Name of student: Natanael Gil Vidal

Academic year: 2009/2010

Number of project: 93

Title of project: Noise Cancelling Stethoscope

Name of first supervisor: Dr. Fernando Rodríguez

Name of second supervisor: Prof. A. Catrina Bryce

Noise Cancelling Stethoscope

Natanael Gil

2009/2010

Contents

Chapter 1

Introduction

The objective of this project is to design a *Noise Cancelling Stethoscope*, this is, a stethoscope that will eliminate all the sounds that are not from inside the body. To achieve this, the method employed will require one sensor (microphone) to capture the body sound and other one to capture the ambient noise; after some digital processing, the sound of the first sensor will be cleaned and the output through the earphones will be only the sound of heart, lungs... without any external interference.

Combined with a good insulation in the ”ear-tip”, this stethoscope can be used in noisy environments where it is difficult for the nurse or doctor to discriminate the body sounds from the ambient sounds.

Chapter 2

Background

Before start working directly in the project we need some background about what we want (this is, what a medicine professional would expect of the product) and how we want it and how we can obtain this (the technical basis that is needed to achieve our goals).

2.1 Medical

"The stethoscope is an acoustic medical device for auscultation, or listening to the internal sounds of an animal body."[1] Usually it is used to listen the sound of heart and lungs, but sometimes also intestines and blood flow are listened.

Before the invention of the stethoscope, doctors used to listen the sounds of the body directly putting the ear in the patient, but in 1816, Rene Theophile-Hyacinthe Laennec because he felt embarrassed with touching directly the body, so he took a wooden "trumpet" and he discovered that was much easier to hear the sounds in this way.

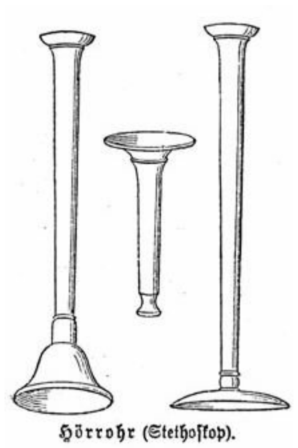


Figure 2.1: Stethoscopes [2]

After this, came stethoscopes made of rubber, bin-aural (with two ear pieces), improved acoustics, electronic instruments... [3]

Nowadays, acoustic stethoscopes are the most used ones, the sound is transmitted through tubes filled with air; electronic stethoscopes (also called stethophones) are a bit more complicated but they have more possibilities. The easier way to take the sound is placing the microphone directly over the patient, but then external noise interferences will appear, so manufacturers design their own methods to listen the sound of the body (piezoelectric, electromagnetic diaphragms...). Apart of processing the sound (amplify, filter it...) as it is transmitted electronically wireless transmission can be used to an external device (heavy signal processing, graphic representation, record the sound, telemedicine, etc.) [1]

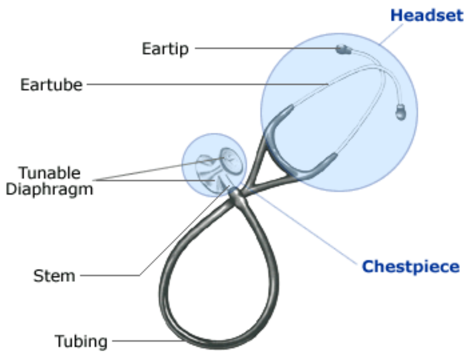


Figure 2.2: Parts of a stethoscope [4]

Current stethoscopes have the possibility of filter the body frequencies using the *bell* or the *diaphragm*, one side acts like a high-pass filter that allows the listener to hear clearly higher frequencies that can be overlapped by stronger lower sounds.I've been suggested the possibility of implement that option also in the code (and in hardware would be activated through some kind of switch), but considering that in our system the speed is a critical parameter, I think that is much better to leave that filtering in an analog way to avoid any extra delay.

As the end of this product is, mainly, medical application in a, more or less, controlled environment (we can know most of the situations where is going to be used), we can optimise it to fit the needs.

The stethoscope is a light and portable instrument, so our design needs to achieve the same requirements: the "heart" of the product will be a microprocessor, DSP... something small that can be used with comfort. As is portable we also need it to work with batteries (small batteries), so the power consumption has to be low, this is achieved by choosing the right IC, making a good design, keeping the algorithm as simple as possible, if it needs too much processing operations, we will need to increase the speed (the system also needs to be fast, real time if possible) and our energy need will also rise, this is why LMS algorithm is chosen for the filter (this will be explained deeply in the next section); and the amount of data has to be just what is required to work properly, too little the system won't be effective, too much and the system won't be efficient.

In this case, the amount of data is determined by the *resolution* and the *sampling frequency*. Resolution determines how many values can we represent inside of our working limits: higher resolution, more values (and more processing time), I haven't be able to develop this part of the specifications, but the objective would be to find the minimum value that is enough to represent and process the signal without much distortion. Sampling frequency is the number of samples that we take per second, the human ear can hear to about 20kHz, so to have everything we should sample at more than 40kHz and our clock should go much faster to allow our system to do all the operations needed in the time left in between samples. But as I said before, the product will be in a controlled environment, it will be used for listening the sounds inside the body, so we can research what are that sounds and its frequencies because we know that higher frequencies won't be of anything "interesting" happening in the body.

Doing some research I've found the maximum frequency generated by an adult human body (to be listened with an stethoscope, e.g. blood circulation also generates sounds, quite high frequency sounds, but is not something the we want to listen in this situation) is about 1kHz. To be more sure about that values, the best thing to do is to analyse the different sounds that we want to listen, as it was not possible to acquire this samples on "live", after a bit of web researching I've found a site [5] designed to teach Medicine and Nursery students to know what they should search when auscultating patients, the quality of these samples is not very good but is enough to our purpose (in order to have more accurate and clean sounds, better quality packs can be bought).

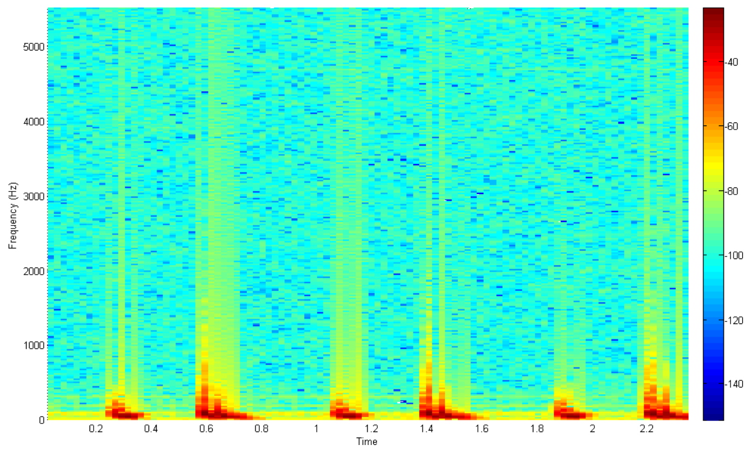
As seen in figure 2.3 almost all of the power of the samples are around 1kHz, some of them just a bit above that, so we can take a safety margin to 1.5kHz, in this case the sampling frequency will be *3kHz*. Probably the stethoscope will be used with babies and children, in this case the sounds will be of higher frequencies, we should do some extra research and find out what are the limit frequencies in this case (I've tried to search for this data but without results, at least "free" results). When we have also this information, instead of raising the sampling frequency all the time, I would choose to add a switch that will allow the user to use a higher frequency if a child is auscultated; except if he or she is working with babies or children (pediatricians, neonatal department, etc.) most of the time will only treat with teens and adults so will be an easy way of saving battery.

2.2 Technical

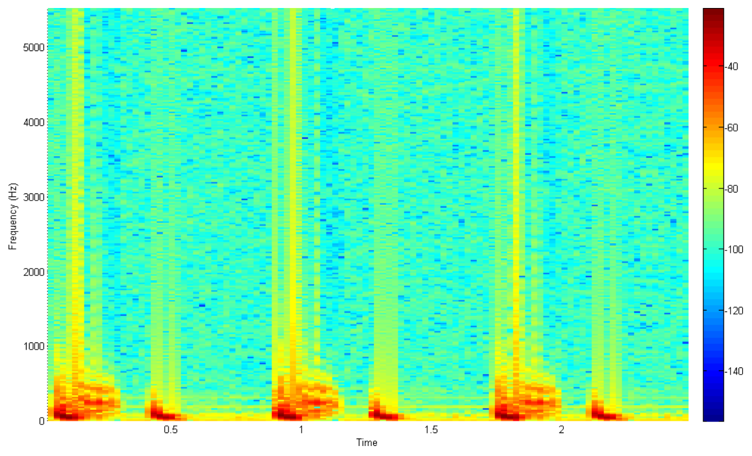
2.2.1 Digital signals

The main difference between analog and digital is that when we are working with the first type of signals, we have data during all the timeline, infinite points; but when we want to work with a digital signal the amount of data that we can handle and process is limited, what we do is to take one sample each certain period of time, doing this instead of talking of time, we talk about "number of sample"

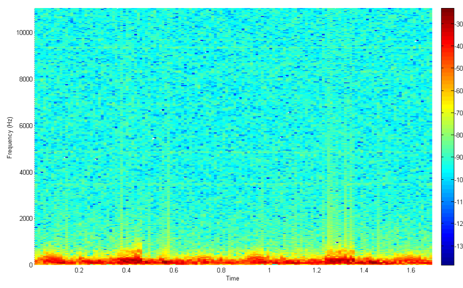
Other useful way to represent the signal is, instead of *time-domain* use *frequency-domain*, this is analyse it and show what frequencies are in it. With this representation is easier to modify and work with the signal. To change our working domain a powerful tool is used: *Discrete Fourier Transform* (and its inverse).



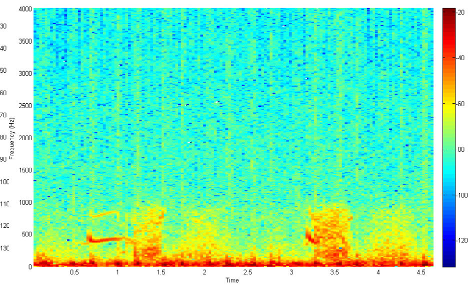
(a) Normal Sinus Rhythm



(b) Innocent Flow Murmur



(c) Friction Rub



(d) Wheezes

Figure 2.3: Some analysed examples of heart and breath sounds. Higher resolution and more examples can be found in the attached CD

2.2.2 DFT

Discrete Fourier Transform is based in the Fourier Transform (2.1) used in the analog world, but how this project is going to be developed entirely in the digital

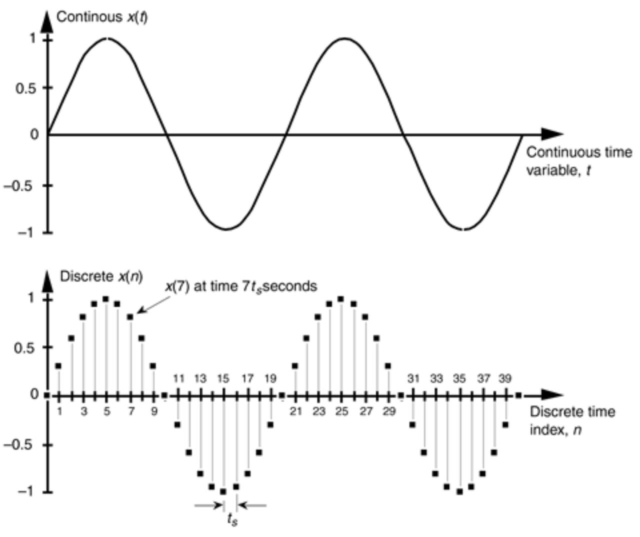


Figure 2.4: Difference between continuous and discrete signals [6]

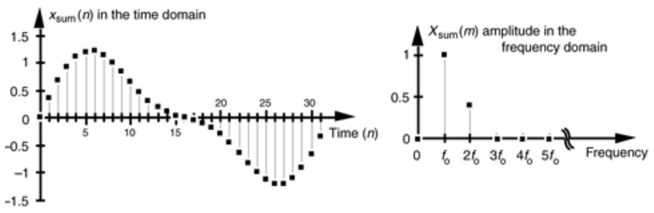


Figure 2.5: Example of frequency domain representation [6]

domain I'm going to talk only about the discrete one (2.2).

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-2j\pi ft}dt \tag{2.1}$$

$$X(m) = \sum_{n=0}^{N-1} x(n)e^{-2j\pi nm/N} \tag{2.2}$$

$$X(m) = \sum_{n=0}^{N-1} x(n)[\cos 2\pi nm/N - j\sin 2\pi nm/N] \tag{2.3}$$

Thanks to *Euler's relationship* the exponential form (2.2) can be written in a rectangular form (2.3) that will show a different point of view. First of starting to understand how to work with this equation, we need to know what is each component:

- N is the number of input samples and number of outputs.
- $X(m)$ is the output, being m its index (0 to $N-1$)
- $x(n)$ is the input sample

We just need to fill the values and we'll obtain one complex value for each output index. To know the frequency of these terms, we will use the following equation:

$$f_{analysis}(m) = mf_s/N \tag{2.4}$$

As said before, the output for each frequency is a complex value, so it will have two components, and we will able to know what are both the magnitude of determined frequency and it phase. Also its power $X^2_{mag}(m)$

One important property of the DFT is linearity

$$X_{sum}(m) = X_1(m) + X_2(m) \tag{2.5}$$

Thanks to this property we are able to analyse "real" signals, not only "pure sinewaves".

Another useful property is what is called *shifting theorem*

$$X_{shifted}(m) = e^{j2\pi km/N}X(m) \tag{2.6}$$

If we know that the data that we are looking have a phase of k samples, we still can obtain the correct results.

IDFT

The DFT is to change from time-domain to frequency-domain, if we want to do the inverse way, a very similar equation is used, it's called *Inverse Discrete Fourier Transform*

$$x(n) = 1/N \sum_{m=0}^{N-1} x(n)e^{-2j\pi mn/N} \tag{2.7}$$

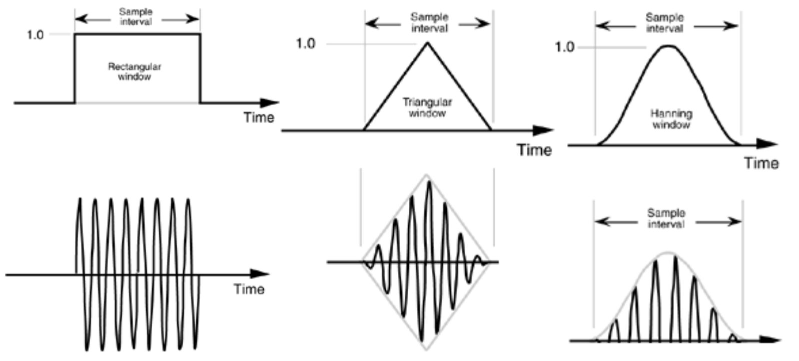
$$x(n) = 1/N \sum_{m=0}^{N-1} x(n)[\cos 2\pi mn/N - j \sin 2\pi mn/N] \tag{2.8}$$

Using the frequency components in the exponential equation (2.7) or in the rectangular one (2.8), the result will be the magnitude (and phase) for each sample.

DFT leakage and windowing

Outside of the theory domain, things become a bit more complicated that explained before, the working conditions cannot be controlled to have a perfect analysis. For example, we are analysing certain signal at a sampling frequency f_s, when we do the DFT, our results are only for nf_s/N (being n positive integers) but in the real world, we have more complicated things than that, so the signal will have some intermediate frequencies, but instead of just being invisible for us, this component will be in all our outputs, this is called *leakage*.

To fix this problem, the first thing that we can think is increase N, the number of output to read, but this is not always possible due processing, handling data... limitations; and to be totally sure that all frequencies are taken in account N should be infinite.



(a) Rectangular window (b) Triangular window (c) Hanning window

Figure 2.6: Some examples of windows [6]

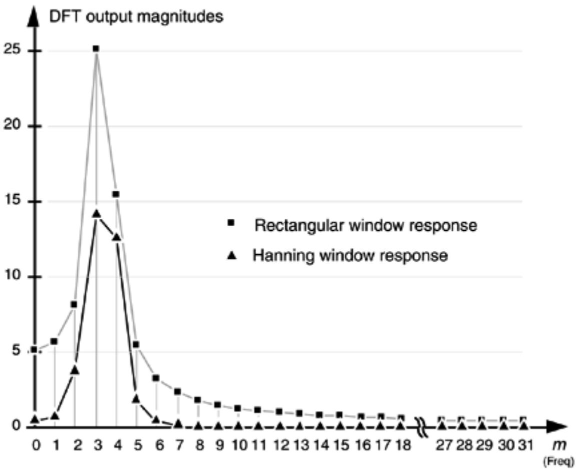


Figure 2.7: Effect of a Hanning window [6]

So the method used to reduce this leakage is what is called *windowing*, that consists in changing the shape of each sample interval and is done easily through mathematical expressions applied to the samples before doing the DFT.

In figure 2.6 are some of the most used windows, each one has different properties and, depending of what we need to watch (narrower results, more accurate magnitude...) and we will choose the most adequate for each system.

FFT

As we can imagine, DFT is a *very inefficient* process to analyse a signal, when the points of the DFT are increased, the data to handle becomes huge. So some faster methods were developed (and still working on them), they are called *Fast Fourier Transform* (FFT), these are algorithms that obtain very similar results to standard DFT but with much less processing. Nowadays, the most common is the *radix-2 FFT*.

2.2.3 Filters

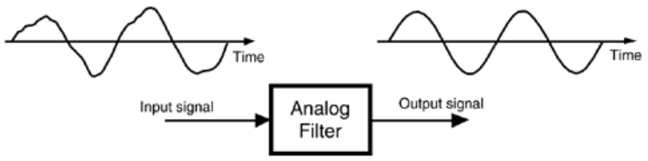


Figure 2.8: Filter [6]

The signal processing is mainly based in what are called *filters*, we take a signal and we modify it with a filter. Depending of the result, some common types of filters are *lowpass* (the output will only have the lower frequencies of the input signal), *highpass* (only higher frequencies), *bandpass* (only will contain certain frequencies in between a high limit and a low one)...

Other types of filter, classified by *how* they work, are *Finite Impulse Response* (FIR) and *Infinite Impulse Response* filters (IIR). FIR filters are simple and stable while IIR filters can be more accurate but they are more unstable and may need a big computational load. Here only will be explained the first one, because the one that is needed in this project.

FIR filters

One of the main properties about this type of filter (also called non-recursive) is that it uses a limited amount of data, only *past and present* inputs.

Operation of his filter is basically, multiply the value of k samples by certain coefficients h and then add the results.

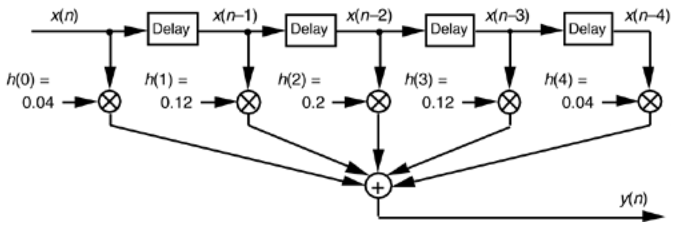


Figure 2.9: How a FIR filter works [6]

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k) \tag{2.9}$$

- M is the size of the filter, the number of *tap*, how many values are used.
- n is the output index.
- $h(k)$ are the filter coefficients, k is the index.
- x are the input values.

This operation receives the name of *convolution*, and can be very simplified through DFT as shown in equation (2.10)

$$y(n) = h(k) * x(n) \underset{IDFT}{\overset{DFT}{\rightleftharpoons}} H(m)X(m) = Y(m) \tag{2.10}$$

To design a filter we ”only” need to calculate the coefficients, in broad outlines this can be done following the next steps:

- Define what is going to be the response in the frequency domain (in order to obtain better results, windowing will be used).
- Convert that expression to the time domain (IDFT) to get the filter coefficients.
- Apply the filter (convolution).

There are also software routines that allow us to do that with a lower possibility of making mistakes.

All the information explained until this point (Digital signals, DFT, filters...) has been studied and learnt from [6]

2.2.4 Adaptive filters

If the conditions that the product would be always exactly the same, we could use the filter described before, but obviously this is no going to happen, so we need something that allows us to work in changing non predictable conditions.

This is done withe *adaptive filters*, filters that instead of having some fixed parameters, they adjust them by themselves.

The filter chosen to do this project is called *Least Mean Squares* (LMS) because is a stable and very simple filter, so we don't need much computation to achieve the results.

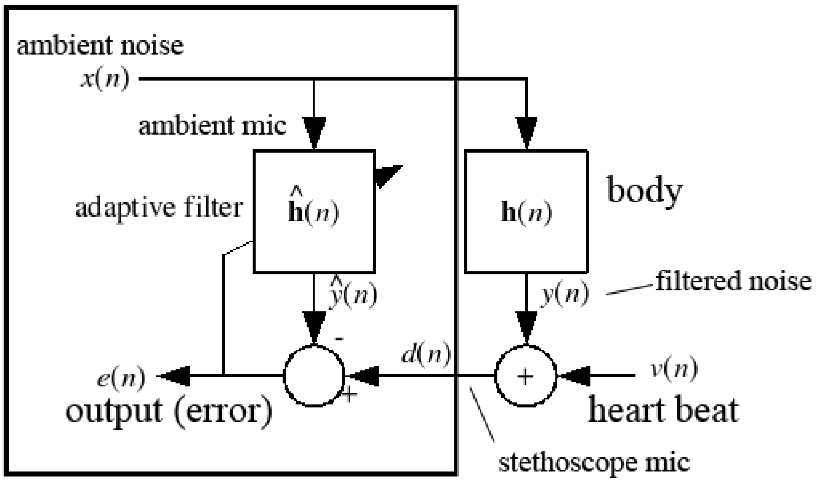


Figure 2.10: LMS filter [7]

This filter is basically a FIR filter with changing coefficients, each time that we apply it we need to update them by an algorithm.

I don't have enough background (and I haven't had enough time to acquire it) to understand how the LMS algorithm works, so I'm going to write the standard approximation commonly used and explain the parameters

$$\hat{h}(n+1) = \hat{h}(n) + \mu e * (n)x(n) \qquad (2.11)$$

- h are the coefficients, now called *weights*.
- e is the "error" as seen in the picture, the sound of the stethoscope minus our modified sound.
- x is the noise taken from the ambient microphone.
- μ is the step size, the "speed" that the filter uses to find the optimum weights. A high value means more speed but poor accuracy, and a lower value is slower but with a better results.

LMS filter tries to minimise the difference between the both input modifying one of them. Without the sound of the heart (interference), and with a perfect filter, the output would be zero.

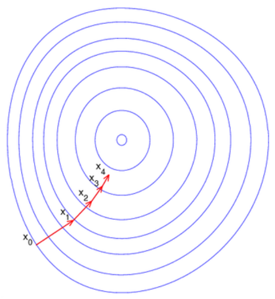


Figure 2.11: With high values of μ the filter will be far from the desired result [8]

I μ is too high, the system can become unstable, to reduce this, the $NormalisedLMS$ algorithm can be used.

$$h(n+1) = h(n) + \mu e * (n)x(n)/p \qquad (2.12)$$

The only difference with LMS is to divide by the power of the samples that are being handled, following notation in figure 2.10 x^2.

Chapter 3

Design and Implementation

3.1 Design and simulation of a prototype

$$\begin{tikzcd}
{[Read\_Data]} \arrow[r] & {[Prepare\_Data]} \arrow[d] \\
{[Calculate\_New\_Coefficients]} \arrow[u] \arrow[d] & {[Apply\_Filter]} \arrow[l] \\
{[Write\_Samples]} &
\end{tikzcd}$$

This is a general idea of what is needed to be done.

For the first practical approaching to a prototype, GNU/Octave has been chosen because is a software that provides an easy way to work with digital signals: matrices (in our case matrices of order nx1) and processing...

```
[dirty,fs] = wavread('file1.wav');
[noise,fs] = wavread('file2.wav');
```

Before of all we need to read the files that we want to process, this can be done with the command *wavread*, in my case I'm reading the audio data and also the sampling frequency, this is because by default Octave will consider that the frequency is 22kHz.

```
u = noise(n:-1:n-M+1);
```

As the filter is based in a FIR filter, our *weights vector* must have their values in inverted order, so, our next step will be to fill this vector in that way. Octave allows do this with one simply adjustment in the function used to extract a part of a vector: we can take set the direction (and period) that the data will be read (e.g. 1 will read in the standard way, 2 will read each to values, -1 will reverse the values).

This is not completely necessary but it will help to simplify the operations to be done.

```
yn = w' * u; %Applying filter
e(n) = d(n) - yn; %Output
```

```
p = (u'*u);
w = w + (mu * u * e(n))/(p); %Normalized LMS
```

With *GNU/Octave* we can apply filters easily with the *filter* function, or with the *convolution* one, but in this case I've considered appropriate using something that can be ported to a different language more easily. As we have prepared the values, we can just multiply the transpose of the matrix containing the *weights* by the one containing the *input samples* and the result will be a scalar number (we are working with matrix of order 1xn)

The next step will be calculate, and this is done only calculating the difference of the signal that is listened directly in the body with the processed signal, the *error* of the filter.

And finally, before read more new data, we need to update the coefficients, as Octave allows working easily with matrices, we only have to write the equation (2.12), but how in this project only real signals are going to be used, can be a bit more simplified as in (3.1).

$$h(n+1) = h(n) + \mu e(n)x(n)/p \tag{3.1}$$

As we know high values of μ lead give a fast approaching to the expected result, but lower values are more accurate, so one way to have a bit of both is use a variable μ, for a few samples we use a high value (because we are using the Normalised algorithm, the filter will be stable even with a very high value) and after that we change it for a lower one.

```
soundsc(e,fs);
```

Once all the file has been processed, GNU/Octave allows to listen a vector directly, this is made through the *sound* function, as reading (and writing) .wav files by default it uses 22kHz as the sampling frequency, but we can set our own one. There is another interesting command *soundcs* that plays the sound at the maximum volume possible without distort the sample.

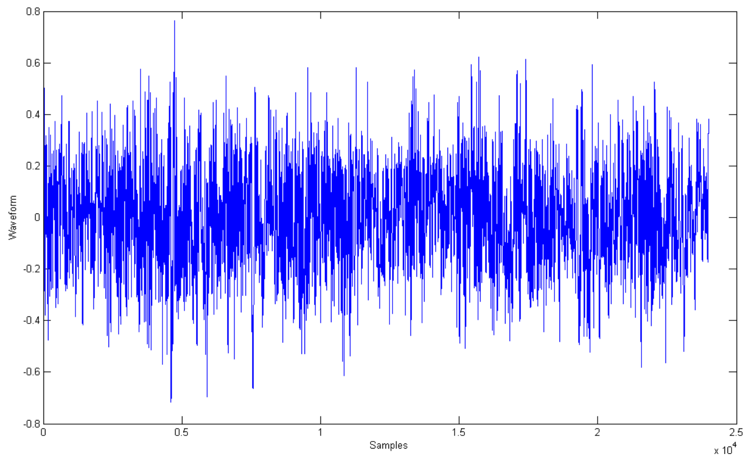
Now that the prototype has been designed and is working with some random parameters we need to choose them:

μ can be a high number because we are working with the *NLMS*, so, I'm going to take first a quite high value, 0.5 but then it will be lowered to reach 0.01.

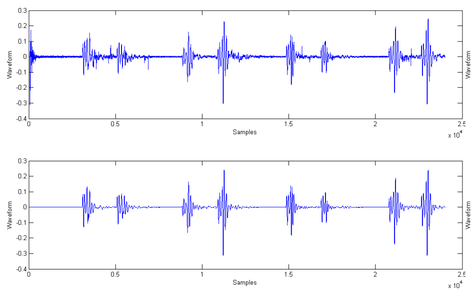
The other key parameter for this type of filter is the order, we need to find the equilibrium between good performance and not too much processing (high order), so I started testing the filter with high values and then lowering until find this point.

To test the filter, first I found some samples of heart sound [9]. For the noise I took various random sounds (sound, music, voices...) and mixed all them together, this will be the *ambience's noise*. Then I filtered this sound with a lowfilter of 150Hz (after some research I found that this was, roughly, the frequency that the body filter the outside sounds) and added the sound of the heart, this will be the sound taken by the *stethoscope* in the body.

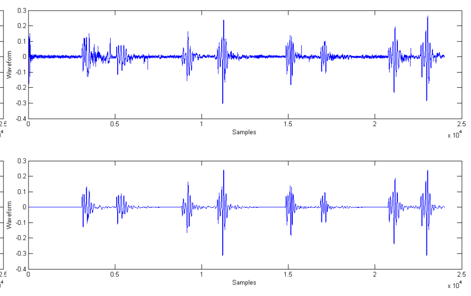
As we can see in figure 3.1, 16 taps is too low but we can take some point before the 32 taps. I found that a good equilibrium point was at 24 taps; in figure 3.3 we can see that the adaptation is achieved quite fast, in less than 250



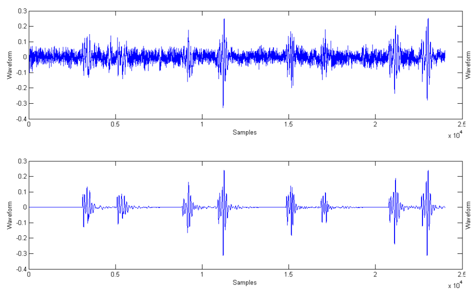
(a) The sound before being processed



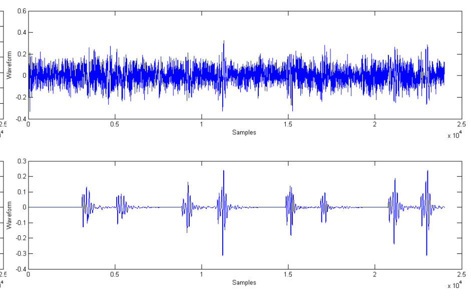
(b) 64 taps



(c) 32 taps



(d) 16 taps



(e) 8 taps

Figure 3.1: Response of the prototype to different orders for the filter

samples; this sound files are at a sampling frequency of 8kHz so, the adaptation takes less than 50ms.

3.2 Software Implementation

The next step in the project is to implement a functional software prototype to test the algorithm, the chosen language has been C, that can use in microprocessors, so if we want to implement this code directly in hardware, not big changes will be required. In order to have an easier comprehension, and test if

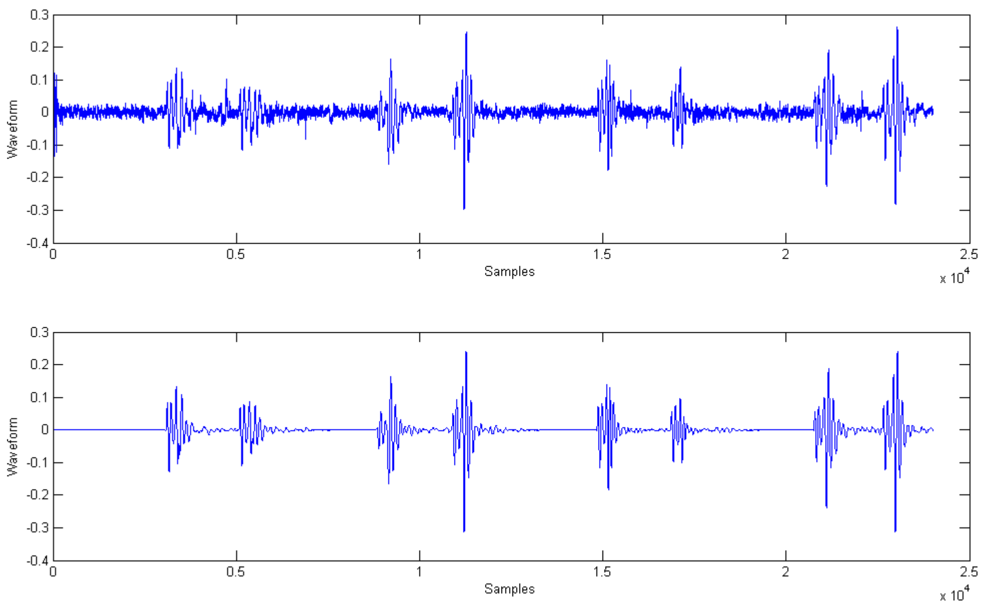


Figure 3.2: Filter response at 24 taps

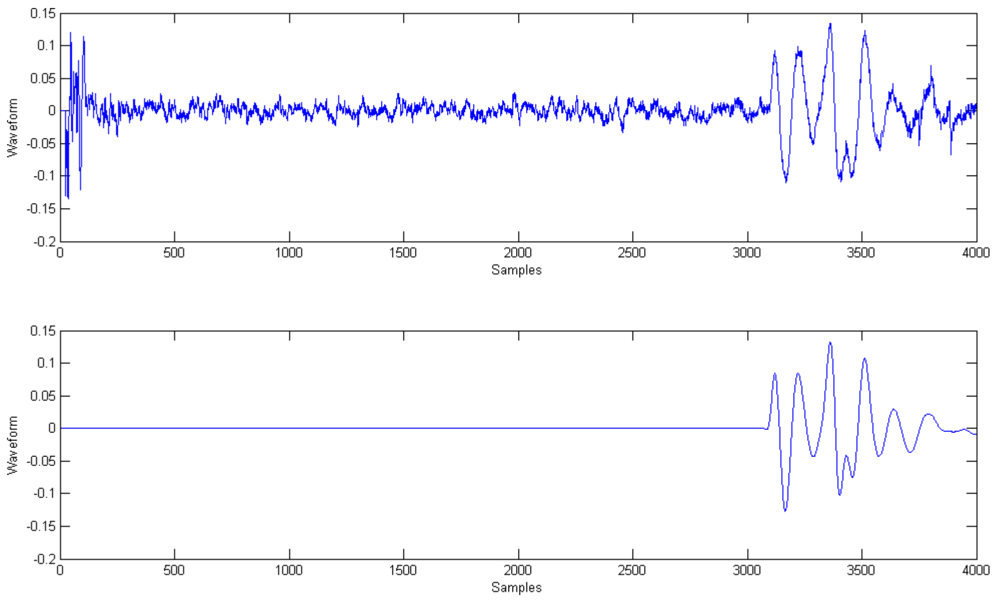


Figure 3.3: Close look to filter response at 24 taps

it is working correctly, first I have used floating point in the operations of my application.

One of the main differences with the GNU/Octave code is the way to handle the .wav files. In GNU/Octave we load the whole file in one vector, we read the values of that vector, save them in a different vector after processing them and finally write that data in a file (or play it directly). If we want to do this with

C, we would need to have a huge amount of memory available and the main problem, our perspective for this code is to implement it in hardware to work in real world, and in this situation we only have access to past (if we store them) and present samples, so to have something more similar of how will be our final application, we are going to read the samples directly of our file and write the result directly in another one.

```
noiseFile = fopen("file1.wav","rb");
dirtyFile = fopen("file2.wav","rb");
outputFile = fopen("output1.wav","wb");

char buffer [44];
fread(buffer, 1, 44, noiseFile);
fwrite(buffer, 1, 44, outputFile);

fseek(dirtyFile, 34, SEEK_SET);
fread(&bs, 2, 1, dirtyFile);
Bs = bs / 8;
fseek(dirtyFile,44,SEEK_SET);

while (!feof(dirtyFile)) {
    fread(&readInputBody, Bs, 1, dirtyFile);
    fread(&readInputAmbience, Bs, 1, noiseFile);
    ...
```

C allows to access files in binary mode through the process *fopen*, we need to indicate in the parameters if we need to access that file in read, write or both modes.

As we are reading the files in binary mode, we need to know how is their format. Looking at the specifications we see that .wav files have a header of 44 bytes. This is a program to be executed on an controlled environment: our two files will be of the same length and, therefore, the output will have the same length so we can use the header of any of the input files to create the output one. Two important values for us in that code are the *sampling frequency* and the number of *bit per sample*, in this case we only need to read the number of bits per sample (we have to know where start and finish each sample). We can do it with C in an easy way, with the *fseek* command that moves the file pointer to determinate byte. To read bits the command *fread* is used, apart of the file and where save the read data, we need to indicate the number of bytes per block, and the number of blocks to read.

The filter cannot work till it has enough values to handle, so this first samples are just stored and copied to the output file, when the vector has enough data, the filter start working. As in the GNU/Octave code, I have three different values assigned to μ, to change them, I'm using a counter that stops working once that the third μ value is used.

To store our sampling values we need a FIFO structure, this is the first value that we have introduced will be the first in be deleted. The first idea was to create and use a queue, but after considering that maybe that would complicate significantly and maybe wouldn't be the most efficient solution (only is needed for this variable) I decided just using a vector and moving all the values each time that new data is incoming. Also storing the values is made in the reverse way than used in the weights vector.

```
yn = 0;
for (i = 0; i<ORDER;i++){
    yn += w[i]*u[i];
}
```

In Octave we cannot handle directly vectors as matrices, so to apply the filter (to do the convolution), we need to multiply each single value of the vectors, as we have prepared one of the structures before to be filled in a reversed way, we can do it all straight, without intermediate operations.

```
e = calcBody - yn;

if (e > 1) {e = 1;};
if (e < -1) {e = -1;};
```

After obtaining the error (the output) we check that the result is inside our working limits, in case of being out of them, we just saturate the value, working on this way, we always will have control on what is happening.

```
power = power*(ORDER-1)/ORDER + u[0] * u [0] /ORDER;
```

As we are using the *Normalised LMS* algorithm, we need to calculate the power of the part of the signal that we are handling, this requires a lot of processing, in each iteration it need to multiply and add N times (being N the order of the filter), one way to relax this requirement would be to store all the values that are being used to work with the filter and in the next iteration add the new one and rest the older one, but this needs a quite big memory... so the better choice would be to use what is called *running average* (also know as cumulative moving average).

I've been trying to implement it in my code, but only with wrong results. I think that the reason for that is that in this case the number are very different ones from the others, so this approximation is no valid. As when I change to fixed point, this is no needed, I decided to leave the operations in the "theoretical" form (multiplying all the numbers one by one).

```
for (i = 0; i<ORDER;i++){
    w[i] = w[i] + e*mu*u[i]/power;
}
```

Next step is to update the weights, again we need to iterate to add all the terms of the vectors.

Last operation before reading the next values of the files is to write the output the file, the process *fwrite* is used, using the same parameters than *fread* before.

3.2.1 Fixed Point

As a next step towards the hardware implementation, instead of using floating point that is the easier way for mathematical operations but a dedicated *floating*

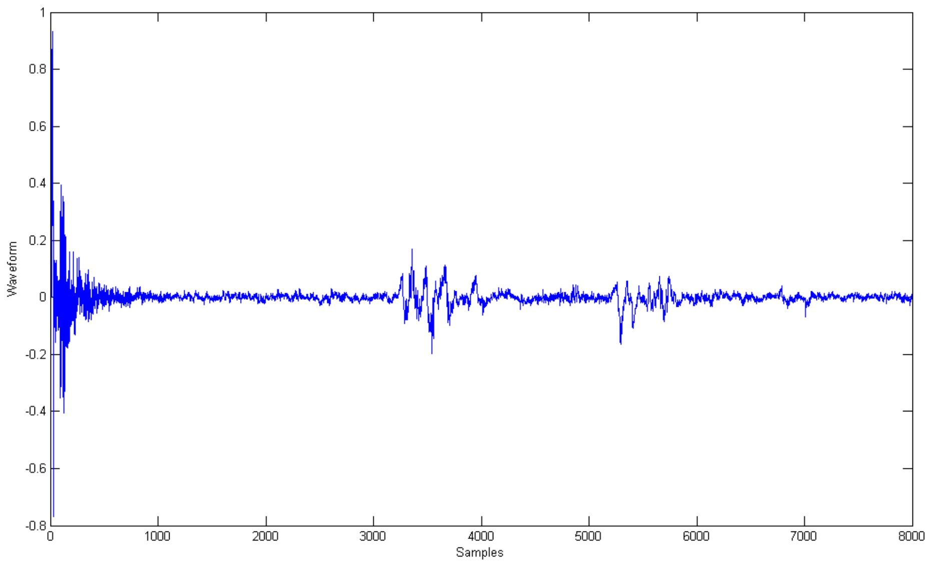


Figure 3.4: Output of the C code using *floating point*

point processor unit is needed to be efficient (and probably the hardware used to build the final product won't have it), we will implement *fixed point*. A fast explanation can be that when we are using decimal numbers in our digital system we divide our variable in two parts and consider some bits as the integer part and some other bits as the decimal part, this will limit the number of values that we can represent and we'll lose resolution in our decimals, but we'll be able to work with real numbers. To use this system we only need to create some processes to replace the standard floating-point operations In my case I have chosen to consider all the bits as decimal, this is, no integer part.

First of all, to have a more reusable code, I define a new type *QRESOLUTION*, that allows me to change the resolution that I'm working only modifying that definition.

```
QRESOLUTION AddFix(QRESOLUTION a, QRESOLUTION b) {
    long int c;
    c= (int)a+(int)b;
    if (c>=MAX_POS) c=MAX_POS;
    if (c<=MAX_NEG) c=MAX_NEG;
    return ((QRESOLUTION)c);
}
```

Starting with the addition, the first thing to consider is the possibility of having an overflow, probably working in a microprocessor we would have a flag that indicates after doing an operation if we have had overflow, but in our situation the easiest way of taking into account this problem is storing the result of the operation in a larger file, check if the result is out of our limits, and, in that case, saturate it forcing our maximum (or mini mun) value.

```

QRESOLUTION MultFix(QRESOLUTION a, QRESOLUTION b) {
    long long int c;
    c=(long long)a*(long long)b;
    c=c>>RESOLUTION;
    if (c>=MAX_POS) c=MAX_POS;
    if (c<=MAX_NEG) c=MAX_NEG;
    return (QRESOLUTION)c;
}
```

For the multiplication is almost the same, but we need to take in account one extra matter, as we are working with decimals we also need to shift our bits to the right in order to have a correct result ($3 * 5 = 15$ but $0.3 * 0.5 = 0.15$)

Division in fixed-point requires very complicated processes, and thinking that this is a system where speed is something quite important, I've decided not to use it: it is needed to normalise the LMS algorithm, but that is not a crucial specification because we can avoid an unstable system choosing lower values of μ as we don't need high adaptation speeds, and the system is faster enough with standard LMS algorithm, I've fixed μ to 0.01.

Tests

Once the filter is working I made some testing on it to know what are the limitations. To do that I've made different samples with different relation between the sound of the heart and the "infiltrated" noise, then I've applied my filter to them and show the results. In figure 3.5 we can see the effect of lowering mu, now the filter is more slower, but still only need less than 500 samples to reach an acceptable result. As the noise becomes louder than the sound of the heart, the filter starts having difficulties to clean the sample, and finally it can't distinguish the heart.

Figure 3.6 shows that in some sounds that the heart beat can't be recognised by the waveform (figure 3.5) is still there. This has been also noticed in the pool made to different people, some of them could hear the heart even if it gas indistinguishable through the waveform.

Also I've played the samples to 21 persons of different ages between 16 and 40 years (some of them studying or working related to medicine, nursery...) and asked in what example they can hear the heart obtaining the following results:

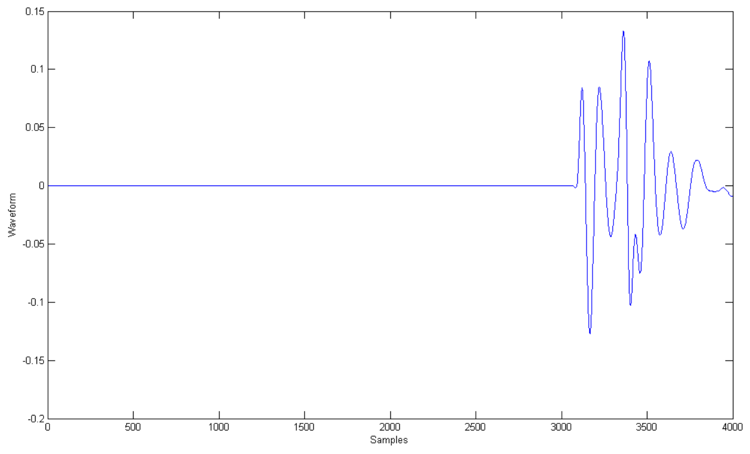
Gender	Samples in dB											
Women	-27	-27	-22	-32	-32	-27	-27	-17	-22	-27		
Men	-27	-22	-12	-22	-27	-22	-27	-22	-17	-27	-22	-17

The poll has been done in a controlled situation, using the same equipment for all the persons.

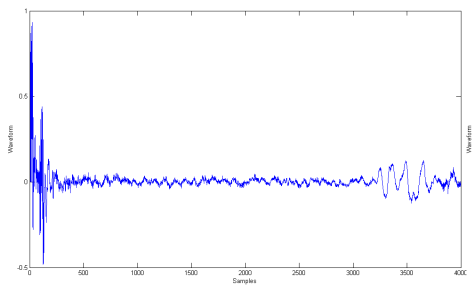
As we can see for most of the people, a difference of 22dB - 27dB is enough to listen clearly the heart, from my point of view it would difficult to have this rates of noise inside the body, so the adaptation is enough.

3.3 Hardware Implementation

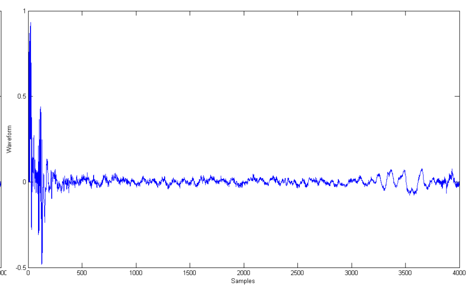
Once that the software implementation is completely functional, the next step is to point to hardware, this could be done in two ways: choose a microcontroller or DSP and modify the C code now written to be functional in this environment or



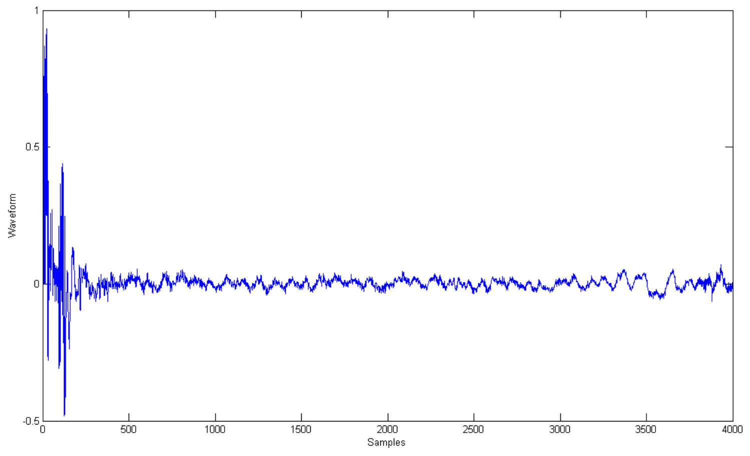
(a) Heart alone



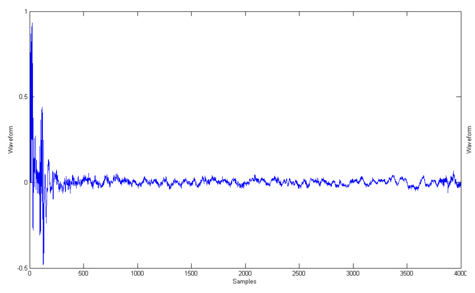
(b) -12dB



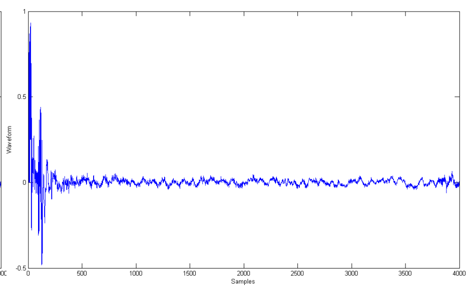
(c) -17dB



(d) -22dB

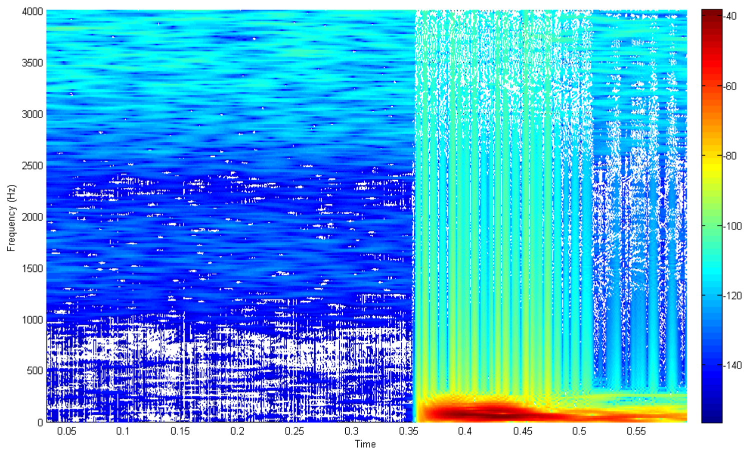


(e) -27dB

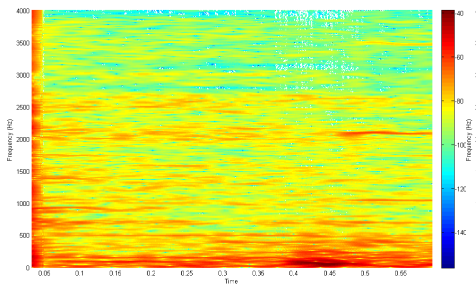


(f) -32dB

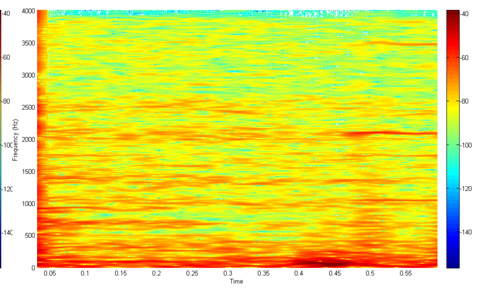
Figure 3.5: Response of the filter to different relations between the noise and the sound of the heart, more examples can be found in the CD
. Only the first 4000 samples.



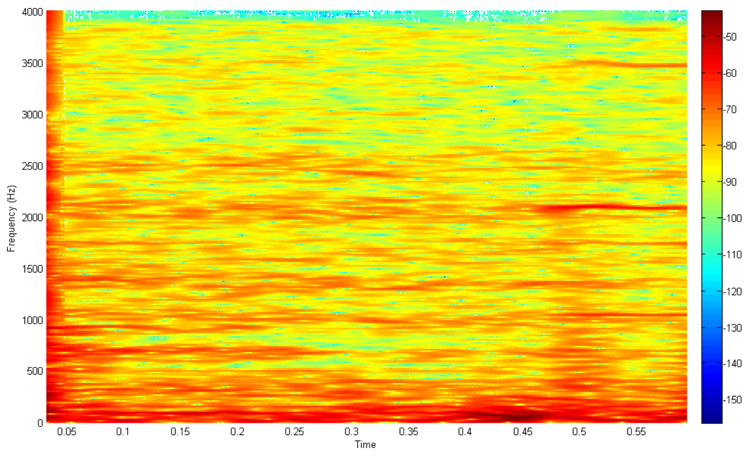
(a) Heart alone



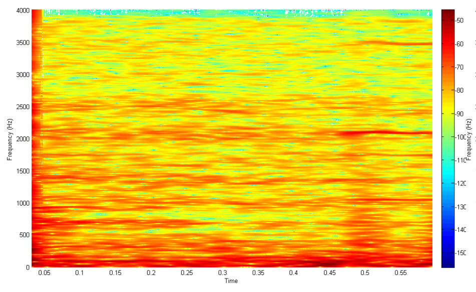
(b) -2dB



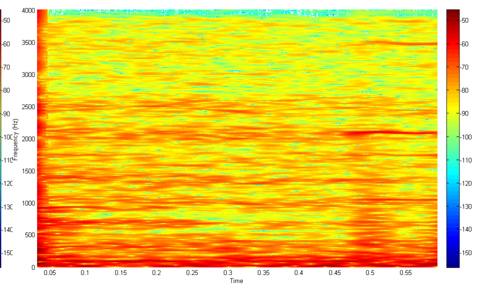
(c) -12dB



(d) -17dB



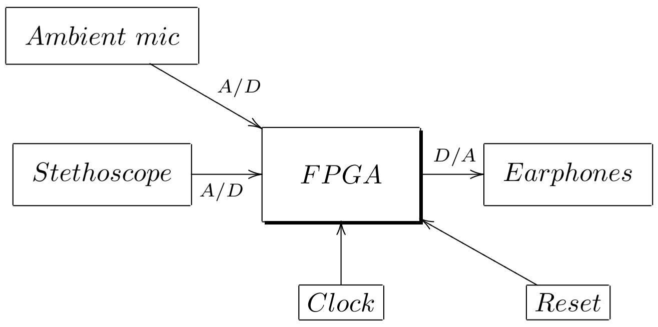
(e) -22dB



(f) -32dB

Figure 3.6: Spectrogram of the response to different relations between the noise and the sound of the heart, more examples can be found in the CD
. Only the first 5000 samples.

design a *FPGA* (Field Programmable Gate Array) that will give us more speed. As there was not enough time to find an appropriate microcontroller and deal with hardware problems and incompatibilities, the FPGA way was chosen.



This hardware needs to be ”programmed” in a completely different way, is completely focused to the electronic hardware (as the name says, is an array of gates), one of the languages to do this is *VHDL*; as this was my first time working with it (and I've never worked with digital design) and our finish goal was only to simulate the code, not build it in hardware, instead of working during all the program with *signals* and *logic operations*, I did it using the library ”IEEE.NUMERIC_STD” that allows to work numerically with *variables*.

```
entity Filter is
 Port (
  Micro : in  std_logic_vector(15 downto 0);
  Estetoscopio : in  std_logic_vector(15 downto 0);
  Salida : out  std_logic_vector (15 downto 0);
  clk: in std_logic;
  rst: in std_logic
 );
end Filter;
```

A standard VHDL code is divided in two parts, first we need to define what is called *entity*, this is where we define our system as a ”black box”, we only know what are the inputs and outputs. I this case we need two data inputs (ambient microphone and stethoscope) and one output (the earphones, only one line, in mono, there is no point about working in stereo); I've fixed the sizes to an array of 16 bits each one because in the files that I'm simulating the bits per sample are also 16.

Also is needed a clock and I've added a reset button in case that the user wants to start the filtering again.

```
architecture Behavioral of Filter is
.
.
.
end Behavioral;
```

The other part is the *architecture* that is where we write what we need our program to do.

As I've commented as a first approaching to VHDL, I'm only using variables and direct assignations in the code, so it will be very similar to the one in C.

The critical points will be, again the addition and multiplication.

```
function AddFix (b,a:sample) return sample is
variable c: signed (2*RESOLUTION-1 downto 0)
    := (others => '0');
variable d: sample;
begin
        c := to_signed(to_integer(a),2*RESOLUTION)
             +to_signed(to_integer(b),2*RESOLUTION);
        if (c>=MAX_POS) then
                d:=MAX_POS;
        elsif (c<=MAX_NEG) then
                d := MAX_NEG;
        else
           d:= to_signed(to_integer(c),RESOLUTION);
        end if;
return d;
end function AddFix;
```

First of all I've defined a new type *sample* that will allow to change the resolution without changing much code, in reality is just a *signed* with the size of the resolution.

The method used is, again do the operation, store the result in a bigger variable and then check if is it is reasonable. But as VHDL is a very *strong typed* language, I needed to do some arrangements. We need to store the result in a variable larger than the operands, to do this all the variables need to be of the same type and size, so I need to change the operands from being *signed* of one size to *signed* but of another size, VHDL doesn't allow to do this directly so, the only way is convert the *signed* to an *integer* type and then to the new *signed*.

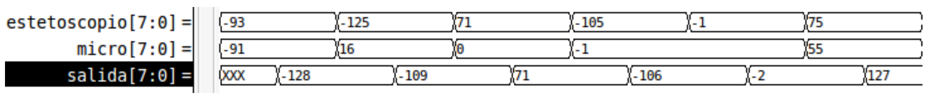


Figure 3.7: Function for addition: *estetoscopio* + *ruido* = *salida*

The only problem with this conversion is that we are limited to a maximum of 16 bit that is the maximum length of an integer.

```
function MultFix (b,a:sample) return sample is
variable c: signed (4*RESOLUTION-1 downto 0)
    := (others => '0');
variable d: sample;
begin
        c:= to_signed(to_integer(a),2*RESOLUTION)
            *to_signed(to_integer(b),2*RESOLUTION);
    c := shift_right (c,RESOLUTION);
        if (c>=MAX_POS) then
```

```
                d:=MAX_POS;
        elsif (c<=MAX_NEG) then
                d:=MAX_NEG;
        else
           d:= to_signed(to_integer(c),RESOLUTION);
        end if;
return d;
end function MultFix;
```

For the multiplication we need to do the same again, plus shift the array to have a correct result in the decimal part (in this case all the array).

Testbench

As VHDL is prepared for hardware design to run a simulation I need to prepare what a *testbench*, this is another piece of code that will write the inputs and read the outputs for my system.

```
variable linea:line;
file micFile:text open read_mode is "mic.dat";
readline(micFile,linea);
read(linea, micData);
```

To test this system a big amount of data is needed, we cannot see if it's working reading only a few values, so the easier way to handle this situation would be to read directly the wav. VHDL doesn't allow this but we can read character arrays from a text file.

I made two little applications in C (they can be found in the CD) that transform a wav file in a text file (or vice-versa). The format of the text file is one sample per line, and each sample is in binary (ones and zeros). So before running the simulation we need to transform the wav file *wav2txt.c*, then we execute the VHDL program and the output will be another text file that can be transformed to a wav file using *txt2wav.c* to check if the filter is working.

-In this case, the code is not completely functional, the output is not as expected, but due the lack of more time to work in the project I haven't be able to find where the code is failing-

Chapter 4

Conclusions

The results achieved in this project, have been quite satisfactory.

This has been my first approach to the "digital signal world" and I'm pleasantly surprised of the potential of this area of the engineering.

I think that the filter is working really well, with a very simple algorithm it achieves nice results in a short time, enough for what it was designed, I don't think that the relation between the sounds from the body and the interferences inside it will be so exaggerated as I have been using in the samples so can be used in real situations.

One error that I haven't be able to correct or even measure it is that the filter inserts a small delay when is compared to the original sample, that could be a problem some future application. Also the VHDL coding could have been much better, the one that I have written cannot be implemented in hardware, but as I said this was my first time working with that language and I don't know anything about digital designs (complex digital designs) with gates, it was no time enough to start learning from 0 all that is involved with this area. One big thing missing from this project is the actual hardware implementation (not only a simulation), but I had to spend most of the time acquiring knowledge needed to be able to start to work with the project. I haven't obtained many visible results, but definitely doing this project has been worth. Ive needed to learn from the very basis about digital signals, digital processing, filters, programming languages obviously I haven't be able to go deep in many areas due the lack of time and in others the background needed was too much for what I knew (internal operation of the LMS algorithm).

Future

Based on what has been done so far, the next step would be to finally implement the code in real hardware. We could use a microcontroller or a DSP, for that we should adapt the C code, and probably some optimisation for a particular hardware will be needed.

Other way to follow would be to implement the FPGA, this would require more work as the code need to be completely rewritten as is prepared exclusively for simulation.

A little improvements that I see necessary is to implement a volume control to amplify the sound, usually the sounds that can be taken from the body have a low intensity, and this was one of the first advantages of electronic stethoscopes before the digital era.

Further additions could be connection with a personal computer that would allow to record, transmit, manipulate... the signal.

Bibliography

[1] Wikipedia, "Stethoscope" `http://en.wikipedia.org/wiki/Stethoscope`

[2] Meyers Konversations-Lexikon (1885-90). Public Domain.

[3] Howard Hughes Medical Institute, History of Stethoscopes and Sphygmomanometers,
`http://www.hhmi.org/biointeractive/museum/exhibit98/content/b6_17info.html`

[4] Steele Supply Company, Stethoscope Anatomy,
`http://www.steeles.com/Littmann/anatomy.html`

[5] "Chris", The Auscultation Assistant,
`http://www.med.ucla.edu/wilkes/intro.html`

[6] Richard G. Lyons, Understanding Digital Signal Processing, Second Edition, Prentice Hall,2004

[7] Modified from: Wikipedia,
`http://en.wikipedia.org/wiki/File:Lms_filter.png`

[8] Wikipedia,
`http://en.wikipedia.org/wiki/File:Gradient_descent.png`

[9] The free sound project, "greyseraphim"
`http://www.freesound.org/samplesViewSingle.php?id=21409`