Project Report: R-DSP Solution for Elevator Problem (Option #2)

EE3-19 Real-time Digital Signal Processing

Timo Thans {01668411}, **Tayyab Bhandari** {01334547},
Department of Electrical and Electronic Engineering, Imperial College London

I. Introduction

This report develops a DSP solution that will take an audio input and will detect when different specific sounds are heard. These sounds will be linked to the status of a lift:

- Arriving to a specific floor
- User pressing a button

We used an 8 kHz sampling rate which is an anti-alias filtered and downsampled version of the 44.1 kHz file. We were given a recording of a journey with little background noise, we used this to test the basic detection and very basic robustness against other sounds. Furthermore, we would indicate the detection of the sound by lighting up one of the four LEDs in the DSK board for 1 second.

Ultimately, the goal is to accomplish the three levels of achievement. Firstly, we have to detect when the user presses a button with very little background noise involved. Secondly, we will attempt to detect the lift arriving at a specific floor under relevant levels of background noise. Finally, our implementation will need to be robust enough to ignore fake noises and accomplishing this would mean we have successfully met the criteria.

In our design process we have followed three steps where in every step, we increased the complexity of our algorithm. We started with doing a simple cross correlation. The algorithm was then improved by adding a Fast Fourier Transform (FFT). In the end we tried to actively suppress noises by using estimate noise spectrum subtraction.

II. Survey of Potential Techniques/Shortlisting

The potential techniques discussed in the preliminary report included cross-correlation, filtering and goertzel algorithm, as well as energy estimation. Firstly, we were quick to make the energy estimation technique redundant which consisted of calculating the energy of a vector of samples and each time we do this with a new set of samples, a new value is obtained. We did not take into account how we would detect the sound by calculating the energy of a fragment of the signal, therefore in the given time period, this would have taken more effort to implement.

With the filtering method, the idea was to identify the specific frequency made when the button is pressed and use a narrow band pass filter to only pass this frequency to the output whilst suppressing all others. We would then do a comparison test to detect this frequency and subsequently flash the board during the process. However, this method was only potentially valid for the button press, not for the detection of the floor announcement. Also, the goertzel algorithm was essentially an alternative to the FFT and we decided a different approach to an already working method was unnecessary.

In terms of shortlisting feasible techniques we could implement, any method used would definitely benefit from a form of pre-processing or conditioning of the input signal to reduce background noise, if any.

Reaching a judgement on the most promising solution was determined by the computational complexity and ease of implementation, as well as effectiveness and the method with the best compromise was the cross-correlation. It was relatively easy to combine this with the pre-processing filter to get rid of any noise systematically appearing in the recordings outside the frequency range of the voice.

In terms of implementation, this would firstly require a transition into the frequency domain via FFT and we discovered that the most feasible way to do this in real time would be a frame-processing method with frame overlapping using the precompiled FFT library. Unfortunately, due to unforeseen circumstances we were not able to implement this on our hardware in C. Therefore extra care is taken to design an algorithm that could also be transferred to C and further elaborations are given where necessary.

III. Cross-Correlation

At first we evaluated the most easy to implement algorithm to perform a cross-correlation on the signal. The corresponding equation 1 explains the algorithm which is very efficient and fast but is obviously not very precise.

$$Corr_{x,y}[n] = \sum_{n=1}^{N} x[m]\dot{y}[m+n] > MIN\_VALUE \rightarrow "hit" \tag{1}$$

This equation combines a cross-correlation with a detection method consisting of a threshold to determine a 'hit' which means a pulse has been identified.

Again, since it was not possible to perform this task on the DSK board in real time, we chose to do the signal detection in python as an offline validation method, in a similar way as would be done in C. This similarity in implementation deserves importance as in Real-Time Digital Signal Processing, performance is critical. Writing the code in this way instead of

using *numpy* functions will make sure that we understand the underlying concept and will give us the freedom to improve the performance by, for example, implementing *cython* or rewriting the algorithm to C.

We evaluated our algorithm by looking at the speed and how it classified two sound samples of a typical elevator journey. The first sound sample had three beeps in it and the second

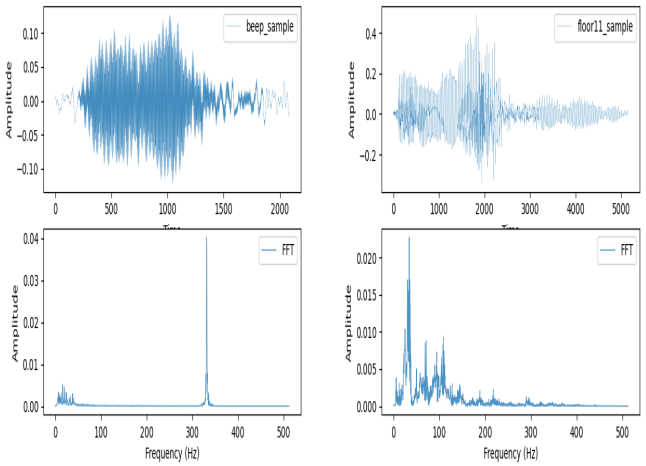

Fig. 1. Spectogram and FFT of beep sample

Fig. 2. Spectogram and FFT of *"floor eleven"* sample

sound sample had the announcer saying *"floor eleven"* and *"doors closing"*. The signals that are compared are stored locally. The Spectogram and FFT of the beep and announcer saying *"floor eleven"* are shown in figures 1 & 2 respectively. In figures 3 & 4 which show the cross-correlation algorithm working, in terms of an increase in amplitude when the cross correlation detects a pulse, we can see that it does a reasonably good job in classifying the beeps and *"floor eleven"* in the samples. However, the algorithm is slow, this is because we are continuously calculating the cross correlation, the compared signals are a few seconds long so we don't need this resolution.

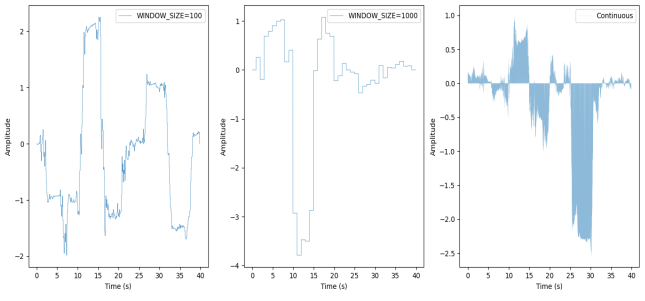

Fig. 3. Cross correlation algorithm with different window sizes performed on a sample with *"floor eleven"*

Instead it is better to use a window size which defines the resolution of the cross-correlation. As shown in table 1 this

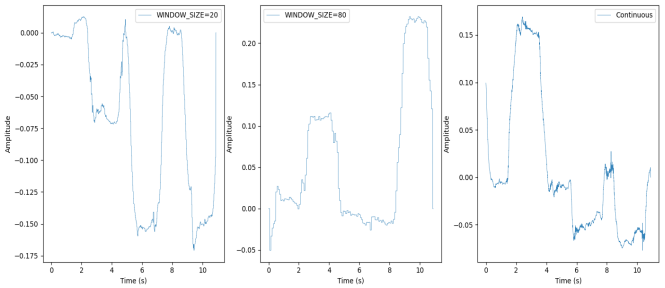

Fig. 4. Cross correlation algorithm with different window sizes performed on a sample with three beeps

improves the speed of the calculations significantly. The result of using different window sizes is also shown in figures 3 & 4.

Algorithm	Window size		Proc time	Error	
	Beep	Floor11		Beep	Floor11
Cross correlation	20	100	800 s	81/14	12/1
Cross correlation with FFT	400	1200	42 s	52/14	1/1
Cross correlation with spectral subtraction	200	1200	123 s	41/14	1/1

TABLE I
PERFORMANCE OF DIFFERENT ALGORITHMS ON THE COMPLETE JOURNEY WITH NO NOISE

Now that we have an algorithm that works reasonably fast we can test it on the *long journey* sound sample that we have received. We have used multiprocessing in python both to speed up the algorithm but also, considering the algorithm is implemented in real time, to have it listen for both the beep sample and the *"floor eleven"* simultaneously as shown in Fig. **??**. This multiprocessing can be done by starting two threads in python which separately calculate the cross correlation of the beep and the *"floor eleven"* signal. How this is done is shown in appendix A. We evaluated the performance of the sound detector by dividing the number of detection points by the true number of beeps and *"floor eleven"* announcements, also shown in table 1. The algorithm that performed best on detecting the signals is shown in figure 5. This figure shows the process of the long recording with no noise and displays what it detected.

A pulse is received whenever a beep sound or *"floor eleven"* is detected shown as red or blue pulses respectively. This was the algorithm with window size 80 and 1000. It is however, important to note that the results of the algorithm for different window sizes are slightly arbitrary since an important factor is the minimum value size. It is still valuable information since it shows us that a larger resolution (smaller window size) results in several false positives, as shown in the figure.

Providing we could implement our algorithm with the hardware, we would switch LED #2 on to mark the detection of the beep sound for 1 second and this would require a timing operation. Similarly, LED #3 would be used for the detection of *"floor eleven"* on the DSK board. This

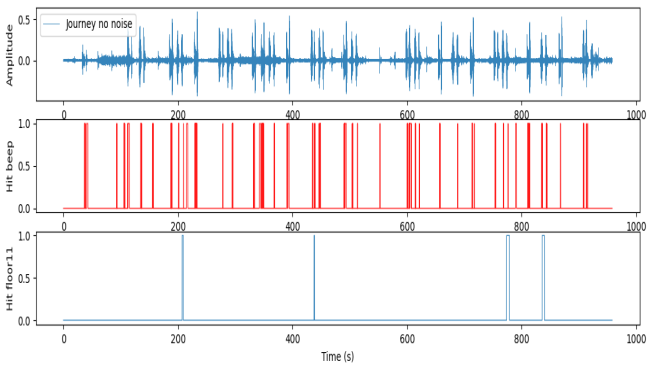

Fig. 5. The graphs show when the algorithm detected the sounds during the journey

would be done by initialising the LED functions by running **DSK6713_LED_init()** and the LEDs are switched on by using the **DSK6713_LED_on()** command.

IV. Cross-Correlation with FFT

As seen above the algorithm struggled to separate beeps from other noises thus leading to several false positives. One of the suggested improvements is to perform a FFT before doing the cross correlation. The algorithm is very similar to the one used in the previous section only now, we will first perform a Fourier Transform on our *"beep"* and *"floor eleven"* samples. The computations done by the algorithm will inherently be more complex due to the FFT but since it does a much better job at classifying the samples, we can increase the window size which will make the algorithm faster as shown in table 1. Figure 6 shows the results of this updated algorithm. The

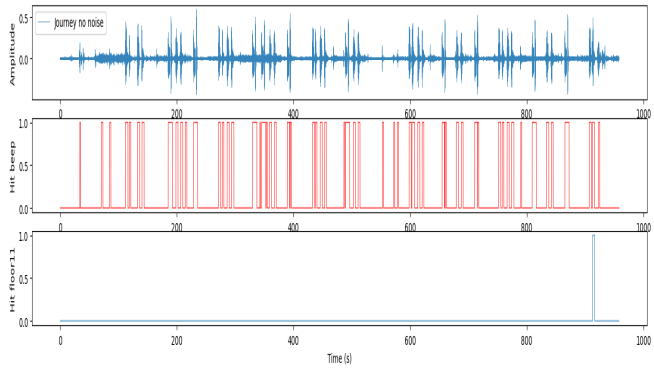

Fig. 6. The graphs show when the algorithm with FFT detected the sounds during the journey

used FFT function is from *scipy* in python. The advantage of using this function is that many people have worked on this and have spent a lot of thought into making the function as efficient as possible. This is similar to C where we can use the FFT function from the FFT library as will be discussed later as well. We can clearly see that using the FFT function gives a much better performance when evaluating both processing time and classification error. There is however one problem, especially in the beep classification. Since the beep in the frequency domain is only one spike, it is susceptible for false positives. It practically classifies every loud signal as a beep. One way of solving this problem is by using estimate noise spectrum subtraction.

V. Estimate Noise Spectrum

As the name suggests this technique subtracts background noises in the frequency domain. This will work really well when there are constant background noises. It unfortunately does not do much when there are random loud signals. This is the problem that we experienced when classifying the *"beep"* signals. The lift announcer is too loud, to the extent that the algorithm classifies the announcer speaking as beeps.

Therefore a small adjustment in implementing the estimate noise spectrum algorithm can be to take people talking as sample noise and take the Fourier Transform of this sample. Subsequently, it is possible to again take the cross-correlation with the FFT of the input. We now are able to detect when people are talking in the elevator. If we subtract the result of this cross correlation from the result of the cross correlation with the "beep" sample, we are able to filter out both the announcer and people talking in the elevator. The results of this algorithm are shown in table 1 and how the algorithm classified the *"beep"* and *"floor eleven"* signals is shown in figure 7. Now that we have used a noise filtering technique we can also

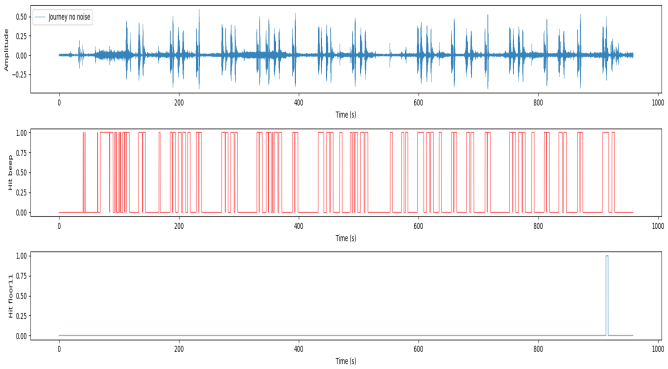

Fig. 7. Detection of *"beep"* and *"floor eleven"* during the journey

try and test our algorithm when there are people talking in the elevator, these results are shown in figure 8. It can be seen that the estimate noise spectrum subtraction technique indeed works very well when there are constant background noises. It is however much harder to classify the *"floor eleven"* as the announcer is also partly filtered in the process.

VI. Implementation in C

The largest difference between python and C will be the way we do signal processing in the frequency domain. We had to decide the best way to perform processing in the frequency domain in real time on the DSK board which consisted of taking the Discrete Fourier Transform (DFT) of the signal, removing frequency content at frequencies in our required stop band (i.e. set values to zero infrequency bins we do not wish to

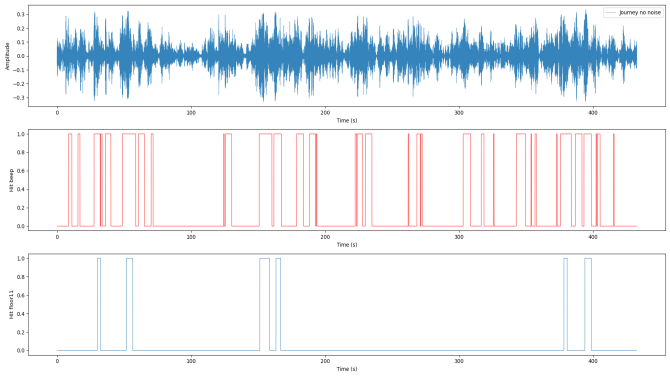

Fig. 8. Detection of *"beep"* and *"floor eleven"* during a noisy journal

let through the filter) and performing the Inverse DFT (IDFT) of the modified spectrum. The following method outlines how to perform the FFT using frame processing in C to be used on the DSK board, although keeping in mind that this method could not be tested due to external circumstances.

We decided to use a technique called triple buffering which uses three buffers: an input buffer which fills up as we sample the input, an intermediate buffer where we do the DFT on the data and frequency domain processing as well as an output buffer from where we write the output samples. Frame processing is necessary for a DFT implementation, each output sample uses a frame of data to get a short time Fourier Transform, we therefore process data in frames similar to the way we have used a window size before in python.

Each time a McBSP Transmit interrupt is generated, we always need to output another sample, and therefore also read in another sample, this will allow processing in real time on the intermediate buffer. However, at some point the input buffer will be full and all the samples in the output buffer will have been output, buffer rotation must then be performed. To implement this in C, the ISR function needs to input and output a sample at each sample instant and update the buffer index which is shown in listing 1. The frame buffer length must be even for a real FFT.

```
void InteruptFunc(void)
{
short sample;
float scale = 11585;
sample = mono_read_16Bit();
// add new data to input buffer
input[index] = ((float)sample)/scale;
// write new output data
mono_write_16Bit((short)(output[index]*scale));
// update index and check for full buffer
if (++index == BUFLEN)
    index=0;
}
```

Listing 1. ISR for Frame Processing

The input index variable is updated at the end of the ISR and this has to become zero in the main function before passing the input buffer to the processing stage. The main function cannot begin processing data until the input buffer is full as we need a full frame of data to do a DFT.

When the input buffer is full the program should pass the input to the intermediate buffer for processing, pass the processed intermediate buffer to the output buffer for outputting and pass the output buffer to the input buffer to receive the next inputs. The program to be used is shown in listing 2, this function handles buffer rotation and processing while the ISR deals with input and output of individual samples.

```
void triple_buffering(void)
{
float *p;
// wait for array index to be set to zero by ISR
while(index);
// rotate data arrays
p = input;
input = output;
output = intermediate;
intermediate = p;
// wait here in case next sample has not yet been
    read in
while(!index);
}
```

Listing 2. Buffer Rotation

This function will loop indefinitely in the main function, waiting for interrupts. We also wanted to use a precompiled FFT library for transition into the frequency domain as this would allow a relatively easy implementation in C. The object file *fft_functions.obj* is an existing routine and contains optimized code to perform a complex FFT on frames of data. *fft_functions.h* is the corresponding header file to be used in the C program and the function definition is written as *void fft(int N, complex *X)*. *Fft(N,*X)* calculates the FFT of its input buffer X. The parameter N contains the number of data values in X. The output spectrum is returned by overwriting the input buffer X. This would be incorporated in the *triple_buffering* function after rotating the buffers to perform a FFT on the data stored in the intermediate buffer. Furthermore, the *cabs()* function is used to calculate the magnitude of the resultant spectrum.

VII. Conclusion

To conclude, the ultimate purpose of the project was to have the algorithm running in real time in the DSP therefore constraints of the DSP such as memory and calculation times would have to be taken into account. However, the quality of this report was somewhat compromised in this respect as we could not discuss and test our algorithm on the DSK board in C implementation due to reasoning beyond our control. Therefore, the implementation of our algorithm in C code has not been greatly talked about, although some parts have been outlined.

In terms of the performance of our algorithm, if time and circumstances permitted, further improvements could be made. An important factor for improving our algorithm is the minimum value size and noise filtering. Corresponding

improvements could be made in python with relative ease, for example by using a peak detection method or fancy filtering techniques. On the contrary, it is important to not get carried away and be realistic with what we can implement with our level of skill in C on a DSP. The algorithm that we have come up with would be directly transferable to C with our current knowledge of R-DSP systems.

APPENDIX A
PYTHON CODE

This appendix shows the code that is used to classify the ”beep” and *”floor eleven”* by making use of cross correlation, fast fourier transform and estimate noise spectrum subtraction. As the file is quite large a link to our github directory is given. *https://github.com/TimoThans33/RTDSP*