BPM Detector (Android)

Final Report

Duc Dao

CSC 492, Fall 2017

Computer Science and Software Engineering Department
College of Engineering
California Polytechnic State University, San Luis Obispo

Abstract

The BPM Detector is an Android application that detects the tempo or beats-per-minute (BPM) for a band conductor's conducting. Generally, the conductor moves their arms in a predictable pattern in order to maintain the band's synchronization, acting similarly to a metronome. The purpose of this application is to provide a tool for concert and marching band conductors to detect their precise tempo while conducting. This will help them determine how well they are coordinating the band and whether they need to adjust their tempo.

Contents

1 Introduction

1.1 Purpose

This document is the culmination of all my work for CSC 491 and 492 for **BPM Detector**, an Android application that detects the beats-per-minute of a conductor's tempo while conducting. The main purpose of the BPM Detector is to aid conductors in seeing how well they are conducting the band and determine whether they need to make adjustments or not. This will benefit marching bands because these types of bands are generally more sensitive to tempo inconsistencies compared to other types of bands.

1.2 Document Conventions

1. Main Section

1.1 Subsection

1.1.1 Minor Subsection

Body text

References

1.3 Intended Audience

The intended audience for this document includes:

- Developers
- Advisors
- Musically Inclined Users

1.3.1 Developers

Along as a the Final Report for my senior project, this document will serve as a repository of my thoughts and work on BPM Detector, which maybe useful to other developers, including my future self.

1.3.2 Advisors

Advisors include:

- My senior project advisor, Professor Franz Kurfess
- All other resources that aid in my project which may include other professors, students, and other developers

1.3.3 Musically Inclined Users

Users that have knowledge and capability of conducting a song may find this project useful for them. This includes, but is not limited to:

- Band directors
- Drum majors, student conductors
- Professors in the Music department

2 Overall Description

2.1 Background

In the context of music ensembles, conducting is the act of directing the group in order to synchronize various components of the band and the performance. Members rely on the conductor to initiate the piece/song that they are performing, set the beats-per-minute or tempo of the song, cue points of interest in the piece, and other musical duties.

With regards to the tempo, there are no applications to automatically detect the conductor's BPM while they are conducting. Existing solutions on the Google Play Store have users tap on the screen to manually calculate the BPM. Metronome applications only sound off the BPM by setting a certain tempo, or listening to the environment.

2.2 Problem

In marching and concert bands, there needs to be some sort of method to synchronize all the members of the band so that they play, rest, and breathe at the appropriate time. This synchronization is typically done by a person such as a conductor, director, or drum major. The conductor synchronizes the band by moving their arm(s) in a predictable pattern in order to maintain the music's tempo. Periodically, bands may mistakenly change tempos not at the fault of the conductor. This becomes hard to identify depending on the song.

Currently, there are no applications to automatically detect the conductor's BPM while they are conducting. Existing solutions on the Google Play Store have users tap on the screen to manually calculate the BPM and metronome applications only sound off the BPM by setting a certain tempo, or listening to the environment.

2.3 Solution

My solution is an Android Wear application that will automatically detect the BPM of the conductor's conducting pattern. The application will utilize the smartwatch's accelerator to calculate the BPM in real-time.

2.4 Product Functionality

BPM Detector's will perform one major function: to automatically detect the conductor's tempo while they are conducting.

2.5 User Class and Their Characteristics

BPM Detector will be used by one general user class: conductors. Conductors' experiences varies from person to person but all of them generally have knowledge of:

- Conducting patterns
- Feeling of tempos
- Technicalities of the song they are conducting

2.6 Operating Environment

BPM Detector will be an Android application running on a smartwatch with at least Android Wear 2.0. The smartwatch requires a Bluetooth connection to an Android phone running at least 6.0 Marshmallow.

3 External Interface Requirements

3.1 User Interfaces

Users of BPM Detector will interact with the UI of the Android Wear smartwatch. Users will be able to:

- Start the BPM detection
- Stop the BPM detection
- View past detections

BPM Detector will utilize elements from Material Design, a design language developed by Google.

3.2 Hardware Interfaces

BPM Detector will be installed on the smartwatch. With the edition of Android Wear to 2.0 (from 1.5), Google has encouraged the use of standalone applications on Android Wear smartwatches.

3.3 Software Interfaces

The smartwatch will have the capacity to initiate the BPM detection. During detection, the smartwatch must be able to interpret data from the sensors of the smartwatch and output a number representing the BPM.

4 Functional Requirements

BPM Detector functional requirements include the following use cases:

1. Starting BPM detection
2. Stopping BPM detection
3. Viewing past detections

4.1 Starting BPM Detection

Use Case ID	1
Name	Starting BPM Detection
Description	The act of starting the tempo detection.
Preconditions	User must have the smartwatch on their wrist and the application launched.
Postconditions	None
Priority	High
Frequency of Use	High
Normal Course	User commences tempo detection.
Actor Action 1. User selects ”DETECT” on either the smartwatch or smartphone.	**System Responses** 2. Smartwatch receives initiation and tells sensors to retrieve information in real-time. 3. Smartwatch interprets sensor information and calculates a BPM number. 4. Smartwatch displays BPM.

4.2 Stopping BPM Detection

Use Case ID	2
Name	Stopping BPM Detection
Description	The act of stopping the tempo detection.
Preconditions	User started the application and initiated detection. Smartwatch is currently detecting BPM.
Postconditions	Smartwatch has stopped detecting BPM.
Priority	High
Frequency of Use	High
Normal Course	User ceases tempo detection.
Actor Action 1. User selects "STOP" on the smartwatch.	**System Responses** 2. Smartwatch receives cease selection and tells sensors to stop retrieving information in real-time. 3. Smartwatch asks user if they want to save their detection data for future viewing.

4.3 Viewing Past Detections

Use Case ID	3
Name	Viewing Past Detections
Description	Viewing detections user made in the past.
Preconditions	User has done at least one detection and is saved correctly.
Postconditions	None
Priority	Low
Frequency of Use	Medium
Normal Course	User views past detections.
Actor Action 1. User selects appropriate page containing detection information. 3. User selects which past detection they want to view in detail.	**System Responses** 2. Smartwatch receives selection and takes user to a listing of past detections. 4. Smartwatch displays past detection the user has selected and displays information associated with that detection.

5 Non-Functional Requirements

5.1 Performance Requirements

All actions performed in the application must take a reasonable amount of time; the faster, the better. The goal is to keep all requests made (starting, stopping, viewing) under 2.5 seconds in terms of response time. To achieve this, the developer must implement the application with consideration to the smartwatch's limited resources.

5.2 Safety Requirements

During detection, the smartwatch's temperature cannot exceed 110 F. To prevent this, the developer must be able to utilize the smartwatch's limited resources efficiently.

6 Development Tools

6.1 Software

6.1.1 Android Studio

BPM Detector was developed on Android Studio 3.0.1, an integrated development environment (IDE) made specifically for Android, built-in on JetBrains' IntelliJ IDEA.

I used Android API 27 (codenamed Android 8.1 Oreo) as the target software development kit (SDK) version and Android API 23 (codenamed Android 6.0 Marshmallow) as the minimum SDK version. Java 8 was the implementation language utilized.

6.2 Hardware

6.2.1 Windows 10 PC

Android Studio ran on a Windows 10 PC with the following specifications:

- Central Processing Unit (CPU): Intel Core i5 6400 @ 2.7GHz
- Graphics Processing Unit (GPU): Asus GeForce GTX 780 3GB DirectCU II
- Motherboard: MSI H110I PRO AC Mini ITX LGA 1151
- Memory (RAM): Kingston HyperX Fury 8GB DDR4-2133
- Storage: SanDisk Ultra II 480GB, Western Digital Back 2TB
- Power Supply: Corsair RMx 650W 80+ Gold ATX

6.2.2 Moto 360

A second generation, 42mm Moto 360 was used to load and test BPM Detector. The Moto 360 has the following specifications:

- Chipset: 1.2 GHz quad-core Qualcomm Snapdragon 400 CPU
- Memory (RAM): 2 GB
- Storage: 4GB
- Connectivity: Bluetooth 4.0, LE, aptX

7 Implementation Theory

7.1 Beat Detection

In music, most songs can be broken down into beats of 4 (which I'll refer to as **4/4**). In 4/4, there are four points a conductor must "hit" in physical space in front of them.

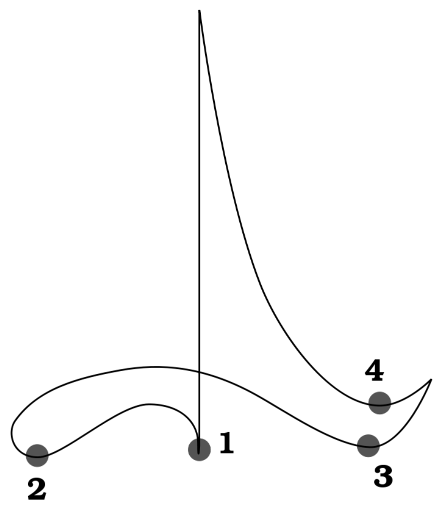


Figure 1: The 4 beats the conductor must "hit." 4/4 is referred to as a **time signature**.
Image Source: https://upload.wikimedia.org/wikipedia/commons/thumb/9/94/Conducting-44time.svg/1024px-Conducting-44time.svg.png

When a person is conducting, their arms will accelerate up and down rhythmically. Going from beat 1 to 2, most conductors will hit 1, and sharply accelerate their movement in a very minuscule time interval to hit beat 2. During this time interval, the acceleration will increase very rapidly until beat 2 is hit. When beat 2 is hit, the acceleration will instantaneously drop close to 0. This cycle of increasing acceleration in a small time frame followed by a nearly instantaneous acceleration down to 0 happens between every beat. In the context of 4/4, it happens between beats:

- 1 to 2
- 2 to 3
- 3 to 4
- 4 to 1
- 1 to 2...(and so on)

Using this observation, you can generate a graph that has periodic peaks. These peaks occur between the end of the sharply increasing acceleration and the beginning of the near instantaneous acceleration decrease to near 0. These peaks can be identified as beats because the peaks occur when the conductor "hits" a beat in their conducting pattern, the end of the sharp acceleration increase.

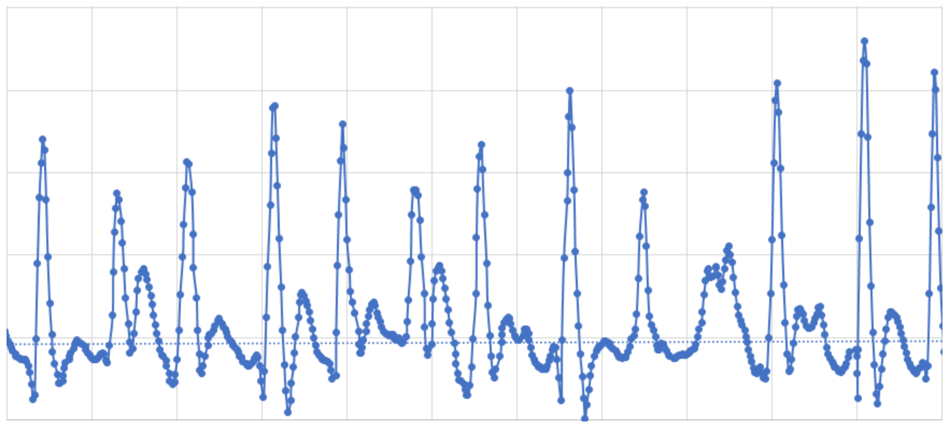


Figure 2: A graph generated from conducting UCLA Bruin Marching Band's rendition of "Word Up" while holding my phone. **X-axis** represents **time**, **y-axis** represents **total linear acceleration**.
Word Up Source: https://www.youtube.com/watch?v=BkRbJ20C72I

7.2 Acceleration

During our acceleration calculation, gravity is an undesirable variable that could skew our results. To resolve this, we need to use **linear acceleration**, which accounts for gravity exclusion. Because we have to account for acceleration along the x, y, and z-axis, calculating **total linear acceleration** will be needed. Total linear acceleration is necessary to account for movement in three dimensional space.

$$\sqrt{x^2 + y^2 + z^2}$$

Figure 3: Formula to calculate total linear acceleration where x, y, and z are the accelerations for each respective axis.

7.3 Beats-Per-Minute Calculation

Knowing when the beats (peaks) occur in the graph shown in Figure 2 allows us to calculate the BPM. By knowing when the beats occurs, you can determine the **time interval between each beat**. Once you know this time interval, you can figure out how many of these time intervals can fit inside 60 seconds by dividing 60 by some time interval. This means a song that has a BPM of 120 will have a time interval between each beat of .5 seconds, 2 beats a second.

To stabilize the BPM calculation and make it more accurate, we can take the average of a set of time intervals. This is under the assumption that the BPM of the song remains

$$\text{BPM} = \frac{60 \text{ seconds}}{\text{time interval between each beat}}$$

Figure 4: Formula to calculate the BPM.

constant. This would not work if the song changes tempo because past calculations would affect the current one. If that's the case, we must resort to the formula in Figure 4.

$$\text{BPM} = \frac{60 \text{ seconds}}{\overline{(\text{time interval between each beat})}}$$

Figure 5: Same formula from Figure 4 but using an average time interval.

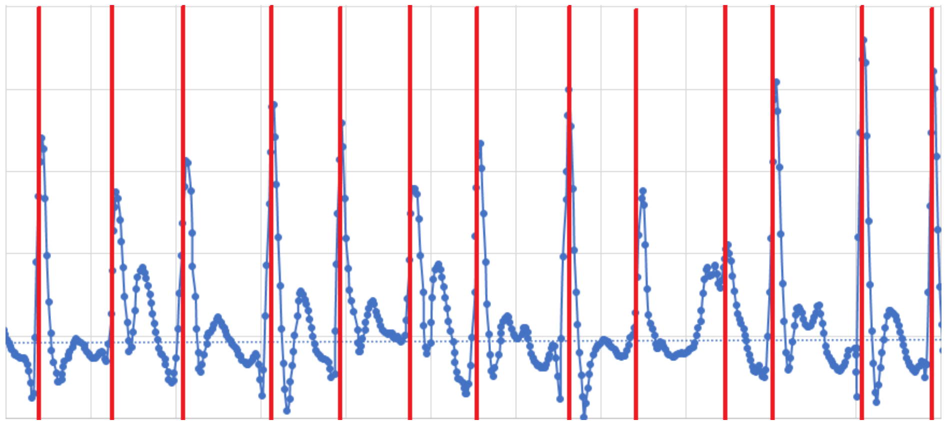


Figure 6: The same graph from Figure 2 but with the peaks marked in red. Marking the peaks also makes the time interval between each beat more apparent. Note that there are false positive peaks caused by the conducting style of the user.

8 Software Implementation

Android consist of many **activities**, a component that provides a screen for the user to interact with. Every activity has a method called `onCreate` which serves as the starting point. When switching activities, an `Intent` is used to describe some operation to perform when making the transition.

My implementation can be found here, or at https://github.com/ducdao/BPM-Detector.

8.1 Activities

8.1.1 MainActivity

`MainActivity` is the initial starting point for BPM Detector. It contains most of the application code including:

- UI elements (labels, button colors)
- Sensor logic
- Beat detection logic
- Beats-per-minute calculation

8.1.2 QuerySaveActivity

`QuerySaveActivity` gets called by `MainActivity` after the user stops the detection. This initiates a new screen asking the user if they want to save or delete the most recent detection.

8.1.3 ConfirmationActivity

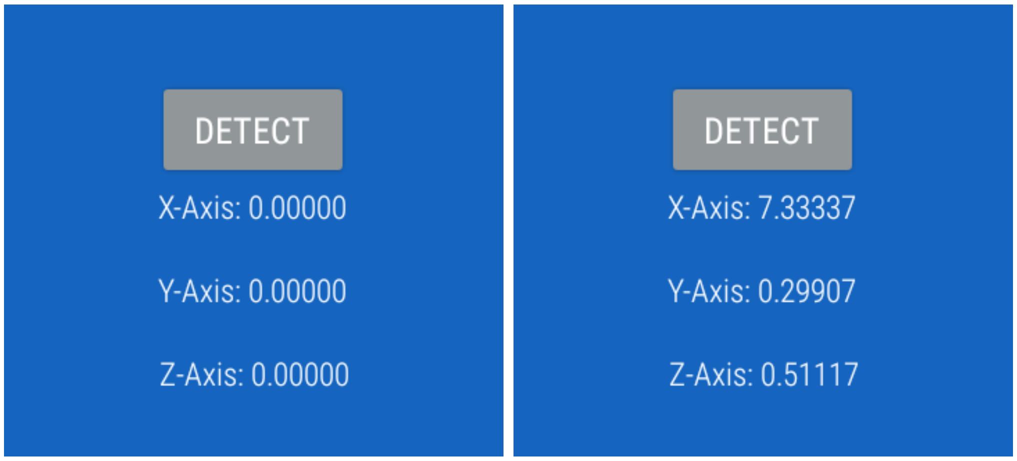
`ConfirmationActivity` is a class that's defined in the Android Wearable library, meant to standardize confirmation responses.

9 Testing

The kinesthetic nature of BPM Detector requires testing the application physically through conducting. Development turnaround (the time between compiling the application and transferring it to my Moto 360) was a massive overhead during testing, taking as long as 2 minutes from clicking the Run button to launching the application. I would spend a minute or two waiting for it to compile, only to having the application crash. This made functional testing impractical.

10 Development Screenshots

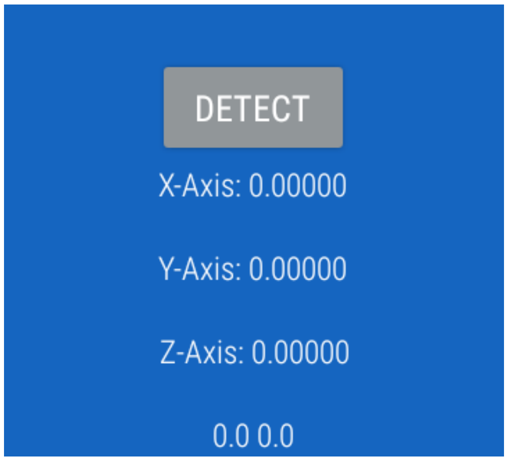
10.1 Accelerometer - Linear Aceeleration in X, Y, and Z Axes



X-Axis: 0.00000
Y-Axis: 0.00000
Z-Axis: 0.00000

X-Axis: 7.33337
Y-Axis: 0.29907
Z-Axis: 0.51117

10.2 Start Detection Time + Duration of Detection

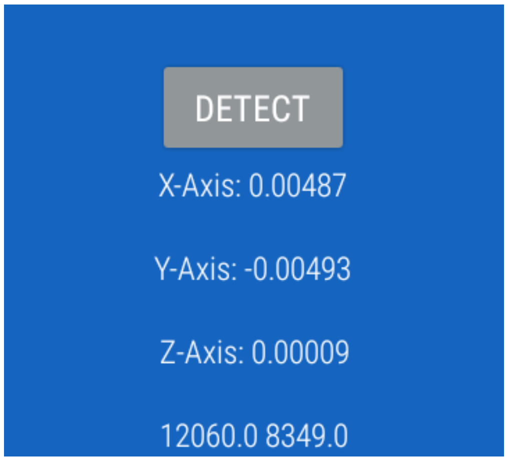


X-Axis: 0.00000

Y-Axis: 0.00000

Z-Axis: 0.00000

0.0 0.0



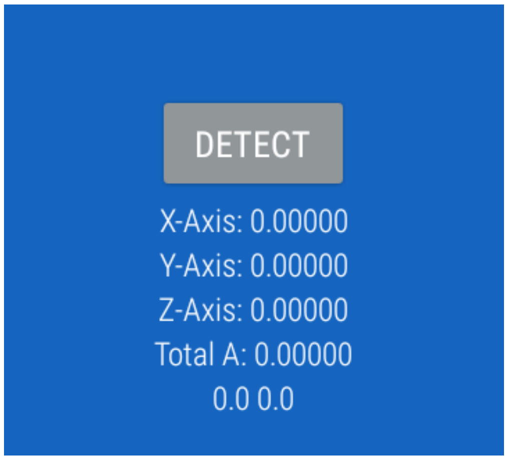
X-Axis: 0.00487

Y-Axis: -0.00493

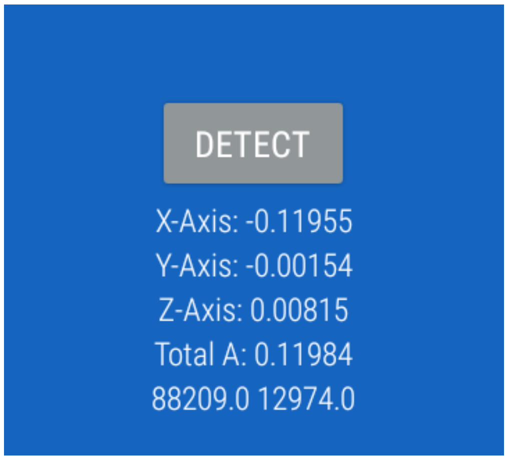
Z-Axis: 0.00009

12060.0 8349.0

10.3 Calculated Total Acceleration

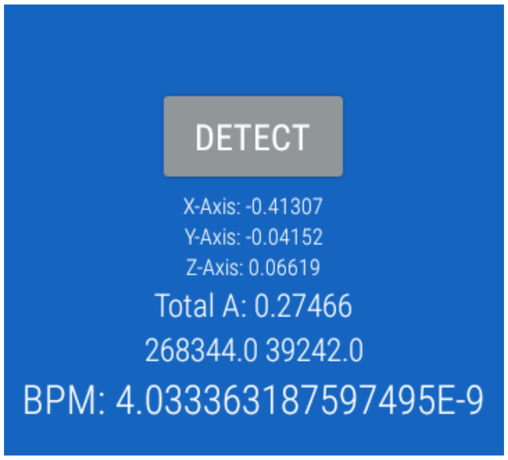


X-Axis: 0.00000
Y-Axis: 0.00000
Z-Axis: 0.00000
Total A: 0.00000
0.0 0.0



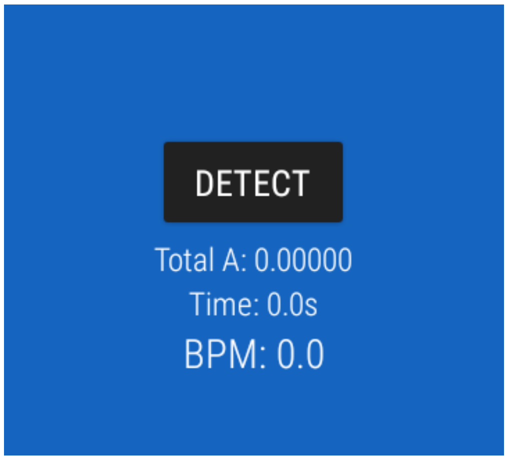
X-Axis: -0.11955
Y-Axis: -0.00154
Z-Axis: 0.00815
Total A: 0.11984
88209.0 12974.0

10.4 Started BPM Calculation

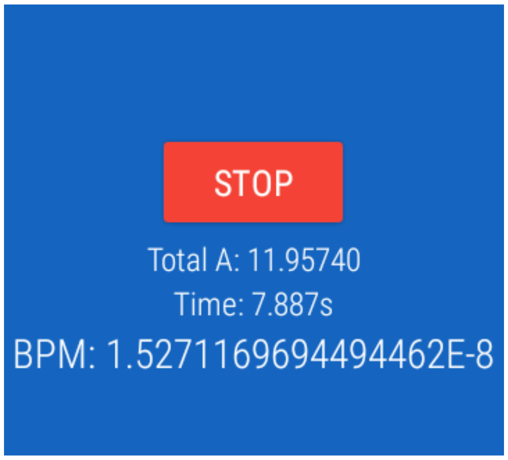


X-Axis: -0.41307
Y-Axis: -0.04152
Z-Axis: 0.06619
Total A: 0.27466
268344.0 39242.0
BPM: 4.033363187597495E-9

10.5 Removed X, Y, Z Values from UI

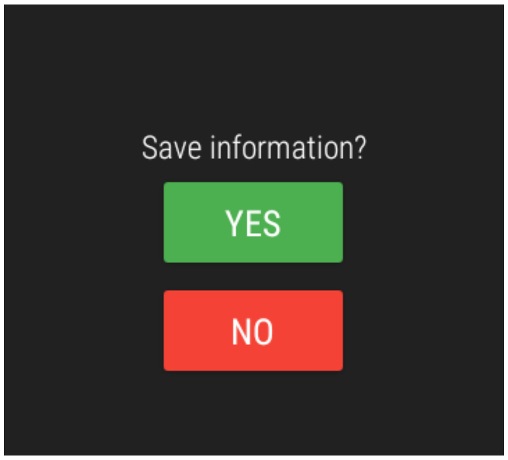


Total A: 0.00000
Time: 0.0s
BPM: 0.0

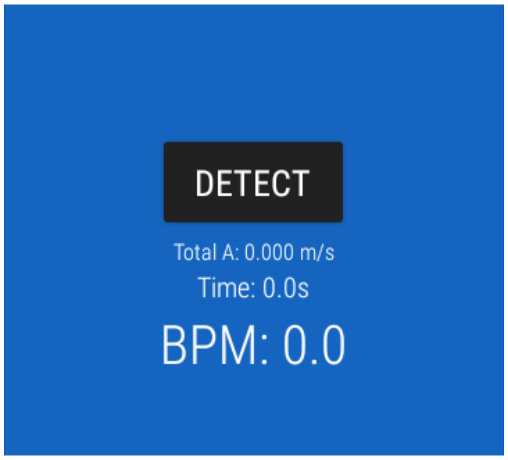


Total A: 11.95740
Time: 7.887s
BPM: 1.5271169694494462E-8

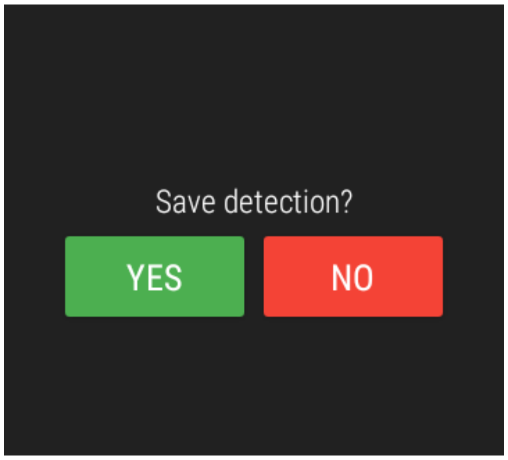
10.6 Confirmation Page for Saving Latest Detection Information



10.7 UI Changes



Total A: 0.000 m/s
Time: 0.0s
BPM: 0.0



11 Problems

11.1 Testing

As said in the Testing section, development turnaround was long. This made development rather tedious, so I had to make sure what I was programming was as bug free as possible to prevent another repetition of waiting only for the application to crash.

11.2 Moto 360 Display

During CSC 491, my Moto 360's screen started to act up. Only the top portion of the screen was actually displaying anything. This was problematic because the Moto 360 screen is already really small. Having half the screen broken would hinder UI testing. At first, I wanted to replace the screen but finding spare parts, let along the screen, was difficult. I ended up just purchasing another second generation Moto 360 for as cheap as possible.

11.3 Real-time Detection

Doing the calculation in real-time proved to be the most challenging part of the project. Saving every total linear acceleration in a TreeMap would cause the application to actually use all of my Moto 360's RAM and throw a OutOfMemoryError exception. Unfortunately, I couldn't get to debug this in time.

12 Prospective Improvements

12.1 Frontend

- An Android phone counterpart so that the user can see the BPM in real-time
- Fully implement `RecyclerView` to render saved detections
- Graph view of detections
- Conducting aid in the form of periodic vibrations

12.2 Backend

- Optimize beat detection to increase application accuracy and robustness
- Account for tempo changes, most likely changing Figure 5 back to Figure 4
- Load detection data to some data store like Firebase

13 Conclusion

In conclusion, developing BPM Detector has been a fantastic and practical experience for me. I always wanted to do Android development ever since my first year at Cal Poly but always felt overwhelmed by the Android framework. Fast forward a few years and I've gained enough experience through classes and other projects to really dive into Android development head first. The Android platform has really matured over the years, making documentation reading and problem solving much easier than previous years. For the time being, I plan to continue development on BPM Detector to build an application that is not only practical to me but for other entities such as bands as well. I hope to bring this experience developing BPM Detector into my career as a prospective Android developer.