AUTONOMOUS BASEBALL PITCHING MACHINE

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering, Electrical and Computer Engineering

Submitted by
Rajiv Kommareddy and Asad Marghoob
MEng Field Advisor: Joseph Skovira
Degree Date: December 2019

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: Autonomous Baseball Pitching Machine

Authors: Rajiv Kommareddy and Asad Marghoob

Abstract: The focus of this project is to add on an attachment to a baseball pitching machine, which could detect a target and make the necessary adjustments to the launch angle so that a pitched ball would hit that target. This can be achieved by using a Pi camera and aligning it at a fixed position. The camera will get a "field of view" from the perspective of the robot pitcher and will be focused approximately 60 feet away. Then, a target made of ripstop nylon will be placed 60 feet away from the camera, and its picture will be taken.

A simple algorithm will run to determine the pixel coordinates for the location of the target. This information alone can tell us the position in space that the target is relative to the camera. From there, the machine can be calibrated to be fed the inputted desired distance and launch a ball to it (based on data collection).

The target will also be equipped with an array of microphones connected to an Arduino, so that upon impact we can determine where the ball struck the target by triangulating the distance based off the difference in the timings that the sound impulse was detected. Once again via Bluetooth communication, the Arduino can relay back to the launcher where the ball hit the target, and if the target was not hit close enough to the desired position, then the launcher can make slight adjustments to the machine settings and try again, and the cycle continues until the ball hits the target as accurately as possible.

Once the machine has self-calibrated itself to the target, the user can use the Android controller to save the position and continue receiving that exact pitch if they'd like. They can also send the machine back into calibration mode and move the target.

This project should serve to improve the pre-existing machines on the market while adding marginally priced components (compared to the price of the machines themselves) to automate the cumbersome task of calibration to hitting a target and allow batters a more efficient way to practice their technique for very specific pitches, by turning a machine which requires two users into one that only requires one user controlling the whole system through their Android device running an app that we developed.

Individual Responsibilities:

Rajiv Kommareddy:
Responsible for all initial development during Summer 2019 and continued work through Fall 2019. This includes all image processing software development, integration of BlueDot, coordination with Cornell Baseball, physical building of target and support beams, calibration of the Home Plate Machine, code implementation and testing of sound triangulation software in MATLAB and Arduino. Also involved in hardware microphone circuit design and solving of sound triangulation systems of equations.

Asad:
Joined project this semester (Fall 2019) in September. Responsible for all Android application storyboard and backend implementation on virtual device (testing) and Android device (launching), coordinating with other Android app teams, and enabling bluetooth communication between “client-side” Android device and “server-side” Raspberry Pi. Also worked on hardware circuitry for microphones and and solving triangulation system of equations including consultation with professors.

Executive Summary:

When using a standard pitching machine, tedious trial-and-error of aiming and firing (calling for an additional individual tuning the machine) is necessary to achieve the desired pitch location. We set out to address this inconvenience by fully automating the experience via a user-controlled application, a pi-enabled camera, and a microphone-based feedback system.

Ultimately, all of the aforementioned components in our system were fully operational on their own, with issues primarily arising when trying to attain a smooth integration of two or all of them. The image processing algorithm runs reliably, taking note of a calibration color and showcasing accurate target detection capabilities when the target is stationed 60 feet away from the Pi camera.

Our Pi camera module allows us to take pictures and extract information from the pixel data to determine where a target set 60 feet away is in space. It achieves this task by first asking the user to provide a calibration image, where the user moves the target in order for it to be in the center of the image, and with this knowledge the program can observe the image and determine the pixel color in RGB representation that the target is. Once this is done, and the target has been moved to a desired location, the program can take a new picture and determine the pixel range where the target is located, and map this pixel coordinate information into a real distance. This module relies heavily on preprocessing of data which must be done any time the camera orientation or the location of the entire system changes. This is due to the fact that the mapping relies on specific pixel numbers to correspond to specific distances, so any fluctuation in the orientation of the camera will have drastic effects on where a target's detected pixel locations would be and therefore its mapping, despite it not having physically moved. When the module has been calibrated, the image processing works reliably and can output the target distances to be within 2 cm of their actual values. A possible addition to improve this module could be to write a script which takes several pictures of the target when the user moves it to generate the linear pixel-distance mapping functions on its own.

Our microphone circuit and triangulation provides us with a method for triangulating the impact of the ball with the target, and clearly provides the correct feedback for getting closer to pitch selected by the user. This was accomplished by using a configuration of three microphones that passed their respective delays to an Arduino. The Arduino uses expressions derived from the geometry of the target's dimensions and microphone delay readings towards locating the ball's impact in the plane of the target. Occasionally, the seem to be erratic, which may be due to the microphones not being anchored securely enough or ambient noise interfering with the delay readings. The Arduino at this stage communicates with the Raspberry Pi via serial communication, we were not able to figure out how to make multiple ports available for concurrent Android and Arduino communication. This would make for a worthy pursuit in a next iteration of the project.

The Android app showcases highly reliable communication with the Raspberry Pi via bluetooth. When connected to a monitor, our main Pi-based python script could run and receive all pertinent bluetooth requests and respond accordingly. We were able to remotely take calibration images, target detection images, select a desired pitch location (or change the desired pitch location), and trigger a steady stream of pitches once the pitch has been calibrated from Arduino feedback. However, this file did not run upon boot of the Pi, nor did it run on the TFT screen when not connected to a monitor. Due to this, we were not able to test the Android communication with this script remotely; we could only provide a viable demo when connected to a monitor. One issue with our bluetooth protocol is the need for every communication to be client-server based, each connection involves the Android making one request, and the Pi

responding. It would make for a more flexible protocol if we could have some instances in which only one party sends data and the other waits, or the Pi makes a request and the Android responds.

Additionally, the baseball team administration discarded the pitching machine we were planning to demo on. As a result, we needed to recalibrate a “home plate” in the halls of Philips and physically hit our target to emulate impacts from pitched baseballs. Thankfully, this was enough to show that our system was indeed functional.

Problem Description:

There are currently many different pitching machines on the market today, with drastically varying price ranges from below $100 up to $10,000. The variance stems from the difference in quality and features provided by the more expensive machines, such as improved precision or ability to throw pitches with different types of spin. Some devices even offer controller attachments which can be used to help the system be used by a single user.

However, one thing is common among all of the machines available for sale to the public: aiming the pitches rely on human adjustment via trial and error. Some machines have modes to continue pitching in the same position with good precision once initially calibrated by the user, but the downfall comes from when the user would like to hit a target which changes often, because the process of trial and error must be repeated each time. Automation would be the logical step in order to eliminate the need to manually calibrate the machine to hit a particular point.

System Requirements

- A remote application (i.e. Android) shall be able to interface with Pi-based pitching machine in order to take images and select pitch parameters
- Raspberry Pi camera shall take images of target for (1) target color calibration and (2) image processing to detect target in space.
- Three microphones connected to an Arduino shall accurately triangulate the location of ball impact on target and feedback to Raspberry Pi.
- Raspberry Pi will output updated parameters for subsequent pitches; Arduino will alert the machine when a satisfactory pitch has been received.

Overview of Proposed Solution(s):

The solution to this problem would be to add on an attachment which could detect a target and make the necessary adjustments to the launch angle such that a pitched ball would hit that target. This can be achieved by using a Pi camera and aligning it at a fixed position. The camera will get a “field of view” from the perspective of the robot pitcher and will be focused approximately 60 feet away. Then, a target made of ripstop nylon can be placed in front of the camera, and its picture will be taken via a Bluetooth controller as the shutter. Since the target will be in the center of the frame (it will take up most of the frame) the RGB color values can be extracted from the calibration picture.

Later, when the system is to be used, the target will be placed 60 feet away from the camera, and its picture will be taken. Using the RGB color extracted from before, the new image will transform the image such that all pixels that are not the desired color will become black, and all of the pixels that do match will become white. A simple algorithm will run to determine the pixel coordinate that the location of the target. This information alone can tell us the position in space that the target is relative to the camera, assuming it is 60 feet away. (A potential addition can be to use the size of the detected target to provide insight into how far depth-wise the target is away, but for the intended use this is not as important). From there, depending on the pre-built machine being used, it can be calibrated to be fed the inputted desired distance and launch a ball to it (based on data collection).

In the current prototype, we are using a now discontinued machine called the Home Plate Machine made by SportsTutor. The current interface for the machine provides flexibility for different heights, horizontal distances, and types of pitches. However, these are discrete values and therefore we are limited to a certain number of possible positions. Thankfully, there are many of them, and a combination of using different types of pitches at different position settings could provide a “denser” range of possible positions to choose from. Since it is known where we want to launch the ball to, we can deduce which combination of settings should theoretically launch a ball closest to the desired position. Ideally, this changing of settings on the machine and firing mechanism could be controlled by a remote. However, the protocols associated by the serial port for the Home Plate Machine are proprietary and they are unfortunately unwilling to release the information.

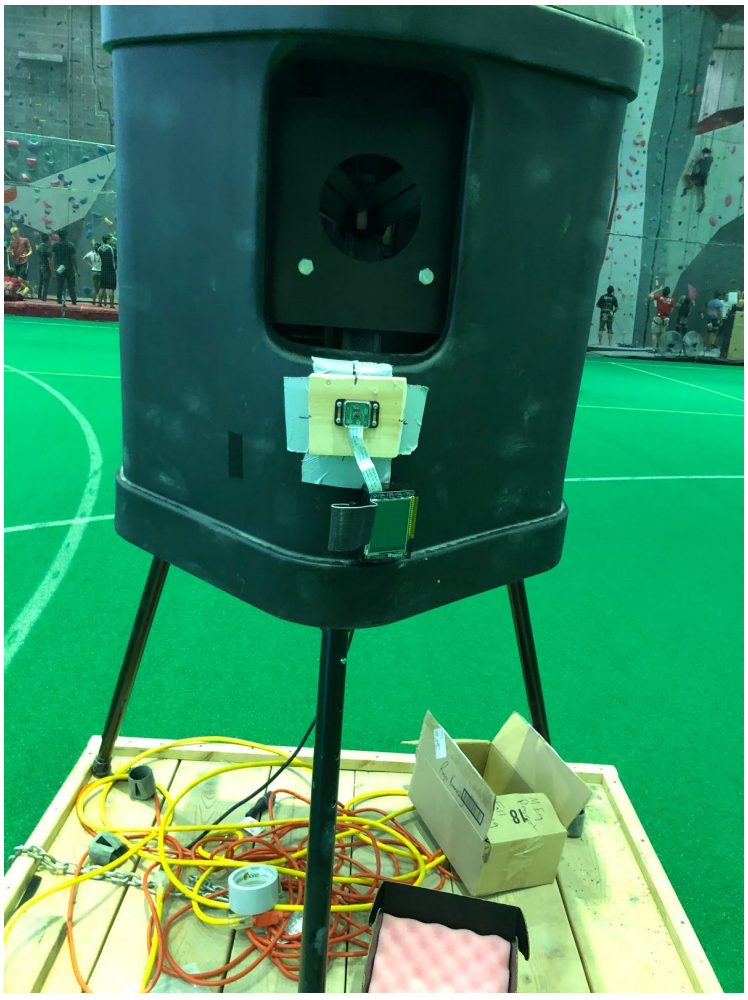

Figure 1: Home Plate Machine in Bartels Hall with Pi Camera Attached

The target will also be equipped with an array of microphones connected to an Arduino, so that upon impact we can determine where the ball struck the target by triangulating the distance based off of the difference in the timings that the sound impulse was detected. Once again via Bluetooth communication, the Arduino can relay back to the launcher where the ball hit the target, and if the target was not hit close

enough to the desired position, then the launcher can make slight adjustments to the machine settings and try again, and the cycle continues until the ball hits the target as accurately as possible.

Ball placement is not the only important factor which determines a good pitch either since different types of pitches (e.g. curveballs, splitters, knuckleballs) take different paths to get to the target. For this reason, another extension to the device would be for the camera to continually take pictures once the ball is launched, and when the microphone circuit detects the impulse it will stop. The pictures can be analyzed to tell if the ball traveled the way it should have as well, but this extension is only important with the more expensive and sophisticated pitching machines with pitch-varying functionality.

Once the machine has self-calibrated itself to the target, the user can use the Android controller to save the position and continue receiving that exact pitch if they'd like. They can also send the machine back into calibration mode and move the target.

This project should serve to improve the pre-existing machines on the market while adding marginally priced components (compared to the price of the machines themselves) to automate the cumbersome task of calibration to hitting a target and allow batters a more efficient way to practice their technique for very specific pitches, by turning a machine which requires two users into one that only requires one user controlling the whole system through their Android device.

There were some potential extensions that were discussed but not implemented in this current prototype, due to their complex nature, but would be worthwhile during future development of this project. The first possible extension would be to improve the image processing algorithm to rapidly continually take pictures after a ball is fired from the machine, and generate ball tracking data in the air. This could add another level of tunability and choice for types of pitches for a user, because not all balls which hit a target in the same spot are treated equally, as pitches with more complicated trajectories are more difficult to hit and therefore very attractive choices when using a system designed for practice use, like this one. This would require much more data collection for the different types of pitches and speeds, as well as some code optimization for the camera to be able to work reliably will a real-time constraint.

Another discussed expansion would also involve improving the image processing. As the current prototype is set up, the user must use an Android app to communicate to the machine which type of pitch it would like, and when. However, when the pitch is chosen, the user must step back and wait for the calibration using the microphones, press another button on the app, and hastily put away their device and wait for the pitch. A suggested solution to this issue would be to create an image processing algorithm which could detect when a person has stepped into the batters' box after the calibration stage and only pitch when the user is in position for a pitch to be fired their way. This would likely involve an ML model to be trained to determine when a person is in an appropriate spot.

Implementation:

This project, as described above, was broken into three main independently developed modules which had to work together to accomplish our proposed solutions. These three modules are the image-processing

software, the microphone circuit and software, and the Android App interface. They all had their own development timeline to develop mostly proprietary solutions for their prescribed tasks, and each had challenges associated with them, discussed individually below.

Image Processing:

This was the first step to getting the project underway, and the development process for this module began in June 2019 and worked on alone by group member Rajiv Kommareddy.

The first step to getting this portion of the project working was to first attach a Pi camera to our Raspberry Pi controller (with an attached TFT). Next, I downloaded the appropriate library necessary for capturing and processing the images, OpenCV. Once I got familiar with the command line arguments necessary to take an image, namely raspistill, I was successfully able to take pictures using my Pi camera using command-line inputs using a keyboard connected to the Pi in the lab. At the same time, I knew that I would need to control the "shutter" for our camera wirelessly, and was directed to look through former student projects under project advisor Joseph Skovira. I came across another Bluetooth controlled project (COOPY, created by Henri Clarke and Michael Rivera) which used an Android phone as the main controller, and I looked into how it was implemented from their report, and found out that they used a free, downloadable application called BlueDot.This application displays a single blue button in the center of the device's screen, and has the capability of detecting several different strokes (i.e. single tap, double tap, swipe, top portion of button pressed, etc.). I found this application useful to implement my shutter button by writing a simple python code called **Camera.py**, which starts with the Pi searching for a connected Bluetooth device, in our case a Kindle Fire, and waiting for that device to open the BlueDot application. Once BlueDot was ready on the device, the code in Camera.py was configured so that any time a user presses anywhere in the dot on the Kindle, the callback function for this button triggers an OS command-line call to raspistill as described earlier, and takes a picture and labels the image and places it in the same directory as the rest of the python files, with the name img_<number of times the button was pressed during the current session>.png.
In order to get all of this running outside of the lab, where I had the ability to use a keyboard interrupt to stop the program and type in the command line to turn off the Pi, I would need to include button functionality. This happened in the form of TFT button callback routines, specifically for buttons 17 and 27. Button 17 was configured such that its callback routine would trigger an OS call, similar to the way BlueDot works, and quits the program using a simple quit() command. Similarly, in the more common case where I would want to shut off the entire Pi, (for example, when I'm done collecting pictures and want to save all of them and shut off the Pi to transport it back to the lab without a keyboard) the callback routine for Button 27 makes an OS call for the command "sudo shutdown -h now".

Once I set up my wireless camera interface, the next step was finding out how to import, manipulate, and process the images to eventually accomplish the goal of detecting the location of an object in space from a single image being taken. The first "meaningful" test images I took were of a rectangular piece of pink foam I found in the lab, and I wanted to see if I could determine the dimensions and location of the rectangular foam in the picture using functions available in the OpenCV library. The picture can be seen below, for reference:

Figure 2: Initial Test Image for Thresholding

I consulted an online resource to figure out the RGB color associated with the pink color of the foam, and used a function cv2.inRange() which defined thresholds slightly above and below the R,G, and B values for pink and set the pixels within the range to white and all the other pixels to black. There were many different erosion and dilation functions I played around with on my thresholded binary image in an attempt to create a smooth enough looking rectangle for the OpenCV function findContours() to be able to detect. The best looking smoothed rectangle I could produce after countless iterations of trial and error with dilation and erosion resulted in the following image:

Figure 3: Thresholded Test Image

Unfortunately, even this shape was not smooth enough to be detected as a rectangle by the pre-built functions in OpenCV, and I realized that if it would be this difficult to get working in a specific case where I am analyzing the image myself and performing several operations on it with minimal success, there would be no way to get detections in a general case where I have to allow the system to do all the image processing entirely itself. This was a big turning point in the project, where I decided to abandon trying to use findContours() to detect my rectangle, and had my sights set on writing my own proprietary algorithm.

I recognized that the entire goal of my image processing would be to output the pixel location of the center of the rectangle, because with that information and a separate function (developed with further data collection, discussed in section labeled **Home Plate Machine Calibration**), I would be able to find the real world coordinates in space where the object is. With the knowledge that all I would need to do is find

the center pixel, I began work on my next python function, **BinarySearch.py**. I realized that as long as the thresholded image produced a shape that generally resembles a rectangle, I could take the average position of the left, right, top, and bottom edges of the white object in the image and that would return a reasonably accurate estimate for the center pixel position of the object. This was achieved, as the name of the file suggests, using a binary search algorithm. For example, when trying to detect the x pixel position for the left edge of the "rectangle," one could look at a point in the rectangle, then look for one out of the rectangle, and slowly keep testing positions until you find a spot where the distance between an x position where you find a black pixel and a white pixel are reasonably close, indicating you found the edge. This process repeats in a similar fashion to locate the top, right, and bottom edges as well.

However, to get these binary search functions to work, it assumes we know the location of at least one white pixel to begin with. I came up with an idea to find a white pixel by first starting by testing the center pixel of the image, and if the pixel is black, test a pixel slightly to the right by a predefined increment, and then move upward, and then left, and so on in a spiral fashion until a white pixel is found. To do this, I wanted to make sure I wouldn't "overshoot" the entire cluster of white object pixels, so after taking a picture of an object 60 feet away and observing the size of the cluster of pixels, I set my increment size for successive tests to be slightly smaller than the size of the width and height of the cluster.

Once I had these two functions properly, I was able to take a picture of an object of specific, hard-coded color, and determine the pixel coordinates of the center of it. The next step was to translate this pixel coordinate into a real distance measurement in units of meters. To do this, I had to build a stand which would keep the camera and Pi in a fixed height position from the ground and align it the same way it so that it would point straight down the long hallway on the second floor of Phillips Hall. The floor tiles were especially useful for alignment. Next, I put the target in various known locations at a constant distance of 60 ft away and took pictures and processed them to get pixel coordinates. With enough data points, I was able to create a linear mapping using Microsoft Excel, and therefore by adding a simple conversion function, my program was then able to take a picture of an object and output its x distance (relative to the camera) and y distance (relative to the floor) in meters.

Home Plate Machine Calibration:

All the work done in Phillips Hall to devise the image processing software was done with the goal of eventually attaching the camera to the baseball pitching machine located in Bartels Hall. This transition was not nearly as smooth as I had anticipated.

On the first day of attempting to get the system running, I knew I would have to recalibrate the function which would translate pixel position to real world distance, because the offset height would be slightly different, and the distance from the target to camera could be slightly different than the 60ft measured in Phillips Hall. What I did not anticipate was that I would not be able to detect my target object at all. Therefore, all I could do on the first day was take plenty of pictures of the target in different positions, mark up where the camera would be attached via velcro on the machine, and mark the position in the room the machine would be located for use.

When I returned to the lab to analyze the images to determine the new pixel-distance mapping function, I also wanted to figure out why I was not getting detections at the time. I realized that the lighting in Bartels Hall is significantly different than in Phillips Hall, giving all images a yellow tint. I knew this would be an issue moving forward and did some research on color representations and found out that for use cases like this, it is best to convert RGB colors into HSV colors when doing thresholding. This is because instead of using the values of three colors in certain proportions like RGB, HSV uses the hue, saturation, and value (brightness) to determine a color. This is useful in our thresholding because generally when moving to different locations, an object's perceived hue and saturation do not vary as much, but the value parameter is what changes significantly. The advantage of this is that we can set the range on the value to be quite large to not risk missing the detection of the object, but also limit the possibility of a false detection because the hue and saturation numbers are set to a tight range of specific colors.

Despite this change, I wanted to guarantee that my image processing software would provide detection regardless of the color of the target and wanted to make it robust for any lighting condition. This led me to add an additional stage in my program which would be responsible for setting the color of the target to be detected for the rest of the duration of the program. Upon starting the program, the user is to bring the target close to the camera such that the center pixel of an image taken will belong to the desired detection color. This color is saved, and any subsequent image taken will use that color to make detections and translate that to a distance. In case the calibration image was taken poorly for some reason (e.g. the lighting changed, the target wasn't in the center of the camera frame, a new target is used, etc.) I added functionality to another button. Button 22's callback routine clears the color calibration data and makes it so that the next time the BlueDot button is pressed, the image taken will be used as the color calibration image, and the system continues running normally.

The final step to developing the system to fire a ball at the target once it is detected is understanding how the machine behaves. Upon looking at the machine's interface:

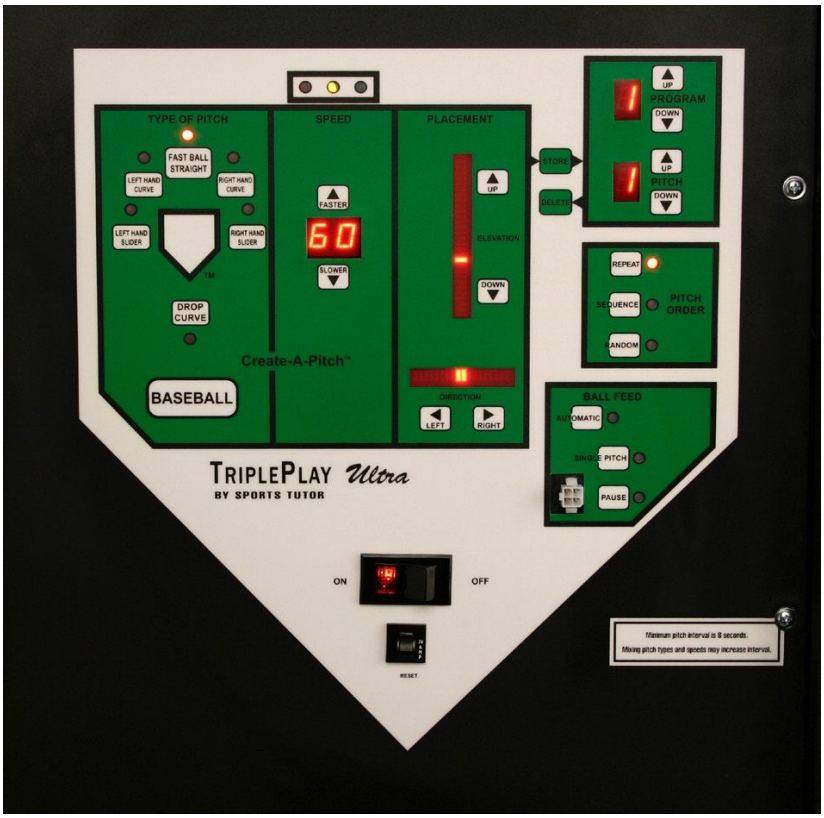

Figure 4: Home Plate Machine Control Interface

I saw that there were plenty of options for types of pitches, but the choices were discrete and therefore for a certain pitch type I was limited to about 90 different pitch positions (13 vertical options and 7 horizontal options). I chose a ball speed and set the machine to fire fastballs, and went through each setting on the machine to determine where in space the balls landed 60 ft away relative to the machine, by observing the impressions left in the backing screen where the target would be located. Once I could get a reasonable distance-machine setting mapping, I was able to set up a basic 1-Nearest Neighbor machine learning model to output the baseball machine settings given a desired position. In doing so, the entire system could automate detecting the target and cleanly outputting what to set the machine to in order to fire a ball at it. All of the helper functions developed in BinarySearch.py were moved into a separate file called **BS.py** and imported for use in our main python file which would start upon bootup of the Raspberry Pi, **Translation.py.**

Unfortunately we were unable to close the loop and fully automate the system due to a legal hurdle. If you observe the interface image for the Home Plate Machine, you can see in the bottom right corner that there is a white serial port. This was designed for use with a controller that the manufacturer, SportsTutor, also sells. More importantly, this tells that a serial protocol must exist which could be used to set the machine settings and fire a pitch. However, when we contacted the manufacturer, they refused to reveal the protocol for legal reasons. Despite having the coach reach out to the company and reinforcing the notion that this is an academic project with no intention of commercial gain, we were still unsuccessful. Although the closed loop system cannot come to fruition for this particular machine, the proof of concept

has been thoroughly demonstrated. The full image processing system can be seen in the following video: https://tinyurl.com/uke5bjj

Microphone Circuit:

For accurate triangulation of ball impact upon the target, we used a microphone circuit that leveraged distinct time delays to pinpoint the location. We used three Grove sound sensors from Seeed Studio. Each had a signal output, Vin, and GND. We connected each signal output to an analog input to our Arduino Uno. We also used an additional solderless breadboard to define both a common ground and Vin line for all three microphones and the Arduino. We used a 3.3V signal from the Arduino to power these components. Seeing as this circuitry was relatively lightweight, we were able to draw power from the Arduino. We verified that the current drawn through the Arduino from the microphones was less than an Arduino's maximum recommended current.

Once the target is positioned 60 feet away, the machine runs the processing algorithm and fires a first calibration shot based off of the user-inputted target. Once this shot is fired off, the arduino-based system mounted on the target is responsible for providing accurate feedback regarding the location of the pitched ball's impact compared to the actual desired position. This is an iterative process; the machine will continue to pitch until the ball's impact with the target is within an acceptable margin from the inputted location. The feedback readings come from a configuration of the three microphones, each mounted at the bottom left, bottom right, and top left corners of the target, respectively. Our approach was to implement a triangulation using the three microphones' delay readings from the impact detection. Each microphone will have its own time delay reading, which can be readily converted to a distance measurement when accounting for the speed of sound. The geometry and associated system of equations described by these three distances allows us to exactly locate the location of the ball's impact.

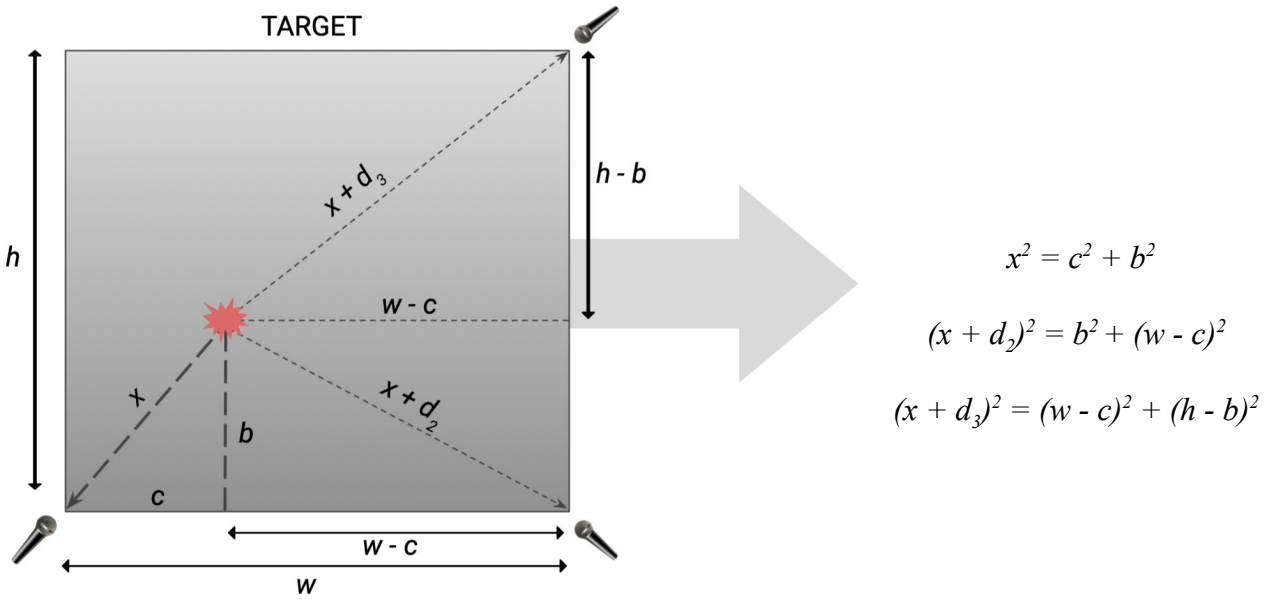

Figure 5: Geometry and System of Equations from Microphone Delays

Although at first glance the equations seemed feasible enough to solve for our unknowns x, b, and c, we faced a lot of difficulty in doing so. After a few hours worth of attempts, we consulted numerous professors in ECE and the mathematics department for assistance. Before long, Professor Ravi Ramakrishna, chairman of mathematics at Cornell, responded with a sure-fire method getting both b and c in terms of one another. We explore this more in our description of implementing the code in the Arduino sketch.

Note that the above image pertains to the scenario in which the ball's impact occurs closest to the bottom-left microphone. If the ball was to make contact closer to one of the other microphones, the geometry would change accordingly.
To make this more clear, see the following sets of equations that result when the ball makes contact closest to the bottom-right microphone (with delays d_1 and d_3 for microphones 1 and 3, respectively, and x being the delay from microphone 2):

$$x^2 = c^2 + b^2$$

$$(x + d_1)^2 = (w-c)^2 + b^2$$

$$(x + d_3)^2 = c^2 + (h - b)^2$$

Now, note the system of the equations derived from the geometry of the top-right microphone being the closest to the impact location (with delays d_1 and d_2 for microphones 1 and 2, respectively, and x being the delay from microphone 3):

$$x^2 = c^2 + b^2$$

$$(x + d_1)^2 = (w-c)^2 + (h - b)^2$$

$$(x + d_2)^2 = c^2 + (h - b)^2$$

In the case of the microphone 2 hearing the impulse first, $(w-c)$ represents the x-axis coordinate and b represents the y-axis coordinate of the contact location on the target. In the case of microphone 2 hearing the impulse first, $(w-c)$ represents the x-axis coordinate and $(h - b)$ represents the y-axis coordinate. Note: when we refer to "x-axis" in the previous two sentences, it is not to be confused with the x value that is shown in the systems of equations. The x value shown in the equations is simply the absolute distance from the contact location and the closest microphone.

Once we solved the equations, Rajiv was in charge of implementing them in code. Since the equations were solved in the form where it was two equations where the variables were in terms of each other (i.e. c(b) and b(c)), the easiest way to find the solution was through plotting the functions. However, this is not an option in our system since we have to process the values and output the x and y coordinates for where the ball landed, relative to the bottom-left corner of the target, without inspecting the plots.

The way to solve this issue was to iterate through possible values of c, and calculate c(b(c)) and detect where the two functions are reasonably close to each other. Once this c-value is determined, we can easily calculate b and then convert these values into x-y coordinates, if necessary.

We also needed to adjust some code found online to get it to work with our sound triangulation scheme. Using this code as a base, we removed the extraneous computations and set it up so that we could detect the sound delay for the second and third microphone detections after the first microphone detection is used as a trigger. We take these time delay values and converted them into distance delays usable in our equations, and then compute the x-y coordinates using the proper equations, determined by which microphone was triggered first.

Unfortunately, due to Bluetooth constraints, we had to limit the Bluetooth connections to go from the Pi to our application (discussed in the next section). This forced us to use a serial communication protocol in order for the Pi to get information from the Arduino. To achieve this, we provided functionality to our final TFT button, 23, in our program **Serial_Translation.py**. This program is very similar to Translation.py which was discussed earlier, but now when the target is detected and its coordinates are printed out, we were able to press Button 23, whose callback routine would establish a serial connection to a connected Arduino (via USB) and continually read the Serial.print statements from it. Eventually, the need for this button should be eliminated, as this forces the program into a While loop which can only be broken out of when a pitch is satisfactorily registered. This poses a problem when a pitch never gets fired, so a timeout condition should be implemented. Additionally, if the serial communications can be replaced with Bluetooth communications, there would be no need for constant serial reads and therefore could help improve overall program flow. The statements from the current serial communication system come in the form of (x,y) coordinates, which are then parsed within the while loop, and configured to provide feedback on the Pi TFT screen to aid the user in aiming a pitch low and outside.

Android Program Flow:

The Android application saw a couple of iterations throughout this semester due to our evolving design needs and constraints across the semester. At first, we created a storyboard that displayed each “state” of the process, with one screen dedicated to each of these states. For example, we programmed a start screen to enable bluetooth, the next screen to take the first picture, the next to take the target detection picture, another to select the pitch, and finally one more to start pitching upon a successful calibration. This approach reflects our vision for the flow of the program well, and provides a comprehensible and convenient interface for the user. However, this was all done before we started to integrate the bluetooth communication. When we started to integrate bluetooth with the rest of the system, we realized there was a lot of overhead in setting up and establishing for every Android Activity class, even when the class was only dedicated to a single button press. To this end, we began consolidating our Java code and Android storyboard such that a Bluetooth communication setup would not have to be re-instantiated at every small state change of the system.

When the Android application has launched, the Pi script to run the baseball pitching machine is already running and waiting for a connection. We trigger MainActivity.java, and start with a screen that displays two buttons: one for calibration and one for image detection. We used TextView classes for visualizing text and Button class for any and all buttons. Upon hitting the calibrate button, the Android device sets up a connection with the Pi and tells it to take a picture for color calibration of the target. Once this completes, the Pi sends a confirmation back to the device to be displayed on the device screen. The same logic holds for the target detection image, i.e. a touch of the button sends a request to the Pi to take a picture of the target. Upon receiving the confirmation and displaying the text, our program displays a button to proceed to the pitch select screen.

Our pitch select screen allows the user to pick from four possible pitch locations. We used a Floating Action Button class for each of the four possible locations on the target. Once the user selects one of the buttons, the corresponding pitch location (ex: “High and Outside”) is sent to the Pi. The Pi will respond with a confirmation that the information has been received, and said confirmation is displayed on the Android pitch select screen. At this point, the Pi script breaks out of its bluetooth connection loop and begins the feedback calibration process with the Arduino.

The TextView and Button instances and their pertinent parameters (size, location, color, etc) are defined in our dedicated frontend XML file content_main.xml. This file contains the code for all of the objects, and is included in main_activity.xml, which is the file that launches the frontend.

Android-Pi Bluetooth Protocol:
Although the bluetooth protocol and Android application are inextricably linked, we think it would be worthwhile to talk about the bluetooth separately since it comes with its own set of considerations in terms of setup and the actual communication. In covering bluetooth, we will also detail the script (Final2.py) that acted as the receive- or server-side that runs on the Pi to handle requests from its Android client. We used a 3rd generation Kindle Fire, since this was the device that was used during the Summer to run BlueDot. Since our files in Android Studio are compatible with any device that supports an API level between 15 and 29, this code can be readily flashed to any Android phone (ex: Pixel) as well.

First and foremost, on the Pi side, we downloaded PyBluez with a sudo apt-get command. This is a module that allows us to leverage the built-in bluetooth resources of the Pi-3 such as sockets and binding to ports. In our script, we took the necessary steps to setup our program. Of course, included in our import statements is the bluetooth module, which comes from the PyBluez installation. Then, we define an RFCOMM server socket and bind to a port. In order for two devices to have a common reference to the bluetooth service being used, we used a unique identifier string known as a UUID that is shared by both the Android and Pi. Once the Pi “advertises” its service with this UUID and server socket, it enters an infinite loop that waits for a connection request from another device. This loop first prints a statement conveying that it is waiting for a connection, and will only exit upon a keyboard interrupt. Now that it is waiting for a connection, we can shift our scope to the context of the Android file to see how the bluetooth is carried out on the other end to make use of this advertised availability.

In our MainActivity.java file, we needed to import three bluetooth dependencies, namely BluetoothAdaptor, BluetoothDevice, and BluetoothSocket. The program first enables bluetooth with BluetoothAdaptor, and then iterates through any paired devices to find the device of interest. This can be identified with the MAC address, or simply the device name (in our case, rrk64). When any of the buttons we described above are triggered, we programmed the file to execute the bluetooth communication thread that connects with the pending port on the Pi side. This script is called workerThread, and is always called when a button is hit that requires some sort of bluetooth communication. The thread first sends a message to the Pi. This entails creating a socket using the same UUID defined in the python script, and connecting to the port being broadcasted by the Pi. It sends the message by writing the information to the output stream before reading the data being sent back from the Pi. A unique integer value is assigned to each type of button click on the storyboard, so that the triggered thread accordingly indexes into the appropriate code to handle the received data. For example, when the user hits the button to calibrate image, the integer value will be set to 0. Therefore, in the thread (upon receiving confirmation from the Pi that the picture was taken), we index into the conditional block that updates the text below the button to read “calibration image taken”.

In the python file, we have a series of conditional statements listed after the “waiting for connection” print statement. Essentially, the program will read the string received from the Android, and take appropriate action. For example, if the string reads “color calibration”, the script will call the function ColorCalibrate(). A keyboard interrupt is used to break out of this infinite while loop after a pitch location has been selected, so that the serial calibration between the Pi and Android can commence. We also provide the user with the option of re-calibrating to another pitch location (after selecting one already) should he or she change their mind.

Conclusions and Comparison to Expectations:

Unfortunately, several reasons contributed to why our project could not accomplish the final goal of being a “closed-loop” system. Aside from the lack of access to the serial port on the physical machine, as discussed earlier, the first major issue we had came approximately two weeks before our demo date. It was brought to Rajiv’s attention that the Home Plate Machine had been discarded by a Cornell Baseball staff member. This posed serious issues, aside from the obvious one that we no longer had a firing mechanism. The camera system was calibrated to work in the Ramin Room of Bartels Hall (where the machine was stored) because the zero point for aligning the physical machine was based on firing a ball with the machine set to (0,0) and moving the machine so that the machine would hit a particular spot on the cloth netting. Since the machine itself was marked and had specific velcro locations to place the Pi with the camera on it, the calibration for the image processing was also gone. This forced us to improvise, and develop a new demo sequence which could showcase the functionality of the different components of our project.

Although the firing mechanism was gone, we were able to retake data for image processing in the hallway on the second floor of Phillips Hall to generate the mapping from pixel locations to physical distance measurements, the same way it was initially done in the early stages of developing the image processing

algorithm. Without a machine to actually have settings mapped to these distances, the way we designed our app fundamentally changed.

We designed around an ideal machine which would have infinitely tunable inputs such that any given location of a desired pitch within the camera's field of view. From here, we could provide the user with four choices for pitches in the four corners of the target (high and outside, high and inside, low and outside, and low and inside) and assume that all four of these pitches would be possible, despite where the target was detected in the camera's field of view.

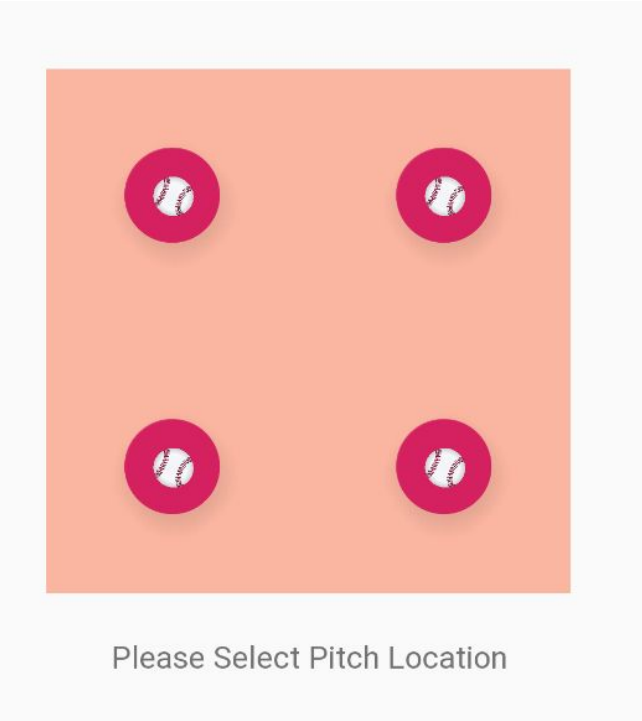

Figure 6: Pitch Selection Screen UI

Another difference in our expected results versus our demo came from issues which arose from Bluetooth connectivity. It became increasingly difficult to write code for our Pi to sustain a reliable Bluetooth connection with both our Android app and also the Arduino. When faced with the choice, we opted to prioritize the Bluetooth link to the app and relegated the link from Pi to Arduino to be a serial connection using the Arduino-USB programming cable.

Since in our demo we could physically hit the targets ourselves (without a machine), the image processing had little effect (although it did work quite well), and we could verify the correctness of the sound triangulation module by heeding the adjustment instructions printed to the Pi TFT screen. In the next prototype, integrated with a real machine, the app would need to be updated so that the selection of pitches available to the user after the image processing occurred is directly affected by where in space the target was detected.

Up until the day of our demo, we had been running and testing our pseudo-closed-loop system while still in the lab room, and therefore had the Pi connected to an external monitor with print statements all going to a terminal window on the screen rather than the TFT. When it became time to run all the components working together in the hallway, we updated our .bashrc script to run our final demo code. Once we plugged in the Pi to our power bank, we expected our program Final2.py to begin executing the way it had been during testing in the lab. To our surprise, the code did not execute and it appeared that the output on the Pi TFT was frozen. This was very alarming, because none of our bailout buttons were functioning and we therefore had to edit our bash script externally on Professor Skovira's own Raspberry Pi.

The root of this issue is still yet to be determined, but we did notice that when we edited the bash script to have the code run in the background, the Android application is able to control the system normally and take pictures, but the print statements do not print to the TFT screen, as expected. In future development it would be strongly advised to see the effect of switching from an external monitor to a Pi TFT alone has on the execution of a bash script. We believe that there is a strong possibility that setting the environment variables as shown below at the beginning of our program file may fix this issue, but this is unconfirmed at this time:

```
#os.putenv('SDL_VIDEODRIVER', 'fbcon')
#os.putenv('SDL_FBDEV', '/dev/fb1')
#os.putenv('SDL_MOUSEDRV', 'TSLIB')
#os.putenv('SDL_MOUSEDEV', '/dev/input/touchscreen')
```

One final bug was pointed out by Professor Skovira during our demo. The code as we have it involves a While(True) loop when the Pi is reading serial communications from the Arduino to calibrate fired pitches, with the only exit condition being when the ball is pitched satisfactorily. This could pose issues if the system is unable to hit that location or the choice was made by accident with no desire to pitch. This could be easily solved by adding in a timeout breakout condition which could trigger after a certain amount of time with no sound detections, perhaps 60 seconds.

Summary of Future Development Recommendations:

- In-flight ball tracking
- Automated script for calibration of pixel-distance mapping
- Machine Learning human detection model to trigger when to fire balls
- Multiple Bluetooth port functionality to include Android-Arduino communication and Pi-Arduino communication
- Closed-loop functionality by setting environment variables
- Eliminate need for button 23 for serial communications
- App development for use on a phone rather than Kindle

Appendix:

User's Manual:

The first step to using the system as it is would be to recalibrate the mapping from pixel coordinates to real-world distances. Unfortunately, since there is no way to reproduce the exact height and orientation for the camera that we did, this is a necessary (one time) step any time the system is used in a new location. You will additionally need an Android device which has the BlueDot app on it, and it must be paired with your Raspberry Pi ahead of time. Once you have a reliable stand which will hold your Pi in the same fixed position and orientation for multiple uses (i.e. a piece of velcro to attach the Pi and camera in a rigid position, as shown in the picture below) you can run the Camera.py app. To do this without the use of a keyboard or monitor, it is advised to edit /home/pi/.bashrc and add the line "sudo python /home/pi/Project/Camera.py" to the end of it.

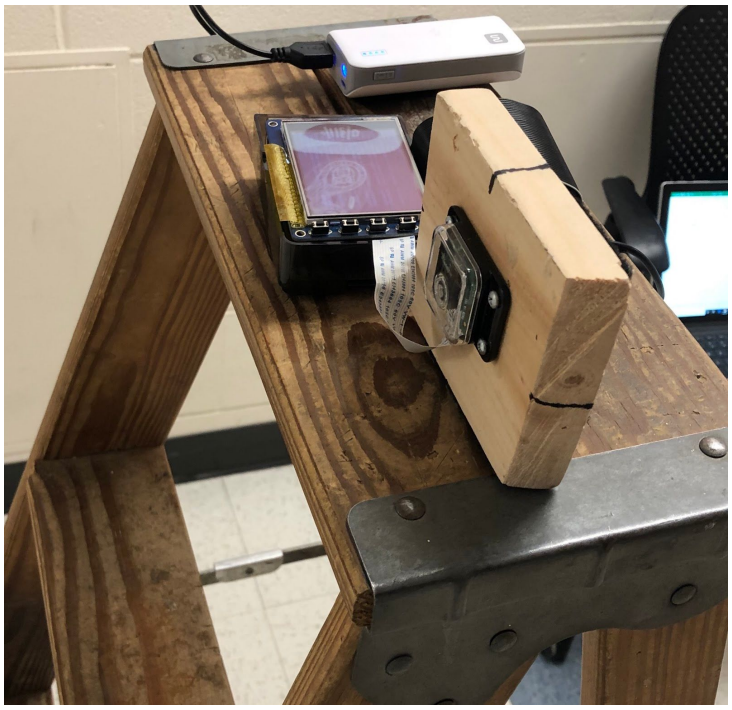

Figure 7: Rigid Position of Camera on Stand, Supported by Velcro

Next you can take the take various pictures of target from 60 feet away, while taking note of the physical x-y location of the center of the target in each picture. The easiest way to keep track of this is to use a distinct mark somewhere and define that to be your x = 0 [m] point, and measure any distances to the left of it to be a negative x value and anything to the right as a positive x value. You can set your y = 0 [m] point as the ground, and therefore all the data you take will have a positive y value.

Figure 8: Camera Setup in the Hallway with Target Positioned at 60ft

Once the images and data are taken, you can run them all through BinarySearch.py individually, by editing inside the file the name of the image file it is reading. The output of this file will give you the x-y coordinates of the detected target. If you do not get detections, you may have to change the threshold RGB color values being used, which can be figured out by uploading any of your pictures to one of many online resources to figure out the RGB colors of your target pixels. Once you process all of your images, you will have a mapping of several different x-y pixel coordinates to x-y real world distances. Plot these into your favorite linear regression tool to get a function which will translate pixel coordinates into distances. You should be left with two linear equations, one to map the x-values and one to map the y-values. Finally, open up Translation.py and change the functions for x_distance and y_distance according to your linear equations.

Following this step, the system should be ready for use as intended. You can start by running the Final2.py program (beware of running this using a .bashrc script, there is work to be done for this as described earlier). While running, open up the Android app which should be downloaded to your device using the files MainActivity.java, content_main.xml, and activity_main.xml. With the app running, you can take a color calibration image while your target is close to the camera (such that the middle pixel in the field of view of the camera is the color of the target). If you mess this step up, you have the option to press the top button again to recalibrate the color.

Next, you should move your target 60 feet away and press the bottom button, which will take another picture and return the real-world distance coordinates on the Pi TFT screen. Next, a button should pop up which you can click to get to the next screen which provides a GUI to allow you to select one of four possible pitches. Once you select a pitch location, you can then bring the Pi over to the target, which has an Arduino attached to it, as seen in the image below.

Figure 9: Arduino and Microphone Circuit Attached to Back of Target

Connect the USB cable from the Arduino to the Pi and press Button 23 on the TFT, which allows for serial communication. You may now send pitches at the target, which will provide feedback on the TFT to help you hit the desired position. Once this is achieved, you will get a printout indicating that you have hit the desired target, and you have the option on the app to go back and select another pitch, if you decide to continue. If not, you may disconnect the USB wire to stop the program.

Bill of Materials:

- Arduino Uno ($16) (https://tinyurl.com/u4m8dzn)
- Raspberry Pi 3 ($35) (https://tinyurl.com/w7eogyo)
- Raspberry Pi TFT ($35) (https://tinyurl.com/uynph2k)
- Pi camera ($9) (https://tinyurl.com/w9uwysz)
- Home Plate Machine (donated and discontinued, retailed at ~$10,000) (https://tinyurl.com/txpvbuo)
- Bluetooth LE Shield ($20) (https://tinyurl.com/wxqxnen)
- Microphones (3x at $5 each) (https://tinyurl.com/vfknchw)

References:

- COOPY - https://tinyurl.com/sjgrsjd
- Online Arduino Sound Triangulation Reference - https://tinyurl.com/syfpvzx
- David Vassallo's Blog - Android Linux / Raspberry Pi Bluetooth Communication

https://tinyurl.com/y6ulytrf

Code Appendix:

Please visit the GitHub repo to view the code files discussed in the report:
https://github.com/kvijar10/Autonomous-Baseball-Pitching-Machine

Python Files:

- Camera.py (used for taking pictures for later use in data collection):
- BinarySearch.py (used for testing image processing capabilities of the system):
- BS.py (used as a helper file to hold important helper function definitions, very similar to BinarySearch.py):
- Translation.py (the main file for the system running on the Raspberry Pi):
- Serial_Translation.py (extension to Translation.py with added Pi-Arduino serial communication functionality)
- Final2.py (file used in demo to demonstrate Android App functionality)

MATLAB Files:

- test_tri.m (used to test the implementation for sound triangulation when microphone 1 was triggered first)
- test_tri2.m (used to test the implementation for sound triangulation when microphone 1 was triggered first)
- test_tri3.m (used to test the implementation for sound triangulation when microphone 1 was triggered first)

Arduino Files:

- Three_Mic_Print.ino (code constantly running on Arduino for ball detection and sound triangulation):

Android Files:

- MainActivity.java (main code for controlling storyboard of Android file and running bluetooth threads)

- content_main.xml (code that includes all frontend objects included in storyboard such as buttons, text boxes, and shapes)

- activity_main.xml (code that launches the frontend and includes content_main.xml)