E4: Run With It

Aarushi Wadhwa: Electrical and Computer Engineering, Carnegie Mellon University

Akash Bansal: Electrical and Computer Engineering, Carnegie Mellon University

Mayur Paralkar: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—A system capable of adapting the BPM of a song to the pace of a long distance jogger. We will use the step detector hardware sensor on a smartphone to detect footsteps of the runner, which we will set as a target BPM for a song which will be scaled in the time domain by a phase vocoder. The user will hit a button on the smartphone app and begin running, and the music will start playing while being refreshed every one and a half minutes. A score-based algorithm will select songs from the playlist to be played next - the score is based on how close the song is to the runner's pace, and how recently the song was last played.**

***Index Terms*—Android, Beats Per Minute (BPM), Dual-Tree Complex Wavelet Transform (DTCWT) Phase Vocoder, Footstep Detection, Pace, Short-Time Fourier Transform (STFT) Phase Vocoder, Time-Scale Audio Modification**

A. Introduction

Running with music can be very difficult, because if the pace of the music does not match the runner's natural pace, it will throw them off and possibly ruin their run. We have talked to runners who hum or sing their own music instead of listening to music so that they can naturally speed up or slow down the song to fit their running pace. Our goal is to alleviate this burden for long distance joggers. By solving this issue, runners will be able to get more enjoyment out of their run, leaving them more fulfilled and feeling better.

We have seen this type of application created using just a Short-Time Fourier Transform (STFT) based phase vocoder, but our advantage was to come from using a Dual-Tree Complex Wavelet Transform Phase Vocoder (DTCWT). The latter was assumed to be more efficient and to produce higher quality audio. However, after testing both phase vocoder techniques, we found that the DTCWT phase vocoder was significantly inferior to the STFT phase vocoder. Our goal was to measure the pace of the runner every minute, take a song within a -15/+10 BPM range of the pace, and warp it to the necessary new BPM. The phone app is very intuitive and easy to use, so the user just puts in their own music and presses play; the rest is taken care of automatically.

B. Design Requirements

Central to our problem is the acquisition of accurate step detection data. For a proper solution, we must be able to achieve an accuracy great enough to ensure optimality for the runner. State-of-the-art algorithms from recent years have attained around 94% accuracy (defined to be true positives out of total steps measured) [2], and so our goal was to produce the same results. We believe this was a worthy objective, as gaining an even greater accuracy would require research or expertise that would require more time than our constraints for the project allowed. State-of-the-art results should be enough to satisfy users.

Next, we wanted to begin playing music within 225 milliseconds after the "play" button on our main app activity is pressed. This measurement falls in line with human reaction speed. Essentially, our goal was for there to be too little time for users to react between the time they expect the music to begin playing and when it actually starts.

Then, from personal testing, we have found that our pace is between 150-180 steps/minute when we run at the speed of an average long-distance runner. As our target is to match beats in the song to user steps, we decided to impose a requirement that all songs in the playlist originally have tempos between 150 and 180 BPM. Lastly, we constrained these songs to stay between -15/+10 BPM of their natural tempo after the warping process. In our personal experience, we have found that music scaled outside of this range begins to pick up too many artifacts to listen to enjoyably.

Furthermore, when warping songs, we wished for the pitch to stay relatively the same. Only 1-5 people in 10,000 have absolute pitch and can reliably distinguish incredibly small differences well [3]. Our project targets the average individual. Consequently, we necessitated that music should stay within 25 cents, or a quarter of a semitone, of the original. This is almost unnoticeable for most people [4].

Another specification we believe is crucial to the success of our product is the time between updates for songs. In more detail, we wanted the songs to be "updated" (warped) every 90 seconds. If the current song ends before the 90 second interval is over, then we wanted the next song to play immediately, but at the same tempo as the previous one. Hence, runners will not be impeded by the inevitable shift. Long-distance joggers tend to stay fairly consistent in their pace throughout their run. We believe that using a smaller increment of time may accidentally take too many inconsequential variations into account. For example, we do not want to change the tempo of the music based on brief fluctuations caused by a stop light, miniature stretches of muddy terrain, or untied shoelaces. At 90 seconds, the product would be robust to each of these situations.

Those 90 seconds should also encompass the time required for our algorithm to completely process the music. No more than 30 seconds should be used for this step. We believed this metric was reasonable for processing large chunks of data (eg song audio). Shorter restrictions will not give time for a reasonable algorithm to run, while longer ones may begin to extend the update time passed desirable.

Finally, we imposed one final design requirement upon our application. Given a list of songs, the next chosen song to be played should be based on how recently it was played. As a result, we could avoid irritation caused by repeatedly hearing the same song.

C. Architecture and/or Principle of Operation

The process of operation of *Run With It* is shown in Fig. 1. A more detailed block diagram is shown in Fig. 2. Prior to using the mobile application for its purpose, the user needs to input a set of songs into the app's playlist. This set of songs is what will be played back to the user during their run. The mobile app filters user song inputs. This is shown in the block diagram as "Music Filter" in the Android Mobile App. First, for the preliminary version of this product, the application will only support songs of WAV file formats. Additionally, to meet the defined design requirements, the application only allows songs of 150-180 BPM. The app will rely on the audio file's metadata on tempo as its "BPM Detection" method. At this point, the app is ready to use.

The user will start the app, and indicate to start music by pressing a play button. The application will receive the user's running data through the phone's built-in and interfaceable step counter sensor. These measures will allow the mobile application to calculate the user's pace, in terms of steps per minute, over a given time period. The time period in the preliminary version of the application will remain one minute.

The next component of the application is the music component. This includes the "Song Choice Algorithm", the "Audio Signal Transformation", and the "Time-Scale Audio Modification". The jogger's calculated pace will be sent to the "Song Choice Algorithm" and the "Time-Scale Audio Modification Algorithm".

The song-choice algorithm chooses a song from the user's playlist with a score-based algorithm that depends on (1) the proximity of a song's tempo to the jogger's pace, and (2) how recently the song was played. The song's original tempo is passed in from the "Music Filter", and the jogger's pace will come from the "Pacing Calculations". The chosen song will be inputted into the "Audio Signal Transformation" function to read the audio signal from its provided WAV format into matrix representation such that it is compatible with the "Time-Scale Audio Modification Algorithm".

The "Time-Scale Audio Modification Algorithm" then modifies the song such that the new tempo matches the jogger's pace, as acquired from the pacing calculations. This modified song will be played back to the user during their run.

The pacing calculations repeat every minute. However, the next modified song, in accordance with the user's most recent pace, will only play upon completion of the previous song.

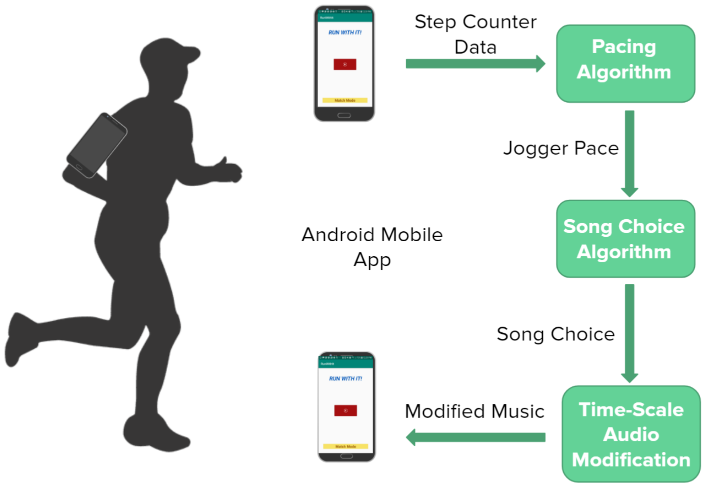


Fig. 1. Principle of Operation

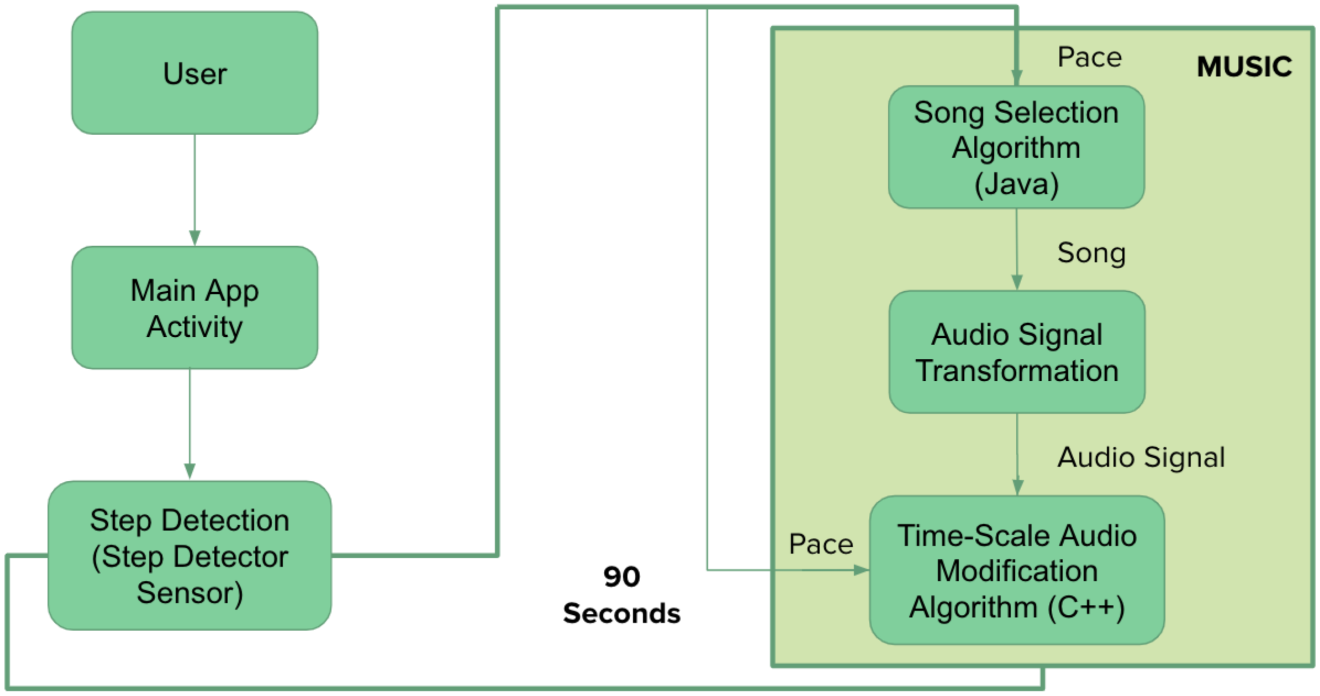

User
Main App Activity
Step Detection (Step Detector Sensor)
90 Seconds
MUSIC
Pace
Song Selection Algorithm (Java)
Song
Audio Signal Transformation
Audio Signal
Pace
Time-Scale Audio Modification Algorithm (C++)

Fig. 2. Block Diagram

TABLE 1. Device Properties

	iPhone	**Android**	**Android Smartwatch**
Sensor Accuracy	--	< 5%	10.4%
Development Language	Swift	Java & C/C++	Java (Wear OS)
Power Capacity	2700 mAh (10-17 hrs)	3000 mAh (15 hrs)	472 mAh (4 days)
Processing Power	6-Core 2.39 GHz	8-Core 2.8 GHz	Dual-Core 1.5 GHz
Ease of Use / Comfort	Low	Low	High
Storage	16+ GB	16+ GB	4 GB
Our Accessibility	1	2	1

D. Design Trade Studies

A. *Device Consideration for Pace Measurements*

We considered using several different devices for our project as shown in TABLE 1. The first was an iPhone, the second was a Samsung Galaxy phone, and the third was a Galaxy Smartwatch. For each, we looked at a host of device specifications that may affect our app.

As stated in the design requirements section, our ability to capture accurate step information was paramount to our overall project. Hence, we ran an experiment to investigate whether the built-in hardware sensors of the devices were accurate enough to meet our design specifications. The results are shown in TABLE 2.

The iPhone only displayed step measure updates every 10 minutes, so we were not able to use it in our experiment. Instead, two of us ran on a treadmill at 1 mile per hour increments between 5 miles per hour and 10 miles per hour. We counted our steps manually and compared it to the device measurements for 30 seconds, 1 minute, 2 minutes, and 5 minute intervals.

TABLE 2. Step Counter Accuracy

	Samsung S7	**Samsung S9**	**Galaxy Watch**
Mayur Error	13.80%	2.34%	8.36%
Aarushi Error	7.37%	5.83%	12.41%
Error per Device	10.59%	4.08%	10.38%

For each measurement, we ran with all three (Samsung S7, Samsung S9, Samsung Galaxy Watch) devices simultaneously. We found that the Samsung S9 was accurate enough for the design requirement of 94% accuracy, but the other two devices were not.

Also, we checked the accessibility to each of the devices. Noticeably, each device is expensive. Phones can cost upwards of our budget ($600), while a single watch can take up to half of it. Therefore, a crucial property of the devices is whether we can use them at all. In our case, the three of us have one iPhone, two Samsung Galaxy phones, and one Samsung Galaxy Smartwatch between us.

Next, we needed to check the memory of the devices. After all, the app and related audio takes up some amount of memory. However, the minimum of the three was 4 gigabytes, which is clearly more than enough for the application. Similarly, the power capacity of the devices was analyzed. Again, even the minimum was enough for our needs.

Perhaps one of the greatest advantages of the watches is their comfort as compared to a phone. If using a phone during the course of a run, you have to either hold it in your hand or buy a sleeve to attach it to your arm. Even the latter does not always fit comfortably. Meanwhile, a watch is simple and lightweight.

On the other hand, phones are somewhat better for app development. Audio processing applications obviously benefit from stronger processors. Phones have significantly faster clock speeds as well as more cores. As shown in TABLE 1, the Android phone has four times the number of cores as the watch and almost twice the clock speed.

Phones also tend to have more help resources than watches. After all, most developers create apps for phones rather than watches. Furthermore, we found that the Android Smartwatch requires an additional Wear OS package to be integrated with the app, which may add complexity to the overall application. In general, Android is developed using the Android Studio IDE, which uses either Java or Kotlin. The three of us have prior experience with Java. Swift, the primary development language for iPhones, is unfamiliar territory though. We believe it is important to consider our familiarity with the languages in our final design decision since our limited time frame might be further shortened if we had to spend time learning a new language.

B. *Programming Language per Component*

As previously mentioned, we factored our familiarity with languages into consideration. Our team members have the most experience with Python, and so researched the possibility of using it with Android devices. TABLE 3 lists each software that helps integrate Python with Android Studio and the associated issues with each of them.

SL4A was a community-developed software tool to port Python code to Android devices. However, it is no longer in development. QPython is built on top of SL4A and may allow for top-down development in Python, but lacks a way to access Android's Step Counter and Step Detector hardware

TABLE 3. Python on Android

Software	Issue(s)
SL4A (Scripting Layer for Android)	Dead! No longer in development
QPython (Script engine that extends SL4A)	No way to access Step Detector
Kivy	No way to access Step Detector Not well documented Complex: Requires 2 more software tools to work on Android
External Tools	Doesn't act as a callable function

sensors. Kivy is a comprehensive software tool that allows users to create a GUI and associated logic for almost any device in Python. As a tradeoff though, it is not completely optimized for Android. To get it working on Android devices, it needs another two software tools and a Linux OS to work correctly. Getting all three tools to work with each other adds complexity to the project. Lastly, we explored the option of using Android Studio's External Tools menu to import code. But, this option does not allow us to call our functions inside our main Java app. Since we needed to provide arguments to functions and call them repeatedly for our algorithms, the External Tools menu was not feasible for our use case. We also considered using MatLab, since the team has prior experience in using it with the wavelet transform algorithm. However, MatLab has the same issue as Python when trying to use it for apps; it is not natively supported by Android Studio.

As expected, Java and Kotlin were two languages that could be used to build the entirety of our app software. These two languages are both supported by Android Studio, and are in fact the main languages used to design the app UI. Although preference seems to be moving towards Kotlin, there does not seem to be any significant advantage to using it for a simple application like ours.

Nevertheless, there were options aside from Java and Kotlin. Namely, the C/C++ languages are natively supported with the IDE. These languages are known to be fast and effective for audio processing, but are less familiar to the team. Their speed made them strong choices for implementing the audio scaling algorithm. Additionally, while they are supported, they are not trivial to use. Time had to be allocated towards gaining familiarity with the Java Native Interface (JNI), which allows Java and C/C++ to be integrated, and to learn the syntax and libraries used in C++. One last advantage of allowing the time warping component to be in C++ was the fact that MatLab code could be mostly converted to C++ by way of the MatLab Coder software.

C. *Music File Format*

Android smartphones support audio in the file formats of MP3, WMA, WAV, MP2, AAC, AC3, AU, OGG, and FLAC [7]. These file formats do not all include tempo or BPM metadata for musical files. However, additional metadata tags can be appended to these file formats.

The listed file formats are also not equally readable by signal processing algorithms. Signal processing is most commonly performed with WAV file formats. This is because WAV files are lossless and uncompressed. As a result, they lose no quality from the original audio [6]. Since Android smartphones support this audio file type, the preliminary version of this product only supports WAV files. Decreasing the scope of allowed file formats is imperative to ensuring that all processed audio in our C++ algorithm is treated the same way.

For our testing purposes, we used third party online softwares that converted our files from MP3 or MP4 into WAV files.

D. *Time-Scale Audio Modification Algorithm*

TABLE 4, below, shows the various possible methods that could successfully implement time-scale audio modification. It also displays their tradeoffs according to Livingston's studies [8].

The first methods considered are simple time domain

TABLE 4. Time-Scale Audio Modification Algorithm Tradeoffs

	Examples/Process	Computation Time	Artifacts	Cause of Artifacts
Time Domain Techniques	Overlap Add (OLA), Synchronous OLA, Time-Domain Pitch Synchronous OLA	inexpensive	"buzziness" / "choppy"	period discontinuities
STFT based Phase Vocoder	STFT & ISTFT additive synthesis	O(NLogN)	"glissing" / "phasing"	tradeoffs between window size & computation / loss of resolution through unmatched window overlap
Dual-Tree Complex Wavelet Transform Phase Vocoder	Repeated DT-CWT, phase unwrapping, map coeff, IDT-CWT	O(N)	Few erroneous frequency components	minimized aliasing effect, higher temporal resolution, limited frequency resolution

techniques. This includes overlap-add (OLA) and its variations such as synchronous OLA (SOLA) and time-domain pitch SOLA (TD-PSOLA). These methods are listed in order of least to most expensive, in terms of computation time. While these methods are more efficient than the others listed in the chart, these methods result in the most drastic and noticeable artifacts. While pitch and magnitude of signals are preserved, the artifacts are a result of segmentation and simple time-scale stretches that do not result in smooth tones [8].

The next method we considered was the short-time Fourier transform (STFT) based phase vocoder. This method has successfully been used by a 18-500 capstone group in the past. It is assumed to be efficient in its simplistic implementation. However, the simplistic implementation creates greater artifacts. This is because smaller window size implementation lends to greater resolution in the time domain of the signal processing. However, smaller window sizes increase the number of windows being processed. As a result, this increases computation time. This advanced and more accurate approach results in a computation of time that is considered inefficient [8].

Lastly, we looked at the Dual-Tree Complex Wavelet Transform (DTCWT) based phase vocoder. It is expected to be slightly more efficient in comparison to the STFT based phase vocoder. It also results in fewer artifacts - namely minor erroneous frequency components. The DTCWT based phase vocoder is a modification to the attempt of using discrete wavelet transform (DWT) as a basis for phase vocoders. Namely, the DTCWT improves upon the DWT base by providing shift invariance where previously it did not exist. This decreases the variations in the distribution of energy that is found between DWT coefficients. As a result, a DWT based phase vocoder would produce the same artifacts as seen through the STFT based phase vocoder. The DTCWT, however, is known to perform better since its shift invariance deters the aliasing effects otherwise observed [8].

The described literature studies clearly show numerous advantages to using the DTCWT phase vocoder. Thus, our implementation of the time-scale audio modification algorithm was originally going to follow this preference. However, we tested and compared two of the described audio modification methods: the STFT phase vocoder, and the DTCWT phase vocoder, and had surprising results.

The STFT phase vocoder was implemented in Matlab and was adopted from Dan Ellis's implementation [1]. The DTCWT phase vocoder was also implemented in Matlab, and was adopted from Livingston's implementation [8]. Both implementations were appropriately modified for our purposes. These alterations included function inputs, function outputs, array and matrix sizes throughout the original audio signal's modification process, and including functionality to visualize pitch and hear differences between the output signals.

In comparing the two time-scale audio modification techniques, we performed two basic experiments that compared the results of using the two techniques to modify

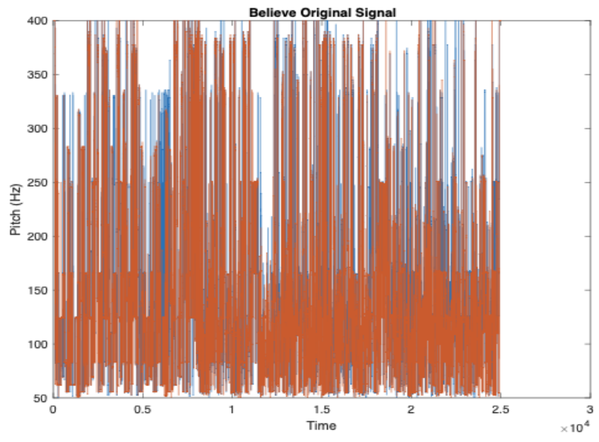


Fig. 3. Experiment 1: Original Signal Pitch vs. Frequency

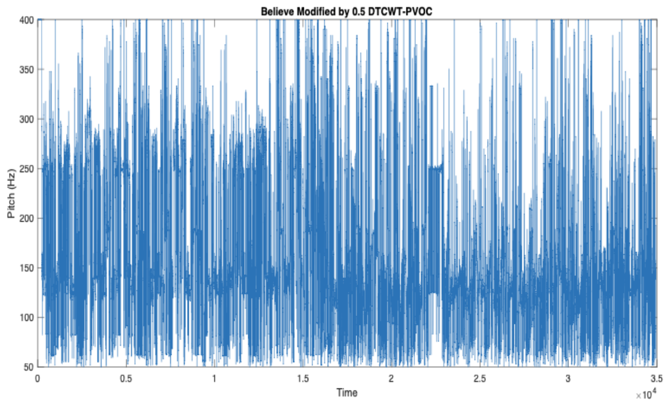


Fig. 4. Experiment 1: DTCWT Phase Vocoder Signal Pitch vs. Frequency

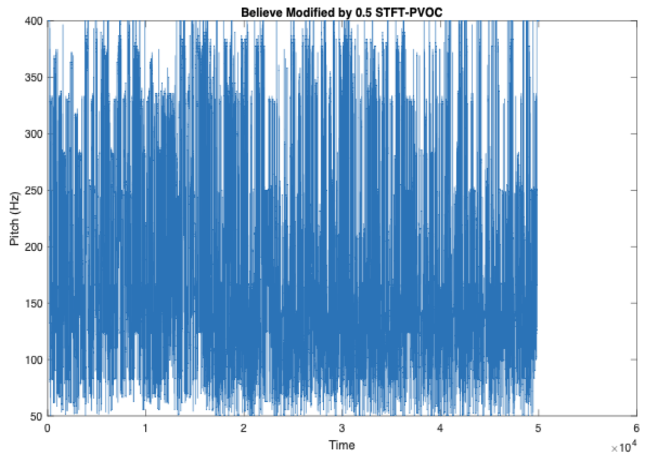


Fig. 5. Experiment 1: STFT Phase Vocoder Signal Pitch vs. Frequency

music by half of its original BPM. It is important to note that the DTCWT signals are cut off due to its implementation that requires padding the original array such that its size is a power of 2, and due to Matlab array size restrictions.

The goal of experiment 1 was to test the audio modification's impact on vocals, and to test the speed of the audio modification techniques. The audio used was the full song: "Don't Stop Believing" by Journey. The song's duration was 4:10 minutes, and its BPM was 121. The pitch of the original signal in relation to time is shown in Fig. 3 and is titled "Believe Original Signal".

The result of experiment 1's DTCWT phase vocoder output is shown in Fig. 4, and the result of experiment 1's STFT phase vocoder is shown in Fig. 5. In comparing the pitch vs frequency graphs, it is clear that Fig. 4 (the DTCWT phase vocoder output) shows significantly greater data loss than that of Fig. 5 (STFT phase vocoder output).

The goal of experiment 2 was to test for accuracy in the audio modification process with music of more complex layers, and to test the algorithms' performances on a song within our range of 150-180 BPM. The audio used here was the instrumental intro of the song: "Eye of the Tiger" by Survivor. This intro snippet's duration was 46 seconds, and its BPM was 165. The pitch of the original signal in relation to time is shown in Fig. 6 and is titled "Tiger Original Signal".

The result of experiment 2's DTCWT phase vocoder output is shown in Fig. 7, and the result of experiment 2's STFT phase vocoder is shown in Fig. 8. Again, it is clear the Fig. 7 of the DTCWT phase vocoder output shows significantly greater data loss than that of the Fig. 8 STFT phase vocoder output.

In Experiment 1, the warping process with the STFT phase vocoder took 36.33 seconds, and the warping process with the DTCWT phase vocoder took 212.68 seconds. In Experiment 2, this process with the STFT phase vocoder took 2.37 seconds, and this process with the DTCWT phase vocoder took 29.87 seconds. Through these timing measures. we see that the runtime of both algorithms varies and is dependent on the size of the input audio signal that must be modified. However, we consistently see that the STFT phase vocoder takes drastically less time to compute audio modification in comparison to the DTCWT phase vocoder.

While measuring pitch loss visually and through pitch values is consequential for verifying whether these audio modification methods meet our design requirements of 25 cents variance between the original and modified signals, audible pitch loss is most important to our use case, since our user's will be listening to the modified music. To that end, we compared the modified signals between the STFT phase vocoder and the DTCWT phase vocoder for each experiment. Again, the results for each of these experiments were the same: the DTCWT phase vocoder generated fuller sounds. However, on average, they were unclear. On the other hand, the STFT phase vocoder generated thinner sounds which caused pitch to definitely increase, but the sounds were clearer, which made the output signal discernable as the original song that was modified. It is important to note that there are significant extraneous artifacts generated by both audio modification techniques in these experiments since we are modifying the songs to be half of their original BPM. Thus, this is an intense modification that suffers intense artifacts from the output signals. There will be less artifacts when computing the modified signals for the purpose of *Run With It* since we restrict all modification to remain with -15/+10 BPM (at most 10% modified).

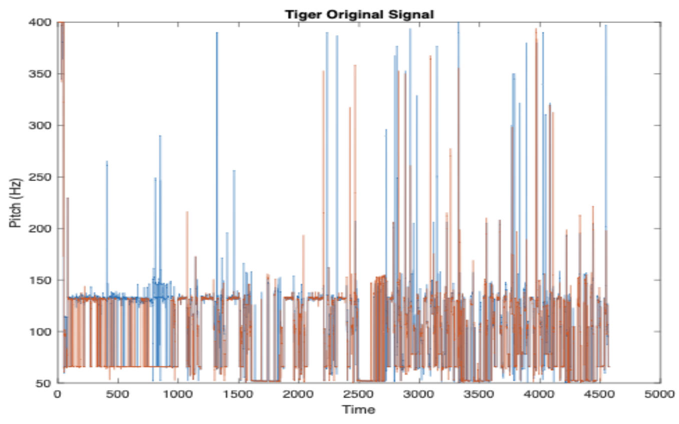


Fig. 6. Experiment 2: Original Signal Pitch vs. Frequency

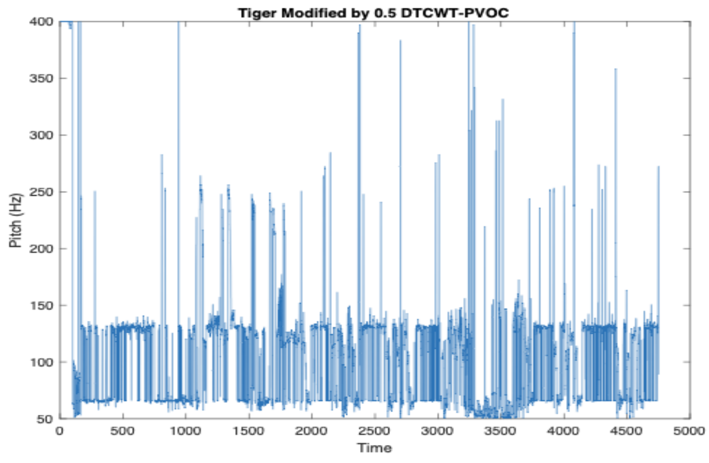


Fig. 7. Experiment 2: DTCWT Phase Vocoder Signal Pitch vs. Frequency

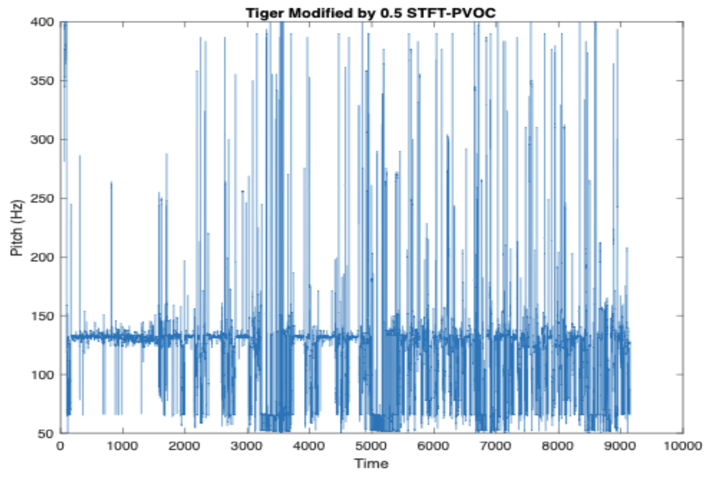


Fig. 8. Experiment 2: STFT Phase Vocoder Signal Pitch vs. Frequency

Through these two experiments, we compared STFT and DTCWT phase vocoder techniques for signals of differing lengths, BPMs, and layered sound types. We learned that regardless of variations of these song characteristics, one of the time-scale audio modification techniques always performed far better than the DTCWT phase vocoder. This was in terms of speed, visual pitch loss, and audible song comprehensibility. Thus, we implemented the superior time-scale audio modification technique in our app: the STFT phase vocoder.

E. System Description

A. *Mobile App*

We decided to use Android Studio for developing the Android application. This allowed us to leverage our knowledge of Java while taking advantage of the rich community resources provided. Furthermore, Android Studio is the environment of choice for developing Android mobile applications. We chose to use Java in Kotlin because of our familiarity with the language, and the lack of disadvantages with using the former over the latter. The app has a single Main Activity that users can use to start the app. Once the "play" button is pressed, the app will trigger our business logic by calling the song selection algorithm with a default tempo. A selection of songs, already tagged with tempos, will be chosen from a playlist. The first screen of the app is shown in Fig. 9.

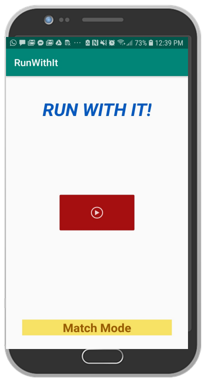


Fig. 9. Main Activity

For signal processing, we used Matlab and ported the code to C++, and for the song selection algorithm, we implemented it directly in Java. We found research that showed the DTCWT and STFT implemented in Matlab. Furthermore, it is extremely fast compared to some other languages. In the end, the complexities of integrating Python with the IDE and its relative slow speed were too great of sacrifices to use it for our project.

The Main Activity is the main point of interface between other subsystems. Pace measurements from the step counter system are taken here and sent to the song selection algorithm. A queue is implemented to deal with songs that do not end on 90 second intervals. Chunks are taken from the queue and sent to the time-scale audio modification algorithm to be warped. When the audio is returned, it is directly fed to the user.

The speed that the music starts playing after hitting the start button was an important metric to test. We required that the music start playing within 225 ms of the user hitting the play button. We tested this multiple times by simulating a timer in the code. We found that the maximum time between the button press and the music playing was 5 ms; thus meeting our human reaction time requirement.

B. *Step Detection Sensor*

Since we created the application on the phone, we felt that in order to make the user experience better, we should minimize the amount of extraneous hardware the user must wear. For that reason, we decided to use the step counter sensor in the smartphone since it is always included and integrated with the system we are using. We found the step counter sensor superior to the step detector and accelerometer sensors, since it is optimized in the hardware of the phone itself, and already accounts for false positives in footsteps. Before we blindly chose to use the smartphone for collecting the step count data, we wanted to test it to make sure it was accurate enough to work for our project.

To test the sensor, we recorded the data from the phone and our manual count from the treadmill tests, corrected for error, and calculated the difference. In the end, we found that the newest generation of phones had 94% accuracy, and the generation prior to that had 90% accuracy. With this data, we felt confident that the phone was a good device to use to collect our data and use for our project.

We found after implementing the step counter sensor that it provided inaccurate data because it removed the false positives. We also found that it was much slower since it took time to process the data to remove the false positives. We could not have this as we wanted to collect data on exactly 60 second intervals, therefore we went with implementing the step detector function which still met our 94% accuracy requirement and was able to report the data on our needed time intervals.

We considered using an iPhone as well but had too many issues trying to collect the footstep data from it, so it ruled out that device. We also considered the Samsung Galaxy Watch, however, we found through our testing that the error was quite high which made us unsure whether it would be reliable enough to use as the main device collecting and sending our data. For that reason we ended up using Android smartphones, which made it easier for us to build the app too, since we could just use Android Studio to build it and test it right on the phone.

C. *BPM Refresh Rate*

The tempos of running songs chosen are between 150 and 180 BPM, following our design requirements exactly. Additionally, no song should be warped to a tempo that is less than 15 or greater than 10 beats per minute of its original. We chose to use a predefined list of songs tagged with preprocessed tempos. Note that it is trivial to ping app APIs to obtain song tempos. Thus, we felt it unnecessary to implement a custom tempo detection algorithm. If this project was extended into a real product, integrating the app with a music player like spotify would eliminate the need for the tempo detection.

The refresh rate for songs is 90 seconds. This allotted 60 seconds for determining pace and 30 seconds for audio modification. After a song ends, a new song is chosen to continue playing at the user's pace; in essence, the BPM between the previous song and the new song should theoretically be the same if a runner's pace remains the same.

The reasoning for picking 90 seconds is because we found through our testing that collecting less than 60 seconds of running data was inaccurate and also would require us to make extra calculations that getting exactly 60 seconds of data does not require. The extra 30 seconds is used to send the song through the time scale audio modification algorithm, which we found takes up to roughly 30 seconds for a whole song using the implemented STFT phase vocoder. If we chunk the songs into smaller portions, it would take even less time to warp.

D. *Song Selector Algorithm*

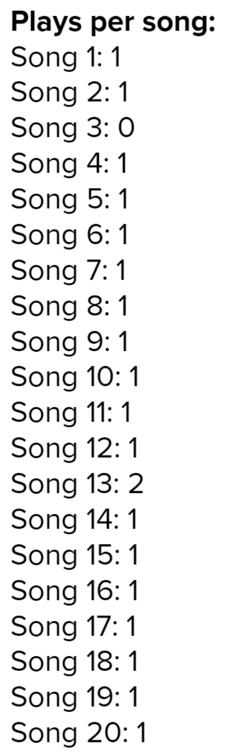
This component is software that is very important for the end quality of the music that is produced by the app. This component takes the pace of the runner that was previously collected and uses it to find the next song to play- the one that best matches the runner's current pace. As stated earlier, we require that the song should fall in a -15/+10 BPM range of the runner's pace so that when the song goes through the time-scale audio modification algorithm, it will still sound pleasant to the ear of the user.

The songs are scored on a scale where 10 is the perfect song and all other songs are scored relatively (including negative scores). Our goal is for the BPM of the song to be close to the pace, so the distortion and artifacts created in the song are kept to a minimum after the audio modification. Thus, the scoring works in a way where a smaller difference between the runner's pace and the BPM of the song will score the song higher, and the opposite will score the song lower. Because our goal is to keep the audible distortion to a minimum, we would rather have the algorithm replay a song that fits, rather than pick a song that is out of the range. However, to take that into account, there will be a penalty system for playing a song more than once. Thus, we start each song with a score of 10. From there, we subtract the absolute value of the runner's pace from the song BPM and divide that by 10, where dividing by 10 provides a scaling factor since the highest score a song can have is 10. Finally, we subtract a multiplier if the song has been played multiple times (i.e, subtract 1 point each time a song has been played).

Only songs that fit within the -15/+10 relative BPM between the song and runner are scored, if the song does not meet this requirement, it is skipped for that round of evaluation.

As mentioned earlier, the songs scores can go negative if the song has been played enough, this is allowed as all the songs are scored and picked relative to each other. Therefore, the actual number does not matter as much as the rankings based on the scores.

Overall, this system is not too complicated, as we are going to tag the songs with their BPMs and we just need to compare that to the base runner's pace and find the best song. Through our testing, we found this algorithm to be quite fast on a playlist of 20 songs. We tested with all different user input BPMs to test the different aspects of the requirements. In all cases we found our algorithm to meet the requirements. Fig. 10 shows the results of running the algorithm on our own running data. Looking at the list, Song 3 is never played and Song 13 is played 2 times. This shows that even though there was deduction for Song 13 having already been played once, it was still closer in BPM than Song 3, thus got selected again.



Plays per song:
Song 1: 1
Song 2: 1
Song 3: 0
Song 4: 1
Song 5: 1
Song 6: 1
Song 7: 1
Song 8: 1
Song 9: 1
Song 10: 1
Song 11: 1
Song 12: 1
Song 13: 2
Song 14: 1
Song 15: 1
Song 16: 1
Song 17: 1
Song 18: 1
Song 19: 1
Song 20: 1

Fig. 10. Song Selection Algorithm

E. *Audio Signal Transformation (for Language Integration)*

In order for the JNI to work, we needed to transform the Java types that were output from the step detector and song selection algorithm into types readable by C++. Secondly, we had to find a way to transform the output of the time warping algorithm into something readable by Java. Lastly, our use of MatLab Coder transformed the STFT phase vocoder code from MatLab into C++. However, not all functions can be fully ported by the software. For example, we wrote the functionality of the Matlab audioread() function in C++, since the program does not have an analogous function provided in C++. Ironically, the argument passing was the last step of integration missing from our system. While we wrote our own implementation, it was not fully functional by the deadline of our project. We believe we had the more theoretically complex audioread function and other integration steps fully complete.

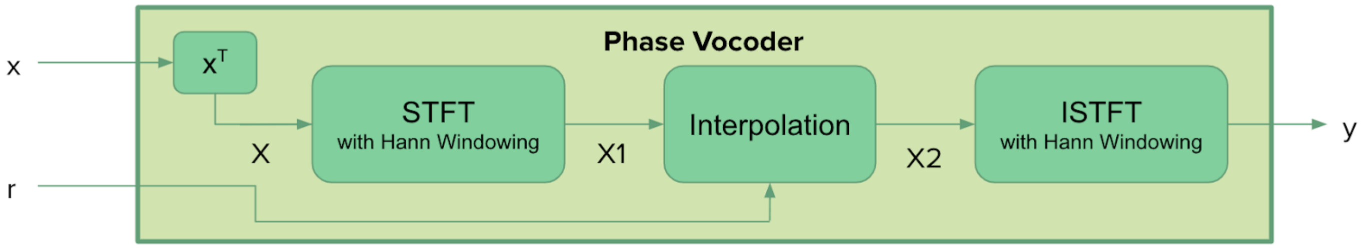


Fig. 11. Time-Scale Audio Modification Block Diagram

F. *Time-Scale Audio Modification Algorithm*

The time-scale audio modification algorithm used in the prototype of this mobile application is a short time fourier transform (STFT) phase vocoder algorithm. Fig. 11. depicts a block diagram of how this algorithm works. The phase vocoder takes two inputs: (1) original input signal, x and (2) the change ratio, r.

$$r = current\ running\ pace\ /\ original\ song\ BPM \qquad (1)$$

While the original input signal matrix is acquired from the Audio Signal Transformation method, the change ratio acquires the current running pace from the Step Detection function, and the original song BPM from the song's metadata when it is imported into the Song Selection algorithm.

The original input signal, x, is transposed to X. The STFT coefficients of X are calculated using Hann windowing which samples and smoothes the signal symmetrically. The STFT coefficients are stored in an array, X1, which is used to reconstruct the signal in accordance to the change in phase as seen by the change ratio r. Finally, performing an inverse STFT on the reconstructed signal, X2, transforms the modulated signal back into waveform in the same way the original signal's STFT coefficients were deconstructed. This output, y, is the modified signal in accordance with the change ratio, r, of the original song's signal, x.

F. Testing/validation

TABLE 6. Implemented System Values

Design Specifications	**Specifications of Implemented System**
Step detection of at least **94%** accuracy	Achieved up to **95.4%** accuracy
Music plays within **225 milliseconds** of button-press	Music plays within **5 milliseconds**
Music tempo range: **150 - 180 BPM**	Implemented as a restriction when inputting music
Require music of **-15/+10 BPM** relative to pace	Implemented as hard constraint within software
The probability of playing any song in playlist is equal	RNG for initial song; else algorithmically chosen
Music should be modified every **90 seconds**	Implemented; dummy fn pinged on time
Pitch should stay within **25 cents** of original (1.01 Hz)	Within around 5 Hz on average

Our first design specification related to the accuracy of our app in detecting each step of the user. For an unknown reason, the S9 phone we originally planned to use stopped providing data after a small window of time. We turned to using the S6, a model with the theoretically least accurate step detector sensor, to test this metric. We ran for several minutes at a time several times in a row, manually counting our steps along the way. For each individual run, we compared the measured steps (sent as an email on the app) versus our manual count. Overall, we found that we achieved about 95.4% accuracy when detecting individual steps. It is reasonable that we achieved higher than our goal, as the research papers we consulted measured their accuracy over a wider variety of paces and terrains, for which their results suffered. Our narrow scope and use of the built-in sensor let us achieve high results.

Next, we added a timing block within our code to find how long it took our app to begin playing music after the "play" button was pressed on the main activity. With 20+ presses, the longest time taken was only 5 milliseconds, which was far better than our original goal.

Many of our specifications were simply implemented directly in the code. A minimum and maximum function was used to make sure that the BPM was the maximum of 150 and the measured, or the minimum of 180 and the measured. All songs which were added to the playlist had tempos between 150 and 180 BPM as well. Together, these points satisfied the requirement for music to always stay in the desired range. Similarly, the -15/+10 requirement was implemented with min/max functions too. Yet another requirement that we achieved in software was the constraint that music was to be warped every 90 seconds. While this part of the integration was not complete, the native function was called every 90 seconds via a timer in the code.

As described in the design requirements, we wanted the probability of choosing any song within the playlist to play to be equal. For the very first song, we choose a random number and use the modulus function. Other songs are picked as described in the song selection algorithm section above. Hence, the probability of any song being played is roughly equal.

Our last design specification was that the pitch of the modified signal should be within 25 cents, or 1.01 Hz, of the original signal's pitch. With the STFT phase vocoder time-scale audio modification algorithm, we did not successfully meet this system requirement. A few of our experiments in testing this specification are shown below in TABLE 7.

TABLE 7: STFT Phase Vocoder Pitch Variance Experiments

Pace / BPM	Mean Pitch	Computation Time	Signal's Max, Min Pitches	Pitch at Original Signal Min
Original (165/165)	108.05 Hz	1.40 sec	400 Hz, 50 Hz	50 Hz
150/165	113.52 Hz	1.41 sec	400 Hz, 50 Hz	50.00 Hz
175/165	111.82 Hz	1.27 sec	400 Hz, 50.17 Hz	341.86 Hz
140/165	115.22 Hz	1.59 sec	400 Hz, 50 Hz	52.88 Hz
200/165	114.09 Hz	1.17 sec	400 Hz, 50 Hz	65.43 Hz

🟧	IN RANGE
🟪	OUT of RANGE

We performed four experiments on the same "Eye of the Tiger" intro instrumental snippet as used in Section D.D's experiment 2. Each experiment is unique by the change ratio (Pace/BPM). The change ratio for two of the experiments, shaded in orange, are within our design specifications of -15/+10 BPM modification. The change ratio for two other experiments, shaded in purple, are outside of our design specification range of -15/+10 BPM modification.

For each of the experiments we calculated and compared the following characteristics of our audio modification results: (1) the mean pitch of the signal, (2) computation time, (3) the minimum and maximum pitches of the modified signals, and (4) the modified signal's pitch at the corresponding index of the original signal's absolute minimum. These characteristics are represented as individual columns in TABLE 7.

We see that computation time on the same signal varies slightly amongst runs; however, the change ratio has no impact on the runtime, as expected. We also see that each of the signal's absolute minimum and maximum pitches are of the same frequency. This implies that the signals' frequency ranges have not been altered overall.

We see differences in the mean pitches, and the pitches at original signal minimum columns. The mean pitch for the modified signals *within* our range are within 5 Hz of the original mean pitch. The mean pitch of the signals *outside* of our range are similar but greater than those of the signals within our range. In the pitch at original signal minimum, we see that the pitch for the 150/165 change ratio remained the same as the original pitch at 50 Hz. This held true for change ratios around 170/165 as well. The 341.86 Hz value that we see for the 175/165 change ratio is a major artifact that was formed in this one specific test case. Although it is not ideal, it presents an accurate depiction of some unexpected artifacts through this audio modification. Additionally, it was an important change ratio to record since it is the boundary of our -15/+10 BPM modification range. We see that this measure for the change ratios *outside* of our range are comparatively close to the original pitch of 50 Hz. However, the majority of the values for this measure, for the experiments *within* our -15/+10 BPM modification range, are within our 1.01 Hz pitch variance specification. Thus, the 52.88 Hz, and 65.43 Hz are relatively insignificant.

As a result, these test experiments have validated the necessity for our -15/+10 BPM modification range specification. Thus, this was programmed as a restriction into our overall system through the Song Selection Algorithm.

G. Project Management

A. *Schedule*

The Gantt Chart in Fig. 12 below shows a visual mapping of our roadmap for the semester. We broke down the schedule into small sections based on the tasks assigned to us. The chart is organized in sequential order. We spent a lot of time on research to figure out exactly what components our project needed, and from there, we divided the subcomponents to work on based on our expertise. Since the components are tricky, we wanted to make sure that we worked properly individually, which is why we put so much time into planning and testing before even starting the main parts of the project. We had to alter our original schedule a little bit after spring break to allot more time for integration, and some of our area implementations took less time. We knew being remote would make it harder to put the components together which is why we made this change to our schedule.

B. *Budget*

Our bill of materials did not contain many components, because we only required a smartphone and headphones. Our bill of materials is shown in TABLE 8.

Two of our team members own Android smartphones and bluetooth headphones, and we used those for testing and working on the project. We did not need to buy these materials for the project.

TABLE 8. Bill of Materials

Item	Price
Android Smartphone	$800
Bluetooth Headphones	$200

C. *Team Member Responsibilities*

Our project was split up into three parts, allowing each one of us to be assigned a different part and work on it. Aarushi worked on the signal processing phase vocoders. Akash worked on the song selection algorithm and the footstep detection with Android. Finally, Mayur built the app that hosts and would bridge each of the subsystems. He also ported the audioread() code and worked with Aarushi and Akash to integrate their code with the app. Akash and Aarushi worked together for getting the data from the step detection to the song selection, and then taking the song and data needed into the time-scale audio modification algorithm. Mayur helped Akash with the step detection with Android since that is all linked together in Android Studio, which we used to build our application.

Once we all got our individual parts done, we revised each individual subsystem such that they would be compatible with each other, and could be easily integrated. Finally, we tested the revised versions of each of our independent subsystems. We tried to optimize and improve features to make the overall product more efficient where possible. Our goal was to meet or surpass the metrics we set in the design requirements. The parts we picked match our strengths as teammates. Aarushi has done numerous projects in signal processing while Akash and Mayur have worked more on the software side. This allowed us to adapt easily and finish our parts on time, since we have decent knowledge on the parts we are working on.

D. *Risk Management*

We built risk management into our project in a couple ways. First in terms of the schedule, as we mentioned before, we added slack time into it to make sure we could account for any issues we ran into as we developed our project. This allowed us to work out these issues without running out of time at the end of the semester. With the rearranged schedule, we still had enough time to complete the goals for each smaller piece of the system.

From the design side, we had backups for potential problems that could have come up while working on the project. We had four specific cases for our metrics that we laid out earlier. The main risk factor was using the DTCWT phase vocoder. If it did not work, we would fall back on using the STFT based phase vocoder that we knew works well for music. We considered the possibility of having to implement our own, or simply using a library on GitHub. We did our research and put most of our time into this aspect of the project since it was the focus point that differentiates us from other apps like this, while also being the primary risk factor. As it turns out, we were able to verify that the STFT based Phase Vocoder performed in a superior fashion to the other. Our risk management worked well here.

The second biggest risk factor was the accuracy of the step detection from the smartphones. We did testing and saw that the phone met our accuracy requirements, so we hoped that this would not be an issue, but if we had found out during implementation and testing that the accuracy was not as good as we thought, we were going to order a pedometer that we could collect the data from instead. The smaller risk factors involved the timing of the application, which would result in us widening the timing windows for our refresh rate, and minimizing the time it takes for the app to start when initially opened. We found no issues with the step detection or timing, and thus did not need to fall back on back-up plans.

In terms of budget, we did not run into any issues since we already had everything we needed.

H. Related Work

There have been several similar products or projects to our own. Spotify used to have a Running Feature that matched the music with running cadence. In fact, the algorithm used the phone's internal hardware and similar BPMs as ours. However, this feature was retired in 2018 and no longer exists.

Some apps exist that implement similar features as well. PaceCoach and PowerRunner are two examples, but only exist for iPhones. This is in contrast to our app, which runs on Android devices. A Play store app is RockMyRun, which offers playlists that can match music tempo with runner pace. However, matching BPM to steps is a premium feature. Furthermore, users can only listen to playlists instead of songs already on their device.

In the past, two similar capstone projects have been attempted by other students. A common stumbling block for these groups is the actual audio warping; often, too many artifacts are picked up, making it difficult to enjoy listening to songs. DJ Run, a project from 18-551 during the Spring of 2013, managed to implement algorithms for both pace detection and tempo detection of songs. They used a phase vocoder algorithm for the time-scale audio modification. We planned on using the DTCWT phase vocoder instead, since we assumed it to be more accurate based on our literature reviews, However, as seen in Section E, we learned that the STFT phase vocoder was the superior time-scale technique. Additionally, we did not recreate the tempo detection and pace detection algorithms, since those have already been implemented by others. Instead, we integrated those tested functionalities, and wrote an accompanying song selection algorithm.

During the Spring 2018 semester of 18-500, one team created the project "Song Hunter", which warped song for cyclists. They also used a vocoder. However, they noted that while the algorithm mostly worked, it sometimes left behind artifacts or a muffled sound. Obviously, the target audience for this project was different than ours.

I. Summary

To reiterate, *Run With It* is an Android mobile application that modifies music such that its beats per minute matches an average long-distance jogger's pace (steps per minute). It is limited by its target audience being the average long-distance jogger. As a result, this limitation caused the time period of pace and music modification to be 90 seconds. This limitation also shaped what kind of music this mobile application can interface with. The allowed music was restricted to be within a tempo range of 150-180 BPM - similar to the pace of an average long-distance jogger. These limitations contributed to defining requirements for a preliminary version of this product. Methods to improve these restrictions are discussed in the Future Work section J.

To conclude, the following is a summary of our considerations that influenced the product design of *Run With It*: step detection accuracy, software speed of responsiveness, tempo of supported music, optimal difference in original music tempo and jogger's pace, sequencing of music on a run, time period of pace and music modification, pitch difference between original and modified music, audio modification techniques, and the technological means to implement each of the components of the product (i.e. devices, languages, libraries, algorithms, etc.). Future work will entail creating advanced user options and functionalities of our thus far implemented prototype.

J. Future Work

While we currently do not plan on continuing this project after this semester, we have extensive plans for possible future work to bolster this product. First, we recommend reestablishing the design requirement for measuring quality of a modified audio signal. Along with implementing the product as designed in this document, we suggest extending the functionality of the product.

This extension plan includes expanding the tempo range of allowable music and expanding the file types of allowable music. This can be done by increasing the initial allowed music BPM range to 135-180BPM since our new running test data suggests that joggers also commonly run between 140-150 steps/minute. Additionally, music within the range of 68-90 BPM could also be included since this range is half of a runner's pace range. In this case, the modified music would simply use a new change ratio, r_{new}, as shown in Equation 2 below.

$$r_{new} = (current\ running\ pace/2)/original\ song\ BPM \quad (2)$$

This would be in accordance with our -15/+10 BPM modification design restriction.

Additionally, to advance the app to function better as a product, future work should include creating multiple features on the app that allow the user to set preferences for their run. Such preferences could include setting a tempo to be matched throughout a run (as opposed to the tempo matching the runner's pace), and/or setting a desired sequence for song play throughout the run (as opposed to the software choosing the most optimal song based on the runner's pace and previously played music). These preferences for user's to use the application as it fits to their personal needs.

Our app is based on research that proves that matched music tempo and pace enhances an athlete's energy and running experience. However, this has been realized to be contingent on the runner setting their own pace independent of any external factors. A few peer athletes gave feedback that if a runner attempts to match pace to the tempo of the music, then there will be a constant loop of pace and music tempo decreases. This may not be optimal for the running user. Therefore, we tested this concern by running to our time-scaled music, and measuring our pace accordingly. From this personal experience, we agree that listening to a slowed song causes an urge for the runner to slow their pace. As a result, we suggest that the app should include a mode that allows the user to set their desired running pace, and scale the music only to match the runner's pace *if* the runner's pace is greater than the runner's desired pace. This would allow the runner to speed up and have the music match their faster pace. However, if the runner's pace was slower than their desired pace, the music's BPM would not slow down enough to prompt the user's pace to slow down below their desired pace.

Additionally, the application could be further integrated with a user's preferred music applications such as Spotify or Google Play. These settings and extensions would give the user more control and accessibility with their playlist, without defeating the purpose of *Run With It*.

References

[1] D. P. E. Ellis, "A Phase Vocoder in Matlab", 2002. [Online]. Available: http://www.ee.columbia.edu/~dpwe/resources/matlab/pvoc/

[2] H. Lee, S. Choi, and M. Lee, "Step Detection Robust Against the Dynamics of Smartphones," *Sensors*, vol. 15, no. 10, Oct. 2015. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4634483/. [Accessed Jan. 20, 2020].

[3] WA Brown et al, "Autism-related language, personality, and cognition in people with absolute pitch: results of a preliminary study," *Journal of Autism and Developmental Disorders*, vol. 33, no. 2, Apr. 2003. [Online]. Available: https://www.ncbi.nlm.nih.gov/pubmed/12757355. [Accessed Jan. 26, 2020].

[4] J.M. Zarate, C. R. Ritson, and D. Poeppel, "Pitch-interval discrimination and musical expertise: Is the semitone a perceptual boundary?" *The Journal of the Acoustical Society of America*, vol. 132, no. 2, Aug. 2012. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3427364/. [Accessed Jan. 26, 2020].

[5] A. Morton, *Average Foreground Battery Drain for Android App Categories - An M2 App Insight Report*, AT&T Developer Program, Feb. 2015. Accessed on: Feb. 29, 2020. [Online]. Available: https://developer.att.com/blog/average-foreground-battery-drain-for-android-app-categories-an

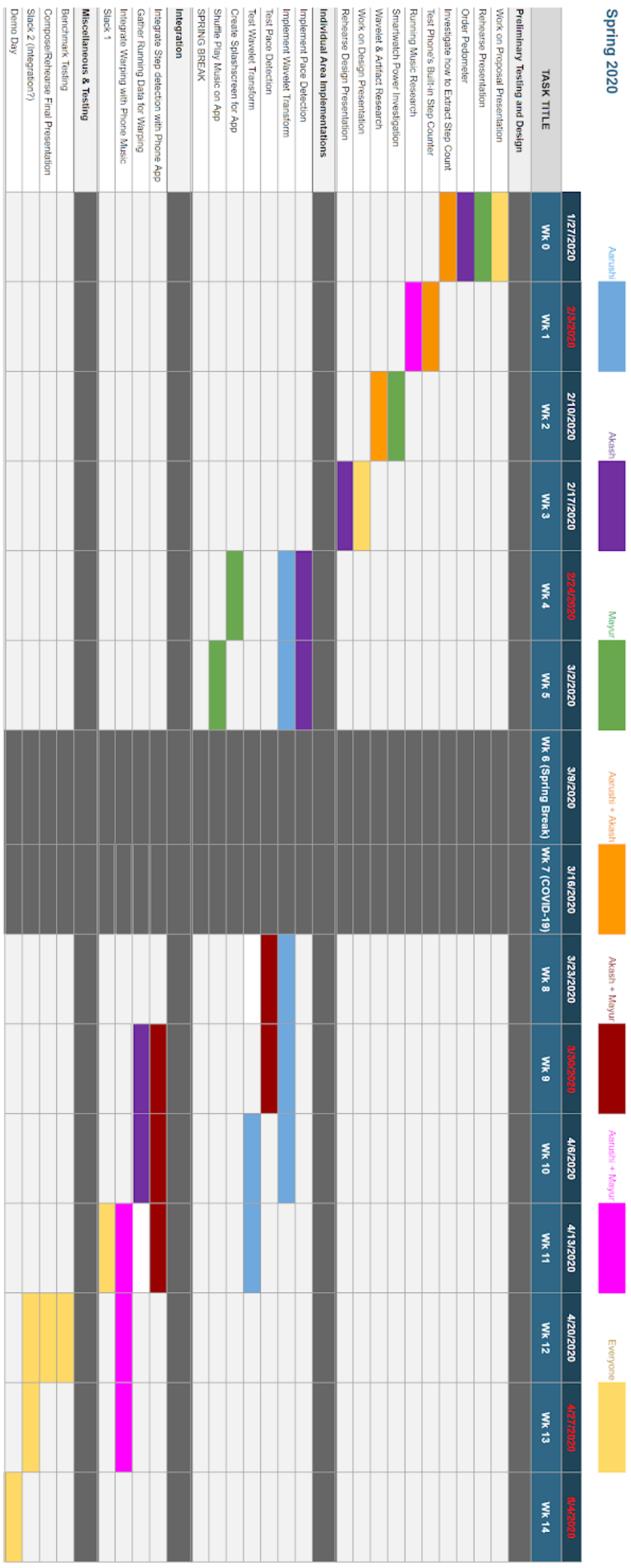
[6] J. Nugent, "WAV or MP3: What's the Difference?," *Audio Buzz*, 30-Jan-2020. [Online]. Available: https://www.audiobuzz.com/blog/wav-or-mp3-whats-the-difference/. [Accessed: 03-Mar-2020].

[7] "Supported media formats : Android Developers," *Android Developers*. [Online]. Available: https://developer.android.com/guide/topics/media/media-formats. [Accessed: 03-Mar-2020].

[8] J. B. Livingston, "Time-Scale Modification of Audio Signals Using the Dual-Tree Complex Wavelet Transform."

[9] "C Wavelet Libraries," *C Wavelet Libraries*. [Online]. Available: http://wavelet2d.sourceforge.net/. [Accessed: 04-Mar-2020].

[10] Kingsbury, N. G. *Complex Wavelets for Shift Invariant Analysis and Filtering of Signals.* Journal of Applied and Computational Harmonic Analysis, vol 10, no 3, May 2001, pp. 234-253.



Spring 2020

- 🟦 Aarushi
- 🟪 Akash
- 🟩 Mayur
- 🟧 Aarushi + Akash
- 🟥 Akash + Mayur
- ■ Aarushi + Mayur
- 🟨 Everyone

TASK TITLE	1/27/2020	2/3/2020	2/10/2020	2/17/2020	2/24/2020	3/2/2020	3/9/2020	3/16/2020	3/23/2020	3/30/2020	4/6/2020	4/13/2020	4/20/2020	4/27/2020	5/4/2020
	Wk 0	Wk 1	Wk 2	Wk 3	Wk 4	Wk 5	Wk 6 (Spring Break)	Wk 7 (COVID-19)	Wk 8	Wk 9	Wk 10	Wk 11	Wk 12	Wk 13	Wk 14
Preliminary Testing and Design															
Work on Proposal Presentation	🟨														
Rehearse Presentation	🟩														
Order Pedometer	🟪														
Investigate how to Extract Step Count	🟧														
Test Phone's Built-in Step Counter		🟧													
Running Music Research		■													
Smartwatch Power Investigation			🟩												
Wavelet & Artifact Research			🟧												
Work on Design Presentation				🟨											
Rehearse Design Presentation				🟪											
Individual Area Implementations															
Implement Pace Detection					🟪	🟪									
Implement Wavelet Transform					🟦	🟦			🟦	🟦	🟦				
Test Pace Detection									🟥	🟥					
Test Wavelet Transform											🟦	🟦			
Create Splashscreen for App					🟩										
Shuffle Play Music on App						🟩									
SPRING BREAK															
Integration															
Integrate Step detection with Phone App										🟥	🟥	🟥			
Gather Running Data for Warping										🟪	🟪				
Integrate Warping with Phone Music												■	■	■	
Slack 1												🟨			
Miscellaneous & Testing															
Benchmark Testing													🟨		
Compose/Rehearse Final Presentation													🟨		
Slack 2 (Integration?)													🟨	🟨	
Demo Day															🟨

Fig. 12. Gantt Chart