Smartphone-based Mobile Robot Navigation

Nolan Hergert, William Keyes, and Chao Wang
18-551 Group 3

***Abstract*—Physical systems are subject to noise and other environmental issues not present in most academic versions of these systems. To understand these issues, we created a mobile robot that can follow a white line, outdoors, in varying lighting and pavement conditions using the video processing and motion sensing capabilities of an Android smartphone. A custom vision algorithm takes as input the smartphone camera feed and outputs control data to a motor controller connected to the motors of a hobby radio-controlled vehicle. We competed in the eighteenth annual Mobot competition, completing the course and taking first place.**

I. Introduction

For the past eighteen years, on an afternoon in the spring, people gather near the sidewalk in front of Wean Hall to watch robots. Not any robots, but robots specially designed to complete a course running the length of the building along the sidewalk. This is the Mobot (from Mobile Robot) [1] competition, where groups of students and members of the community build autonomous vehicles that can drive through a series of fourteen gates in the correct order, passing over two steep hills in the process. The gates lie along a white line painted on the sidewalk which provides helpful guidance for the vehicles. See Figure 1 for a map of the course. Vehicles are ranked by how many gates they successfully clear and then by time in the event of a tie.

Though it may seem like a simple problem to solve with a robot, examining the event closely reveals subtleties that make it much more difficult. Primary among these is that the course is outdoors, whereas most robotics competitions take place in a carefully controlled and specified environment. The problems of line following changes from detecting a high contrast line on a consistent background, to detecting a noisy line in changing contrast on a noisy background, where parts of the line may be missing and parts of the background can be very similar to the target. Each Mobot must be able to overcome these problems of the performance surface under changing lighting and weather conditions.

Building a vehicle for the Mobot competition is an excellent way to gain practical experience dealing with noisy data and other constraints imposed by the need to create a physical, function system. We believe that this makes it an ideal candidate for an 18-551 project. It includes a significant signal processing component in detecting the line, basic elements of control theory, and requires integrating these parts into a electromechanical system. This integration component should not be underestimated, as many important signal processing problems are motivated by the needs of real devices. In building a Mobot, we hope to develop reliable, reusable solutions to the problems of noisy data and control and provide a foundation for future 18-551 groups attempting mobile robotics problems.

Submitted May 9th, 2012
Department of Electrical and Computer Engineering
Carnegie Mellon University

A. Terminology

We refer to the device we built interchangeably as the "vehicle", "robot", or "Mobot". The "line" is always the white line painted on the course. The "vision problem" is the problem of detecting the line. The "decision problem" and "decision points" refer to the final section of the course, from gate 8 to gate 14 where the line diverges into multiple paths and the robot must choose the correct paths in order to pass through the correct gates. A "split" or "decision point" is a point where the line splits into two lines. "Android" refers to the Android operating system, an open-source smartphone operating system developed by Google [2].

B. History

Because the Mobot event is a competition, there are few sources of detailed information about previous attempts, despite the events long history. However, general information is available, as are videos of most past competitions, and we used this information while developing our strategy.

While many groups start thinking about the vision problem in terms of how to solve the decision problem, we were aware that it is a more important problem to have reliable vision on the upper part of the course. Similarly, we were warned to solve the problem of control on the hills early, as this is a common point of failure. [3] In fact, following the single line and maintaining control on hills is so significant a problem that in many years no teams finish the course, let alone reach the decision points.

From video of previous runs, we saw that many groups choose to create their own base from scratch and design their own microcontroller and sensing system. These custom sensing systems are commonly a strip of infrared sensors pointed at the ground in front of the vehicle. While update speeds are very fast for these sensors, they need to be shielded from sunlight, a difficult problem with the changing terrain of the course. While we know several past teams have used camera systems with varying levels of success, we have no details of their vision algorithms. We also know that at least one previous group used an Android smartphone as the processing device. To the best of our knowledge, no previous 18-551 groups attempted similar imaging tasks.

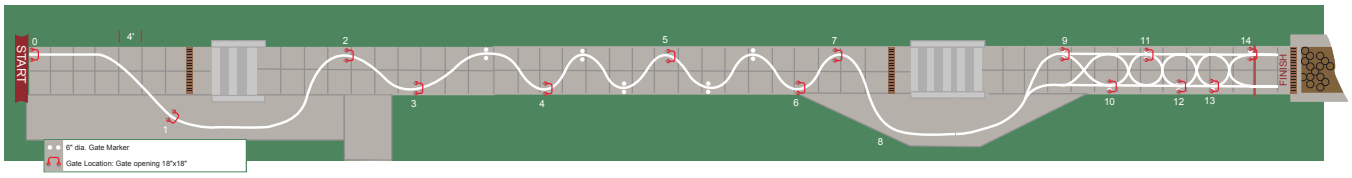


Fig. 1. The Mobot course [1].

II. System Overview

Our Mobot is made of four main components: an Android-powered smartphone, a USB interface board, a motor controller, and the wheelbase of a radio-controlled hobby model tank.

Given the unreliable nature of protecting infrared sensors from sunlight, we chose a camera-based approach. This was also influenced by the easy access to the built-in camera on our smartphone. Given our knowledge of signal processing we were confident that we could overcome any challenges that image processing might present.

We chose to build on the Android platform for several reasons. First, like most smartphone platforms, Android devices provide a powerful processor paired with a variety of sensors in a compact, energy-efficient format. For our purposes, this meant we would only need external hardware to handle output, not input. Android also provides an open, easy to use development environment and is compatible with the powerful OpenCV computer vision library [4]. Finally, Android was the chosen platform for the lab component of our course and we were encouraged to use it in our projects.

Android provides several options for interfacing with hardware. Until recently, these were limited to wireless technologies like Bluetooth and WiFi. However, in the last year, Sparkfun Electronics released a USB interface board called the IOIO ("yo-yo") and Google released an officially supported USB platform called the open Accessory Development Kit (ADK). Given the potential unreliability of wireless interfacing, we preferred a USB solution. Ultimately the IOIO proved to be the simplest and most easily available device that met our interface needs.

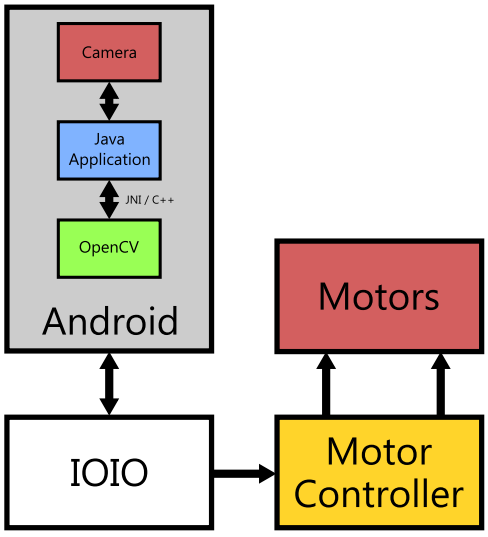


Fig. 2. High-level view of system architecture

Because no one on our team has significant mechanical engineering experience, we chose to modify an existing wheelbase instead of building our own. The tank base has excellent torque and traction at a reasonable speed. Although we would like a higher top speed, the torque provided by the base makes the hills in the course a trivial problem and successful traversal of the hills proved more important than speed.

III. Vision

A central problem to building a successful Mobot is developing a technique to identify and locate gates in the proper sequence, allowing the robot to traverse the course. While it is theoretically possible to identify the gates themselves, this significant computer vision problem can be avoided by instead developing a method to detect the white line painted through the course. By following the line, a robot is guaranteed to pass through all gates in the proper sequence, provided the correct lines are chosen at the final decision points. This is the approach most competitors use, and is the approach we adopted for our system.

This is not to say that the problem of detecting a line is simple. While finding a line in an image is easier than finding gates against widely varying background, recall that the detection must be invariant to local and global illumination changes, invariant to changing contrast between the line and the background, tolerant of missing line sections, and resistant to the textural noise of outdoor concrete. Further, the detection must run in real time on our Android device. Given our performance constraints, we found that many existing techniques from the computer vision literature were ill-suited for our application.

The line detection problem is fundamentally a segmentation problem: some section of an incoming frame is the concrete (background) and some section is the line (foreground). We must extract the foreground and use its properties to compute an error value for input to the control system.

The first and most basic segmentation technique we tried was binary thresholding. This quickly failed on our test scenes because light reflectance decays linearly with a decreasing angle of reflection, and binary thresholding assumes a constant threshold that will work across the entire scene. For example, a threshold that segmented the line at the top of the frame had too low of a threshold for the bottom of the frame, classifying everything as a line. Block-based adaptive thresholding, which computes a possibly different threshold for each pixel in the image, is a standard solution for this illumination problem. However, we found that an adaptive threshold not

only highlighted the line, but also enhanced the details in the background concrete texture. The failure of these basic methods lead us to research more advanced techniques, such as graph-cut algorithms, the Hough transform, and k-mean clustering. While these may have produced excellent results, we did not actually implement any for testing; the apparent complexity combined with a new insight into the nature of our problem prompted us to pursue a different approach.

The insight, which became a critical assumption underlying our vision system, is that when our robot is correctly oriented on the line, the line will always enter the camera's field of view at the top of the frame and exit at the bottom of the frame. This effectively eliminates one dimension of our problem; the two dimensional problem of finding a line region in the full image is now the one dimensional problem of finding horizontal position of the line in each row of the image. Additionally, we can assume that light reflectance will be constant across this horizontal line as the angle from the camera to the surface is constant. Horizontal positions are also natural predecessors to our chosen error value, the angle from the vertical of the line in our view.

Our first attempt at applying these ideas simply found the location of the maximum pixel value in each row of a down-sampled grayscale version of the input frame. This worked surprisingly well on our test images, but we soon realized it was quite susceptible to outliers caused by illumination changes and texture noise. With this in mind, we developed the final algorithm used in our Mobot.

A. Algorithm

For each grayscale frame, first blur the frame significantly, through a combination of downsampling to M x N and application of a box filter of size k_b. Next, extract L, $1 < L < N$ evenly spaced pixel rows from the blurred frame. Note that each row of pixels effectively contains information from adjacent rows in the original frame because of the blurring operation. For each row $\mathbf{r_i}$, compute the mean pixel value, $\bar{\mathbf{r}}_\mathbf{i}$. Rows are indexed from 0 to $L-1$, with $\mathbf{r_0}$ at the top of the frame.

Let $\mathbf{n_i} = \text{truncnorm}(\mathbf{r_i} - \bar{\mathbf{r}}_\mathbf{i}, 0)$, where $\mathbf{x} = [x_1, x_2, \ldots, x_N]$ and $\text{truncnorm}(\mathbf{x}, c)$ is defined as

$$\text{truncnorm}(\mathbf{x}, c) = \begin{cases} 0 & \text{if } x_i < c, \\ x_i / \max(\mathbf{x}) & \text{else} \end{cases}$$

Thus, each $\mathbf{n_i}$ contains the amount each pixel value was greater than the average intensity, normalized to lie in the range $[0, 1]$. We assume here that the white line will be the brightest section relative to the other pixels in the row. This generally holds, but can be violated by small shadows in bright sunlight that block the line and make the background brighter than the line. This failure case was not encountered in the course, but is important to consider.

Now, blur each $\mathbf{n_i}$ by convolving with a gaussian kernel of size k_g and variance σ^2 to produce a smoothed pixel intensity curve, $\mathbf{c_i}$. Let P_i be the set of the horizontal locations of local maxima (peaks) in $\mathbf{c_i}$. Peaks are found using a modified version of Billauer's procedure [5]; we have no need to report local minima, so this information is not stored. We expect that each peak is approximately centred on the intersection of the row and the target line. When there are multiple target lines that intersect a row, as is the case during the decision point section of the course, we expect to find a peak for each line, assuming k_b, k_g, and σ^2 are appropriately set to avoid merging unique lines.

Subject to the validity constraints discussed in the next section, we now compute an error angle using the peak locations. Let P_i' be the possibly empty set of valid peaks for row i. Note that in the multiple line case, P_i' is empty if P_i has only one peak. If P_i has multiple peaks, P_i' will contain the leftmost or rightmost peak dependant on the desired turn direction. Define

$$P = \bigcup_{i=0}^{N-1} P_i'$$

to be the set of all valid horizontal locations. Compute $\bar{P}$, the average horizontal position. The error angle is defined as

$$\theta = \frac{\pi}{2} - \tan^{-1}\left(\frac{y_{\text{mid}} - y_{\text{anchor}}}{\bar{P} - x_{\text{anchor}}}\right)$$

where $x_{\text{anchor}} = N/2$, $y_{\text{anchor}} = 3M/L$, and $y_{\text{mid}} = M/2$. We subtract the tangent from $\pi/2$ because we require the angle from the vertical, not from the horizontal. This angle finding procedure is shown in Figure 3.

While this angle computation technique proved effective, it is important to note that it is not theoretically ideal. When the robot is heading off course, the line will appear near the edges of the frame and we would like the corresponding error value to also be large. Unfortunately, because the anchor point and the vertical separation used to compute the angle are fixed, the angle changes slower when $\bar{P}$ is near the edges of the frame than when $\bar{P}$ is near the center of the frame. This could possibly avoided by using a peak in P_{N-1}' as the anchor

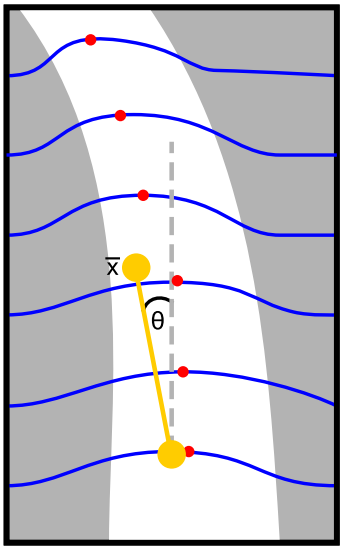


Fig. 3. Angle computation. The angle, θ, is defined as the angle from the vertical of the line from a center point near the bottom of the frame to the average horizontal component, $\bar{x}$, located at half the frame height

instead of a fixed value. However, because the fixed-point method worked in practice, other methods remain untested.

B. Rejection

Because we select multiple lines from our image, there is redundancy in our representation of the target line location. This allows us to discard peaks that are likely the result of noise without reducing the accuracy of our error angle.

Before any further processing, we mark rows $\mathbf{r}_{L-L_0}$ to $\mathbf{r}_{L-1}$ as invalid. These rows are too close to the front of the robot to be useful in computing the error and likely have interference from the robot's shadow. The first criteria we consider for the remaining rows is the maximum to mean ratio. Let $mmr_i = \max(\mathbf{r}_i)/\bar{\mathbf{r}}_i$ for $i = 0 \ldots N-1$. If $mmr_i > \alpha_{mmr}$ then the peaks in P_i are valid. This check is based on the idea that the maximum value in a row intersecting the line will be significantly higher than the average value of the row, in comparison to the maximum and average values of a row that does not intersect the line. Also, because this is a ratio, the threshold can be set independent of illumination.

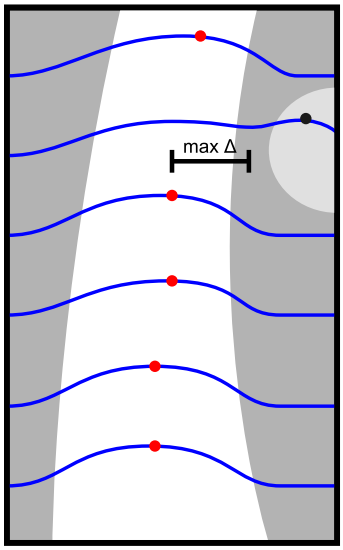
Second, we ensure that peaks that meet the ratio criteria have realistic relations to each other in the frame. Assuming that P_i contains only one point for all i, compute $\Delta_i = |P_i - P_{i+1}|$. If $\Delta_i > \Delta_{\max}$, then mark P_{i+1} as invalid. Formally, this constrains the maximum slope that the line can have in the frame, but practically it rejects single outlier peaks, most commonly caused by the white paint circles that mark potential gate locations, as shown in Figure 4a.

A similar method is used to reject incorrect splits when rows have multiple peaks. We note that the dual peaks representing a valid split will always move together from the top of the frame to the bottom of the frame. That is, if a split is in the field of view, it is split at the top of the frame and joined at the bottom. The inverse split, depicted in Figure 4b, exhibits divergence while moving from the top of the frame to the bottom. Therefore, when we have two peaks per row, if $\text{dist}(P_i) < \text{dist}(P_{i+1})$, where $\text{dist}(P_i)$ is the distance between the two peaks in P_i, the split must be invalid and we consequently mark all dual peaks invalid.

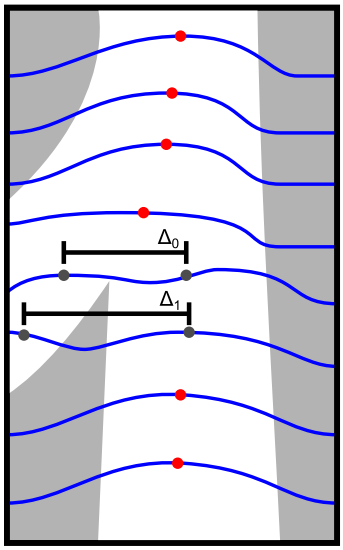
Lastly, after the previous rejections measures are applied, we require at least V_0 rows to have valid peaks before computing a new angle. If this threshold is not met, the previous error is maintained, allowing us to potentially rediscover the line.

C. Implementation and Performance

The algorithm is implemented in C++ and uses functionality from the OpenCV library. While it is unclear if a C++ implementation is necessary for performance, we found that C++ and the OpenCV C++ API allowed for cleaner implementations of our algorithms when compared to Java, which was used to implement the remainder of our Android application. Our implementation runs as fast as the camera frame rate will allow, making explicit optimizations unnecessary. As a general principle, we tried to avoid allocating and copying memory whenever possible to keep our implementation performant.



(a) Slope constraint



(b) Inverse split rejection

Fig. 4. Two rejection methods. If the distance between peaks in adjacent rows is greater than $\Delta_{\max}$, mark the peak as invalid (a). In an invalid split, dual peaks diverge while moving from top to bottom, meaning $\Delta_1 > \Delta_0$ (b).

IV. Control

In order to be useful, error angles produced by the vision algorithm must be converted into motor commands that can correct the error; this is the purpose of a control system. Because we are controlling a tank, steering is achieved by running the treads on one side of the tank slower than or in the opposite direction of the treads on the other side. The tank will turn towards the side running slower or backwards.

We use a proportional derivative (PD) controller, a simplification of the more general proportional integral derivative (PID) controller, for our Mobot. The proportional term maps the error angle directly to motors speeds. In other words, it is a unit conversion term. This worked well on gradual curves, but was not sufficient for the sharper curves near the end of the course without over-steering. To correct this,

we introduced the derivative component, which corrects for a smaller proportional term by amplifying errors that change quickly. We had no use for an integral component, because our system has very little steady-state error; the line is rarely straight, meaning error is never constant. Our implementation is based on that described by Wescott [6].

V. Hardware

A. Electronics

Our application requires a camera with a reasonably high frame rate (15 - 30 fps) and an ARMv7 processor running at 1 GHz or faster. The ARMv7 platform includes hardware support for floating point operations, which is critical for the performance of our filtering operations.

We used a Samsung Galaxy S II for all processing because it was available to us and exceeded our requirements. In fact, our algorithm was limited by the camera shutter speed rather than processing time. We also tried running our algorithm on a Motorola Droid and Samsung Nexus S, where we were limited by processing time rather than camera parameters. As there were no incentives to perform well on these older and slower devices, no attempts were made to optimize our implementation.

The IOIO interface board connects to Android devices using USB and a custom protocol build on Androids USB debugging capabilities. An open-source Java library handles communication from within an Android application, providing a simple interface for programmers. Standard function calls turn digital pins on and off and control the pulse-width modulation (PWM) pins used for motor speed control.

We use a L298 H-Bridge [7] to allow us to control the high-current, high-voltage motor load with the low-current, low-voltage pins on the IOIO. The h-bridge also provides easy methods to change motor direction or enable electronic braking.

B. Vehicle Bases

Finding an appropriate vehicle base for our Mobot proved more difficult than we expected; the final tank base was the third base we tested. In choosing a base, we were concerned primarily with available torque, traction, and top speed. High torque and traction are necessary to smoothly navigate the hills in the course. Speed was important because we hoped to not only finish well below the maximum allowed time, but to beat the previous undergraduate course record.

With speed in mind, the first base was a modified hobbyist radio-controlled (RC) car (Figure 5a). Designed for racing, the car's top speed far exceeded the maximum speed we would be able to control on the course. However, the motor and gear system provided little torque, which meant that the only reliable way to slow the car was to drive the motor in reverse. This was required on the hills to avoid accelerating out of control while turning at the bottom. Under manual control, this worked fairly well, but we could not automatically determine when to do this braking procedure reliably.

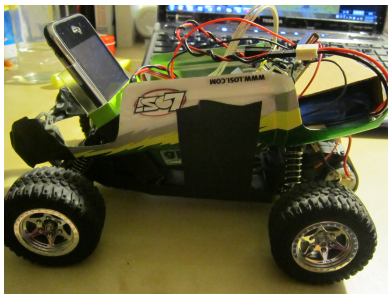
We attempted to determine when to brake by measuring the change in angle of the vehicle relative to its rest position. Most smartphones have a built-in accelerometer to detect screen rotations and we tried to use this capability to detect our declination. Unfortunately, the noise caused by acceleration down the hill and bumps in the terrain made it too difficult to separate the acceleration component due to gravity. We attempted to use the gyroscope in the phone to augment and correct the acceleration data, but had difficulties integrating the two sensors. A more straightforward option for controlling speed is to simply measure wheel speed using rotary encoders. However, given our time frame and the fact the attaching the sensors required modifying the vehicle, we did not try this method, and instead moved to a different base.

Our second platform was a basic tank, with two wheels on each side connected by a belt. This provided the torque and stability we needed while having a reasonable top speed. On further testing, we found that the treads would come off the wheels when the robot made sudden turns; the treads were not designed to handle the increased friction of skidding on rough pavement. In an attempt to keep the treads on, we tried driving the inside tread backwards instead of simply driving it slower. This allowed the treads to stay on for longer, but we were unable to complete the entire course consistently.

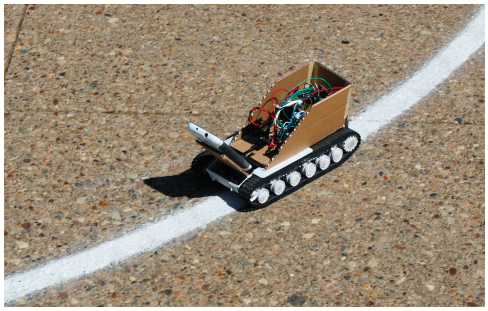
With the tread problem unsolvable and the competition date approaching, we switched to a third tank base (Figure 5b), a modified hobby RC tank. This base has an authentic tread system, including tensioning wheels, and has proved very reliable with no more mechanical issues. Despite a further reduction in speed, this base allowed us to test our full system on all parts of the course, which proved crucial to our success.

VI. Testing

Developing a robust system required testing in as many different conditions as possible. To this end, we tested our



(a) First car base



(b) Final tank base

Fig. 5. Two of the three vehicle bases tested.

Mobot in varying levels of sunlight, on cloudy days, in the rain, and briefly in the snow. Because the competition is held from noon to 3:00 pm, we focused on testing in the 11:00 am to 3:00 pm window, in an effort to duplicate the conditions we would compete under. We were also able to test at night, thanks to the addition of a headlight on our vehicle. Performance was similar under all conditions.

The system was designed from the beginning for ease of testing. To this end, all important vision and control parameters are settable dynamically using the touchscreen on the smartphone. This saved valuable time while determining the correct parameters and allowed us to adjust to new condition rapidly. We also endeavoured to keep the hardware connections non-permanent so that the electronics could be easily moved from one base to another. As we switched bases, we spent minimal time getting the new base functional, leaving more time for performance testing.

VII. Schedule

The project was completed in the second half of the spring 2012 semester. While work was split between our three team members, Nolan Hergert focused on the control and vision algorithms, William Keyes focused on the vision algorithm and Android application, and Chao Wang focused on vehicle design and logistics. See Table I for an approximate schedule.

VIII. Results

With the car base, we competed in the mini-slalom challenge, which requires completing gates 2 through 8 of the course, the section between the two hills. We took first place in the challenge, finishing the section in 36.74 seconds. The following week, we competed against ten other teams in the full competition. We completed all fourteen gates in 2 minutes, 13.84 seconds, taking first place again. We were also the only group in the 2012 competition to complete the course and the third-ever undergraduate team to do so.

TABLE I
Schedule

Week 1	Research image processing techniques, ADK, IOIO, and robot hardware
Week 2	Develop image segmentation, data collection, reach consensus on hardware
Week 3	Develop path extraction and improve segmentation, data collection, bontrol hardware with IOIO
Week 4	Improve extraction and segmentation, begin porting to Android/OpenCV, improve hardware control
Week 5	Develop basic control algorithms (steering and hills), data collection, integrate hardware control into Android application
Week 6	Tweak control algorithms, integrate all hardware on chassis. Begin testing full system
Week 7	Switch to first tank, testing, debugging, and improvement. Compete in Mini-challenge
Week 8	Switch to second tank, develop decision-point strategy, testing and debugging. Compete in Mobot Races

Despite our standing in the competition, our Mobot was not completely successful. In particular, the vision algorithm consistently failed to detect the decision point immediately following gate 10. We suspect that the shadow of the gate flag or misapplied paint are the cause of this failure, but we were not able to determine a definite cause. Instead, we assumed that our algorithm would miss this split and then re-programmed the desired turns so that our vehicle would loop around, driving through the backside of the next gate, and then continue the course to completion. Surprisingly, this worked, with the algorithm only missing the split at gate 10 and successfully detecting all other splits.

IX. Conclusion

We succeeded in developing and construction a mobile robot that can reliably follow a white line under the conditions of the annual Mobot competition. In doing this, we significantly increased our understand of signal processing systems that deal with non-ideal, noisy signals. While our robot and vision algorithm did not perform perfectly, we believe the split detection failure can be easily solved with some additional development and are satisfied with the error rates of our system, particularly when compared to other entrants in the competition.

For future 18-551 groups, our experience developing vision algorithms for Android devices and constructing mobile robot platforms should be a helpful starting point. We do not expect our vision algorithm in whole to be built on, but hope that parts of it can be adapted to help with similar illumination problems in different domains. At minimum, we have demonstrated the feasibility of a mixed Java and C++ vision algorithm that is performant on modern devices. Further, we hope our failures in developing a reliable vehicle base will be instructive for groups that attempt mobile robotics projects.

Given our successful attempt at completing the Mobot course in a reliable, if traditional manner, we are excited by the possibility of competing again in the future and completing the course faster or in a more exotic fashion.

Acknowledgment

We would like to thank Professor Marios Savvides and his lab for lending us the smartphone and electronics needed to complete this project and Professor Thomas Sullivan and the 18-551 course staff for their support.

References

[1] "Carnegie Mellon School of Computer Science Mobot Race," http://www.cs.cmu.edu/mobot/.
[2] "The Android operating system," http://www.android.com/.
[3] J. S. Palmisano, "Robot competitions - Mobot," 2008. [Online]. Available: http://www.societyofrobots.com/competitions_mobot.shtml
[4] "Open-source computer vision (OpenCV) library," http://opencv.willowgarage.com/.
[5] E. Billauer, "peakdet: Peak detection using MATLAB," http://billauer.co.il/peakdet.html.
[6] T. Wescott, "PID without a PhD," *Embedded Systems Programming*, October 2000. [Online]. Available: http://www.eetimes.com/ContentEETimes/Documents/Embedded.com/2000/f-wescot.pdf
[7] "L298 data sheet," http://www.sparkfun.com/datasheets/Robotics/L298_H_Bridge.pdf.