Design and implementation of a Beat Detector algorithm

(ETIN80 report)

Jaime Gancedo, ja1626ga-s@student.lu.se
Sakif Hossain, sa5536ho-s@student.lu.se
Wenpeng Song, we3543so-s@student.lu.se

March 11, 2018

Abstract

This document presents the theoretical study and design of a beat detection algorithm. This program aims at being able to estimate the Beats Per Minute (BPM) of any song by simply feeding it as an input to a Digital Signal Processor (DSP) and displaying the detected beat by lighting up a Light Emitting Diode (LED). The algorithm was first developed in Matlab, for testing and design purposes, and later translated to C. The translation to C was carried out to be able to program a specific DSP board (SHARC ADSP-21262) and run the algorithm in it. Final simulation results achieved with this algorithm, development setbacks, challenges in the physical implementation and conclusions are also presented.

1 Introduction

This report contains the theory, design and implementation on a DSP board of a beat detection algorithm, a project for the course "ETIN80 - Algorithms for Signal Processors". The algorithm was first studied and simulated in Matlab, checking that, given any song, it may calculate a reasonably accurate estimation of its BPM for every input sample block. Namely, the obtained beat should have a precision of at least 0.1 BPM. Once validated for this specific case, the algorithm was then translated to the C language, witih the aim to be implemented and tested in a physical DSP system.

2 Theoretical study

To achieve the described behavior, it is necessary to develop a signal processing algorithm capable of discerning the frequency at which the song's rythm happens. This is done in several steps, which are described in order in this section. It is important to note that the following operations and steps are performed every time a new block is available for the DSP to process. The size of the block, in sound samples, is configurable, and different sizes yield different accuracies for the algorithm's output, while at the same time drawbacks such as lower or higher memory consumption and computational cost.

The value we chose to achieve a compromise was $M = 280$ blocks. The following steps are repeated for a specific number of blocks, which we define as the "Window size" W. For a good performance, the chosen value for the window size is $W = 2048$. This theoretical study will be exemplified with a 5 seconds extract of Iron Maiden's "*Run to the hills*" song, which signal amplitude in time is shown in Figure 1.

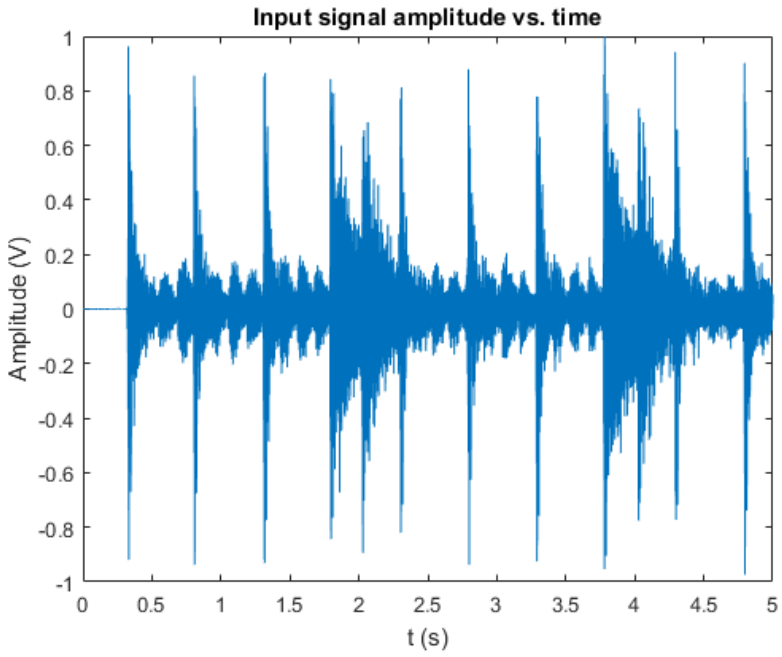

Figure 1: Amplitude in time of "*Run to the hills*"

2.1 Power recursive calculation

The first step consists on calculating the accumulated power of the signal (based off previous signal values). A shift-register of W values is used to store power results, so for every new calculation this register is shifted left, making room for the new power value. The block's average is the first calculated term in this process as:

$$x_{avg} = \sum_{n=0}^{M-1} x[n]^2 \tag{1}$$

Where M is the block size. Then, the accumulated power (represented as a recursive operation) is expressed as:

$$P[n] = \alpha \cdot P[n-1] \cdot (1-\alpha) \cdot x_{avg} \tag{2}$$

Where $P[n]$ is the current power value, $P[n-1]$ is the last calculated power value and α is a constant that can be selected for the best possible results. In our case, this

value is $\alpha = 0.8$. The result of this series of steps, once at least W blocks have been processed, is a registered history of the instantaneous power of the signal, which can then be used to perform frequency calculations. A graphical representation of this history can be observed in the following section.

2.2 Frequency representation and fine-graining

With the latest signal's power history, it is then possible to obtain the spectral power density of the input by performing the Fast Fourier Transform (FFT) to the stored power values. The spectral power density represents how the power of the signal is distributed for each frequency, and in this case it may be used to determine in which specific frequency the highest power is contained. The spectral power density, along with the power history, is presented in Figure 2.

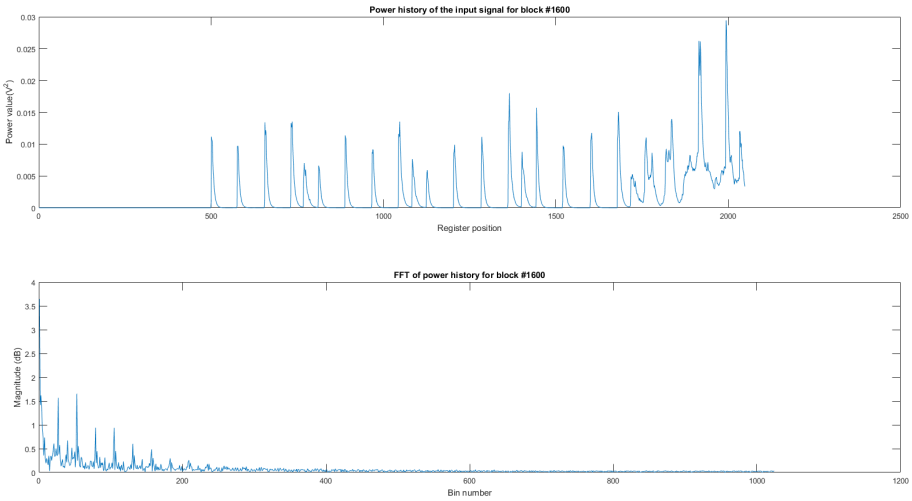

Figure 2: Power history (up) and spectral power density (down) for block #1600

For this specific application, we are interested in the frequency range corresponding to:

$$BPM \in [60, 180]$$

As this is the typical range for modern music. Taking into account that the conversion between frequency (in Hz) and BPM is defined as:

$$f = \frac{BPM}{60}$$

The frequency range that is interesting for this application is:

$$f \in [1, 3]Hz \tag{3}$$

Since the window size is $W = 2048$, the result of the FFT of the power history will also have 2048 points, or bins. Therefore, the bins of interest for this particular case will be from #11 to #36, and the maximum power value in these frequencies can be obtained to extract the song's BPM.

However, with the previous taken into account, the frequency separation between two consecutive bins will be:

$$\delta_f = \frac{f_s}{M \cdot W} = \frac{48000}{280 \cdot 2048} = 0.0837Hz$$

Where f_s is the sampling frequency in Hz. δ_f is, in fact, the frequency resolution we will obtain with this procedure. A resolution of $\delta_{BPM} = 0.1$ corresponds to a $\delta_{f_{obj}} = 0.0017Hz$, which is much finer than that our algorithm so far can provide. To overcome this inconvenience, it is possible to improve the frequency resolution in a specific range by performing the Discrete Fourier Transform (DFT) of the power history for that range. This operation is very computational costly, but in this case that should not be a problem, if designed intelligently.

The first step is deciding which frequency range will be fine-grained with the DFT. For that, some variables are known: The FFT bin in which the maximum power is (k_{max}) and the fact that, because of the way the FFT work, the real fine maximum is situated somewhere next to this maximum power index. More concretely, it will happen between the identified maximum bin and its neighbor with the highest power. Exploiting this, the DFT can then be calculated in the range:

$$DFT_{range} \in [k_{max}, k_{max} + 1]$$

or

$$DFT_{range} \in [k_{max} - 1, k_{max}]$$

depending on the current FFT values.

Once this range is known, the number of points for the DFT must be determined. In this decision, a compromise between frequency resolution and computational cost has to be made. Taking into account that de desired value for frequency resolution is $\delta_f = 0.0017Hz$, the DFT step can then be defined as:

$$\delta_k = \frac{\delta_{BPM}}{60 \cdot \delta_f} \tag{4}$$

which, in this case, yields a DFT step of:

$$\delta_k = 0.0199$$

or, equivalently, 50 DFT points.

With all the previous in mind, the DFT can then be calculated via its expression (particularized with the variables for this case):

$$X[k] = \sum_{n=0}^{M-1} P[n] \cdot e^{-\frac{j2\pi kn}{M}} \tag{5}$$

Which can then be rewritten as:

$$X[k] = e^{-\frac{j2\pi k}{M}} \cdot \sum_{n=0}^{M-1} P[n] \cdot e^{-\frac{j2\pi k(n-1)}{M}} \tag{6}$$

In this new form, an optimization can be performed to reduce then number of calculations in every DFT iteration. This is, for the first calculation in every iteration where $n = 1$, the DFT is then be evaluated as:

$$X[k]|_{n=1} = e^{-\frac{j2\pi k}{M}} \cdot P[1]$$

which allows to estimate the DFT-iteration value of the common term and re-use it for the rest of the summation iterations, saving a huge amount of precious computational time. In Figure 3 a comparison of the FFT and the DFT in the power maximum region obtained with the described algorithm is shown.

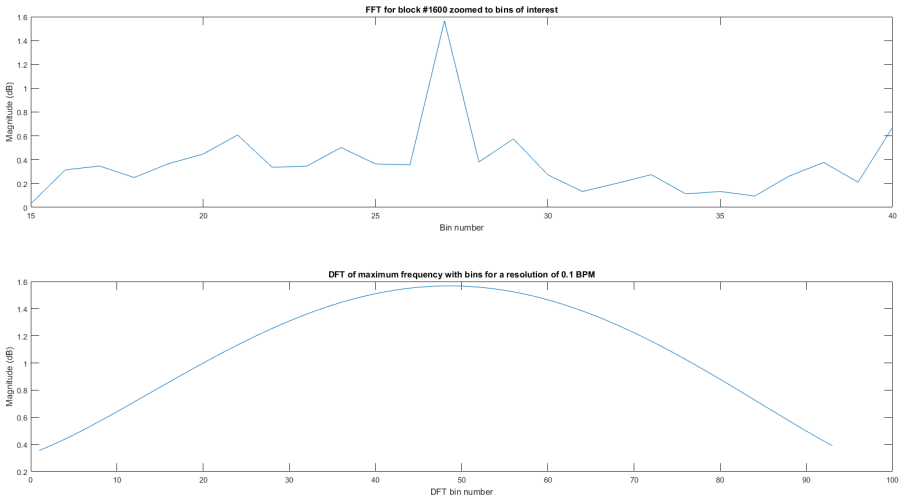

Figure 3: FFT of region of interest (up) and DFT of maximum and neigbors (down) for block #1600

At the end of every DFT iteration, the result may be evaluated as the current maximum. If this result is not the new maximum, since in this case the function is unimodal, we may assume that then no more maximums will be found and terminate early the DFT calculations saving the previous DFT maximum index as the result. Again, this can save a lot of computational time.

In the end, the fine frequency value corresponding to the obtained DFT index can be calculated as:

$$f_{beat} = \left\{ \begin{array}{ll} \frac{f_s \cdot (k_{max} + m_{max} \cdot \delta_f)}{M \cdot W}, & \text{for } DFT_{range} \in [k_{max}, k_{max} + 1] \\ \frac{f_s \cdot ((k_{max} - 1) + m_{max} \cdot \delta_f)}{M \cdot W}, & \text{for } DFT_{range} \in [k_{max} - 1, k_{max}] \end{array} \right\} \tag{7}$$

where m_{max} is the DFT index (from 0 to 49 for the example case). This value can then be converted to BPM again multiplying by 60.

A Matlab simulation of the output of this algorithm is shown in Figure 4, where it can be seen that, without the DFT optimization, the instantaneous value of the output is correct, but presents steps which are maybe too broad to obtain a correct precision (specially in the case of a real-time algorithm). In other words, this lack of frequential resolution also means that fine steps in the BPM output would be overlooked.

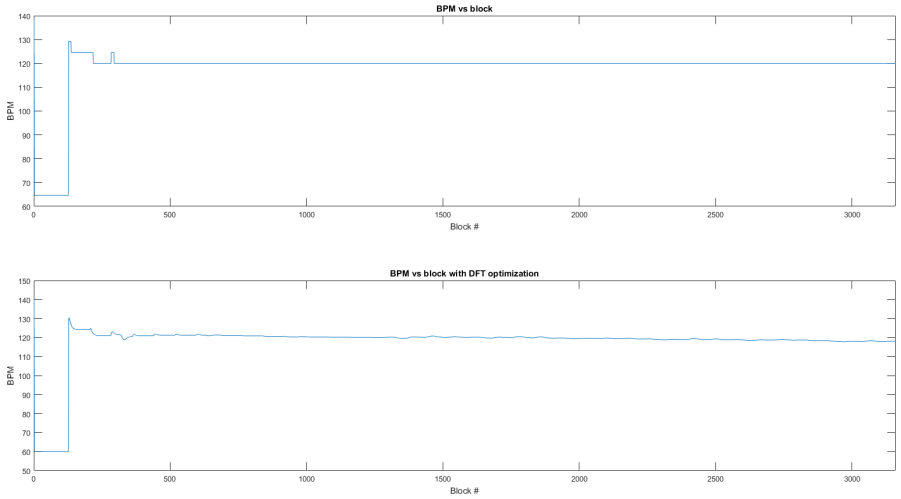

Figure 4: Algorithm's output BPM without optimization (up) and with the DFT optimization (up)

3 Implementation on the DSP

Once validated with the theorical study and the shown simulations, the beat detection algorithm had to be translated to C in order to use such translation to program the DSP board provided in the course, which core is the SHARC ADSP-21262. This phase was probably the one that required the most effort, as many problems related to the hardware constraints were faced.

One of the most important was the memory size constraints imposed by the DSP architecture. The algorithm had to be adjusted several times to try to overcome this issue. At first, the block size was reduced from the estimated 350 samples in simulation down to 280, which was the highest possible value that could still fit in the data memory. However, this block size is not enough for the proposed algorithm to function properly. After some research, the solution to "expand" memory space and be able to use a block size of 350 samples was found: the compiler directive "*pm*". This keyword is used when declaring a variable to indicate the compiler that it may be stored in the

program memory, if enough space is available. Making use of this, some variables in the C implementation could be re-allocated to the program memory in order to fit all necessary variables between the data and program memories, allowing the use of a block size of 350.

It is also interesting to explain the reasons why this algorithm presents such a high memory consumption:

- **The nature of the magnitude being measured demands the algorithm to store information for a long period of time**. The BPM information of a song, that is, the rythm or actually the "main beat" is repeated cyclically throughout the song, but only happens once every 0.33 to 1 seconds (see (3) in Section 2). Depending on the sampling rate, the minimum necessary observation time for the algorithm to start giving an output may be very high. In this case, a 16kHz sampling rate was found to work best but, for that configuration, the DSP needs around 45 seconds of initialization time.

- **The tools used in the proposed algorithm demand the use of complex numbers**. These are, mainly, the DFT calculations, apart from some auxiliary variables that are needed for the calculation of the initial FFT in every iteration. Every *complex_float* represented variable is actually two float variables contained in a struct, so this multiplies the used memory by a single float by two which, in the case of arrays, demands huge amounts of memory.

On a different note, due to the fact that the C implementation of this algorithm in the provided DSP needs a long initialization time, the time that the output needs to reflect changes from the input is also high. This means that, for instance, if the song currently being played stops for some seconds, and then a different song starts playing, the algorithm will provide incorrect values for approximately 45 seconds (in this case).

The initialization time, or the processing latency of the algorithm in seconds, can be expressed as:

$$L = \frac{M \cdot W}{f_s} \tag{8}$$

where f_s is the sampling rate. Since the latency is inversely proportional to the sampling rate, the higher the sampling rate, the lower the latency. Therefore, one possible solution for using this algorithm with a reduced latency would be to either optimize it further, or utilize a different DSP with higher performance, so that the sampling rate can be increased while still being able to process every block in time.

4 Conclusions

The different phases of this project, and the challenges that it has posed have been very educational. The theoretical study and high-level implementation that were carried out for simulation and validation purposes were extremely valuable and interesting, and

made us understood better how the algorithm actually worked, and what the effect of the configuration parameters were, such as the block size or the sampling rate.

During the implementation on the physical DSP, it was also interesting to realize the challenge in translating a Matlab script to a "real-world" C program to be run in physical hardware, with its own limitations and requirements. In the beginning, this task's difficulty was clearly underestimated.

Finally, it is worth highlighting the use of tools like the FFT or the DFT in a practical case, which is fairly enriching, as a personal opinion. Furthermore, facing the challenge of improving what one thinks is an already good program/algorithm is not an easy task, and this situation is rarely encountered in most academic courses.

In conclusion, despite the difficulties and problems throughout the course, this project has been very valuable, interesting and fun as whole.