MoovMe: A Motivational Exercise App

Piotr Holc
Imperial College London

June 19, 2012

Abstract

In this day and age, nearly everyone wants to be fit and attempting to achieve this, they exercise. Currently exercising follows a primordial scheme of working out to randomly chosen music attempting to serve as a motivational stimulus. This project attempts to create a solution that delivers a motivational impetus higher than that of playing arbitrary music. We have created an iPhone application that utilises a bespoke implementation of a tempo analysing algorithm using digital signal processing algorithms. This implementation works on both accelerometer data (to identify the user's pace) and on music (to identify song tempo). It works in real-time and uses the users music collection; not imposing any restrictions on what music the user can play. Finally we have compared the performance and accuracy of our algorithm with known solutions and achieved much better results with a higher degree of flexibility.

Acknowledgements

I would like to thank Prof. Krysia Broda and Dr. John Charnley for their direction, assistance, and guidance. I also would like to thank Graham Deane for the help on the intricacies of digital signal processing, Peter Mason from Iconic Media for the business and marketing insights, and Bartek Podkowa for help in debugging.

Contents

1 Introduction

The problem tackled by the project stems from the fact that simply playing music whilst exercising doesnt offer the person anything more than the bare minimum. Due to the proliferation of smartphones, especially the Apple devices (iPhone, iPod, and iPad), more people are turning away from dedicated MP3 players and in turn using their phones to play music. This opens up an array of possibilities in the realm of personal sport. Our solution is an iPhone application (colloquially called an app). It takes the existing musical layer (access to the user's music library) and attempts to deliver it more relevantly (by playing music matching the tempo of the user, rather than simply playing random songs). Harnessing hardware available on the smart phone; the accelerometer, multi-core CPU, hardware accelerated routines, and sample level access to the user's music library, the app is capable of very detailed analysis and thus majestic results. It analyses both the Beats Per Minute (BPM) of music and the user pace (also BPM), and plays music at the desired tempo. Such a desired tempo may be based upon the user wanting to run at a steady pace, run a specified distance (and thus requiring an implied speed), or a musical hills workout. Because almost every aspect is custom and proprietary code, the flexibility and thus application are infinite.

The basis of the project is Digital Signal Processing (DSP) which heavily relies upon the Discrete Fourier Transform (DFT). The DFT will be examined in detail in section 3.2. On top of DSP lies BPM analysis to be able to determine the tempo of the music and the pace of the person running. Furthermore, activity recognition using the accelerometer is employed to be able to determine if the user has stopped, continued to workout, slowed down, or sped up.

The report that follows begins with a technical analysis. The core driver of the algorithms is the Fourier transform. Section 3.2 presents the theory behind the discrete Fourier transform, and some of it's properties. We then take a look at two different types of BPM analysis algorithms. We examine beat matching, and why we do not have to implement it and activity recognition, to enforce user pace detection. Finally we recognise related work and show how no one has yet conquered musical motivation.

Following the technical analysis there are several hypotheses which need to be assessed and evaluated beforehand. This is what section 4 is all about. The app needs to serve motivational stimuli, thus various ways of motivating someone with music will be investigated. Furthermore, because the app will

run on an iPhone, there are several physical restrictions such as music library access and hardware accelerated routines. The intricacies and limitations of these are also examined in detail. Moreover, movement awareness research is explored, as being able to truly discern the users activity context is crucial to serving right music. Finally implementations of BPM analysis algorithms are looked upon, resulting in the decision to create a custom algorithm.

After both technical analysis and testing of hypotheses, implementation takes over. The design patters and choices are examined, as well as specific implementations of various hypothesis. The essence of this section is a detailed explanation of the bespoke BPM analysis algorithm, presented both textually and pictorially.

The BPM analysis algorithm is now accurate, bespoke, and very flexible. The only property it lacks is speed. Section 6 shows the effects of optimisation; making the algorithm a staggering 49% faster!

2 Accomplishment

Figure 1 presents the basic UI of the app. It is simple and elegant. The user can start running, and the app will do the hard work. It's simplicity, however, is a design choice, not a shortcoming. The heavy lifting occurs behind the scenes: in a highly multi-threaded manner the app performs: accelerometer data gathering, tempo detection using a proprietary algorithm (on both accelerometer and music data), and activity recognition (to know when the person is running or not).

Figure 1: The user interface of the application. The left screen gives three options: free run, constant, and hills. These are three different modes of exercise. Free run matches the user's tempo, constant plays a predefined tempo, and hills is the mix of the two, explained in the future work section. The right image presents the free run view: The current song being played, as well as basic controls. The heavy lifting of the app is invisible – allowing the user to focus on what is important to them – the run.

The final application has two modes: Free run and Constant. Free run allows the person to start running, and the app automatically chooses the tempo to match the user's pace. The constant run mode allows the person to set a tempo with which they want to run. The app will then maintain this tempo by time-scaling songs to match.

Figure 2 is a use case of the app. It shows what the user can do and how these actions use the technical components of the system. It is a high-level image; the components of which will be explained in detail in the forthcoming sections. The user has a few options available to him such as two different run modes and basic music controls. Running includes BPM detection on either a song signal or accelerometer signal. Finally this detection relies heavily on the DFT transform. The last part of the system is the SQLite data store which maintains persistent information on analysed music.

The application utilises several core tools: BPM detectors, Fourier transformers, and activity recognisers. These are all bespoke implementations that offer a very high degree of flexibility. The biggest example is using BPM detection on both accelerometer and music signals. This makes sense cine they are both signals and their only difference is the sampling frequency (how many samples per second). Moreover, any error in the BPM detection will be expressed in both accelerometer and music readings. External libraries, however, did not offer the capability of being able to use any type of signal (as explained in section 4.5); hence the custom implementation. These tools enable pushing the limit of what is possible on a smartphone, as explained in the future work section.

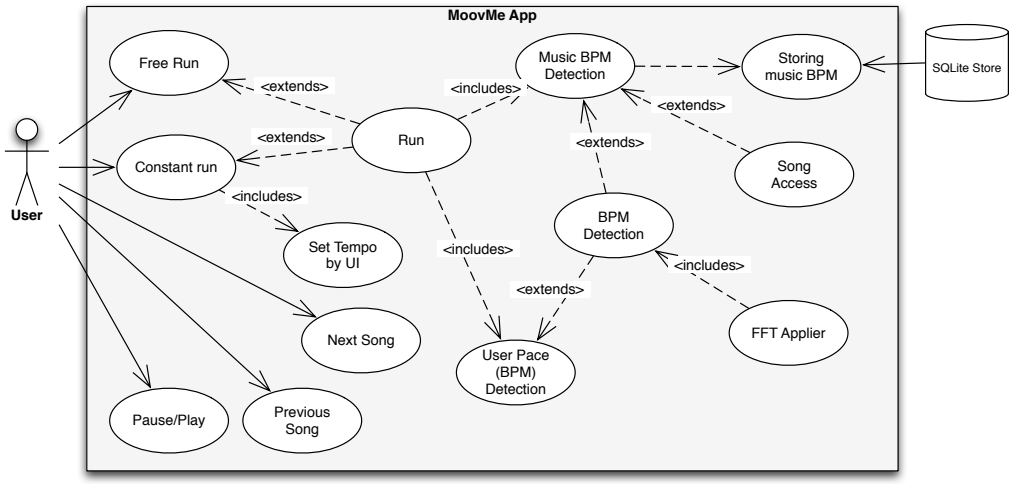


Figure 2: The use case diagram shows that the user can perform Free run or constant run. He can also get the next and previous song, and pause or play the song. The different use cases require and extend other processes, such as BPM detection for both sound and movement, storing BPM values in the datastore and setting tempo via the UI (for the constant run mode).

3 Background

3.1 Overview of Smartphone Development

This project is an investigation into assembling together a solution, using various aspects of computer science, to create a fitness tool. The device to run the app has been chosen as the iPhone because of: portability, popularity, familiarity, and power. Portability comes from the form-factor of the smartphone, which offers the best performance to size ratio. Popularity originates from the fact that as of Q1 2012 the iOS platform has 22.90% [1] market share with 3 types of devices (iPhone, iPad, iPod touch) consisting of 6 different devices (3 iPhones, 2 iPads, 1 iPod Touch)[1]. This equals approximately 3.82% market share per device. This is in contrast to the Android operating system, which as of Q1 2012 had 56.10% market share with 162 phones (183 devices), giving it a 0.35% market share per phone [2]. This fragmentation of the android devices market poses a great problem for app development, since testing must be done on as many phones as possible (since each come with different specifications, screen sizes, and hardware). This was the reason for choosing the iPhone where the real testing involved only 3 devices. The familiarity reason for choosing the iPhone was that I had previous experience programming the iPhone and was more familiar with its Application Programming Interface (API) than of any other mobile platform. Finally the power came from the fact that the 3 different iPhones offer great performance. The iPhone 4S, the newest iOS device, released October 7, 2011, offers a very fast 800 MHz dual-core ARM Cortex-A9 CPU with 512 MB of RAM. This hardware can be used by an API for hardware accelerated calculations (such as the DFT).

3.2 Technical Research

The following section will present the theoretical background information, and the solutions tailored to the iOS platform and device capabilities.

Discrete Fourier Transform

The BPM analysis implementation relies heavily on the forward and inverse Fourier transforms to perform a filter-banking of the signal and fast convolu-

[1] The 6 devices currently on sale (iPhones: 3GS, 4, 4S, iPads: 2, 3rd Generation, and 4th generation iPod Touch) without the discontinued units (first generation iPod, iPhone 3G, and older iPod touches).

tions. This section describes the Fourier transform, as it pertains to digital signal processing, rather than the application at hand.

The Fourier transform is a mathematical operation that enables expressing a function of time in its frequency spectrum. The notion of time-domain and frequency-domain is crucial to the Fourier transform. The time domain shows how a signal or function changes over time. The frequency domain shows "how much of the signal lies within each given frequency band over a range of frequencies [3]".

A forward Fourier transform expresses a time domain function in the frequency domain, whereas the inverse Fourier transform expresses the frequency domain in the time domain. Fourier analysis is currently widely used in physics, partial differential equations, number theory, combinatorics, signal processing, imaging, probability theory, statistics, option pricing, cryptography, numerical analysis, acoustics, oceanography, geometry, and even protein structure analysis [3].

Signal processing on sound involves discrete data, we thus focus on the discrete Fourier transform. It changes an input signal of length N into two point output signals (complex numbers). The input signal (in time domain) is decomposed into the two signals (frequency domain), one containing cosine amplitudes of the component, and one the sine wave amplitudes [4].

The forward Fourier transform equation is shown in equation 1. The transform takes as input a sequence of numbers $x_0, ..., x_{N-1}$ (in the time domain) and outputs a sequence of $N/2+1$ complex numbers (in frequency domain).

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi \frac{k}{N} n}, \quad 0 \leq k \leq \frac{N}{2} \tag{1}$$

The inverse forward transform is described by equation 2. Similarly this transform takes $N/2+1$ complex numbers (in frequency domain) and output N complex numbers (in time domain).

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{i2\pi \frac{k}{N} n}, \quad 0 \leq n \leq N-1 \tag{2}$$

Figure 3 displays an exemplary forward Fourier transform. The x-axis of the frequency domain poses various issues, but is crucial to understand for manipulation of frequency data. The confusion comes from the fact that there are 4 ways to refer to the frequency domain in DSP.

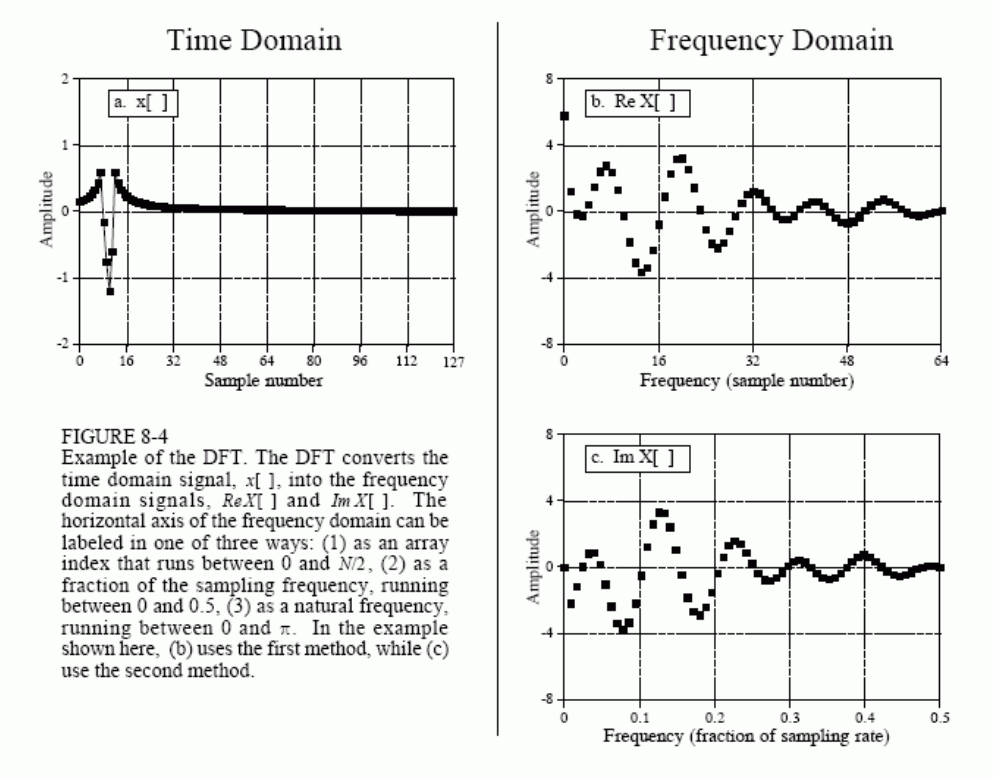


Figure 3: A forward Fast Fourier Transform (FFT) calculation [4]

1. The axis can be labelled with the index of the sample. In our example the time domain signal has N=128 points, thus the two output signals will have N/2 + 1 (65) points. This labelling is synonymous to the way the samples are stored in various data types (by an index).

2. Another way to label the axis is using the fraction of the sampling rate. The values are thus from 0 to 0.5 (since the transform outputs half the length, equaling half the sampling rate).

3. The third way multiplies the second option by 2π. This causes the values to be equally spaced between 0 and π. This allows expressing the value easier (since it is a sine or cosine).

4. The fourth method is application specific. If the sampling frequency has a sampling rate of f_s kHz, then the frequency domain can be

labelled from 0 to $f_s/2$ kHz. Thus if the system is 44.1 kHz (a common sampling frequency for CD's) the frequency domain will run from 0 to 22.05 kHz (22050 samples per second).

When the signal can be quickly turned into the frequency domain, more elaborate analysis can be run on the signal. Furthermore, simply removing particular bands (for instance clearing values between 0 and 32, in our previous example), would remove those frequencies from the signal. Thus low or high passing the signal. Also analysis can be performed on particular portions of the signal. In music, usually the lower frequencies (bass kicks) are periodic and thus dictate the tempo of the song. Performing BPM analysis only on that portion of the signal (and not on the higher frequency portion of the signal) could give much clearer information about the tempo. Because of this, the time-comb BPM detection algorithm relies heavily on performing forward and inverse transforms. The detailed application, therefore, will be explained in the next section.

3.3 Required Processes and Components

The solution has various components and processes required to create a working system. They are:

- BPM Analysis
- Beat Matching
- Activity Recognition

Each of the different components are presented in a different section since each has its own research, problems, and solutions.

BPM Analysis

BPM analysis comes in two parts: music and movement BPM analysis. Both parts utilise the same algorithm, so that any margin of error is expressed in both outcomes. Analysing the BPM of the users music library is a background process that takes each song in the library, runs the BPM analysis algorithms on it and stores the songs BPM in an SQLite database. This process is the most computationally intensive, as well as the most important. An incorrect BPM value causes the song's tempo to be unequal to the pre-defined tempo. This will cause the user to hear the music out of tempo to his pace of exercising, and, being the worst-case scenario, will be such as

listening to random music.
Frederick Patin [5] identified two flavours of beat detection:

- Statistical streaming beat detection
- Filtering rhythm detection

Statistical Streaming beat detection This class of algorithms rely on the notion of sound energy. Humans infer beats as pseudo-periodical successions of sound [5]. The beat is a sound whose energy is higher than its close neighbours, with large changes in energy, expressing periodicity. Statistical streaming beat detection identifies these energy peaks, and calculates their periodicity, thus achieve a BPM value. The algorithm compares the average sound energy with the instant sound energy, and upon finding a local maximum denotes it as a beat.

Filtering rhythm detection The problem with statistical streaming beat detection algorithms is that a sound may be full of noise, and such noise needs to be removed. Filtering rhythm detection algorithms split the signal into frequency sub bands, or filter bank the signal, and perform analysis on each band. The reason for doing this is that different instruments are expressed in different frequencies. For example, a kick drum would be in the 44 Hz - 200 Hz range, whilst a high hat would be in the 8 kHz - 12 kHz range. By separating the signal into sub bands we can perform analysis on bands which are very close to particular instruments. Well chosen sub-bands could almost separate particular instruments[2].

To filter bank the signal, the forward FFT is taken, which transforms the signal from the time domain to the frequency domain. The signal is then separated into frequency bands. The filtering rhythm detection algorithm attempts to find the BPM by time-combing the signal. Such a time comb is shown in figure 4. For each BPM value from the chosen range (for instance, 100-200 BPM), a time comb signal is created. This time-comb signal has the highest energy at the time when the beat is to occur, and 0 everywhere else [6].

The DFT of the original signal is then multiplied with the DFT of the time comb (convolution). When the time comb corresponding to the BPM

[2]Filter banking does not guarantee that each particular band will contain the signal of a single instrument since single instruments may contain noise, i.e. a kick drum occurring in the 44-200 Hz range may contain parts in the 2 kHz frequency. However filter banking will achieve the result of muffing other instruments, and this is enough to achieve good results.

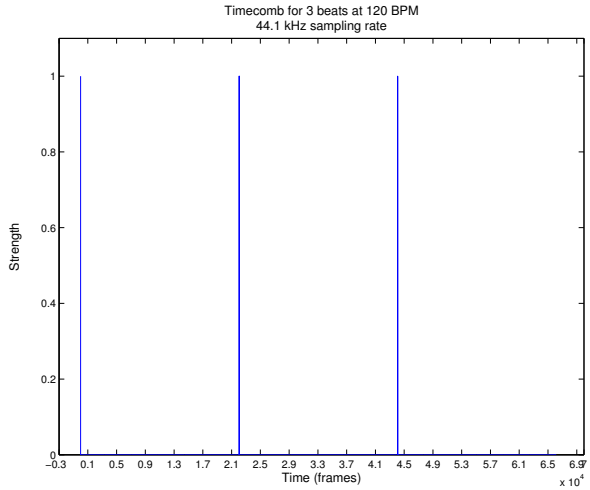


Figure 4: A time comb representing 3 beats at 120 BPM using a sampling rate of 44.1 kHz. The whole signal is zero apart from where the beat occurs (0, 22050, and 44100). A time comb like this one is created for every BPM in the range tested. Each comb is then convolved (multiplied in the frequency domain) with the signal. The time comb which gives the highest sum of the product, corresponds to the signals underlying tempo.

of the signal is multiplied with this signal, the sum of the energies is greater than for any other time comb. Thus the BPM underlying the sound sample is found on the time-comb where the sum of the convolution is greatest.

Beat Matching

Beat Matching is the synchronisation of one signal to another, such that their tempos are the same, and their beats occur at the same time. Simply matching tempo is not enough to ensure correct beat matching, as the beats and bars begin and end at different moments in time. For the purpose of the project, and with insights from research into neuroscience [7] it can be concluded that "one universal of human music perception is the tendency to move in synchrony with a periodic beat". Such a nature in humans to adjust themselves to a beat, such as the music the app plays, makes it very difficult to sync our music with the users movement. Figure 5 presents a situation where two signals, both beat-matched to the same tempo, are offset by alpha (α) , which equals a tenth of a second: 4410 samples (44.1 kHz

sampling rate). The human would automatically attempt to shift their pace to equal that of the music, and the application would attempt to shift itself to equal that of the human, leading to undesired effects. The application would constantly miss the perfect tempo moment with the human. The proper solution is to exploit the natural human tendency, and not perform any syncing. The user, without even noticing, will automatically shift his tempo to match the apps.

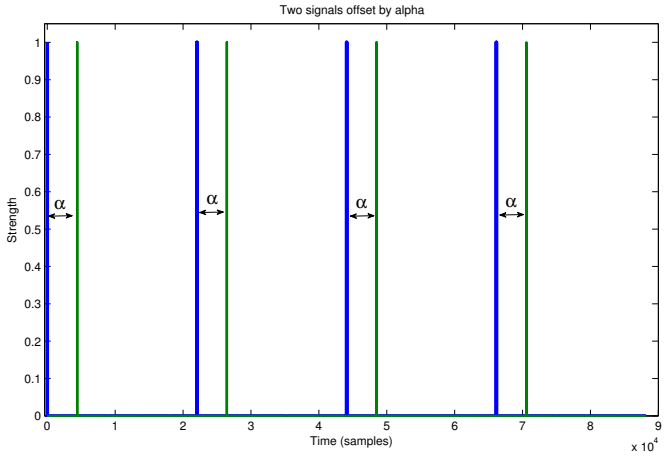


Figure 5: The two signals presented here (one blue, the other green) are in the same tempo, but offset by alpha (α). Beat matching would need to be done to shift the sound sample by alpha, such that the signals are overlaid. However, because of the human condition, users will automatically attempt to synchronise with the external tempo stimuli. This would cause a problem as the phone would try to calculate alpha and then scale to this alpha, the human, however, would perform the same thing. This would require synchronisation of a constantly moving and unpredictable alpha. It is much easier to simply not scale, and leave the human to perform this computation by himself.

Activity Recognition

The application needs to distinguish between the different actions of the user. For instance, the app needs to be notified when the user stops running, start running, slows down, or speeds up. A lot of research has been

done in the past on activity recognition using accelerometer data. Niskham Ravi, et al [8] investigated using machine learning on accelerometer data to discern between standing, walking, running, climbing up stairs, climbing down stairs, performing sit-ups, vacuuming, and brushing teeth. The experiment was run using various classifiers, with the Plurality Voting classifier offering the best results. 4 different settings were used for the experiment. Setting 1 was "data collected for a single subject over different days, mixed together and cross-validated". Setting 2 was "data collected for multiple subjects over different days, mixed together and cross-validated". Setting 3 was "data collected for a single subject on one day used as training data, and data collected for the same subject on another day used as testing data". Finally setting 4 was "data collected for a subject for one day used as training data, and data collected on another subject on another day used as testing data". The outcome of Plurality Voting was thus 99.57% for setting 1, 99.82% for setting 2, 90.61% for setting 3, and 65.33% for setting 4. Based on these results the implementation of activity recognition is based in majority on the research by Niskham Ravi, et al.

3.4 Related Work

The proposed solution is a combination of existing technology, in a unique way. Related work, therefore, span the whole technological landscape.
Myung-kyung Suh [9], et al, of the University of California, Los Angeles, created a smartphone application which proposed music for exercise. Their solution streamed music from an online database, matching the users training plan with the music played. Their results show that "compared with the uncontrolled condition, the experiment shows that exercise commands generally help users to exercise more accurately. [...] the accuracy of the exercise is improved from 53.97% to 88.71%" Accuracy in this sense is the ability to maintain a steady pace. This solution, however, only played music that the user preferred, learning what songs which gender, age, and residential area prefer. Moreover, the app chose music server-side. It did not allow the person to choose their own music, something our solution does. This is poor user experience as it creates a restriction on what the user can listen to. There is no clear correlation between the gender, age, and residential area and the type of music listened to. Perhaps if the app analysed what the user had on their phone and matched that, it would give people more freedom.

Nike, the major sportswear and equipment supplier, released successful apps such as Nike+ Global Positioning System (GPS), Nike BOOM, and

Nike Training Club. All of these apps attempt to seize the opportunities of offering more to the amateur athlete. The currently existing apps, however, offer additional layers of information, without truly harnessing the musical layers. *"Map your runs, track your progress and get the motivation you need to go even further. Hear mid-run cheers every time your friends like or comment on your run status, or outrun them in a game of Nike+ Tag"* [10] (excerpt from Nike+ GPS description.)

Musical beat detection is the cornerstone of DSP. Various research has been done into different ways of performing BPM detection (explained in section 3.3). The most promising algorithm is the 'Beat This' implementation, based upon a filter-bank approach, from Rice University [6]. The implementation is straightforward, not requiring any extra hardware or performance than is offered on the iPhone. Furthermore, weighing the tradeoff between accuracy and complexity, this algorithm offers adequate results for what is required. Valtino Afonso [11], et al, employed ECG beat detection also using a filter bank-based approach. This enabled "time and frequency dependent analysis". This approach achieved a positive accuracy of 99.56%.

For activity recognition using the accelerometer Nishkam Ravi [8], et al harnessed the flexibility of machine learning to predict various movements, and received very strong results. Cliff Randell and Henk Muller [12], of the University of Bristol, proposed utilising clustering algorithms to differentiate between different activities, using only "Root Mean Square (RMS) and integration of the last 2 seconds of measurements", achieving an accuracy of 95%. Amit Purwar [13] et al, proposed using simple RMS to reduce 3-dimensional accelerometer data, and used simple thresholds to infer actions, attaining an 81% accuracy.

4 Initial Assessments

There were many different options for addressing the various challenges of the application. In this section we describe the work we performed to identify our favoured approach to each of motivation, iOS implementations, Fourier transforms, and movement awareness.

4.1 Motivation

The first assumption is what kind of transformations to music are the most effective to deliver a motivational message. These results will be used to notify the user when he is going out of sync with the required workout (as measured by the phone's hardware). The tests were performed on a treadmill with a laptop using the open source program Audacity to experiment with changing the sound to motivate the person to speed up. The following table presents the various tests and a quantitative rating awarded. The rating ranges from -5 (didn't work, even cause to go slower) to +5 (worked very well). A rating of 0 means that the test had no effect.

Test	**Rating**	**Comment**
Increasing BPM (tempo)	+3	This worked very well if the song played had initially been in the same speed as running. Increasing the tempo slowly is natural to speeding up.
Volume (increasing to increase speed)	-1	This had no effect on my results, one could argue that it was more annoying as the volume could get to unbearable volumes.
Phasing	-5	This did not cause me to want to go faster. The dreamy sound of phasing, if anything, made me want to stop.
Overlay delay	+1	Echo was interesting as it caused the beats to rumble (making the song seem faster), however this had little or no effect on speed.
Increasing pitch	+2	This experiment actually was motivational as it, without drawing too much attention to itself, made me feel that something was off with the sound and wanted to correct it (by increasing my speed).

The tests were performed with electronic music that had a 4/4 beat. The techniques were also employed on music without a steady beat (classi-

cal/ballad), and the effects were the same apart from increasing speed made a smaller difference (harder to tell change in speed without a persistent, driving beat). One of the tests ran was phasing. Phasing is when two identical tracks (the same song) are played overlapping themselves and gradually shift out of unison. The first thing that happens is an echo. As they begin to drift apart doubling within each beat occurs, then a ringing effect, and then the same effects happen backwards as the songs start falling back into the same phase.

4.2 iOS Library Access

iOS 4.1 (the update to the operating system for Apple portable products released on September 8, 2010) added a number of new classes to the AV Foundation framework to provide an API for sample-level access to media. This section explores this API since it is not straightforward, nor obvious.

Primarily accessing samples using the AVAssetReader and AVAssetWriter classes was explored. They are a low-level API and thus have a small latency. They also enable taking chunks of a signal into a buffer, thus the adequate amount of samples can be chosen rather than loading the whole signal into memory (which might take in excess of 40 MB). This increases performance radically, and therefore, these classes were chosen for BPM analysis [14].

To modify samples being played, Audio Units, are going to be used. They enable, using the RemoteIO classes, to setup a chain of input to output units. A callback function is added before the output, which needs to deliver the samples to be played. There is a short 'window' of time before they need to be returned [15]. The exact length of the window is defined by function (3), where the size of the window is dependent on the buffer size. A typical implementation will use a buffer size of 512 or 1024 samples. A typical sampling rate for digital music is 44.1 kHz, and thus the length of the window will be 0.012 seconds for a 512 frame buffer and 0.023 seconds for 1024 [16].

$$L_{window} = \frac{B_{size}}{S_{rate}} \tag{3}$$

where
L_{window} is the length of the window (seconds)
B_{size} is the size of the buffer
S_{rate} is the sampling rate (Hz)

This means that within the 0.023 seconds all of the processing needs to occur on the samples. If this time is exceeded and the function does not

return the phone will not output any sound. Thus it is advised to use C to write the callback function and not allocate any memory, take locks, or perform any redundant things when executing inside the callback [16].

4.3 Hardware Accelerated FFT

Signal processing relies heavily on working with time-domain signal, as-well as frequency-domain, for manipulation based on frequency (such as low pass or high pass filters). To implement a filtering rhythm detection algorithm, calculating the DFT is required. The most commonly used algorithm for doing so is the FFT (Fast Fourier Transform) algorithm. The benefit of the FFT algorithm is the complexity; it reduces the required computations for N points from $\mathcal{O}(2N^2)$ to $\mathcal{O}(2Nlog_2(N))$, thus decreasing the computational burden [17]. Refer to figure 6 for a graphical complexity comparison.

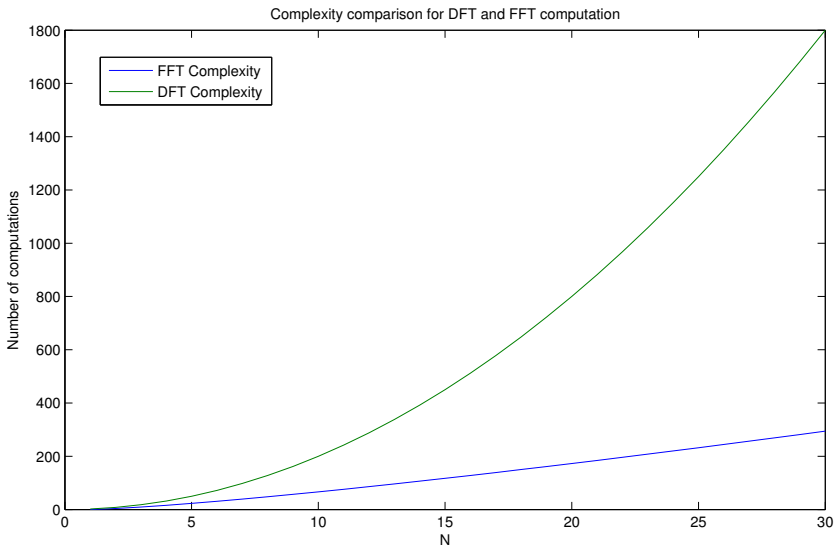


Figure 6: The complexity comparison for the DFT and FFT algorithm shows that the FFT algorithm is a far better (in terms of complexity) algorithm for computing the DFT. The DFT complexity is $\mathcal{O}(2N^2)$, whereas the FFT complexity is $\mathcal{O}(2Nlog_2(N))$. With an increasing input (N), the FFT complexity far outweighs the DFT complexity.

FFT Implementation

Apple, through its Accelerate Framework, offers a hardware accelerated API for performing FFT transforms. The downside, however, is that the documentation is of poor quality, and truly understanding how the API works requires excessive experimentation. I have done both, and have finally achieved fast FFT computation on the iPhone. The confusing part comes from the preparation of the FFT environment. An *FFTSetup* object needs to be created first to create appropriate twiddle factors:

> "To boost performance, vDSP functions that process frequency-domain data expect an array of complex exponentials (sometimes called twiddle factors) to exist prior to calling the function. Once created, this FFT weights array can be used over and over by the same Fourier function and can be shared by several Fourier functions [18]".

One of the parameters to *vDSP_create_fftsetup(...)* is a log2n argument:

> "Argument log2n to these functions is the base-2 logarithm of n, where n is the number of complex unit circle divisions the array represents, and thus specifies the largest number of elements that can be processed by a subsequent Fourier function. Argument log2n must equal or exceed argument log2n supplied to any functions using the weights array. Functions automatically adjust their strides through the array when the table has more resolution, or larger n, than required [18]".

This step thus causes a sound signal, S_{input} of length l to be modified by padding with zeros to get signal S_{fft} to be exactly n in length, following conditions 4 and 5.

$$\forall x \in N.(x < n \wedge x < l \implies S_{fft}[x] = S_{input}[x]) \tag{4}$$

$$\forall x \in N.(x < n \wedge x > l \implies S_{fft}[x] = 0) \tag{5}$$

I wrote a separate FFT applier class which took in a signal of length $< n$, and performed the padding transformation 4 and 5. This is shown in listing: 1.

Listing 1: Performing the transformation

```
int toZeroFill = N - L;
memcpy(fftA.realp, sampleBuffer, L*sizeof(float));
memset(fftA.realp+L, 0, toZeroFill);
```

Any algorithm that then wishes to use the data in the frequency domain has to operate on the whole n-length signal. Finally when transforming back to time domain, the whole n-length signal is input, giving an output of length n, following conditions 4 and 5. Thus, only the signal up to l is used (to avoid using the padded zeros). The whole code listing for FFT Applier can be found in appendix 1.

Listing 2: Performing FFT and IFFT

```
// Perform the forward FFT
vDSP_fft_zip(fftSetup, &fftA, stride, log2n, kFFTDirection_Forward);

// Perform the Inverse FFT
vDSP_fft_zip(fftSetup, &fftA, stride, log2n, kFFTDirection_Inverse);

//we need to scale the output as per documentation instructions
//by a factor of n. (thus every item will be divided by n).
float scale = (float)1.0 / N;

// scale
vDSP_vsmul(fftA.realp, 1, &scale, fftA.realp, 1, numberFrames);
vDSP_vsmul(fftA.imagp, 1, &scale, fftA.imagp, 1, numberFrames);
```

4.4 Movement Awareness

The possible movement awareness hardware that can be harnessed to give insight to the person exercising is:

- Accelerometer
- GPS
- Gyroscope
- External, more accurate GPS/Accelerator/Gyroscope device
- External heart monitor

For local movement awareness the accelerometer was chosen as the most useful tool because it is available on all Apple devices. The accelerometer gives information about changes in acceleration in the three axes. Figure 7 presents data from the accelerometer whilst running, which shows that the signal expresses periodicity, and thus can be analysed for tempo. Gravity, the omnipresent force, is always expressed in at least one axis. Other forces, such as when a person moves the phone up, are picked up by the accelerometer as changes in acceleration. This gives the basics of being able to discern when the phone is moved. The GPS can be used to track how far the person has moved, to find global movement patterns. The Gyroscope

is, unfortunately, only available on newer Apple devices, thus was not currently considered. It will definitely give more insight into the user's local movement, and thus could used as a possible extension to the project. An external device is unfeasible at this time as designing one takes money and time. Such a device, however, would also greatly benefit the results. Finally, an external heart monitor is a potential extension to the project as playing music in the rate of someones heart beat (or by taking it into consideration) would be an interesting way to deliver motivation.

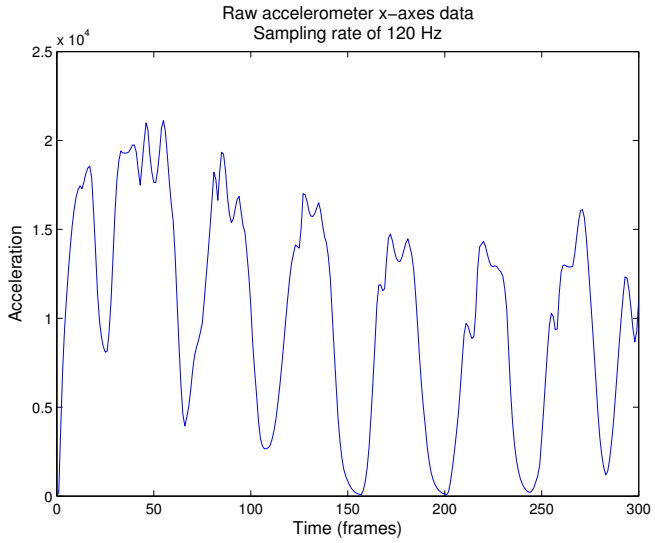


Figure 7: Raw accelerometer data showing 8 peaks corresponding to changes in acceleration. The data was recorded in the hand of a person running, as shown with the sinusoidal motion. Since the accelerometer picks up changes in acceleration, when the person is hitting of the ground the acceleration is positive, and when starts to fall back down it is negative. This data shows that running expresses periodicity, and thus can be analysed to find the underlying tempo.

4.5 BPM Detection Algorithm

In the beginning a statistical streaming beat detection algorithm was implemented. The algorithm was a C++ implementation from the open-source audio processing library, SoundTouch. The beat detection algorithm per-

formed the following steps [19].

1. Input chunks of samples to the algorithm to analyse (8192 samples).
2. The input sound frames are decimated to approximately 500 Hz to reduce the amount of frames on which to perform calculations. This can be done as mostly low frequency sound defined the beat rate (drum kicks, for instance). The signal is decimated by removing every n^{th} frame.
3. The decimated signal is then enveloped. Enveloping detects the shape of the amplitude by removing signals that are below a certain threshold. The threshold is calculated as the sliding average multiplied by a scalar value. This only leaves peaks in the signal.
4. An autocorrelation function is ran on the signal, which returns the sound patterns in the enveloped signal.
5. After the whole sample has been analysed, the algorithm finds the precise location of the highest peak of the autocorrelation function and converts to BPM.

The results of the algorithm, however, were not good enough. The following table presents the results of running the algorithm on various songs. The real BPM value was found by running the sound through an acclaimed beat detection implementation in the Serato Software.

Song Title	**Artist**	**Genre**	**Real BPM**	**Approx. BPM**	**Mean Square Error**
An American in Paris	STP Ft. Kevin Yost	Chill House	124	125.299	1.687
Maria	Blondie	Rock, Pop	160	35.125	15593.8
Amazing	Seal	Upbeat, Pop	121	123.99	8.94
In My Arms	Kylie Minogue	Pop	128	125	9
I Found You	Axwell	House	130	146.5	272.25
Good Vibrations	The Beach Boys	Rock	77	150	5329
Got 2 B U	Solar House	House	124	123.99	0.0001
I Will Survive	Gloria Gaynor	Disco	117	115.5	2.25
One More Time	Daft Punk	House	125	125	0

As can be seen the mean square error of the algorithm is good for electronic music, however sometimes the algorithm returns nearly half of the BPM (found the half-beat).

The algorithm was then run on accelerometer data, to find the tempo of the person exercising. Because the algorithm implemented decimation throughout the code and assumed a 44.1 kHz sampling rate, a lot of code had to be rewritten for the first steps to work. Ultimately the whole code would have to be changed, and since the accuracy of the algorithm was not good enough, it was decided to try a filtering rhythm detection algorithm.

4.6 Choices

The initial assumptions looked at motivation, iOS library access, hardware accelerated routines, movement awareness, and the Soundtouch BPM analysis implementation. The following sections will utilise the previous parts in the implementation. The BPM detection implementation provided by Soundtouch was, unfortunately, deemed unusable. A custom bespoke implementation was thus created, explained in the forthcoming section.

5 Implementation

The following section presents the implementation of the points talked about in section 3 and 4.

5.1 Design

This section describes the design of the proposed solution in a class diagram. Figure 8 presents the overview of the class diagram, whereas figure 9 presents the detailed class diagram (spanning three pages). Detailed explanations will follow.

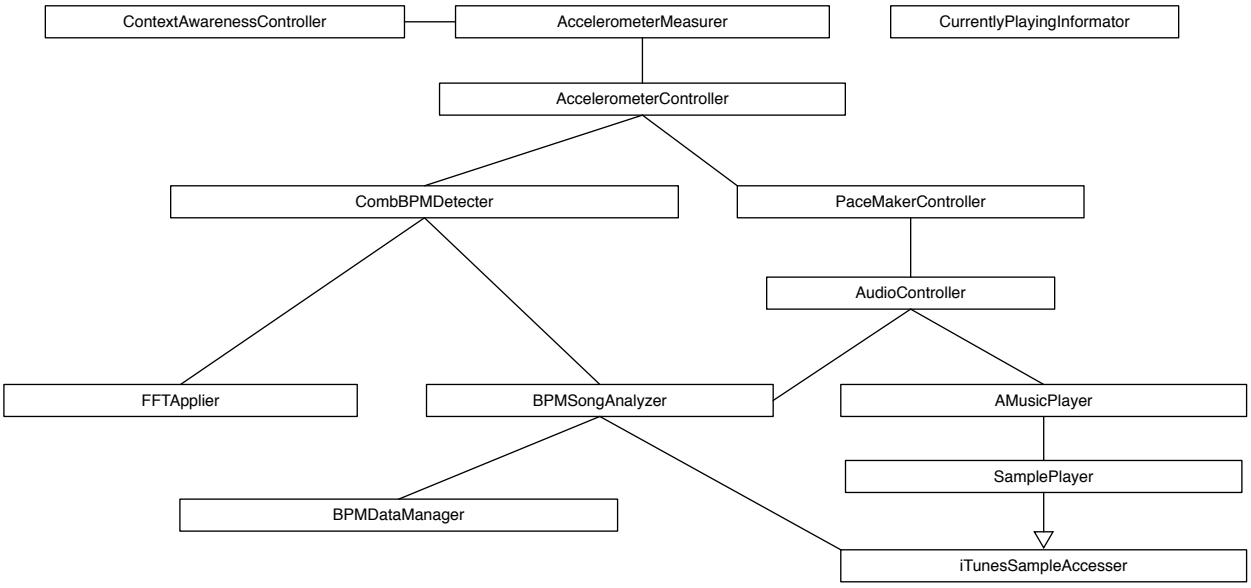
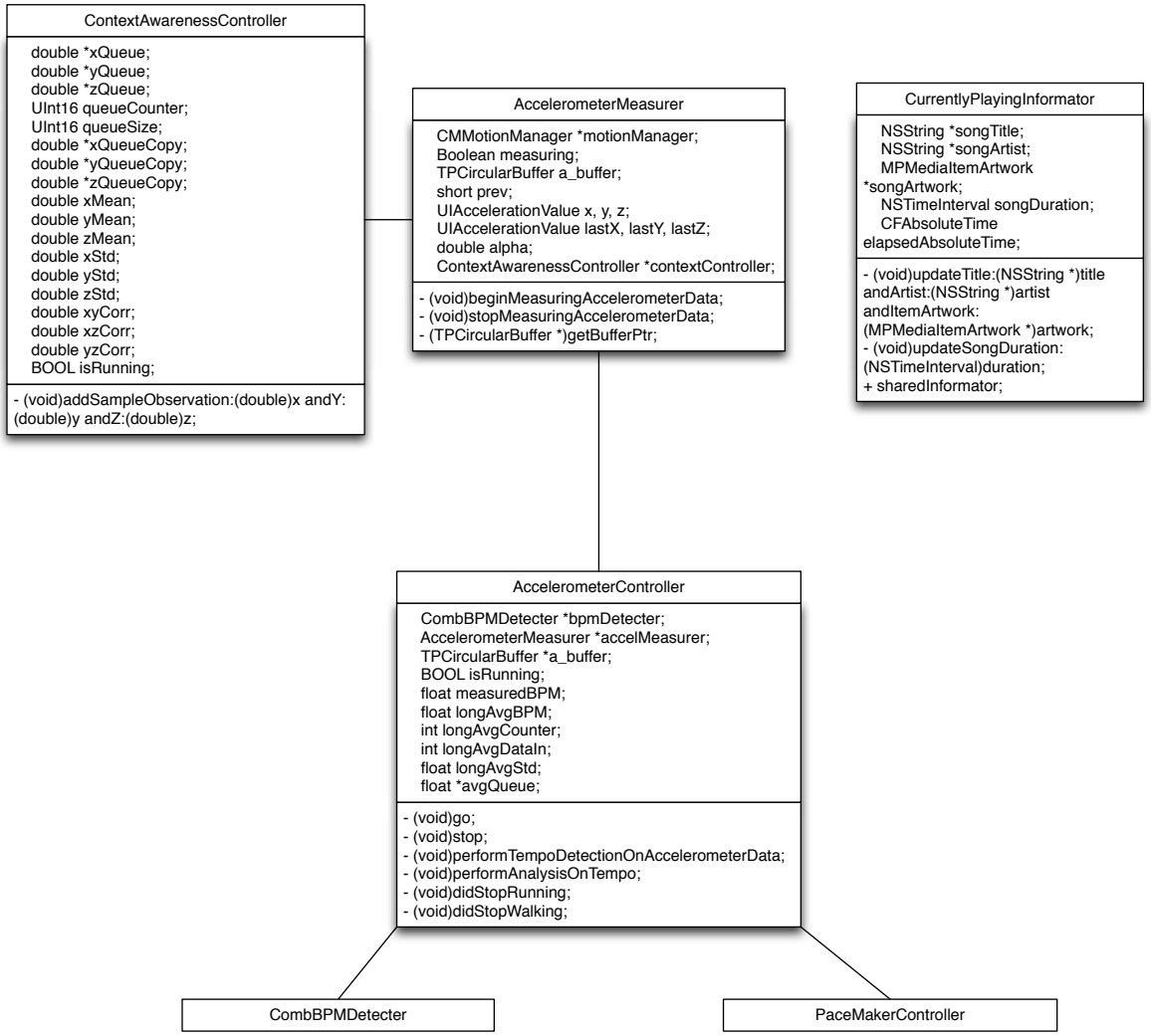
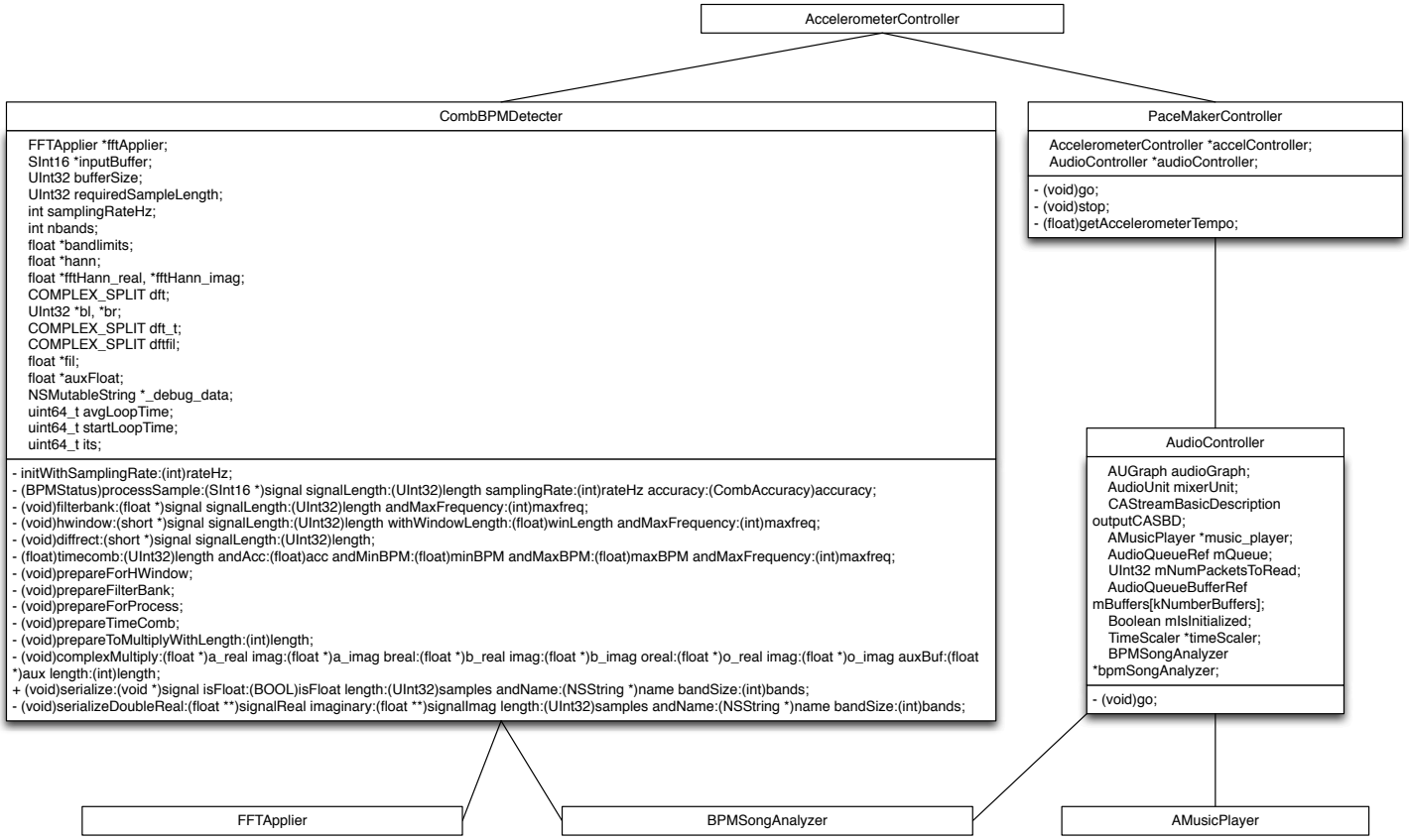


Figure 8: The main class diagram, without methods or ivars. This figure shows the associations between the classes, figure 9 presents the full class diagram including the methods and ivars.


ContextAwarenessController
double *xQueue;
double *yQueue;
double *zQueue;
UInt16 queueCounter;
UInt16 queueSize;
double *xQueueCopy;
double *yQueueCopy;
double *zQueueCopy;
double xMean;
double yMean;
double zMean;
double xStd;
double yStd;
double zStd;
double xyCorr;
double xzCorr;
double yzCorr;
BOOL isRunning;
- (void)addSampleObservation:(double)x andY:
(double)y andZ:(double)z;
AccelerometerMeasurer
CMMotionManager *motionManager;
Boolean measuring;
TPCircularBuffer a_buffer;
short prev;
UIAccelerationValue x, y, z;
UIAccelerationValue lastX, lastY, lastZ;
double alpha;
ContextAwarenessController *contextController;
- (void)beginMeasuringAccelerometerData;
- (void)stopMeasuringAccelerometerData;
- (TPCircularBuffer *)getBufferPtr;
CurrentlyPlayingInformator
NSString *songTitle;
NSString *songArtist;
MPMediaItemArtwork
*songArtwork;
NSTimeInterval songDuration;
CFAbsoluteTime
elapsedAbsoluteTime;
- (void)updateTitle:(NSString *)title
andArtist:(NSString *)artist
andItemArtwork:
(MPMediaItemArtwork *)artwork;
- (void)updateSongDuration:
(NSTimeInterval)duration;
+ sharedInformator;
AccelerometerController
CombBPMDetecter *bpmDetecter;
AccelerometerMeasurer *accelMeasurer;
TPCircularBuffer *a_buffer;
BOOL isRunning;
float measuredBPM;
float longAvgBPM;
int longAvgCounter;
int longAvgDataIn;
float longAvgStd;
float *avgQueue;
- (void)go;
- (void)stop;
- (void)performTempoDetectionOnAccelerometerData;
- (void)performAnalysisOnTempo;
- (void)didStopRunning;
- (void)didStopWalking;
CombBPMDetecter
PaceMakerController


AccelerometerController
CombBPMDetecter
FFTApplier *fftApplier;
SInt16 *inputBuffer;
UInt32 bufferSize;
UInt32 requiredSampleLength;
int samplingRateHz;
int nbands;
float *bandlimits;
float *hann;
float *fftHann_real, *fftHann_imag;
COMPLEX_SPLIT dft;
UInt32 *bl, *br;
COMPLEX_SPLIT dft_t;
COMPLEX_SPLIT dftfil;
float *fil;
float *auxFloat;
NSMutableString *_debug_data;
uint64_t avgLoopTime;
uint64_t startLoopTime;
uint64_t its;
- initWithSamplingRate:(int)rateHz;
- (BPMStatus)processSample:(SInt16 *)signal signalLength:(UInt32)length samplingRate:(int)rateHz accuracy:(CombAccuracy)accuracy;
- (void)filterbank:(float *)signal signalLength:(UInt32)length andMaxFrequency:(int)maxfreq;
- (void)hwindow:(short *)signal signalLength:(UInt32)length withWindowLength:(float)winLength andMaxFrequency:(int)maxfreq;
- (void)diffrect:(short *)signal signalLength:(UInt32)length;
- (float)timecomb:(UInt32)length andAcc:(float)acc andMinBPM:(float)minBPM andMaxBPM:(float)maxBPM andMaxFrequency:(int)maxfreq;
- (void)prepareForHWindow;
- (void)prepareFilterBank;
- (void)prepareForProcess;
- (void)prepareTimeComb;
- (void)prepareToMultiplyWithLength:(int)length;
- (void)complexMultiply:(float *)a_real imag:(float *)a_imag breal:(float *)b_real imag:(float *)b_imag oreal:(float *)o_real imag:(float *)o_imag auxBuf:(float *)aux length:(int)length;
+ (void)serialize:(void *)signal isFloat:(BOOL)isFloat length:(UInt32)samples andName:(NSString *)name bandSize:(int)bands;
- (void)serializeDoubleReal:(float **)signalReal imaginary:(float **)signalImag length:(UInt32)samples andName:(NSString *)name bandSize:(int)bands;
PaceMakerController
AccelerometerController *accelController;
AudioController *audioController;
- (void)go;
- (void)stop;
- (float)getAccelerometerTempo;
AudioController
AUGraph audioGraph;
AudioUnit mixerUnit;
CAStreamBasicDescription outputCASBD;
AMusicPlayer *music_player;
AudioQueueRef mQueue;
UInt32 mNumPacketsToRead;
AudioQueueBufferRef mBuffers[kNumberBuffers];
Boolean mIsInitialized;
TimeScaler *timeScaler;
BPMSongAnalyzer *bpmSongAnalyzer;
- (void)go;
FFTApplier
BPMSongAnalyzer
AMusicPlayer

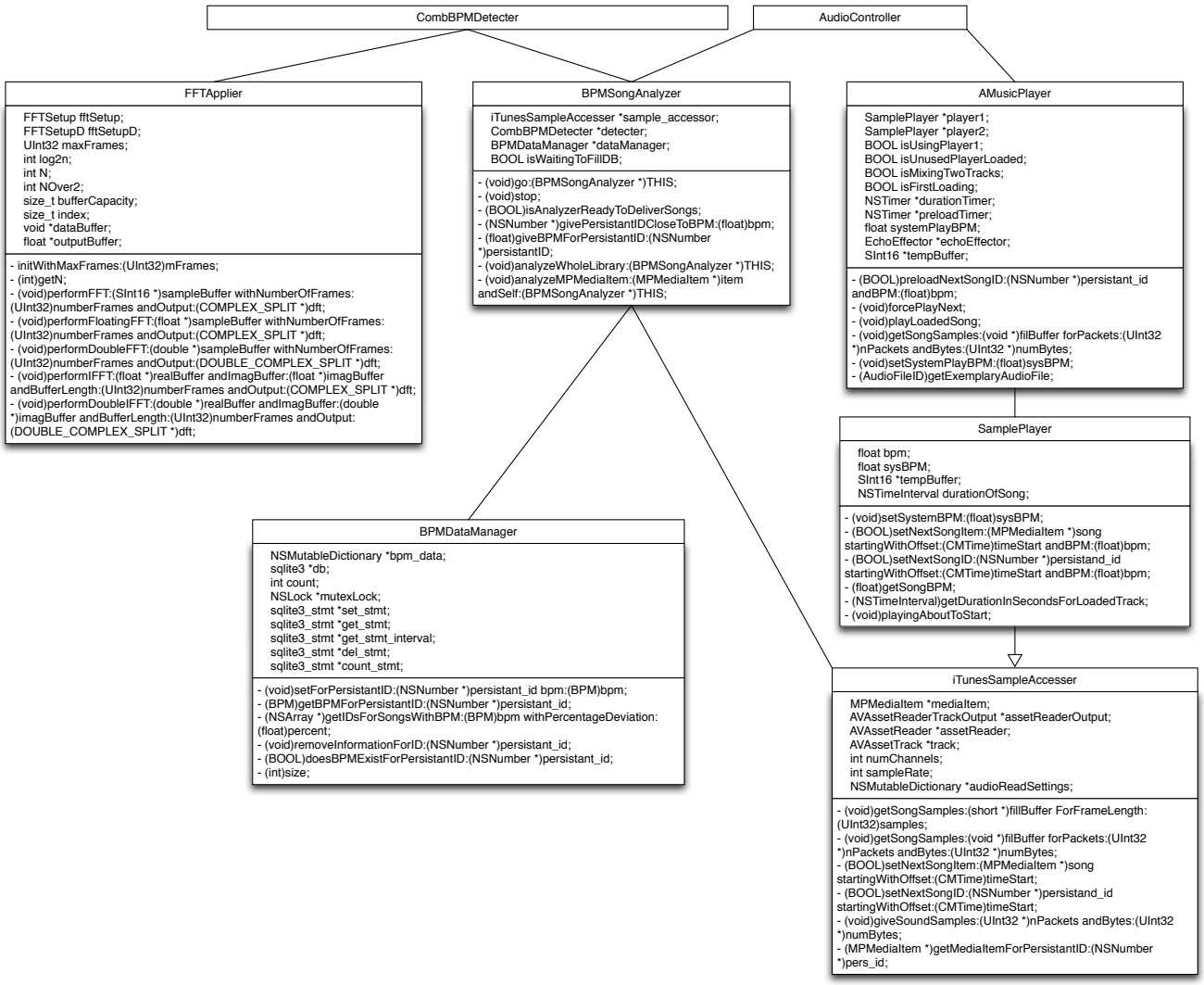


Figure 9: The full class diagram of the proposed solution.

Figure 9 presents the class diagram of the solution showing the various associations between the classes. This part briefly describes the purpose of each class.

ContextAwarenessController This controller receives accelerometer updates from the AccelerometerMeasurer class, and performs activity

recognition as described in section 5.3.

AccelerometerMeasurer This is the class that received motion updates (accelerometer data) using the CMMotionManager class. It performs smoothing of the accelerometer data and places it into the circular buffer that it shares with the AccelerometerController.

CurrentlyPlayingInformator This class is a *singleton* which maintains app-wide information about the song currently being played. The data can be written to from any class and read by any class. This is important as this doesn't require an association between the views and the music playing controllers, thus decoupling logic from presentation.

AccelerometerController The main job of this controller is to coordinate the accelerometer data processing tasks. This class has the AccelerometerMeasurer, which it instruments to collect raw accelerometer data. Every 5 seconds this class performs analysis on the accelerometer data, using the CombBPMDetecter class.

CombBPMDetecter This is the main class of the project, as it performs the tempo analysis on signals. Both the AccelerometerController and the BPMSongAnalyzer use this class (their own instances) to perform analysis on accelerometer and sound signals, respectively. The majority of the methods of this class relate directly to section 5.4, which explains in detail the stages of the BPM analysis algorithm.

PaceMakerController This is top-level class which executes methods on other classes, such as to start analysing songs in the background, to start playing music, and to setup other controllers.

AudioController This is the controller responsible for audio. It has an association to the BPMSongAnalyzer class and tells it to perform or stop analysis. Furthermore it also acts as the middle-man between the BPMSongAnalyzer and the AMusicPlayer, dictating which songs to play and when.

AMusicPlayer This music player has two SamplePlayer's that it loads with songs, as per AudioController instructions, so that there is always a ready player to start playing the next song. This also enables the music player to perform mixing of the two songs. It is also in charge of managing when to inform the higher controllers that a new song needs to be loaded, and when a song ends.

FFTApplier Fourier analysis is required for BPM detection, and this class performs it. It has various different methods to perform different kinds of Fourier analysis (on double or single precision floating numbers or the forward and inverse transforms).

BPMSongAnalyzer The analyser coordinates the background analysis of songs. It uses the BPMDataManager class to serialise and retrieve BPM values. This class also tells higher level classes which songs are available within range of a specific BPM value.

BPMDataManager This class interfaces with the SQLite database which maintains information on the BPM values of different songs.

iTunesSampleAccesser This class performs low level access to samples. It has a method by which samples are placed into a provided buffer. This class also has basic information about the item it is playing.

SamplePlayer This class is a subclass of the iTunesSamplesAccesser which is used to play music. It therefore contains more information about the song being played, such as it's computed BPM and duration. This player also uses the SoundTouch library to change the tempo of the song, and reflect this in it's duration. Furthermore, this class also writes currently playing songs to the CurrentlyPlayingInformator.

5.2 Accelerometer Data for User Pace (BPM)

The accelerometer was the main hardware input for analysis of the user's tempo. It provides information on the change in acceleration in the 3 axes. There were a few things to consider with using this data:

Noise The accelerometer data can have lots of noise - gravity, spurious movements, factors arising from phone shift.

Dimensionality Reduction The accelerometer provides 3-axis data, whereas the tempo algorithm expects a single number.

Accuracy We want to extract the periodicity of movement, and need to attain this from a change in acceleration reading.

Points 1 and 2 were tackled together. Figure 10 displays raw accelerometer data when jumping around with the phone in the pocket. The motion measured was running, which as in Figure 5 expressed periodicity. However, because there are three axes, and the rotation of the phone cannot

be assumed, the periodicity will be expressed in some (or all) of the axes. Therefore the reduction operation, which takes as input the three axis values, and output 1 value, needs to take into account this fact.

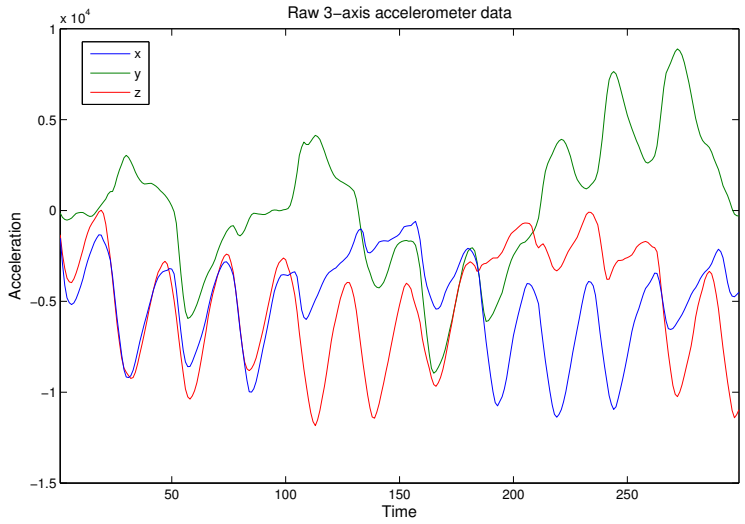


Figure 10: Raw accelerometer data for the 3-axis. The motion measured was running, which as in Figure 7 expressed periodicity. However, because there are three axes, and the rotation of the phone cannot be assumed, the periodicity will be expressed in some of the axes. Therefore the reduction operation, which takes as input the three axis values, and outputs 1 value, needs to take this into account.

For simplicity, it is assumed that at least one axis will express periodicity, because situations with no movement will be filtered out in a previous stage. The approach taken in the implementation was to low-pass filter the signal. Equation 6 presents the low pass filter. In the implementation the Δt is $1/60$ (sampling rate) and RC is $1/5$. This gives the input contribution factor of 0.077 and previous output factor of 0.923. This means that the new sample x_i will only factor 7.7% to the final value and the previous processed sample, y_{i-1}, will factor 92.3%. The new sample will not influence the final outcome too much. This is precisely the point of the low-pass filter. Large changes (high frequencies) are to be attenuated. This thus decreases any big changes in the signal, hence leaving lower frequencies intact. Figure 11 presents the discrete low pass filter on exemplary accelerometer data.

$$y_i = \overbrace{\left(\frac{\Delta t}{RC + \Delta t}\right) x_i}^{\text{input contribution}} + \overbrace{\left(\frac{RC}{RC + \Delta t}\right) y_{i-1}}^{\text{previous output factor}} \tag{6}$$

where
y_i is the i$^{\text{th}}$ output sample
x_i is the i$^{\text{th}}$ input sample
Δt is the sampling rate $(1/f_s)$. f_s is the sampling frequency, in hertz.
RC is a time constant, originating from a RC circuit. We use $1/5$

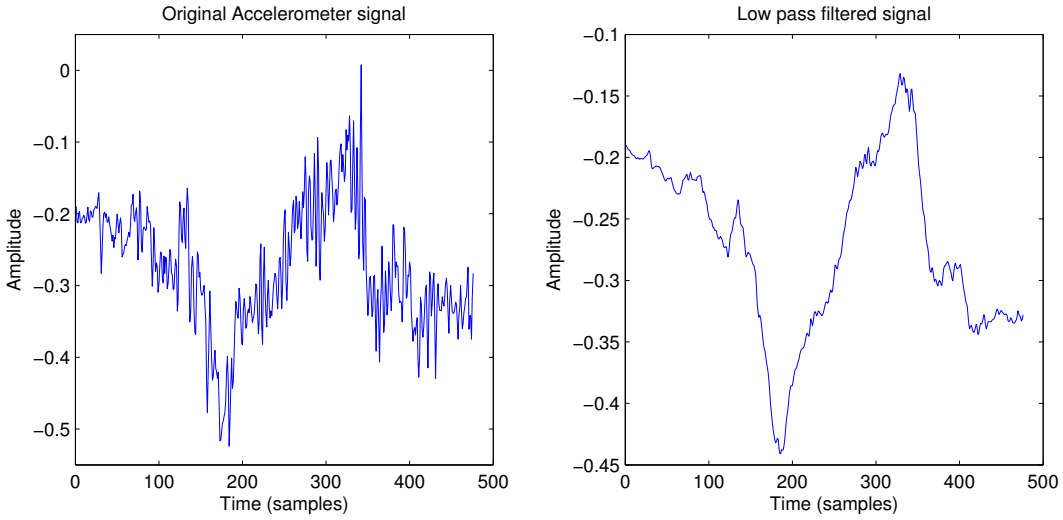


Figure 11: The image presents a low pass filter applied to an accelerometer signal. The left image presents the raw signal taken from the accelerometer, and the right image presents the low-pass filtered signal, following equation 6. 2 of the 3 goals are met: noise reduction and periodicity (accuracy). Spurious movements are removed from the signal, removing noise. Furthermore the filtered signal expresses a more clear periodicity, one which the BPM detection algorithm can better pickup.

The reason for using the low-pass filter is that it attenuates (reduces the amplitude of) high-frequency signals. This is in contrast to the high-pass filter which attenuates low-frequency signals. Because movement in running is expressed mostly in low frequencies, such a filter would not work.

Figure 12 presents the high pass filter on our exemplary accelerometer signal. As can be seen no periodicity is expressed in the signal (mostly just high frequency noise). A further potential development would be a band pass filter, which removes frequencies at both sides of the spectrum using specific cutoff frequencies. This is a future development, since a deeper analysis would need to be made in order to find the adequate cutoff frequencies.

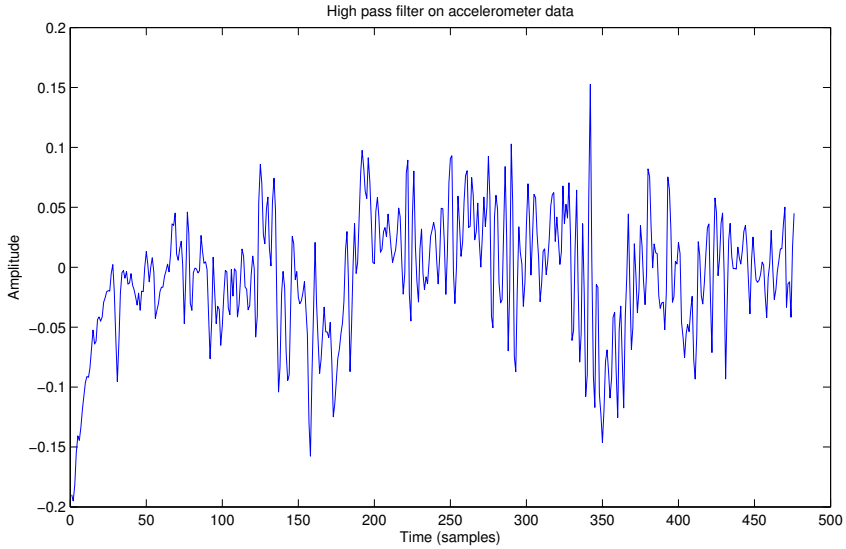


Figure 12: The image presents a high pass filter applied to our accelerometer signal. As can be seen no periodicity is expressed in the signal (mostly just high frequency noise). This filter is not adequate for analysing periodicity when running or exercising, because such a motion is expressed in the lower frequencies of the motion spectrum.

The final step is to reduce the dimensions of the signal from 3 to 1 (3 dimensional axis data into a single dimension). This is done by taking the length of vector of each sample, following equation 7. This is the Euclidian normalisation of a vector, which gives the distance from 0 for the three dimensional point.

$$s_i = \sqrt{x_i^2 + y_i^2 + z_i^2} \tag{7}$$

where
s_i is the i$^{\text{th}}$ output sample
x_i, y_i, z_i are the i$^{\text{th}}$ x, y, and z samples

5.3 Accelerometer Data for Activity Recognition

The following section explains the implementation of activity recognition as described in section 3.3. The approach taken by Nishkam et al [8] was to use artificial intelligence classifiers to discern between various activities. In order to simplify the implementation of activity recognition for the purpose of achieving a working demo, the learning step was omitted. Rather than learning dictating activities, a simple threshold of parameters was used. In order to decrease errors that creep into accelerometer data, a window of 2 seconds was used. For each accelerometer axis (X,Y,Z) a queue is created that stores 2 seconds worth of data (at 60 Hz sampling rate equals 120 samples). The context is thus calculated every 2 seconds, giving enough time for the buffers to be filled[3]. When the buffers are full the mean and standard deviation is calculated, and the logic follows listing 3.

Listing 3: Activity Recognition Logic

```
double avgStd = (xStd + yStd + zStd) / 3;
if (avgStd > 0.25) {
    // now we are running
    if (!isRunning) {
        isRunning = YES;
        [self informStoppedWalking];
    }
} else if (avgStd < 0.1) {
    // now we stopped (are walking, or slow)
    if (isRunning) {
        isRunning = NO;
        [self informStoppedRunning];
    }
}
```

5.4 BPM Analysis

A signal going through the BPM analysis implementation has several stages. Each stage is examined with diagrams showing an exemplary signal transformation.

[3] In the worst case scenario, it will take the phone 2 seconds to realise that the user has changed activity. From experimentation it shows that this is an adequate time interval.

Overview

Figure 13 presents the flow chart of the BPM analysis algorithm. Each of the stages are reflected in the processes.

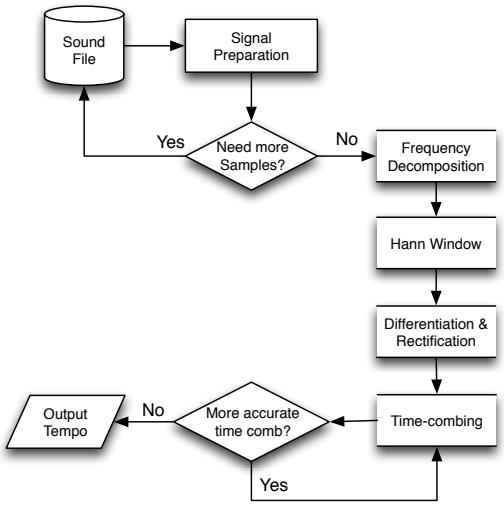


Figure 13: The flowchart of the BPM analysis stage. The samples are fed into the signal preparation method, and when there are enough the algorithm runs frequency decomposition, hann window, differentiation and rectification, and time-combing. The time-comb algorithm can be called iteratively to gain better results. Finally the tempo is returned.

Signal preparation

Signal preparation is the stage where, at first, the signal is read into buffers. The filtering rhythm algorithm matches the beats with a time comb, thus requiring the sample analysed to contain at least 2 beats. Equation 8 displays the relation between the required sample length and the sampling rate. Thus in our case, with a 44.1kHz sampling rate, and the lowest BPM tested for of 70, we require 75600 samples.

$$L_{sample} = \lfloor \frac{120 * f_s}{\beta} \rfloor \tag{8}$$

where
L_{sample} is the length of the sample (in samples)

f_s is the sampling rate of the file (in Hz)
β is the minimum BPM being tested)

Once the buffers are filled, the data is converted from 16-bit signed integers into floating point numbers, for ease of computation. Figure 14 presents an exemplary signal.

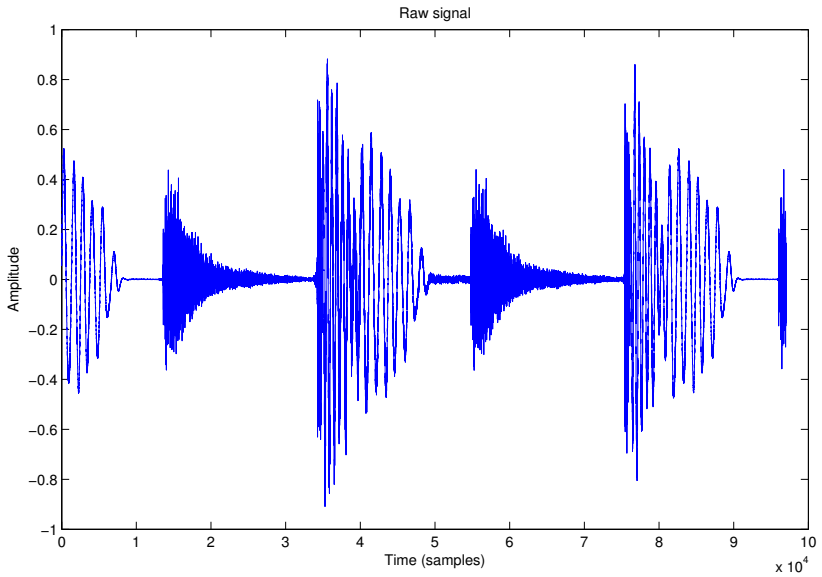


Figure 14: Raw signal (in time domain) representing 2.2 seconds of the song (97020 samples at 44.1 kHz sampling rate). This is the raw signal that is fed into the buffers of the BPM analysis engine. The tempo of the song is calculated based upon this sample. The length of the signal is enough to withhold at least 2 beats (as seen on the image).

Frequency decomposition

Frequency decomposition, or filter banking the signal, is the second stage. Here the FFT of the signal is taken, and separated into 6 different bands based upon frequency:

- 0 - 1000 Hz
- 1000 - 4000 Hz

- 4000 - 6000 Hz
- 6000 - 8000 Hz
- 8000 - 10000 Hz
- 10000 - 44100 Hz

The values for the different bands were taken from Cheng et al [6] and music theory specifying where certain instruments are expressed (in frequency). Image 15 displays the filter-banked signal in both the time and frequency domains.

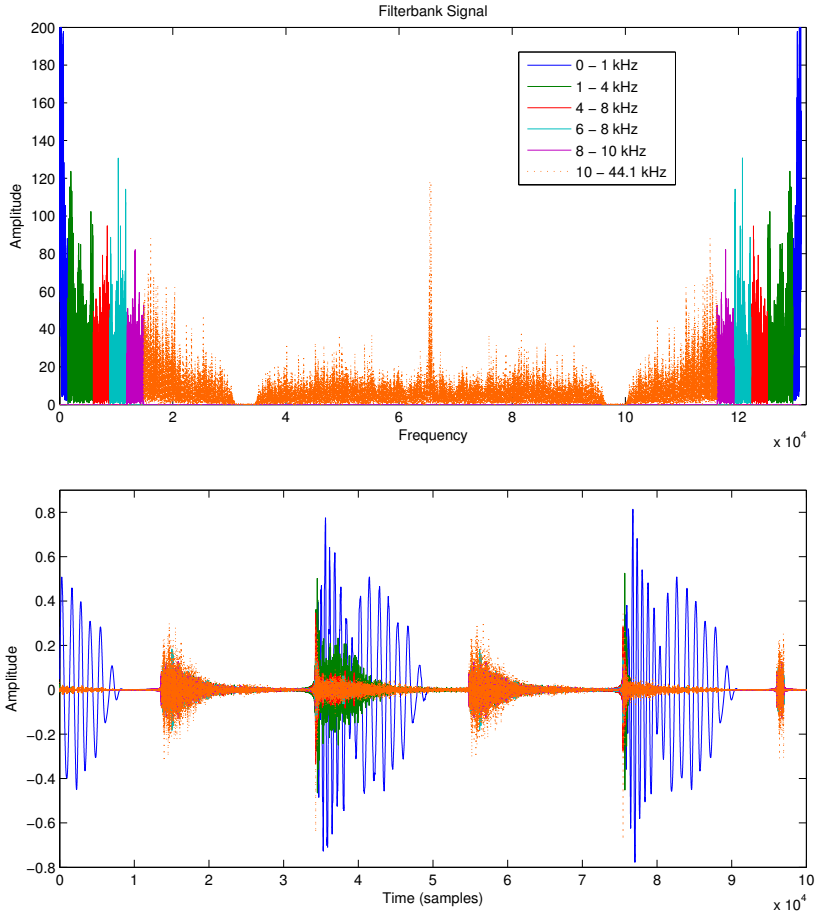


Figure 15: The top image presents the signal in the frequency domain, expressed in the different frequency bands (as shown in the legend). The image is mirrored at the centre, thus the blue bands represent the lowest frequencies (0-1 kHz), whereas the orange band represents the highest signal (10-44.1 kHz). The bottom image represents the same signal in the time domain, maintaining the frequency band colour scheme. Thus the blue band once again corresponds to the lowest frequencies (this is a drum kick).

Hann window

A Hann window is "an apodization function, also called the Hann function, frequently used to reduce aliasing in Fourier transforms" [20]. An apodiza-

tion function is one that changes the shape of a mathematical function or signal. The function creates a smoothing of the signal around a specific interval. The implemented signal only performs a half Hann window. The purpose of this operation is to smooth the signal's end, since it is cut from the middle of the song, and thus could have noise. Before the Hann window is applied on the signal, full-wave rectification occurs. Full wave rectification is taking the absolute value of the signal, such that it is bound from 0. Because the algorithm attempts to find moments of great change in the signal, the sign of the signal is unimportant. Figure 16 shows the output of the Hann window stage on the signal.

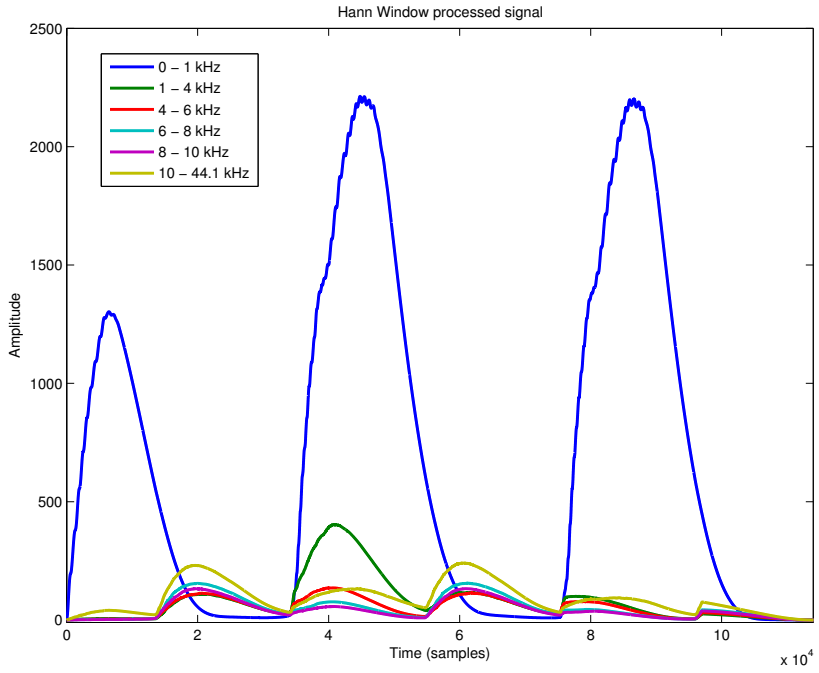


Figure 16: The signal after the Hann window transformation, expressed in the particular frequency bands (in time domain). The signal first is full wave rectified - taking the absolute value of the signal, such that it is bound, from 0. It then is multiplied with the Hann window to smooth the signals end. The highest amplitude band corresponds to the 0 - 1 kHz frequency band. This is the lowest frequencies, or the bass. In images from previous stages this band had the highest amplitude, thus this transform preserves the particular amplitudes.

Differentiation and Rectification

The penultimate stage is differentiation and half wave rectification. First the signal is differentiated, so that only the change in sample amplitude are shown. The signal thus maintains negative changes (when going from maxima to minima, i.e. from higher amplitude to lower). Therefore the signal is half-wave rectified to remove any values that are lower than zero. Figure 17 shows this stages transformation on the signal.

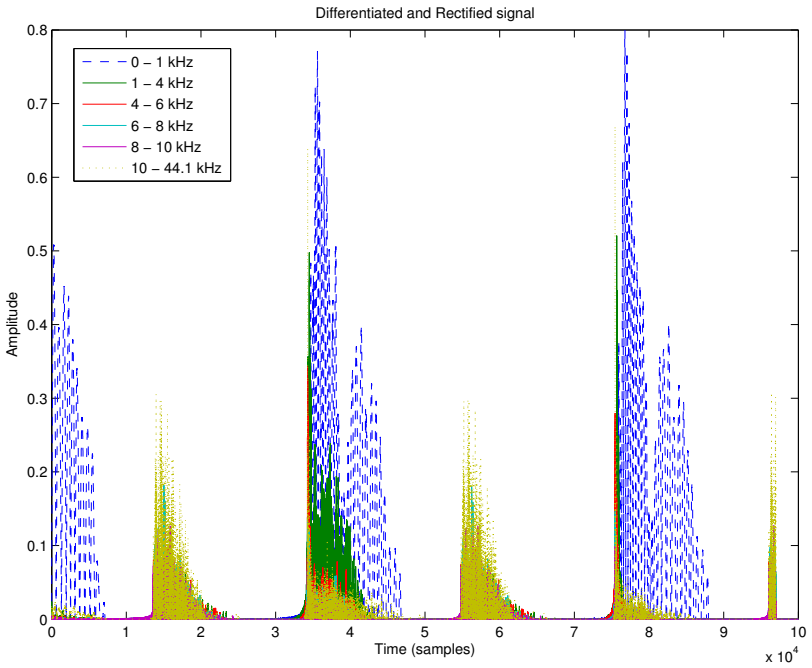


Figure 17: The differentiated and half-wave rectified signal, expressed in the particular frequency bands (in time domain). The signal was differentiated and half-wave rectified (so that only positive differences are recorded). This causes the output to have maximum amplitude in places where the signal sharply increased (such as introductions of a beat).

Time-combing

Time combing is the last stage of the BPM analysis algorithm. It is also the most computationally demanding. In this stage the processed signal (by the filter bank, Hann window, and differentiation and rectification stages) is analysed to determine the underlying BPM.

Overview The inputs to the time comb stage are the minimum BPM (*minBPM*), the maximum BPM (*maxBPM*), and the *increment factor*. The function analyses the BPMs from the *minBPM* up to the *maxBPM* every *increment factor*. The function is called with more accurate results, thus finding improving approximate BPMs. Listing 4 displays computation 3

levels deep. The first level inputs the *minBPM* and *maxBPM* as specified by the system, and the next calls use the resulting BPM from the first call ± an offset (0.5 and 0.01).

Listing 4: Iterative time comb

```
float bpm;
bpm = [self timecombAcc:1 andMinBPM:70 andMaxBPM:165];
bpm = [self timecombAcc:0.5 andMinBPM:bpm-2 andMaxBPM:bpm+2];
bpm = [self timecombAcc:0.01 andMinBPM:bpm-0.1 andMaxBPM:bpm+0.1];
```

Detail Each iteration performs the same calculation, but with different input. Firstly, the sound signal is transformed into the frequency domain. Then a for loop is created which goes through every BPM in the range of *minBPM* to *maxBPM* by increment factor.

1. The variable e_{max} is set to zero. This variable represents the highest observed energy (for determining the max energy).

2. The variable s_{bpm} is set to zero. This variable will contain the BPM giving the highest energy (e_{max}).

3. For each *tested BPM* starting from *minBPM* to *maxBPM* incrementing by the *increment factor*

   (a) A time comb is created using equation 9 for the desired BPM being tested (*tested BPM*). It uses the N_{step} values to create 3 time combs (as shown in figure 4). The time comb is created by setting the samples at position $N_{step} * \gamma$ to 1, where γ is the iteration ($0 \leq \gamma \leq 2$). The rest of the samples are 0. This time comb, therefore, corresponds to the tested BPM value, where each samples of value 1 is where the beat could occur.

   $$N_{step} = \lfloor \frac{f_s * 60.0}{bpm} \rfloor \tag{9}$$

   where
   N_{step} is interval (in samples) between beats at the given BPM
   f_s is the sampling rate (in Hz)
   bpm is the BPM being tested in the loop

   (b) The time comb is transformed to the frequency domain, thus both the signal and time comb are in the frequency domain.

(c) The signals (sound and time comb) are multiplied together. Since they are in the frequency domain, they are both complex number vectors. Thus a complex number multiplication occurs on each pair of samples (where $C_1 : a+bi$ is the first sample and $C_2 : c+di$ is the second, thus the product is $C_1 * C_2 = (ac-bd)+i(ad+bc)$).

(d) Now that the output is still a complex number vector, both components are squared and added together. Thus each sample $C_o : e + fi$ is transformed into $e^2 + f^2$, resulting in a real ($\mathbb{R}$) number.

(e) All of the real number samples are then summed together, and the sum is compared to the variable e_{max}. If the sum is greater than e_{max}, then s_{bpm} is set to the *tested BPM*, and e_{max} is set to the sum (now the highest energy observed). If the sum is not greater, the algorithm continues with the next *tested BPM*.

4. The s_{bpm} now holds the underlying BPM of the song, and is returned back so that another iteration of the time-comb algorithm can be called.

The sum of energies for each BPM, for our exemplary signal, is shown in Figure 18. The underlying BPM of this signal is 128.

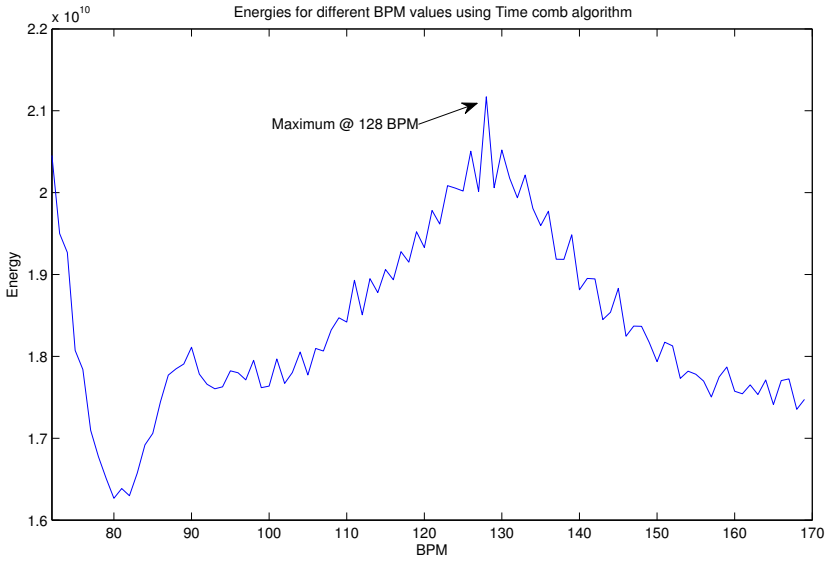


Figure 18: The energies calculated for different BPM values using the time comb algorithm. It can be noted that the maximum occurs at 128 BPM's, thus specifying the underlying BPM of the song. However, on the lower BPM's the energies start quickly climbing. This is because the signal also finds high energies for the half beats (half of the BPM). In this case half of the BPM is 64. If the range of tested BPM's was increased to cover 64, the maximum might be found there. Thus a possible future optimisation is to classify signals that fall below a certain threshold, and use the lower end of the BPM spectrum to find the underlying BPM.

6 Benchmarking

6.1 Tempo Analysis Complexity/Speed

The tempo analysis is done in two different tasks. Firstly, it is used as an offline process to classify the tempo of music. Secondly, it is used to analyse the tempo change of the user. The second task occurs more often (approximately every 5 seconds), since the application needs to know whether the user has sped up, or stopped. It is therefore very important to have the tempo algorithm implementation heavily optimised. This section presents the optimisation efforts with benchmarking data for every stage of optimi-

sation, and more elaborate benchmarks for the final implementation.

The first attempt at the algorithm was not optimised, as it had been debugged for more than a month. The following table displays the relative speeds of the different parts of the system in seconds.

Test iterations	**Filterbank**	**Hwindow**	**Diffrect**	**Total timecomb**	**Timecomb avg iteration**	**Total Analysis**
10	0.0252	0.7038	0.0563	6.0368	0.2810	18.4810
10	0.0251	0.7045	0.0566	6.0507	0.2817	18.5061
10	0.0254	0.7049	0.0562	6.0311	0.2808	18.4574
Avg	**0.0253**	**0.7044**	**0.0564**	**6.0395**	**0.2812**	**18.4815**

The first series of optimisations was to remove useless clearing of memory (using the C memset command), and change the places where memory needed to be cleared to hardware accelerated using Accelerate Framework methods. The results are as follows:

Test iterations	**Filterbank**	**Hwindow**	**Diffrect**	**Total timecomb**	**Timecomb avg iteration**	**Total Analysis**
10	0.0250	0.7067	0.0559	5.7068	0.2651	17.5125
10	0.0256	0.7063	0.0560	5.6145	0.2608	17.2476
10	0.0255	0.7178	0.0559	5.5230	0.2566	16.9891
Avg	**0.0254**	**0.7103**	**0.0559**	**5.6148**	**0.2609**	**17.2498**
% Change	0.42%	0.83%	-0.76%	-7.03%	-7.22%	-6.66%

Overall such an optimisation decreased the total analysis time by 6.7%. The next series of optimisations were by further using Accelerate Framework methods for basic arithmetic (such as absolute value of each element in a vector, or summing a vector). The results, in seconds, were thus:

Test iterations	**Filterbank**	**Hwindow**	**Diffrect**	**Total timecomb**	**Timecomb avg iteration**	**Total Analysis**
10	0.0256	0.6635	0.0559	5.5950	0.2598	17.1554
10	0.0256	0.6635	0.0559	5.5950	0.2598	17.1554
10	0.0263	0.6725	0.0559	5.5649	0.2584	17.0733
Avg	**0.0257**	**0.6697**	**0.0559**	**5.5460**	**0.2575**	**17.0190**
%Δ Orig	1.95%	-4.93%	-0.73%	-8.17%	-8.41%	-7.91%
%Δ Prev	1.52%	-5.71%	0.03%	-1.23%	-1.28%	-1.34%

The final optimisation was to complex number multiplication. Equation 10 & 11 present the case for complex number multiplication.

$$\begin{aligned} C_1 &: a + bi \\ C_2 &: c + di \end{aligned} \tag{10}$$

$$C_1 * C_2 = (ac - bd) + i(ad + bc) \tag{11}$$

Such a complex multiplication occurs in the time-comb iteration, or loop within the time-comb method. The complex numbers in this loop are in 4 vectors (two vectors per complex number for the real and imaginary parts), and every elements needs to be multiplied with its counterpart in the other vector. A naive approach, as presented in equation 11, is to loop through every element and extract the a, b, c, and d coefficients and perform the arithmetic. Such an approach is rather costly as a typical complex vector, in our example, will have approximately 2^{17} or 131,072 elements. For each of these 131,072 elements equation 11 must be performed, and such an operation must happen for every BPM value the algorithm wants to test. This comes out to 56 iterations, or 29,360,128 multiplications, and 7,340,032 additions and subtractions[4]. Using vDSP methods such multiplication was rewritten as shown in listing 7.

Listing 5: Complex vector each element multiplication using vDSP routines

```
void complexMultiply(float* A_real, float* A_imag, float *B_real, float *B_imag,
                float *o_real, float* o_imag, float *aux, int length)
{
    // a = a_real
    // b = a_imag
    // c = b_real
    // d = b_imag

    // a * c = o_real
    vDSP_vmul(A_real, 1, B_real, 1, o_real, 1, length);

    // b * d = o_imag
    vDSP_vmul(A_imag, 1, B_imag, 1, o_imag, 1, length);

    // (a*c - b*d) = a_real;
    vDSP_vsub(o_imag, 1, o_real, 1, o_real, 1, length);
    // need to retain o_real.

    // a * d = o_imag
    vDSP_vmul(A_real, 1, B_imag, 1, o_imag, 1, length);
```

[4]For the first Time-comb call, 40 values are checked incrementing by 2, giving 20 iterations. The second call checks 4 values incremented by 0.5, thus 8 iterations. The third call is 1 value incremented by 0.1, thus 10 iterations. The final call is 0.2 values, by 0.01, thus 20 iterations. This gives 56 iterations total. For each pair of complex number, from equation 11, we have 4 multiplications, 1 addition, and 1 subtraction. The typical vector has length of 131072 (approximately 2.2 seconds at 44.1kHz sampling rate rounded to closest whole power of 2), and thus 524,288 multiplications, and 131,072 additions and subtractions. Finally for the 56 iterations, this surmounts to 29,360,128 multiplications, and 7,340,032 additions and subtractions.

```
    // b * c = aux
    vDSP_vmul(A_imag, 1, B_real, 1, aux, 1, length);

    // (a*d + b*c) = o_imag
    vDSP_vadd(o_imag, 1, aux, 1, o_imag, 1, length);
}
```

The results for this optimisation are:

Test iterations	**Filterbank**	**Hwindow**	**Diffrect**	**Total timecomb**	**Timecomb avg iteration**	**Total Analysis**
10	0.0257	0.6068	0.0562	3.0243	0.1374	9.6029
10	0.0255	0.6067	0.0567	3.0021	0.1335	9.7323
10	0.0256	0.6049	0.0563	2.9393	0.1335	9.3778
10	0.0258	0.6025	0.0563	2.8888	0.1311	9.2293
10	0.0245	0.6069	0.0544	2.8500	0.1308	9.2625
10	0.0256	0.6035	0.0561	2.8777	0.1345	9.2223
10	0.0262	0.6041	0.0565	2.9653	0.1341	9.3442
10	0.0260	0.6031	0.0599	2.7611	0.1333	9.5031
10	0.0259	0.6064	0.0583	3.0442	0.1362	9.6132
10	0.0262	0.6021	0.0523	3.1552	0.1356	9.1464
Avg	**0.0257**	**0.6047**	**0.0563**	**2.9508**	**0.1340**	**9.4034**
%Δ Orig	1.74%	-14.15%	-0.18%	-51.14%	-52.33%	-49.12%
%Δ Prev	-0.21%	-9.70%	0.56%	-46.79%	-47.95%	-44.75%

This optimisation made a huge difference, with a 49% change in speed, from 18.5 seconds down to 9.4. This same algorithm is used for accelerometer data, however, the sampling rate is much higher (at 60 Hz), and thus performing the analysis on a 2 second window (120 samples) takes 0.004 seconds.

6.2 Tempo Analysis Accuracy

The implemented tempo analysis was much better than the initial implementation (as described in section 4.5). The following table presents the inferred BPM by the algorithm, the actual BPM value (per outcome from Serato Software), and the mean square error. It can be noted that this algorithm offered a much smaller mean square error than the previous implementation.

Song Title	Artist	Genre	Real BPM	Approx. BPM	Mean Square Error
An American in Paris	STP Ft. Kevin Yost	Chill House	124	122.6	1.96
Maria	Blondie	Rock, Pop	160	159.2	0.64
Amazing	Seal	Upbeat, Pop	121	122.01	1.0201
In My Arms	Kylie Minogue	Pop	128	68.09	3589.2
I Found You	Axwell	House	130	128.86	1.3
Good Vibrations	The Beach Boys	Rock	77	80.72	13.84
Got 2 B U	Solar House	House	124	122.61	1.9321
I Will Survive	Gloria Gaynor	Disco	117	80.12	1360.1344
One More Time	Daft Punk	House	125	122.93	4.28

The following table shows BPM analysis run on additional songs:

Song Title	Artist	Genre	Real BPM	Approx. BPM	Mean Square Error
All My Love	Avalon Superstar	House	128	128.09	0.0081
You'll See Me	Jon Fitz	House	128.5	128.5	0
Say Say Say	Michael Jackson & Paul McCartney	Pop	118	117.1	0.81
Umbrella	Rihanna	Pop	88	128.61	1649.17
Rakfunk	Pryda	House	126	125	1
Conga	Gloria Estefan	Pop	117	81.1	1288.81
Miami	Will Smith	R&B	107.5	107.96	0.2116
Give It To Me	Timbaland	R&B	111.5	73.84	1418.3

The mean square error is significantly lower than analysis done in section 4.5 using the statistical streaming beat detection algorithm.There still are songs which are incorrectly classified (anomalies), however ones which miss the BPM by a small amount are the more serious errors, since these are songs that will be chosen. Songs that are classified at half tempos are usually too low to meet the cutoff to actually run. The future work proposes solutions to make the accuracy even higher.

7 Conclusion

This project attempts to create a solution that delivers a motivational impetus higher than that of simply playing music. This is accomplished successfully by tracking the users tempo and analysing their own music in order to play exactly the right song at the right time (in the right tempo). The BPM analysis implementation uses DSP techniques on both accelerometer and sound signals. It achieves a low degree of error, lower than existing solutions (libraries). Furthermore activity recocgnition allows the phone to know when the user stops or continues exercising.

Throughout the project we have taken a look at the overview of the smartphone market, looked at technical research such as the discrete Fourier transform, investigated beat matching and activity recognition, and finally tested these ideas in the initial assessments section. From there an implementation for BPM analysis was crafted, which offered much better results than existing and available libraries which can be ported to the iPhone. Accelerometer data, driving the user pace analysis, was pre-processed and transformed to remove unwanted parts of the signal (such as noise). Finally the complexity and speed were analysed, and decreased.

The final app has 2 modes: free run and constant. Free run plays music matching the users tempo (user controls tempo), and constant mode plays music at the same tempo (app controls tempo).

Motivation, therefore, is challenged with music. The extent of motivation, however, at this stage, is not measurable. A large scale test would need to be employed which tracked users running without and with the application. The offset between desired BPM and actual BPM would be measured and this could give more insight into the level of motivation. Comparing to related work which attempt to motivate the user, however, this app is at a forefront. It is the one that harnesses the potential of music.

7.1 Future Work

The implementation described in the earlier sections was limited to development time and hardware. This section describes parts of the system that require more exploration in order to achieve higher analysis results, better performance, and a better user experience.

Musical Hills The ability to pre-set a workout before starting it, and having music take you on a journey through this workout, is the biggest and most promising future work. Through the user interface the user could choose different BPM levels at different time lasting for various

amounts of time, and thus create a hill resembling workout schedule. Then when the user would start running, the app would play music at these predefined tempos. The user would follow, and if not, the app would notify the user and mark it in the workout summary. Hills would be the epitome of musical motivation.

Movement Awareness Section 4.4 presented various hardware that can be employed to perform movement awareness. The implementation used only accelerometer data, because it was the easiest to interpret and is widely available (since the iPhone has a built-in accelerometer). Newer iPhones, however, have shipped with a gyroscope, allowing access to information on the rotation about an axis. Harnessing both the accelerometer and gyroscope could offer even greater results to both user BPM analysis and activity recognition.

Learning Learning, the forefront of truly personalised software, is a big part that can be introduced into the project. Such learning is both client-side and back-end side, meaning learning from gathered data prior to shipping. Client-side learning means learning from users' actions based solely on what the user does in the app (to tailor specific actions to the user). The most obvious client-side implementation would be for activity recognition. Each person has a specific way of moving when performing different activities, and learning these specifics would increase accuracy of prediction. Furthermore, back-end learning of activity recognition would be a more complex, yet more accurate method of determining activity recognition. Currently the implementation performs a simple threshold operation on standard deviation, but this threshold is arbitrary. It was chosen through simple observation, rather than a deep study of motion. Learning would infer much more deeply that which simple observation cannot.

Cloud The iPhone is limited in processing power and battery life. Various implementation of intensive computationally algorithms has been done in the cloud. Rather than perform calculations on the phone, the signal can be sent to more powerful servers which can perform even more accurate calculations that would be infeasible on the phone. This approach has even more advantages such as other means of analysis and massive learning potential. Other means of analysis could exploit properties of the song being played. The iPhone has access to the title, artist, album information about each song, and the cloud server could utilise this information to look up pre- analysed songs. The drawback

is that users download their music from unknown sources, some pirated, which may give inaccurate information of the songs (such as wrong titles). This, however, is simply a hypothesis thus the current implementation should gather as much data (and send to logging servers) to be able to determine whether this is a feasible solution.

User Experience The purpose of the application is to aid in motivation during running. This implies that the user experience is crucial to achieving this goal. The first optimisation comes in the form of continuous musical sensation. Currently the app plays a song and switches to the next one. This causes a bit of silence or a sudden jerk to a completely different key and sound. Future work on this will involve an effector unit which will mix together the songs so that there is a blend between the two, thus delivering a continuous musical sensation.

Another possibility to enhance the user experience would be to track, for songs that the person presses next for (ie skips), the time interval between starting to play the song and pressing next. This time would encompass the level of hesitation, ie, how much the user enjoys or wants to hear the song whilst exercising. Songs with a very small interview could be excluded from every playing again (until the person accepts them again). This could enhance what the BPM analysis engine cannot determine - whether the song is right for running (even if it is in the correct tempo).

Future optimisations

The project has future work that is more optimisation than new implementation. The approach taken in the application was to create tools that then could be easily extended to perform a plethora of tasks. The future optimisations are thus:

Feedback The Constant mode in the app plays music at a constant tempo. The user is supposed to run at this speed. If, however, the user strays of the desired tempo, the app currently does nothing. It should notify the user via audio that he is either going too fast or too slow. Such a feedback can be something easy such as a few clicks, or use insights from section 4.1: Motivation.

BPM Analysis The BPM analysis implementation analyses a small window of the original song. In most cases this is adequate, however, certain songs have moments of build-up which do not contain tempo

discerning features, or have additional sounds which may mask the tempo within this window. Therefore, in order to attain a more accurate result, this analysis can be done multiple times on different windows. A simple decision engine can then take the BPM's and determine what the underlying BPM is, or place the song back into analysis (on a different window). Through simple experimentation with the BPM implementation, it can be noted that performing analysis multiple times on different windows does indeed give much better results. Such an experimentation was done on 3 windows (from the beginning, middle, and end of the song), and in most cases at least one of the returned values contained the near exact BPM (with a mean square error < 2). More investigation would need to be done for the algorithm to tell which BPM value is correct.

Parameter optimisation The parameters to the BPM analysis stage and accelerometer pre-processing stage, are largely chosen either based upon related research or simple testing/observation. Taking research values is adequate for the purpose of this project, but may not be adequate for this specific implementation. In order to truly attain the best results, every parameter needs to be deeply analysed in order to determine under which value it best fits the whole system. The basic optimisations that can be run is mean error minimisation, but more elaborate optimisation techniques could be employed to test the combination of parameters that offer the best results.

Information For the application to truly make sense, it needs to be augmented by more information. These information could be GPS tracking, burning of calories, and a history of runs. GPS tracking would track the distance travelled and display a heat map overlaid on the actual map that showed the tempo in different parts of the run. Furthermore calorie burning could be calculated from available information. Finally a history of all previous runs could be displayed, perhaps exportable to the computer for a more detailed analysis.

Bibliography

[1] Statistica, "Smartphone os: global market share q1 2012," 2012. Available from: http://www.statista.com/statistics/73662/quarterly-worldwide-smartphone-market-share-by-operating-system-since-2009.

[2] "Android device gallery." Available from: http://www.android.com/devices.

[3] A. Terras, *Fourier Analysis on Finite Groups and Applications*. No. ISBN 978-0-521-45718-7, Cambridge University Press, 1999. p. 30.

[4] S. W. Smith, "The scientist and engineer's guide to digital signal processing."

[5] F. Patin, "Beat detection algorithms," 2003.

[6] K. Cheng, B. Nazer, J. Uppuluri, and R. Verret, *Beat This - A Beat Synchronisation Project*. PhD thesis, Rice University, Houston, Texas, 2001.

[7] A. D. Patel, J. R. Iversen, M. R. Bregman, I. Schulz, and C. Schulz, *Investigating the human-specificity of synchronization to music*. PhD thesis, University of California San Diego, 2010. The Neurosciences Institute.

[8] N. Ravi, N. Dandekar, P. Mysore, and M. L. Littman, *Activity Recognition from Accelerometer Data*. PhD thesis, Rutgers University, Piscataway, NJ 08854, 2005. Department of Computer Science.

[9] M. kyung Suh, K. Lee, A. Nahapetian, and M. Sarrafzaded, *Interval training guidance system with music and wireless group exercise motivations*. PhD thesis, University of California Los Angeles, 2009. Department of Computer Science.

[10] Nike-Inc., "Nike+ gps." Apple App Store description.

[11] V. X. Afonso, W. J. Tompkins, T. Q. Nguyen, and S. Luo, *Filter Bank-based ECG Beat Detection*. PhD thesis, University of Wisconsin-Madison, 1996. Department of Electrical and Computer Engineering.

[12] C. Randell and H. Muller, *Context Awareness by Analysing Accelerometer Data*. PhD thesis, University of Bristol, 2000. Department of Computer Science.

[13] A. Purwar, D. U. Jeong, and W. Y. Chung, *Activity Monitoring from Real-Time Triaxial Accelerometer data using Sensor Network*. PhD thesis, Graduate School of Design and IT, Dongseo University, Busan, Korea, 2007. Division of Computer Information Engineering.

[14] "From ipod library to pcm samples in far fewer steps than were previously necessary," 2010. subfurther.com.

[15] M. Tyson, "Using remoteio audio unit," 2008. atastypixel.com.

[16] T. Zicarelli, "ios audio processing graph," 2011. zerokidz.com.

[17] E. W. Weisstein, "Fast fourier transform." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/FastFourierTransform.html.

[18] A. Inc., "ios documentation - vdsp programming guide," 2011. developer.apple.com.

[19] O. Parviainen, "Soundtouch audio processing library," 2001-2011. Surina.net/soundtouch/.

[20] E. W. Weisstein, "Hanning function." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/HanningFunction.html.

Glossary

Accelerate Framework An iOS framework which contains C APIs for vector and matrix math, digital signal processing, large number handling, and image processing, accelerated in hardware. 43, 52

API Application Programming Interface. 6, 16, 18, 52

BPM Beats Per Minute. 2, 4, 9–11, 13, 15, 16, 21, 22, 40, 42, 44, 46, 48, 52

convolution A convolution is an integral that expresses the amount of overlap of one function as it is shifted over another function. It therefore "blends" one function with another. 10, 52

DSP Digital Signal Processing. 2, 7, 13, 47, 52

FFT Fast Fourier Transform. 1, 2, 4, 10, 17–19, 52, 55

GPS Global Positioning System. 2, 13, 19, 52

Gyroscope a device consisting of a wheel or disk mounted so that it can spin rapidly about an axis that is itself free to alter in direction. The orientation of the axis is not affected by tilting of the mounting; so gyroscopes can be used to provide stability or maintain a reference direction in navigation systems, automatic pilots, and stabilizers. 19, 52

Hz The SI unit of frequency, equal to one cycle per second. 10, 11, 16, 21, 22, 33, 44, 52

iOS Apple Inc.'s mobile operating system. Runs iPod Touch, iPad, iPhone, Apple TV. 1, 6, 16, 52, 55

iTunes An Apple application that comes with Mac OS X and lets you import music, podcasts, and video from CDs or from the iTunes Store; organise media into custom albums and playlists; and burn it onto disks or transfer it to an iPod or other media player. iTunes is also widely used on Windows-based computers. 52

latency A measure of time delay experienced in a system. 16, 52

memset Sets the first num bytes of the block of memory pointed by ptr to the specified value (interpreted as an unsigned char). 43, 52

noise Irregular fluctuations that accompany a transmitted electrical signal but are not part of it and tend to obscure it. 10, 28, 52

RMS Root Mean Square. 14, 52

1 Appendix 1: FFT Applier Implementation

This section holds the implementation of the single precision FFT algorithm using the iOS Accelerate framework.

Listing 6: FFTApplier.h

```
//////////////////////////////////////////////////////////////////////////////
//
//  FFTApplier.h
//  PaceMaker
//
//  Created by Piotr Holc on 16/01/2012.
//  Copyright (c) 2012 Piotr Holc. All rights reserved.
//
//////////////////////////////////////////////////////////////////////////////
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.
//
//////////////////////////////////////////////////////////////////////////////

#import <Foundation/Foundation.h>

#include <stdio.h>
#include <stdlib.h>
#include <Accelerate/Accelerate.h>


@interface FFTApplier : NSObject {

    // fft vals
    FFTSetup                fftSetup;   //fft predefined structure
                                        //holding weight vector for vDSP.

    // fft predefined structure for double precision ffts
    FFTSetupD               fftSetupD;


    UInt32                  maxFrames; // the max amount of frames given.
    int                     log2n; // base 2 log of fft size (1024)
    int                     N; // fft size
    int                     NOver2; //half fft size
    size_t                  bufferCapacity; // buffer size in samples
    size_t                  index; // read index pointer for fft buffer.

    void                    *dataBuffer; // input buffer
    float                   *outputBuffer; // fft conversion buffer
}

-(id)initWithMaxFrames:(UInt32)mFrames;


-(int)getN;

-(void)performFFT:(SInt16*)sampleBuffer
withNumberOfFrames:(UInt32)numberFrames
         andOutput:(COMPLEX_SPLIT*)dft;

-(void)performFloatingFFT:(float*)sampleBuffer
        withNumberOfFrames:(UInt32)numberFrames
                 andOutput:(COMPLEX_SPLIT*)dft;

-(void)performDoubleFFT:(double*)sampleBuffer
```

```
        withNumberOfFrames:(UInt32)numberFrames
                andOutput:(DOUBLE_COMPLEX_SPLIT *)dft;


-(void)performIFFT:(float*)realBuffer
     andImagBuffer:(float*)imagBuffer
   andBufferLength:(UInt32)numberFrames
         andOutput:(COMPLEX_SPLIT*)dft;

-(void)performDoubleIFFT:(double*)realBuffer
     andImagBuffer:(double*)imagBuffer
   andBufferLength:(UInt32)numberFrames
                andOutput:(DOUBLE_COMPLEX_SPLIT *)dft;


@end
```

Listing 7: FFTApplier.m

```
//////////////////////////////////////////////////////////////////////////////
//
//  FFTApplier.m
//  PaceMaker
//
//  Created by Piotr Holc on 16/01/2012.
//  Copyright (c) 2012 Piotr Holc. All rights reserved.
//
//////////////////////////////////////////////////////////////////////////////
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.
//
//////////////////////////////////////////////////////////////////////////////

#import "FFTApplier.h"

@interface FFTApplier (hidden)

-(void) setupFFT;

@end


@implementation FFTApplier

-(id)initWithMaxFrames:(UInt32)mFrames {

    if ( (self = [super init]) ) {
        maxFrames = mFrames;
        [self setupFFT];
    }
    return self;
}

// setup the FFT structures for vDSP.
-(void) setupFFT {

    printf("SetupFFT: Setting up FFT structures\n");

    // setup input and output buffers to the max frame size
    dataBuffer = (void*)malloc(maxFrames * sizeof(SInt16));
    outputBuffer = (float*)malloc(maxFrames * sizeof(float));

    memset(dataBuffer, 0, maxFrames *sizeof(SInt16));
    memset(outputBuffer, 0, maxFrames *sizeof(float));
```

```
    // set the init stuff for fft based on number of frames

    // log base2 of max number of frames, ie: 10 for 1024.
    log2n = ceilf(log2f(maxFrames));
    // actual max number of frames, since n = 10, 1024.
    N = 1 << log2n;
    NOver2 = N / 2;


    // zero return indicates an error setting up internal buffers
    fftSetup = vDSP_create_fftsetup(log2n, FFT_RADIX2);
    fftSetupD = vDSP_create_fftsetupD(log2n, FFT_RADIX2);

    if (fftSetup == (FFTSetup)0) {
        printf("Error: FFTSetup -- unable to allocate fft setup buffers");
    }
    if (fftSetupD == (FFTSetupD)0) {
        printf("Error FFTSetupD -- unable to allocate fft setup buffers");
    }
}

// Performs a forward FFT.
-(void)performFFT:(SInt16*)sampleBuffer withNumberOfFrames:(UInt32)numberFrames
        andOutput:(COMPLEX_SPLIT*)dft
{
    assert(numberFrames <= N);
    UInt32 stride = 1;

    // we need to convert SInt16 to floating point
    vDSP_vflt16((SInt16*)sampleBuffer,
                stride,
                (float*)outputBuffer,
                stride,
                numberFrames);

    [self performFloatingFFT:outputBuffer
          withNumberOfFrames:numberFrames
                   andOutput:dft];
}

-(void)performFloatingFFT:(float*)sampleBuffer
       withNumberOfFrames:(UInt32)numberFrames
                andOutput:(COMPLEX_SPLIT*)dft
{
    NSAssert2(numberFrames <= N,
              @"Number frames (%d) not less than N (%d)",
              numberFrames, N);
    UInt32 stride = 1;
    int toZeroFill = N - numberFrames;

    memcpy(dft->realp, sampleBuffer, numberFrames*sizeof(float));
    memset(dft->realp+numberFrames, 0, toZeroFill);
    memset(dft->imagp, 0, N * sizeof(float));

    // Perform the forward FFT
    vDSP_fft_zip(fftSetup, dft, stride, log2n, kFFTDirection_Forward);
}

-(void)performDoubleFFT:(double*)sampleBuffer
     withNumberOfFrames:(UInt32)numberFrames
              andOutput:(DOUBLE_COMPLEX_SPLIT *)dft
{

    assert(numberFrames <= N);
    UInt32 stride = 1;
    int toZeroFill = N - numberFrames;

    memcpy(dft->realp, sampleBuffer, numberFrames*sizeof(double));
    memset(dft->realp+numberFrames, 0, toZeroFill);
    memset(dft->imagp, 0, N * sizeof(double));

    // Perform the forward FFT
    vDSP_fft_zipD(fftSetupD, dft, stride, log2n, kFFTDirection_Forward);
}
```

```
// Performs an inverse FFt, using seperated buffers for complex number, ie:
// a number a + bi, ie 5th in the buffer would be:
// realBuffer[4] + imagBuffer[4]i.
-(void)performIFFT:(float*)realBuffer                                          137
     andImagBuffer:(float*)imagBuffer
   andBufferLength:(UInt32)numberFrames
         andOutput:(COMPLEX_SPLIT*)dft
{                                                                              142
    numberFrames = N;
    assert(numberFrames <= N);

    UInt32 stride = 1;
    int toZeroFill = N - numberFrames;                                         147

    // copy the input to the buffer.
    memcpy(dft->realp, realBuffer, numberFrames*sizeof(float));
    memcpy(dft->imagp, imagBuffer, numberFrames*sizeof(float));
                                                                               152
    // clear the additional bits
    memset(dft->realp+numberFrames, 0, toZeroFill);
    memset(dft->imagp+numberFrames, 0, toZeroFill);

    //perform in place fourier transform                                       157
    vDSP_fft_zip(fftSetup, dft, stride, log2n, kFFTDirection_Inverse);

    //we need to scale the output as per documentation instructions
    //by a factor of n. (thus every item will be divided by n).
    float scale = (float)1.0 / N;                                              162

    // scale
    vDSP_vsmul(dft->realp, 1, &scale, dft->realp, 1, numberFrames);
    vDSP_vsmul(dft->imagp, 1, &scale, dft->imagp, 1, numberFrames);
}                                                                              167

-(void)performDoubleIFFT:(double*)realBuffer
           andImagBuffer:(double*)imagBuffer
         andBufferLength:(UInt32)numberFrames
               andOutput:(DOUBLE_COMPLEX_SPLIT *)dft                           172
{

    numberFrames = N;
    assert(numberFrames <= N);
                                                                               177
    UInt32 stride = 1;
    int toZeroFill = N - numberFrames;

    // copy the input to the buffer.
    memcpy(dft->realp, realBuffer, numberFrames*sizeof(double));               182
    memcpy(dft->imagp, imagBuffer, numberFrames*sizeof(double));

    // clear the additional bits
    memset(dft->realp+numberFrames, 0, toZeroFill);
    memset(dft->imagp+numberFrames, 0, toZeroFill);                            187

    //perform in place fourier transform
    vDSP_fft_zipD(fftSetupD, dft, stride, log2n, kFFTDirection_Inverse);

    //we need to scale the output as per documentation instructions            192
    //by a factor of n. (thus every item will be divided by n).
    double scale = (double)1.0 / N;

    // scale
    vDSP_vsmulD(dft->realp, 1, &scale, dft->realp, 1, numberFrames);           197
    vDSP_vsmulD(dft->imagp, 1, &scale, dft->imagp, 1, numberFrames);
}

-(int)getN {
    return N;                                                                  202
}

-(void)dealloc {

    vDSP_destroy_fftsetupD(fftSetupD);                                         207
    fftSetupD = NULL;
```

```
    vDSP_destroy_fftsetup(fftSetup);
    fftSetup = NULL;

    free(dataBuffer);                                                    212
    free(outputBuffer);

    [super dealloc];
}                                                                        217
@end
```