Beat Detection

Algorithms in Signal Processors
ETIN80

by

Luis Cavo
Siyu Tan
Adam Urga

2016-03

Table of content

Introduction

This report covers the project work done for 'ETIN80 Algorithms in Signal Processors - Project Course'. The objective of the project is the implementation of a beat detection algorithm in a Digital Signal Processor (DSP) board. This board will process the audio input and will calculate the beats per minute (BPM) of the inputted music. A LED present on this board will be blink at the same rate as the beats of the music. Furthermore, one of the onboard keys is used for manually synchronizing the LED flashing frequency with the beat of the music. For example, when a beat occurs, the user presses the key immediately. The LED will reset and flash again in phase with the music.

The DSP used in this course is SHARC ADSP-21262 and the programming environment used is Visual DSP++. The DSP platform is connected to an emulator platform, the HPUSB-ICE platform. Input is provided by a 3.5mm cable connected to external source, and a headphone is used to monitor the output music.

Theory

As we will discuss later, the most important mathematical tool used in this project is the Discrete Fourier Transform (DFT). The DFT plays an important role in the analysis, design and implementation of discrete-time signal processing algorithms. The DFT is used to analyze the frequency content of a continuous-time audio signal, which has been previously sampled in the DSP processor.
The digital computation of an N-point DFT is done by means of an efficient class of algorithms called fast Fourier transform (FFT) algorithms [1]. These algorithms reduce the computational complexity of the DFT from $O(N^2)$ to $(N/2)\log_2(N)$.

Methods

This sections covers an explanation of the different algorithms used to find the BPM of music. The implementation of these algorithms on a DSP is also covered.

Algorithms

The algorithm used in this project was first developed and tested using Matlab. The algorithm can be separated into two different parts: time domain analysis and frequency domain analysis [2].

Time domain algorithm

When the audio signal is sampled by the DSP, it is stored in a buffer. This buffer has a limited size which is chosen by the designer. Every time this buffer is filled in with new data, an interrupt is raised by the system and the corresponding handling routine is executed in the DSP. Therefore, the

algorithm can only process a limited number of samples at a time. In our algorithm, we set the buffer size to 375 samples.
First, an average power calculation of the input sequence is performed. The average power of each block of 375 samples is calculated using the equation below:

$$X_{avg}(n) = \frac{1}{N}\sum_{k=0}^{N-1} x(n-k)^2 \qquad (1)$$

For each new block we calculate the new power average by using the following recursive formula:

$$P(n) = \alpha P(n-1) + (1-\alpha)X_{avg}(n)^2 \qquad (2)$$

Where P(n) and P(n-1) represent the current and previous recursive average power calculations respectively, and α is a constant to control the effectiveness of the algorithm. A greater α value will make P(n) less variant on new input average values (X_{avg}). We set α=0.85 in the current project. All the values of P(n) are stored and are used for further frequency processing. P(n) is a signal that provides information about the power of the input sequence. Therefore, by performing an FFT to this signal, we will be able to find the frequency at which the power is maximum. The size chosen for the FFT should be a power of 2, and for this reason after different simulations with different values, we chose N = 2048 as FFT size and therefore 2048 values of P(n).
Furthermore, P(n) is not re-calculated every time, but a sliding window is implemented. When the vector containing P(n) is filled in with values, the next value will be added to this same vector and the first value in the vector will be discarded. Without implementing a sliding window, the calculation of the FFT would be performed every time 2048 samples are acquired, which would approximately correspond to:

$$T_{FFT} = \frac{buffer\ size}{N \times F_s} = 16\ sec \qquad (3)$$

This would mean that we would only be able to detect variations in the BPM only every 16 seconds. By using the sliding window approach, the BPM calculated is continuously following the BPM of the music.

Frequency domain algorithm

The P(n) data stored in the FIFO is used to perform an FFT calculation. At the beginning, the FIFO is empty. After more data is stored, FFT spectrum increases in power. However, even if there are zeros in the FIFO, the FFT can still perform the correct result.

Based on the theory, we know that the frequency which represents the beat, has the highest power. Therefore, by analyzing the power spectrum of the P(n) sequence and finding the bin that corresponds to the highest power, we are able to find the BPM of the sound which should be proportional to the highest bin. However, we have limited the searching algorithm to a certain bin range, which is in between of the lowest BPM (60) and the highest BPM (200). It is from bin #16 to bin #52. The correspondence between the bin number and frequency of the BPM is given by the following formula:

$$\text{BPM} = N_{bin} \times \frac{F_s}{buff\_size} \Big/ N_{samples} \times 60 \qquad (4)$$

Implementation

Once we have a working algorithm in Matlab, an implementation on a DSP is straightforward. Every time the input buffer is full, meaning 375 samples are available, an interrupt is triggered and the program execution jumps to the interrupt handler. The recursive averaging FIFO shifts left one position, and the last position stores the previous recursive value plus the instantaneous power, with coefficient α. Then, a 2048 points FFT is performed by calling the built-in function rfft2048(). Finally, the bin with the highest power represents the BPM value, which is then calculated using (4). Only the bins in the range between 60 BPM and 200 BPM are checked as we mentioned before. The following code extract performs the aforementioned task:

```
// Copy audio from left channel to input buffer. Audio samples are 32 bit
// fixed-point values in the range [-1, 1] so no additional scaling is required.
for(n=0; n<DSP_BLOCK_SIZE; ++n) {
    X[n] = audioin[n].left;
    X_avg = (X[n]*X[n]+X_avg);
}

    // Instantanious average per block:
    X_inst = (1/block_size)*X_avg;

    // Sliding window average power:
    for (n=0; n<(REC_BLOCK_SIZE-1); ++n) {
        X_Rec[n] = X_Rec[n+1];
    }

    X_Rec[REC_BLOCK_SIZE-1]=(X_Rec[REC_BLOCK_SIZE-2])*alpha+(1-alpha)*X_inst;

    // Calculate the FFT of the power signal
    rfft2048(X_Rec, FFT_Real, FFT_Imag);
    for (n=0;n<(REC_BLOCK_SIZE/2)-1;n++){
        Y[n] = sqrt(pow(FFT_Real[n],2)+pow(FFT_Imag[n],2));
        if ((Y[n]>Y_max)&&(n<(BPM_MAX+1))&&(n>(BPM_MIN-1))) {
            Y_max=Y[n];
            n_max=n;
        }
    }
```

The limitation of the algorithm, however, is the BPM resolution. Neighboring bins have a BPM difference of ±4 BPM, and thus, if we want to have an increased resolution, some optimizations need to be done. When a bin n_max is found, the power of neighbor two bins n_max-1 and n_max+1 is checked, and compared between each other to find which one is larger. In the range that goes from n_max to n_max±1, we directly perform the DFT of the P(n) signal, as the equation shows below.

$$Xk = \sum_{n=0}^{N-1} P(n) \cdot e^{-2\pi ikn/N} = \sum_{n=0}^{N-1} P(n) \cdot e^{-2\pi ik(n-1)/N} \cdot e^{-2\pi ik \cdot 1/N} \quad (5)$$

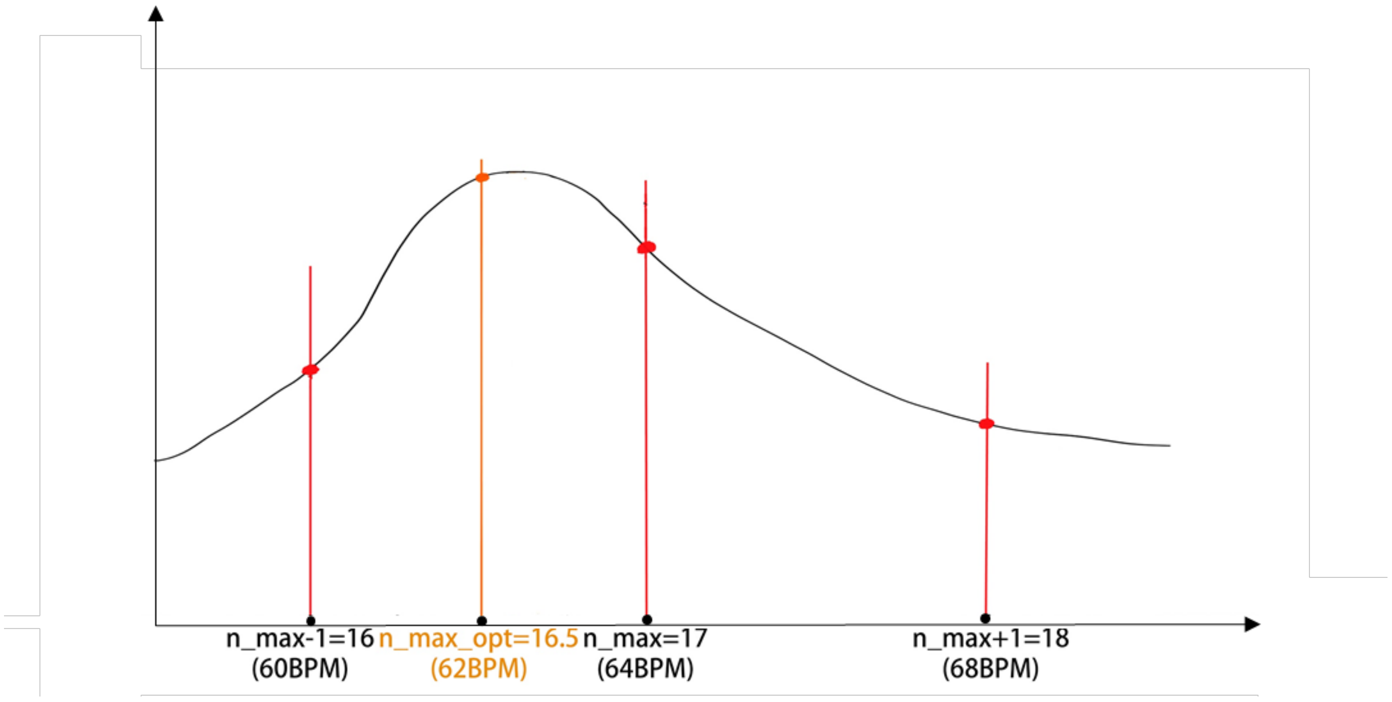

Figure 1 BPM resolution improvement algorithm

The first value, when n=1, leads to a result $Xk|_{n=1} = P(1) \cdot 1 \cdot e^{-2\pi ik\cdot\frac{1}{N}}$. Unlike the normal DFT operation, the complex exponential $e^{-2\pi ik\cdot 1/N}$ is only calculated once and acts as a constant in the iteration, and $e^{-2\pi ik(n-1)/N} \cdot e^{-2\pi ik\cdot 1/N} = e^{-2\pi ik(n-2)/N} \cdot e^{-2\pi ik\cdot 1/N} \cdot e^{-2\pi ik\cdot 1/N}$. This saves precious amount of CPU time when calculating DFT result. Fig. 1 illustrates the concept of such algorithm. The code extract is shown below.

```
for (j;j<1 && j>-1;){
            E_0.re = 1;
            E_0.im = 0;
            E_1 = E_0;
            E.re = X_Rec[0];
            E.im = 0;
            z.re = 0;
            z.im = n_max+j;
            y    = cexpf(z);
            for (n=1;n<REC_BLOCK_SIZE-1;n=n+4){
                E_1.re = y.re * E_0.re - y.im * E_0.im;
                E_1.im = y.re * E_0.im + y.im * E_0.re;
                E.re = E.re + X_Rec[n] * E_1.re;
                E.im = E.im + X_Rec[n] * E_1.im;
                E_0 = E_1;
            }
            E_abs = pow(E.re,2)+pow(E.im,2);

            if (E_abs >= E_max) {
                E_max = E_abs;
                n_max_opt = n_max+j;    // Refresh the bin
            }
            if(Y[n_max+1] > Y[n_max-1]){
                    j = j+0.25;}
                else{
                    j = j-0.25;}
        }
```

Every time an accurate BPM value is found, this value will be used to blink a LED according to the BPM value found. The DSP has a frequency of 98304000 Hz, and by knowing this value a timer can be set to blink the LED. The timer is loaded with a new value, and the timer controls the flashing period of a LED. This value is calculated as follows:

$$T = \frac{F_{DSP} \times Block_{size} \times N}{2 \times Nbin \times Fs} \qquad (6)$$

Where Block size = 375, N = 2048 and Fs = 48000 samples/second.

In addition, the LED can then be synchronized with music manually. This will be done by the user by pressing a button on the DSP board. The button press will trigger an interrupt, and the keyboard interrupt handler reset the timer by loading the cycle value, by using the function: timer_set(cycles, cycles).

Finally, when there is no audio input, the recursive averaging value will gradually decrease to zero. We keep track of the 20th value in the FIFO, if it is smaller than a threshold, then LED is turned off.

Results

Time domain power averaging

In Matlab, a time domain averaging is performed and the result is illustrated in Fig. 2. The orange plot is the input signal in time domain, and the purple one is the recursive averaged power of this signal, corresponding to the P(n) sequence. The power follows the peak of the input signal quite well.

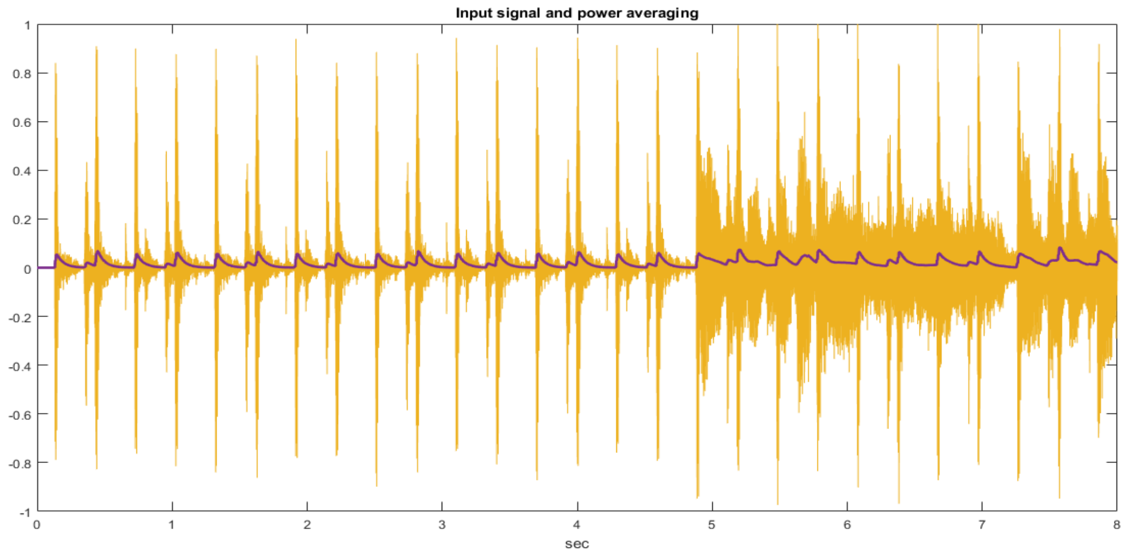

Figure 2 Time domain average power calculation

Frequency domain FFT and bpm searching

The power samples are further processed using a 2048-point FFT, and the result is shown in Fig. 3. As shown in the figure, bin number 25 has the highest peak, which represents a bpm of 93. Finally, in Matlab, we plotted the calculated bpm during every FFT calculation. It shows a flat result without large fluctuation.

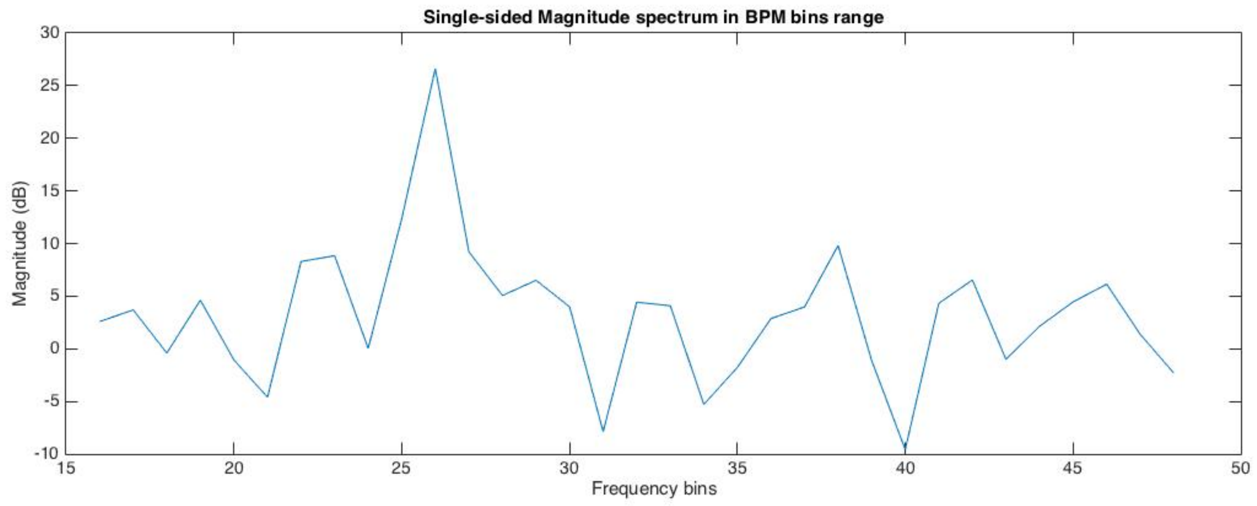

Figure 3 FFT spectrum, with a peaking indicates the bpm

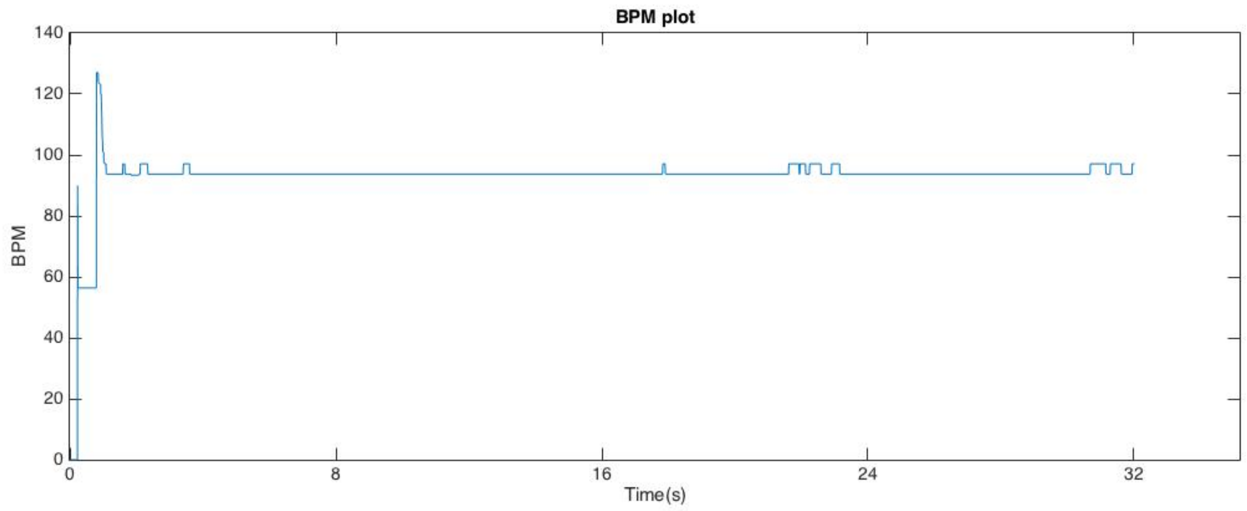

Figure 4 BPM plot in Matlab

Discussion

Performance

Unsurprisingly, the performance was highly dependent on the various parameters and even minor changes could change the result. The resolution could change depending on various factors, but what set the base limitations was the amount of stored sampled data. The more data that was stored and used gave a higher resolution while costing memory. By configuring what type of data was stored (block size, recursive average, etc.) it affected in various ways the different performance related results such as accuracy and precision.

Issues and solutions

We had some issues with the limitations on both the computational power as well as the limited amount of memory on the DSP. While neither of them caused us any large amount of grief, they could be a bit tricky to work around. To avoid the problem with limited amounts of memory it was necessary to keep a close eye on the balance between the amount of stored values for block sizes, the recursive averages, FFT and DFT. The DFT was heavy on the computational power, but by careful adjustments to the various parameters used by it and the other functions we were able to get around the problem.

We had some minor problems with deactivating the LED quickly when the music was stopped. This was solved by monitoring the recursive average power and selecting a threshold. This resulted in a very minor delay before deactivating the LED.

We had several issues before we were able to accurately detect the beats. While there is still an error, it's is now minor. We managed to minimize the delay as well. Much of this was thanks to careful tuning of parameters and the optimization by DFT.

We had some problems with irregular beats showing up on the LED rhythm due to the code we were using, but by using optimization and configuring the recursive averaging and the FFT that problem was solved.

A problem our implementation was unable to solve was songs with beats that were represented with various level of power, for example heavy techno-based music or synthesizer music. To solve something of this magnitude we concluded that a more advanced algorithm would be needed with higher accuracy and other ways to discern patterns to be able to find the beat rate.

Conclusions

We have seen from our code that the parameters are very important. Even minor changes can make or break the program. As mentioned in the issues and solution section certain types of music would require a more advanced implementation than what is covered in this course. As far as we was able to discern you would need a combination of a more advanced base algorithm as well as several more secondary methods to optimize the accuracy. This would almost certainly require more computational power and memory.

References

[1] Weisstein, Eric W, "Fast Fourier Transform." From MathWorld–AWolfram Web Resource, 2014. Available from: http://mathworld.wolfram.com/FastFourierTransform.html
[2] S. Hillbom, R. Lindberg, E. Jing, Realtime BPM and Beat Detection using ADSP-21262 SHARC DSP.
[3] VisualDSP++ 5.0 Run-Time Library Manual for SHARC® Processors:
http://www.analog.com/media/en/dsp-documentation/software-manuals/50_21k_rtl_mn_rev_1.5.pdf