Computer Vision for Autonomous Robot Motion Planning

ME 133b Final Project

Neha Sunil

March 16, 2018

Motivation

The 2018 ME72 competition "Tank Wars" was broken up into matches during which two teams fielded three robots each. Multiple "bases" were set up across the game field, also featuring a ramp, seesaw, and cardboard obstacles. Bases could be captured by robots pressing a button on the base, and teams were awarded points for the time spent controlling a base.

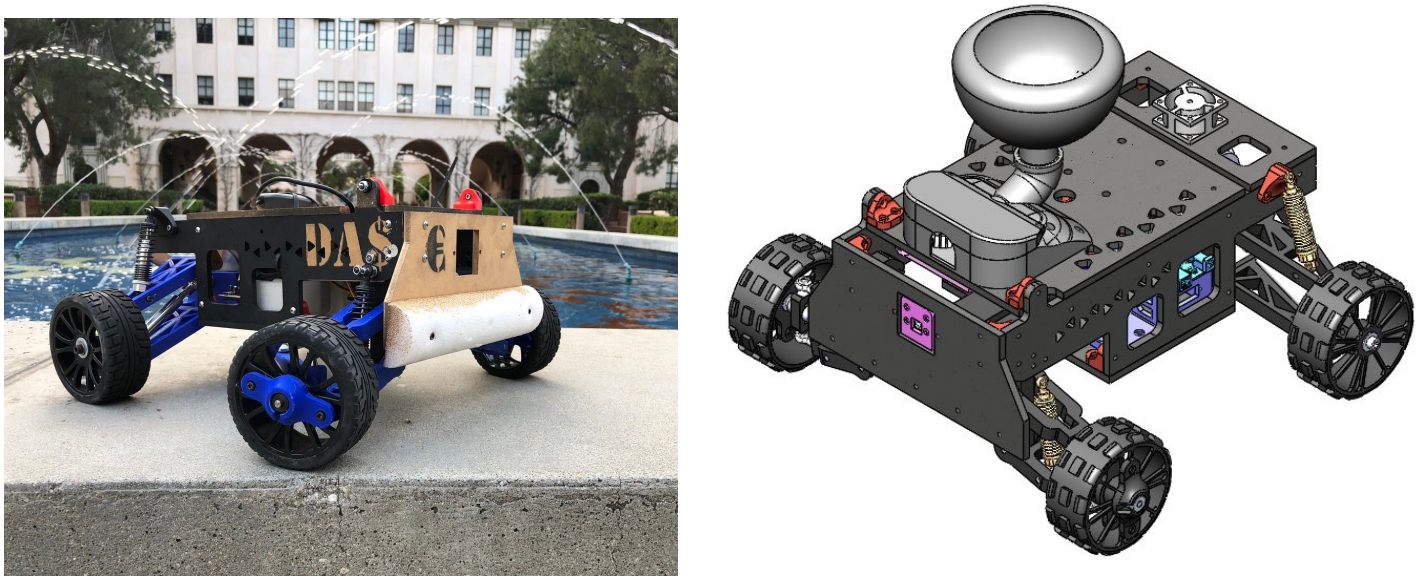

Figure 1. Photo and Solidworks model of one of our team's robots. The camera was placed in the centered square hole shown in the front plate.

The beginning of each match featured a 40s autonomous period before the remaining five minutes of tele-operated play. Some teams attempted to use magnetometers or IMUs for localization, while others simply coded instructions to move forward during the autonomous period. Our team endeavored to use computer vision to position our robots in the ideal starting positions, with one robot aiming towards the edge of the nearest ground base and the other two aligning with the ramp and the far seesaw respectively.

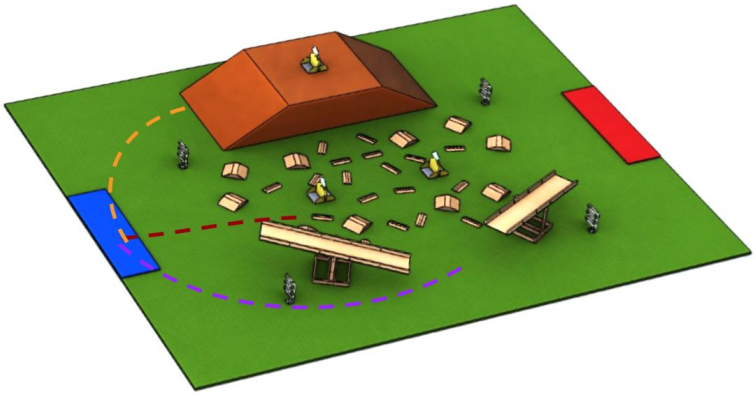

Figure 2. Ideal strategy for movement during autonomous period with original field layout.

Final Implementation

We connected an Arducam camera module to a Raspberry Pi, and all processing was done on the Pi using SimpleCV. We used I^2C communication between the Pi and the Arduino which was connected to our Xbox 360 remote receiver and the RoboClaw motor controller.

The Arducam 5MP OV5647 Mini Camera Video Module was the least expensive camera module ($15) we could find that was compatible with the Raspberry Pi. The camera is most comparable to the Raspberry Pi V1 camera module. The sensor has a resolution of 2592 x 1944 pixels, but we chose to do our computation with 320 x 240 pixels for faster processing. The horizontal field of view is 53.5 degrees, and the vertical field of view is 41.4 degrees. The UV4L driver was used to integrate the stream with Linux.

We chose to use an Arduino to interface with our Xbox 360 remote and motor controllers due to its ease of use. However, it does not have the processing capability in order to handle real-time image processing. The Raspberry Pi 3 Model B has a 1.2GHz 64-bit quad-core Broadcom BCM2837 processor, making it much better suited for computer vision applications.

SimpleCV is a Python wrapper for OpenCV, a widely used open source computer vision library . Because we didn't have any computer vision experience prior to this project, we chose to use SimpleCV, which has less features than OpenCV, but allows for faster prototyping.

We implemented two different vision algorithms for positioning the robots aiming for the ground base and the ramp. The final algorithm for the ground base robot involved segmenting the distinctively colored lavender base and using the pixel bounds of the segmentation to determine how much the robot needed to turn to approach a specific edge of the base with enough clearance to avoid hitting it. The code is available on GitHub.

The algorithm for approaching the ramp involved the robot turning with a pre-determined turning radius until it was oriented straight with respect to the ramp, as confirmed by the presence of horizontal lines surrounding the orange features of the ramp (Fig. 3). Once the robot was aligned, it moved forward or backward to be within 2 to 3 feet of the ramp to allow for the future possibility of hard-coding an ascent of the ramp. The code for this algorithm is also available on GitHub.

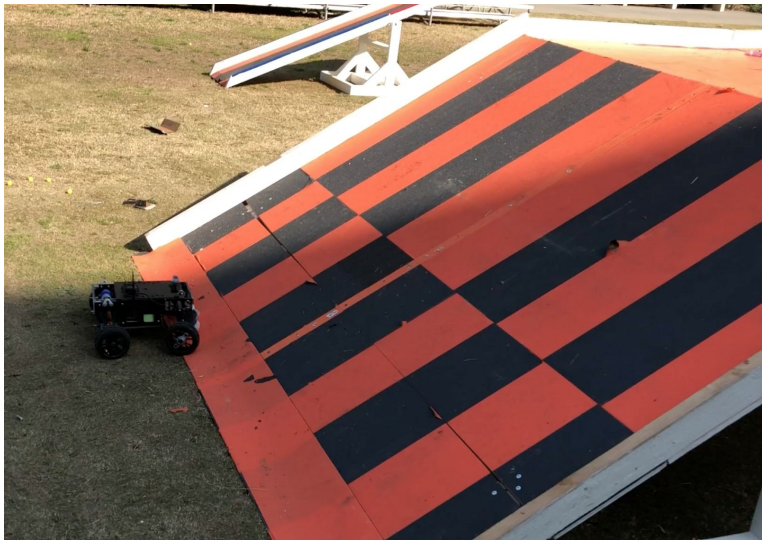

Figure 3. Distinctive coloring of ramp with a horizontal orange stripe at the base.

We initially communicated with the Arduino using a USB connection, but the RoboClaw's serial communication interfered. Therefore, we switched to I^2C, which was also easier to debug due to the availability of the serial port during testing without the motors.

This video shows a robot operating with the ground base computer vision.

Development Process

We did extensive testing with colorspaces to debug the discrepancies between what we could see, what the camera saw, and what the display showed. Furthermore, we determined that hue-based segmentation was more effective than RGB color-based segmentation because the former was flexible to changing lighting conditions. We also tested different vision strategies to determine which values we could obtain consistently and accurately, and found that we could segment blobs for distinctive hues and obtain accurate angles. Analysis on calibration images provided us with relationships between what the camera saw and how much the robot needed to turn. In the development process, we simplified the algorithms considerably by solving more specific problems.

A. Colorspace Analysis

The first discrepancy we encountered was from SimpleCV importing RGB values from the camera, but handling them as if they were in BGR format, meaning the red and blue channels were flipped. After manually switching the channels back to the RGB format, we noticed that the display showed increased contrast in the green channel, making it impossible to visually distinguish cyan from light green or orange from pink. Further analysis on the raw values revealed the green channel contrast appeared only in the display and the colors were distinguishable based on their RGB values. The display discrepancy correlated to the intensity of the blue channel, suggesting that an internal auto white balance mechanism adjusted the appearance of the display, but not the RGB values themselves.

Furthermore, data was collected during the day under dim natural lighting conditions with a color temperature of 5000K (similar to what would be seen during the competition in the shade), and at night with tungsten lighting with a color temperature of 3000K (Fig. 4). With the exceptions of white or extremely dark objects (black and brown), hue was a far more accurate measure for segmentation than RGB value for the changing lighting conditions. In the HSV colorspace, the H channel represents hue, the S channel corresponds to saturation, and the V channel relates to value, or the lightness of the color. A saturation of zero corresponds to gray, and a value of zero corresponds to black. The raw data and further analysis on colorspaces can be found in this informal write-up. The raw color calibration data, used for segmenting images by color, is summarized in Table 1.

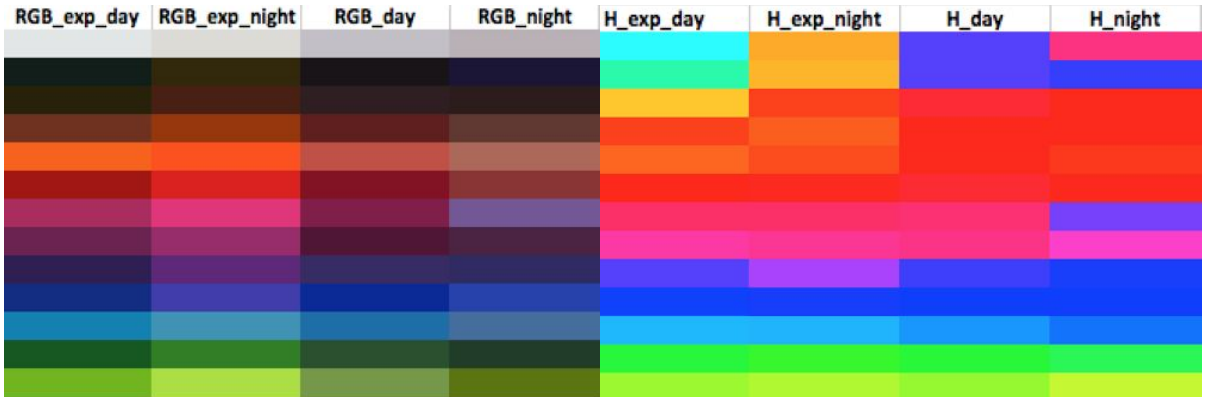

Figure 4. The expected and actual RGB and hue values during the day and at night.

Table 1. Mean RGB and HSV and standard deviation of hue for the samples segmented

Sample	Mean B	Mean G	Mean R	Mean SD	Mean H	SD H	Mean S	Mean V	Color
Grass	114	131	129	45.8	173	30.5	13	51	
Red Button	42	38	82	4.1	245	1.5	54	32	
Base, MDF	133	137	152	50.3	227	27.7	13	60	
Cardboard	232	224	239	8.4	300	9.2	7	94	
Seesaw, Orange	78	105	243	4.1	230	0.5	68	95	
Seesaw, Black	59	57	57	11.4	360	46.6	3	23	
Green Lid - Room	41	77	34	3.2	110	3.7	56	30	
Pink Gear - Shop	108	55	100	5.8	309	1.2	49	42	
Flashlight - Shop	233	234	233	0.2	105	27.8	0	92	
Seesaw, Orange - Shop	80	80	160	4.0	330	1.9	67	94	
Nerfball - Shop	77	146	160	33.9	190	14.4	52	63	
Floor - Shop	129	125	130	15.0	288	30.1	4	51	
Blue PLA - Shop	121	73	59	26.5	13	2.7	53	48	
Base, MDF - Shop	94	99	109	22.3	218	8.2	14	43	
Base, Purple - Shop	145	80	102	33.1	340	6.4	45	57	

The original competition guidelines specified that the bases would be painted yellow. However, since yellow (H=60 with RGB, H=180 with BGR correction) falls within the range of the grass hues, we requested that the bases be painted a more distinct hue that would not be present on competition field, and settled on lavender.

B. Seesaw Line Analysis

The first tests we ran with the field elements involved taking images of the seesaw from the camera mounted inside the robot at various positions and orientations to determine which values extracted from the image processing best indicated the original position and orientation.

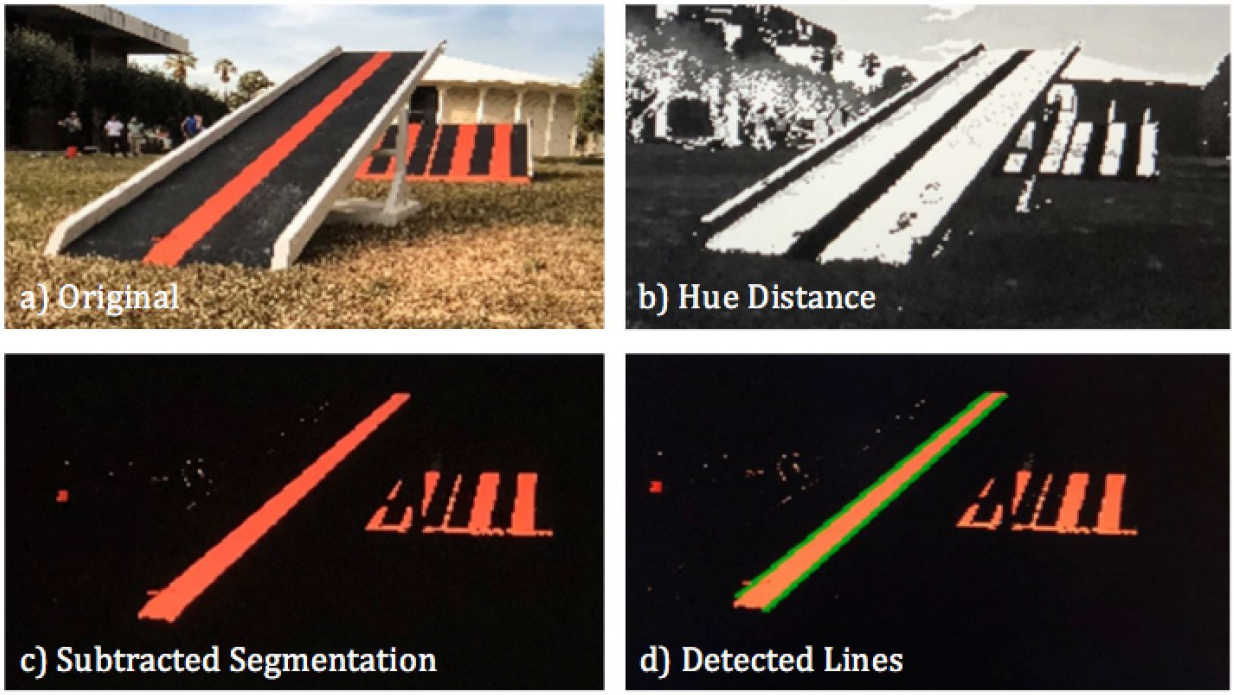

Figure 5. Steps for detecting lines on seesaw. The lines on the ramp are not detected due to the high threshold parameters of the findLines function. Future iterations of the binarized image use stricter thresholds and the erosion function to remove the noise visible in this segmentation.

We detected the lines on the seesaw (Fig. 5d) by taking a hue distance image with the orange of the seesaw as the hue parameter. The returned image (Fig. 5b) transforms pixels with a similar hue to black and scales the distances from 0 to 255 for a grayscale image. The image was then binarized so that all pixels under a specific value were set to a value of zero, and the rest were set to 255. This binarized image was then subtracted from the original image, resulting in Fig. 5c. We then used SimpleCV's findLines function with a minimum line length and line quality threshold.

The limited data set suggested a relationship between the pixel distance from the bottom of the image and the distance to the object. Moreover, the larger takeaway from this round of testing was that the segmentation quality between images affects the length and reported position of the lines, but the angles were consistently accurate. The pixel distance from the bottom of the image was better determined using the findBlobs function (groups similarly colored light pixels) on the segmented image than the findLines function. A more detailed analysis of the data can be found in this informal write-up.

C. Obstacle Avoidance Considerations

Initially, we were planning to implement an algorithm that allowed the robot to avoid any non-grass colored obstacle. We considered aiming for the highest visible point and calculating the turning angle as a function of the x and y position of the point we were aiming for using empirical testing (Fig. 6).

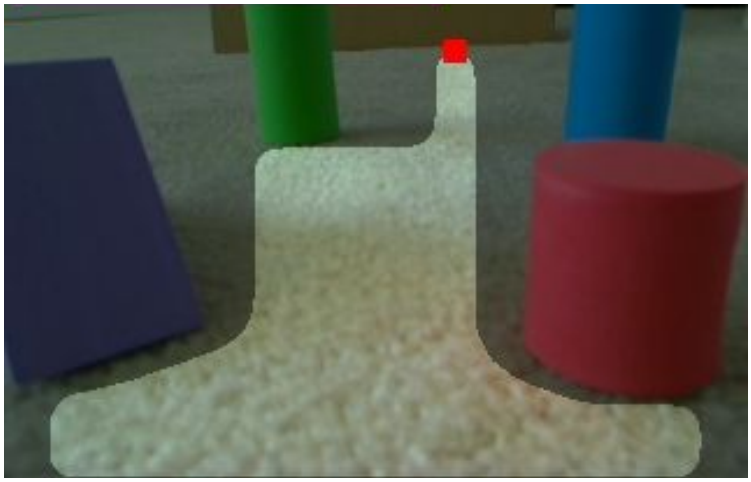

Figure 6. Obstacle avoidance algorithm using highest visible point [1].

The algorithm first segments the ground, then horizontally erodes the image in consideration of the robot's width. Since closer objects would require a greater horizontal pixel distance from the center for clearance than further objects, we conducted a new test to empirically determine a pixel location on the display for the robot to aim toward using the distance from the obstacle and the necessary clearance width.

D. Clearance Calibration

For this round of calibration, a distinctively colored blue box with a width of 4.625" and a depth of 4.375" was offset to the left of the robot with just enough clearance so the robot would not touch it if it moved straight forward, which corresponded to offsetting the edge 9" from the center of the robot. The box was placed at incremental distances and the coordinates of the visible bottom corners were recorded. The data is available in this spreadsheet. We found the relationship between pixel distance from the bottom of the screen and physical object proximity (Fig. 7) and between proximity and the pixel clearance width needed (Fig. 8).

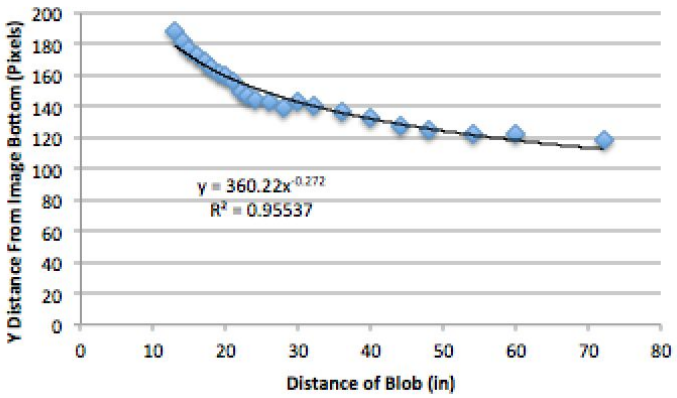

Figure 7. Relationship between pixel distance from bottom of image and physical proximity in inches.

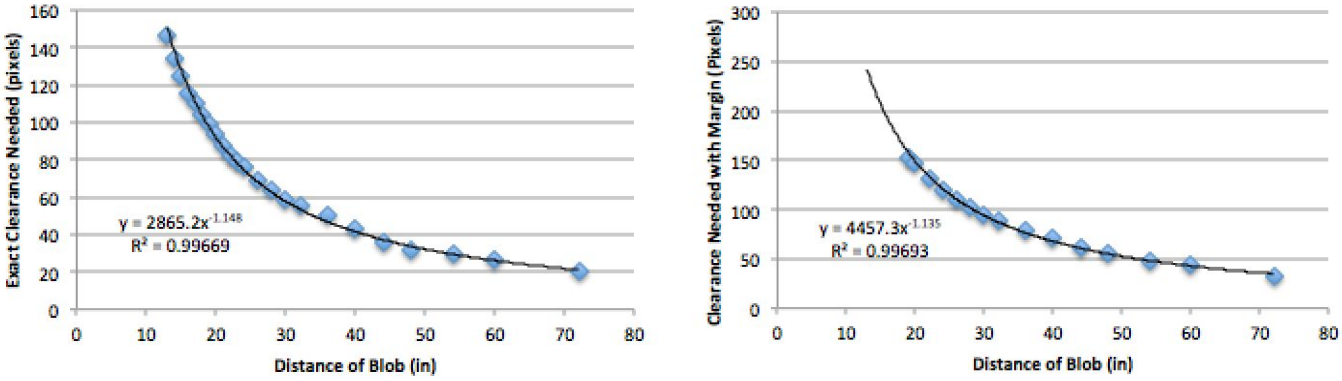

Figure 8. Relationship between physical proximity in inches and horizontal pixel clearance needed with and without margin of 4.375".

We wrote out the code to determine the pixel location and send it to the robot. Instead of using the highest point, the algorithm first filtered blobs that were valid obstacles by checking the area, proximity, and whether there was enough clearance to pass it. Then the algorithm considered the closest two obstacles and chose a pixel location to aim towards.

We refrained from pursuing the corresponding servo angle calculations for this method because of the newly apparent issue that the depth of the object increases the apparent width of the object in the segmentation. Another version of the code addresses this issue using the known depth of the base, but we decided to recalibrate our algorithm using the base itself, a smaller clearance width, and direct use of the servo angle. Furthermore, at this point, we realized our robot could easily clear the cardboard bumps from most angles. Therefore, we decided to ignore the cardboard obstacles and focus on moving toward the base.

E. Finalized Base Algorithm

The purple base was easy to segment using hue. We used the erosion function to remove noise from the binarized image and the dilation function to both join broken parts of the segmentation and as a factor of safety. The largest region returned by SimpleCV's findBlobs function applied to the segmented image was marked as the base, as long as the blob was larger than a specified area threshold (Fig. 9).

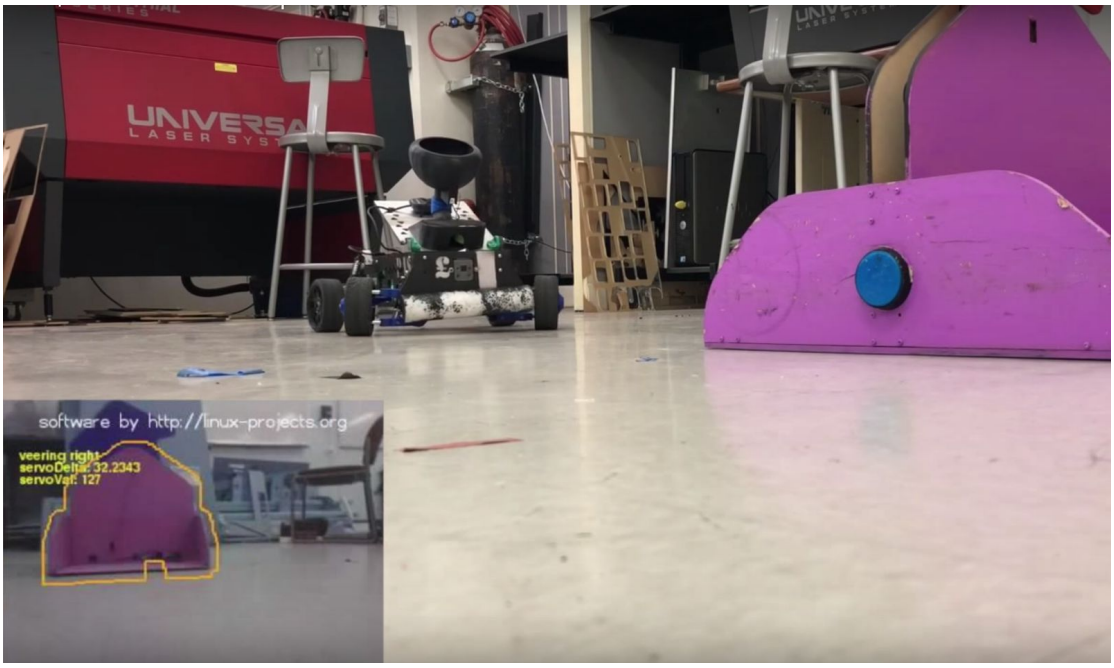

Figure 9. Still from video illustrating the real-time segmentation of the base.

If the base was over four feet away, as determined by the distance between the bottom of the frame and the bottom of the base, the algorithm focused on aligning the right edge of the base with 3 inches of clearance using a limited servo angle range. When the robot was closer to the base, the algorithm used a surface fit model to map the X and Y pixel bounds to the necessary change in servo angle (Fig. 10). The robot aims for the right of the base by default, but an alternate version of the code allows the robot to aim for the left of the base. Both the vision code and the Arduino code are available on GitHub.

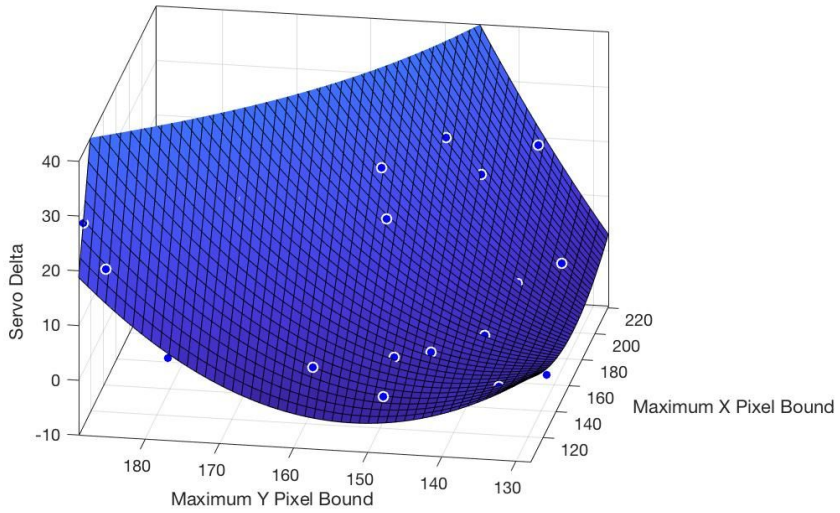

Figure 10. Surface fit of required servo angle to clear base for given maximum X and Y pixel bounds of base segmentation. The raw data is available in this spreadsheet. N = 27.

F. Finalized Ramp Algorithm

As mentioned earlier, for the ramp/seesaw computer vision strategy, we found that line detection could provide us with accurate angles, but the quality of the segmentation affected positioning and length. Since the vertical stripes get warped when translated from 3D space, we found it difficult to use only the angles of the vertical stripes to determine relative position and orientation. The horizontal stripe on the ground however remains horizontal whenever the robot is oriented straight. We decided to use this fact to pursue autonomy for the ramp. The algorithm involved turning until an orange horizontal line was visible for multiple frames in a row, then positioning the robot so it was within 2 to 3 feet of the base of the ramp. We planned on hard-coding drive settings to autonomously climb the ramp if we had enough time. Both the vision code and the Arduino code is available on GitHub.

A week before the competition, one of the two seesaws and one of the two ground bases were removed from the layout. With the adjusted map layout, autonomy for the seesaw would only be useful for half of the games and was more difficult to implement, so we decided to use the autonomous period to simply go straight. After observing the course in person the night before the competition, we realized that the ramp was positioned close enough to the starting region so that the ramp robot could move straight as well, so we focused on implementing computer vision for the robot pursuing the ground base.

Conclusions

We were not able to use computer vision on the day of the competition, ultimately because we didn't have sufficient time to test our systems, and we were prioritizing general mobility over autonomy. We realized half an hour before our first match that we didn't have the same wifi access on the field required for Virtual Network Computing (VNC), resulting in the need for regular access to the HDMI port on the Pi. This disrupted our plan for how we would secure the Pi, and the wires we had available weren't long enough for the alternative solution. Furthermore, the way the competition was designed and modified made the strategic advantage from the autonomous capabilities of our robot rather negligible. We defaulted to simply moving forward during the autonomous period. Despite not being able to use it during the competition, the learning opportunity from implementing computer vision for the first time was well worth the effort.

Avoiding a specific obstacle with known dimensions made the problem much easier to solve. In order to avoid general stationary obstacles, we would need to use OpenCV to get more information on the blob. For example, if we wanted to avoid any obstacle by veering right, we would need to know the lowest y-position of the blob at the maximum x-position of the blob. The clearance calibration code with additional servo angle calculations would be sufficient to implement this algorithm.

Supplementary Materials

Most of the following materials are referenced in the report. They are mentioned again here for accessibility:

- Videos
  - Computer Vision Demo
  - ME 72 Robot
- Code
  - Base Vision
  - Ramp Vision
- Additional Data Analysis
  - Colorspace
  - Seesaw Lines
- Raw Data Spreadsheets
  - Color Calibration
  - Clearance Calibration
  - Servo Angle (Ground Base) Calibration

References

[1] "Obstacle Avoidance," RoboRealm Available: http://www.roborealm.com/tutorial/Obstacle_Avoidance/slide010.php.