one that has the best decoded output (mybits). The best output is determined by a pair of variables called “mistakes1” and “mistakes 2”. While decoding in each path, when the 2 checked peaks are equal “2 highs or 2 lows” (Invalid sequence) mistakesX variable is incremented by 1, and the decoded value is set to zero, else the 2 peaks are decoded using Manchester decoding so the bit=the first symbol. State saving is then implemented in this by saving the last “symbol” of the peaks vector if it was not used, this is done by checking if the peaks vector size was even or odd and which path was used for decoding output. For example if the 2nd path was taken and the peaks vector was odd this would mean the last symbol in “peaks” would be matched and decoded, state saving for the next block would not be necessary, but if the peaks vector was even and the 2nd path was used this means the last symbol would not be matched so state saving is enabled and the last symbol is appended to the start of the next block’s “peaks” vector.

Differential decoding and Frame synchronization

A new vector named “rds_bitstream” is sized to the same as the Manchester decoded bits vector “mybits”, and the contents are just each bit in the mybits vector XOR’ed with previous. State saving is implemented by saving the last bit of each block to a variable and then doing the first XOR for the subsequent blocks with said previous state bit (first block first XOR is done with first bit and a zero, since XOR of anything with zero is itself). The resulting output of differential decoding “rds_bitsteam” is then used for frame synchronization.

Fast matrix multiplication function is created for multiplying matrices M*P=out that only contain values of “0” and “1”, the M.columns = P.rows, and out being the same size matrix as M. This is done by a nested for loop where the outer loop iterating through each column of the output, and the inner loop iterating through each row of the output, where the output is the sum of each multiplication of each M row element and each P column element. But instead of multiplying and addition, we use & for multiplying and XOR for addition since our input are only 1s and 0s, and these operations are faster.

Frame synchronization is done by declaring a vector called “frame” which will be used to compare frames in each block to its resulting offset syndrome words. We take the 1st 26 bits from “rds_bitsteam”, with an iterator variable “frame_index” that’s initialized to 0 every block, and do a fast matrix multiplication with a Parity check matrix from the RDS standard “P”. The output frame “mat_mult_result” is then compared to the 5 different syndrome words corresponding to a different offset, and if non match, the 2nd to 27th bits are checked by incrementing frame_index . When finally matched a flag “synched” is set to true and synch_type is set to the corresponding offset type. This is all done in a while loop with the condition of (synched==false) and the frame_index+26 being inside the rds_bitstream vector (This is the initial “window shifting”). Exiting the while loop If synched==true and frame_index+26 is within the rds_bitsteam vector then the frame index is incremented by 26 (moving onto next frame), then another while loop with the condition of frame_index+26 is within rds_bitsteam vector, the same matrix multiplication and syndrome word comparisons are done and resynchronization by incrementing frame_index is also done, if synched within the while loop the frame_index is incremented by 26 (go to next frame). Finally the state is saved by adding the non-checked values of rds_bitstream to the frame state which is appended to the start of the next rds_bitstream block.

Application Layer

Our application layer implementation only reads PI and PT information.

When a frame of type A is detected by the frame sync system, the application layer handler reads the first 16 bits of this frame, converts it into a short integer value, and compares it with a stored PI value. If the new value is different from the stored one, it prints a message (to the standard error interface) with this PI code in hexadecimal format.

Similarly, when the frame sync system encounters a frame of type B, the application layer handler reads the 6^{th} to 10^{th} bits from the frame, converts these bits to an unsigned char value, and compares this PT code to the stored one from previously. If the codes differ, the new code replaces the old one, and it is used as an index into an array of strings to extract the human-readable program type, which is printed to the standard error interface.

Analysis and Measurements

Multiply / Accum per output sample – Mono Path

At the RF front-end, the number of multiplications required for convolution for the filter and downsampling is equal to $101 \cdot IF \cdot 2$. This is because the number of multiplications required for a convolution is equal to the size of the filter multiplied by the number of output samples and multiplied by 2 since the calculations are required for both the I and Q paths. In the mono path, the number of taps for the low-pass filter is equal to 101 and the output rate is equal to the audio Fs, which is equivalent to the IF Fs multiplied by U/D. Therefore, the number of multiplications required is equal to $101 \cdot IF\left(\frac{U}{D}\right)$. As a result, the number of multiplications per audio output sample in the mono path is equal to $\frac{101 \cdot IF \cdot 2 + 101 \cdot IF\left(\frac{U}{D}\right)}{Audio\ Fs}$. This calculation can be used to determine the number of multiplications and accumulations per audio output sample for each mode:

$$Mode\ 0\ = \frac{101 \cdot 240000 \cdot 2 + 101 \cdot 48000}{48000} = 1111$$

$$Mode\ 1\ = \frac{101 \cdot 288000 \cdot 2 + 101 \cdot 36000}{36000} = 1717$$

$$Mode\ 2\ = \frac{101 \cdot 240000 \cdot 2 + 101 \cdot 44100}{44100} \approx 1200$$

$$Mode\ 3\ = \frac{101 \cdot 160000 \cdot 2 + 101 \cdot 44100}{44100} \approx 833$$

Multiply / Accum per output sample – Stereo Path

The stereo path processing is the same as the mono path, with the addition of two IF-to-IF filtering stages, and one extra IF-to-AF filter/resample stage. Thus, the MAC per sample is:

$$\frac{101 \cdot IF \cdot 4 + 101 \cdot IF\left(\frac{U}{D}\right) \cdot 2}{Audio\ Fs}$$

$$Mode\ 0\ = \frac{101 \cdot 240000 \cdot 4 + 101 \cdot 48000 \cdot 2}{48000} = 2222$$

$$Mode\ 1\ = \frac{101 \cdot 288000 \cdot 4 + 101 \cdot 36000 \cdot 2}{36000} = 3434$$

$$Mode\ 2\ = \frac{101 \cdot 240000 \cdot 4 + 101 \cdot 44100 \cdot 2}{44100} \approx 2401$$

$$Mode\ 3\ = \frac{101 \cdot 160000 \cdot 4 + 101 \cdot 44100 \cdot 2}{44100} \approx 1668$$

Multiply / Accum per bit– RDS Path

The RDS path to a single bit has the same RF front end as mono which is 101*IF*2 mults, then 2 bandpass filters are applied which is 101*IF mults each, after a resampler is applied the same way as in mono so $101 \cdot IF\left(\frac{U}{D}\right)$ mults, then the resampled signal is filtered again which is $101 \cdot IF\left(\frac{U}{D}\right)$ mults again. Finally, we have the number of mults/sec we now divide by the number of bits/sec to get the final number of mults/bit

$$\frac{101 \cdot IF \cdot (2 + 1 + 1) + 101 * IF * \frac{U}{D} * 2}{\frac{symbols}{sec}\frac{1}{2}\frac{bits}{symbol}}$$

$$Mode\ 0 = \frac{101 \cdot 240000 \cdot (2 + 1 + 1) + 101 * 240000 * \frac{133}{384} * 2}{2375 * \frac{1}{2}} \approx 14\ 754$$

$$Mode\ 2 = \frac{101 \cdot 240000 \cdot (2 + 1 + 1) + 101 * 240000 * \frac{19}{120} * 2}{2375 * \frac{1}{2}} \approx 10\ 761$$

Atan2/sin/cos – PLL and NCO

The number of iterations of the loop used in the PLL is equal to the size of the input. In each iteration there is 1 atan2, 2 cos, and 1 sin calculation, for a total of 4 nonlinear operations. In the stereo path, the size of the input to the PLL is equal to the size of the block of FM demodulated data, which is equal to IF Fs. The output of the PLL/NCO is also equal to the size of the input block. Therefore, for each output sample of the PLL/NCO there are 4 nonlinear operations.

Runtimes per Function

Note that all audio samples in the below tables were recorded for 3 seconds.

101 Taps - 3 seconds	Mono								Stereo							
Mode	0		1		2		3		0		1		2		3	
All times in ms	Total	Average	Total	Average	Total	Average	Total	Average	Total	Average	Total	Average	Total	Average	Total	Average
Blocks	73		70		73		73		73		70		73		73	
Read time	4267.48	58.45863	4105.19	58.64557	4261.54	58.37726	3370.58	46.17233	4278.96	58.61589	4134.32	59.06171	3974.04	54.4389	2532.86	34.69671
frontend filt time	720.655	9.871986	856.407	12.23439	712.035	9.753904	475.349	6.51163	720.131	9.864808	857.018	12.24311	703.23	9.633288	475.664	6.515945
demodulation time	6.61116	0.090564	7.79976	0.111425	6.61433	0.090607	4.22313	0.057851	6.83557	0.093638	7.97523	0.113932	6.60278	0.090449	4.26137	0.058375
mono impulse time	0.436096	0.005974	0.619542	0.008851	4.61854	0.063268	14.4503	0.197949	0.263484	0.003609	0.346003	0.004943	6.57029	0.090004	14.0834	0.192923
stereo impulse time	0.149557	0.002049	0.157891	0.002256	0.163705	0.002243	0.181724	0.002489	0.147297	0.002018	0.149575	0.002137	0.187724	0.002572	0.173261	0.002373
write time	1.24647	0.017075	1.53196	0.021885	3.18955	0.043692	4.59522	0.062948	8.96138	0.122759	9.23274	0.131896	9.52208	0.130439	10.056	0.137753
mono lpf time	158.864	2.176219	120.248	1.717829	1001.75	13.7226	1046.48	14.33534								
pilot recovery									795.463	10.89675	954.177	13.6311	798.37	10.93658	535.159	7.330945
stereo channel extraction									801.331	10.97714	943.444	13.47777	802.125	10.98801	539.841	7.395082
pll time									203.979	2.794233	243.645	3.480643	207.738	2.845726	135.223	1.85237
mixer time									3.35655	0.04598	3.98933	0.05699	3.39452	0.0465	2.24123	0.030702
allpass time									2.79095	0.038232	3.11321	0.044474	2.77353	0.037994	1.71746	0.023527
digital filtering time									314.929	4.314096	569.751	8.1393	1869.89	25.61493	2054.37	28.14205
combiner time									1.15005	0.015754	0.878401	0.012549	1.30628	0.017894	1.2291	0.016837
Total (- read/write time)	886.7158	**12.14679**	985.2322	**14.07475**	1725.182	**23.63262**	1540.684	**21.10526**	7129.337	**97.66215**	7718.807	**110.2687**	8376.228	**114.7429**	6296.823	**86.25785**

13 Taps - 3 seconds	Mono		Stereo	
Mode	0		0	
All times in ms	Total	Average	Total	Average
Blocks	73		73	
Read time	4815.05	65.95959	4885	66.91781
frontend filt time	189.745	2.599247	92.2676	1.26394
demodulation time	12.9793	0.177799	6.43225	0.088113
mono impulse time	0.200087	0.002741	0.480266	0.006579
stereo impulse time	0.026554	0.000364	0.049406	0.000677
write time	2.08863	0.028611	54.6386	0.748474
mono lpf time	34.3145	0.470062		
pilot recovery			77.7041	1.06444
stereo channel extraction			78.4236	1.074296
pll time			201.308	2.757644
mixer time			3.34727	0.045853
allpass time			2.59179	0.035504
digital filtering time			32.0028	0.438395
combiner time			1.08024	0.014798
Total (- read/write time)	237.2654	**3.250212**	495.6873	**6.790237**

301 Taps - 3 seconds	Mono		Stereo	
Mode	0		0	
All times in ms	Total	Average	Total	Average
Blocks	73		73	
Read time	2988.11	40.93301	2363.09	32.3711
frontend filt time	2099.24	28.75671	2149.54	29.44575
demodulation time	6.66703	0.091329	6.70984	0.091916
mono impulse time	0.400018	0.00548	0.220407	0.003019
stereo impulse time	0.459054	0.006288	0.174352	0.002388
write time	1.32365	0.018132	10.6023	0.145237
mono lpf time	484.729	6.640123		
pilot recovery			2390.43	32.74562
stereo channel extraction			2379.39	32.59438
pll time			203.577	2.788726
mixer time			3.31003	0.045343
allpass time			2.75276	0.037709
digital filtering time			962.489	13.18478
combiner time			1.22362	0.016762
	5579.605	**76.43295**	10462.91	**143.3275**

Comparison to Analysis

All measurement times are for an average block of data.

For the mono path with 101 taps with 3 seconds of audio data, we measured mode 0 to 1 being about 16% slower VS the multiplication difference being about 54%, and measuring mode 2 to 3 to be about 10.7% faster VS the multiplication difference being about 30.5%.

0-1: 16% slower VS 54%, and 2-3: 10.7% faster VS 30.5%.

These measured values are somewhat related to the multiplication difference the modes 0 & 1 and 2 & 3 cannot be compared with each other, since besides multiplications there are a lot of differences between the downsampler (modes 0 & 1) and resampler (modes 2 & 3).

For the stereo path with 101 taps with 3 seconds of audio data, the same order of %difference is the following, 0-1: 13% slower VS 54%, and 2-3: 25% faster VS 30.5%.

Comparing the measured time VS multiplications #taps changing from 13-101-301 with 3 seconds of audio data is as follows:

Mono mode 0: 3.25, 12.146, 76.432. The increments (13 taps to 101, and 101 taps to 301) being 3.74X and 6.29X

The expected mult increments are: 7.77X and 2.98X. These apply to both mono and stereo mode 0

Stereo mode 0: 6.79, 97.66, 143.33. The increments being 14.38X and 1.47X

Comments on Audio Quality

With 101 taps, all four modes sound alright. There is a little bit of static present, so none of them sound as good as a normal FM radio, but that is to be expected. With 13 taps, the mono audio has a little more static than it does with 101 taps, but overall, it does not sound bad. The stereo audio on the other hand has much more static than it does with 101 taps, to the point where listeners are not able to make out any music being broadcast. With 301 taps the mono audio again has a little more static than with 101 taps but does not sound bad. The stereo audio again has much more static than with 101 taps, but it sounds slightly better than it did with 13 taps, as if listeners know what they are listening for, they can make out music being broadcast.

Proposals For Improvement

A feature that can be added to improve the project's user-friendliness is a user interface. As of now, the project is run from the command line using UNIX pipes to read I/Q data from the RF hardware and to feed the output audio data to aplay. To listen to a different station, you must exit the program and run the command again with the frequency of the different station as an argument for the rtl_sdr command. Instead, the SDR can be controlled using a graphical user interface, so the program doesn't have to stop and restart anytime a change is to be made. A user interface could be implemented using the tkinter library in Python, for example, and the same Linux command to run the SDR can also be called from Python using the os module. Command line arguments for the SDR can be set by inputs from the user interface and passed by Python.

A feature that would enable us to improve on the project would be reimplementing parts to better take advantage of the object-oriented nature of C++. Our current code uses vectors of floats for almost all data handling, which makes the code difficult to understand. Instead, we could create wrapper classes for storing blocks of data, and dedicated classes for storing filter coefficients and their corresponding state together in one object. This would simplify the way data is passed between functions and make the code clearer and more concise.

Project Activity

Week	Activity			
	Cass	Ty	Oscar	Ginoth
1		Briefly read the project document.		Reviewed the project document
2	Refactored lab code to C++, implemented mono path. Implemented	Worked on the convolution with downsampling function in C++ and spent some time	Same as Ty, added some timing stuff to show block process time.	Worked on refactoring lab code to C++ for slow implementation of mono path

	functions for handling stream I/O and path/mode selection.	debugging modes 0 and 1 in the mono path.		
3	Implemented functions for bandpass FIR generation and mixing. Refactored PLL in C++ and added state saving. Implemented stereo path with Ginoth.	Worked on the Python modelling and C++ implementation of the resampling functions and spent time debugging modes 2 and 3 in the mono path. Finished debugging mono path.	Same as Ty, did not contribute much to Python model. Finished debugging mono path. Read about stereo path highlevel	Worked on the implementation of the stereo path in C++.
4	Implemented threading and mutex-synchronized queueing for audio path based on lecture notes.	Implemented initial threading setup for the stereo path (which was later scrapped for a better implementation).	Missed this week	Worked on multithreading for the RF front-end and audio thread with Cass.
5	Implemented the channel extraction, carrier recovery, and demodulation pathway for RDS.	Wrote C++ implementations of the zero and peak detectors, Manchester and differential decoders, and frame synchronization / traversal, and the Python modelling and C++ implementation of matrix multiplication for the syndrome calculator (for frame synching).	Same as Ty	Refactored root raised cosine code in C++, started debugging RDS path
6	Implemented RDS application layer for PI and PT codes.	Added multi-threading support for the RDS path and working on debugging RDS path (we were not able to finish).	Debugging RDS data processing.	Worked on debugging multithreading issues for the RDS thread, and debugged issues in RDS implementation regarding segmentation faults and array indexing.

7	Report	Writing this report.	Report	Worked on writing the report

Conclusion

In conclusion, this project provided a great learning experience. It provided an opportunity for us to strengthen and consolidate the theoretical knowledge we had in digital signal processing by applying it to a real-life system. It also proved to be effective in teaching us how real-time systems are developed and how we can optimize our code for efficiency. Other programming techniques such as creating modular functions for functional blocks, visualizing data and modelling our program in a higher level language like Python were also helpful when working on a project of this scale.

References

Project specification:
https://avenue.cllmcmaster.ca/d2l/le/content/554053/viewContent/4569193/View

Multi-threading notes:
https://avenue.cllmcmaster.ca/d2l/le/content/554053/viewContent/4587842/View