# Baseball Shagger

**Students:** Kai Paresa, Anthony Velasquez, Nick Walker
**Advisor:** Dr. Andrew Danowitz
**Department:** Cal Poly SLO Computer Engineering
**Quarter/Year:** Spring/2018

**Abstract**

The purpose of our project is to allow players to hit baseballs on a baseball field and not have to worry about picking them up. By combining our knowledge of software and hardware, we developed the first design of a robot that “shags” baseballs. Our endeavor was only partially successful. The device was tested on grass, turf, and concrete. The motors did not have enough torque to get moving on grass. The device faired better on turf where it could move, but was quite jerky as the motor drive needed to be high to start moving, but once it was moving, full motor drive was often too much. On concrete, the device could move smoothly but this surface was not indicative of the conditions specified by our use case. Our software uses an onboard camera to take pictures of the field and relay the position of the baseballs to our Raspberry Pi (which acts as the brains of the device). We turn the balls location in the photo into an angle and a direction, and power our motors accordingly. After arriving at the ball, we attempt to pick up the baseball using a custom built arm. While the arm and motors worked independently, there were issues whenever they were both running at the same time.

## 1. Introduction

Two of the authors of this paper are Club Baseball participants at Cal Poly. Due to the nature of being engineering majors, we were often unable to schedule times with other players on our team and thus, relied on each other. While we could play catch with each other and do defensive drills efficiently, that was not the case with hitting. The best method for correcting one's swing is to hit the ball, watch where and how the ball travels, and then make corrections based on that information. When there are only two participants, this process is incredibly inefficient as more than half of the time is spent picking up baseballs. In this project, we set out to design a solution to this problem.

## 2. Design

### Detailed Software Flow

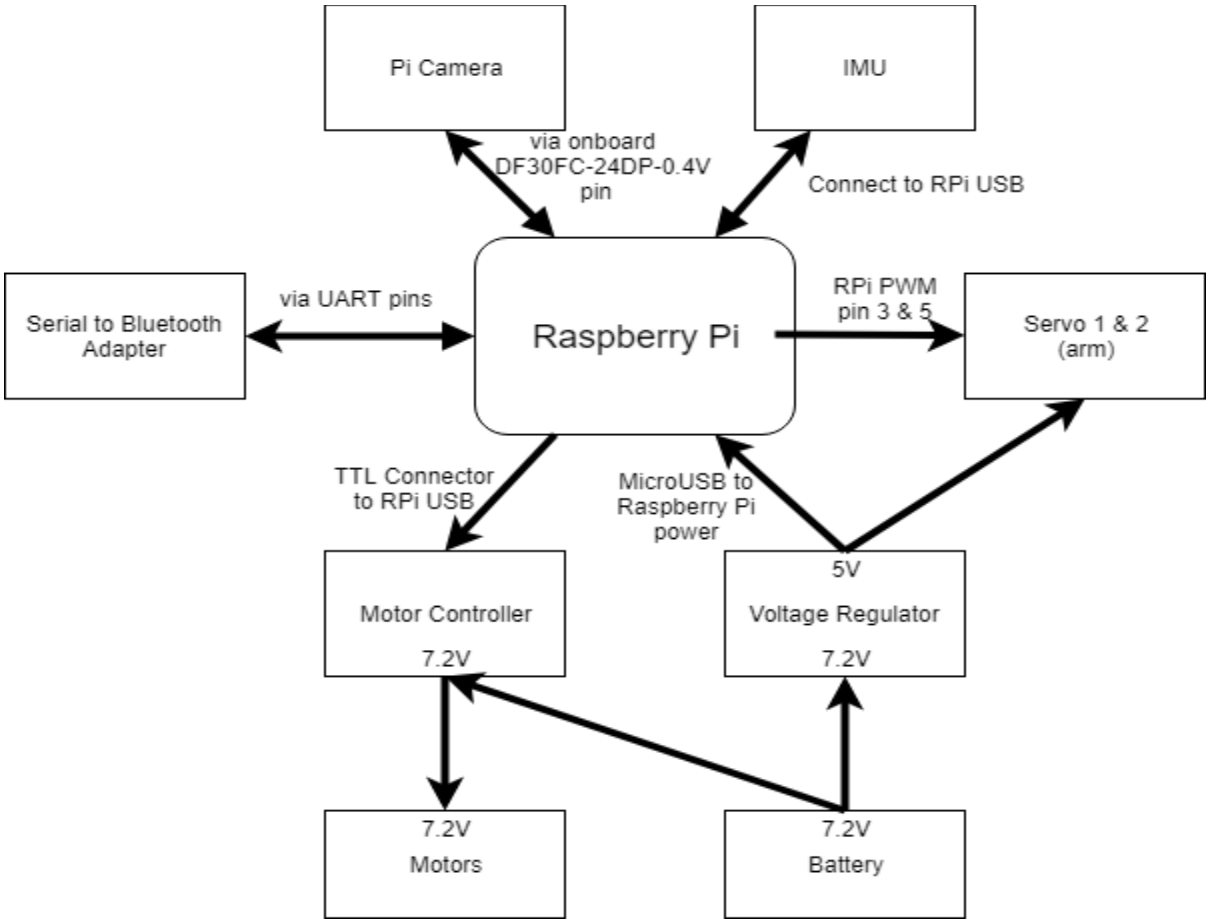


*Figure 1: System block diagram.*

The following procedure describes how the system in Figure 1 operates.

Use characteristics to identify the baseball from the grass

The first step was to find distinct characteristics that could be used to separate the baseball from the rest of the image. A number of different techniques were used on the image including extracting image textures and looking for smooth areas, using image segmentation, and seeking channels in various color spaces in which the baseball stood out. After analysis and testing of each of the different methods it was concluded that the saturation and intensity channels of the HSV color space would be the most effective in finding the baseball. The seams of the baseball

created problems when trying to use texture or the red color channel or hue channel in HSV, while the blue channel worked well for well lit baseballs but failed to capture baseballs in shadows or dim lighting. The saturation channel worked very well in all cases but also highlighted bright patches of grass, and the intensity channel again found well lit baseballs but did not account well for shadows. By combining the inverse of the saturation in with the intensity the baseballs were made to stand out because the intensity channel tended to cancel out the false positive areas in the saturation channel while still highlighting most baseballs to a lesser extent.

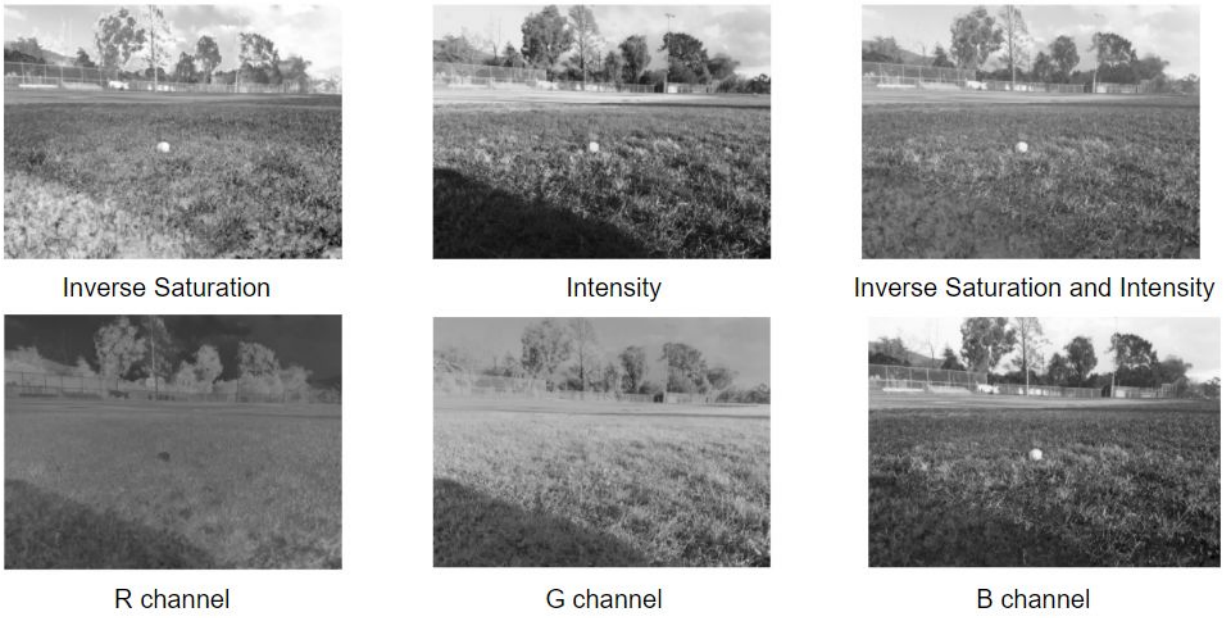


*Figure 2: Shows various color space channels for the same test image.*

As shown by the images in Figure 2 the RGB channels were unable to accurately detect the ball due to the fact that the lighting can cause significant variations in the values for each of the channels making the RGB color space not viable for the system. The inverse of the saturation was able to detect the ball but had the flaw of also giving high values to other parts of the image. The intensity though was able to detect the ball and would give low values for the parts of the image that the inverse saturation would detect as high values. The combination of the two channels was able to counteract each other giving an image where the ball is pronounced with no other features being close the same values. As shown by the top right image in Figure 2 The procedure for creating the pixel values used in the rest of the process goes as follows:

1. Convert RGB image to HSV
2. For each pixel p in the image:

a. Inverse_Saturation(p) = 255 – Saturation(p)
b. Final_Value(p) = (Inverse_Saturation(p) + Intensity(p)) / 2

Apply Median Filter

The image needs to have a median filter applied in order to try to limit the amount of noise the images contain. A median filter blurs an image by replacing every pixel with the median value pixel of its neighborhood. The sunlight was found to create a glare on a small number of pixels in the grass that would interfere with the detection of the baseball. The glare made parts of the grass have both low saturation and high intensity that would cause the system to confuse those pixels with pixels of the baseball. It was evident that some kind of blurring was going to be necessary to isolate the ball in the image. However, it was essential to maintain the edge of the baseball for edge and circle detection later. This, along with the fact that the pattern of noises created by the bright tips of the grass along with the dark roots resembled a salt-and-pepper type noise, led to a median filter being the most effective choice to apply in order to reduce the effect of the noisy grass in the image.

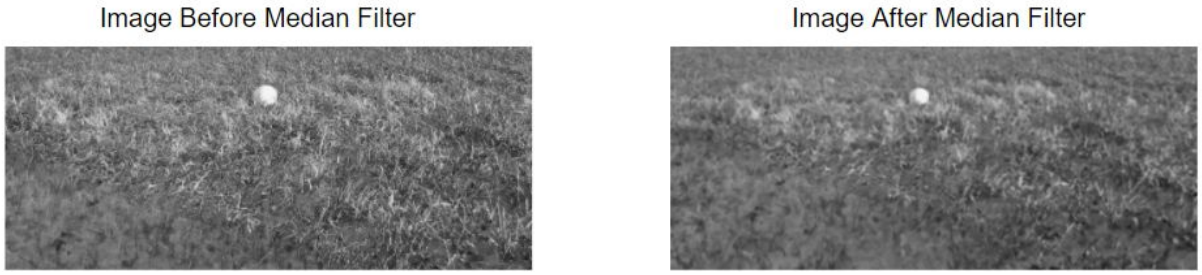


*Figure 3: Shows inverse saturation and intensity before and after applying the median filter.*

As shown in Figure 3, a number of the pixels of the grass had a high value for the combination of the inverse saturation and intensity (shown by the white) which is the same as the baseball before the median filter was applied. After applying the median filter, the effects from the glare was reduced as demonstrated by the difference in white patches between the two images in Figure 3. The most effective kernel size for the median blur for this task was determined to be 17x17 based on its performance on several test images.

Create Mask

Once the color space has been selected and the effects of the noise were reduced the image is used to create a mask. The mask is created by first selecting a threshold value for the inverse saturation and intensity and then setting all pixels with values greater than that of the threshold to

1 and the rest of the pixels to 0. The threshold value was selected by first finding the maximum inverse saturation and intensity value for the image then setting the threshold to 92% of that maximum value. We found that using the maximum value for inverse saturation and intensity made it so the system can tolerate the different lighting conditions that can occur in the images. The value of 92% was also found to keep the maximum number of pixels in the ball while also not including too many other pixels from the background.

*Figure 4: Shows an example mask for an image.*

Erode and Dilate Mask

When creating the mask some pixels from the background may be able to get through the thresholding. This causes problems when trying to find circles in the mask using the Hough Circle Detection.

*Figure 5: Shows a mask with pixels that have made it through thresholding.*

To remove these pixels, erosion and dilation can be applied to the image. Image erosion involves using a kernel to find the minimum pixel value in a neighborhood and setting the center pixel to that minimum value (minimum filter). This step will eliminate those pixels that made it through the thresholding but are not a part of a large grouping of pixels (not the ball). The problem with using erosion is that it will also reduce the number of pixels of the ball. Since the radius of the ball is used in future steps the radius of the ball must be maintained. In order to maintain the radius of the ball dilation must be applied to the image after erosion. Dilation is using a kernel and finding the maximum pixel value in that kernel and setting the center pixel to that maximum value (maximum filter). Three iterations of a 3x3 erosion kernel were was used followed by three iterations of a 3x3 dilation kernel. The result of the total process is shown in Figure 6. Notice the radius of the ball is maintained while the pixels that made it through the thresholding but do not belong to a larger grouping of pixels have been removed.

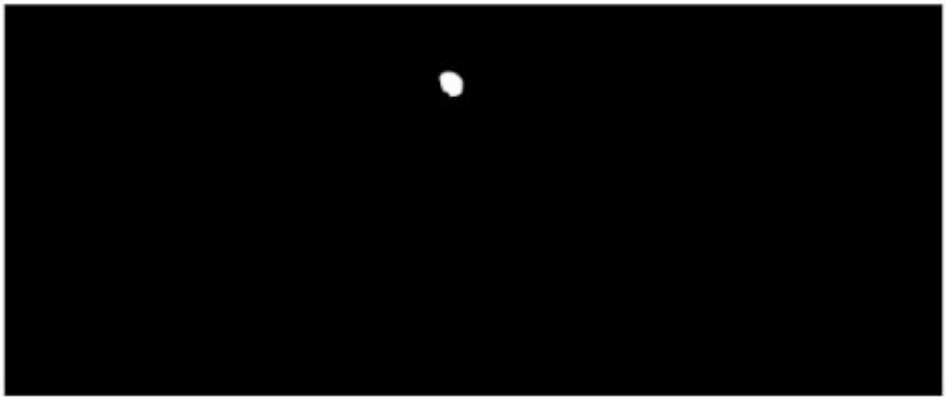

*Figure 6: Shows the same mask as figure 5 after erosion and dilation.*

Remove Background from Mask

In order to remove pixels from the top of the image which do not contain the ground , all pixels in the mask above the vanishing point of the ground in the image mask were set to be zeros. In order to compute the vanishing point where the ground meets the background in the image, the camera focal length, vertical angle of view, and the "tilt angle" of the camera are used as follows (see figure 12 for a visualization) :

To find the number of pixels in the image per meter on the light sensor 3mm behind the camera lens, denoted as ppm:

$$ppm = R_Y / ((2 * \tan(\alpha_Y) * f)$$

Where $R_Y$ is the y resolution of the image in pixels, $\alpha_Y$ is the y-axis field of view, and f is the focal length of the camera. Then the vanishing point of the ground, V, can be found with

$$V = f * tan(\alpha_Y - \phi)$$

Where $\phi$ is the so called "tilt angle", or the angle between the direction the camera is pointing and an imaginary line through the camera and parallel to the ground, as shown in Figure 12. Finally, the y-axis pixel value at which everything above does not contain ground can be calculated as

$$Y_{cutoff} = ppm * V$$

For our camera, the tilt angle is 20°, focal length is 3mm, and y-axis field of view is 24.4°. The above calculation result in the value 41 for the y-axis pixel cutoff, so all pixels with a y value less than or equal to 41 were set to be black in the mask (image coordinates place (0,0) in the top left corner, so values less than 41 correspond to pixels above that point). Note that the example below was created with a smaller tilt angle than 20°, so the y cutoff was greater and more of the mask is eliminated.

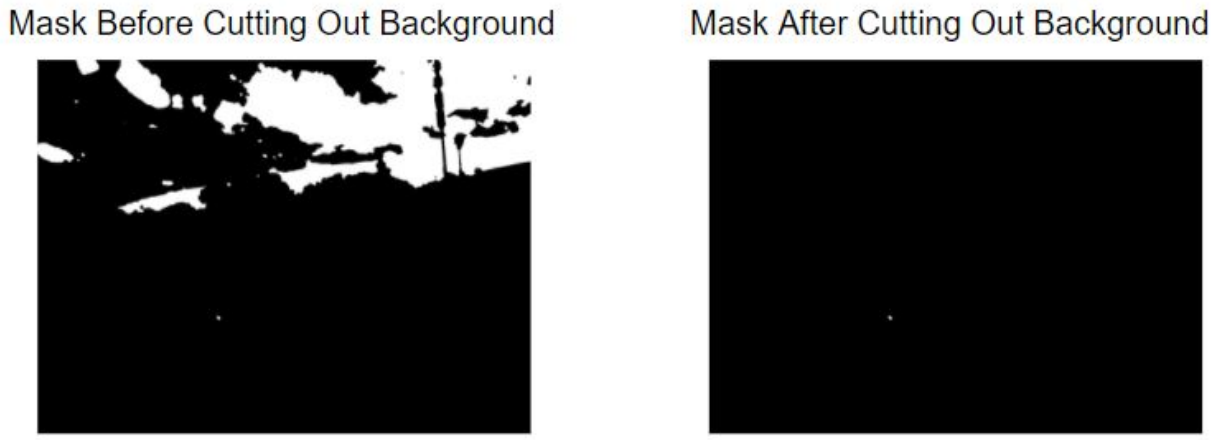


*Figure 7: Shows a mask before and after removing the background.*

Use Hough Circle Detection

After the mask is created and the background has been removed from the mask the Hough Circle Detector is applied to the mask to find the location and radius of the ball in the image. Hough Circle Detection works by taking a binary image and determining all possible circles within a

range of radii that could contain each white pixel and then using a voting system to find the most likely candidate circle that will contain as many pixels along its edge as possible. The Hough Circle Detector was chosen due to the fact that it will work even when some of the data is missing from the image. If only half of a circle is visible, it can still find it because all the points in the half circle will vote for all circles which could contain them, and at the end the circle candidate with the most votes will be the circle which contains all of them, even though that circle only received half of a full circle worth of votes. This is important because the full circle from the ball will usually not be found since the grass will cover part of the ball. In order to find the ball and not any of the other blobs that might be present in the mask a series of Hough Circle Detectors are used. This series starts with the first Hough Circle Detector with parameters that will only find balls that are very pronounced. If no ball is found in the Hough Circle Detector the parameters will be loosened and the Hough Circle Detector will be run again. The strictest search for circles uses a Hough space with a resolution 1/5 of the resolution of the original image and requires 150 votes in a bin for a circle to be detected. The radius must be between 120 pixels and 20 pixels for this test to detect the ball, eliminating the possibility of very large circles being detected out of scattered noise. The last circle detection uses smaller bin sizes but requires only 5 votes to produce a match, and it looks only for balls with a radius less than 20 pixels to prevent noise from being detected as a ball. It should be noted that prior to performing the Hough Circle Detection algorithm, a canny edge detector is run to determine the edges of the ball. Generally the parameters to the edge detector would be critical in Hough Circle Detection, but because we have already created a binary mask in the thresholding step the edge detector will always find the same edges regardless of parameter choice. The process of loosening the circle detection parameters and running the Hough Circle Detector will continue until either a ball is found or until a total of 4 Hough Circle Detectors are run, and if no balls are found after the fourth detector then it will be assumed that the image does not contain a ball. For the purpose of data visualization, if a ball is found, then a red circle will be placed around the ball in the original image. The size of the red circle will also indicate the size the Hough Circle Detector believes the ball is. A few examples of the ball detector are shown below:

*Figure 8: Shows ball detector at 5ft with zoomed in image on the right.*

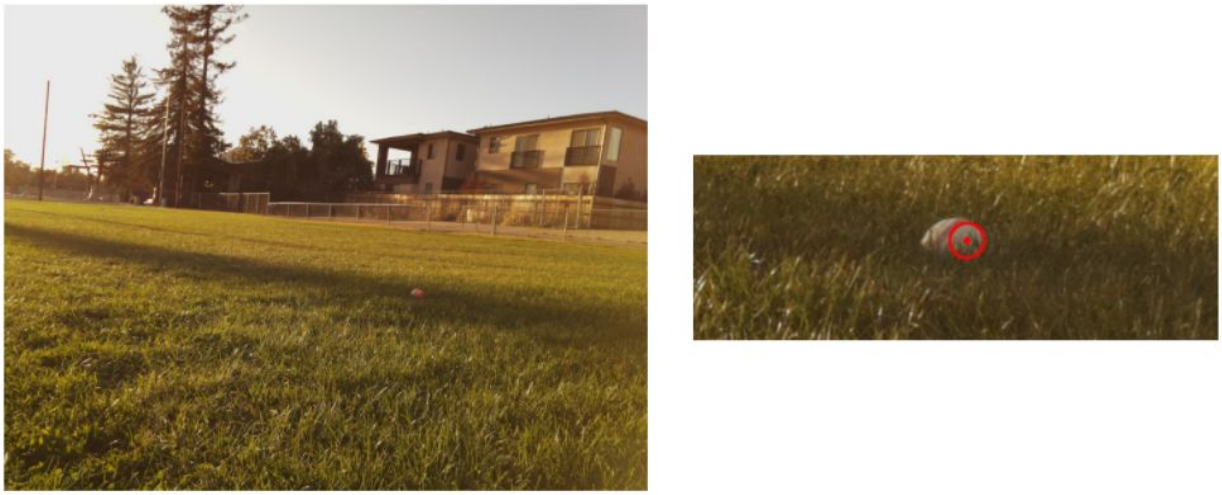

*Figure 9: Shows ball detector at 8ft with zoomed in image on the right.*

*Figure 10: Shows ball detector at 15ft with zoomed in image on the right.*

*Figure 11: Shows ball detector at 25ft with zoomed in image on the right.*

As shown by the images above the Hough Circle Detector used on the mask is able to find the ball regardless of the lighting conditions or the distance to the ball. However, the detected radius is affected significantly by lighting and grass conditions.

Calculate Ball Distance and Angle

Once a ball is detected, it's position in the image is used to calculate a relative distance and angle from the robot. The first step in determining the distance and angle is to convert from the pixel position in the image to a physical distance in the 2D image plane in the camera. After finding the pair (x,y) representing how far in meters the ball is from the center of the 2D image plane, the angle can be determined by taking the inverse tangent of x over the focal length. The distance is more complex. Let H be how far the camera is from the ground, f be the camera focal length, and ϕ be the tilt angle, which is the angle that represents how far the camera is from being pointed parallel to the ground.

$$Distance = H / \tan(\phi - \arctan(y / f))$$

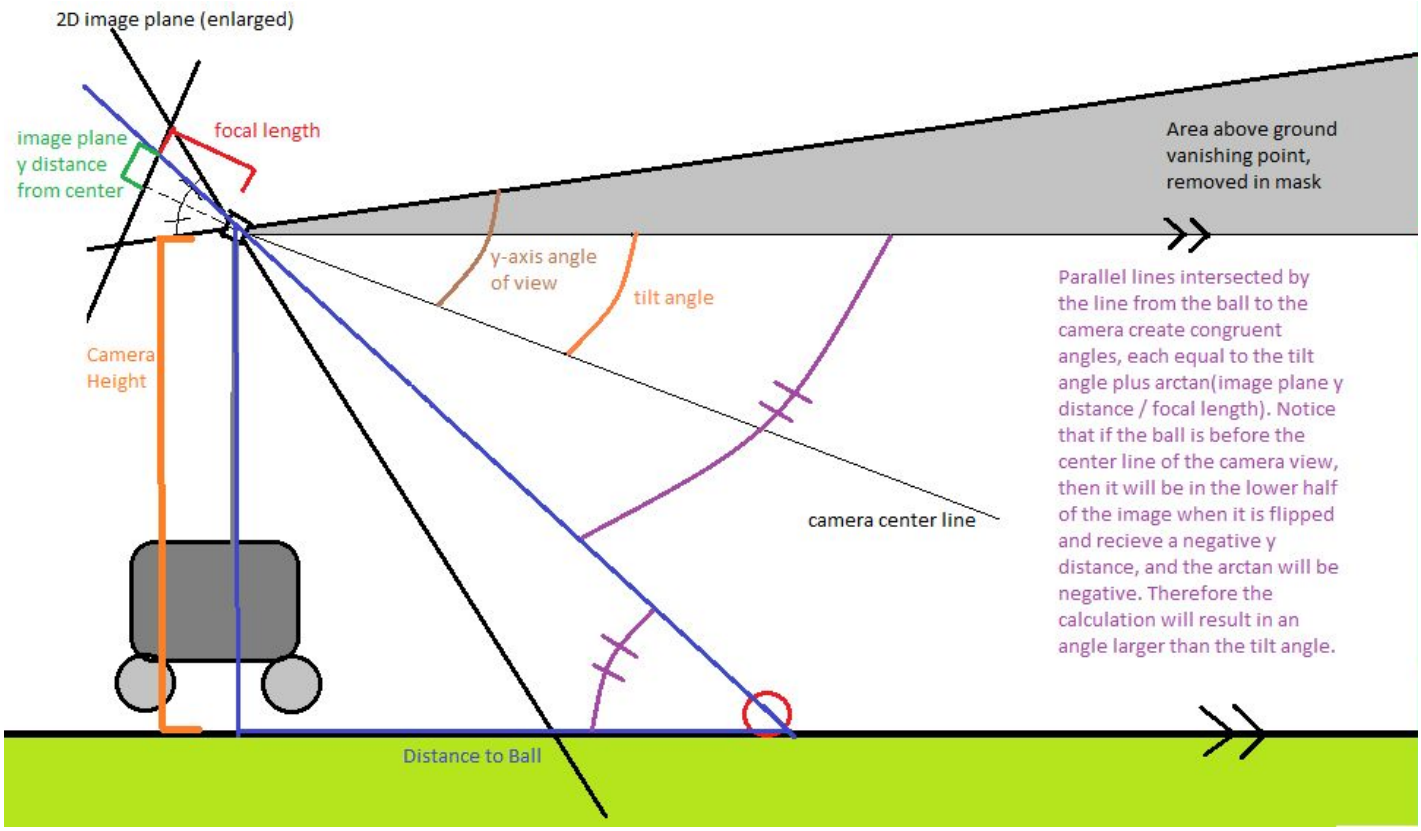


*Figure 12: Shows key elements of ball distance calculation*

The distance and angle to the ball are then sent to the control module of the robot so that it can determine appropriate motor control to approach the baseball.

**Robot Control Flow**

Once the distance and relative angle to a ball are produced by the computer vision module, they are sent to the controller which will determine the necessary motor drive needed to approach the baseball. The flow for moving towards the baseball is as follows:

1. If the ball is within grabbing distance and directly ahead, then stop and attempt to pick up the ball. Otherwise, turn until the the directional heading of the robot is equal to the previous direction plus the angle to the ball from the computer vision module.
1. Move forward until half the calculated distance to the ball has been covered.
2. Stop and take a new photo and calculate a new distance an angle.
3. Repeat steps 1-3 until the ball is picked up or the ball is no longer in the image.

**Sensor Input and Motor Control**

There are two forms of sensor input aside from the camera used in the closed loop control of the system. The first is an inertial measurement unit (IMU), which can determine compass heading measured from 0 to 360 degrees. This is used for angular control in the system. The second sensor is a rotary encoder on the back wheels. The encoder tells the controller how far the wheel has spun, and with the addition of the raspberry pi internal timer, how fast the wheels are spinning. To determine the  necessary motor drive to achieve a new position and heading, two PID loops are used. One uses the error in position as determined by the encoder to determine a motor drive that increases with positional error and gives the same drive to each motor. The other loop uses the error in directional heading based on the data from the IMU to determine a drive that increases with angular error and adds that drive to the positional loop drive of the motors on one side, while subtracting it from the drive given to the other side. The result is that the wheels on one side will spin faster than the other when the system needs to turn , but the overall speed is based on how far away from the target the robot is. If the positional error is zero but the directional error is not, then the motors will spin in opposite directions and the robot will turn in place.

**Robotic Arm**

The robotic arm was designed to be as simple as possible. Initial designs and thoughts for how the arm should operate were unnecessarily complicated as the students had become accustomed to seeing humanoid robotic arm tutorials online. Ultimately, the final design has only two levers, both controlled by their own servos. These two levers were referred to as the arm and the hand.

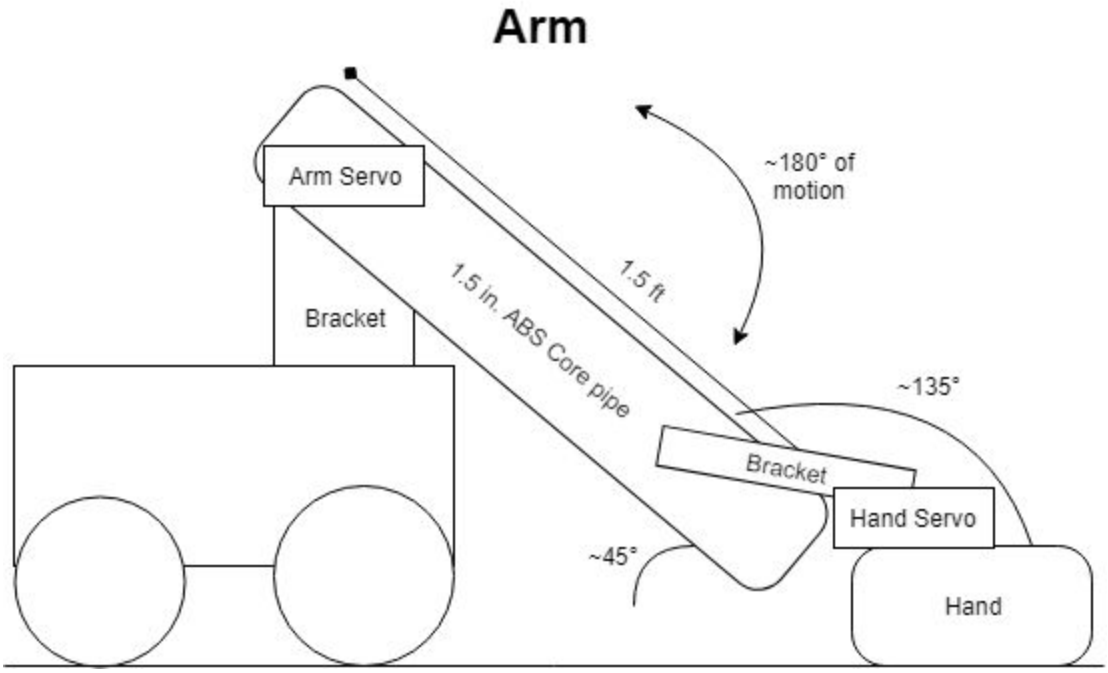


*Figure 13: Diagram of the robotic arm. Approximate measurements included but drawing is not to scale.*

When designing the arm, weight was a primary concern as the servo motors that the students had access to were underpowered. Thus a foam core pipe at 1.5 inch diameter was chosen as it was relatively lightweight yet still strong enough for the given application. Ultimately the arm servo is responsible for raising and lowering the hand from and to the ground so that the robot can lift the baseball.

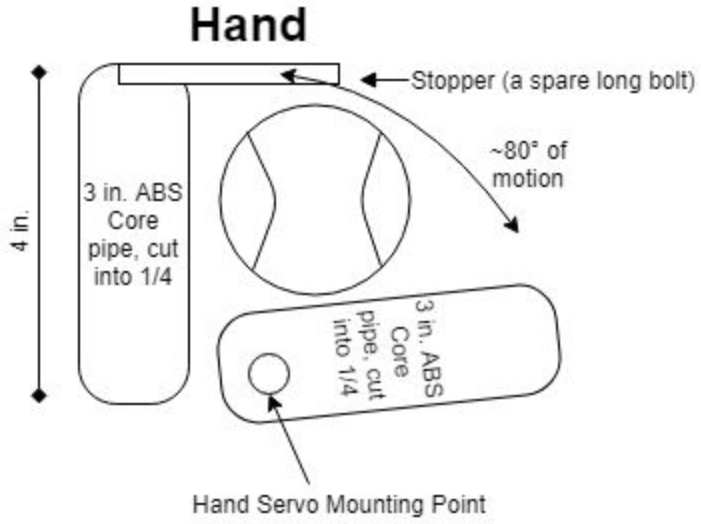


*Figure 14: Diagram of the hand on the robotic arm. Approximate measurements included but drawing is not to scale.*

The hand is similarly simple. A piece of 3 inch diameter foam core pipe was cut into quarters at 4 inches long. Two of the quarters were used to create the hand, where one quarter is stationary while the other pivots to contain the 3 inch diameter baseball. Closing the hand in this way often pushed the baseball away, so something needed to be added to prevent the baseball from rolling out the front of the hand while it closed. To handle this, a spare 2 inch bolt was added, labeled stopper in figure 14.

Both the arm and hand servos used the same basic code routines, which required controlling a pulse width modulation pin each from the Raspberry Pi. The code was written in Python which would set up general purpose pins as pulse width modulation. Then a quick calculation in the code was used to convert angles into the required duty cycle. For the servos used in this project, the range of acceptable duty cycles was 2.5%-12.5%. Thus the calculation ideally was to divide the angle by 18 and add 2.5 to get the necessary duty cycle. In practice the servos responded better to adding 2.1 instead. Simple for loops were used to create stepper functions to slowly control the servos through their range of motion. These stepper functions were necessary as the shift in weight of the arm abruptly swinging up or down greatly affected the suspension of the robot. The angles used in practice with the robot were found through guess and check as the components were mounted to the servo. These angles were hard-coded into the Python code and used to fully raise and lower the arm and open and close the hand.

## 3. Testing and Results

Camera Tests

Our initial camera test involved taking many pictures of baseballs in a field at varying distances. This allowed us to calibrate our computer vision code to allow us to reliably find baseballs in grass. We also tested different camera angles, trying to maximize the amount of field visible in the image while also being able to see the ball up close.

Motor and Encoder tests

Our first series of motor tests were conducted on a hardwood floor. The first involved simply seeing if the wheels would spin. After, we used 30/45/60 degree turns to calibrate our PID loop. By watching the oscillation, we tuned the parameters to ultimately have one smooth turn. When the arm was added, the weight distribution changed enough to affect our movement control significantly. Our decision to use a PID loop meant that we only had to re-adjust parameters.

Our initial attempt to calculate distance involved integrating the IMU's acceleration data to get a velocity and then integrating that to give us a distance. We found that the small error given by the acceleration data significantly increased the error in our distance calculation, because a small amount of noise in the measured acceleration when integrated would result in the calculated velocity being off by a constant amount, and the constant error in velocity would then be integrated over time resulting in a positional error that grows linearly over time even while the robot is stationary. This method would have allowed us to save the cost of the motor encoders, but due to the issues above, the cost was unavoidable. To test the encoders, we wrote a script that converts the counted encoder ticks into a distance. By inputting a distance of 1m, we could see that the encoders were working properly. Although done on hardwood, this method of calculating distance works regardless of the surface being tested on by measuring how much the wheels actually turned, so long as the wheels are making solid contact with the ground and not spinning out. The torque necessary to get started on grass is another issue. While initially the rover could traverse the grassy terrain without issue, the weight added by the addition of the arm was too much for our motors to handle. The motors did not have enough torque to start moving. Since this roadblock could not be overcome without a higher torque version of our rover or a complete refactoring of our custom arm, we moved to turf for testing. An additional issue arose when the wheel with the motor encoder was not in solid contact with the grass (due to there being a low point in the grass). Since the wheel could spin more freely than the rest, it would be counting distance before the system actually started moving, throwing off our distance calculation. With the addition of the arm, this became less of an issue due to the added weight.

On the turf, the movement was jerky. The amount of motor drive necessary to get started is so high that when the system starts moving, it lurches forward before it is able to realize that it is moving too fast and needs to slow down. To resolve this issue, we would want to use motors with higher torque and lower speed or motors with a higher gear ratio, or run our control loop at faster intervals. However, because we did not use dedicated hardware for counting encoder ticks and there is no external interrupt support on the Raspberry PI, and because running the control loop took longer than the amount of time for the encoder to change states, running the control loop more often would result in more missed encoder ticks and ultimately poor distance measurement. Ideally the motor control loop would be run on microcontroller with real-time guarantees to solve this issue.

Arm Tests

Initial tests of the arm were done entirely separate from the robot as a whole. These were basic tests of the arm and hand servos and their capabilities to raise and hold the baseball. These tests were successful, however testing while on the robot saw several failures. The first arm test while on the robot involved lowering the arm, closing the hand, and then raising the arm into a vertical position where it could rest. The weight of the arm while holding a baseball caused the connection between the servo and the arm to completely strip. This ultimately wasn't a failure of the arm, but a failure of the plastic piece being used for the connection. The connection was then reevaluated and made greatly stronger by creating a mount on the arm that was screwed into the servo. The test was rerun and this time the robot was unable to lift the ball. The moment created by the ball on the servo was too much for the servo to handle. It was unclear if the power draw of the entire system was resulting in a lower voltage being sent to the arm or if the torque of the motor was just not enough. Instead of incurring additional costs by purchasing another servo, the team decided to use a lighter ball for the testing.

During the testing of the system as a whole, other issues arose that affected the arm. One issue was that the code was written initially to set the angle of the arm abruptly between on the ground and in the air. This abrupt switch jerked the robot around greatly, which led to inaccuracies in the system as a whole. This is why the stepper functions were written for the motion of the arm. This resulted in the robot remaining more stable. Likewise an issue with powering the Raspberry Pi was found during practical tests, which caused issues with powering the arm. This was because the servos on the arm are drawing their power from the Raspberry Pi in this design. When the Raspberry Pi would shut down, the drive to the arm would give out. This resulted in the arm dropping from its position to the ground. A final revision of the robot would involve powering the servos independently from the board to eliminate this issue from occuring.

## 4. Conclusions and Recommendations

The current performance of our system seems better tailored for an environment like a tennis court. Slight changes to the computer vision code can make it find tennis balls, and the torque related issues of the motors and servos would not exist on the hard surface tennis court and with light tennis balls. Maybe a future application could replace the sprinters that shag tennis balls at professional tennis matches.

If a second iteration of this project were to be created, the following changes would be made:

Higher torque, lower speed motors / higher gear ratio on motors: The motors currently used have too little torque to reliably move the system in long grass.

Stronger servo motor: The servo motor currently used to lift the arm generates enough torque to lift the arm but not enough to consistently lift the arm while it is holding a baseball.

Stiffer suspension: The weight of the arm causes the current suspension to bend significantly which can cause issues in long grass.

Include a quadrature counting chip: The Raspberry Pi, unlike many microcontrollers, does not have pins that can be used as counters for a motor encoder, and does not support externally triggered interrupt. Additionally, it is not a real-time system, meaning there is no guarantee that the operating system will choose to run the control code at any particular time interval. This meant that the encoders had to be polled frequently to measure distance travelled, and even still occasionally encoder ticks would be missed. There are chips that can be purchased which would count encoder ticks and report back to the Raspberry Pi over a serial connection which would remove the need to poll the encoders.

Use a traditional microcontroller: For reasons addressed above, it may be worth dropping the Raspberry Pi altogether and switching to a more traditional microcontroller. While working out the camera and computer vision aspects may be more challenging without the Raspberry Pi, having access to interrupts, timers, counters, and other typical microcontroller components would have been useful especially in controlling the motors. Alternatively, a separate microcontroller could be used specifically for motor control and encoder reading, and could take instructions from the raspberry pi over a serial interface.

Use a larger chassis: The current system lacks the space to carry multiple baseballs at a single time. There is a six wheel version of the chassis we used which would give space for more balls

to be carried at once, in addition to potentially providing a more stable platform on which the system operates.

## 5. Appendices

[A] Parts List

| Part Name | Part Cost | Quantity | Total Cost |
|---|---|---|---|
| Simpson Strong-Tie 3 ⅛ in. x 5 in. plate | 0.99 | 1 | 0.99 |
| Simpson Strong-Tie hurricane tie | 0.79 | 2 | 1.58 |
| Simpson Strong-Tie 4 inch bracket tie | 0.99 | 1 | 0.99 |
| Simpson Strong-Tie 3 inch plate | 0.69 | 1 | 0.69 |
| 1.5 inch diameter ABS pipe | 2.42 | 1 | 2.42 |
| 3 inch diameter ABS pipe | 6.94 | 1 | 6.94 |
| 6x6 rubber sheeting | 4.83 | 1 | 4.83 |
| LewanSoul LD-27MG full metal digital servo | 17.99 | 2 | 35.98 |
| 6-32 x ½ inch bolt | .13 | 4 | .52 |
| 8-32 x ½ inch bolt | .14 | 8 | 1.12 |
| 6-32 x 1 inch bolt | .14 | 4 | .56 |
| 8-32 x 1 inch bolt | .16 | 4 | .64 |
| 6-32 x 2 inch bolt | .17 | 4 | .68 |
| 6-32 locking nut | .14 | 4 | .56 |
| 8-32 locking nut | .11 | 8 | .88 |
| 4-32 x 1 ½ inch bolt | .17 | 1 | .17 |
| 4-32 x ½ inch bolt | .11 | 5 | .55 |
| Raspberry Pi 3 | 39.95 | 1 | 39.95 |
| 32 GB SD card | 12.54 | 1 | 12.54 |
| Adafruit BNO055 IMU Breakout | 14.95 | 1 | 14.95 |

| | | | |
|---|---|---|---|
| PiCamera V3 | 29.95 | 1 | 29.95 |
| USB to TTL Cable | 9.95 | 2 | 19.90 |
| Wild Thumper 4WD All-Terrain Chassis w/ 34:1 Motors | 174.95 | 1 | 174.95 |
| Pololu Qik 2s12v10 Dual Serial Motor Controller | 74.95 | 1 | 74.95 |
| 7.2V 5000mAh NiMh Rechargeable Battery Pack | 21.99 | 1 | 21.99 |
| 5V Step-Up/Step-Down Voltage Regulator | 14.95 | 1 | 14.95 |
| 43:1 6V HP Metal Gear Motor with 48 CPR encoder | 36.95 | 2 | 73.90 |
| HC-06 Bluetooth Serial Adaptor | 7.39 | 1 | 7.39 |
| | | Total: | 545.52 |