# Neato Keeper

Paul Krusell and Jeff Pflueger

April 2017

## 1 Introduction

The original goal of Neato Keeper was to make our Neato play soccer. We wanted the robot to detect the position of a ball, calculate its velocity, and predict where it will go. Then we would try to intercept the ball and deflect it. We ended up with a system for tracking the position and depth of the ball and a method of calculating velocity, but not interacting with it.

## 2 Implementation

Our process for tracking the ball is broken down into a few different steps:

- Image Processing
- Ball Detection
  - Circle Finding
  - Differentiation
- Depth Detection
- Velocity Detection

### 2.1 Image Processing

This is the part of the code where we modify the image. The Node takes in an image from the Neato and saves it with OpenCV. Then we apply a color filter to a copy of the image. Because the ball we are trying to track is blue, we filter specifically for blue. We also copy the original image and transform it to Gray-Scale. We then use this image for detecting circles.

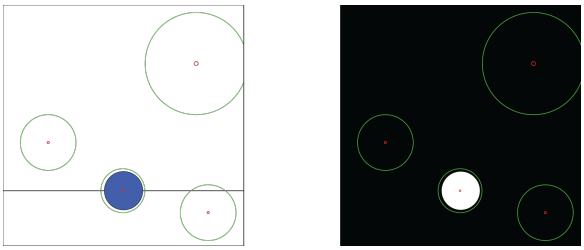

Figure 1: *Left*: Image with detected circles, *Right*: Binary masked image

## 2.2 Ball Detection

We divided finding the ball into two parts: Circle Finding and Differentiation. We decided instead of using the color mask to isolate part of the ball, that we would start by attempting to detect circles in the image. Then, we create a mask out of each individual circle. Each of those individual masks get overlain on the color-filtered binary. The entire process is shown in Figure 1. The program gets rid of all the values that aren't within the circle, and then counts up the amount of pixels active within each circle. The circle with the most white pixels is chosen as the one with the ball in it.

### 2.2.1 Hough Circle Transform

To find the circles in the image, we decided to use OpenCV's Hough Circle Transform function. It works like this:

- Edge Detection
- Define Accumulator Matrix
- Populate Accumulator Matrix
- Find Local Maxima

The function then returns the locations of the local maxima of the accumulator matrix as potential circles.

To detect the edges in the picture, the function uses Canny Edge Detection. It then creates an accumulator matrix, essentially a grid, for the image. The function checks each edge, and looks at each cell in the accumulator matrix. If it would make sense as the center of a circle, it is incremented. This is determined by the gradient at the edge. Then, when all of the edges are checked, The local maxima of the accumulator matrix are identified as potential circles and returned.

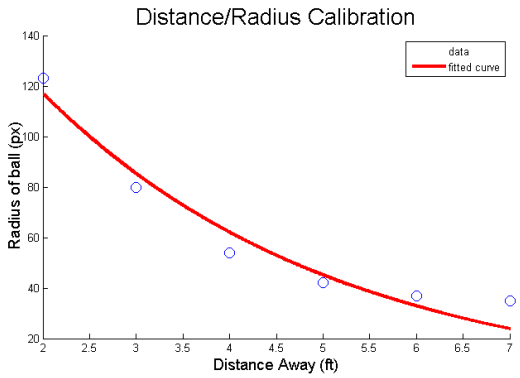


Figure 2: Calibration curve for the Depth of the ball.

### 2.3 Depth and Velocity

Once the ball could be consistently detected, we had to figure out how to determine a depth and velocity using a 2-D image. For the Y-direction, we took measurements of how wide the ball was in pixels at several different distances. We then fit a curve to this data and used this function to predict the ball's depth, to good accuracy.

The curve-fit model fails when the ball is too far away to follow the expected curve. This distance is around 6.5ft. You can see in Figure 2, that the measured values of pixels are roughly the same for six and seven feet, and our fit doesn't match the data from six feet on.

For the X-direction, we used the fact that the ball was 7 inches across to convert the size of the ball from pixels into feet. We calculated the difference from the center of the previous ball to the center of the current ball in pixels and converted to feet, which gave us the horizontal distance in feet. With a full distance vector, and could use the camera message's time-stamp to calculate the difference in time from one frame to the other. Dividing the distance by this time gave us our velocity in the correct direction. From here we used python's math.hypot and math.atan to find the magnitude and phase of the vector.

## 3 Results and Lessons Learned

In the end, our robot was able to see a ball moving on its camera and determine the magnitude and phase of it's velocity vector. Although we planned to move the robot to intercept the ball, we didn't have enough time to implement this.

We lost about one and a half weeks fighting with the Hough Circle transform and ball detection, which hampered our process a lot. If we had pivoted to a

different method of circle detection earlier, or been more agile in trying new things, we could have found a solution sooner.

Another lesson we learned is that we should have had a more solid plan regarding our geometry in the beginning. We ended up doing a lot of math as we went along, which could have been worked out earlier in the process. This would have made our work sessions more efficient towards the end, and given us more structure to work with.