Title of Project: DOGBOT, A Robot Responding to Laser Stimulus
Team Members: Andrew Levin, Shiva Mehta, Jonathan Zarger

I. Executive Summary

DOGBOT is a robot designed to interact with a laser pointer. It was inspired by the way dogs and cats will try to chase after and catch lasers. DOGBOT uses sensors to track the laser and follow it within a 4’x6’ arena while avoiding colliding into the walls of the arena. DOGBOT chases a green laser pointer at about 5 feet per second.

Image processing consisting of laser detection and obtaining physical distance coordinates of the laser point relative to the robot was the heart of this project. Images were captured by a Raspberry Pi camera and masked for green pixel values. The masked image was thresholded and the pixel coordinates of the center of the laser point were found. These coordinates were then passed through a perspective transform to obtain physical distance coordinates that could be sent to a microcontroller that actuated robot motion. All of the image processing was performed on a Raspberry Pi 3 and programmed using Python and OpenCV.

The robot implements standard digital control algorithms to move towards the laser pointer. These algorithms use feedback and state estimation to achieve this goal. The algorithms provide the robot with an effective and robust way to chase the laser pointer at a high rate of speed.

DOGBOT has many important pieces of hardware. It uses a Raspberry Pi 3 single board computer as a main processor. This device is used for image processing and communications. It allows the robot to find the laser. This information is then passed to an Atmel SAMD21 microcontroller. This microcontroller interacts with many other hardware components to chase the laser. These other hardware pieces will be described later in the report. Working in unison with software, the robot’s hardware components allow DOGBOT to chase new laser locations every 100 milliseconds.

The original project goal was the same, with some extra additions of scope. The team planned to have DOGBOT interact with both a green and a red laser pointer. Green would initiate a chase behavior while red would initiate an avoidance behavior. Other goals were to add obstacles to avoid within the arena and to make the arena reconfigurable. Unfortunately, we did not have time to implement these goals. In terms of the milestones our team set, we were a week and a half late on meeting our first milestone. However, we met our second milestone on time and were ready for Design Expo a week in advance.

At Design Expo, DOGBOT was presented within a 4’x6’ black arena. Visitors were given the green laser to point inside the arena and observe the robot chase the laser and act like a dog.

II. Project Description

This section will discuss the technical details and decisions that made DOGBOT work successfully. This discussion includes identifying the hardware that made everything possible, describing the image-processing steps required to detect the laser, and discussing the development of the controls algorithms that actuated the robot motion.

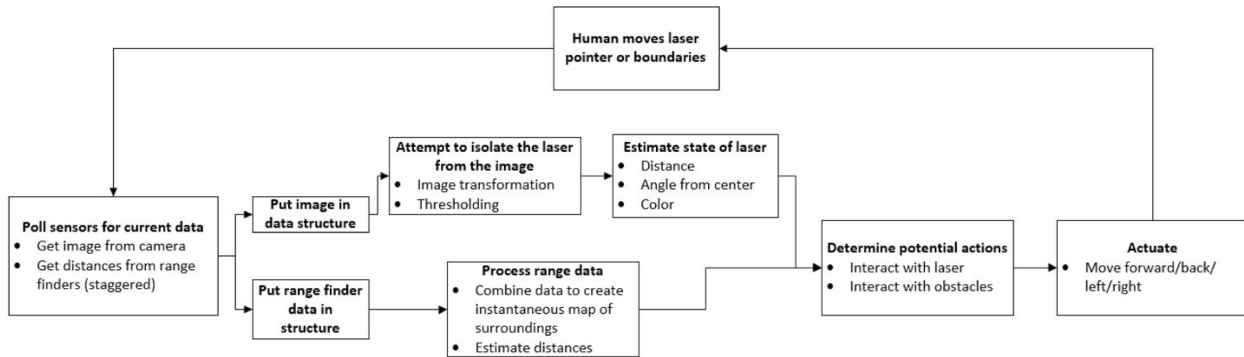

Figure 1: Project High Level Block Diagram

DOGBOT is equipped with two main hardware components: a Raspberry Pi and an Amtel SAMD21 microcontroller. The Pi finds the laser while the microcontroller steers the robot to it. Using a Raspberry Pi camera, the robot continuously captures images of the area in front of it. Image processing algorithms implemented in Python on the Raspberry Pi detect the laser point's center location. These center pixel coordinates are then converted into a physical location relative to the robot. This relative location is then sent to the microcontroller from the Pi. The microcontroller does not only receive data from the Pi, but also from infrared sensors. The infrared sensors mounted on the front of the robot detect how far the front of the robot is from the nearest walls. This information is used to prevent the robot from crashing into its surroundings. Feedback control algorithms implemented on the microcontroller use the laser point location and the infrared data to steer the robot towards the laser point. The microcontroller sends signals to motor controllers that actuate the wheels. In addition, rotary encoders in the motors allow the robot to estimate wheel speed, which is fed back into the control algorithm to help it steer. A high-level overview of the functionality of DOGBOT is shown in Figure 1 above. A picture of DOGBOT is shown in Figure 2.

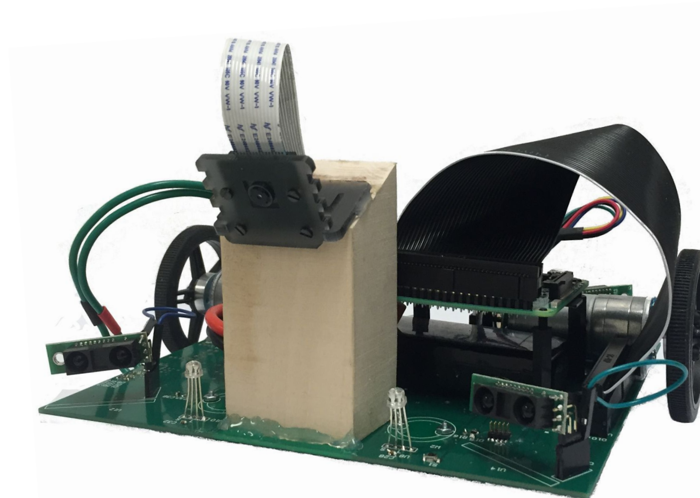

Figure 2: DOGBOT Robot System

Laser Detection and Image Processing

DOGBOT would not be able to chase the laser without efficient image processing algorithms. Image processing algorithms enable the robot to detect the laser and use that information to determine the physical location of the laser relative to the robot. The image processing algorithms were implemented on the Raspberry Pi using OpenCV [1] and Python.

The way the algorithms were developed, it was essential that the laser was directed on a non-reflective black background. In choosing a non-reflective black background, any glare and reflection is minimized. This makes image processing easier, especially in the case of thresholding and filtering certain colors and intensities. The image processing was tested on backgrounds of various colors, and they did not provide results that were as good as those obtained on a black background. To ensure that the background was completely black, a camera mount angle of approximately 45 degrees was used so that the robot could not see anything outside of the arena it was constrained within. Another design choice for the image processing algorithms was to use only a green laser. Creating algorithms that worked only for green lasers eliminated additional noise due to different colors. Overall, the design choice of using a green laser on a black background made image processing much easier to perform.

For all attempted image processing implementations, the Raspberry Pi camera captured an image in the RGB (red, green, blue) pixel color space. The RGB image is then converted to the HSV colorspace. Converting to the HSV color space enables one to manipulate the image based on the color (hue), the intensity of the color (saturation) and the brightness (value) of each pixel in that image. Each category; H, S, and V have values ranging from 0 to 255. For hue, the range is broken up into different colors. For saturation, 0 corresponds to white/faded/low intensity and 255 corresponds to intense colors. For value, 0 corresponds to low brightness or dark pixels and 255 corresponds to bright and white pixels. Consequently, processing the image captured by the camera in the HSV color space provides numerous advantages in the case of a bright green laser pointer on a black surface.

Contours

The first implementation of laser detection did not work perfectly. This implementation took the image captured by the Raspberry Pi camera and converted it from the RGB color space to the HSV color space. Once the image is converted to HSV, the image is masked using green HSV values. Masking the image as such turns any pixels that do not fall within the mask black. For this case, the green masks HSV values were from 60-90 for Hue, 50-255 for Saturation, and 100 to 255 for Value. The masked image was then converted to grayscale to obtain only white and black pixels that could be thresholded. Before thresholding the grayscale image, the image was blurred to smooth out the spot that corresponded to the laser and to filter out excess noise.

An image in grayscale has pixels with values between 0 and 255, where 0 corresponds to black and 255 corresponds to white. The argument is that once the masked HSV image is converted to grayscale, the spot corresponding to the laser point will have values closer to 255 or white in grayscale. Thus, the thresholding in grayscale was tuned to range from from 127 to 255. This way, the darkest pixels were turned absolutely black and only the whitest pixels remained.

Next the contours function in OpenCV was used on the thresholded grayscale image. Contours connect all the pixels that have similar (color and intensity) values. The idea was that the white area corresponding to the laser point would have the same contour. This area could then be averaged and the center of the area could be calculated using the moments function in OpenCV. The moments function finds the center of a given area of the thresholded contour image. If everything worked perfectly, this algorithm would be able to accurately pinpoint the center of the laser point in the image captured by the Raspberry Pi.

This image processing method using contours, however, failed to provide accurate results with good consistency. The method was quick to run and worked on some cases of a steady laser point and a moving laser point. However, the method failed to completely eliminate noise and was sensitive to contours and variations in the black background/walls on which the green laser point was located. As a result, the moments function averaged many more pixels than just the pixels corresponding to the laser pointer. This miscalculation of the image's moments resulted in detecting the center of the laser at the wrong location. Due to the inconsistency of this method, obtaining physical distance coordinates of the laser was not attempted. Figure 3 shows three images that were processed using the contours method, excluding the grayscale. The red dot shows the calculated moment for the image. For the left two images the algorithm works well, but the excess noise in the rightmost image results in a miscalculation of the moment.

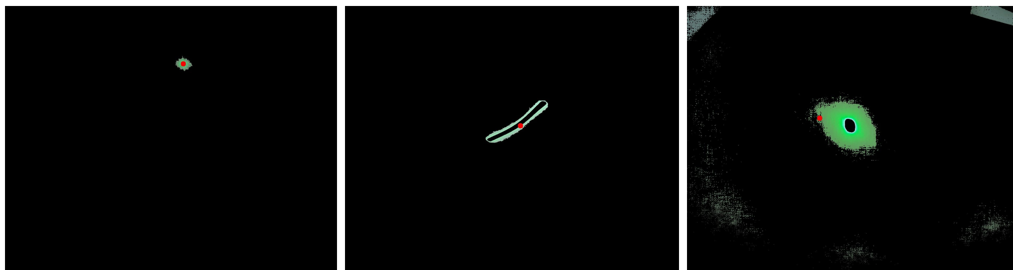

Figure 3: Image Processing Algorithm Using Contours

Blob

The second algorithm implemented and tested was the blob detection function in OpenCV. This algorithm works by detecting groups of connected pixels - called blobs - with grayscale values that fall within a specified range. The programmer chooses the range of grayscale values that all blob pixels must be within. Blobs are not only detected if all of their pixels share a range of grayscale value. One can also choose to filter out blobs based on their color, size, circularity, convexity, or level of elongation. Filtering by color does not appear to work. The detected blobs were the same with or without the color parameter. It is worth noting that it was not necessary to filter by all five parameters. In fact, one can opt to filter by none of these parameters, but too many blobs will be detected. After selecting which parameters to use to detect the blobs, minimum and/or maximum values of these parameters are selected. For example, if one wants to find all large circular blobs, they might choose to process the image by looking for blobs whose pixels have grayscale values between 20 and 200 (on a scale of 0 to 255). Then they could choose to look only for the blobs with a minimum size of 1500 pixels and least 90% circularity. Then the algorithm creates a binary image. White pixels in these images correspond to potential blobs. Any potential blobs with less than 1500 pixels are eliminated. Additionally, only blobs that meet the minimum circularity of 90% are kept. Lastly, the center and radii of each blob is calculated.

During initial testing, blob detection worked about 80% of the time filtering only by circularity. More than one blob was never detected, only one or none. The algorithm was performed on captured images after converting them to the HSV color space. However, blob detection took much longer to execute than contour detection. Blob detection alone took roughly 150 milliseconds for each image. This algorithm would be too slow to provide the robot with new laser coordinates at a desirable rate of roughly 10 Hz. For the 20% of the time that blob detection did not work, the HSV images were passed into the contour detection function described earlier. After running this two part algorithm on a batch of test images, the implementation appeared robust enough to make up for its lack of speed. The laser was always detected and detected in the correct position. It appeared that the test cases on which contour detection failed were always solved by blob detection. Likewise, images that blob detection failed on were always correctly processed by contour detection.

After building the robot's arena - its floor and boundaries, this new implementation was retested in a strongly lit area. During these tests, the code failed miserably. This discrepancy between test images and testing on the actual surface was due to the imperfections and texture of the surface. The algorithm began detecting tens of tiny blobs all over the captured image. Then after filtering to only find blobs of minimum pixel size, no blobs would be detected. Many combinations of the possible parameters were tested in hopes to find a combination that consistently detected the laser, but these tests were unsuccessful. Every combination either led to multiple tiny blobs being detected or no blobs at all. Likewise, when removing blob detection and only using contour detection, the algorithm performed poorly as it had before. This testing lead to rewriting the image processing code once again.

Actual Implementation

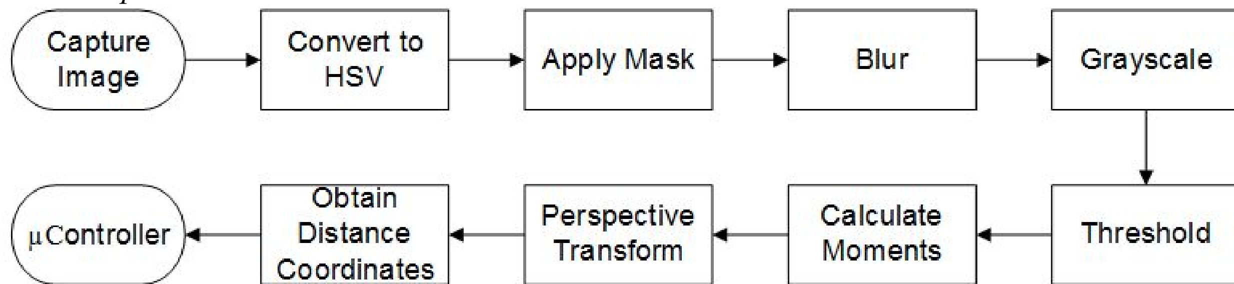

Figure 4: Image Processing Steps for Final Implementation

The major steps of the actual implementation of the image processing algorithm are shown in Figure 4. The core of this method was similar to the way the algorithm using contours was implemented. The RGB image was converted to HSV and then masked for Green. This time, the parameters of the green mask were slightly different. The green mask had a saturation range of 20 to 255 and a value range of 100 to 255. Increasing the saturation range helped capture more of the center of the laser point on the black surface. The masked image was then blurred, converted to grayscale, and thresholded. The thresholding in grayscale was tuned so that the range was increased to 20 to 255. At this point, there was almost no noise in the image and only the laser point remained. Instead of using contours, the thresholded grayscale image was averaged, The average X and Y coordinates (moments) of the white pixels are then calculated. These calculated coordinates serve as the center of the detected laser point in the image.

Figure 5 shows intermediate stages for two captured images as they are processed. The top row of images correspond to a static laser point and the bottom row of images correspond to a moving laser point. On the far left are the original images. Second from the left are the images converted to the HSV color space. Next the images are shown after masking the images for green color. Lastly, the far right pictures illustrate the image state after the final steps of the image processing. These last images have been blurred, converted to grayscale, and thresholded. What remains is a binary image with only black and white pixels. The secondary purpose of Figure 5 is to show that the final implementation of image processing works effectively for both a non-moving laser point and a moving laser point.

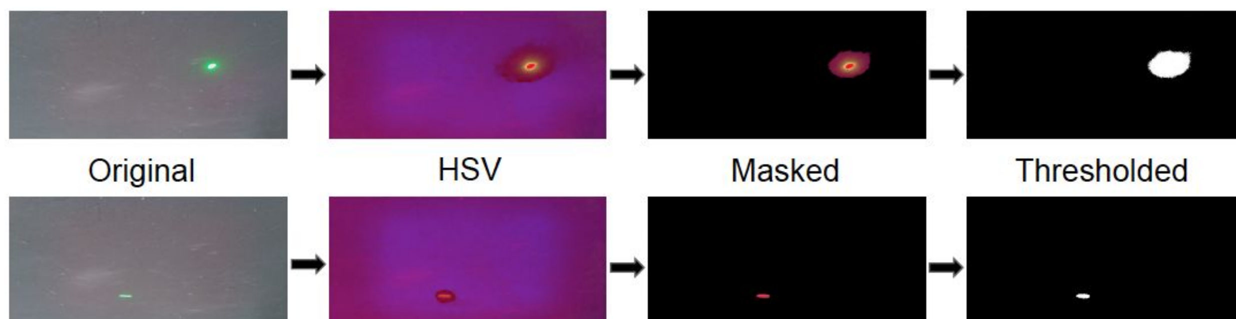

Figure 5: Intermediate Stages of Image Processing

Pixel Coordinates to Real World Coordinates
After locating the pixel coordinates of the center of the laser, the robot can convert them into physical coordinates relative to itself. Pixels at the top of the image correspond to locations further away and pixels at the image bottom correspond to locations right in front of the robot. Additionally, pixels on the left half of the image were captured when the laser was to the left of the robot and vice versa for the right side.

The pixel coordinates resulting from the image processing could not be used directly for DOGBOT to navigate. They had to be transformed into coordinates relative to the robot, as shown in Figure 6. The coordinate system/frame of reference of the image captured by the Raspberry Pi camera is shown on the left-hand side of Figure 6. The right-hand side shows the transformed, birds-eye view of the same image. This image illustrates that the robot's view does not correspond to a perfect square. In fact, the two bottom corners of a captured image are much closer together to each other than the two top corners are to one another. Additionally, a pixel at the top center of an image is much more than twice as far away from the robot as a pixel halfway up and in the center. The bottom corners of the captured images corresponded to locations roughly four inches from one another while the top corners represent spots about three feet from one another. In order to convert the pixel location to coordinates relative to the robot, a perspective transform was used.

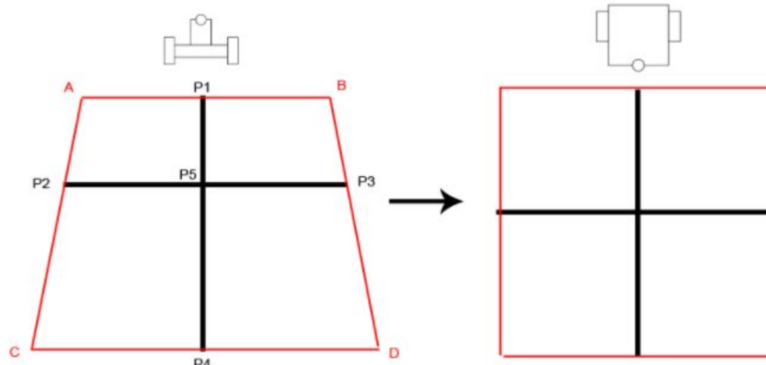

Figure 6: The Perspective Transform as it Applies to DOGBOT

In a perspective transform, one frame of reference is converted to another frame of reference. The transform is executed by applying a matrix function to points in one frame of reference to obtain points in another frame of reference. In order to obtain the matrix used to make these linear transformations, the OpenCV *getPerspectiveTransform(source, destination)* function was used. The first argument is a four coordinate array of values in the original (image) frame of reference. The second argument is a four coordinate array in the destination frame of reference.

This function was executed by passing in the image corner pixels for the first array and corresponding physical locations relative to the robot for the second array. The coordinates for the second array were determined by placing four red markers in front of DOGBOT's camera until each marker occupied one corner of the image. From the front center of the robot, a tape measurer was used to find the corresponding physical distance in the X and Y directions. These

measurements constituted the coordinates for the second array. The function then returned a matrix that could be used to transform all pixel locations. Then for each image, matrix multiplication was performed on the laser point center coordinates with this predetermined matrix. These transformations provide physical distance coordinates that the robot can use to move towards the laser.

Optimization Through Multiprocessing

In order to increase the rate at which the robot captured and processed images, the Python code takes advantage of multiprocessing on the Raspberry Pi. Multiprocessing allows for code to execute in parallel (simultaneously) on multiple processor cores of the same computer. The Raspberry Pi has a quad-core processor, so the code could be broken down into a maximum of four truly parallel processes. Data can be passed from one process to another using pipes. Fortunately, the Raspberry Pi code could be usefully abstracted into four parts. The first process captures the image. This captured image is then piped into the second process where the pixel coordinates for the laser's center are calculated. These coordinates are then passed to the third process where the pixel coordinates are transformed to coordinates relative to the robot. The last process packages the coordinates into a packet and sends the packet to the microcontroller over UART. This process is pipelined. While a new image is captured; the previous image is undergoing image processing, the image before the previous one is undergoing the linear transformation, and the picture from three cycles prior is currently having its data packaged and sent to the microcontroller.

At first, the image capture stage could only capture images at a rate of slightly more than two image per second. This frame rate was far too slow. In order to speed up the capture process, a parameter was added to the function *capture()*. By simply including *use_video_port* and setting it to true, the frame rate increased to over twenty frames per second.

The second stage of the Python code - the laser detection - was the most time consuming process, taking slightly under 100 milliseconds. By placing all the other processes in parallel, the total time taken to capture an image, process it, and send its location to the microcontroller averaged out to about 100 milliseconds. Without multiprocessing, it would have taken about 200 milliseconds to capture the images, process them, and send the data.

Robot Movement and Controls

This section will detail the software that runs on the chassis microcontroller. This code is the digital control software that manages the behavior of the robot. It handles the laser pointer coordinates transmitted from the image processing state, drives the wheels, and avoids the walls. All of the major hardware components mentioned in this section will be described in further detail in the following section on hardware selection.

Hardware and Software Platform

Controlling the robot's movements is performed on an Atmel SAMD21 microcontroller. It is the same microcontroller used in the Arduino Zero series or the Arduino MKR1000 series of microcontroller breakouts. It runs software through the Arduino platform, which uses a language that is a variant of C/C++. This platform abstracts the most convoluted aspects of embedded systems, which primarily includes the serial interfaces and clocks.

To receive data from the Raspberry Pi, a simple packet structure is used. The microcontroller looks for a valid start of packet and end of packet marker, and reads the data bytes in between. There was also support written into the packet structure for a checksum in the event of poor signal integrity, but this potential problem was never an issue. A checksum is a unique value generated by the contents of a packet, and can be used to check the validity of data on both ends. When a valid packet is received, the laser pointer coordinates are reconstructed from the packet and are set as a global variable to be used in the control law loop.

This software executes at 200Hz on the processor, giving it a loop time of about 5ms. This allows the control law algorithms to run through 20 iterations between new commands given by the Raspberry Pi which has a loop time of 100ms.

Range-Finding

Before the control law loop is executed, the microcontroller polls infrared range-finding sensors. These sensors return an analog value corresponding to the distance of the walls in front of the robot. When a certain threshold is reached, the robot behavior asynchronously exits the control loop and enters a separate set of behavior to avoid colliding with the walls. The robot attempts to back up and turn until the walls are no longer within the detection threshold. This threshold was determined through empirical testing. The detection threshold was gradually increased until the robot no longer had issues with running into the walls for most collision angles.

Control Laws

When the walls are not detected, the robot enters the control loop behavior. This loop updates the current estimated state of the robot, and generates outputs for the motors based on the difference in state estimation and location of the laser. Figure 7 below shows a block diagram of the control loop aspect of the code.

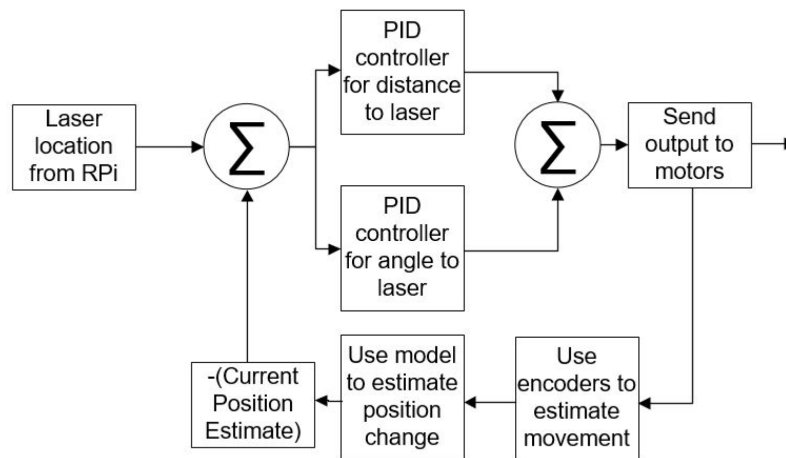

Figure 7: Block Diagram of Control Architecture

As described by this figure, the code goes through several steps to manage the control outputs. The first step is to look at the difference between the last transmitted location of the laser, and the current estimate of robot state. This generates an error in x-position and y-position between the robot and the laser. These errors in X and Y position are then converted into errors in distance and angle. Figure 8 below shows a visual representation of the two differences the controller acts to minimize.

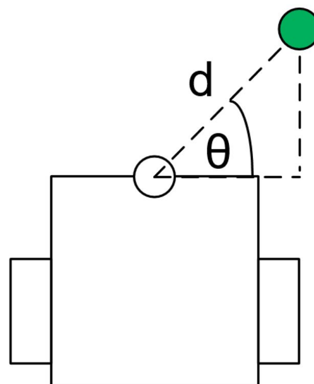

Figure 8: Distance and Angle Relative to the Robot

After the errors in distance and angle have been generated, they are sent into the control law algorithm. This algorithm implements a Proportional-Integral-Derivative (PID) controller, which is a standard closed loop control algorithm. Figure 9 below shows a standard block diagram interpretation of this algorithm from [5]. It takes in the error as input, and generates motor control values as output. A PID controller acts independently, through tuned gain values, on the value of the error (proportional), the long term accumulation of error (integral), and the rate of change of error (derivative). The sum of these three parts can be used to create an output that will attempt to drive the error input to 0, allowing the robot to reach its target.

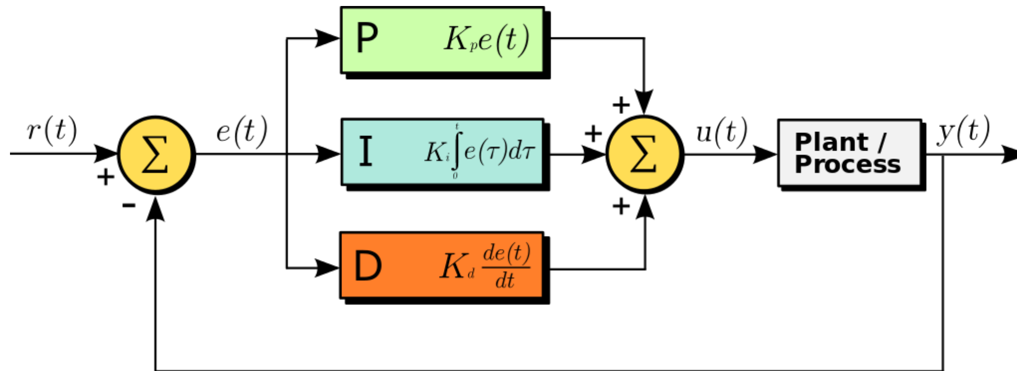

Figure 9: Standard Block Diagram of a PID Controller

The control algorithm first generates control outputs for both motors based on the distance. It then modifies one of these wheels with a value generated by the angle controller. This is treating a coupled dynamic system as though it is decoupled, but it was empirically tested to work in this case.

Handling Output Signals

After the outputs are generated, they are written to the motor control part of the software. This causes the microcontroller to generate a Pulse Width Modulated (PWM) signal and a direction signal to the motor driver ICs (integrated circuits) to make the motors move correctly. A PWM signal is a square wave signal at a specified frequency with a variable duration. The duration of the wave changes the speed of the motors. Code was written to initialize the PWM signal to 10kHz, a somewhat slow but acceptable frequency for driving motors. The microcontroller can drive a 10kHz signal with 7 bits of resolution. A higher frequency signal, though able to give a better response, would have reduced the resolution of the drive signal to an unacceptable level.

Differential Drive Robot Model

After the control outputs are sent to the motor, the internal model of the robot is updated. The robot is a differential drive robot (two wheels pivoting on center caster wheels), which has a well described dynamical model. This model is used to estimate the current state of the robot so that it can be used to determine the next set of control outputs. Figure 10 below, from [2], describes the algorithm used to update the state of the robot.

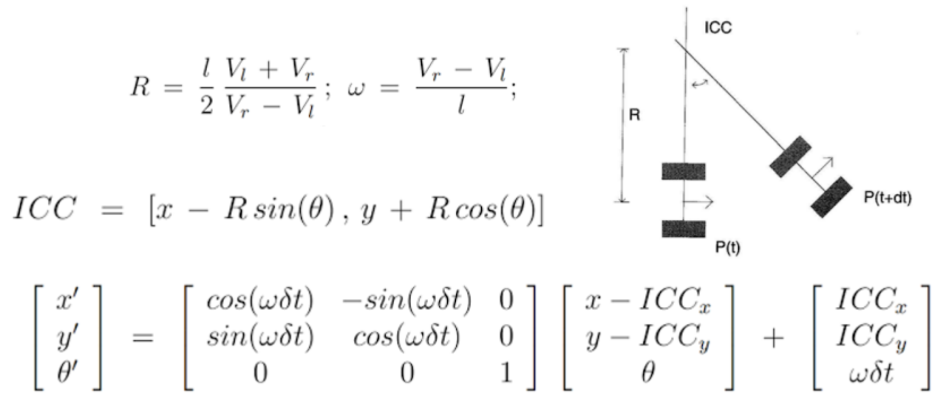

Figure 10: Mathematical Description of the Control Algorithm

Every time the state of the robot (also called the pose) is meant to be updated, the algorithm looks as the current states of the robot, applies the above formula to them described by the dynamics of the robot, uses the difference in time since it was last called to determine the magnitude of change, and sets the new state values. The pose of the robot is described by the X, Y, and theta values in its coordinate space, as well as by the current speed estimate given by the rotary encoders in each wheel. This method of updating state through the model is used as the feedback mechanism for the system.

This system is an implementation of dead-reckoning through odometry, which means that the estimate of the next state is being generated from prior estimates, and the only information it has available to do so is the wheel speed estimate. Dead-reckoning is known for being notoriously unreliable, as it is vulnerable to drifting over time because it is a chain of estimates derived from prior estimates. However, the state estimate of the robot resets every time a new laser coordinate is detected, because the robot coordinate space is derived relative to the laser pointer. This result almost entirely mitigates issues with drift.

Control Algorithm Modifications

Several standard improvements are added to this digital implementation of a PID controller. These include integrator saturation, which limits the amount of error that can be summed together. This helps mitigate integrator windup, which causes massive overshoot due to the physical limitations of the motors. There is also a set saturation value on the outputs to the motors, which keeps the speeds limited to the desired output bounds. The derivative calculation is also run through a moving average filter, which helps prevent small amounts of noise from causing instability.

Initial Implementation Attempts

Several implementations of a controller were attempted before settling on the described one. The first was an open loop controller, which attempted to map input values into the motors directly to speeds. These speeds were chosen such that the robot would move in a circle of a radius that would bring it directly to the laser. It would attempt to run the motors for the right amount of

time to allow it to arrive at the correct distance. However, it was quickly realized that the input to the motors could not easily be mapped to speed. They had a significant dead-zone, which meant that an input less than a certain value would not allow the motors to overcome static friction. It was also discovered that an equal input value to both motors would not necessarily result in the same output speed. Because of these issues, it was determined that a closed loop controller would be the better implementation. The first closed loop implementation tried to improve on the open loop method of generating a circle path, by trying to keep the wheel speeds (most importantly, their ratio such that the robot moved in the correct arc) at controlled values. The hope was that the integral portion of the control would overcome the static friction issue. However, it was shown that the microcontroller was not fast enough to maintain this implementation, and the system had difficulties dealing with small amounts of noise in the wheel speed estimates.

Issues Encountered

There were also difficulties with the implementation of the used algorithm. As with any control system without a good model of all necessary components (in this case, the relationship between the input into the motors and the corresponding wheel speeds), the controller gains needed to be tuned empirically. This took a significant amount of time to get an adequate response. The system also had an issue with overflow of the timer used to update the model. Upon overflow it would generate massive outputs, causing the robot to drive forward faster than the wall detection algorithm could catch and stop it. This was not noticed until design expo, as the robot had not been left running continuously long enough before to observe this issue. It could have been easily solved by adding logic to handle the overflow.

There were also general problems encountered while writing the software. It was discovered that several Arduino implementations of hardware peripheral interactions would not perform adequately to the desired specifications. These primarily included the microsecond counter and the PWM output system. The former had significant overhead, and took at least 50 microseconds to access. An implementation with less overhead was written. The PWM output was also only set to run at 500 Hz, the default for the Arduino platform. This speed is two orders of magnitude too slow to drive motors, so that peripheral needed to be set up with custom hardware interaction software as well. There was also an issue with the microcontroller keeping up with the input from the rotary encoders at full resolution. This problem was solved by reducing the resolution by a factor of four.

It also took some time to tune the infrared sensors to achieve the best possible behavior. Their analog response to distance is extremely nonlinear, and could not be well approximated. The specification sheet gave a recommended calibration curve, but the actual response was not consistent with it. The threshold values to prevent collision were in the end tuned empirically. The most significant issue with the sensors was the inability to detect shallow collisions with the wall. This problem could sometimes get the robot stuck.

Hardware Selection

DOGBOT required many pieces of hardware for the success of the project. This section will give a brief overview of the most significant components. For more detailed analysis of each part and the motivations behind each design decision, please see Appendix 1.

Major Hardware Components

The primary hardware component of the project is the Raspberry Pi 3 single board computer. It is used to perform image processing, as well as various communications operations. These included talking to the microcontroller, wireless remote login to monitor code, and running a wireless video stream. The Raspberry Pi interfaced with the standard Raspberry Pi v2 video camera, which is used to capture and process images at about 10Hz.

A green laser pointer was chosen for this project. It has a 5mW power output, which is acceptably bright for use inside, and not so bright that it could inadvertently hurt someone.

Four infrared range-finding sensors are used to help the robot understand the obstacles around it. They sit parallel to the drive plane of the robot, and return an analog value corresponding to the distance of the walls (from 4’ to 60’).

Two motors drive the robot. These relatively high-speed and high-torque motors allow the robot to move somewhat aggressively at about five feet per second. They are attached to 2.4” wheels. The motors have built in rotary encoders that create a digital signal corresponding to the behavior of the wheels. They are used to determine the speed of the robot to model its dynamics.

A custom printed circuit board chassis was used for this project. It created effective attachment points for all of the mechanical parts, and clean connections for all of the electrical parts. It has several components attached to it, including an Atmel SAMD21 microcontroller, two H-Bridge motor drivers, and power electronics. It is described in significantly more detail in Appendices A1.5-1.7.

Table 1 is a list of all of the general hardware components used for this project, and includes links to webpages where they can be ordered. Appendix table A1.1 contains a list of all of the printed circuit board parts ordered.

Motivation Behind Custom Robot Implementation

The choice to design a chassis, choose special motors, build motor drivers and design power electronics from scratch was motivated by alternative options. The first primary option would have been to use the standard robot available in the lab. However, by observation, it was determined this robot was not able to travel nearly as fast as we desired for the project. To adequately create an entertaining emulation of a pet, a significantly more rapid response was needed. This desire motivated a second alternative: modifying an RC car as has been done in past semesters. However, the process of modifying existing devices can be risky without detailed schematic information, and there is not a guarantee of success. There is also not an easy way to

mount parts on the chassis of a standard RC car. A third option then would have been modifying the lab robot (or a similar chassis) by swapping out all the parts. However, at this point, the cost of parts and time required to ensure it integrates correctly are on the same order of magnitude, and it makes more sense to design a streamlined solution that we can guarantee will meet the requirements.

Table 1: List of Parts Used In DOGBOT

Item	**Quantity**	**Link**
Motors	2	https://www.pololu.com/product/2271
Motor Bracket Pair	1	https://www.pololu.com/product/2676
Wheel Hubs	1	https://www.pololu.com/product/1081
Wheels	1	https://www.pololu.com/product/1420
Casters	2	https://www.pololu.com/product/953
Green Laser Pointer	1	https://www.amazon.com/product/B01J
Range Finder	4	https://www.pololu.com/product/2476
Raspberry Pi 3	1	https://www.pololu.com/product/2759
Raspberry Pi Camera	1	https://www.adafruit.com/products/3099
Raspberry Pi Camera Cable	1	https://www.adafruit.com/products/1648
Raspberry Pi GPIO Cable	1	https://www.adafruit.com/products/1988
Raspberry Pi Camera Mount	1	https://www.adafruit.com/products/1434
Printed Circuit Board	1	Donated by Advanced Circuits
PCB Parts	1	See Table A1.1
2S LiPo (2000mAh or greater)	5	Borrowed from MASA
Wood (3 pieces of 1" x 6" x 8ft)	3	Bought at Home Depot
Hinges for Boundary	4	Bought at Home Depot

Hardware Architecture

Figure 11 is a hardware architecture diagram that shows the connections between the major hardware components. It shows the connection between the microcontroller and the low level components, including the rotary encoders on the motors, the infrared range-finding sensors, and the motor drivers. It also shows the connection between the microcontroller and the Raspberry Pi, which is a standard serial UART interface. This diagram also shows the power electronics of the hardware, which describes the voltage regulation on the robot. Battery voltage is taken through a low loss switching regulator, which converts the 7.4V from the battery to 5V. This 5V power rail is used to power the Raspberry Pi. The 5V rail is fed through a linear regulator, which has a faster and more constant response at the cost of more power loss, which creates a 3.3V power rail for the rest of the components.

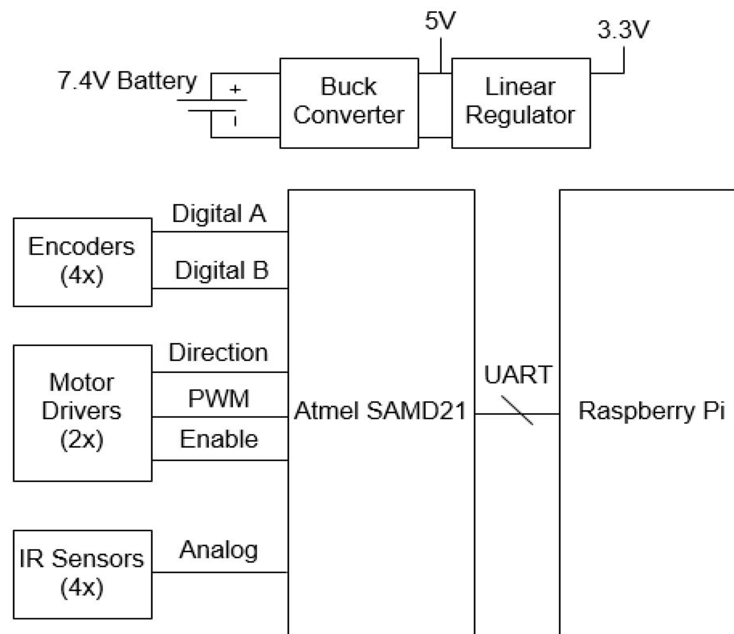

Figure 11: Hardware Architecture Diagram Shows Connections

III. Milestones

Milestone 1 was set to be in early November. We expected to have a general plan of each component of the project. We aimed to have all algorithms planned, image processing and laser detection code completed, embedded control pseudocode completed, and be working on embedded control code. In addition, we hoped to have all parts (boundary and microprocessor/PCB) to build designed, and have our infrared range finders and chassis ordered. Concurrently, we would also be testing our functioning image processing code in adverse conditions (ie. different lighting, extremely fast laser movement).

We did not meet our milestone 1 expectations. We were only able to plan all our algorithms, design and order hardware, and have pseudocode done for image processing and embedded control code. As a result, we set an intermediate milestone 1.5 for which we intended to achieve our milestone 1 goals and present them to Professor Wakefield. For milestone 1.5, we wanted to have working image processing code that was being tested in adverse conditions. We also wanted to have some embedded control code working and have our robot built and working. Two weeks after milestone 1, we had successfully met our milestone 1.5 goals. Our robot was built, we had a functioning image processing algorithm and functioning control code.

For milestone 2, we expected that all systems were functional and that the robot could go on display a week before the design expo, but that its performance might be notably laggy - physically and spatially. At this point, all our hardware should be working properly. We expected the image processing algorithms and software to be functioning accurately and efficiently in adverse conditions. Also, the embedded control firmware should be working at this point, but possibly need fine tuning to make the robot's movements more precise and timely.

We did meet our milestone 2 expectations. Our hardware and software were completed to a point where we would have been comfortable presenting at design expo, even without further tuning. We were able to give a demonstration of our working project to the course staff at this time. We did, however, spend the next week continuing to develop and tune the project. This development mostly involved changing parameters on the image processing to have better laser detection and noise rejection, as well as changing parameters on the control functions to have a better following response.

IV. Project Demonstration

At Design Expo, we were allotted a space of 4 feet by 6 feet for our project setup. This area included a bounded arena in which our robot moved and interacted with the laser pointer. Visitors were given the opportunity to use the laser pointer and point it inside the arena. They would then be able to see the robot chase the laser and serve its ultimate purpose, to act like a dog. A picture of this arena, with the robot chasing a laser, is shown in Figure 12. In addition to our project, we also had a poster describing DOGBOT and its functionality. This poster included some of the visuals in this report along with brief technical descriptions of the various components of DOGBOT.

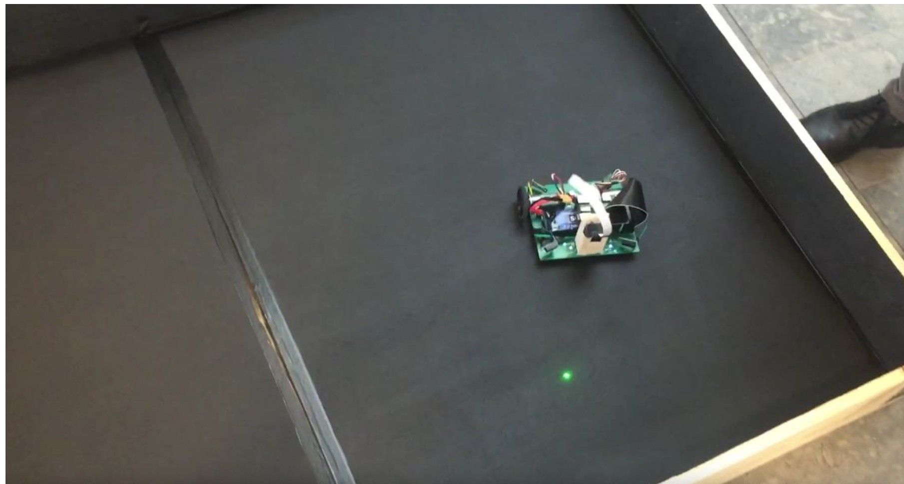

Figure 12: Arena Setup At Design Expo

V. Contributions

The division of work among all three members of our team is shown in the Table 2 below.

Table 2: Contribution and Effort of Each Team Member

Team Member	Contribution	Effort
Andrew Levin	Control & Image Processing Support, Testing, Boundaries, Poster, Report	33%
Shiva Mehta	Image Processing, Testing, Poster, Report	33%
Jonathan Zarger	Hardware Design, Microcontroller Code, Controls, Poster, Report	33%

VI. References and Citations

[1] "OpenCV: Image Processing in OpenCV", Docs.opencv.org, 2016. [Online]. Available: http://docs.opencv.org/3.1.0/d2/d96/tutorial_py_table_of_contents_imgproc.html.
[2] "CS W4733 NOTES - Differential Drive Robots", 2016. [Online]. Available: https://chess.eecs.berkeley.edu/eecs149/documentation/differentialDrive.pdf.
[3] "Infrared vs. Ultrasonic - What You Should Know", 2008 [Online]. Available: http://www.societyofrobots.com/member_tutorials/node/71
[4] "Drive Motor Sizing Tool", 2016. [Online]. Available: http://www.robotshop.com/blog/en/drive-motor-sizing-tool-9698
[5] "PID controller", 2016. [Online]. Available: https://en.wikipedia.org/wiki/PID_controller

VII. Appendices

The following appendices were included to provide a more detail regarding hardware design, initial image processing work, and controls development. The image processing appendix is shorter as much of the image processing details were necessary to explain in the body of the report since EECS 452 is all about signal processing.

<u>Appendix 1: Hardware Design</u>

This section will motivate the design decisions behind all of the hardware components of DOGBOT. It will also include various component selections, the printed circuit board design, and the overall assembly.

1.1 Raspberry Pi 3

The main processing unit for this project was the Raspberry Pi 3 single board computer running Raspbian, a variant of Debian. It is used as the primary processing unit for the project. It runs the image processing algorithms, piped through four processing cores, and communication functions like transmitting laser coordinates to the chassis, handling remote login, and running a video

stream. It was chosen for it's relatively fast processor, it's availability in the lab, and it's significant amount of community support.

1.2 Raspberry Pi Camera
The v2 camera is used for this project. It captures frames in front of the robot. These frames are processed so that the laser can be detected.

1.2.1 Camera Mount
The camera mount was built from a combination of the standard plastic holder for the camera, and a wooden block cut to angle it correctly. It was hot glued onto the chassis.

1.3 Infrared Rangefinder Sensors
Sharp GP2Y0A60SZLF infrared sensors were chosen for this project. They return an analog value that describes how far objects are in front of it. The line of sensors was chosen based on a recommendation from [3], which describes the costs and benefits of using infrared sensors range finding sensors as compared to ultrasonic sensors. As described in this article and by the course staff, ultrasonic sensors used in an enclosed arena would cause undesirable reflection effects, and would also not behave well on off-angle measurements. This particular line of infrared sensors was described in their specification sheet to have fairly accurate analog output, resistance to sun interference (though not in direct sunlight), and would behave consistently on off-angle measurements. It also has a range of 4 inches to 60 inches, allowing it to see most of the arena. Based on testing, they seemed to perform as expected.

1.4 Motors
Two relatively high RPM and high torque motors were chosen for this project to better emulate the behavior of a dog. The estimate of specifications was given by [4] with the parameters 5 pounds and 5 feet/second with 2 inch wheels. These specifications prescribed a motor with at least 20 oz-in of stall torque, and 500 RPM at normal operation. From prior experience (the release of a significant quantity of blue smoke) it is known that this tool underestimates specifications, so a certain amount of over-meeting requirements was necessary. The motor chosen was a "9.7:1 Metal Gearmotor 25Dx48L mm HP 6V with 48 CPR Encoder", which has a 39 oz-in stall output and a 990 RPM rating with no load. It draws 6.5A at stall. It also has a built in rotary encoder, which is useful for getting feedback on the dynamics of the robot.

1.5 Printed Circuit Board Chassis Schematic Capture
The decision to design custom hardware was motivated by the need for better motors. The robots in the lab would not have the desired speed for this project, and modifying another robot chassis (and buying separate power supplies and motor drivers) would be at least as expensive as a custom solution. The custom chassis needed to be designed to support and connect all components electrically and mechanically. It has slots for both motors and wheels, the Raspberry Pi, a camera mount, a battery, and the integrated circuit components for the board. The schematic capture was designed with Altium CircuitMaker, a free version of Altium Designer. An image of the schematic capture is in this report as Figure A1.1. Further questions about the PCB design should be directed to Jonathan Zarger (jzarger@umich.edu).

1.6 Integrated Circuits on Chassis

This section will detail the significant design decisions for integrated circuits used on the PCB chassis. For many small parts (resistor/capacitor choices etc…) the design process is not significant enough to detail. The connections and full parts set are detailed in the schematic capture, designated Figure A1.1.

1.6.1 Microcontroller

The microcontroller chosen for this project was the Atmel SAMD21G18. It comes from Atmel's (now Microchip's) ARM microcontroller line. It is based on an ARM Cortex M0+. This microcontroller is the same one used in the Arduino Zero microcontroller breakout. It was chosen primarily because a team member had prior art involving use of this microcontroller, making it a reliable and safe choice. It also supports use of the Arduino software platform, simplifying the process of most software development.

1.6.2 Power Electronics

This project has two power regulation circuits. The primary is a switching regulator designed to take battery voltage (7.4V nominally) and convert it to 5V. A Diodes INC AP65502 controller was used, along with peripheral component selection (inductance and capacitance) recommended by the application notes. It can support at least 5A of current at 5V, which is more than sufficient to power the Raspberry Pi and the rest of the components. That regulator feeds into a Microchip TC2117-3.3V linear regulator, that creates a 3.3V rail for the rest of the components on the board, including the microcontroller and the motor driver IO.

1.6.3 Motor Drivers

Two Infineon TLE9201 H-Bridge and driver ICs were used for this project. They internally have a full bridge and driver electronics, making it relatively simple to connect a motor to each, and digital IO lines to the microcontroller to drive the motors. They also have SPI diagnostics that can be accessed by the microcontroller to troubleshoot common motor issues like shorts, opens, and over-currents. They are rated for 6A continuous, and will chop if this limit is exceeded. This setting makes them well rated for the 6.5A stall current motors that were chosen. Significant capacitance was placed on the power rail for each motor. These capacitors were chosen so that they could support collectively 4A of ripple current for each motor. Transient suppression diodes were also places on each motor line to protect the rest of the board.

1.6.4 Level Shifter

The encoders that came with the motors required that at least 5V be used to power them, which also created 5V IO lines. The microcontroller is only rated for 3.3V IO, so a NXP 74HC4050PW Level Shifter is used to convert the 5V signal to a 3.3V signal with little signal distortion or lag.

1.7 Printed Circuit Board Layout

Significant care was taken to ensure that the first layout of the board would be correct. This effort involved much time to reduce current loops in the switching regulator, thought and planning into routing of motor power traces, looking at signal integrity and potential interference near the microcontroller, and quadruple checking mechanical layouts (especially the non-integrated circuit parts). The ECAD was designed with Altium CircuitMaker, a free version

of Altium Designer. The board itself was ordered through the Advanced Circuits (4pcb.com) standard 33each order, which is a two layer board with strict restrictions on trace width and spacing. An image of the bottom layer (which has most of the significant layout detail) is attached to this report as Figure A1.2. A full system description would require looking at all of the ECAD files, commonly shared in Gerber format. There isn't a good way to transmit Gerber files through a report, so please contact Jonathan Zarger (jzarger@umich.edu) if there is any interest in them. They will also be attached to this report in the ZIP folder with code.

1.8 Issues with Hardware Design

No major issues were detected with the hardware design. There were several small issues. None of the passive LEDs worked successfully, though it's fairly likely they were soldered on backwards. The battery voltage detection circuit was never soldered on and tested, even though it really should have been used to prevent issues with the lithium polymer batteries. A breakout for a 9DOF IMU was left on the board for experimentation, but was also never used.

There were also general issues with assembly and the assembly process. The large electrolytic capacitor on the 5V rail was on the bottom side of the board, and calculations to ensure it was not tall enough to drag on the ground were slightly incorrect. This capacitor removed itself as the robot was driving and was replaced with a more suitable option. When surface mount parts were ordered, a few resistors were missed, and had to be replaced with less suitable parts. Most notably were 1% tolerance 0603 package 10kOhm resistors, which were necessary for the feedback on the switching power supply. This component was replaced with a 5% 0805 package resistor, which was manually measured to check the resistance and soldered on in a precarious fashion. This replacement caused many issues with the testing of the power supply until it was soldered on successfully.

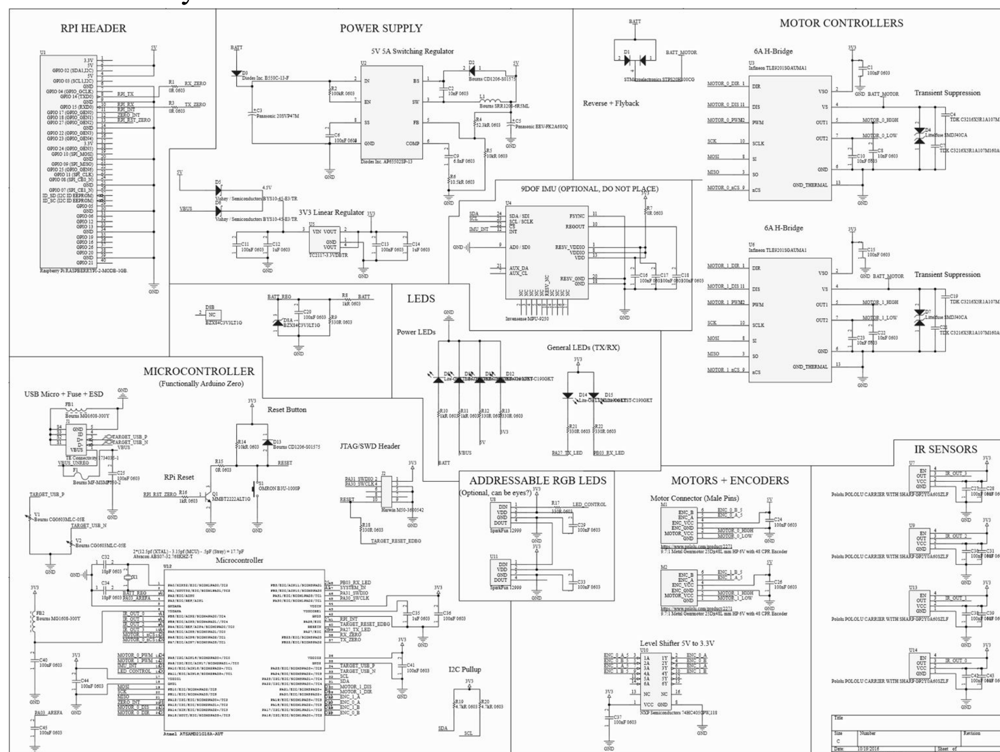

Figure A1.1: Schematic Capture of Chassis PCB Components

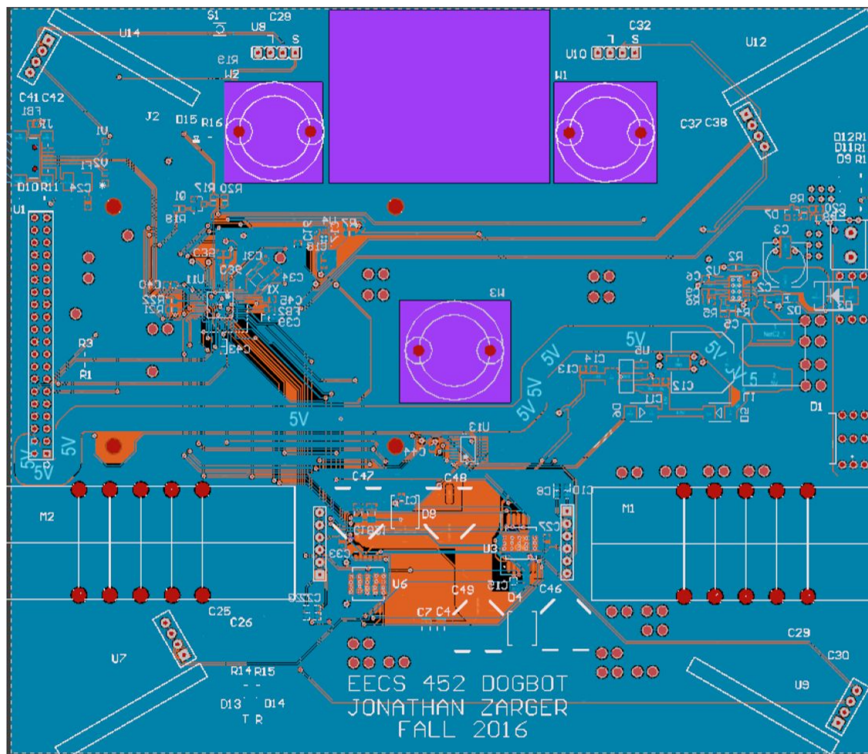

Figure A1.2: Visualization of Bottom Layer of Chassis PCB

Table A1.1: Chassis Components Parts List

Qty	Line Item	Description
1	571-5103308-8	RPi Connector
1	1734035-1	USB Connector
1	M50-3600542	SWD Connector
1	1729018	Battery Screw Terminal
1	B3U-1000P	Reset Button
1	AP65502SP-13	Switching Regulator IC
1	TC2117-3.3VDBTR	Linear Regulator
2	TLE9201SGAUMA1	Motor Driver IC
1	ATSAMD21G18A-AUT	Microcontroller
1	74HC4050PW,118	Level Shifter
1	B550C-13-F	Reverse Protection
2	CD1206-S01575	Switching Flyback
2	BYS10-45-E3/TR	5V OR
1	STPS20H100CG	Motor Reverse Protection

2	SMDJ40CA	Motor TVS
2	LTST-C190GKT	Green Power LED
4	LTST-C190CKT	Red GPIO LED
1	BZX84C3V3LT1G	Zener Diode
1	MF-MSMF050-2	USB Fuse
2	CG0603MLC-05E	USB Varistor
2	MG1608-300Y	Choke
1	SRR1208-6R5ML	Switching Inductor
1	ABS07-32.768KHZ-T	RTC XTAL
1	MMBT2222ALT1G	NPN Transistor
1	20SVP47M	Battery Cap
1	EEV-FK2A680Q	5V cap
0	C3216X5R1A107M160AC	Ceramic motor cap 100uF
4	16SVP150M	electrolytic motor cap 150uF
5	06035C103KAT2A	10nF 0603
1	06035C682JAT2A	6.8nF 0603
7	C0603C105K4PACTU	1uF 0603
2	06035A180JAT2A	16pF 0603
25	0603YC104JAT2A	100nF 0603
5	652-CR0603-J/-000ELF	0R 0603
1	RC0603JR-07100KL	100kR 0603
5	CRCW06031K00FKEA	1kR 0603
1	CRCW06034K70JNEA	4.7kR 0603
1	CRCW060352K3FKEA	52k3R 0603
1	ERJ-P03F1052V	10.5kR 0603
19	CRCW0603330RFKEA	330R 0603
20	CRCW0603100KFKTA	100R 0603

Appendix 2: Image Processing

Figure A2.1 shows some of the images on which we tested initial image processing code. Image processing did not work well on any of these images. This result lead us to using a solid background color for the robot's environment.

The image in the top left was an attempt to simulate a strongly lit environment. In the top right corner of this image you can see us attempting to simulate intense light through the use of our cell phone flashlights. Image processing worked better - in these intensely lit situations - on

black backgrounds than on white backgrounds. Conversely, in situations with little light, image processing performed better on a white background than on a black one. However, knowing that we had to demo the robot in an extremely bright area, we decided to move forward with an all black environment.

Figure A2.1: Some of the Images Used in Testing Image Processing

Appendix 3: Embedded Control

This section goes into more detail regarding the development and tuning of the PID controllers and the tuning of the control algorithms in more detail.

3.1 Controller Tuning

After a control algorithm was chosen, a significant amount of time was taken to set the control gains correctly. Designing a controller offline is certainly the most desirable technique, but requires at least somewhat reasonable models. Manual characterization would have needed to be performed on the motors to create this, so mostly empirical tuning was applied. The tuning process started initially with only proportional controllers to get a baseline response to build from. Initial proportional gain values for the distance and angle controllers were chosen by inspection, looking mostly at the desired relation between distance error and motor output. That is, choosing a proportional value based on the relation between input and output. Once the initial response was established, a similar process was applied to integral and derivative gains.

3.2 Integral and Derivative Gains

It could certainly be argued that the implementation of integral and derivative gains was unnecessary. The proportional gain alone gave a somewhat reasonable response, and could have been used alone. It was however determined that they did improve the performance. Because of the deadzones of the motors, the proportional controller was unable to chase laser location very close to it. The integrator build-up allowed it to generate outputs high enough to break past the deadzone and reach it. However, as is common with integrator responses, this added a significant amount of overshoot. Despite implementing integrator clearing and saturation to reduce this windup effect, there was still significant overshoot. The derivative controller was used to slow down the response of the robot to reduce the overshoot.

3.3 Testing Process

Initial testing was performed with the wheels of the robot lifted. It was given a single point to drive to, and the response of the robot was observed. If the response was incorrect, the gain values were changed accordingly. Once the response seemed reasonable, the process was repeated but with the robot in motion on the ground. Controller gains continued to be tuned.

3.4 Logging to Optimize Tuning

One of the most difficult aspects of initial tuning the robot was being unable to observe the robot's internal state. To resolve this, many relevant values were written over serial to a Python program collecting the data and writing it to a .csv file. The values in this file were observed and plotted in MATLAB, which greatly streamlined the tuning process.

3.5 Coupling Between Distance Controller and Angle Controller

The interaction between the distance and angle controllers was trivialized in the above sections, but it did take a significant amount of work to overcome. As described in the control section, the distance controller generated baseline output values and then the angle controller modified those outputs to cause the robot to turn. The outputs to the motors were being carefully saturated to keep the robot from moving too fast, so the effect of the modification to the outputs from the angle controller was being entirely removed by this saturation. Several attempts were made to fix this. Initially the outputs went through several scaling functions, which ended up putting the modified output into a motor deadzone. A context based modification was written to try to change the values in a way that would not go past the saturation value but also not fall into the deadzone, but it proved to be impossible to satisfy both of those constraints. The best solution ended up simply being to allow the angle controller to write past the saturation value.