THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF MECHANICAL AND NUCLEAR ENGINEERING

DEVELOPMENT OF GROUND ROBOT VALIDATION METHODS AND STUDIES IN OPERATOR AND TERRAIN VARIABILITY FOR NIST TEST METHODS

ADAM CRIMBOLI
SPRING 2014

A thesis
submitted in partial fulfillment
of the requirements
for baccalaureate degrees in
Mechanical Engineering and Nuclear Engineering
with honors in Mechanical Engineering

Reviewed and approved* by the following:

Sean N. Brennan
Associate Professor of Mechanical Engineering
Thesis Supervisor, Honors Advisor

Karl Reichard
Research Associate, Applied Research Laboratory
Assistant Professor of Acoustics
ARL Research Supervisor

Hosam Fathy
Assistant Professor of Mechanical Engineering
Faculty Reader

* Signatures are on file in the Schreyer Honors College

ABSTRACT

In critical emergency situations such as bomb disposal the operational characteristics of emergency response robots must be well understood to optimally predict behavior. Standardized testing allows the development of statistics to quantify robot performance. This thesis presents improvements made to a National Institute of Standards and Technology (NIST) ground robot testing method and a previous student effort in this area.

During a test, overhead cameras capture images, and computer algorithms are employed for further processing. Fiducial tracking algorithms calculate a robot's position, speed, and lap progress. Improvements developed in this work include improved camera calibration and refinement of the fiducial tracking system, as well as the addition of a most common path processing algorithm. In addition, this thesis presents the addition of robot power consumption information to the test method. Lastly, robot testing explores applications for employment in operator and terrain variability studies.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This thesis was made possible through the combined support of the Penn State Intelligent Vehicles and Systems Group and the Penn State Applied Research Lab. I would like to thank Dr. Sean Brennan for his invaluable help in producing this thesis specifically, as well as his continued support and mentoring over the course of my entire undergraduate education. I would also like to thank Dr. Karl Reichard for his enthusiasm in this project and willingness to deploy ARL resources in pursuit of its completion, and for his availability and advice. I would like to thank Jesse Pentzer, a graduate student of the IVSG and a fellow researcher at ARL. His direct support and mentoring helped guide me through many aspects of academic research. Lastly I would like to thank Herschel Pangborn, whose continued support made this work possible.

Chapter 1 Introduction

Mobile ground robots are an important tool for both civilian and government agencies when operation within a hazardous environment is required. Due to the relatively new nature of the industry and the fact that robots vary widely in design and capabilities, development of such robots is accompanied by a lack of standardized performance testing. Standardized specifications of performance would allow potential customers to compare robots and select a design most suited to the required task. The National Institute of Standards and Technology (NIST) has developed a series of testing procedures in an effort to quantify the performance of mobile ground robots [1].

Current NIST tests require a robot to perform a specific task, usually with repetition, such as driving laps around a track. The performance of a robot in a test can be timed or graded on a pass/fail basis. The focus of this work seeks to improve the NIST testing method through automation and the addition of additional metrics of performance. For this project, one NIST testing procedure was analyzed. Robots are tasked to drive in a figure-8 pattern in a testing space for multiple laps. The testing space is a rectangular 8 foot by 24 foot testing arena [2]. An example of a robot in action can be seen in Figure 1-1.

In addition to the standard NIST test method, two automated systems are used to record data during a test. First, ceiling mounted cameras and image processing software are used to track the position of a robot at any given time. Second, a data logger attached to the robot records the battery current and voltage during testing to characterize power and energy use.

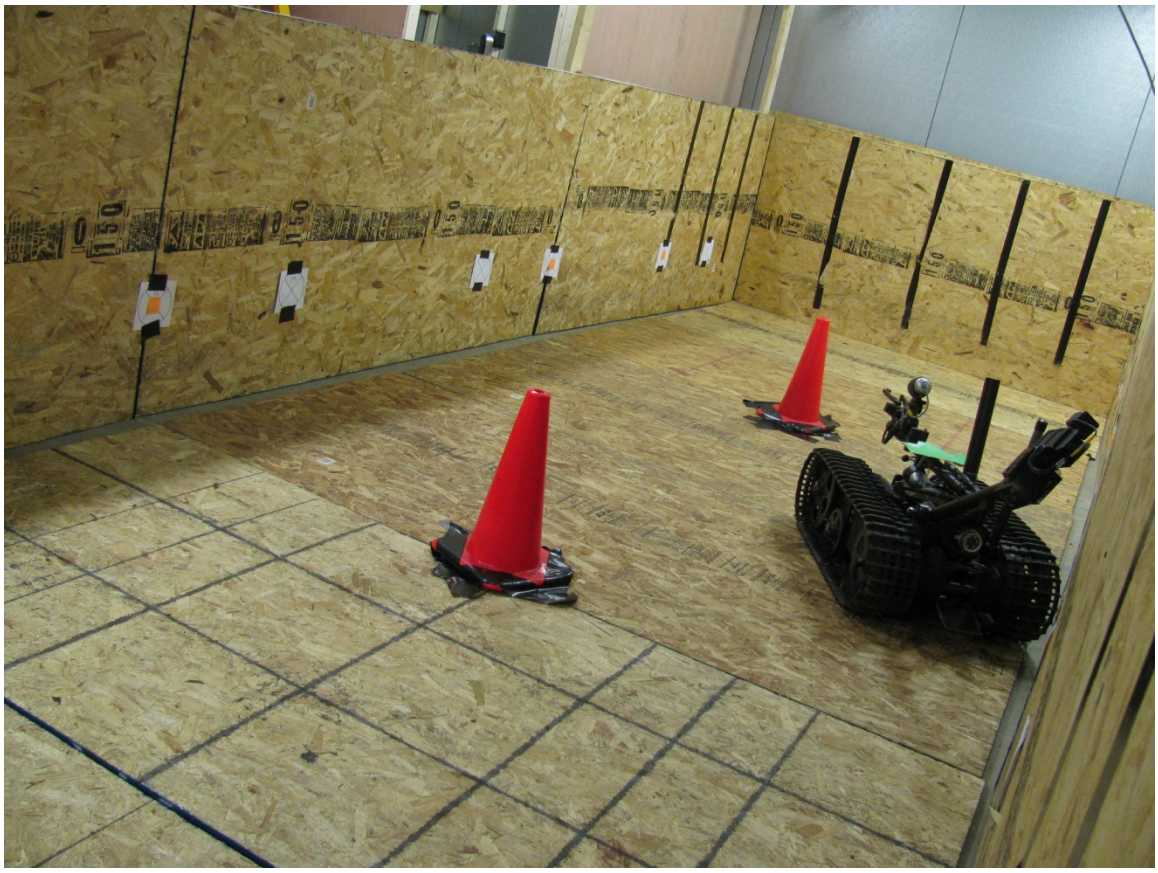

Figure 1-1: Robot testing demonstration

From these data acquisition systems, multiple performance metrics can be derived. For example, instead of manually recording lap time with a stopwatch, the time to complete each lap is recorded automatically from timestamps on collected images. From position information, total distance travelled can be easily calculated. Finite differencing of position information over time produces robot velocity. A less trivial performance metric that was developed is that of consistency, or deviation from the most common path. Using data collected over an entire test, the most common path traveled by the robot is found using watershed transformation processing. Deviation from this path at any given point in the test can then be found. From the onboard data logger, multiplying current and voltage calculates electrical power, which can be integrated to yield the energy used by the robot during a lap or entire test.

Three large issues were encountered in the development of this system. The first involves the distortion correction, calibration, and stitching together of multiple camera images. Due to the size of the apparatus, it would have been difficult for one camera to capture the entire arena. As such, three cameras each capture a section of the arena and these sections must be overlaid to produce a continuous image. Next, another issue was presented in the reliability of the fiducial image tracking system. Over the course of a long test the ability of the image processing software to reliably track the fiducial was questionable. Fiducial tracking often failed for a small percentage of points. Adding a background subtraction technique to the image processing generally improves results, but presented its own problems. Lastly, a challenge presented by the use of two independent systems of data collection was the ability to synchronize produced data. The data loggers chosen do not wirelessly communicate with the camera system. Instead, the data are synchronized later in processing by matching pauses in robot operation.

The testing system developed has the potential for a wide variety of applications. Work that is enabled by this project includes the study of robot operator learning curves as well as fatigue. How new operators improve over time and how fatigued operators decrease in performance is something that can be easily observed and quantified using the testing system.

For this study, robot and terrain variability and its effect on performance is studied. The floor of the testing arena is changeable to accommodate different terrains such as concrete, particle board, and ramps. Testing the same operator and robot on different terrains allows terrain effects to be studied in isolation. Two robots were available for the tests associated with this work, the Talon robot and the BomBot. The Talon robot used for testing is shown in Figure 1-2.

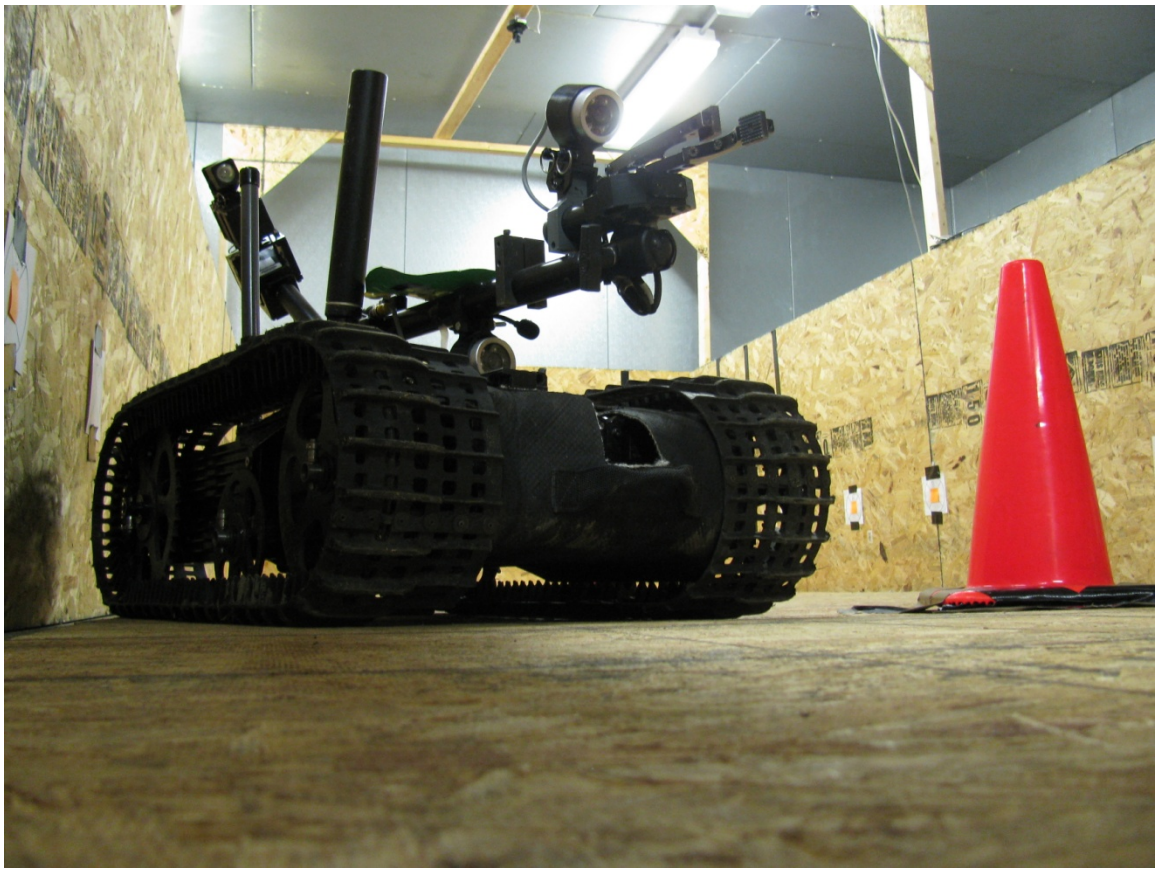

Figure 1-2: Talon robot in action

This project builds off the work in NIST testing and fiducial identification and tracking developed by Herschel Pangborn, who constructed a replica of the testing arena and developed the overhead camera fiducial tracking system [3]. Processing of data was primarily achieved in MATLAB. Chapter 2 of this document focuses on the physical specifications of the testing system and equipment. Chapter 3 details the improvements made to the camera acquisition system and robot-tracking image processing software from the version developed by Pangborn. Chapter 4 deals with the addition of power information and how the two data sets were synchronized. Chapter 5 presents the final product of processing and displays sample results. Chapter 6 details the results of multiple tests, and Chapter 7 comments on the testing system and

suggests future work. The MATLAB code used to process the data has been provided in appendices.

Chapter 2 Testing Equipment

Robot testing for this work took place indoors in a NIST testing arena built to follow mobility test specifications [2]. This chapter will discuss the testing arena, the robots utilized for this research, the power logging devices, and the overhead cameras used to track robot position.

2.1Testing Arena

Two documents of reference were used to motivate this work [1] [2]. Both documents serve as manuals explaining the purpose of standardized robot testing and guides to carrying out tests. Various robot tests require the construction of arenas, steps, or other obstacles. This project utilizes the standard testing arena constructed by Herschel Pangborn [3]. A graphic of the testing arena can be seen in Figure 2-1.

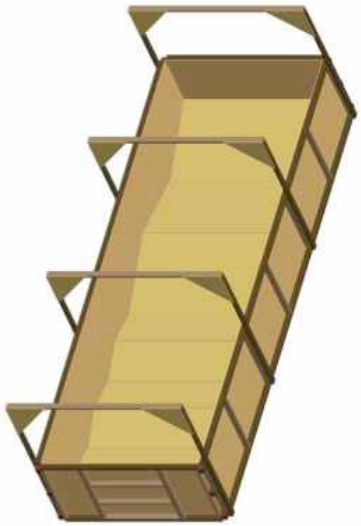

Figure 2-1: Graphic of NIST testing arena [2]

This testing arena was chosen for its versatility for mobile ground robot lap testing. In its 8 foot by 24 foot testing space, ground robots can be operated in a specified driving pattern for multiple laps. The walls of the arena are made of plywood and serve to confine the robot in the testing space and a swinging door on the end of the arena allows for robot entry and exit.

Additionally, the entire arena is reinforced by a wooden frame. A more detailed explanation of the physical description and construction of the testing arena can be found in Pangborn's thesis [3].

2.1.1 Testing Arena Modifications

Several modifications were made to the testing arena that depart from the standard NIST assembly guide, either for convenience or to aid in image processing. The standard NIST testing protocol specifies that robots be driven in a figure-8 pattern around the testing arena, avoiding pylons anchored along the center at 1/3 and 2/3 of the way along the length of the testing arena. In addition, robots must drive through the end zones, which are designated with black and white stripes, at the last 4 feet of each end of the arena. A photo of the original testing arena developed by NIST, located at their testing facility in Gaithersburg, Maryland, can be seen in Figure 2-2.

Figure 2-2: Photo of original NIST testing arena [2]

For the testing arena built for this work, the end zone walls were not painted with black and white stripes. Instead, strips of black duct tape were used on the walls and floor to indicate each end zone. Traffic cones were used in place of pylons at the 1/3 and 2/3 points of the arena and mounted to either the floor or wooden boards with duct tape or screws.

For the purpose of calibrating the testing space for image processing, colored paper squares were placed along the walls of the testing arena. The colored squares were placed along the walls of the testing arena at 2 foot increments, and placed 14 inches off the ground. The calibration process will be explained further in Chapter 3.

2.1.2 Testing Arena Terrains

The original terrain specified by NIST for this testing arena is dubbed “continuous pitch/roll ramps.” This terrain consists of 24 wooden half-ramp elements, with each element having a footprint with length and width of 24 and 48 inches, respectively. The ramps have a 15 degree incline and rise to approximately 7 inches in height. Figure 2-3 shows a half-ramp element.

Figure 2-3: Half-ramp element [2]

For the configuration of the continuous pitch/roll ramp terrain, these ramp elements were laid out in rows side-by-side down the length of the arena, with each subsequent element alternating the angle of inclination. The end result can be seen in Figure 2-4.

Figure 2-4: Continuous pitch/roll ramp setup

The same terrain half-ramp elements can be rotated to form another standard NIST configuration, known as crossing pitch/roll ramps. This configuration can be seen in the graphic in Figure 2-5.

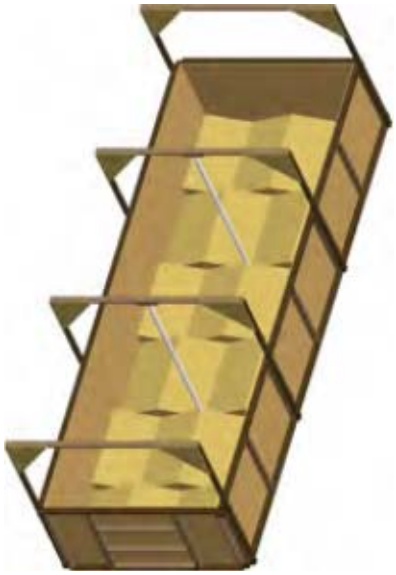

Figure 2-5: Crossing pitch/roll ramps [2]

The ramps themselves are entirely removable from the testing arena. For this work, more terrains were used for the testing arena which are not standard NIST tests but are nevertheless useful standards for comparative robot testing. Removing the ramps from the testing arena allows testing on the smooth concrete floor of the room in which the testing arena is located. In addition, panels of flat oriented strand board (OSB) were placed on the floor of the arena as another terrain.

2.2 Robots

2.2.1 Talon

Two robots were primarily used for this project. The first is the Talon robot. Originally developed by Foster-Miller and currently produced by QinetiQ, the Talon is a popular bomb-disposal robot that has been in use by the United States military for many years [4]. A photo of the Talon in action can be seen in Figure 2-6.

Figure 2-6: Talon robot [4]

The Talon is a tracked robot with a zero degree turning radius. It weighs approximately 130 pounds and measures approximately 2 feet wide by 3 feet long. Four onboard cameras can be used to operate the robot. While various models of Talon exist, the model used in this work is as

shown in Figure 2-6. The primary method of bomb disposal is through the manipulation of a claw arm on the front of the robot, which can extend over 4 feet outwards or upwards. A mast towards the rear of the robot with an attached camera is used to attain a wider field of view for navigation.

The mobility tests in this work do not make use of the manipulator arm so the arm is stored in a compact resting position for the duration of testing. The Operator Control Unit (OCU) for the Talon is shown in Figure 2-7.

Figure 2-7: Talon robot OCU

The Talon is powered by BB-2590 military batteries, an example of which can be seen in Figure 2-8. The Talon can be loaded with anywhere from 1 to 6 of these batteries, connected in parallel, each of which weights 1.4 kg. The battery can be used in two voltage modes, 14.4 V and 28.8 V. For use with the Talon these batteries provide approximately 28.8 V to the Talon.

Figure 2-8: BB-2590 battery [5]

2.2.2 BomBot

The BomBot robot is a wheeled mobile robot meant to serve as a smaller and cheaper alternative to bomb disposal robots such as the Talon. It weights approximately 30 pounds and measures approximately 1.5 feet in length, 1 foot in width, and 1 foot in height. Whereas the Talon has treads, the BomBot is a wheeled robot, and therefore does not have a zero-degree turning radius. The BomBot is notable for its four-wheel drive and very soft suspension. The BomBot's design is based on that of a radio controlled monster truck. The top of the chassis has been modified with a flipper and release mechanism for the purposes of ejecting a payload, such as a pack of explosive material, to detonate a bomb in a controlled fashion [6]. An example of a BomBot can be seen in Figure 2-9.

Figure 2-9: BomBot robot [7]

Like the Talon, the mobility tests focused on in this work do not make use of the BomBot's bomb-disposal mechanism. In addition, the BomBot used for this work has been modified. The camera and antenna columns on the back of the robot have been removed to lower the robot's center of gravity and prevent tipping. For navigation, a small hobby camera was instead attached to the front of the robot. The camera is a generic hobby camera and transmits to a radio receiver. The camera is powered by a 9V battery, which lasts approximately 30 minutes for typical maneuvers. A photo of the camera used has been provided in Figure 2-10. The camera was taped to the front of the BomBot. The controller and screen used to operate the BomBot are shown in Figure 2-11.

Figure 2-10: BomBot camera

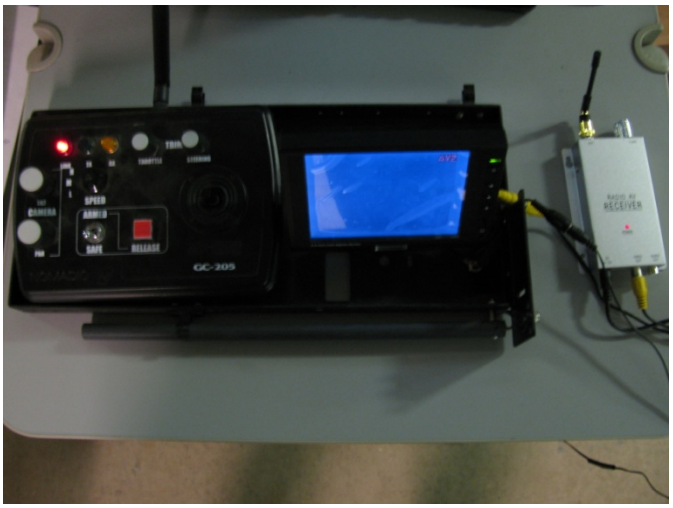

Figure 2-11: BomBot controller and teleoperation equipment

The BomBot was originally designed to operate using two 7 V RC car battery packs connected in series. However, these batteries did not allow the BomBot to operate under heavy use for more than approximately 30 minutes. To extend the operational life of the robot, the BomBot was modified. Originally a BB-2590 operating in 14.4 V mode was attached to the BomBot, but the maximum voltage of the battery proved too great for the design of the BomBot, causing the BomBot motor controller to go into thermal shutdown and cease robot operation. Instead the BB-390 military battery was used instead, an example of which is shown in Figure

2-12. This battery operates at approximately 12 V, and proved highly successful in operating the BomBot for extended periods of time.

Figure 2-12: BB-390 battery [8]

2.3 Data Logger

To record power information during a test, a data logger was attached to each robot. The data logger, developed by the Penn State ARL, records the current and voltage between the robot and its batteries at a sampling rate of 1000 Hz. From this information power consumption at any given time and energy consumed over the course of a test can be calculated. The data collected is automatically stored to a flash drive on the data logger as a comma-separated values (CSV) file and can be transferred to a computer later for processing. After approximately 2 hours of testing a maximum file size is reached and another file is created immediately. An example of a data logger is shown in Figure 2-13. The data logger is about the size of a deck of playing cards.

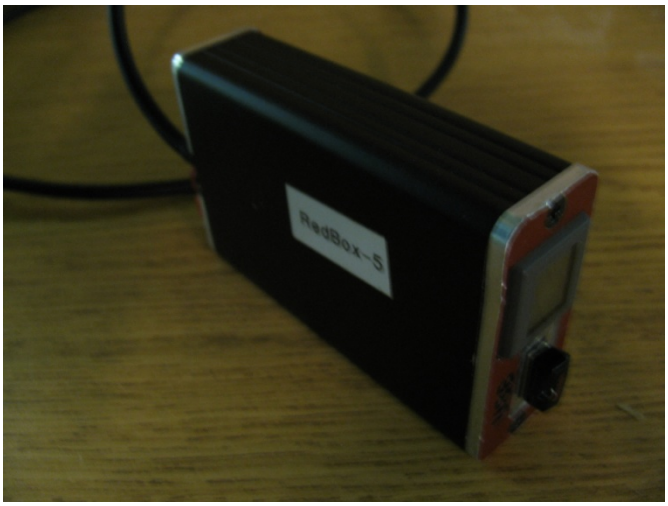

Figure 2-13: Onboard data logger

It is important to note that the data loggers do not have wireless communication capability and therefore do not communicate with the visual data acquisition system. How these two systems of data collection are synchronized is discussed thoroughly in Chapter 4.

2.4 Fiducial

In image processing, a fiducial is an object used as a marker to be identified by machine vision processing algorithms. Originally, the fiducial used was a disk of bright green construction paper approximately 8 inches in diameter. In later testing, an LED fiducial was used instead of the green disk. The LED fiducial was originally designed as a LED floodlight tool and consists of a bank of white LEDs. The LED fiducial is less susceptible to changes in lighting conditions in the testing arena. However, the LED fiducial produces a smaller cross section than the green disk. In addition, the LEDs emit the most light directly upwards. When the fiducial is seen at an angle it appears dimmer. Additionally, when the fiducial moves directly under the cameras the brightness can sometimes cause lens flare. To avoid lens flare a piece of paper was sometimes placed over the LED fiducial. Image processing results experienced with the fiducial will be

explored further in Chapter 3. Photographs of each robot with the LED fiducial are shown in Figure 2-14 and Figure 2-15, respectively. The fiducials are circled.

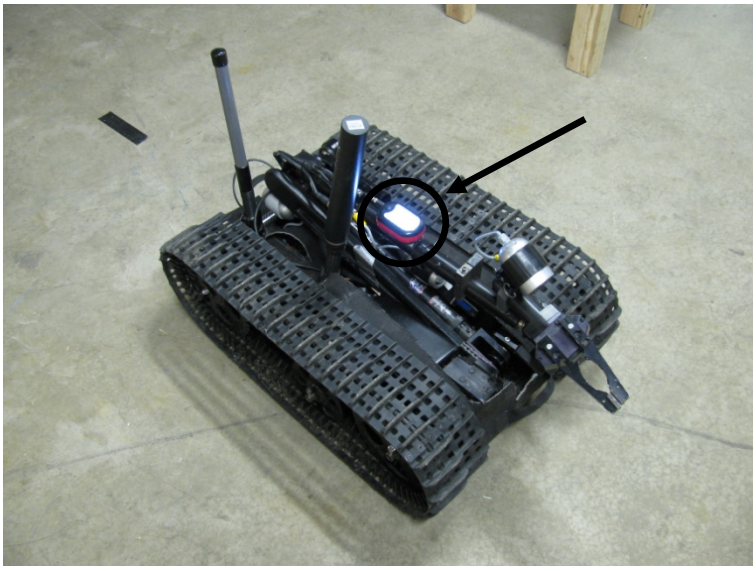

Figure 2-14: Talon with LED fiducial

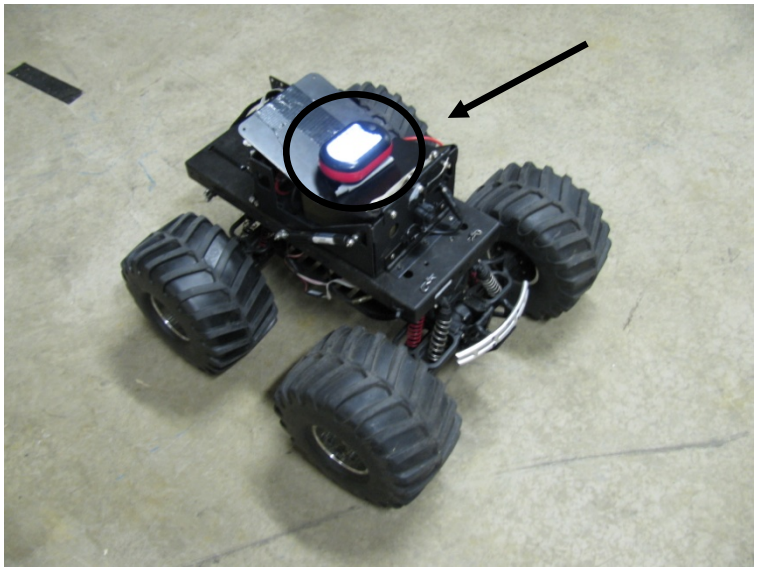

Figure 2-15: BomBot with LED fiducial

2.5 Camera System

Three overhead cameras are used to capture the extent of the testing space in the testing arena. The cameras are AXIS 216MFD network cameras, an example of which is shown in Figure 2-16. The cameras take 1.3 megapixel color images.

Figure 2-16: Overhead camera

Originally, these cameras were mounted to center beams running the length of the testing arena; the beams were attached to the lateral support arches. However, this posed a problem for testing. Robots hitting the sides of the arena during testing shook the arena's support structure. This caused the cameras to wobble and sometimes drift out of calibration. To rectify the problem, the cameras were attached to the ceiling above the arena.

Camera images from the test that prompted the decision to move the cameras can be seen in Figure 2-17. During this test repeated impacts from the Talon robot on the walls of the arena shook one of the cameras loose in its mounting, causing it to move and rotate by as much as 10 degrees, leading to errors in the image processing algorithms and robot position tracking.

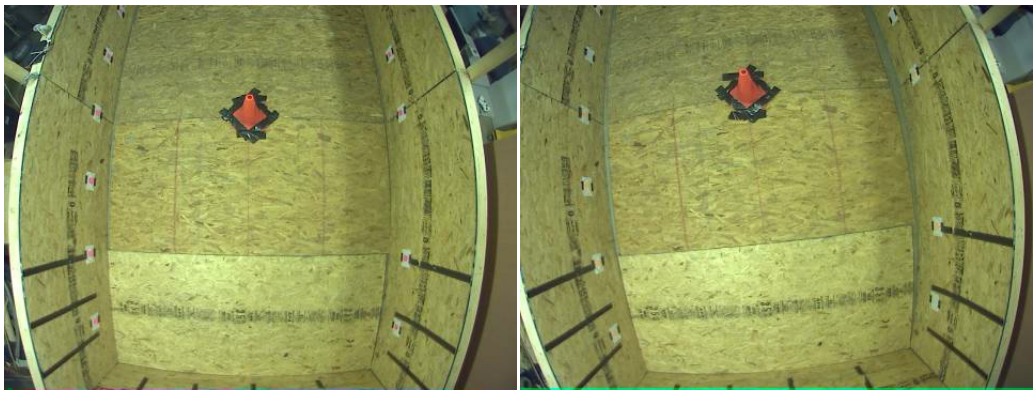

Figure 2-17: Robot test with wobbly camera, before (left) and after (right)

2.6 Ethernet

An Ethernet switch is used to connect the three cameras in the system to the computer. The Ethernet switch used is a TRENDnet TPE-S44. The cameras are powered through this Ethernet switch.

2.7 Computers

Two computer systems are used as part of the testing system. A computer running Ubuntu is dedicated to the testing arena to capture images. Ubuntu version 12.04 LTS running Python 2.7.6 was used. For the processing of images and data, a Windows 7 Enterprise PC running MATLAB version R2013a 64-bit was used. Any Windows or Mac computer can be used to process the data, but computers with more processing ability will process camera images faster. The details of the software developed for each computer system to acquire and process data will be discussed in detail in Chapter 3.

A diagram of the hardware setup can be seen in Figure 2-18.

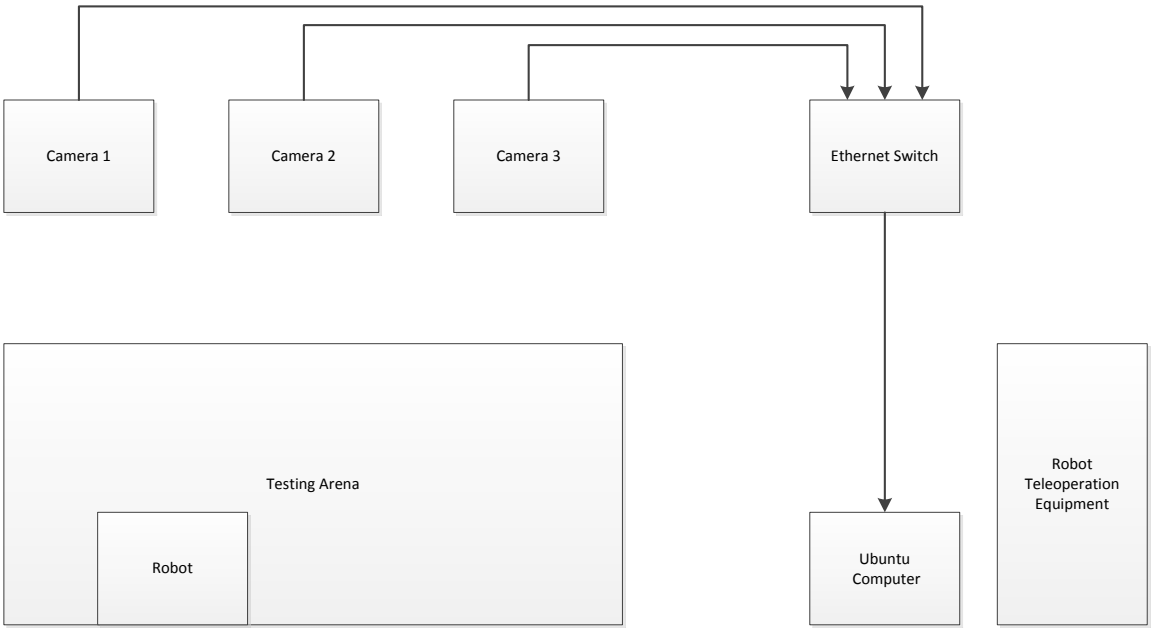

Figure 2-18: Robot testing hardware diagram

Chapter 3 Camera Capture System

3.1 Real-time Camera Collection Code

During a test, images are captured from each of the three cameras above the testing arena. Scripts written in Python are used to collect and save images in real-time. Images from the three cameras are captured synchronously at a rate of approximately 15 Hz and are stored in the jpeg image format locally. Images are color photos 480 pixels by 360 pixels in resolution. The real-time collection system was completed by Pangborn before the current work began and no significant changes were made. The complete code can be found in Pangborn's thesis [3].

3.2 Processing Overview

All data processing takes place in MATLAB after data collection is complete. The code consists of a set of scripts and functions. The main set of scripts is numbered and meant to be executed in a linear order calling various functions as needed. Each script produces an array of data passed to the next script. Non-numbered scripts were created for purposes of debugging and are executed as needed. A flowchart detailing the high level process is shown in Figure 3-1.

Scripts 1-3 were created primarily by Pangborn, while Scripts 4-6, as well as the debugging scripts, were created for this thesis. For a more detailed overview of the development of scripts 1-3, see Pangborn's thesis [3].

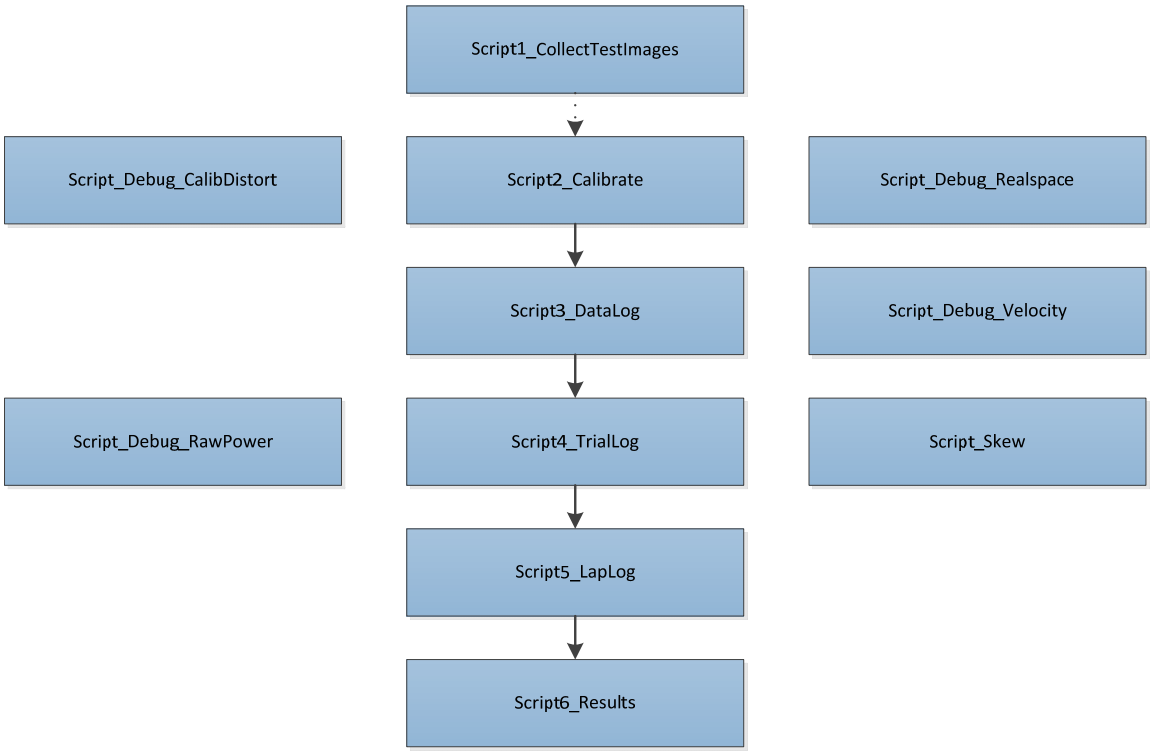

Figure 3-1: MATLAB code flow chart

Script 1 serves as a testing script if one wishes to collect images directly through MATLAB instead of using Python as described previously. Originally created in the hopes of both collecting and processing images in MATLAB, the MATLAB method of collecting images proved approximately 5 times slower than that of the Python method. However, the script is useful for purposes of debugging and the possibility of optimizing in the future. This script was developed before the start of this work and has not been modified.

Script 2 of the sequence calibrates the raw camera images. A challenge of this step is the stitching together of the three separate camera images into one image. The camera images are undistorted from their raw state and aligned using markers placed within the testing arena. In addition, a conversion is established between camera pixels and real-world distance measurements. The locations of the testing arena end zones are also established in digital space.

Script 3 is where the actual visual processing of camera images takes place. In this step a calibrated composite image of all three cameras is loaded and the script uses machine vision algorithms to search for a fiducial, or marker, within the testing space. Once identified, the centroid of the fiducial is pinpointed and its location in real space is recorded. This process continues with the next set of camera images in the sequence until all camera images have been processed. At the conclusion of this process, Script 3 produces information on fiducial position at any given time. Additionally, it calculates the total distance traveled by the fiducial as well as the lap count. Though some modifications have been made, this script is the chief processing accomplishment of the previous work done on this project by Pangborn [3].

Script 4 adds robot power information by loading the CSV files taken from the data logger onboard the robot, as well as calculating robot velocity. In addition, a metric of consistency was created by determining deviation from the most common fiducial path.

The primary function of Script 5 is to match in time the data gathered on robot position from the cameras and the power data gathered by the data logger. The task uses a data fitting technique where pauses taken by the robot between sets of laps are utilized to match the power data to the velocity data. In addition, each lap is separated individually for further processing, and Script 6 displays the results of the data in a variety of forms.

3.3 Camera Calibration

The calibration of camera images to a high degree of accuracy proved to be an unexpected challenge of the project. The goal of camera calibration was to achieve continuity of robot path between multiple camera images within 1-2 inches of accuracy. The calibration and preparation of images can be divided into four categories: distortion correction, real-world distance transformation, end zone determination, and overlap correction.

3.3.1 Camera Distortion Correction

Distortion correction from the raw images is the first step in processing. An example of an original distorted camera image for each camera has been provided in Figure 3-2 .

Figure 3-2: Raw camera images

In previous work on this project, certain parameters, such as skew coefficients, focal lengths, and distortion matrices, were generated through ROS and the OpenCV camera calibration toolbox. In an effort to keep all processing internal to MATLAB, a different method was used wherein image transformations were applied to each image in MATLAB. The method internal to MATLAB is more user friendly and more easily adjustable.

For first time use, it is necessary to generate appropriate parameters input into each image transformation. This was done by manual calibration of sample images until desired results were achieved. More automated methods of calibration parameter generation are conceivably possible, but were not deemed worthwhile here. As the cameras and testing arena remained in place for the rest of testing after calibration, the distortion parameters remained the same throughout. If the cameras or testing arena are moved, redetermination of distortion parameters would be necessary.

The undistortion process proceeds as follows. First, images are shifted vertically and horizontally to center the testing arena in the image using the function *imtransform*. This is necessary for the next step, to rotate the image about its center using the function *imrotate*, for the purpose of aligning the testing arena image segments when they are eventually joined. Next, a simple barrel distortion correction function *LensDistort* was found online, developed by Jaap de

Vries [9]. This function corrects for radially symmetric barrel distortion about the center of an image, and the magnitude of the correction is governed by constant *k*. The function is based on a simple quadratic model of radial distortion, as shown in Equations 3-1 and 3-2.

$$\begin{bmatrix} u_d \\ v_d \end{bmatrix} = (1 + \ k\, r^2) \begin{bmatrix} u - u_0 \\ v - \ v_0 \end{bmatrix} + \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} \quad [3\text{-}1]$$

where

$$r^2 = \ (u - u_0)^2 + \ (v - v_0)^2 \quad [3\text{-}2]$$

In these equations, u and v represent orthogonal axes of an image. u_0 and v_0 are the coordinates of the camera optical center, r is the radial distance from the optical center, and k is the distortion constant previously mentioned. u and v represent the raw distorted coordinates in an image, and u_d and v_d represent the desired undistorted real-world coordinates [10].

In addition, the functions *cp2tform* and *imtransform* are used in conjunction to generate a skew correction. Four points are selected by the user in the image as vertices of a quadrilateral, and four points new point are selected as vertices of a desired quadrilateral. The entire image is then transformed based on this correction. For the testing arena, the wall markers were selected as easy points of reference for this correction. Lastly, the image is trimmed and rotated if necessary to return to the desired 360 by 480 pixel resolution. The progression of image transformations on a sample image can be seen in Figure 3-3, progressing from left to right along the top row of images, then left to right along the bottom row

The distortion parameters necessary to input for each camera include the number of pixels to shift the image vertically and horizontally to center it, the degree of rotation of the image, the constant associated with barrel distortion correction, and 4 input and 4 output points for the skew transformation.

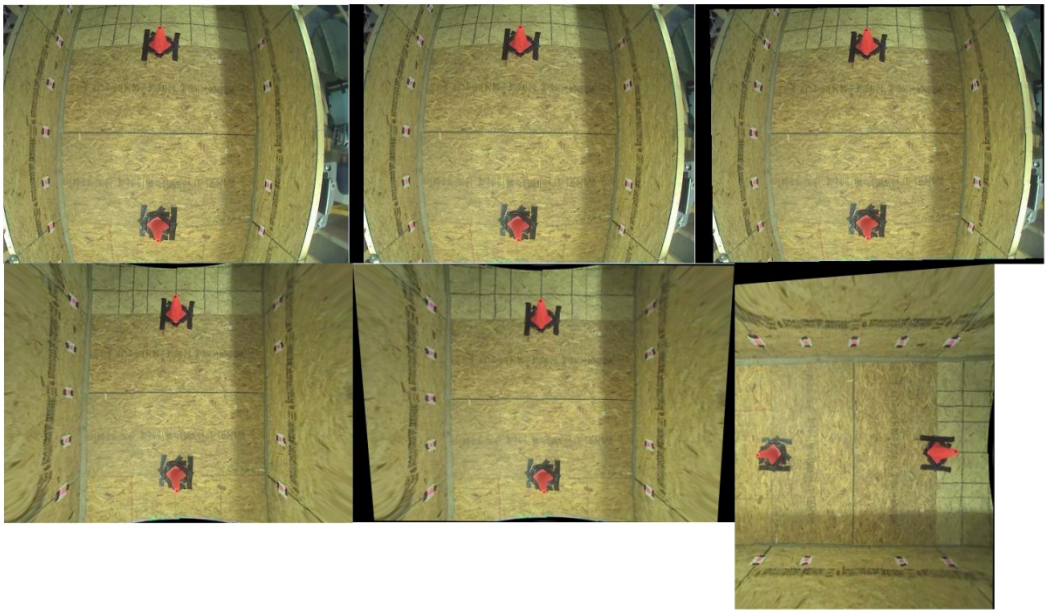

Figure 3-3: Distortion correction image transformation sequence

3.3.2 Lookup Table Generation

After appropriate distortion corrections were determined for each camera, a lookup table method was utilized to correct for camera distortion during subsequent testing. The lookup table method performs sample image transformations on each pixel in a test image and saves the mapped pixel location after the transformations take place, generating a pixel mapping table that can be called instead of performing future sequential image transformations. This technique was utilized to speed processing by approximately a factor of five as compared to performing the previously described sequence of image transformations on each camera image at every iteration. However, the creation of the lookup table itself took approximately 30 hours and distortion corrections cannot be quickly modified. A comparison of a sample image created using sequential image transformations and an equivalent image generated using the lookup table method can be seen in Figure 3-4. The third image is a subtraction of the first two images together, therefore the

dark image that result indicates the first two images are very similar and the comparison is a success.

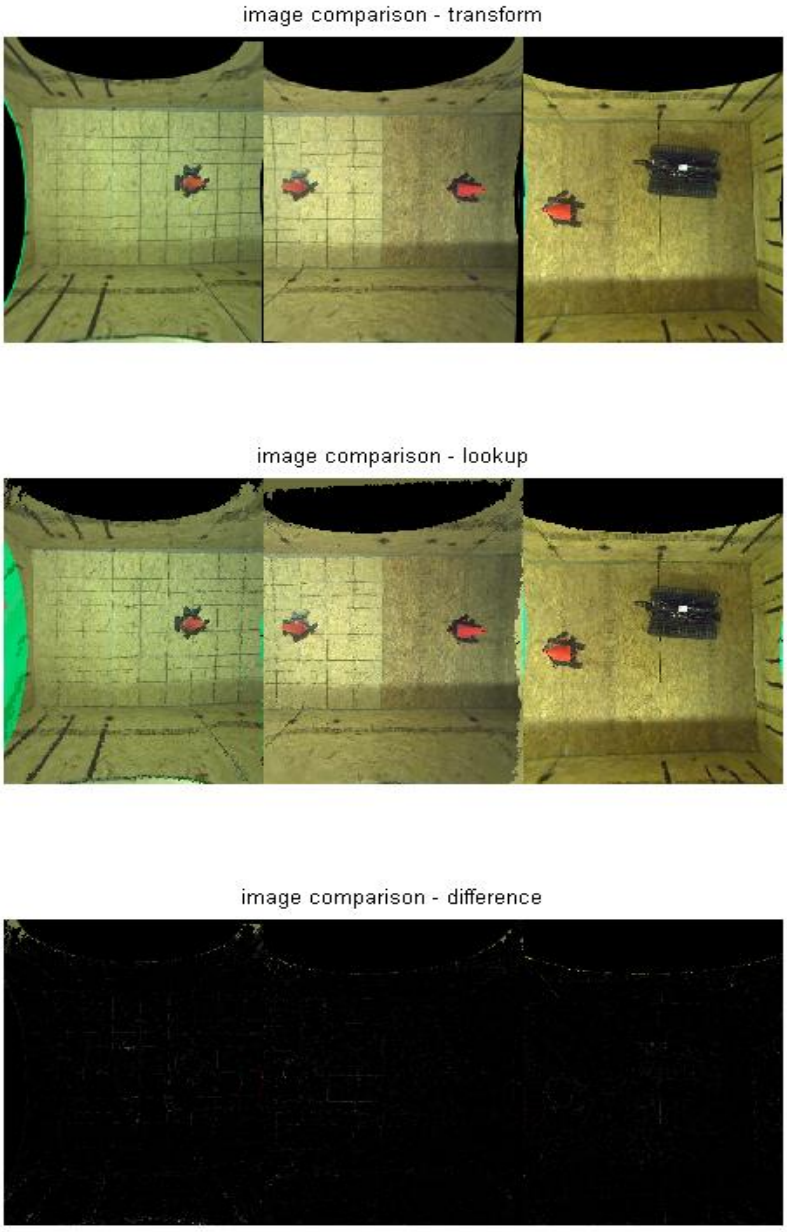

Figure 3-4: Lookup table validation: original image transformation (top), lookup table (middle), and comparison by subtraction (bottom)

3.3.3 Real-World Distance Transformation

Following the completion of distortion correction calibrations for each camera, it is necessary to map the pixel space in each image to a real-world coordinate frame. This processing, performed on each image, will have the dual purpose of mapping each image to a coordinate frame with real-world lengths and mapping all three images to a shared space. This process was originally developed by Pangborn. More details on the accuracy of this system can be found in Pangborn's thesis in Section 3.3.2 [3].

The code for this process remains unchanged from the original code. However, the physical markers used for the calibration were updated. Originally, crosses of black duct-tape were placed at semi-random points along the walls of the testing arena. Their locations were then measured and used for calibration. In this work, the method was updated by placing colored paper squares along the walls of the testing arena every two feet at a constant height above the arena floor. This height is related to the height of the fiducial relative to the ground when it is mounted to a robot. For both the Talon and the BomBot this height was approximately 14 inches. This method presented two problems. First, the colored squares were sometimes mistaken for fiducials. Second, robots scraping the sides of the testing arena would often knock off the markers. To correct this, circles of black spray paint approximately 2 inches in diameter replaced the squares as markers, with their centers located at 14 inches above the floor. Examples of the various iterations of markers can be seen in Figure 3-5.

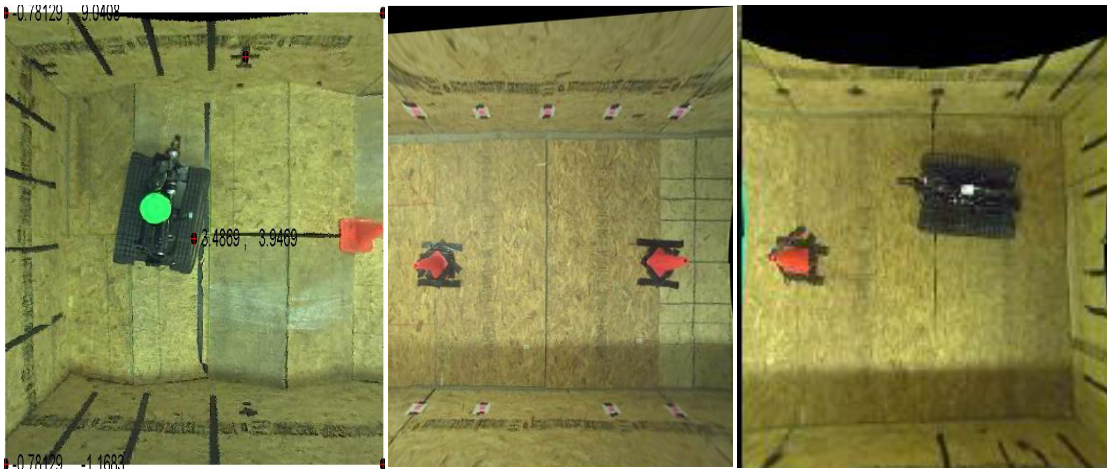

Figure 3-5: Iterations of wall markers: tape, colored squares, paint

The calibration process can be explained as follows. An image from each camera is loaded from file and the user is prompted to enter two sets of coordinates at markers, one horizontally and one vertically aligned. The real-world coordinates in x and y are then entered by the user and provide the transformation parameters necessary both horizontally and vertically to map the pixels of the image to a real-world space. An example of each image after coordinate transformation can be seen in Figure 3-6. The center and four corners of each image are labeled with real-world coordinates. The coordinates are in feet, with the origin located at the bottom left corner of the testing arena.

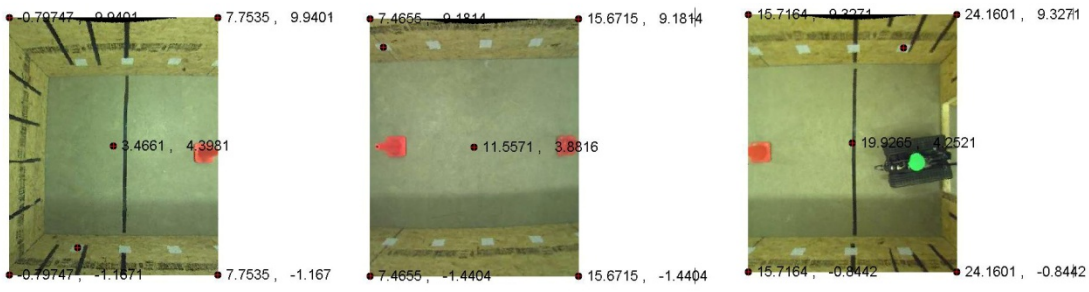

Figure 3-6: Real-world pixel transformation

The real-world coordinate transformation process allows all images to be mapped to a shared space. Though not necessary to visually display for processing, a script was created for debugging purposes to view this real-world space with images fused together – *Script_Debug_Realspace*. An example of this real-world spatial transformation can be seen in Figure 3-7. Note that the overlapping images. How to reconcile image overlap is discussed in Section 3.3.5.

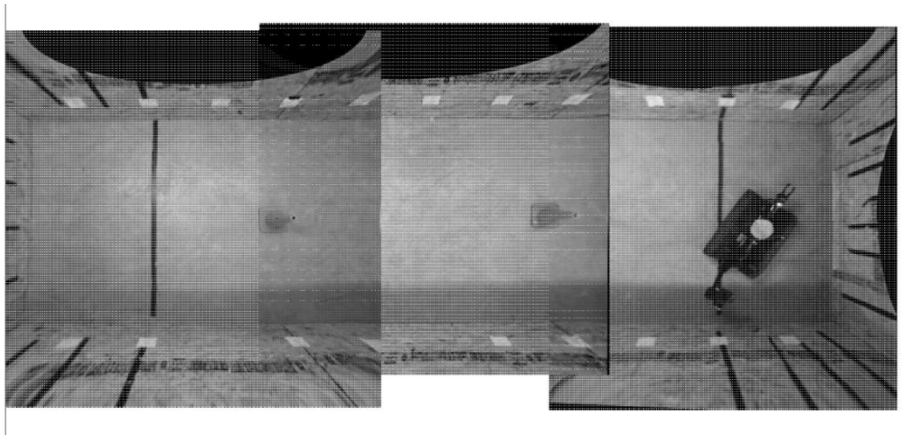

Figure 3-7: Real-world image space

3.3.4 End Zone Determination

A fairly simple calibration is utilized to determine the locations of the end zones in digital space. Two points are selected at each end zone and a line is created representing the boundary which the fiducial must cross in the image to be counted as having traveled a half of a lap. The end zone calibration code remains unchanged from the version developed by Pangborn [3].

3.3.5 Overlap Correction

The last calibration necessary before the composite image is ready for fiducial identification processing is the addition of black boxes at choice locations. As can be seen in Figure 3-7, overlap occurs when the images are displayed in a shared space. The fiducial

identification code that follows is designed to only recognize one instance of a fiducial at a time in the composite image. When the fiducial transitions between camera views, the fiducial is seen for a time by multiple cameras. The result is the appearance of multiple fiducials in the composite image, an undesirable result for fiducial identification processing. This issue was resolved by placing black bars over regions of overlap in the camera images, thus ensuring the fiducial will only be seen in one camera image at a time.

In addition, black boxes were sometimes placed over the orange traffic cones present in the testing arena. Eliminating the bright orange color from the composite image allows the machine vision algorithm to search for the robot fiducial with less error. For some tests, the cones were recognized by the fiducial in approximately 1 out of 100 iterations. An example of the digital end zones and black bars and boxes is shown in Figure 3-8.

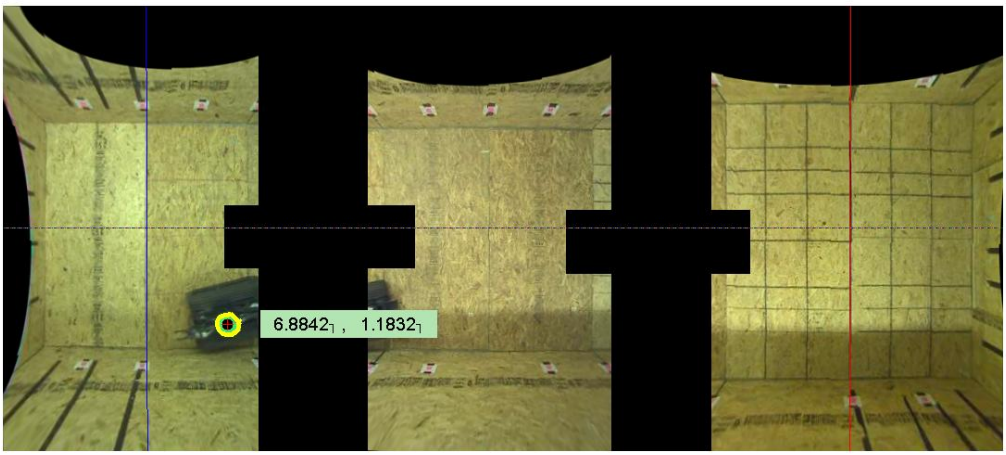

Figure 3-8: End zones and black boxes

3.4 Fiducial Identification

The machine vision algorithms for fiducial identification in MATLAB were largely developed prior to the start of this work. A summary of the method for fiducial identification is explained in the following Subsections. For a more detailed explanation of the image processing code developed to identify the fiducial in the composite image, see Pangborn's thesis [3].

3.4.1 Image Mask

Fiducial identification processing occurs in a single composite image at a time. First, a set of three images is loaded from recorded files and subject to all the calibrations outlined in Section 3.3, producing a calibrated composite image.

The next step is the creation of an image mask. The first step in the process is to convert the image to an HSV image. HSV stands for Hue, Saturation, and Value, and is an image format commonly used for image processing. The most useful image layers depend on the type of fiducial being used. The previously discussed LED fiducial is the brightest object in the testing arena and consequently extracting the Value layer from the image produces the best results. Applying a threshold to the extracted layer produces the image mask. The best layer was determined manually by comparing sample image masks extracted from each layer of the HSV image. An example of the LED fiducial and a comparison to the HSV layers can be seen in Figures Figure 3-9 and Figure 3-11. To eliminate lens flare from the bare LED a thin paper was placed over the fiducial to diffuse its light. In the value layer the fiducial is the only bright spot in the image, whereas the rest of the image is comparatively dark. This translates to the most reliable creation of a good fiducial mask. An example image mask can be seen in Figure 3-10.

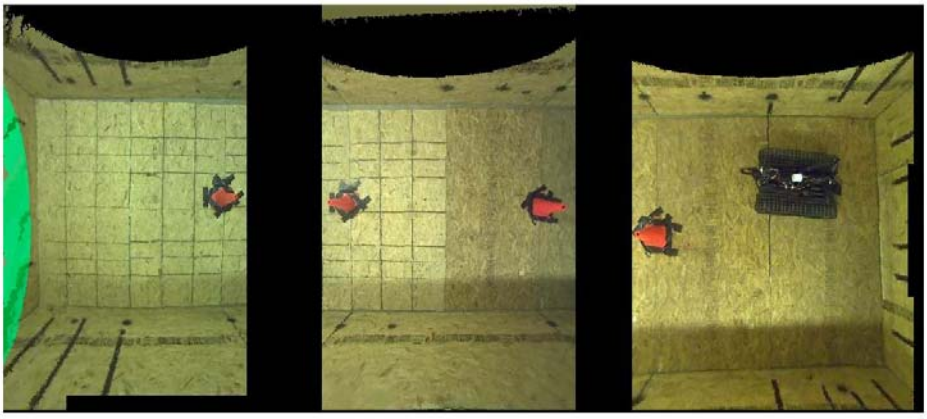

Figure 3-9: LED fiducial example, original image

Figure 3-10: LED fiducial, image mask

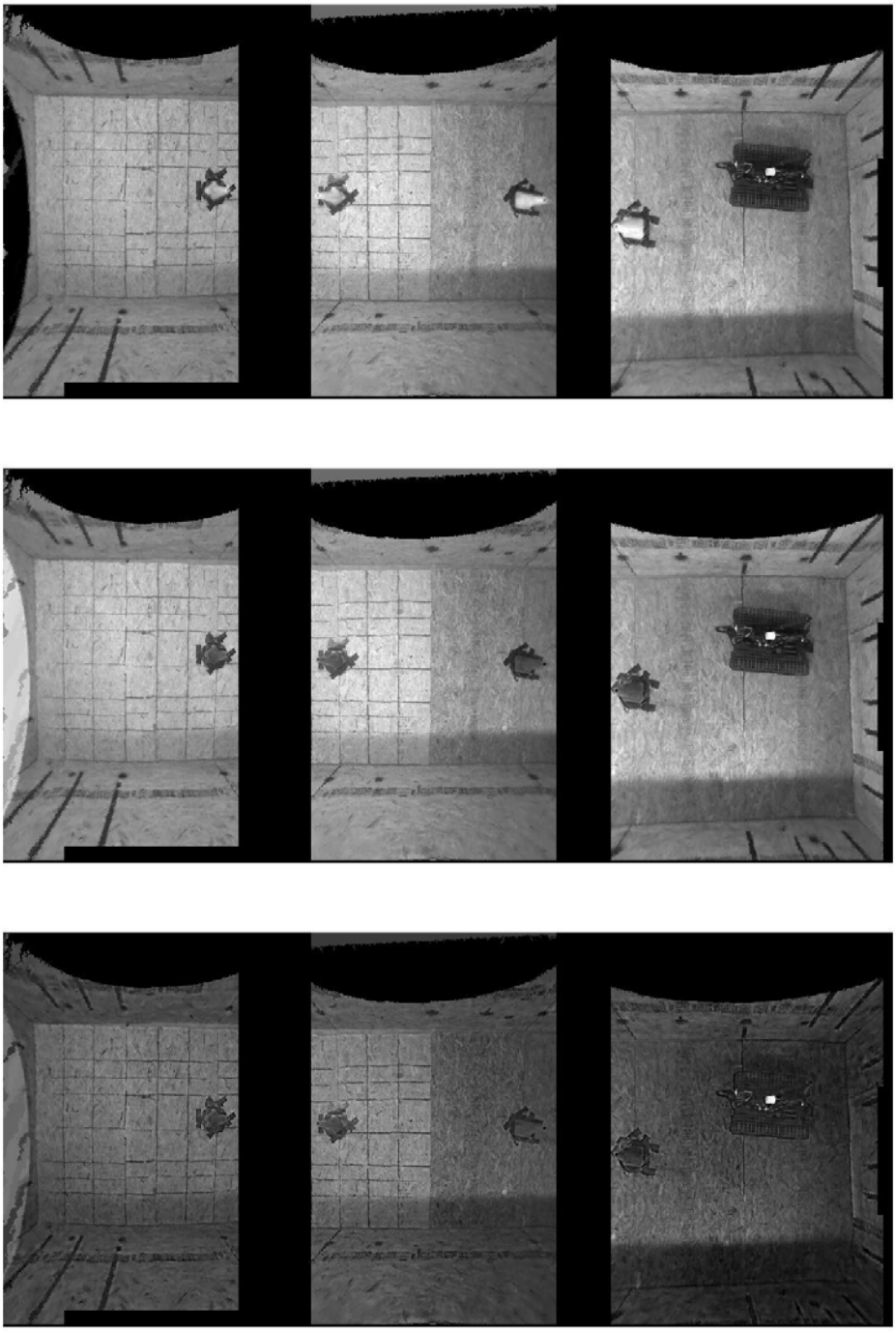

Figure 3-11: LED fiducial HSV layers: hue, saturation, and value layers (top to bottom)

Next, the fiducial mask is cleaned up if necessary. Minimum and maximum possible pixel areas of the fiducial are set; this eliminates small artifacts and ensures an entire wall of the testing arena is not mistaken for the fiducial. A morphological closing operation smoothes the border and fills in holes in the object.

3.4.2 Background Subtraction

Background subtraction was developed for this thesis as a means to enhance fiducial identification, in the hopes of making fiducial identification more reliable, as previous tests often had a fiducial identification error rate of anywhere from 1-10%. A set of reference images is taken without a fiducial present, similarly turned into a composite image, and subtracted from the test image in question. In the resulting image, the background of the testing arena is eliminated, and the robot and fiducial to stand out in the image. The reference images were typically recorded at the beginning of each test. An example of a background subtracted image is shown in Figure 3-12.

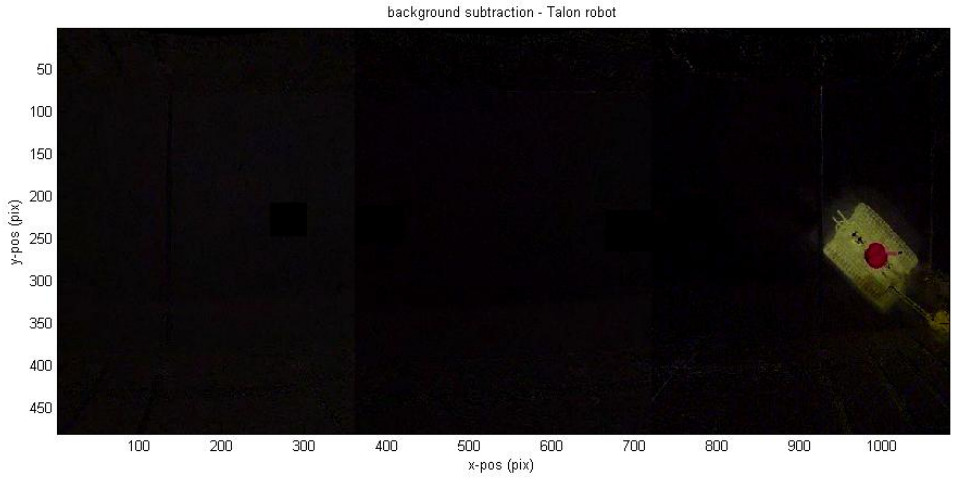

Figure 3-12: Background subtraction

Background subtraction was found useful in certain instances to more reliably track the fiducial. However, background subtraction fails when the testing arena is subject to slight shifts.

For more difficult terrains to traverse, such as the continuous pitch roll ramps, robots are more likely to crash into the sides of the testing arena. This causes the testing arena to shift and errors are introduced because the background images are no longer correct. For this reason this simple background subtraction was not used. An example of the progression of a failing background subtraction method can be seen in Figure 3-13. In this sequence, captured every 20 laps, the testing arena, particularly in view of cameras 2 and 3, began to shift, and one can see the success of fiducial tracking decrease significantly over time in these areas.

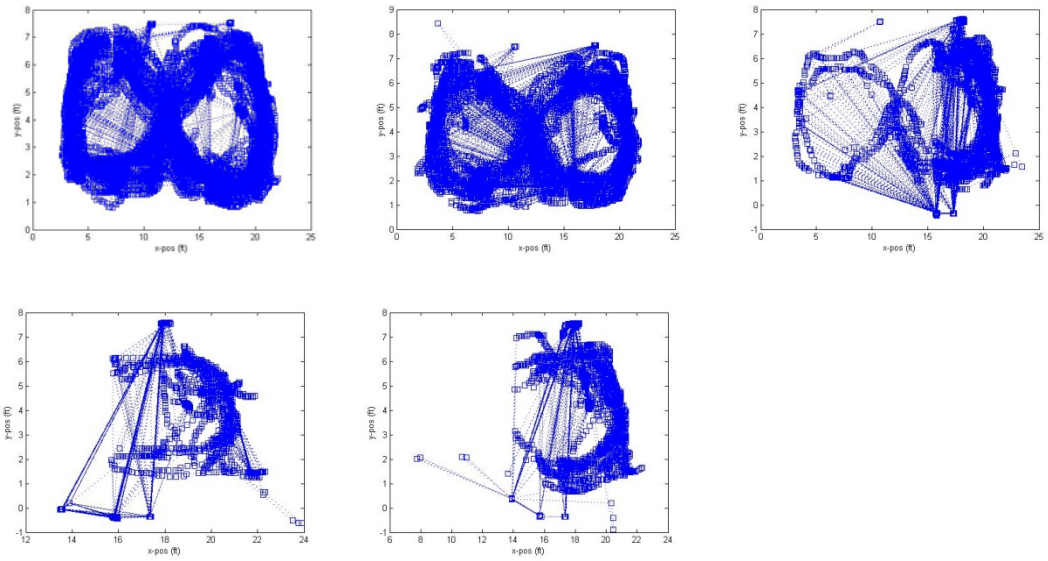

Figure 3-13: Background subtraction failure demonstration

3.4.3 Dark Testing

Both the AXIS 216MFD cameras and the Talon onboard cameras can operate in low level lighting conditions. Turning off the lights in the room of the testing arena allows the LED fiducial to be easily seen by the cameras. In addition, the chance of other artifacts in the image being mistaken for the fiducial, such as the cones, is greatly reduced. This test method was ultimately found to be the most reliable for use with the Talon robot and used for the majority of its subsequent testing. Unfortunately, the hobby camera purchased for the BomBot was found to

not operate in low lighting conditions, and so BomBot testing was conducted with the lights on. An example of dark testing is provided in Figure 3-14.

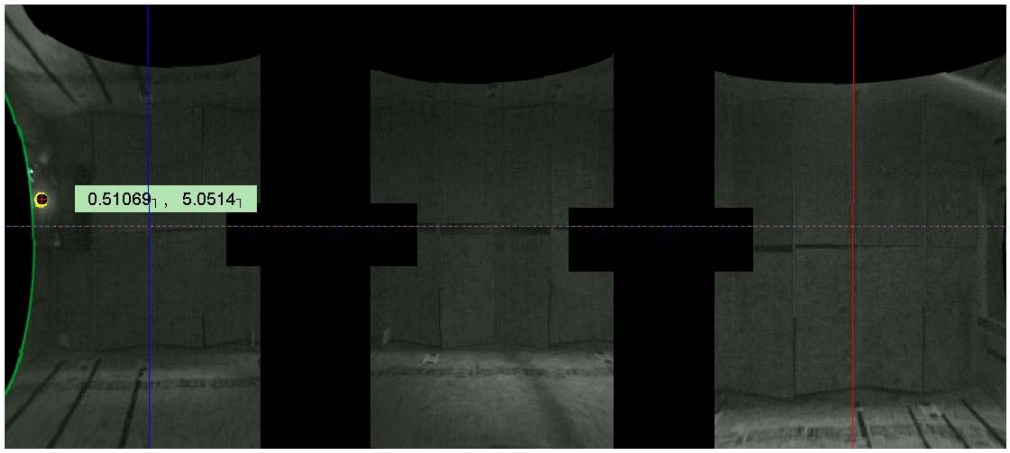

Figure 3-14: Dark testing

3.5 Determination of Robot Position

After the fiducial has been clearly identified, the centroid of the object can be easily identified using the MATLAB function *regionprops* in the MATLAB Image Processing Toolbox. This point is now treated as the effective position of the robot. The position on the image is first determined in pixels, and then, through the real-world distance calibration previously discussed, the pixel position is transformed to a real-world position. For visualization and debugging purposes, a crosshairs is placed over the centroid of the fiducial in the composite image and the coordinate position in feet is displayed. In addition, the fiducial being recognized is outlined in yellow. An example of the resulting composite image with the fiducial correctly identified is shown in Figure 3-15.

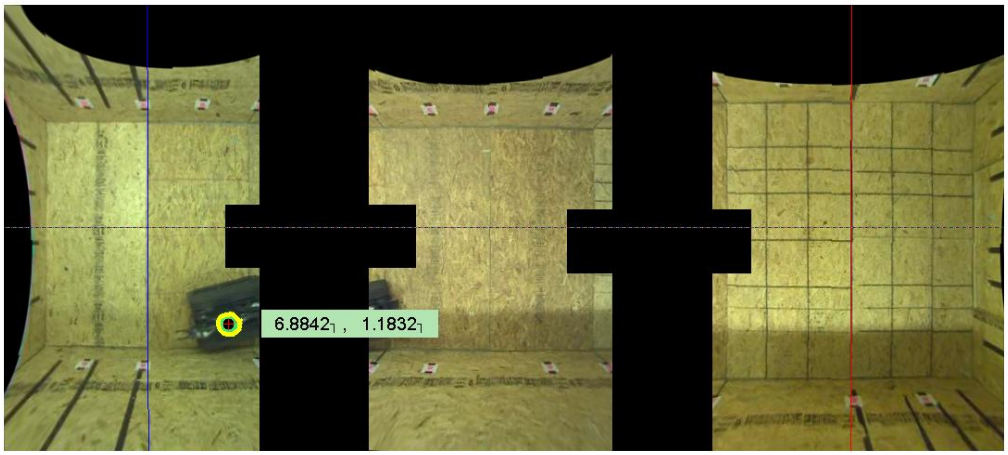

Figure 3-15: Correct fiducial identification

3.6 Plotting Images

After processing a single composite image, the relevant information is saved and the next set of images in the sequence is loaded from file. In this way the entirety of a test is processed. In processing there is the choice of whether to plot every composite image at every iteration. Not plotting at every iteration speeds processing but is worse for debugging purposes. Choosing not to plot the images speeds up the processing by a factor of approximately 2.

3.7 Resultant Image Data

After the completion of the image processing, information is saved to the output file DataLog.mat in the version developed by Pangborn [3]. Each iteration saves information to DataLog as a row. Eight parameters are stored as columns and are identified in Table 3-1.

Table 3-1: DataLog column format

Iteration	X-position (pixels)	Y-position (pixels)	X-position (ft)	Y-position (ft)	Time (s)	Laps Count	Total Distance (ft)

Column 1 records the iteration number, and columns 2-5 store the centroid position in pixels and feet as previously discussed. Column 6 lists the time since the test began, taken from the timestamp filename of each set of images. Column 7 keeps track of the half lap count, based on when the centroid crosses the end zone boundaries at either end of the arena. Using the distance formula shown in Equation 3-3 and the position between the current iteration and the previous iteration, the distance between each iteration can be calculated. Summing the distance at each iteration provides the total distance the robot has traveled, shown in column 8.

$$D = \sqrt{(x - x_0)^2 + (y - y_0)^2} \quad [3\text{-}3]$$

In Script 4, the parameter of velocity is added to the image data. A discrete derivative is performed on the position data to obtain velocity, which is further processed with a 2nd order low-pass Butterworth filter. Both the unfiltered and filtered velocity data are stored as columns 9 and 10 of a new DataLog matrix, now the first cell in the cell array TrialLog.mat.

The script *Script_Debug_Velocity* was created to assess the resulting position and velocity data. In a 2D plot, position at every iteration is plotted as a blue square and iterations are connected by a dashed blue line. Velocity data is plotted similarly, but in 3D. This step is important to verify the fiducial is tracking properly over an entire test and that the three camera images are being stitched together properly. In early testing, poor camera calibrations lead to discontinuity between camera views and inaccurate fiducial positioning. An example of a poorly calibrated 40 lap can be seen in Figure 3-16. Note the right camera image is offset both horizontally and vertically, leaving a gap between the cameras which can be seen in the position data.

The fiducial should smoothly transition from one camera view to another without large horizontal or vertical displacement. Much time was spent ensuring camera images merged properly and careful calibration is required. Using the calibration techniques developed in Section

3.3 better camera calibrations were achieved later in testing, which lead to smoother fiducial transitions between camera images. A better calibration for another 40 lap test is shown in Figure 3-16.

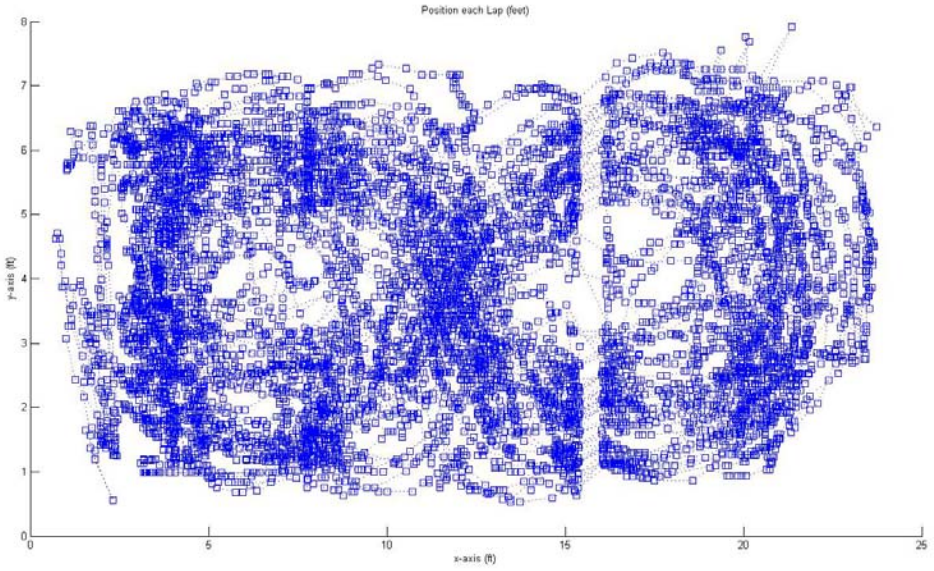

Figure 3-16: Robot position, poor calibration

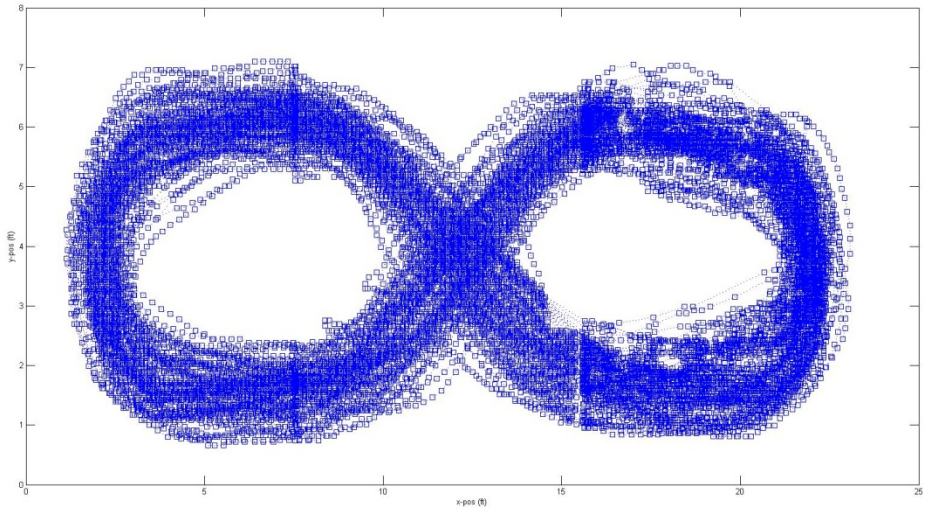

Figure 3-17: Robot position, improved calibration

As mentioned previously, a goal of this calibration effort was to achieve continuity of robot path between multiple camera images with 1-2 inches of accuracy. After data was generated using the calibration techniques previously discussed, sample position data was examined for continuity. An example of one of the worst discontinuities can be seen in Figure 3-18, which is a zoomed in view of one lap in the upper right break between camera images observable in Figure 3-17. The discontinuity leads to a shift in position both backwards and down.

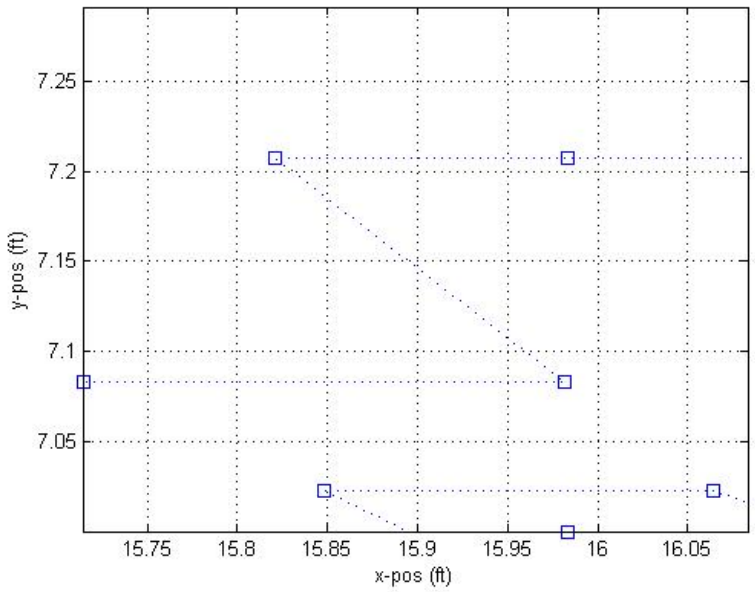

Figure 3-18: Discontinuity analysis, two point comparison

Calculating the distance between the points in Figure 3-18 using the distance formula, the distance is found to be 2.42 inches. Other discontinuities were examined and deemed less severe. This discontinuity fails the initial goal set to achieve continuity between camera images within 1-2 inches of accuracy, however it remains a reasonable calibration to work with, especially when considering distances calculated over a single lap are on the order of 50 feet. For a single test assuming 50 feet travelled, this discontinuity, multiplied by 4 for each camera transition, yields approximately 10 extra inches of distance, or 1.6% of the total distance travelled per lap.

Determination of velocity is sensitive to calibration errors resulting in poor position estimates. Any discontinuity between images leads to a jump in position data, which is interpreted as an increase in velocity by the finite differencing process. Velocity results for a 5 lap test are shown in Figure 3-19. Small increases in velocity can be seen at the transition between camera images, more so between the center and right images in this case. Overall however, the velocity for this test remains fairly consistent, between 1 and 2 ft/s. For calculation of average velocity over a lap, the brief increases in velocity should cause the average to increase only slightly, no more than 1%, considering the large dataset. Furthermore, the bias affects all laps in the same way, making trend comparisons between laps unaffected.

Also note that the effective delay in the calculated velocity due to the use of finite differencing was considering, but determined a negligible source of error when considering the large sample size.

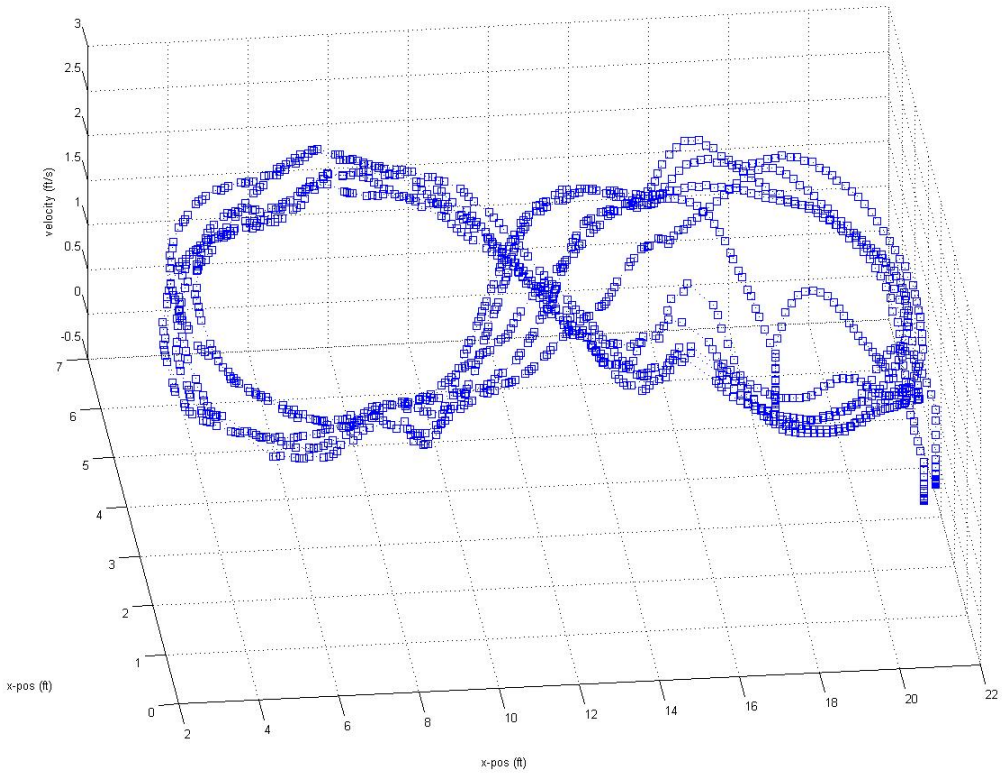

Figure 3-19: Camera discontinuity, robot velocity vs. position, 5 lap test

3.8 Path Consistency

A new metric created for this thesis is that of path consistency, in other words, the deviation from the most common path. The process undertaken to arrive at this metric is described below.

3.8.1 3D Histogram

To find the most common robot path, the first step is the creation of a 3D histogram from position data. Each bin in the histogram represents a small square area in which the robot can be located. The more times a robot enters a bin, the higher the value of that bin becomes. The most common path is based on distance and not time, therefore, the robot must exit and reenter a bin to

be counted a second time. This prevents bins from accumulating counts when the robot is paused between sets of laps.

Before position data is entered into the histogram it is first preprocessed with additional interpolated data points. In this case, 9 additional points of equal spacing were added between actual data points, to increase the effective sampling rate by a factor of 10. This helps prevent bins from being skipped over if the robot is traveling too fast for the data collection rate of the cameras. With a camera sampling rate of approximately 15 Hz and an average robot velocity between 1 and 2 feet per second, the spatial sampling of position data can be calculated, according to Equation 3-4.

$$Spatial\ Sampling = \frac{velocity}{framerate} \quad [3\text{-}4]$$

For a robot velocity of 2 ft/s and a frame rate of 15 Hz (frames/s), the spatial sampling of position is calculated to be 1.6 inches/frame. While histogram bin size was ultimately never chosen to be less than 2 inches square, the interpolated spatial sampling of position, with a value of 0.16 inches/frame, more safely ensures that no bin is skipped due to exceptionally high robot velocities.

An example of the interpolation “filling out” technique can be seen in Figure 3-20 for a sample of random data, and an example of an early histogram can be seen in Figure 3-21. This histogram has square bins 6 inches on each side. It was decided a desirable resolution of histogram would be a 2 inch bin size. The code is easily adjusted for bin size, and a histogram with this smaller bin size is shown in Figure 3-22.

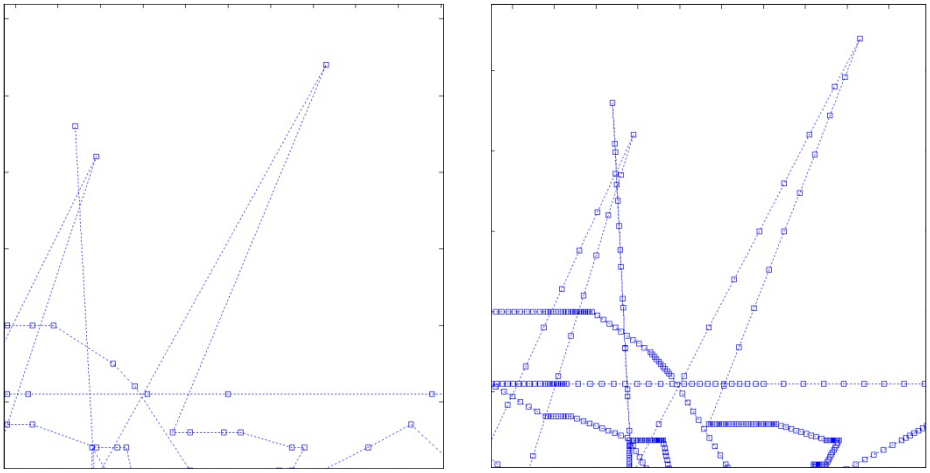

Figure 3-20: Interpolation demo, original data (left) and interpolated data (right)

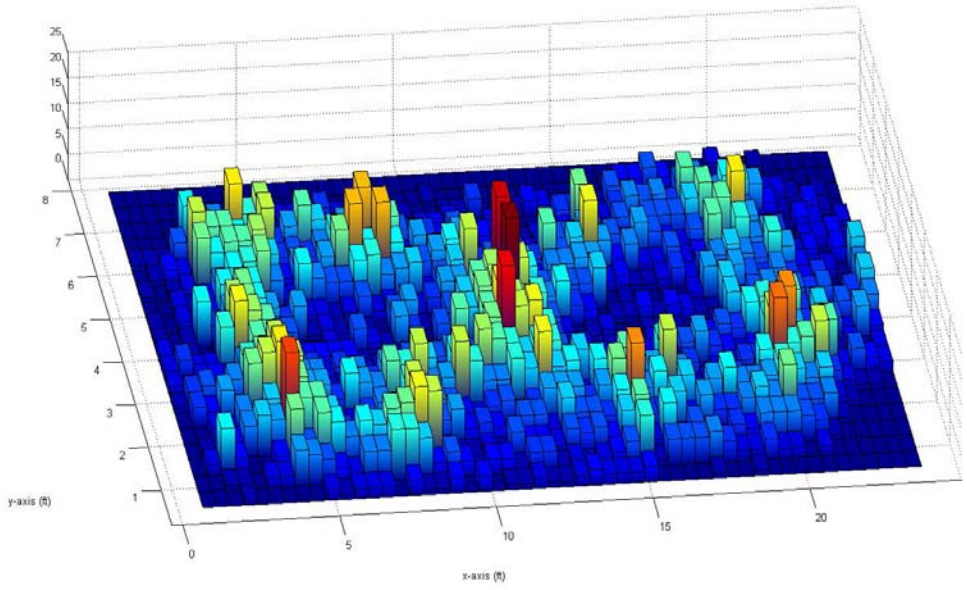

Figure 3-21: 3D histogram, 6 inch resolution

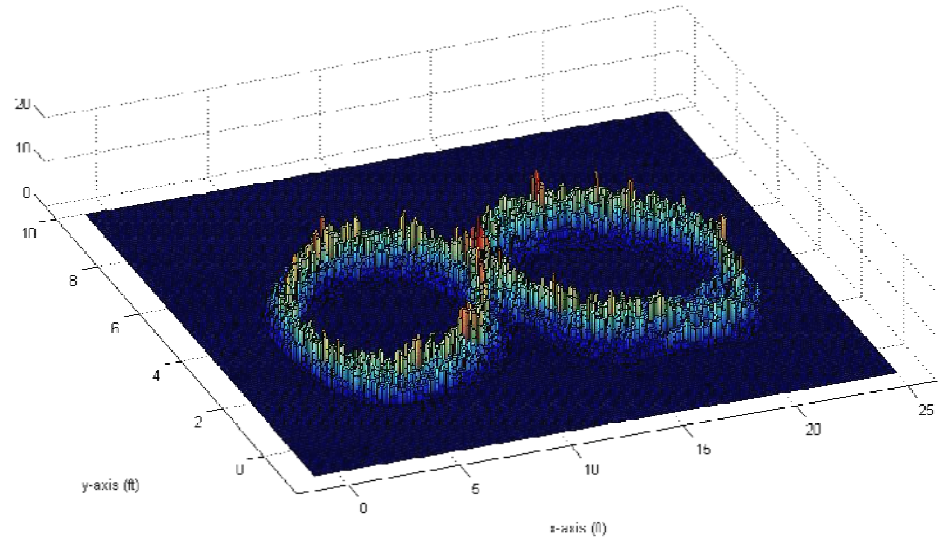

Figure 3-22: 3D histogram, 2 inch resolution

3.8.2 Watershed transformation

The next step in the process is to transform the 3D histogram to a surface plot and perform a watershed transformation on it to determine appropriate ridgelines [11].

A watershed or continental divide transformation is so named due to the fact that when water falls on a mountain range the water flows downhill from points of highest elevation. The points where the water parts and flows in different directions is known as a ridgeline [12]. The watershed transformation in MATLAB finds this ridgeline. An example watershed transformation using MATLAB's built-in function *watershed* is demonstrated on a randomly generated mountain range in Figure 3-23.

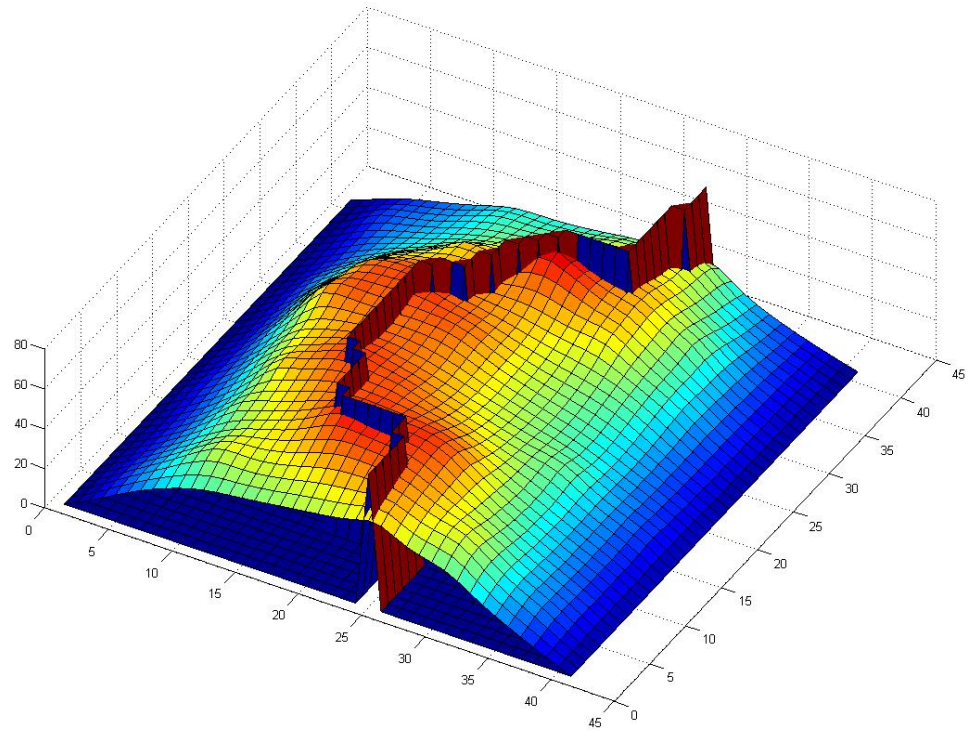

Figure 3-23: Watershed demo

For the figure-8 robot lap testing of this work, converting the3D histograms to surface plots effectively creates a topology of “mountain ranges.” During development, surface plots were first generated from histograms with low resolutions before being increased for reasons explained shortly. An early watershed transformation attempt is shown in Figure 3-24 with a low resolution bin size of one square foot. For visualization purposes, the ridgeline from the watershed transformation is extracted and displayed as a wireframe above the surface plot. As can be seen, the figure-8 pattern is clearly visible, though blocky. The next step is to increase the resolution.

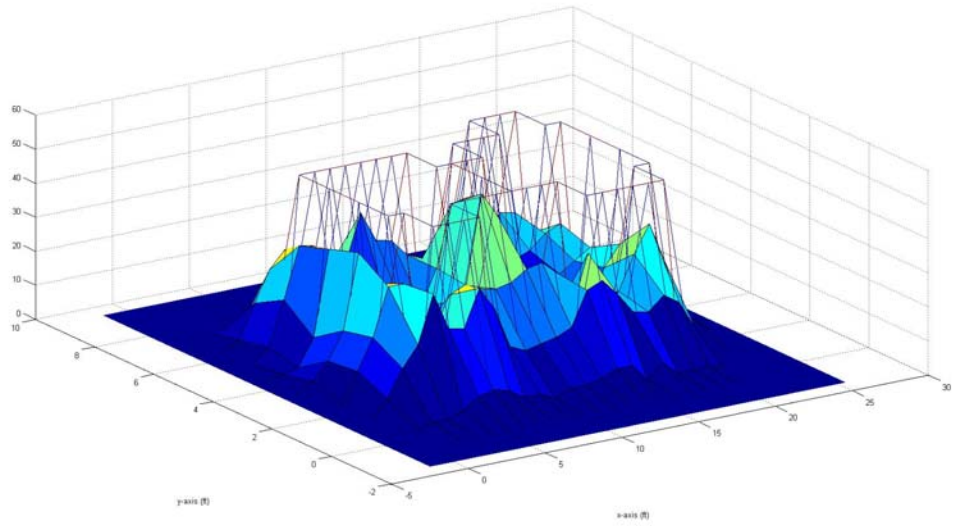

Figure 3-24: Watershed example, resolution 1 ft^2

As can be seen from a top down view in Figure 3-25, increasing the resolution poses a problem. The ridgelines no longer form a clear figure-8. This is a result of over-segmentation of the image, meaning that many unwanted ridgelines are detected. To explain further, in accordance with the mountain range analogy, water falling on the mountain range would pool into many small pools, as opposed to two large pools inside the ideal figure-8. To combat this problem, image processing techniques are applied to smooth the surface plot. First the surface plot is converted to a grayscale image. An example grayscale image can be seen in Figure 3-26. Next, a rotationally symmetric Gaussian lowpass filter is applied using the MATLAB function to blur the image, as seen in Figure 3-27. A morphological opening operation is then performed on the blurred image following by a morphological closing operation, seen in Figure 3-28 and Figure 3-29, respectively. Lastly a contrast filter is applied to the image, shown in Figure 3-30. All the steps mentioned above use built-in MATLAB functions found in the image processing toolbox. This procedure can be examined in more detail in the function *FcnPathDev*, provided in the appendix.

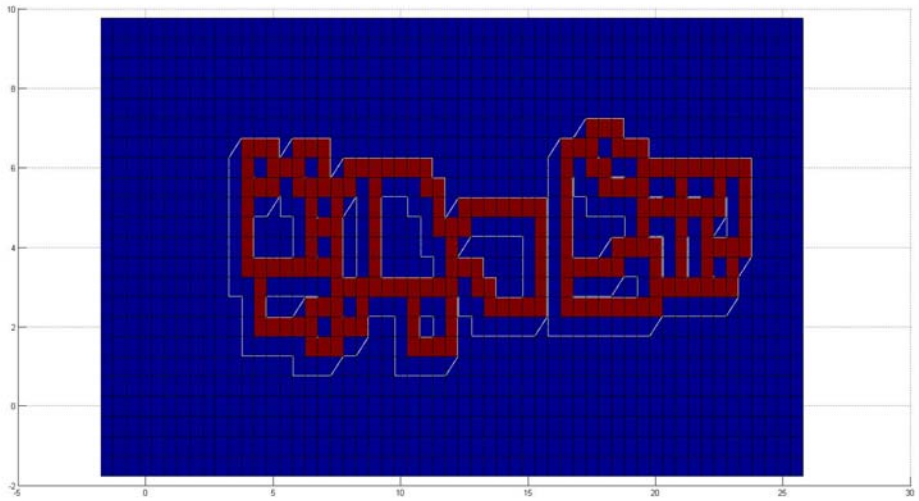

Figure 3-25: Watershed ridgeline over segmentation

Performing a similar watershed transformation as outlined previously on this filtered image produces a desirable figure-8 ridgeline. Figure 3-31 shows the ridgeline in red overlaid on the surface plot of a trial from a top down view. The ridgeline represents the most common path traveled by a robot over the course of a test. After determination of the most common path, the position of the robot at any given time during a test can be compared to this path. Using the built-in function *dsearchn*, the shortest distance to the most common path is calculated. This deviation from the most common path is used as a metric of operator consistency.

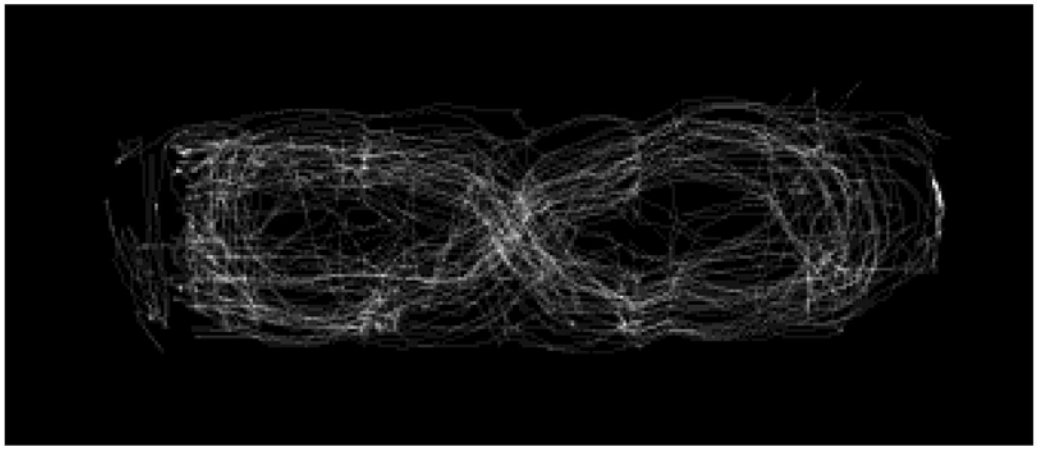

Figure 3-26: Watershed surface plot, normalization

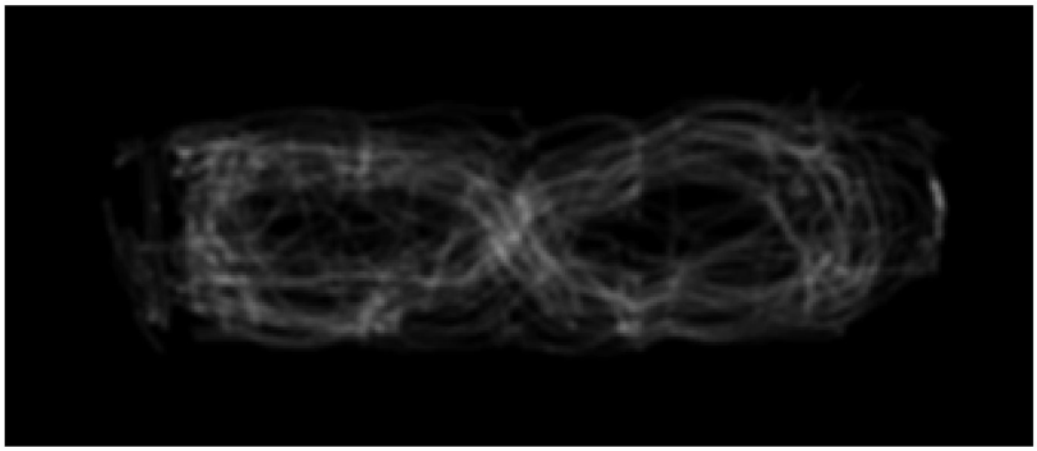

Figure 3-27: Watershed surface plot, blurring

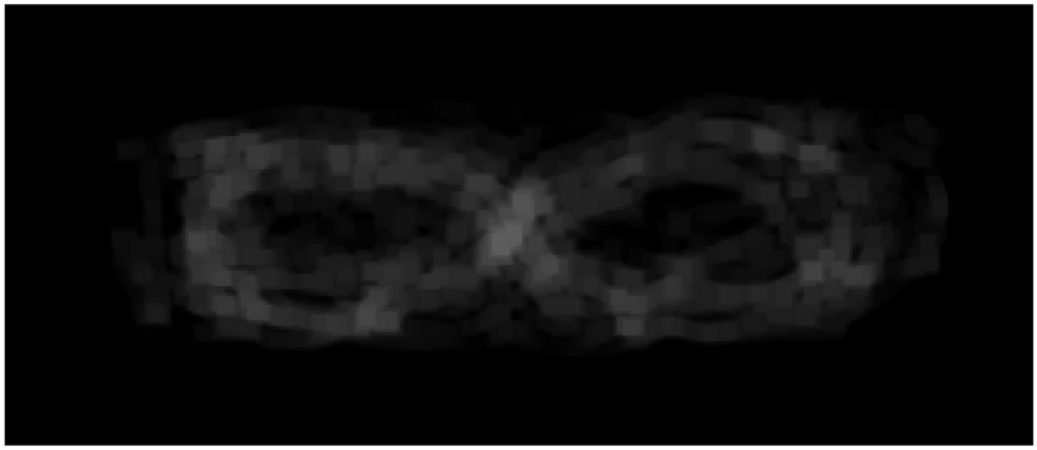

Figure 3-28: Watershed surface plot, opening

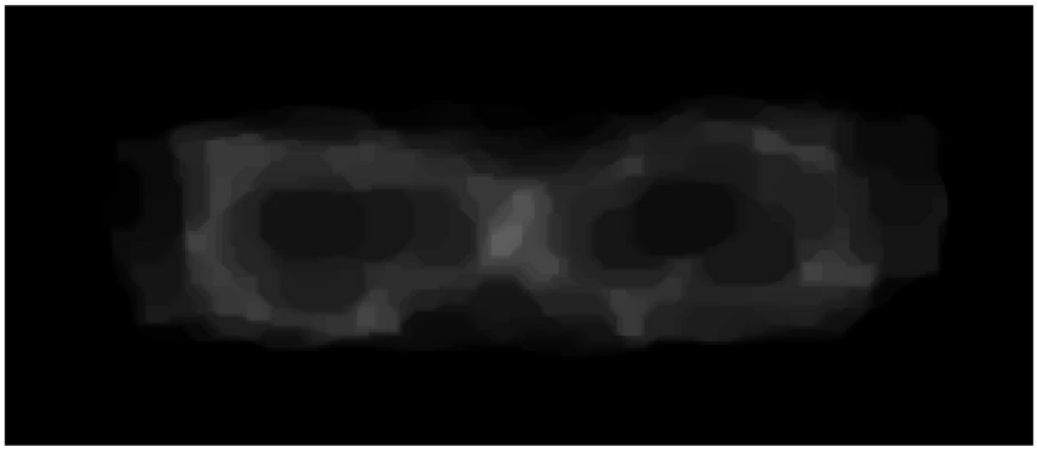

Figure 3-29: Watershed surface plot, closing

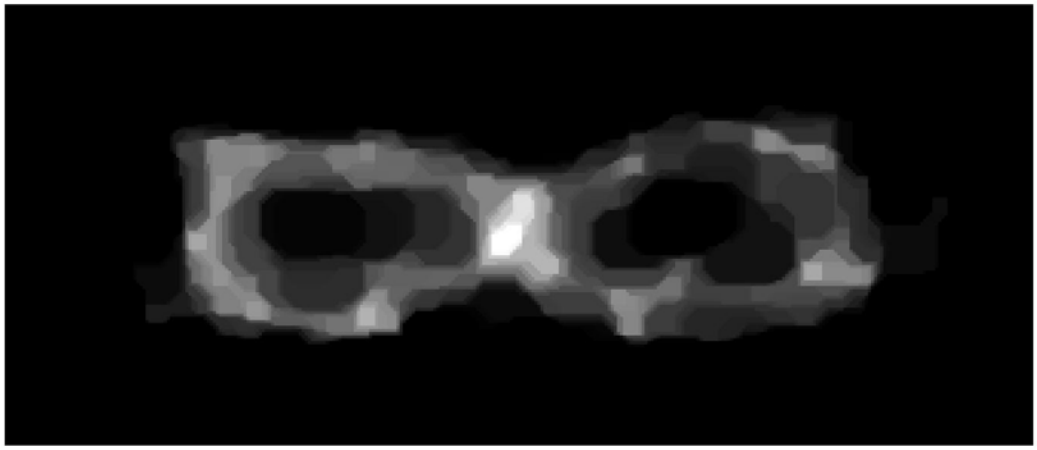

Figure 3-30: Watershed surface plot, contrasting

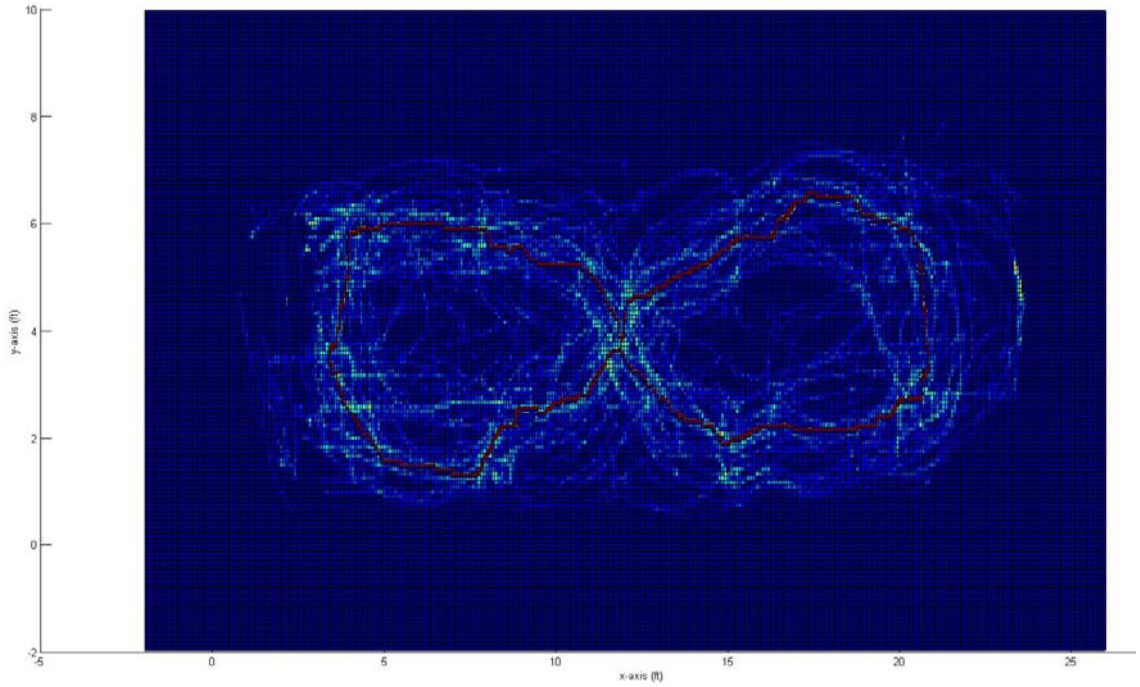

Figure 3-31: Watershed surface plot, final 1 in^2 resolution, with common path ridgeline

3.8.3 Most Common vs. Average Path

An important distinction must be made in what is meant by most common path and how this differs from the idea of the average path. To explain, the most common path determined from smoothing the surface plot follows a track of the most densely packed paths around the arena, with no concern for the influence of outlying paths. The average path on the other hand could be said to be generated with the influence of outliers. An approximate analogy between the most common and average paths would be between the mode and average of a set of numbers.

To provide a comparison between the most common and average paths, the script *Script_Skew* was created. Position data is loaded, and a similar interpolation algorithm is applied as described for the most common path algorithm to fill out the sampled data. The average path algorithm can be explained as follows. The average path is defined about two points for each side of the figure-8. An arc size, in this case 5 degrees, is defined to create a series of angular bins about each origin. The average radial distance to the origin of all position data in a given bin is calculated, and then assigned a single point at the center of the bin. A visualization of this algorithm can be seen in Figure 3-32. In addition one standard deviation above and below the average is calculated. Connecting the points yields the average path.

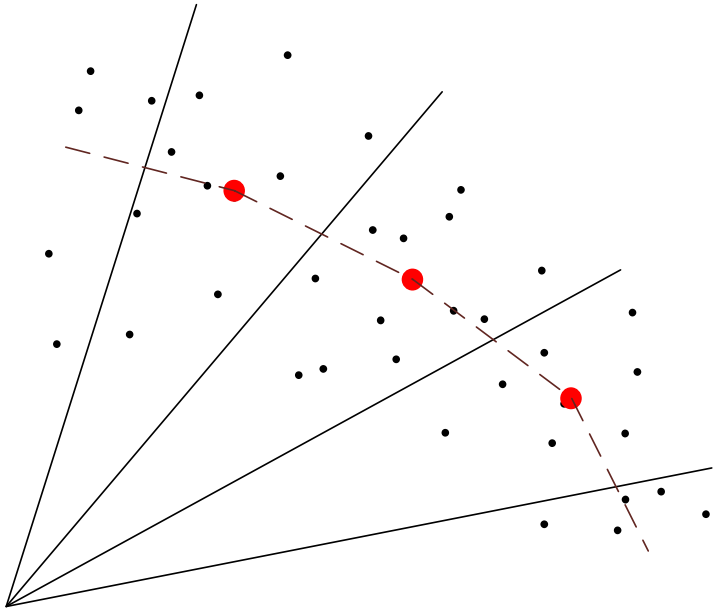

Figure 3-32: Average path algorithm diagram

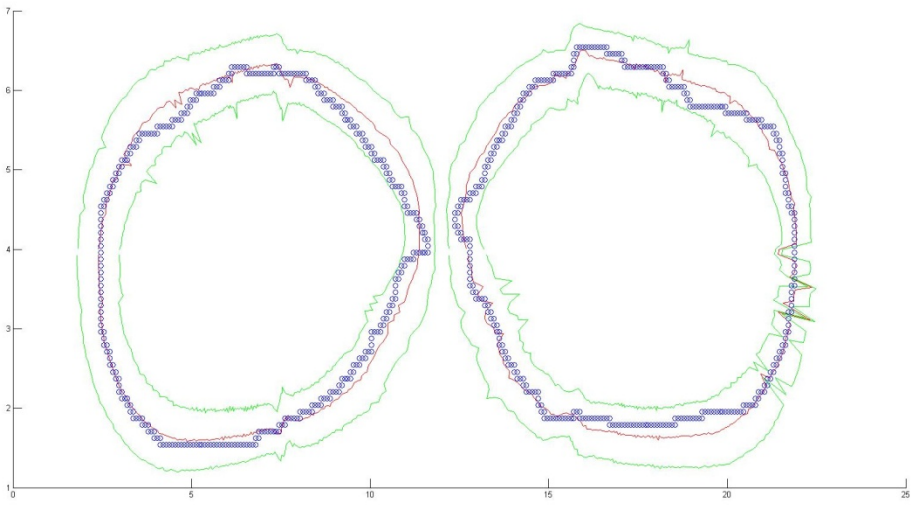

Figure 3-33: Most common vs. average paths

A comparison of the most common and average paths for a sample can be observed in Figure 3-33. The most common path is labeled in circular points, the center line is the average path, and one standard deviation on each side of the average path is also indicated. A qualitative

comparison shows the most common and average paths are very similar. Note that the average path algorithm is created for each loop of the figure-8 individually and does not model the center of the figure-8. Away from the center region, the algorithm is more accurate. This test allows validation of the most common path algorithm.

3.9 Image Processing Summary

The processing of camera images to develop metrics of robot position, velocity, deviation, and lap number formed a large portion of the development time of this project. The core of the camera calibration and fiducial identification processing algorithms were developed by Pangborn [3]. This project refined the camera calibration, and added the robot performance metrics of velocity and path deviation. The metrics developed in this chapter will be used in robot testing to quantify and assess robot performance. Two more important metrics added to the testing system, power consumption and total energy drain, will be explained in the next chapter.

Chapter 4 Energy Consumption and Syncing

One of the more challenging aspects of this project was how to best handle the collection and comparison of two independent streams of data. The camera system developed prior to the start of this work captures images of robot testing and stores the files to the local computer. An important goal of this project was the addition of power information to the set of data collected. To achieve this, an onboard data logger is affixed to the robots during testing. Difficulty arises when one considers how these two sets of data are to be synchronized in time because the data is effectively collected across two computer systems which do not communicate (starting data collection on each system at exactly the same time would be both cumbersome and inaccurate). A method was developed where appropriate synchronization of data was achieved after testing by processing both sets of data and identifying pauses in both robot motion and power consumption when the robot took a designated rest break between completing sets of 10 laps.

4.1 Addition of Raw Power Data

This data logger records current and voltage from the batteries of a robot at a sampling rate of 1000 Hz. Data is stored onboard the robot's data logger and retrieved after the successful completion of a test. Script 4 adds the raw data collected by the logger and stores both camera data and this logger data in one matrix designated TrialLog.mat, for further processing. Further processing takes places in Script 5, which is dedicated to the synchronizing of the two datasets and the division of data into individual laps, the end result of which is a new cell array designated LapLog.mat.

4.2 Grouping Velocity Data

The NIST Endurance testing protocol dictates that a 1-minute break is taken after every 10 laps in a robot mobility test (as well as a 10 minute break every 100 laps). When the robot is at

rest, the velocity of the robot is zero and the power consumption is reduced to a constant baseline value. When the robot resumes operation, both its velocity and power consumption increase dramatically. By matching the periods of rest in both the velocity and power data, synchronization is achieved.

As can be seen in Figure 4-1, robot pauses can be easily identified as breaks in the velocity data. Algorithmically these pauses must now be identified. To do this the standard deviation of velocity is calculated in 5 frame increments. When the robot is at rest, the standard deviation of velocity decreases to zero and is easily distinguished from higher standard deviations of velocity when the robot is in motion. Using standard deviation as an intermediate step achieves more reliable results than applying a threshold based on magnitude of velocity alone. A threshold is then applied to the standard deviation data, creating a binary dataset of paused or not paused robot velocity. For most tests using the Talon robot, the standard deviation threshold for velocity was set at .01 ft/s.

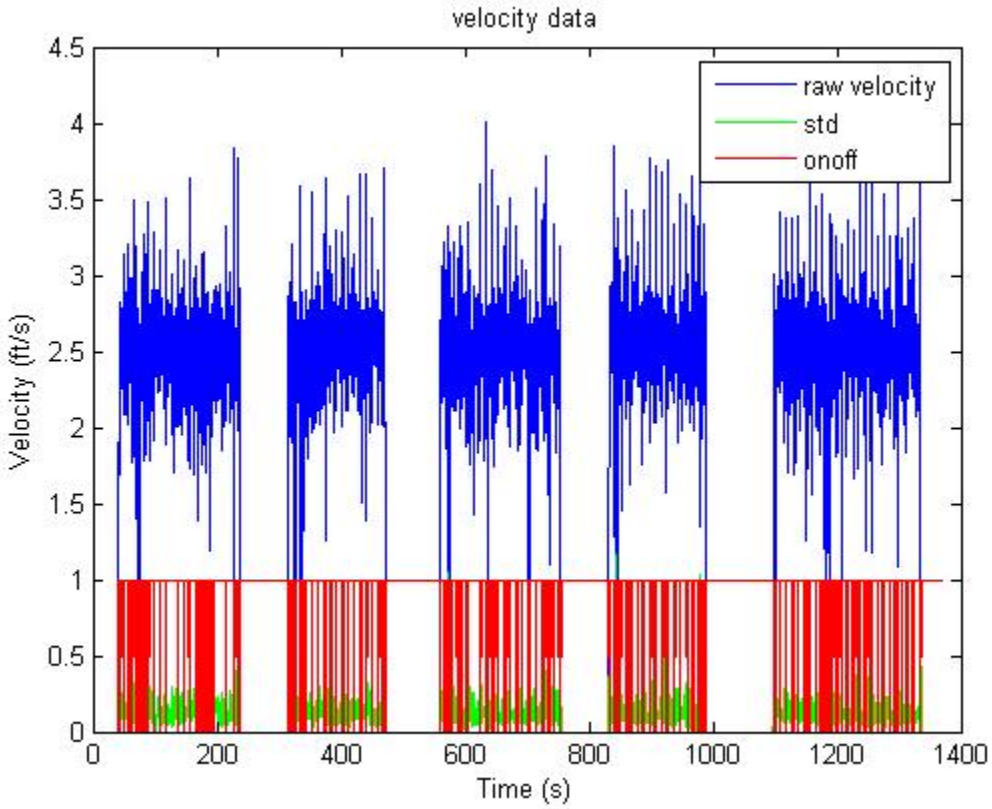

Figure 4-1: Filtering velocity data

4.3 Velocity Group Searching and Sorting

Using the robot pause information outlined in Section 4.2, camera image data can be processed with the goal of separating the data into 10 lap groups (or perhaps more or less if an error occurred during testing). Some consideration is required to achieve this algorithmically, both during a test and in processing. When the robot is at rest before the start of a test or during a break, false starts or other blips in velocity data must not be interpreted as starting a new lap group. In the algorithm, new movement from rest is discounted if robot motion lasts less than 100 seconds. Likewise, when the robot is in motion, small operator pauses must not be interpreted as rest breaks. In the algorithm, pauses in motion of less than 10 seconds are discounted as breaks. An initialization point is created to eliminate velocity data generated before the test has officially

begun, if necessary. If sections of data pass the previous tests, appropriate timestamps are recorded and used to break all image data into the resultant groups based on lap set.

4.4 Grouping Power Data

Next, a similar process as described in Section 4.2 is employed to filter the power data due to the fact that power consumption decreases to a near constant and close-to-zero value when the robot is at rest. First, the onboard robot data logger records voltage and current. A simple multiplication of these quantities yields power, as shown in Equation 4-1.

$$Power = Voltage * Current \quad [4\text{-}1]$$

Knowing the data logger samples at a constant rate of 1000 Hz, a time vector can be initialized. Raw power data can be seen in Figure 4-2. Standard deviation of power data is calculated in quarter second segments and a threshold is applied to the standard deviation data to best determine in binary form whether or not the robot is in motion or at rest. Again, it was observed that using standard deviation and applying a threshold to its magnitude could effectively separate lap pauses from robot activity due to the fact that when the robot is at rest the power nears a constant value and standard deviation over a quarter second dramatically decreases. For most tests using the Talon robot, the standard deviation threshold for power was set at 4 J/s.

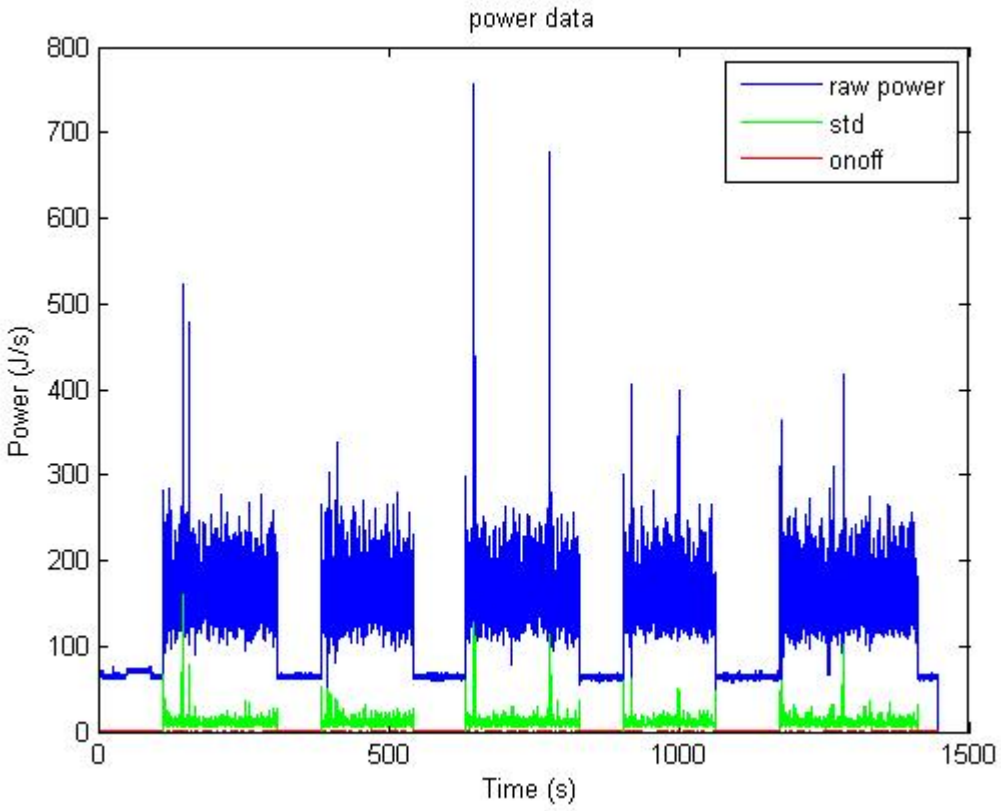

Figure 4-2: Filtering power data

4.5 Power Group Searching and Sorting

A process similar to that used to group velocity data is employed to divide the power data into groups. Note there are four parameters associated with logger data at this point: voltage, current, power, and time.

4.6 Power Drift Correction

Over the course of a test, output from the data logger was found to drift upwards. This drift can be easily seen in Figure 4-2 at periods of rest between lap groups. To correct for this drift, a mean power was calculated over a 2 second window 4 seconds before and after each lap group. These mean powers were used to generate a linear correction baseline, which was then

subtracted from each group's power data. The results of the drift correction algorithm can be easily seen in trend plots of power and energy consumption for each lap over the course of a test.

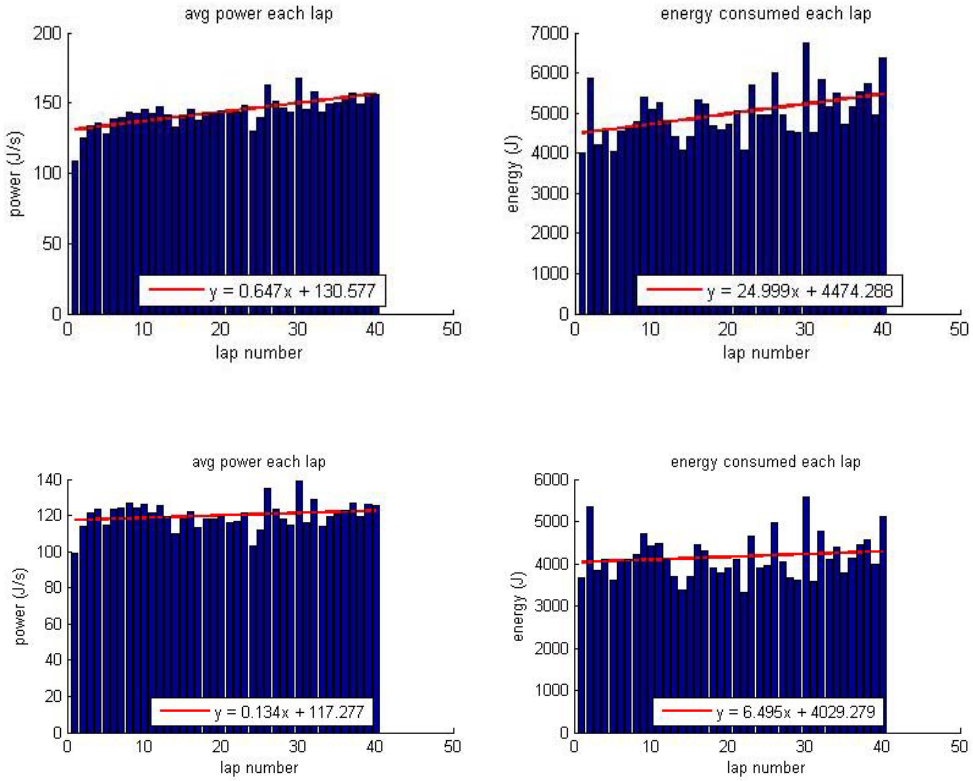

Figure 4-3: Drift correction results on power and energy trends, before (top) and after (bottom)

As voltage on a battery decreases over use, efficiency decreases, leading to an overall power increase over time as the battery is discharged. To verify that the power increase over the course of a test was caused by drift and not this affect, a 4 hour test was conducted where the Talon robot sat idle. The power results, in watts, of this test can be seen in Figure 4-4. Note the spikes in power were from deliberate small adjustments. Based on this data, due to the fact that drift still occurs under very light use, decrease in battery efficiency can be ruled out, and the drift correction is validated.

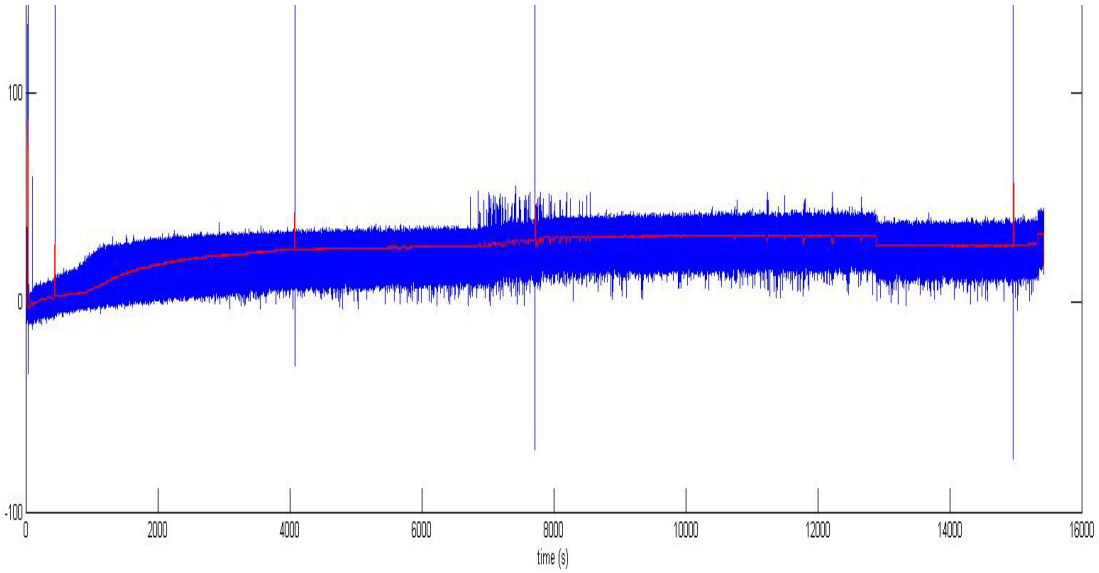

Figure 4-4: Talon 4 hour drift test

4.7 Group Pairing and Division into Laps

After dividing both camera image data and data logger data into groups, these groups can be paired. Note the assumption in this pairing that an increase in power and an increase in velocity occur simultaneously. In actuality, an increase in power causes an increase in velocity. Therefore, the increase in power would occur slightly earlier in time. Also, in the event that a wheel or tread slips, or the robot becomes stuck, power would increase without a necessary increase in velocity.

Instead of syncing the data at the beginning of each group, an experiment was conducted where the robot paused between each lap as opposed to every 10 laps. This allows synchronization to be more effectively tested by comparing the time to complete each lap according to both the camera and logger data. The times to complete each lap are presented in Figure 4-5. Error is present, ranging 0.2 seconds to 7 seconds. However, the largest discrepancies appear to be outliers – the average difference in time is on the order of 1-2 seconds.

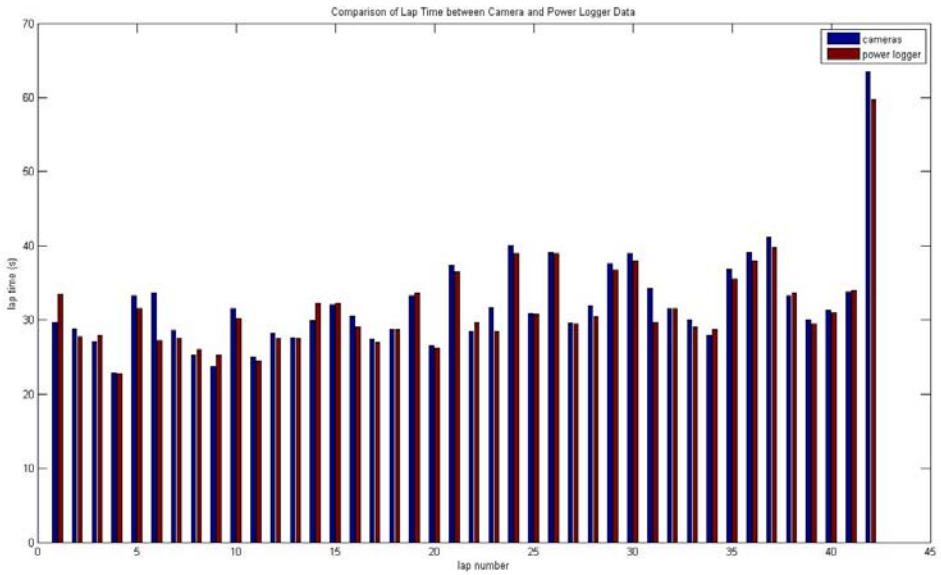

Figure 4-5: Single lap time synchronization

An option for syncing the data was to pause the robot after every lap, but this idea was rejected due to its deviation from standard NIST protocol. In terms of processing, another syncing technique considered is the matching of velocity and power data only at the onset of a test at the initial transition from rest to motion. This idea was rejected due to concerns over time drift in both data collection systems, though this error would most likely be an order of magnitude less than the 1-2 second error established above. Ultimately, it was decided that re-synchronization between continuous sets of 10 laps would yield the best results.

After both camera image data and data logger data groups are synchronized and paired as originally discussed, lap sets are broken further into individual laps. Individual laps are divided based on the camera image data lap count, increased when the robot crosses the correct end zone. To clarify, for intermediate laps in a lap set, the time markers at which a new lap begins and ends are determined by the lap count and applied to both image and power data to extract all information about a lap. For the first lap in a set, the start point is determined by standard

deviation thresholding from a stationary robot position and the end point is determined when the robot crosses the end zone. For the last lap in a set, the end point is whenever the robot crosses the end zone. Note there are usually 1-2 seconds of data between when the robot crosses the end zone and when the robot comes to rest for a break which is technically in the next lap. However, the robot must take a break before continuing on to this new lap set, so this data is discarded.

4.8 Addition of Lap Specific Data

After individual laps have been extracted, they are stored in a cell array. At this point the addition of several lap specific parameters is useful. From the beginning of a lap, integrating power data using trapezoidal quadrature, as seen in Equation 4-2, yields robot energy drain at any given time, as well as total energy consumed at the end of the lap.

$$\int_a^b f(x)\,dx = (b-a)\left[\frac{f(a)+\ f(b)}{2}\right] \qquad [4\text{-}2]$$

It is also useful to “zero” several parameters to the specific lap, including time and distance traveled, yielding time and distance since the beginning of the lap as opposed to since the beginning of the entire test. Every column in the LapLog is assigned to a lap, and each row contains a parameter. The list of parameters is provided in Table 4-1. Measurements from the data logger in SI units have been converted to Imperial units to remain consistent with the camera data.

Table 4-1: LapLog key

Row	Parameter
1	Iteration
2	X-position (pixels)
3	Y-position (pixels)
4	X-position (ft)
5	Y-position (ft)
6	Lap time, camera images (s)
7	Lap number
8	Total distance (ft)
9	Velocity (ft/s)
10	~~Velocity filtered (ft/s)~~
11	Common path deviation (ft)
12	Voltage (V)
13	Current (A)
14	Power (ft-lb_f/s)
15	Iteration, data logger
16	Lap time, data logger (s)
17	Lap energy (ft-lb_f)
18	X-pos, interpolated (ft)
19	Y-pos, interpolated (ft)
20	Lap distance (ft)
21	Velocity filtered (ft/s)
22	Power filtered (ft-lb_f/s)

Rows 1-8 are translated from columns 1-8 of the original DataLog matrix generated by the image processing and fiducial identification algorithms. Rows 9 and 11 are the quantities further derived from image data (Row 10 was originally used to filter velocity data but was abandoned.) Rows 12-16 are raw and derived information determined from logger data. Row 19 is the previously discussed zeroing of row 8 to achieve lap specific distance traveled. Rows 20-21 were created to filter and compare velocity and power information. Due to vastly different sampling rates, it is important to filter each data set appropriately before comparison.

Rows 17-18 were created to further facilitate the merging of position and logger data. The data logger records data at a constant sampling rate of 1000 Hz. Importantly, unlike the logger the cameras do not capture images at a constant sampling rate. Instead, the python code

written to capture images in Ubuntu is designed to capture images as fast as possible. An approximate sampling rate can be determined by comparing the time stamps between images at every iteration. The sample rate is not constant, but is consistent within 5%, and calculating the average yields approximately 15 Hz. This frequency is much slower than the data logger sampling rate.

To synchronize the data, each camera image was assigned to its closest data logger sample point in time. For data logger points without assigned camera data, position data was then interpolated. This interpolation follows a similar process as the interpolation for 3D histograms described in Section 3.8.1. However the interpolation is denser in this case, to ensure a sample point of position is provided to each point of power information collected. This avoids decimation of the power data. This allows data logger data to be plotted as a function of position, useful for creating 3D plots to visualize data collected from the logger.

4.9 Velocity and Power Filtering

As mentioned in the previous section, velocity and power data are filtered to facilitate comparison and observation. Filtering was performed using a second order low-pass Butterworth filter. A cutoff frequency was first chosen. Discrete-time cutoff frequency was then calculated for each set of data, which is the cutoff frequency normalized to sampling rate, as seen in Equation 4-3.

$$f_{c,d} = 2 * \pi * \frac{f_c}{SR} \quad [4\text{-}3]$$

A cutoff of 0.1 Hz was chosen to produce smooth velocity and power data which are useful to observe on the scale of the test. Keeping in mind that the cameras typically collect images at a rate of approximately 15 Hz, this produces a discrete-time cutoff frequency for the camera data of $0.013\ \pi$ Hz. Performing a similar operation using the same cutoff frequency, but

with a sampling rate of 1000 Hz for power data, a discrete-time cutoff frequency of 0.0002 π Hz was then calculated.

Note the Butterworth filtering was performed in MATLAB using the function *filtfilt*, which filters data in both the forward and backward directions, leaving the data with zero phase distortion.

Chapter 5 Robot Testing

5.1 Goals of Robot Testing

The goals of robot testing were to demonstrate the viability of the improved testing system for general use in ground robot mobility testing, as well as to research the performance of one robot more specifically, the Talon robot. As previously described, the testing arena can accommodate multiple terrains. Tests were first attempted using both the Talon and the BomBot on ramps in the continuous pitch/roll configuration. Both robots were damaged during initial testing with the pitch/roll terrain configuration so additional tests were completed on smooth concrete or flat OSB board surfaces. Problems collecting data using the BomBot, discussed further in Section 0, made the Talon the robot subject to the majority of testing. Testing conducted to research the Talon robot, with a single operator under a variety of conditions, is outlined as follows.

Four tests were carried out using the Talon robot: a 50 lap test on OSB, a 100 lap test on OSB, a 50 lap test on concrete, and another 50 lap test on concrete but driving in the reverse direction than typically tested. All tests were carried out by the same robot operator between March 26 and 27, 2014. The Talon was loaded with four BB-2590 batteries for each test.

5.2 Format of Results

From the LapLog output file, many useful plots and statistics from a test can be extracted. Plots produced can be divided into three general categories: as functions of time, as functions of position, and as functions of laps. First, plots tracking various parameters as functions of time for each lap can be generated. Laps are colored differently to allow easy differentiation visually. These plots can be found in Section 5.3. Other plots possible are 3D plots showing various

parameters as functions of position, as seen in Section 5.4. Individual laps are again colored for convenience.

While the proceeding plots can be very useful for visualization purposes and debugging, the most quantitatively useful information can be presented as bar graphs. When lap information for various parameters are graphed as bars, trends over the course of a test can be easily observed. Adding a line of best fit to these graphs allows the trends to be quantified. These plots can be found in Section 5.5. Final statistics for a test are presented in the matrix ResultStats.mat. Parameters included in this matrix are provided in Table 5-1. Final statistics produced from robot testing in this thesis are presented in Section 5.6.

Table 5-1: Resulting statistics key

Row	Parameter
1	Total laps completed
2	Total time (s)
3	Total distance (ft)
4	Total energy (ft-lb_f)
5	Lap time avg (s)
6	Lap time std (s)
7	Lap distance avg (ft)
8	Lap distance std (ft)
9	Lap velocity avg (ft/s)
10	Lap velocity std (ft/s)
11	Lap deviation avg (ft)
12	Lap deviation std (ft)
13	Lap power avg (ft-lb_f/s)
14	Lap power std (ft-lb_f/s)
15	Lap energy avg (ft-lb_f)
16	Lap energy std (ft-lb_f)

5.3 Presentation of Robot Testing Results: 2D Plots

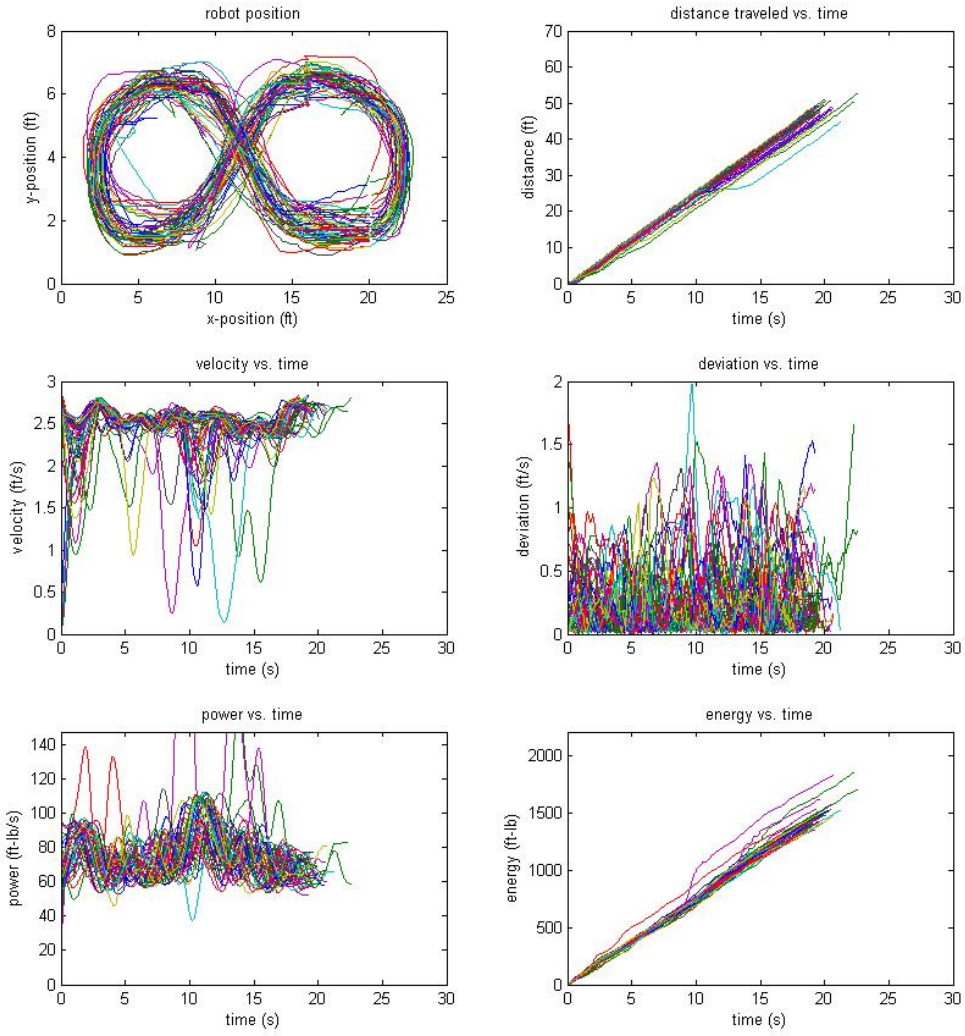

Figure 5-1: Lap plots, Talon 50 lap OSB test

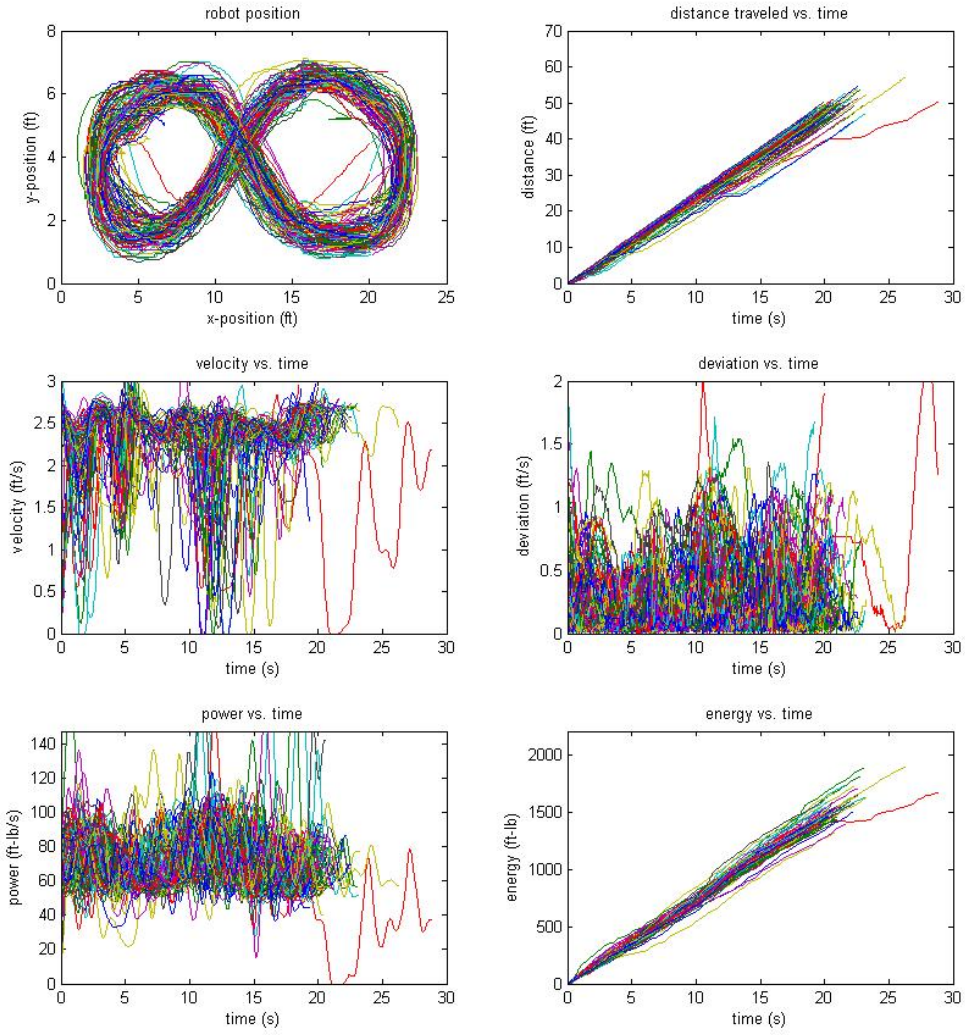

Figure 5-2: Lap plots, Talon 100 lap OSB test

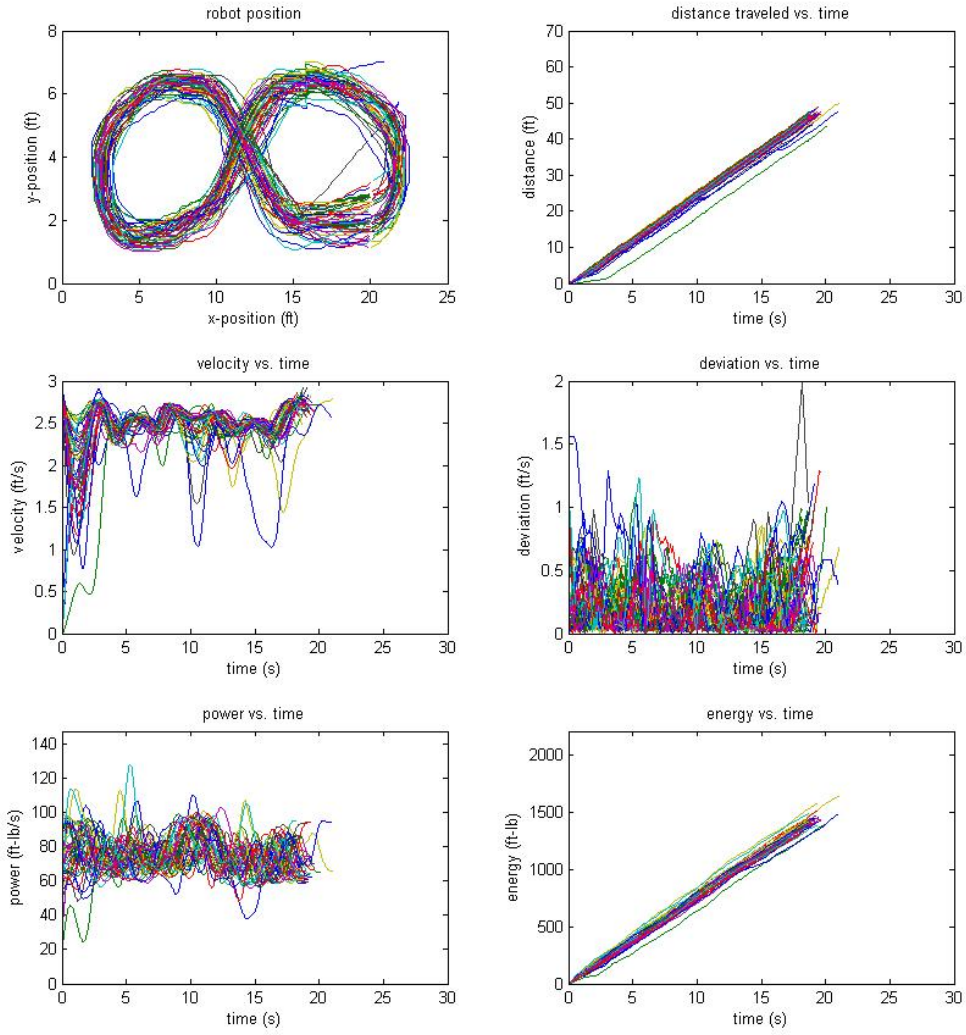

Figure 5-3: Lap plots, Talon 50 lap concrete test

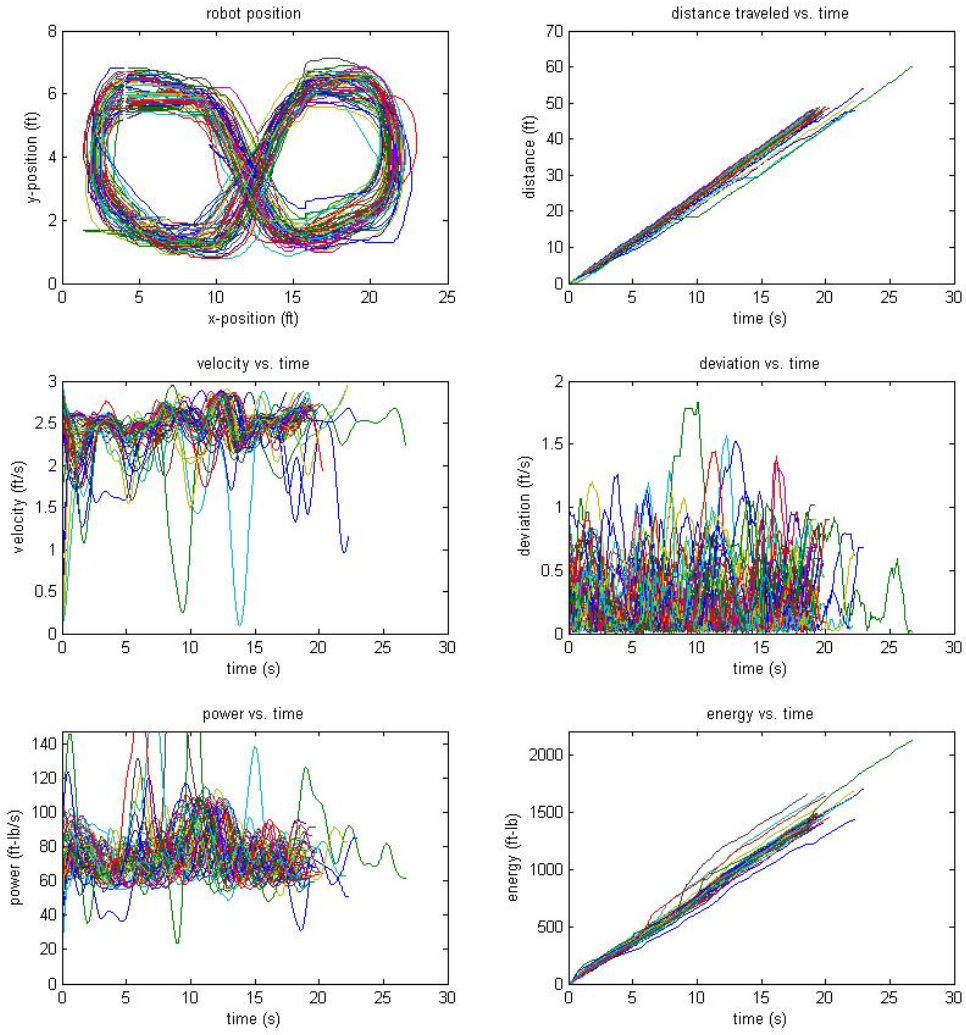

Figure 5-4: Lap plots, Talon 50 lap concrete test, reverse direction

5.4 Presentation of Robot Testing Results: 3D Plots

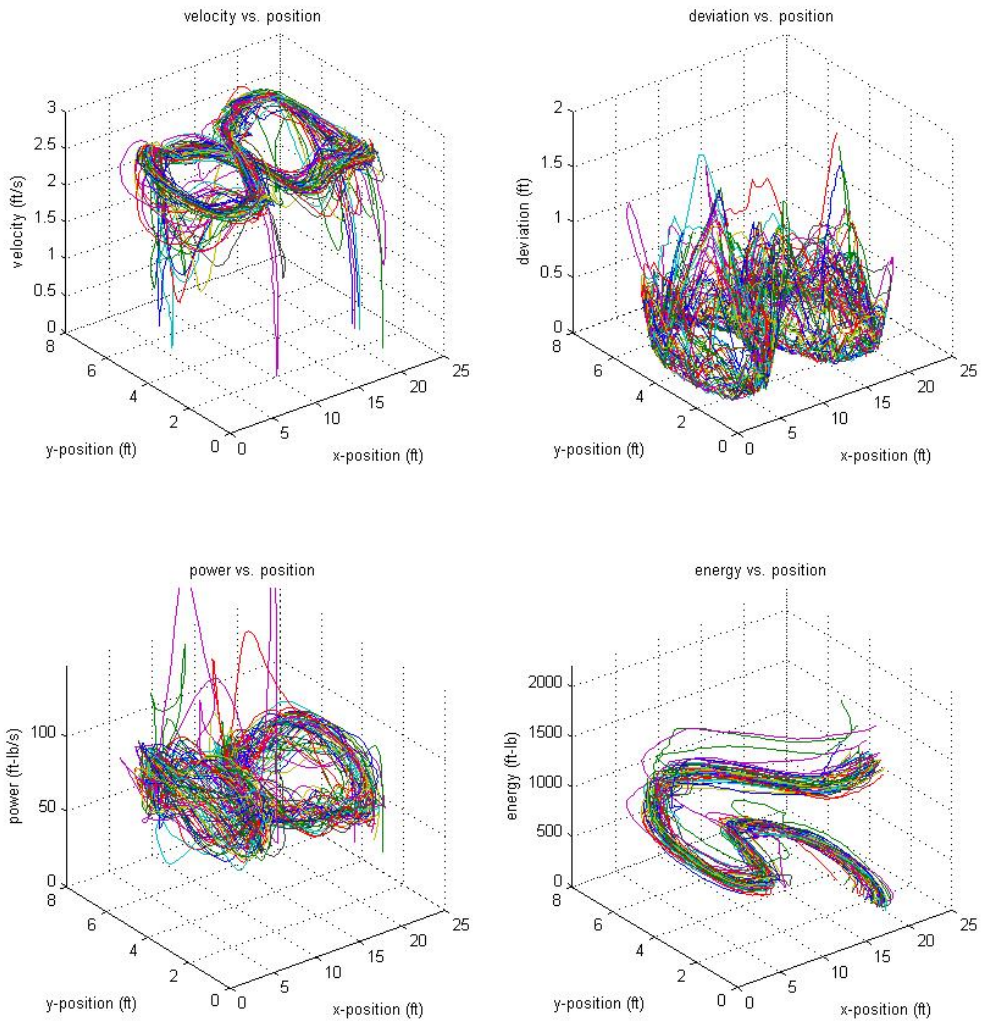

Figure 5-5: 3D lap plots, Talon 50 lap OSB test

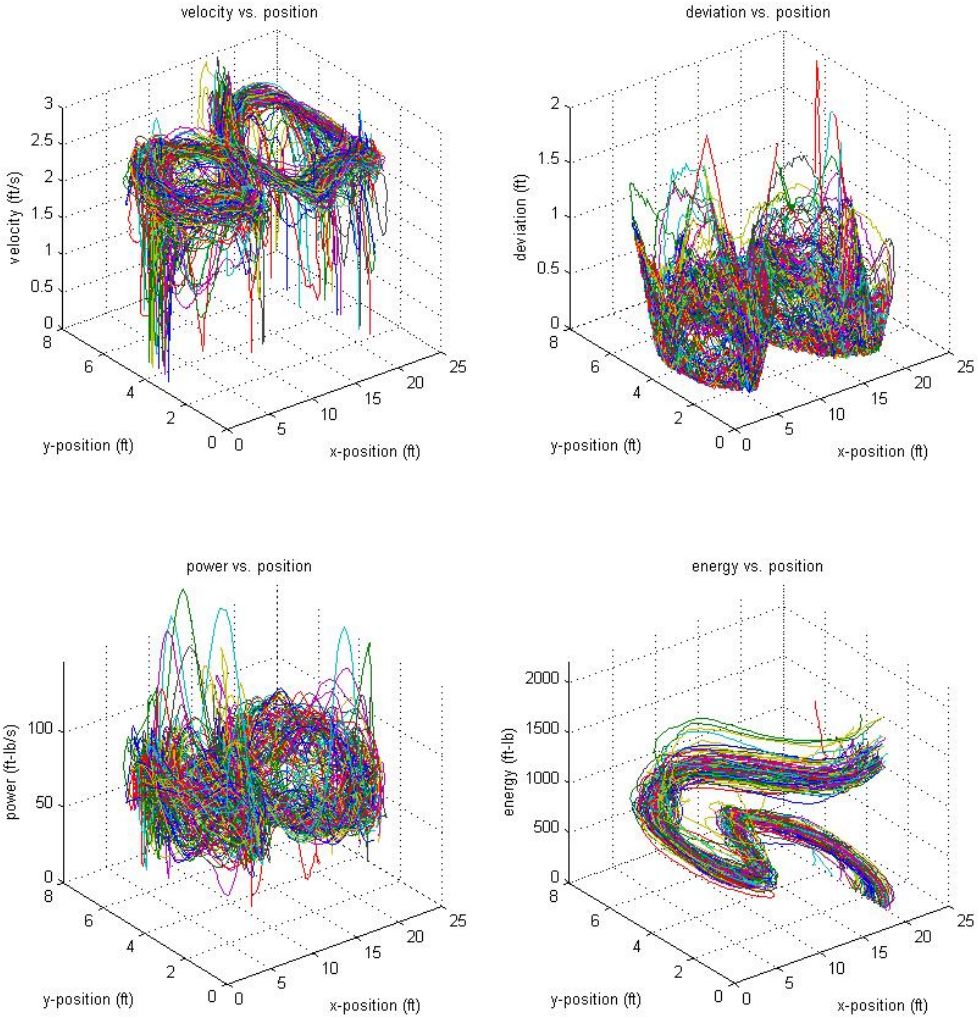

Figure 5-6: 3D lap plots, Talon 100 lap OSB test

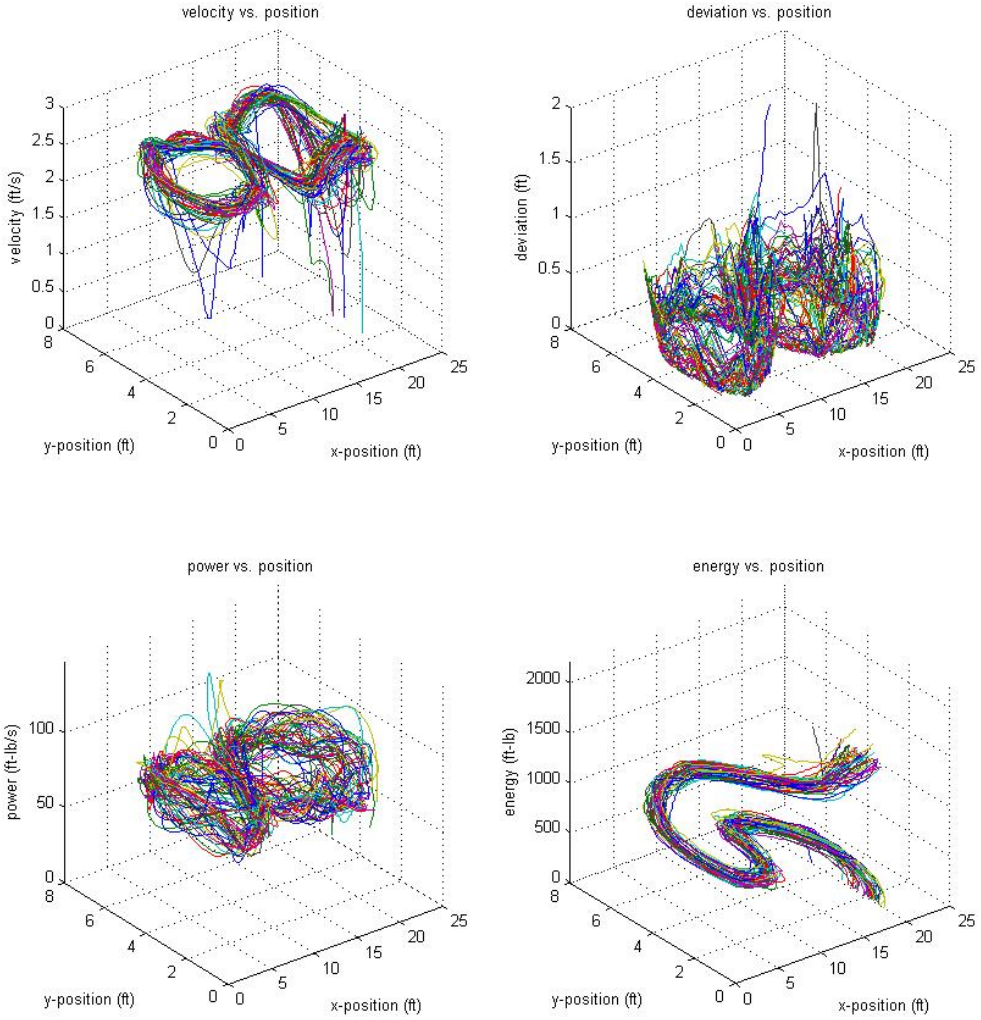

Figure 5-7: 3D lap plots, Talon 50 lap concrete test

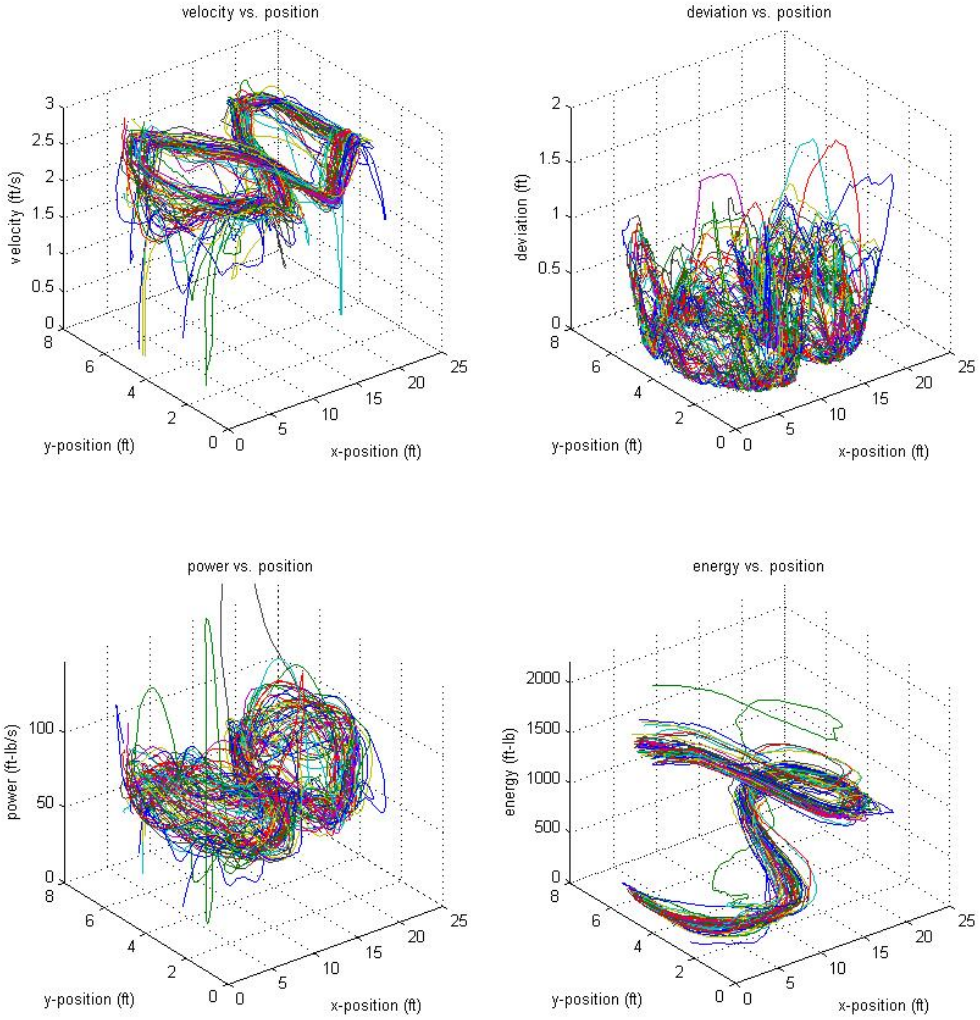

Figure 5-8: 3D lap plots, Talon 50 lap concrete test, reverse direction

5.5 Presentation of Robot Testing Results: Lap Trend Plots

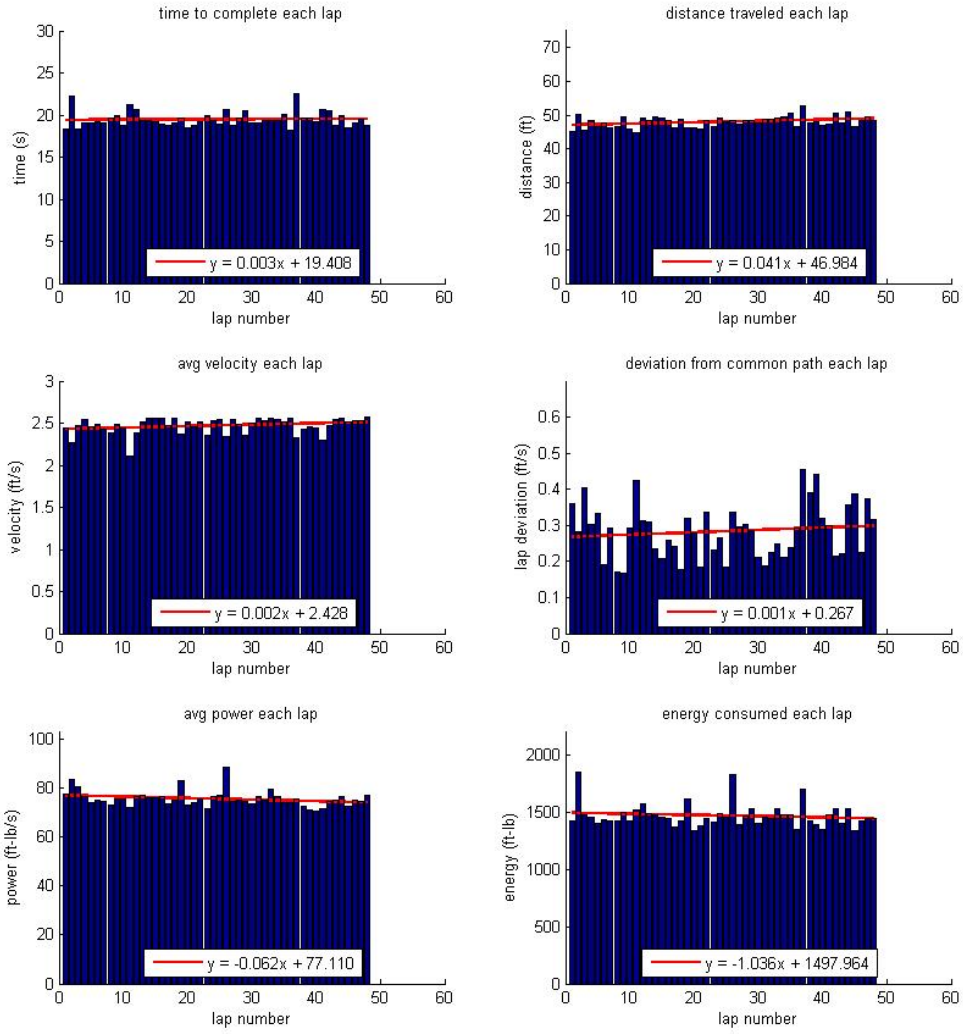

Figure 5-9: Trend plots, Talon 50 lap OSB test

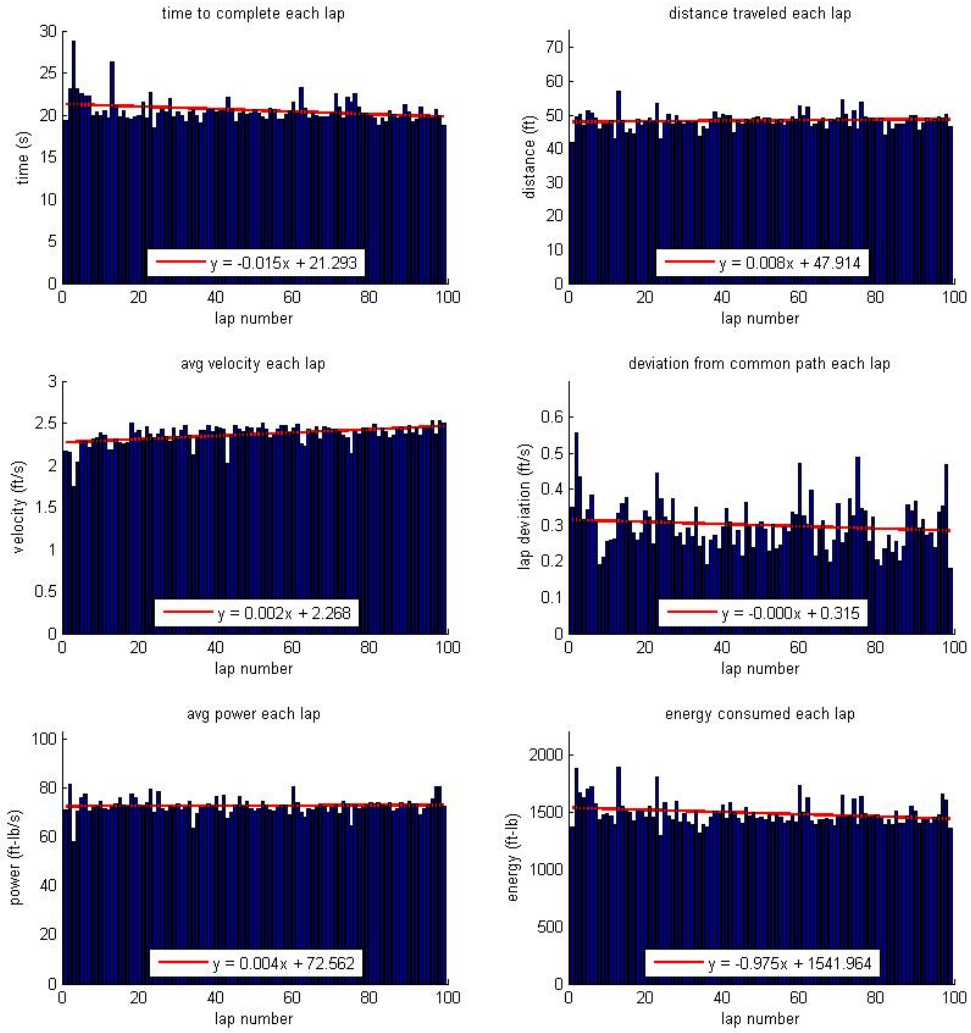

Figure 5-10: Trend plots, Talon 100 lap OSB test

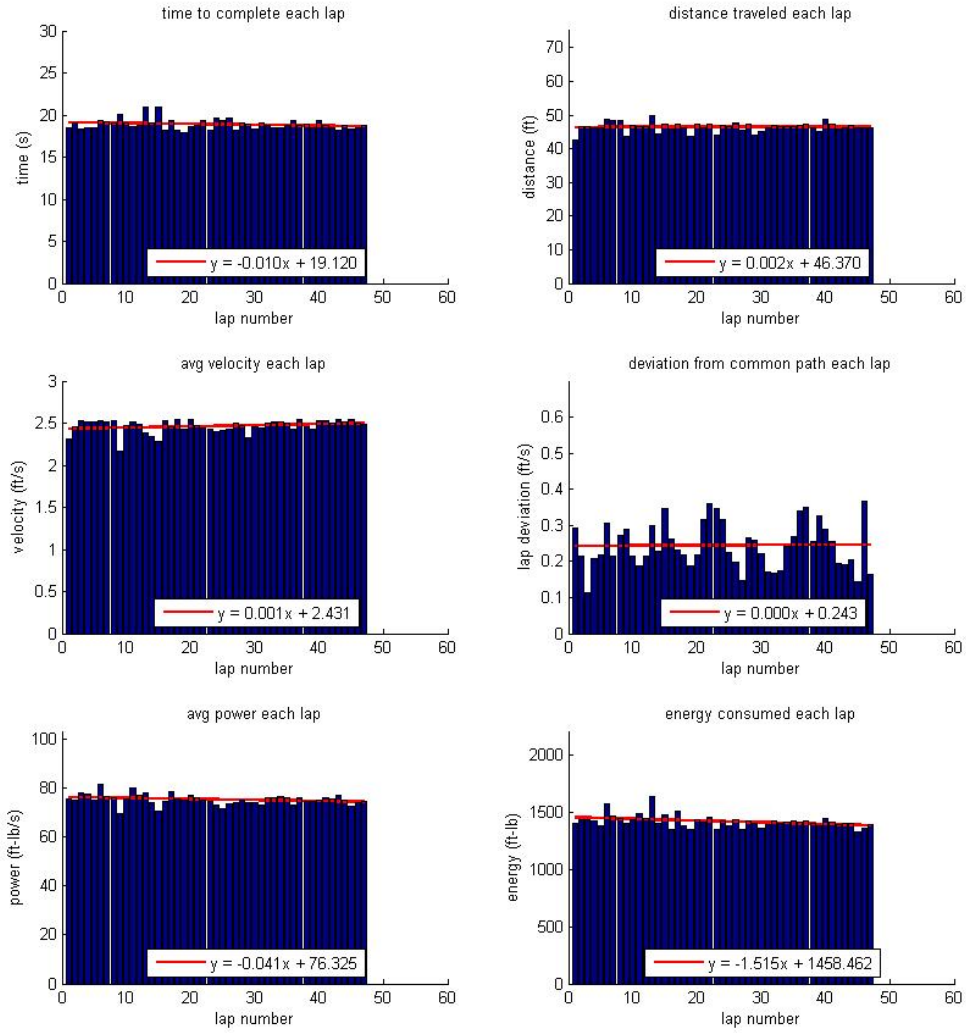

Figure 5-11: Trend plots, Talon 50 lap concrete test

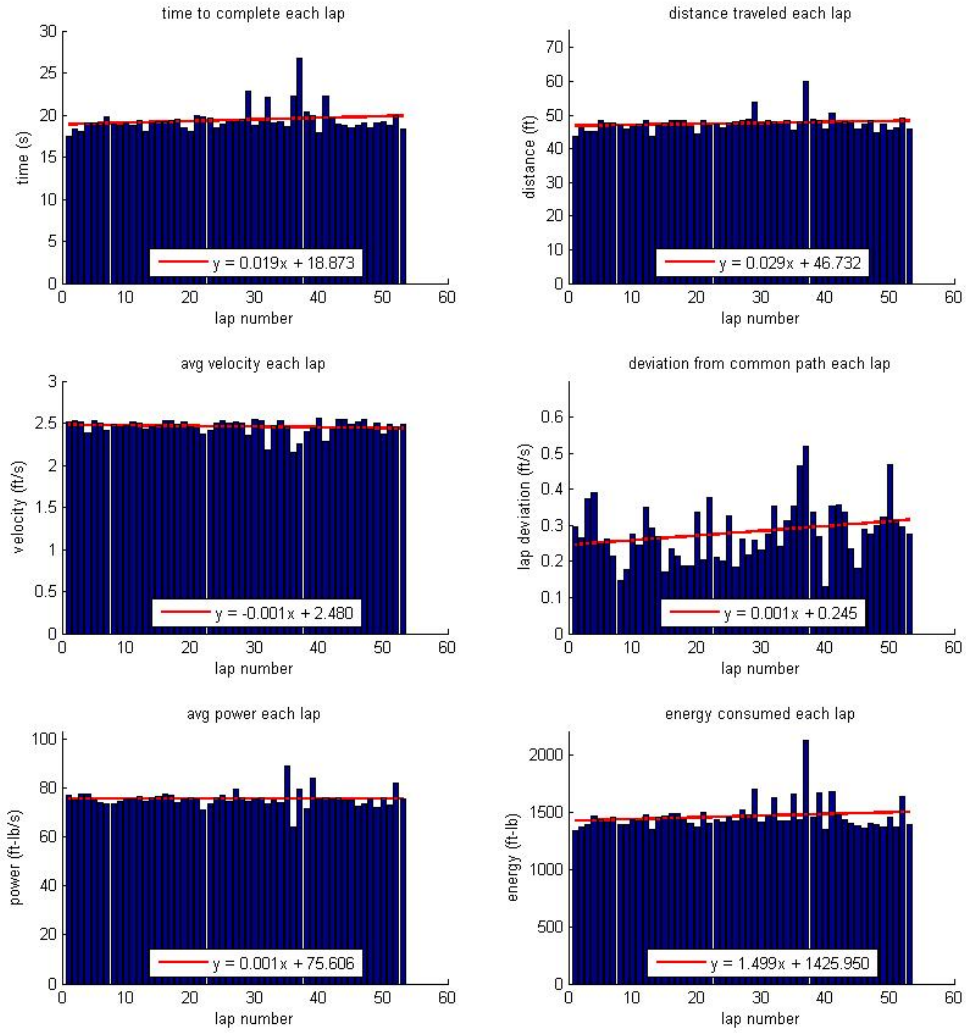

Figure 5-12: Trend plots, Talon 50 lap concrete test, reverse direction

5.6 Presentation of Robot Testing Results: Test Statistics

Lap trends are flat for the majority of the tests. The consistency in lap values is evidence of an experienced operator. With an operator learning how to use the system, one would expect trends showing faster lap times as the tests progressed. Also, after 50 laps, or approximately 30 minutes for an experienced operator, decrease in performance over time, due to fatigue, is generally not observed, due to the short duration of the test. Of most interesting note is the comparison between the 50 lap concrete tests in the forward and reverse direction. Because the reverse direction was not practiced beforehand, with all other tests being performed in the other direction, one might expect the results to display a difference. Indeed in the forward direction, the average time, power, and energy for each lap decreased over the course of the test while average velocity increased, indicating the driver was becoming more confident. The opposite case is true for the reverse direction test: lap time, power, and energy increased while average velocity decreased, indicating a less confident driver. This indicates fatigue since the robot operator was less familiar with the driving direction and was forced to pay closer attention to robot operation. In general, trend data may prove useful in future testing, but for tests performed for this thesis, trends over the course of a test are only slight and their statistical significance is debatable. Of more interest now are comparisons between net data for entire tests. These statistics were generated and presented in Table 5-2.

Table 5-2: Final Test Statistics

Row	Parameter	Talon OSB 50 laps	Talon OSB 100 laps	Talon concrete 50 laps	Talon concrete 50 laps backwards
1	Total laps completed	48	99	47	53
2	Total time (s)	934.81	2033.16	887.45	1027.94
3	Total distance (ft)	2303.70	4783.44	2181.36	2517.82
4	Total energy (ft-lb)	70684.39	147829.52	66839.22	77719.92
5	Lap time avg (s)	19.48	20.54	18.88	19.40
6	Lap time std (s)	0.92	1.43	0.63	1.48
7	Lap distance avg (ft)	47.99	48.32	46.41	47.51
8	Lap distance std (ft)	1.65	2.34	1.39	2.43
9	Lap velocity avg (ft/s)	2.47	2.36	2.46	2.45
10	Lap velocity std (ft/s)	0.29	0.38	0.27	0.26
11	Lap deviation avg (ft)	0.28	0.30	0.25	0.28
12	Lap deviation std (ft)	0.26	0.26	0.21	0.25
13	Lap power avg (ft-lb/s)	75.59	72.69	75.28	75.56
14	Lap power std (ft-lb/s)	15.31	15.77	10.33	15.40
15	Lap energy avg (ft-lb)	1472.59	1493.23	1422.11	1466.41
16	Lap energy std (ft-lb)	104.70	107.78	54.86	126.87

5.7 Analysis of Results

Studying the information in Table 5-2, conclusions can be reached about robot performance and how it changes under various circumstances. A comparison of the tests where the Talon operated for 50 laps and 100 laps on the OSB surface shows that the average lap time was 1 second longer during the 100 lap test. Figure 6-2 rules out the possibility of fatigue accounting for the increased lap time during the 100 lap test because the average time to complete each lap actually decreased over the duration of the test. Average lap distance and velocity are comparable between the two tests. Interestingly, for the 100 lap test, the average lap power decreased but the total lap energy increased. This would make sense if the robot was traveling faster and using less time but ultimately expending more energy. However, the average velocity is actually less for the 100 lap test. While puzzling, these differences are small (the percent decrease

in power is 3.8%) and therefore their statistical significance is questionable. Also note that the standard deviations of all lap data are greater for the 100 lap test than the 50 lap test, indicating a greater variation in lap performance over 100 laps versus 50 laps.

Comparing the 50 lap Talon OSB test to the 50 lap Talon concrete test, it can be observed that the tests are quite similar. Average lap time is 0.6 seconds greater for OSB, average lap distance is 1.58 ft greater for OSB, and average velocity and power consumption are comparable. Average lap energy is 21 ft-lb_f greater for the OSB test. The reason for the increase in average lap length for OSB is an interesting one. However, the increase in lap energy can be attributed to the increase in distance travelled. Standard deviations for all metrics are lower for the concrete test than the OSB test.

Next, the two tests can be analyzed for driving the Talon on concrete for 50 laps but in opposite directions around the figure-8 track. An initial hypothesis was that the direction which the operator was accustomed to driving around would yield more efficient driving when compared to the identical test in the less familiar reverse direction. This hypothesis is proven true by the results: the average time, distance, and energy for each lap were greater than their counterparts in the typical direction. Also note that the standard deviations of lap data for time, distance, power, and energy are all greater for the backwards test. Greater variance in the data indicates a less methodical approach to robot operation and a less experienced operator.

Figure 5-13, showing the most common path for both cases, confirms that a hysteresis effect is present in the robot path figure-8 when driven in the opposite direction, meaning that the robot's position at any point in time is affected by its past position. Note that in the top image the path is shifted up, while in the bottom image the path is shifted down. Several causes for this effect can be speculated. The robot operator may have a natural bias to drive the robot more aggressively through right turns versus left turns, for example. In addition, the robot could be

introducing a bias. For example, if the motors and gears are not delivering torque to the treads equally, this would cause the robot to turns differently when faced with a right or left turn.

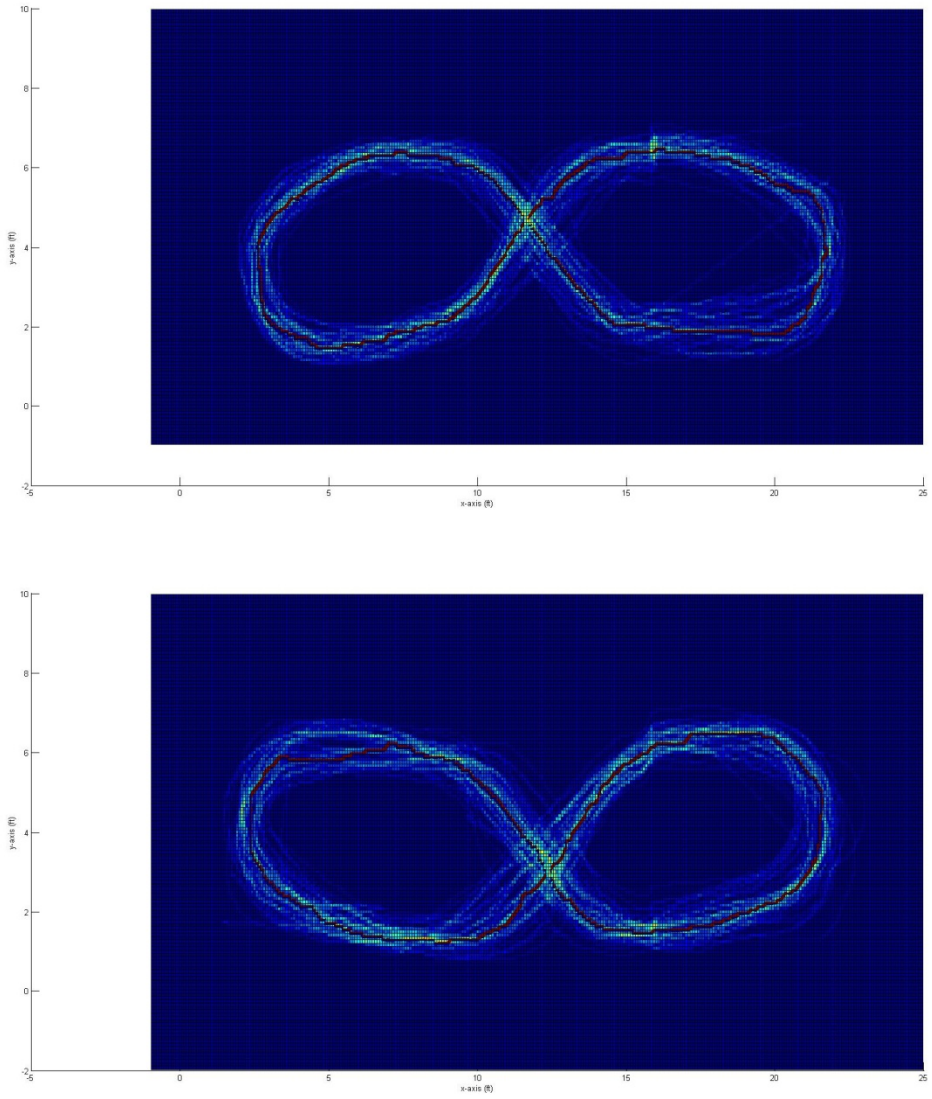

Figure 5-13: Talon 50 lap concrete tests hysteresis, typical direction (top) and opposite (bottom)

Using all 4 tests, average distance travelled per lap can be calculated as 47.56 feet. The NIST standard, which does not employ position tracking, assumes a lap of robot travel equates to approximately 15 meters. Converting to feet, NIST assumes a distance of 49.21 feet traveled per

lap, a value comparable (3.48% greater) than the average lap distance calculated in this thesis. This comparison can be used to both validate NIST's estimation of lap distance and demonstrate the viability of the distance tracking algorithm employed in this thesis.

Of final interest in the analysis of the data produced through robot testing is a calculation of the average energy used per unit of distance travelled over each test. These values calculated for each test can be seen in Table 5-3.

Table 5-3: Energy Usage per Unit Distance

Parameter	**Talon OSB 50 laps**	**Talon OSB 100 laps**	**Talon concrete 50 laps**	**Talon concrete 50 laps backwards**
Avg. energy/distance (lb_0	30.68	30.90	30.64	30.87

These values generated are highly significant, as energy as a function of distance is more useful than energy as a function of time, for purposes of calculating coefficients which can characterize the behavior of robot operation on a given terrain. Analysis of this data shows that the operation of the Talon robot on either concrete or OSB yields comparable energy consumption per unit distance, a significant finding.

Chapter 6 Conclusion

This thesis presents improvements made to a NIST mobile ground robot testing arena and preliminary testing to demonstrate system viability. Goals achieved include improvements made to a previously developed machine vision robot tracking system, development of a most common path processing algorithm, addition of robot power consumption information, and preliminary research into the Talon robot's performance under a variety of conditions. Future work is necessary to more fully explore areas of research and tests possible using the test method, including the effects of robot design, operator experience/fatigue, and terrain type on the performance metrics outlined in this thesis.

6.1 Accomplishments for NIST

The standard NIST testing protocol developed for robot mobility and endurance testing records lap times for a robot to complete a figure-8 over the course of a test. The improved testing system presented in this thesis also generates lap times, but generates a multitude of additional metrics and statistics with the potential for deep analysis. The original NIST protocol requires the manual recording of lap time, as well as the assumption that the robot operator is faithfully entering each end zone to complete a lap. Since development of the original protocol, NIST has developed a simple lap counter using a beam break sensor that counts every time the robot enters an end zone. However, this method may prove inaccurate as it does not guarantee the robot enters the end zone fully, but rather only breaks the beam with any portion of its body. The camera tracking method ensures the robot enters each end zone fully to complete each lap, as the fiducial, located on the center of the robot's body, must be the object which crosses into the end zone for the cameras to count a lap complete.

For testing NIST assumes the participation of an "expert" robot operator, or one who has reached a steady state of robot operation. The statistics of deviation generated in this work make

studies in robot operator variability possible. Operators deemed expert should display low standard deviations across all metrics of activity as compared to less experienced operators, as robot operation should be consistent at an expert level. Comparison across multiple expert operators could be used to determine whether a point of convergence is reached in consistent robot operation across all metrics, or whether each expert operator has his or her own idiosyncrasies in robot operation. In addition, operators deemed expert at operating one robot can be tested immediately on a different robot, to determine how much expert experience transfers between differing robot technologies. In addition, new robot operators can be tested and lap trends observed to analyze learning curve behavior, to determine if certain robots are more user-friendly than others. Also, operators deemed expert can be tested for long periods of time to observe how long it takes before performance is compromise below an acceptable level for any of the metrics developed.

Of all the metrics developed for this improved testing method, the addition of robot power consumption information is where the system truly displays significant potential. By calculating the total energy used per lap for a test with a given robot on a given terrain, coefficients of power consumption can be generated to assist robot operators in planning emergency response scenarios. These coefficients, combined with the knowledge of a robot's battery energy available and the terrain a specific scenario requires the robot to traverse, can help determine the operational range of the robot and predict mission success.

6.2 Recommendations for Future Work on the Testing System

While the robot testing system presents many benefits over the standard NIST testing protocol, one of which being automated lap counting and timing, there are many opportunities to improve the system in terms of accuracy, accessibility, and practicality.

A large challenge of this project was the effective stitching together of multiple camera images covering the entirety of the testing arena. In theory, if the testing arena was located in a room with a high ceiling only one camera could be necessary to cover the entirety of the arena. However, this camera would have to be of a high resolution, and the height clearance required for this idea makes this approach impractical. As a result, currently three cameras are used and calibrated using markers located on the walls of the testing arena. This method proved effective in stitching together images; however, transitions of the fiducial between images still proves noticeable. The fiducial tracking code is designed to recognize only one fiducial at a time; modifying the machine vision algorithm to recognize the fiducial in multiple camera views and transition more smoothly between them could improve results.

Another issue encountered in machine vision processing was the consistent detection of the fiducial. An LED fiducial was substituted for the original green disk fiducial because it is less subject to changes in lighting conditions. A downside of the LED fiducial was the possibility of lens flare in the camera images caused by the bright light. Placing a piece of paper over the LED fiducial negated lens flare, but a more well designed solution such as an LED in a table tennis ball might be an excellent solution for a fiducial. For testing presented in this work, consistent testing and processing was achieved in large part through dark room testing, which allowed the LED fiducial to show up extremely clearly.

One difficulty in finding dependable fiducial detection methodologies is the processing time of the fiducial identification and tracking algorithms. Performing image transformations in MATLAB for every camera image at each iteration is the slowest method. Generating a lookup table based on an image transformation or sequence of transformations speeds processing, but the lookup table itself takes considerable time to generate. Furthermore, cameras cannot be moved or the camera calibration modified without regenerating the lookup table. To speed processing, the best approach may be to write the processing code in another language entirely.

A problem encountered frequently in testing was the consistent operation of the onboard data logger. The data logger itself draws a small amount of power from the robot to run. During testing, particularly on the ramped surfaces, robots would often be shaken violently and momentarily lose power. This caused the data logger to crash. Additionally, large current draws on the logger during particularly intense robot maneuvers could cause the logger to drop below the required operational voltage, also causing a reset. Originally, the software of the data logger stored the data in such a way that in the event of a power loss, the current storage file would become corrupted and all data would be lost. Later, a data logger with updated software was used in testing. In the event of a momentary power loss this logger did not corrupt an entire data set, but rather safely saved all data which had been recorded up to that point. This allowed some data to be saved that would otherwise have been lost. However, the data logger still did not record further data after the power loss. To combat this problem the power logger and connecting power cords were firmly secured with tape.

Possible future work could include the construction of an entirely new power logger. Ultimately, a set of non-wireless data loggers were already on hand for this project, and it was decided to use these because of convenience, proven technology, and a high data collection rate. However, a data logger with wireless communication ability could conceivably communicate with the camera system and greatly reduce efforts to synchronize data in processing. This would require the addition of a wireless router and software to synchronize data collection with respect to time. The selection of a new data logger would be required as well as addressing issues of data collection rates and latency.

The last main challenge of developing the testing system was the incorporation and synchronization of power data from the robot with the already developed camera-based position tracking system. In this work the power data gathered by a data logger onboard the robot was not synchronized through wireless communication with the camera image acquisition system but

rather through synchronized in processing. Considerable time was spent on developing a robust algorithm for this purpose, however, developing a wireless power logger may be the best long-term solution. A wireless power logger with the ability to directly synchronize timestamps with the camera collection system would be even more robust in terms of successful synchronization, generate higher accuracy synchronization, and require less user involvement in processing.

6.3 Recommendations for Future Work in Robot Testing

The small sample of tests performed for this work mainly serve to demonstrate the viability of the testing system. The potential for future testing is vast.

The Talon robot was the primary robot used to demonstrate testing in this work. Problems associated with modifying the BomBot led to unreliable testing. The hobby camera attached to the BomBot was powered off a 9V battery, which was necessary to change approximately every 20 minutes. Future testing should make use of a longer lasting camera. In addition, the BomBot battery modification proved incompatible with the logger to collect power data. During heavy use the BomBot's BB-390 battery was not able to supply the minimum required voltage for the attached data logger to run (approximately 8V), which would cause it to reset. Also, different robots have different power profiles, which could interfere with the synchronization algorithm developed previously with the Talon. This problem was encountered when attempting to process BomBot data. This issue favors the previously suggested idea to employ a wireless data logger for direct synchronization as opposed to lap grouping in order for the system to be truly robust for any robot.

Unfortunately, the ramps designed for the testing arena proved too damaging to the Talon robot for extensive use. The remaining surfaces of flat concrete and OSB proved adequate for demonstrating the test method, however, in the future more interesting and varied terrains should be tested to study their effects on robot performance. Such surfaces could include sand, mulch,

gravel, or Astroturf. The goal of such terrain studies would be the generation of coefficients to better assist in the prediction of mobile ground robot performance and endurance during operation. While this testing proved concrete and OSB comparable in terms of energy consumed per unit distance, other terrains may yields different coefficients.

In addition to the tests already described, path following tests could be conducted where tape or paint is used to mark a path on the floor of the testing arena. Similar to the most common path deviation algorithm already developed, this path could be constructed in computer code and the deviation of robot position from the designated path could be calculated over the course of a test. Another series of tests could be conducted in performance versus payload studies. In these tests, a robot would be run through a series of otherwise identical tests changing amounts of weight. Performance could then be evaluated as a function of payload weight.

6.4 Closing Remarks

The goal of this thesis – to demonstrate viable improvements possible to a NIST ground robot mobility testing procedure and conduct preliminary research on the Talon robot – has been accomplished. While the project leaves much room for future development and testing, the work shows the usefulness of capturing multiple metrics of robot performance in a controlled environment and the potential for future use in the validation of mobile ground robots.

Appendix A MATLAB Code

The Python code used to collect camera images in Ubuntu and general instructions to collect and process images can be found in Pangborn's thesis [3]. Collected camera images were loaded into a folder named images_PY with subfolders cam_1, cam_2, and cam_3. Data logger CSV files were loaded into the folder power_logger. Saved information such as DataLog.mat, TrialLog.mat, Laplog.mat, ResultsStats.mat, and any plots are saved to the current MATLAB directory.

A.1 **Script1_CollectTestImages.m**

```
This script collects in real-time and saves to file images that can be post-processed
% to conduct lap counting and distance tracking

% The basenames of files generated are: im_[camera #]_[iteration #]
% ex. im_1_1, im_2_1, etc.

clear
clc

% Initialize parameters for the cameras
FlagLive = 1;
[ IP,CamRes,CamParam ] = FcnInitCamParams(FlagLive);

% Define a name for the folder in which to store images
% NOTE: The lap and distance tracking scripts looks in the folder "images" by default
% imagefolder = 'images'; % DEFAULT
imagefolder = 'images_MATLAB';

mkdir(imagefolder);
mkdir('images_MATLAB','cam_1')
mkdir('images_MATLAB','cam_2')
mkdir('images_MATLAB','cam_3')

choice = questdlg('START?', ...
    'START', ...
    'START','START');

% Handle response
switch choice
    case 'START'

        % Create a timer for iteration times
        ElapsedTimer = tic;

        % Set up the loop to run forever
        ImNum = 0;
        while ImNum > -1

            % Get the elapsed time
            ElapsedTime = toc(ElapsedTimer);

            % Generate a filename for the image
            filename=strcat(sprintf('%5.7f',ElapsedTime),'.jpg' );
```

```
            for CamNum=1:length(IP)


                % Get IP address for the camera
                name = IP{CamNum};

                % Load image from the camera
                im = imread(name);

                % Save the file as the fiename
                imwrite(im,filename,'jpg');

movefile(filename,strcat('./',imagefolder,'/','cam_',num2str(CamNum),'/',filename))

            end

            % Update the iteration counter
            ImNum = ImNum+1;

            % Print the time in the command window
            disp(num2str(ElapsedTime))
        end
end
```

A.2 **Script2_Calibrate.m**

```
% This script allows users to generate the calibration data files needed for lap and
distance tracking

clear
clc

% Initialize variables for the code that users may want to modify
[ FlagLive,FlagPlot,FlagSavePlot,NumCams,Data2File ] = FcnInitTestConditions;

% Initialize parameters for the cameras
[ IP,CamRes,CamParam ] = FcnInitCamParams(FlagLive);

% Initialize variables for the code
[
Iter,CentroidFT_Last,CentroidPX_Last,CentroidPX_Last_Raw,TotalLaps,TotalDist,LastZone,Fla
gObjFound,DataLog,TimeStamps ] = FcnInitVars( Data2File,FlagLive );

%% Initialization Procedure

% Conduct and save camera distortion calibrations
FcnInitDistortCorrection(CamParam,NumCams,CamRes);

% Conduct and save distance tracking calibrations
FcnInitDistTrack(IP,FlagLive,TimeStamps,Iter,NumCams,CamRes);

% Conduct and save endzone location calibrations
FcnInitEndzones(IP,CamRes,FlagLive,TimeStamps,Iter,NumCams);

% Conduct and save black bar calibrations
FcnInitBlackBars(IP,CamRes,FlagLive,TimeStamps,Iter,NumCams);
```

A.3 **Script3_DataLog.m**

```
% This script conducts lap counting and distance tracking for NIST robot testing methods
on a fiducial using IP cameras.
% A lap is considered to be one full trip about the course, from the starting endzone to
the other and then back.
```

```
% Note that lap counting only begins when the fiducial enters an endzone for the first
time.
% The Matlab Image Processing Toolbox is required to run this code.

% Developed by Herschel Pangborn, Penn State University, 2012, using MATLAB R2011a for
Mac OSX.
% Please direct any quesitons to theherschmeister@gmail.com
% Some algorithms are modified from those written by Professor Sean Brennan and Kevin
Swanson, Penn State University,
% and from the Matlab Camera Calibration Toolbox

clear
clc

%% Initialize Parameters and Variables

% Initialize variables for the code that users may want to modify
[ FlagLive,FlagPlot,FlagSavePlot,NumCams,Data2File ] = FcnInitTestConditions;

% Initialize parameters for the cameras
[ IP,CamRes,CamParam ] = FcnInitCamParams(FlagLive);

% Initialize variables for the code that users don't need to change
[
Iter,CentroidFT_Last,CentroidPX_Last,CentroidPX_Last_Raw,TotalLaps,TotalDist,LastZone,Fla
gObjFound,DataLog,TimeStamps] = FcnInitVars( Data2File,FlagLive );

%% Load Data Files From Calibration Scripts

[ CalibDistTrack,CalibEndzones,newlocation,DistortionMapping,CalibBlackBars] =
FcnGetCalibrations;
clc

%% Obtain and Plot the Starting Position and Begin the Loop on Command

% Get the fiducial position in both pixels and real-world coordinates
[ CentroidFT_Last,CentroidPX_Last,CentroidPX_Last_Raw,FlagObjFound ] = FcnGetPosition(
IP,CamRes,FlagLive,TimeStamps,FlagObjFound,Iter,NumCams,CalibDistTrack,CalibEndzones,Cent
roidFT_Last,CentroidPX_Last,CentroidPX_Last_Raw,FlagPlot,newlocation,DistortionMapping,Ca
libBlackBars);
% Set the iteration counter = 1
Iter = Iter+1;

%% Loop Time!

% Use a dialogue to start
choice = questdlg('START?', ...
    'START', ...
    'START','START');
% Handle response
switch choice
    case 'START'
        %close(1)

        % Start a timer for finding lap times
        ElapsedTimer = tic;

        % Loop indefinitely if running in real-time, or until end of data if loading
images from file.
        Itstop = 1;
        while Itstop

            % Start a timer for fps timing
            FpsTimer = tic;

            % Get the fiducial position in both pixels and real-world coordinates
            [ CentroidFT_Current,CentroidPX_Current,CentroidPX_Current_Raw,FlagObjFound ]
= FcnGetPosition(
IP,CamRes,FlagLive,TimeStamps,FlagObjFound,Iter,NumCams,CalibDistTrack,CalibEndzones,Cent
```

```
roidFT_Last,CentroidPX_Last,CentroidPX_Last_Raw,FlagPlot,newlocation,DistortionMapping,Ca
libBlackBars );
            % Get the total time of the test thus far
            if FlagLive == 1
                TotalTime = toc(ElapsedTimer);
            else
                TotalTime = TimeStamps(Iter);
            end

            % Conduct lap counting
            [ TotalLaps,LastZone ] = FcnCalcLaps(
CalibEndzones,TotalLaps,LastZone,CentroidPX_Current );

            % Conduct distance tracking
            [ TotalDist ] = FcnCalcDist( TotalDist,CentroidFT_Last,CentroidFT_Current );

            % Calculate the fps
            Frame = toc(FpsTimer);
            FPS = 1/Frame;

            % Print some data to the screen if the object was found this iteration
            if FlagObjFound == 1
                fprintf(1,'Iter: %5d Total Laps: %4.1f, Total Distance (ft): %10.2f, X
Location (ft): %6.2f, Y Location (ft): %6.2f, X Location (px): %6.2f, Y Location (px):
%6.2f, FPS: %6.2f\n',...
                    Iter,TotalLaps, TotalDist, CentroidFT_Current(1),
CentroidFT_Current(2), CentroidPX_Current(1), CentroidPX_Current(2), FPS)
            else
                fprintf('OBJECT NOT FOUND\n')
            end

            % Update data log
            [DataLog] = FcnLogData( Iter, FlagLive ,TotalLaps, TotalDist, TotalTime,
CentroidPX_Current, CentroidFT_Current, DataLog, Data2File,length(TimeStamps) );

            % Save plot window to file
            if FlagPlot == 1 && FlagSavePlot == 1
                h = figure(1);
                title(strcat('Iter: ',num2str(Iter),', TotalLaps: ',num2str(TotalLaps),'
TotalDist: ',num2str(TotalDist)))
                print(h,strcat('Iter_',num2str(Iter)),'-djpeg')
            end

            % If loading images from file, stop the loop
            if FlagLive == 0
                if Iter == length(TimeStamps)
                    Itstop = 0;
                end
            end

            % Update centroid locations
            CentroidFT_Last = CentroidFT_Current;
            CentroidPX_Last = CentroidPX_Current;
            CentroidPX_Last_Raw = CentroidPX_Current_Raw;
            Iter = Iter+1;

        end
end
```

A.4 **Script4_TrialLog.m**

```
% This script takes DataLog.mat and adds velocity, pathdev, and power
% information, to create TrialLog.mat

clear
clc
```

```
close all

% Load camera data (from original DataLog.mat file in main folder)
DataLog = FcnDataLogZeros;
% Load power logger data (from CSV files in appropriate folder)
PowerLog = FcnPowerLog;

% Create velocity data from position data
[Velocity,Velocity_Filt] = FcnVelocity(DataLog);
DataLog(:,9) = Velocity;
DataLog(:,10) = Velocity_Filt;

% % Create histogram and path deviation data from position data
[C,Ridge,PathDev,DataLogInterp] = FcnPathDev(DataLog);
DataLog(:,11) = PathDev;

% Creates TrialLog - This log contains all data relevant from a test in a
% 1x2 cell array. (For camera and power collection rates). This log will
% now be used in all post-processing. (Eg. Synchronization, Results,
% Histogram, etc.)

TrialLog{1,1} = {'1:Iteration, 2:X-position(pix), 3:Y-position(pix), 4:X-position(ft),
5:Y-position(ft), 6:Time(s), 7:Laps, 8:Total Distance (ft), 9:Velocity(ft/s),
10:Velocity_Filt(ft/s), 11:Common Path Deviation (ft)'};
TrialLog{1,2} = {'1:Voltage(V), 2:Current(A), 3:Power(J), 4:Iteration'};

TrialLog{2,1} = DataLog;
TrialLog{2,2} = PowerLog;

save('TrialLog.mat','TrialLog','C','Ridge','PathDev','DataLogInterp')
```

A.5 **Script5_LapLog.m**

```
% This script pulls data from TrialLog.mat to sync camera and power data
% in time, based on velocity and power standard deviations over test.

% This script handles camera data produced from TrialLog.mat, using
% scripts originally for power processing, filters velocity data.
% Breaks into 4 groups, based on std.
clear
clc
close all
load TrialLog.mat

%% Filtering velocity (ft/s)

Velocity = TrialLog{2,1}(:,9);
Time = TrialLog{2,1}(:,6);

% plot raw velocity
figure
plot(Time,Velocity,'b')
xlabel('Time (s)')
ylabel('Velocity (ft/s)')
title('velocity data')


% filter data by standard deviation
frames = 5; % number of frames to use for calculating std
NumBlocs = floor(length(Velocity)/frames); % rounding off
BlocArray = [0:NumBlocs]*frames; % array for appropriate frame segments

% Calculate std for frames
for i = 1:NumBlocs
    StdVel(i) = std(Velocity(BlocArray(i)+1:BlocArray(i+1)));
    % Treat std as occuring at time value of last frame of set
```

```
    StdTime(i) = Time(i*frames);
end

% Apply binary threshold to std
for i = 1:length(StdVel)
    if StdVel(i) < .3
        OnOff(i) = 1;
    else
        OnOff(i) = 0;
    end
end

% plotting on raw velocity
hold on
plot(StdTime,StdVel,'g')
plot(StdTime,OnOff,'r')
legend('raw velocity','std','onoff')

%% Some initializations (User Interface might be helpful)
% Set startpoint (in a period of rest before official start)
Startpoint = 94; % seconds
Endpoint = 1767; % seconds

[A,B] = min(abs(StdTime-Startpoint)); % B is the entry in time
Startpointk = B; % starting point in terms of time index
[A,B] = min(abs(StdTime-Endpoint)); % B is the entry in time
Endpointk = B; % ending point in terms of time index

% Finding exact start of first lap
exit = 0; %initialize for while loop
k = Startpointk; % initialize counter
while exit < 1
    if OnOff(k-1)==1 & OnOff(k)==0 % right shoulder search
        alpha = k; % exact startpoint at first lap
        exit = 1; % exits loop
    end
    k = k + 1; % iterates counter
end

%% Group searching and sorting

group = 0; % initialize group counter
leftshoulder = alpha; % initialize left shoulder marker
rightshoulder = alpha; % initialize right shoulder marker
for k = Startpointk:Endpointk
    if OnOff(k-1)==0 & OnOff(k)==1 % left shoulder search
        leftshoulder = k;
    end
    if OnOff(k-1)==1 & OnOff(k)==0 % right shoulder search
        rightshoulder = k;
    end

    if (rightshoulder - leftshoulder) > 10*(15)/frames % ~10 second pause test
        group = group + 1;
        % places each grouping into a cell
        SyncLog_Cam{1,group} = TrialLog{2,1}(alpha*frames-(frames-1):leftshoulder*frames-
(frames-1),:);

        if (leftshoulder - alpha) < 100*(15)/frames % deletes grouping if less than ~100
sec. (movement blip in pause)
            SyncLog_Cam(:,group) = [];
            group = group - 1;
        end
        alpha = rightshoulder; % moves up alpha
        leftshoulder = rightshoulder; % moves up leftshoulder
    end
    if k == Endpointk % closes last lap
        rightshoulder = Endpointk;
        group = group + 1;
```

```
        SyncLog_Cam{1,group} = TrialLog{2,1}(alpha*frames-(frames-1):leftshoulder*frames-
(frames-1),:);
    end
end


%% This is the power section of the code, it does a similar thing as
% velocity, but with a different sampling rate algorithm. (The power data
% is known to have a 1000 Hz sampling rate, std was calculating every
% quarter second.) For velocity, std was calculated over a (setable)
% number of frames, leading to whatever data collection rate that is.

% Generate power & plot

Voltage = TrialLog{2,2}(:,1);
Current = TrialLog{2,2}(:,2);
Power = TrialLog{2,2}(:,3);
Iteration = TrialLog{2,2}(:,4);

% Initialize the samping rate (Hz)
SamplingRate = 1000;

% Initialize a time vector
time = (1:length(Power))/SamplingRate;

% Plot the raw power data
figure
plot(time,Power,'b')
xlabel('Time (s)')
ylabel('Power (J/s)')
title('power data')

% % Low pass butterworth filter (negligible difference)
% [b,a] = butter(3,0.2,'low');
% Powerfilt_low = filtfilt(b,a,Power);
%% Filter power data by standard deviation

% Calculate std for quarter second segments
NumSeg = floor(length(Power)/250);
Array250 = [0:NumSeg]*250;
StdArray = [];
for i = 1:NumSeg
    StdArray(i) = std(Power(Array250(i)+1:Array250(i+1)));
end

% Apply threshold to std to distinguish when robot is idle
% (When robot is idle, power consumption has low std.)
OnOff = [];
for j = 1:length(StdArray)
    if StdArray(j) < 8
        OnOff(j) = 1;
    else
        OnOff(j) = 0;
    end
end

% Plotting above for debugging purposes (helps find start point)
hold on
timeStd = [1:length(StdArray)].*.25 - .25/2;
plot(timeStd,StdArray,'g')
% Plotting OnOff
plot(timeStd,OnOff,'r')
legend('raw power','std','onoff')

%% Some initializations - (User Interface?)

% Set startpoint (shortly before robot starts first lap)
Startpoint = 220; % seconds
Startpointk = Startpoint*4 + 1; % in terms of counter k
```

```
% Set endpoint (shortly after robot finishes last lap)
Endpoint = 1970; % seconds
Endpoint = Endpoint + 4; % 4 second buffer for safety
Endpointk = Endpoint*4 + 1; % in terms of counter k

% Finding exact start of first lap
exit = 0; %initialize for while loop
k = Startpointk; % initialize counter
while exit < 1
    if OnOff(k-1)==1 & OnOff(k)==0 % right shoulder search
        alpha = k; % exact startpoint at first lap
        exit = 1; % exits loop
    end
    k = k + 1; % iterates counter
end

%% Group searching and sorting

group = 0; % initialize lap counter
leftshoulder = alpha; % initialize left shoulder marker
rightshoulder = alpha; % initialize right shoulder marker
for k = Startpointk:Endpointk
    if OnOff(k-1)==0 & OnOff(k)==1 % left shoulder search
        leftshoulder = k;
    end
    if OnOff(k-1)==1 & OnOff(k)==0 % right shoulder search
        rightshoulder = k;
    end
    if k == Endpointk % closes last lap
        rightshoulder = Endpointk;
    end
    if (rightshoulder - leftshoulder) > 12 % 3 second pause test
        group = group + 1;
        % places each lap current, voltage, power data in cells
        SyncLog_Pow{1,group} = TrialLog{2,2}((alpha-1)/4*SamplingRate:(leftshoulder-
1)/4*SamplingRate,:);
        if (leftshoulder - alpha) < 20 % deletes lap if not long enough (movement blip in
pause)
            SyncLog_Pow(1,group) = [];
            group = group - 1;
        end
        alpha = rightshoulder; % moves up alpha
        leftshoulder = rightshoulder; % moves up leftshoulder
    end


end

% Adding time to groups
% Columns: 1.) Voltage 2.) Current 3.) Power 4.) Iteration 5.) Time

for group = 1:length(SyncLog_Pow)
    SyncLog_Pow{1,group}(:,5) = [0:length(SyncLog_Pow{1,group})-1]'/SamplingRate; % time
domain
    % SyncLog_Pow{1,group}(:,6) = cumtrapz(SyncLog_Pow{1,group}(:,3))/SamplingRate; %
energy drain over lap (can plot)
end

% Check to have matching number of groups
if length(SyncLog_Cam) ~= length(SyncLog_Pow)
    disp('ERROR: Camera and power logger data not matching properly. Debug.')
    disp('Tips: Adjust both camera and power logger processing startpoints, adjust both
std thresholds.')
    break
end
% save('SyncLog.mat','SyncLog_Cam','SyncLog_Pow')

%% This part takes sync files (camera data and power data divided into
```

```
% groups of laps), and separates them into individual laps, based on
% endzone crossing. All data saved to LapLog.mat

% clear
% clc
% close all
% load('SyncLog.mat')

for group = 1:length(SyncLog_Cam)
    SyncLog_Cam{group}(:,6) = SyncLog_Cam{group}(:,6) - min(SyncLog_Cam{group}(:,6));
end

% Creating GroupLog
[entries,datatypes_cam] = size(SyncLog_Cam{1});
[entries,datatypes_pow] = size(SyncLog_Pow{1});
for group = 1:length(SyncLog_Cam)
    for class = 1:datatypes_cam
        GroupLog{class,group} = SyncLog_Cam{group}(:,class);
    end
    for class = 1:datatypes_pow
        GroupLog{datatypes_cam + class,group} = SyncLog_Pow{group}(:,class);
    end
    % rounding camera time to .001 place
    num_dig = 3;
    GroupLog{6,group} = round(GroupLog{6,group}*(10^num_dig))/(10^num_dig);
end

%% This part corrects power drift in grouplogs
[classes,groups] = size(GroupLog);

% figure
% hold on
% for i = 1:groups
% plot(GroupLog{15,i},GroupLog{14,i},'b')
% end
% xlabel('Iteration (s)')
% ylabel('Power (J/s)')
% title('power data')

for i = 1:groups
    IterStart = GroupLog{15,i}(1);
    IterEnd = GroupLog{15,i}(end);
    Iter1Range = [IterStart-5000,IterStart-3000]; %range to take average power before
group
    Iter2Range = [IterEnd+3000,IterEnd+5000]; %range to take average power after group
    Index1Start = find(Iteration == Iter1Range(1));
    Index1End = find(Iteration == Iter1Range(2));
    Index2Start = find(Iteration == Iter2Range(1));
    Index2End = find(Iteration == Iter2Range(2));
    Power1Mean = mean(Power(Index1Start:Index1End));
    Power2Mean = mean(Power(Index2Start:Index2End));
    NumbEntries = length(GroupLog{15,i});
    GroupLog{14,i} = GroupLog{14,i} - linspace(Power1Mean, Power2Mean, NumbEntries)';

end

% % Plot the raw (corrected) power data
% for i = 1:groups
% plot(GroupLog{15,i},GroupLog{14,i},'g')
% plot(GroupLog{15,i},zeros(1,length(GroupLog{15,i})),'r')
% end
% hold off


%% Dividing GroupLog into individual laps (cam time and pow time tricky)
lap = 1;
for group = 1:length(SyncLog_Cam)
    iprev = 1; % initialize low shoulder (cam)
    jprev = 1; % initialize low shoulder (pow)
```

```
    for i = 2:length(GroupLog{1,group})
        if GroupLog{7,group}(i-1) ~= GroupLog{7,group}(i) && rem(GroupLog{7,group}(i),1)
== 0
            % transforming camera time to power time
            timehigh = GroupLog{6,group}(i-1);
            j = min(find(GroupLog{datatypes_cam + 5,group} >= timehigh)); % this will
ERROR if you change number of camera data rows
            timelow = GroupLog{6,group}(iprev);
            jprev = max(find(GroupLog{datatypes_cam + 5,group} <= timelow));
            % splitting power data
            for class = 1 + datatypes_cam: datatypes_pow + datatypes_cam
                LapLog{class,lap} = GroupLog{class,group}(jprev:j);
            end
            % splitting camera data
            for class = 1:datatypes_cam
                LapLog{class,lap} = GroupLog{class,group}(iprev:i-1);
            end
            iprev = i; % reset low shoulder to present (cam)
            lap = lap + 1; % add counter
        end
    end
end

% Adding some useful lap-specific data, zeroed to beginning of each lap
% 17.) Lap energy used
% 18.) Lap distance traveled

for lap = 1:length(LapLog)
    LapLog{6,lap} = LapLog{6,lap} - min(LapLog{6,lap}); % time for individual laps (cam)
    LapLog{16,lap} = LapLog{16,lap} - min(LapLog{16,lap}); % time for individual laps
(pow)
    LapLog{17,lap} = cumtrapz(LapLog{14,lap})/1000; % Energy (1000 Hz sampling rate)
    LapLog{20,lap} = LapLog{8,lap} - min(LapLog{8,lap}); % lap distance (total distance
is still 8.)
end

% filtered velocity and power
for lap = 1:length(LapLog)
    for entry = 1:length(LapLog{6,lap})-1
        sample_dif(entry) = (LapLog{6,lap}(entry+1) - LapLog{6,lap}(entry))^-1;
        samplerate_cam(lap) = mean(sample_dif); % find avg. sampling rate of cameras for
lap
    end
end
% assuming near constant sampling rate over course of a trial
samplerate_cam = mean(samplerate_cam);
Wn_cam = 0.1;
Wn_pow = Wn_cam*samplerate_cam/SamplingRate;
[B_cam,A_cam] = butter(2,Wn_cam);
[B_pow,A_pow] = butter(2,Wn_pow);
for i = 1:length(LapLog)
    LapLog{21,i} = filtfilt(B_cam,A_cam,LapLog{9,i}); % filtered velocity
    LapLog{22,i} = filtfilt(B_pow,A_pow,LapLog{14,i}); % filtered power
end

%% Making x-pos and y-pos interp classes for 3D power plot
for i = 1:length(LapLog)
Time_Cam{i} = single(LapLog{6,i});
Time_Pow{i} = single(LapLog{16,i});
    index = [];
    for k = 1:length(Time_Cam{i})
    index(k) = find(Time_Cam{i}(k) == Time_Pow{i});
    end
    for m = 2:length(index)
    space = 1/(index(m)-index(m-1));
    X_Interp{i}(index(m-1):index(m)) = linspace(LapLog{4,i}(m-1),LapLog{4,i}(m),index(m)-
index(m-1)+1);
    Y_Interp{i}(index(m-1):index(m)) = linspace(LapLog{5,i}(m-1),LapLog{5,i}(m),index(m)-
index(m-1)+1);
```

```
    end
    LapLog{18,i} = X_Interp{i}';
    LapLog{19,i} = Y_Interp{i}';
end

LapLogKey{1} = 'Iteration';
LapLogKey{2} = 'X-position(pix)';
LapLogKey{3} = 'Y-position(pix)';
LapLogKey{4} = 'X-position(ft)';
LapLogKey{5} = 'Y-position(ft)';
LapLogKey{6} = 'Lap time (camera) (s)';
LapLogKey{7} = 'Lap number';
LapLogKey{8} = 'Total distance (ft)';
LapLogKey{9} = 'Velocity (ft/s)';
LapLogKey{10} = 'Velocity_Filt (old)';
LapLogKey{11} = 'Common path deviation (ft)';

LapLogKey{12} = 'Voltage (V)';
LapLogKey{13} = 'Current (A)';
LapLogKey{14} = 'Power (J/s)';
LapLogKey{15} = 'Iteration (power)';
LapLogKey{16} = 'Lap time (power) (s)';

LapLogKey{17} = 'Lap energy (J)';
LapLogKey{18} = 'X-pos(interp) (ft)';
LapLogKey{19} = 'Y-pos(interp) (ft)';
LapLogKey{20} = 'Lap distance (ft)';
LapLogKey{21} = 'Velocity_Filt (ft/s)';
LapLogKey{22} = 'Power_Filt (ft/s)';

LapLogKey = LapLogKey';

save('LapLog.mat','LapLog','LapLogKey');
```

A.6 **Script6_Results.m**

```
% Results
clc
clear
close all
load LapLog.mat

% Converting LapLog
convert = 1.355817; % 1 ft-lb = 1.3558 J

for lap = 1:length(LapLog)
    for row = [14 17 22];
        LapLog{row,lap} = LapLog{row,lap}/convert;
    end
end

% %% Plotting engine
% % Specifiy which two parameters from LapLog you would like plotted.
% for i = 1:length(LapLog)
%     LapLogPlot{1,i} = LapLog{6,i}; % parameter 1
%     LapLogPlot{2,i} = LapLog{17,i}; % parameter 2
% end
% figure
% plot(LapLogPlot{:})

%% Lap Plots
% Position of robot (colored by lap)
for i = 1:length(LapLog)
    LapLogPlot{1,i} = LapLog{4,i}; % parameter 1
    LapLogPlot{2,i} = LapLog{5,i}; % parameter 2
end
```

```
fig1 = figure;
subplot(3,2,1)
plot(LapLogPlot{:})
title('robot position')
xlabel('x-position (ft)')
ylabel('y-position (ft)')
axis([0 25 0 8])

% Total distance vs. time (good for checking for errors)
for i = 1:length(LapLog)
    LapLogPlot{1,i} = LapLog{6,i}; % parameter 1
    LapLogPlot{2,i} = LapLog{20,i}; % parameter 2
end
subplot(3,2,2)
plot(LapLogPlot{:})
title('distance traveled vs. time')
xlabel('time (s)')
ylabel('distance (ft)')
axis([0 30 0 70])

% Velocity vs. time
for i = 1:length(LapLog)
    LapLogPlot{1,i} = LapLog{6,i}; % parameter 1
    LapLogPlot{2,i} = LapLog{21,i}; % parameter 2
end
subplot(3,2,3)
plot(LapLogPlot{:})
title('velocity vs. time')
xlabel('time (s)')
ylabel('velocity (ft/s)')
axis([0 30 0 3])

% Deviation vs. time
for i = 1:length(LapLog)
    LapLogPlot{1,i} = LapLog{6,i}; % parameter 1
    LapLogPlot{2,i} = LapLog{11,i}; % parameter 2
end
subplot(3,2,4)
plot(LapLogPlot{:})
title('deviation vs. time')
xlabel('time (s)')
ylabel('deviation (ft/s)')
axis([0 30 0 2])

% Power vs. time
for i = 1:length(LapLog)
    LapLogPlot{1,i} = LapLog{16,i}; % parameter 1
    LapLogPlot{2,i} = LapLog{22,i}; % parameter 2
end
subplot(3,2,5)
plot(LapLogPlot{:})
title('power vs. time')
xlabel('time (s)')
ylabel('power (ft-lb/s)')
axis([0 30 0 200/convert])

% Energy vs. time
for i = 1:length(LapLog)
    LapLogPlot{1,i} = LapLog{16,i}; % parameter 1
    LapLogPlot{2,i} = LapLog{17,i}; % parameter 2
end
subplot(3,2,6)
plot(LapLogPlot{:})
title('energy vs. time')
xlabel('time (s)')
ylabel('energy (ft-lb)')
axis([0 30 0 3000/convert])

%print(fig1,'-djpeg','-r1500','Lap_plots')
```

```
%% 3D Plots
% plot velocity vs. position
for i = 1:length(LapLog)
    LapLogPlot{1,i} = LapLog{4,i}; % parameter 1
    LapLogPlot{2,i} = LapLog{5,i}; % parameter 2
    LapLogPlot{3,i} = LapLog{21,i}; % parameter 3
end
fig2 = figure;
subplot(2,2,1)
plot3(LapLogPlot{:})
title('velocity vs. position')
xlabel('x-position (ft)')
ylabel('y-position (ft)')
zlabel('velocity (ft/s)')
axis([0 25 0 8 0 3])
grid on

% plot deviation vs. position
for i = 1:length(LapLog)
    LapLogPlot{1,i} = LapLog{4,i}; % parameter 1
    LapLogPlot{2,i} = LapLog{5,i}; % parameter 2
    LapLogPlot{3,i} = LapLog{11,i}; % parameter 3
end
subplot(2,2,2)
%figure
plot3(LapLogPlot{:})
title('deviation vs. position')
xlabel('x-position (ft)')
ylabel('y-position (ft)')
zlabel('deviation (ft)')
axis([0 25 0 8 0 2])
grid on

% plot power vs. position (using interpolated position points)
% downsample (for plotting purposes)
downsample = 100;
for i = 1:length(LapLog)
    LapLogPlot{1,i} = LapLog{18,i}(1:downsample:length(LapLog{18,i})); % parameter 1
    LapLogPlot{2,i} = LapLog{19,i}(1:downsample:length(LapLog{19,i})); % parameter 2
    LapLogPlot{3,i} = LapLog{22,i}(1:downsample:length(LapLog{22,i})); % parameter 3
end
subplot(2,2,3)
%figure
plot3(LapLogPlot{:})
title('power vs. position')
xlabel('x-position (ft)')
ylabel('y-position (ft)')
zlabel('power (ft-lb/s)')
axis([0 25 0 8 0 200/convert])
grid on

% plot energy vs. position (using interpolated points)
% downsample (for plotting purposes)
for i = 1:length(LapLog)
    LapLogPlot{1,i} = LapLog{18,i}(1:downsample:length(LapLog{18,i})); % parameter 1
    LapLogPlot{2,i} = LapLog{19,i}(1:downsample:length(LapLog{19,i})); % parameter 2
    LapLogPlot{3,i} = LapLog{17,i}(1:downsample:length(LapLog{17,i})); % parameter 3
end
subplot(2,2,4)
%figure
plot3(LapLogPlot{:})
title('energy vs. position')
xlabel('x-position (ft)')
ylabel('y-position (ft)')
zlabel('energy (ft-lb)')
axis([0 25 0 8 0 3000/convert])
grid on
```

```
%print(fig2,'-djpeg','-r1500','Lap_plots_3D')

%% Cumulative Lap Trend Plots
LapNumber = 1:length(LapLog);
barlaps = 60;
% Time to complete each lap
for i = 1:length(LapLog)
    LapTime(i) = LapLog{6,i}(end);
end
fig3 = figure;
subplot(3,2,1)
title('time to complete each lap')
xlabel('lap number')
ylabel('time (s)')
hold on
k = 1; % subplot counter
% best fit line
P{k} = polyfit(LapNumber,LapTime,1);
Y{k} = LapNumber*P{k}(1) + P{k}(2);
plot(LapNumber,Y{k},'r','linewidth',2)
eqn = ['y = ' sprintf('%3.3fx + %3.3f',[P{k}])];
legend(eqn,'Location','south');
bar(LapTime)
axis([0 barlaps 0 30])
k = k + 1;

% Distance traveled each lap
for i = 1:length(LapLog)
    LapDist(i) = LapLog{20,i}(end);
end
subplot(3,2,2)
title('distance traveled each lap')
xlabel('lap number')
ylabel('distance (ft)')
hold on
% best fit line
P{k} = polyfit(LapNumber,LapDist,1);
Y{k} = LapNumber*P{k}(1) + P{k}(2);
plot(LapNumber,Y{k},'r','linewidth',2)
eqn = ['y = ' sprintf('%3.3fx + %3.3f',[P{k}])];
legend(eqn,'Location','south');
bar(LapDist)
axis([0 barlaps 0 75])
k = k + 1;

% Average velocity each lap
for i = 1:length(LapLog)
    LapVel(i) = mean(LapLog{9,i});
end
subplot(3,2,3)
title('avg velocity each lap')
xlabel('lap number')
ylabel('velocity (ft/s)')
hold on
% best fit line
P{k} = polyfit(LapNumber,LapVel,1);
Y{k} = LapNumber*P{k}(1) + P{k}(2);
plot(LapNumber,Y{k},'r','linewidth',2)
eqn = ['y = ' sprintf('%3.3fx + %3.3f',[P{k}])];
legend(eqn,'Location','south');
bar(LapVel)
axis([0 barlaps 0 3])
k = k + 1;

% Cumulative pathdev each lap
for i = 1:length(LapLog)
    LapDev(i) = sum(LapLog{11,i});
    % normalize to number of frames
    LapDev_Norm(i) = LapDev(i)/length(LapLog{1,i});
```

```
end
% Normalize to maximum deviation (optional):
% LapDev_Norm2 = LapDev_Norm/max(LapDev_Norm);
subplot(3,2,4)
title('deviation from common path each lap')
xlabel('lap number')
ylabel('lap deviation (ft/s)')
hold on
% best fit line
P{k} = polyfit(LapNumber,LapDev_Norm,1);
Y{k} = LapNumber*P{k}(1) + P{k}(2);
plot(LapNumber,Y{k},'r','linewidth',2)
eqn = ['y = ' sprintf('%3.3fx + %3.3f',[P{k}])];
bar(LapDev_Norm)
legend(eqn,'Location','south');
axis([0 barlaps 0 0.7])
k = k + 1;

% Average power each lap
for i = 1:length(LapLog)
    LapPow(i) = mean(LapLog{14,i});
end
subplot(3,2,5)
title('avg power each lap')
xlabel('lap number')
ylabel('power (ft-lb/s)')
hold on
% best fit line
P{k} = polyfit(LapNumber,LapPow,1);
Y{k} = LapNumber*P{k}(1) + P{k}(2);
plot(LapNumber,Y{k},'r','linewidth',2)
eqn = ['y = ' sprintf('%3.3fx + %3.3f',[P{k}])];
legend(eqn,'Location','south');
bar(LapPow)
axis([0 barlaps 0 140/convert])
k = k + 1;

% Energy consumed each lap
for i = 1:length(LapLog)
    LapEnergy(i) = LapLog{17,i}(end);
end
subplot(3,2,6)
title('energy consumed each lap')
xlabel('lap number')
ylabel('energy (ft-lb)')
hold on
% best fit line
P{k} = polyfit(LapNumber,LapEnergy,1);
Y{k} = LapNumber*P{k}(1) + P{k}(2);
plot(LapNumber,Y{k},'r','linewidth',2)
eqn = ['y = ' sprintf('%3.3fx + %3.3f',[P{k}])];
legend(eqn,'Location','south');
bar(LapEnergy)
axis([0 barlaps 0 3000/convert])
k = k + 1;

%print(fig3,'-djpeg','-r1500','Lap_trends')

%% Final test statistics

[M,N] = size(LapLog);
LapLog11 = [];
LapLog21 = [];
LapLog22 = [];
for i = 1:N
    LapLog6(i) = LapLog{6,i}(end); % time
    LapLog20(i) = LapLog{20,i}(end); % distance
    LapLog21 = [LapLog21;LapLog{21,i}]; %vel
    LapLog11 = [LapLog11;LapLog{11,i}]; %dev
```

```
    LapLog22 = [LapLog22;LapLog{22,i}]; %pow
    LapLog17(i) = LapLog{17,i}(end);
end

% Key
ResultStatsKey{1} = 'Total laps completed';
ResultStatsKey{2} = 'Total time (s)';
ResultStatsKey{3} = 'Total distance (ft)';
ResultStatsKey{4} = 'Total energy (ft-lb)';
ResultStatsKey{5} = 'Lap time avg (s)';
ResultStatsKey{6} = 'Lap time std (s)';
ResultStatsKey{7} = 'Lap distance avg (ft)';
ResultStatsKey{8} = 'Lap distance std (ft)';
ResultStatsKey{9} = 'Lap velocity avg (ft/s)';
ResultStatsKey{10} = 'Lap velocity std (ft/s)';
ResultStatsKey{11} = 'Lap deviation avg (ft)';
ResultStatsKey{12} = 'Lap deviation std (ft)';
ResultStatsKey{13} = 'Lap power avg (ft-lb/s)';
ResultStatsKey{14} = 'Lap power std (ft-lb/s)';
ResultStatsKey{15} = 'Lap energy avg (ft-lb)';
ResultStatsKey{16} = 'Lap energy std (ft-lb)';
ResultStatsKey = ResultStatsKey';

ResultStats(1) = LapLog{7,end}(1) + 1; % total laps completed
ResultStats(2) = sum(LapLog6); % total time
ResultStats(3) = sum(LapLog20);  % total distance
ResultStats(4) = sum(LapLog17); % total energy
ResultStats(5) = mean(LapLog6); % avg lap time
ResultStats(6) = std(LapLog6); % std lap time
ResultStats(7) = mean(LapLog20); % avg lap distance
ResultStats(8) = std(LapLog20); % std lap distance
ResultStats(9) = mean(LapLog21); % avg lap velocity
ResultStats(10) = std(LapLog21); % std lap velocity
ResultStats(11) = mean(LapLog11); % avg lap deviation
ResultStats(12) = std(LapLog11); % std lap deviation
ResultStats(13) = mean(LapLog22); % avg lap power
ResultStats(14) = std(LapLog22); % std lap power
ResultStats(15) = mean(LapLog17); % avg lap energy
ResultStats(16) = std(LapLog17); % std lap energy
ResultStats = ResultStats';

% Cell array of trendline information (slopes and intercepts)
TrendStats = P;
% Saving final information
save('ResultStats.mat','ResultStats','ResultStatsKey','TrendStats');
```

A.7 **Script_Debug_CalibDistort.m**

```
% This script loads in test calibration images.
% For each image, tweek parameters and resulting image is displayed.
% Calls function which employs process.

% Image 1
I = imread('TestImg_1.jpg');
imshow(I)
XPixRight = 9;
YPixDown = -20;
RotDegCCW = 0.5;
K = -0.4;
figure
[im1,input_points,base_points] =
FcnUndistort_Transform_Calib(I,XPixRight,YPixDown,RotDegCCW,K);
imshow(im1)

% input_points =   [121.1789  318.0222
%   120.2525   99.8560
```

```
%   354.6307  320.3382
%   361.1155   97.5400];
% base_points =  [120.7157  318.4854
%   120.7157  100.7824
%   360.6523  319.4118
%   361.1155   98.9296];

%% Image 2
I = imread('TestImg_2.jpg');
imshow(I)
XPixRight = 16;
YPixDown = 0;
RotDegCCW = 1;
K = -0.4;
figure
[im1,input_points,base_points] =
FcnUndistort_Transform_Calib(I,XPixRight,YPixDown,RotDegCCW,K);
imshow(im1)

% input_points =  [120.5209  302.0057
%   125.6589   75.9348
%   371.3475  301.0715
%   365.2753   72.1981];
% base_points =  [126.5930  302.0057
%   126.1259   76.4019
%   364.8082  300.6044
%   364.8082   76.4019];

%% Image 3
I = imread('TestImg_3.jpg');
imshow(I)
XPixRight = 22;
YPixDown = 0;
RotDegCCW = -.5;
K = -0.3;
figure
[im1,input_points,base_points] =
FcnUndistort_Transform_Calib(I,XPixRight,YPixDown,RotDegCCW,K);
imshow(im1)

% input_points = [96.1662  299.0311
%    99.8718   32.2291
%   381.9594   32.2291
%   391.6865  299.9575];
% base_points = [96.6294  298.1047
%    97.0926   31.7659
%   393.0761   31.7659
%   394.0025  298.1047];
```

A.8 **Script_Debug_RawPower.m**

```
% This script looks at raw power data (for debugging purposes)
clear
clc

Listing = dir('F:\ARL_New\MATLAB_4\power_logger\*.CSV');

% Get the number of files of data
NumFiles = length(Listing);

% Initialize 'RawData'
RawData = [];

% For every file...
for n=1:NumFiles
```

```
    % Get the data from the current file
    Import = CSVread(strcat('F:\ARL_New\MATLAB_4\power_logger\',Listing(n,1).name),10,0);

    % Add the data to 'RawData'
    RawData(length(RawData)+1:length(RawData)+length(Import),:) = Import;
end

% Produce powerlog file with voltage, current, power

PowerLog(:,1) = RawData(:,2); % voltage
PowerLog(:,2) = RawData(:,3); % current
PowerLog(:,3) = PowerLog(:,1).*PowerLog(:,2); % power

SamplingRate = 1000;

%Sample = PowerLog(:,1);
Voltage = PowerLog(:,1);
Current = PowerLog(:,2);
Power = Voltage.*Current;
time = (1:length(Power))/SamplingRate;

figure
plot(time,Power,'b')
xlabel('Time (s)')
ylabel('Power (J/s)')
title('power data')
```

A.9 **Script_Debug_Realspace.m**

```
% This script allows users to plot calibrated camera images in realspace,
% good for verifying calibration
clear
clc

% Initialize variables for the code that users may want to modify
[ FlagLive,FlagPlot,FlagSavePlot,NumCams,Data2File ] = FcnInitTestConditions;

% Initialize parameters for the cameras
[ IP,CamRes,CamParam ] = FcnInitCamParams(FlagLive);

% Initialize variables for the code
[
Iter,CentroidFT_Last,CentroidPX_Last,CentroidPX_Last_Raw,TotalLaps,TotalDist,LastZone,Fla
gObjFound,DataLog,TimeStamps ] = FcnInitVars( Data2File,FlagLive );

%% Loading pixel image
im = FcnGetImage_All( IP,CamRes,FlagLive,TimeStamps,Iter,NumCams);
% Undistort the images
load DataCalibCamDistort.mat
im = FcnUndistort(im,DistortionMapping,NumCams,CamRes);

% Real-world position calibration for mapping later
load DataCalibDistTrack.mat

figure
imshow(im)
% im2 = imresize(im,[500 1080]);
% figure (2)
% imshow(im2)

% Grayscale
imgray = rgb2gray(im);
% Contrast
imgray = imadjust(imgray,[.15 .95],[]);
figure
imshow(imgray)
```

```
% Splitting imgray into individual images
im1 = imgray(:,1:360);
im2 = imgray(:,361:720);
im3 = imgray(:,721:1080);
im1Vec = im1(:);
im2Vec = im2(:);
im3Vec = im3(:);

%% Creating meshpoints from CalibDistTrack
% Cam1 meshgrid
[Cam1MeshX,Cam1MeshY] = meshgrid(CalibDistTrack(1:360,1,1),CalibDistTrack(:,2,1));
Cam1VecX = Cam1MeshX(:);
Cam1VecY = Cam1MeshY(:);
% Cam2 meshgrid
[Cam2MeshX,Cam2MeshY] = meshgrid(CalibDistTrack(1:360,1,2),CalibDistTrack(:,2,2));
Cam2VecX = Cam2MeshX(:);
Cam2VecY = Cam2MeshY(:);
% Cam3 meshgrid
[Cam3MeshX,Cam3MeshY] = meshgrid(CalibDistTrack(1:360,1,3),CalibDistTrack(:,2,3));
Cam3VecX = Cam3MeshX(:);
Cam3VecY = Cam3MeshY(:);

CamTotVecX = [Cam1VecX;Cam2VecX;Cam3VecX];
CamTotVecY = [Cam1VecY;Cam2VecY;Cam3VecY];
imTotVec = [im1Vec;im2Vec;im3Vec];

figure
scatter(CamTotVecX,CamTotVecY,[4],imTotVec);
colormap(gray)
title('Camera real-space mapping')
xlabel('x-position (ft)')
ylabel('y-position (ft)')
```

A.10 **Script_Debug_Velocity.m**

```
% Plot individual data points, from TrialLog.mat

load DataLog.mat
DataLog = FcnDataLogZeros;
X_Pos = DataLog(:,4);
Y_Pos = DataLog(:,5);
figure
plot(X_Pos,Y_Pos,'bs:')
xlabel('x-pos (ft)')
ylabel('y-pos (ft)')

% Plot velocity
[Velocity,Velocity_Filt] = FcnVelocity(DataLog);
DataLog(:,9) = Velocity;
DataLog(:,10) = Velocity_Filt;
figure
plot3(X_Pos,Y_Pos,Velocity_Filt,'bs:')
grid on
xlabel('x-pos (ft)')
ylabel('x-pos (ft)')
zlabel('velocity (ft/s)')
```

A.11 **Script_Skew.m**

```
% This script uses position data from TrialLog.mat to calculate average
% path taken over a whole test. (Work this script into script 4).
```

```
% Average path data will be plotted, and compared to most common path data,
% also plotted.

close all
clear
clc

load TrialLog.mat
load DataLog.mat
DataLog = FcnDataLogZeros;

%%
X_Pos = DataLogInterp(:,4);
Y_Pos = DataLogInterp(:,5);
Coord = [X_Pos,Y_Pos];

%% Selecting center of figure-8
plot(X_Pos,Y_Pos)
display('Select center of figure-8 (approx.)')
[X_Mid,Y_Mid] = ginput

% Grouping data into two halves
countA = 1;
countB = 1;
for i = 1:length(Coord)
    if Coord(i,1) < X_Mid
        GroupA(countA,:) = Coord(i,:);
        countA = countA + 1;
    else
        GroupB(countB,:) = Coord(i,:);
        countB = countB + 1;
    end
end

% Selecting center points of each half
display('Select midpoint of left loop')
[X_CentA,Y_CentA] = ginput;
display('Select midpoint of right loop')
[X_CentB,Y_CentB] = ginput;

%% Side A / Side B
[X_AvgPathA,Y_AvgPathA,X_Std1UpA,Y_Std1UpA,X_Std1DownA,Y_Std1DownA] =
FcnArcAvg(GroupA,X_CentA,Y_CentA);
[X_AvgPathB,Y_AvgPathB,X_Std1UpB,Y_Std1UpB,X_Std1DownB,Y_Std1DownB] =
FcnArcAvg(GroupB,X_CentB,Y_CentB);

figure
hold on
plot(X_AvgPathA,Y_AvgPathA,'r',X_Std1UpA,Y_Std1UpA,'g',X_Std1DownA,Y_Std1DownA,'g');
plot(X_AvgPathB,Y_AvgPathB,'r',X_Std1UpB,Y_Std1UpB,'g',X_Std1DownB,Y_Std1DownB,'g');
%set(gca,'Color',[0 0 0])

%% Generating common path
[C1,C2] = meshgrid(C{1},C{2});

%% Most common path overlay
count = 1;
CommonPoints = [C1(:),C2(:),Ridge(:)];
for i = 1:length(CommonPoints)
    if Ridge(i) ~= 0
        CommonSparse(count,:) = CommonPoints(i,:);
        count = count + 1;
    end
end

scatter(CommonSparse(:,1),CommonSparse(:,2))
```

A.12 **FcnArcAvg.m**

```
function [X_AvgPath,Y_AvgPath,X_Std1Up,Y_Std1Up,X_Std1Down,Y_Std1Down] =
FcnArcAvg(Group,X_Cent,Y_Cent)

% This function takes the position data from one side of a figure-8, and
% the centerpoint, and calculates the average and +1 and -1 std path.

Shift(:,1) = Group(:,1) - X_Cent;
Shift(:,2) = Group(:,2) - Y_Cent;
[Theta,Rho] = cart2pol(Shift(:,1),Shift(:,2));
Pol = [Theta,Rho];
Sort = sortrows(Pol);

%figure
%polar(Sort(:,1),Sort(:,2),'square')

arcsize = 1; % in degrees
binbounds = [-180:arcsize:180]*pi/180;


for i = 1:length(binbounds)-1
    count_bin = 1;
    for k = 1:length(Sort)
        if Sort(k,1) >= binbounds(i) && Sort(k,1) < binbounds(i+1)
            ThetaBins{i}(count_bin,:) = Sort(k,:);
            count_bin = count_bin + 1;
        end
    end
end

for i = 1:length(ThetaBins)
    Rho_Avg(i) = mean(ThetaBins{i}(:,2));
    Rho_Std(i) = std(ThetaBins{i}(:,2));
end

Bin_Centers = binbounds + arcsize/2*(pi/180);
Bin_Centers(end) = [];

Rho_Std1Up = Rho_Avg + Rho_Std;
Rho_Std1Down = Rho_Avg - Rho_Std;

[X_AvgPath,Y_AvgPath] = pol2cart(Bin_Centers,Rho_Avg);
[X_Std1Up,Y_Std1Up] = pol2cart(Bin_Centers,Rho_Std1Up);
[X_Std1Down,Y_Std1Down] = pol2cart(Bin_Centers,Rho_Std1Down);

X_AvgPath = X_AvgPath + X_Cent;
Y_AvgPath = Y_AvgPath + Y_Cent;
X_Std1Up = X_Std1Up + X_Cent;
Y_Std1Up = Y_Std1Up + Y_Cent;
X_Std1Down = X_Std1Down + X_Cent;
Y_Std1Down = Y_Std1Down + Y_Cent;

end
```

A.13 **FcnCalcDist.m**

```
function [ TotalDist ] = FcnCalcDist( TotalDist,CentroidFT_Last,CentroidFT_Current )

% This function uses the distance formula to compute the distance traveled
% by the fiducial since the last frame.
% It then adds this to the previous total distance to find a new total.


% Calculate the distance traveled between iterations
```

```
DistChange = sqrt( ( CentroidFT_Current(1)-CentroidFT_Last(1) )^2 + (
CentroidFT_Current(2)-CentroidFT_Last(2) )^2 );

% Use for horizontal axis distance calibrations
%DistChange = abs(CentroidFT_Current(1)-CentroidFT_Last(1));

% Use for vertical axis distance calibrations
%DistChange = abs(CentroidFT_Current(2)-CentroidFT_Last(2));

% Add the distance traveled between iterations to the previous total
TotalDist = TotalDist + DistChange;

end
```

A.14 **FcnCalcLaps.m**

```
function [ TotalLaps,LastZone ] = FcnCalcLaps(
CalibEndzones,TotalLaps,LastZone,CentroidPX_Current )

% This function keeps tracks of laps completed by the fiducial.


% Extract endzone parameters;
Lm = CalibEndzones(1,1);
Lb = CalibEndzones(1,2);
Rm = CalibEndzones(2,1);
Rb = CalibEndzones(2,2);

% When the object is first in an endzone, start lap counting by changing
% 'LastZone' to 1 or 2 depending on the endzone it is in
if LastZone == 0
    if CentroidPX_Current(1) >= CentroidPX_Current(2)*Rm+Rb
        LastZone = 1;
    end
    if CentroidPX_Current(1) <= CentroidPX_Current(2)*Lm+Lb
        LastZone = 2;
    end
end

% Look for the fiducial to enter the opposite endzone from the last it entered
% and update the lap counter
if LastZone == 1
    if CentroidPX_Current(1)<=CentroidPX_Current(2)*Lm+Lb
        LastZone = 2;
        TotalLaps = TotalLaps+.5;
    end
end

if LastZone==2
    if CentroidPX_Current(1)>=CentroidPX_Current(2)*Rm+Rb
        LastZone = 1;
        TotalLaps = TotalLaps+.5;
    end
end

end
```

A.15 **FcnDataLogZeros.m**

```
function [DataLog] = FcnDataLogZeros()

% This function removes the zeros off the end of the datalog matrix
load DataLog.mat
Endlog = max(DataLog,[],1);
```

```
MaxIter = Endlog(1);
DataLog = DataLog(1:MaxIter,:);

end
```

A.16 **FcnGetCalibrations.m**

```
function [ CalibDistTrack,CalibEndzones,newlocation,DistortionMapping,CalibBlackBars] =
FcnGetCalibrations

% This function loads data files for calibrations necessary to the algorithms


% Initialize a flag for whether we have all the necessary data files so that the loop
runs at least once
Flag = 0;

% While we DON'T have all the data files
while Flag == 0

    % Try to load all the data files and set Flag = 1 if we make it all the way through
    try
        load DataCalibDistTrack.mat % Creates variable: CalibDistTrack
        load DataCalibEndzones.mat  % Creates variable: CalibEndzones
        load DataCalibCamDistort.mat   % Creates variable: CalibDistort
        load DataCalibBlackBars.mat % Creates variable: CalibBlackBars
        Flag = 1;
    catch fail
        Flag = 0;
    end

    % If we DIDN'T find all the data files last iteration, run the calibration script
    if Flag == 0
        commandwindow
        disp('WARNING: One or more of the calibration data files could not be found.
Check which is missing and press any key to run the calibration script!');
        pause;
        ScriptCalibrate
        Flag = 0;
    end
end
end
```

A.17 **FcnGetImage.m**

```
function [ im ] = FcnGetImage( IP,FlagLive,TimeStamps,Iter,CamNum )

% This function loads an image from a camera and corrects it for barrel distortion.


% If taking images in real-time
if FlagLive == 1

    % Get IP address for the camera
    name = IP{CamNum};

    % Load image from the camera
    im = imread(name);

    % If loading image from file
else

    % Use the first image in the initialiation step
    if Iter ==0
```

```
        Iter = 1;
    end

    % Generate the file name for each image to be loaded
    name =
strcat('images_PY/','cam_',num2str(CamNum),'/',num2str(TimeStamps(Iter),'%f'),'.jpg' );

    % Load image from file
    im = imread(name);
end

% Rotate the image appropriately
switch CamNum
    case 1
        im = imrotate(im,-90);
    case 2
        im = imrotate(im,-90);
    case 3
        im = imrotate(im,-90);
end

% Crop out the top and bottom of the image
im(1:50,1:360,:)=0;
im(450:480,1:360,:)=0;

end
```

A.18 **FcnGetImage_All.m**

```
function [ im ] = FcnGetImage_All( IP,CamRes,FlagLive,TimeStamps,Iter,NumCams )

% This function gets an image from every camera in the system.


% Make an empty matrix to hold the camera images
im = uint8(zeros(CamRes(1),CamRes(2)*NumCams,3));

% Get an image from each camera and concatenate
for CamNum=1:NumCams
    newim = FcnGetImage( IP,FlagLive,TimeStamps,Iter,CamNum );

    if CamNum == 1
        im(:,1:CamRes(2),:) = newim;
    else
        im(:,(CamRes(2)*(CamNum-1) ) + 1:CamRes(2)*CamNum,:) = newim;
    end
end

end
```

A.19 **FcnGetImage_Select.m**

```
function [ im,LeftBound,TopBound ] = FcnGetImage_Select(
IP,CamRes,FlagLive,TimeStamps,Iter,NumCams,CentroidPX_Last,CalibBlackBars)

% This function get images from only those cameras within 'MatWidth' pixels of
% the last fiducial location and crops the image to be more or less
% centered at that location.


% Set the width of the box to search, more or less centered at the previous centroid
location
MatWidth = 300;  % The defualt is 300
```

```
LeftBound   = CentroidPX_Last(1)-MatWidth/2; % Set the left boundary
RightBound  = CentroidPX_Last(1)+MatWidth/2; % Set the right boundary
TopBound    = CentroidPX_Last(2)-MatWidth/2; % Set the top boundary
BottomBound = CentroidPX_Last(2)+MatWidth/2; % Set the bottom boundary

% If the left boundary is out of bounds, set it as the boundary
if LeftBound  < 1
    LeftBound = 1;
end
% If the right boundary is out of bounds, set it as the boundary
if RightBound  > NumCams*CamRes(2)
    RightBound = NumCams*CamRes(2);
end
% If the top boundary is out of bounds, set it as the boundary
if TopBound  < 1
    TopBound = 1;
end
% If the bottom boundary is out of bounds, set it as the boundary
if BottomBound  > CamRes(1)
    BottomBound = CamRes(1);
end

% Get the number of the camera in which each bound lies
CamNum_Left     = ceil( LeftBound     / CamRes(2) );
CamNum_Right    = ceil( RightBound    / CamRes(2) );

% Generate a list of the border cameras from from which we need images
Cams = sort(unique([CamNum_Left,CamNum_Right]));

% If cameras 1 and 3 are needed, add in camera 2 as well
if Cams == [1 3]
    Cams = [1 2 3];
end

% Make an empty matrix to hold the camera images
im = uint8(zeros(CamRes(1),CamRes(2)*max(Cams),3));

% Get an image from each camera
for CamIndex=1:size(Cams,2)
    newim = FcnGetImage( IP,FlagLive,TimeStamps,Iter,Cams(CamIndex) );

    % Crop out overlap in the images
    % (this is done in FcnUndistort when FlagPlot is 1
    switch Cams(CamIndex)
        case 1
            % newim(1:480,330:360,:) = 0;
            % black bars
            newim(:,CalibBlackBars(1):CalibBlackBars(2),:) = 0;
            newim(:,CalibBlackBars(3):CalibBlackBars(4),:) = 0;
            % cone boxes

newim(CalibBlackBars(5,2):CalibBlackBars(6,2),CalibBlackBars(5,1):CalibBlackBars(6,1),:)
= 0;

newim(CalibBlackBars(7,2):CalibBlackBars(8,2),CalibBlackBars(7,1):CalibBlackBars(8,1),:)
= 0;

        case 2
            % newim(1:480,1:30,:) = 0;
        case 3
            % newim(1:480,330:360,:) = 0;
        case 4
            % newim(1:480,1:10,:) = 0;
    end

    if CamIndex == 1
        im(:,1:CamRes(2),:) = newim;
    else
        im(:,(CamRes(2)*Cams(CamIndex-1) ) + 1:CamRes(2)*Cams(CamIndex),:) = newim;
```

```
    end
end

% Adjust the bounds to refer to the indices of the images we just compliled
LeftBound_Ref  = LeftBound  - (Cams(1) - 1)*CamRes(2);
RightBound_Ref = RightBound - (Cams(1) - 1)*CamRes(2);

% Crop the image by taking these bounds
im = im( TopBound:BottomBound,LeftBound_Ref:RightBound_Ref,: );
end
```

A.20 **FcnGetPosition.m**

```
function [ CentroidFT_Current,CentroidPX_Current,CentroidPX_Current_Raw,FlagObjFound ] =
FcnGetPosition(
IP,CamRes,FlagLive,TimeStamps,FlagObjFound,Iter,NumCams,CalibDistTrack,CalibEndzones,Cent
roidFT_Last,CentroidPX_Last,CentroidPX_Last_Raw,FlagPlot,newlocation,DistortionMapping,Ca
libBlackBars)

% This function gets the position of the fiducial in both pixels and feet.

if FlagObjFound == 1 && FlagPlot == 0
    % If we found the fiducial last iteration AND are NOT plotting every iteration,
    % get an image from nearby cameras and crop the image around the last known position
    [ im,LeftBound,TopBound ] = FcnGetImage_Select(
IP,CamRes,FlagLive,TimeStamps,Iter,NumCams,CentroidPX_Last,CalibBlackBars );
    FlagScopeLimited = 1;
else
    % If we didn't find the fiducial last time, get an image from every camera
    im = FcnGetImage_All( IP,CamRes,FlagLive,TimeStamps,Iter,NumCams );
    FlagScopeLimited = 0;
end

% If we're in the zero iteration, can test to benchmark validity of
% the distortion lookup table.
if Iter==0
    im_bench_transform = FcnUndistort_Transform(im,NumCams,CamRes);
    im_bench_lookup = FcnUndistort(im,DistortionMapping,NumCams,CamRes);
    im_bench_difference = abs(im_bench_transform - im_bench_lookup);
    figure
    subplot(3,1,1)
    imshow(im_bench_transform)
    title('image comparison - transform')
    subplot(3,1,2)
    imshow(im_bench_lookup)
    title('image comparison - lookup')
    subplot(3,1,3)
    imshow(im_bench_difference)
    title('image comparison - difference')
end

% If we're IN the zeroth iteration OR the plot flag is ON...
if Iter==0 || FlagPlot == 1
    % Undistort the entire image
    im = FcnUndistort(im,DistortionMapping,NumCams,CamRes);
    % Display black bars on image
    im(:,CalibBlackBars(1):CalibBlackBars(2),:) = 0;
    im(:,CalibBlackBars(3):CalibBlackBars(4),:) = 0;
    % Display cone boxes on image
    im(CalibBlackBars(5,2):CalibBlackBars(6,2),CalibBlackBars(5,1):CalibBlackBars(6,1),:)
= 0;
    im(CalibBlackBars(7,2):CalibBlackBars(8,2),CalibBlackBars(7,1):CalibBlackBars(8,1),:)
= 0;
end

% Get the fiducial location in pixels and update FlagObjFound
```

```
[ Mask,FlagObjFound,CentroidPX_Current ] = FcnMask( im,CentroidPX_Last );

% If the fiducial was found...
if FlagObjFound == 1

    % If the scope was limited, adjust the centroid locations to refer to
    % the full range of the camera images
    if FlagScopeLimited == 1
        CentroidPX_Current(1) = CentroidPX_Current(1) + LeftBound;
        CentroidPX_Current(2) = CentroidPX_Current(2) + TopBound;
    end

    one_sigmaX  = .7;%= 0.2265;
    one_sigmaY  = .7;%= 0.2477;
    % If we are within the 3-sigma bounds of the expected steady state noise, set the
position to be equal to the last position
    if FlagPlot == 1
        if abs(CentroidPX_Current(1)-CentroidPX_Last(1)) < 3*one_sigmaX
            CentroidPX_Current(1) = CentroidPX_Last_Raw(1);
        end
        if abs(CentroidPX_Current(2)-CentroidPX_Last(2)) < 3*one_sigmaY
            CentroidPX_Current(2) = CentroidPX_Last_Raw(2);
        end
    elseif FlagPlot == 0
        if abs(CentroidPX_Current(1)-CentroidPX_Last_Raw(1)) < 3*one_sigmaX
            CentroidPX_Current(1) = CentroidPX_Last_Raw(1);
        end
        if abs(CentroidPX_Current(2)-CentroidPX_Last_Raw(2)) < 3*one_sigmaY
            CentroidPX_Current(2) = CentroidPX_Last_Raw(2);
        end
    end

    % Round the centroid locations to integers
    CentroidPX_Current = round(CentroidPX_Current);

    % Save the pixel position (for when FlagPlot = 0
    CentroidPX_Current_Raw = CentroidPX_Current;

    % Get the number of the camera in which the fiducial was found
    CamNum = ceil( CentroidPX_Current(1) / CamRes(2) );

    % Get the X location of fiducial WRT that camera's indices only
    CentroidPX_Current_Ref = CentroidPX_Current(1) - ( (CamNum-1) * (CamRes(2)) );

    % If we're AFTER the zeroth iteration and the plot flag is OFF...
    if Iter~=0 && FlagPlot == 0

        % If we have not undistorted the entire image
        if FlagScopeLimited == 1
            % Correct the centroid position only for barrel distortion
            linearInd =
sub2ind([CamRes(2),CamRes(1),1],CentroidPX_Current_Ref,CentroidPX_Current(2));
            [CentroidPX_Current_Ref, CentroidPX_Current(2)] =
ind2sub([CamRes(2),CamRes(1),1],newlocation(linearInd,CamNum));
        end
    end

    % Get the real-world coordinates of the pixels
    CentroidFT_Current(1) = CalibDistTrack(CentroidPX_Current_Ref,1,CamNum);
    CentroidFT_Current(2) = CalibDistTrack(CentroidPX_Current(2),2,CamNum);
    %CentroidFT_Current(2) = CalibDistTrack(480-CentroidPX_Current(2) ,2,CamNum);

else
    % Otherwise, position variables don't change
    CentroidFT_Current = CentroidFT_Last;
    CentroidPX_Current = CentroidPX_Last;
    CentroidPX_Current_Raw = CentroidPX_Last_Raw;
end
```

```
% If we're IN the zeroth iteration OR the plot flag is ON...
if Iter==0 || FlagPlot == 1

    if FlagObjFound == 0 && Iter == 0
        % If we're in the zeroth iteration and the object is not found, display an error
message
        error('Object not found at first check.  Please place it in view of the camera
and run the script again.')
    else

        % Clear the figure window
        if Iter > 1
            pause(.01)
            clf
        end

        % If the object is found, plot it

FcnPlot(im,Mask,CalibEndzones,CentroidPX_Current,CentroidFT_Current,CamRes(2)*NumCams,Cam
Res(1),FlagObjFound );
    end
end
```

A.21 **FcnGetTimestamps.m**

```
function [ TimeStamps ] = FcnGetTimestamps()

% This function extract timestamps from the image filenames

% Check for the necessary file and don't run without it
while isdir('images_PY') == 0
    disp('WARNING! The "images_PY" folder was not found in the current directory.  Move
it there and press any key to continue. ');
    pause
end
while isdir('images_PY/cam_1') == 0
    disp('WARNING! The "cam_1" folder was not found in the current directory.  Move it
there and press any key to continue. ');
    pause
end
while isdir('images_PY/cam_2') == 0
    disp('WARNING! The "cam_2" folder was not found in the current directory.  Move it
there and press any key to continue. ');
    pause
end
while isdir('images_PY/cam_3') == 0
    disp('WARNING! The "cam_3" folder was not found in the current directory.  Move it
there and press any key to continue. ');
    pause
end

% Get the filenames in the images_PY folder
listing = dir('images_PY/cam_1/*.jpg');

% Initialze a variable for iteration number
Iter = 1;

% Cycle through all the files in the folder
for file=1:length(listing)

    % Get the number of characters in the filename
    numchars = length(listing(file).name);

    % If it is a valid filename and for camera 1
    if numchars > 8
        % Save the timestamps
```

```
        TimeStamps(Iter,1) = str2num(listing(file).name(1:(numchars-4)));

        % Increase the iteration counter
        Iter = Iter+1;
    end
end

TimeStamps = sort(TimeStamps);

end
```

A.22 **FcnInitBlackBars.m**

```
function [] = FcnInitBlackBars(IP,CamRes,FlagLive,TimeStamps,Iter,NumCams)
%  This function initializes the process of generating or loading black bar
%  calibrations, to eliminate overlap in the camera images.

% Use a dialogue to ask whether the user wants to create new calibrations
choice = questdlg('Load last black bar calibrations or create new ones?', ...
    'Black Bar Calibrations', ...
    'Use Last','Create New','Create New');
% Handle response
switch choice
    case 'Create New'
        CalibBlackBars =
FcnInitBlackBars_Calib(IP,CamRes,FlagLive,TimeStamps,Iter,NumCams); % Take new
calibrations
        save('DataCalibBlackBars.mat','CalibBlackBars')
end
end
```

A.23 **FcnInitBlackBars_Calib.m**

```
function [ CalibBlackBars ] = FcnInitBlackBars_Calib(
IP,CamRes,FlagLive,TimeStamps,Iter,NumCams )
% This functions generates the black bar calibrations

% Get an image for each camera
im = FcnGetImage_All( IP,CamRes,FlagLive,TimeStamps,Iter,NumCams);

% Undistort the images
load DataCalibCamDistort.mat
im = FcnUndistort(im,DistortionMapping,NumCams,CamRes);

figure (1)
imshow(im)

disp('Select points as where to put black bars in between camera images.')
disp('Choose points in order of left to right.')
disp('Select two corners (top-left to bottom-right) of boxes for cones')
CalibBlackBars = ginput;
CalibBlackBars = round(CalibBlackBars);
im(:,CalibBlackBars(1):CalibBlackBars(2),:) = 0;
im(:,CalibBlackBars(3):CalibBlackBars(4),:) = 0;
im(:,CalibBlackBars(5,1):CalibBlackBars(6,1),CalibBlackBars(5,2):CalibBlackBars(6,2)) =
0;
im(:,CalibBlackBars(7,1):CalibBlackBars(8,1),CalibBlackBars(7,2):CalibBlackBars(8,2)) =
0;

end
```

A.24 **FcnInitCamParams.m**

```
function [ IP,CamRes,CamParam ] = FcnInitCamParams( FlagLive )

% This function initialize parameters for all the cameras.


% Save URLs for the cameras (also sets their resolutions)
IP = {'http://172.16.1.01/axis-cgi/jpg/image.cgi?resolution=480X360';
      'http://172.16.1.02/axis-cgi/jpg/image.cgi?resolution=480X360';
      'http://172.16.1.03/axis-cgi/jpg/image.cgi?resolution=480X360'};

if FlagLive == 1
    % Extract and package camera resolutions from URLs above
    IP1 = IP{1,1};
    CamRes  = [str2double(IP1(1,54:56)),str2double(IP1(1,58:60))];
else
    % Set a camera resolution for whne images are loaded from file
    CamRes = [480,360];
    %CamRes = [320,240];
end

% Load and organize camera calibraiton parameters
% --> kc,cc,fc are taken with the OpenCV camera calibration toolbox
% --> alpha_c is always zero

% For Camera 1 (sees the door)
alpha_c1 = 0; % Skew Coefficient
fc1 = [302.149012  ; 295.704801 ]; % Focal lengths for each axis in pixels
cc1 = [223.006212 ; 182.921981]; % Image center for each axis
kc1 = [-0.342700; 0.108096; 0.007426; 0.003782; 0.000000]; % Distortion matrix
Cam1 = struct('kc',kc1,'cc',cc1,'fc',fc1,'alpha_c',alpha_c1);

% For Camera 2 (sees the middle of the track)
alpha_c2 = 0; % Skew Coefficient
fc2 = [276.514980  ; 247.499571]; % Focal lengths for each axis
cc2 = [222.343345 ; 205.275207]; % Image center for each axis
kc2 = [-0.268546; 0.055665; -0.010588; 0.001720; 0.000000]; % Distortion matrix
Cam2 = struct('kc',kc2,'cc',cc2,'fc',fc2,'alpha_c',alpha_c2);

% For Camera 3 (sees the back wall)
alpha_c3 = 0; % Skew Coefficient
fc3 = [348.098814  ; 313.024017 ]; % Focal lengths for each axis
cc3 = [209.421260 ; 185.194388]; % Image center for each axis
kc3 = [-0.406560; 0.150852; 0.001216; 0.005513; 0.000000]; % Distortion matrix
Cam3 = struct('kc',kc3,'cc',cc3,'fc',fc3,'alpha_c',alpha_c3);

CamParam=struct('Cam1',Cam1,'Cam2',Cam2,'Cam3',Cam3);

end
```

A.25 **FcnInitDistortCorrection.m**

```
function [] = FcnInitDistortCorrection(CamParam,NumCams,CamRes)

% This function initializes the process of calculating or loading camera distortion
calibrations.


% Use a dialogue to ask whether the user wants to create new calibrations
choice = questdlg('Load last camera distortion corrections or create new ones?', ...
    'Camera Distortion Corrections', ...
    'Use Last','Create New','Create New');
% Handle response
switch choice
    case 'Create New'
```

```
        FcnInitDistortCorrection_Calib_Part1(CamParam,NumCams,CamRes); % generating table
        FcnInitDistortCorrection_Calib_Part2; % clean up holes and save

end

end
```

A.26 **FcnInitDistortCorrection_Calib_Part1.m**

```
function [] = FcnInitDistortCorrection_Calib_Part1(CamParam,NumCams,CamRes)

% This function conducts calibrations for camera distortion.

% Distorted images for each camera must be in the current directory and
% saved as "Calib_im_1" , "Calib_im_2" , etc.

commandwindow
disp('WARNING: This calibration takes a very long time to process (~20 hrs). Press any
key to run it anyway!');
pause;

% Create a place to store where indices move to:
im_distorted = imread(strcat('Calib_im_1.jpg'));
greyim_distorted = rgb2gray(im_distorted);
newlocation = zeros(numel(greyim_distorted),NumCams);

DistortionMapping = ones(length(newlocation),NumCams);
DistortionMappingSparse = zeros(size(DistortionMapping));

for CamNum = 1:NumCams

%     % Load the cameara calibration parameters
%     name = strcat('Cam',num2str(CamNum));
%     alpha_c = CamParam.(name).alpha_c;
%     fc = CamParam.(name).fc;
%     cc = CamParam.(name).cc;
%     kc = CamParam.(name).kc;
%     Cam = struct('kc',kc,'cc',cc,'fc',fc,'alpha_c',alpha_c);
%     KK = [fc(1) alpha_c*fc(1) cc(1);0 fc(2) cc(2) ; 0 0 1];

    % Open an image from that camera from file and make it greyscale
    im_distorted = imread(strcat('Calib_im_',num2str(CamNum),'.jpg'));
    greyim_distorted = rgb2gray(im_distorted);

    % Save the number of rows & columns in the original image
    [rows cols] = size(greyim_distorted);

    % Create a linear array of zeros... many rows, one column
    zerotemplate_distorted = zeros(numel(greyim_distorted),1);

    for i=1:length(zerotemplate_distorted)
        % Fill in one pixel with 255, leaving all others to be zeros.
        template_distorted = zerotemplate_distorted;
        template_distorted(i) = 255;

        % Convert back to an array
        matrixtemplate_distorted = reshape(template_distorted,rows,cols);

        % Correct distortion
        % matrixtemplate_undistorted =
uint8(FcnFixDistort_Rect(double(matrixtemplate_distorted),eye(3),fc,cc,kc,alpha_c,KK));
        matrixtemplate_undistorted =
uint8(FcnUndistort_Transform_Inputs(double(matrixtemplate_distorted),CamNum));

        if 1==1 % Change to 1 to see it working live... painfully slow
            % Plot the distorted and undistorted versions side by side
```

```
            figure(3)
            subplot(1,2,1)
            imshow(matrixtemplate_distorted)
            title('DISTORTED')
            subplot(1,2,2)
            imshow(matrixtemplate_undistorted)
            title('UNDISTORTED')
            xlabel(sprintf('%3.2f percent
complete',i/length(zerotemplate_distorted)*100));
            pause(0.01);
        end

        % Find maximum
        template_undistorted = reshape(matrixtemplate_undistorted,rows*cols,1);
        [~,max_ind] = max(template_undistorted);

        % Store resulting index, e.g. where the original pixel moved to
        newlocation(i,CamNum) = max_ind;

        % Print a percent completion
        fprintf('Stage 1, Camera %d, %3.2f percent complete
\n',CamNum,i/length(zerotemplate_distorted)*100)

    end
    fprintf('100.00 percent complete\n')



    % Now flip the mapping (could do this in the code above, but forgot and don't want to
re-run it!)
    for i=1:length(newlocation)
        DistortionMapping(newlocation(i,CamNum),CamNum) = i;
    end

    % Save result, because it illustrates where interpolation is necessary
    DistortionMappingSparse(:,CamNum) = DistortionMapping(:,CamNum);

end

save('DistortionMappingSparse.mat') % checkpoint for debugging purposes (table generation
takes a long time)
```

A.27 **FcnInitDistortCorrection_Calib_Part2.m**

```
function [] = FcnInitDistortCorrection_Calib_Part2()

load('DistortionMappingSparse.mat')
for CamNum = 1:NumCams
    %% Now, fix locations where mapping is sparse
    for i=1:length(DistortionMappingSparse)
        if 1==DistortionMappingSparse(i,CamNum)

            % Identify the pixel values that are adjacent to an empty pixel
            % Uncomment the one below if need to do corners as well
            neighbors = [i-rows-1, i-rows, i-rows+1, i-1, i+1, i+rows-1, i+rows,
i+rows+1];

            % Grab adjacent rows
            % neighbors = [i-rows, i-1, i+1, i+rows];

            % Make sure they are valid neighbors , e.g. they are not hanging over edge of
image
            good_neighbors = neighbors(neighbors>0);
            good_neighbors = good_neighbors(good_neighbors<(rows*cols+1));

            % Make sure the map isn't = 1 at these locations
```

```
            indices_to_chose_from =
good_neighbors(DistortionMappingSparse(good_neighbors,CamNum)>1);

            % Pick one at random and assign the gap to this neighbor
            value = ceil(rand*length(indices_to_chose_from));
            if value > 0
            DistortionMappingSparse(i,CamNum) =
DistortionMappingSparse(indices_to_chose_from(value),CamNum);
            end
        end
    end

DistortionMapping(:,CamNum) = DistortionMappingSparse(:,CamNum);
    %% Now fix missing locations in newlocation matrix

    % First, save sparse version of newlocation
    newlocationSparse = newlocation;

    % Fill in some arrays
    good_values = find(newlocationSparse(:,CamNum)>1);
    [good_rows, good_cols] = ind2sub(size(greyim_distorted),good_values);

    % Define the pixel we are looking for (I do an entire column to illustrate
    % situations where the pixel is found AND not found)
    count = 0;
    for row = 1:rows
        for col = 1:cols
            count = count+1;
            % First, find the indices of the point inside the distorted image
            linearInd = sub2ind(size(greyim_distorted),row,col);

            % Below is unnecessary. I used to need it before I fixed the
            % newlocation array to point to nearest term
            if 1==1
                is_good = find(good_values==linearInd); % gives a number if it is good

                % If you don't find the pixel, we have to search for nearby ones.
                if isempty(is_good) % Pixel wasn't found!
                    % Find distances from this row/col to all good rows/cols
                    distances = (good_rows - row).^2 + (good_cols - col).^2;

                    % Take minimum... keep only the index of the minimum
                    [junk,min_i] = min(distances);

                    % Assign this good index to replace the bad index value
                    newlocation(linearInd,CamNum) =
newlocation(good_values(min_i),CamNum);
                end
            end
            fprintf('Stage 2, Camera %d, %0.2f percent complete
\n',CamNum,100*count/(rows*cols));
        end
    end

    %% Save data from calibration
    name = strcat('Cam_',num2str(CamNum));
%     info_newlocation.(name).Cam = Cam;
%     info_newlocation.(name).fc = fc;
%     info_newlocation.(name).alpha_c = alpha_c;
%     info_newlocation.(name).cc = cc;
%     info_newlocation.(name).kc = kc;
%     info_newlocation.(name).KK = KK;
    info_newlocation.(name).rows = rows;
    info_newlocation.(name).cols = cols;

    save DataCalibCamDistort.mat newlocationSparse newlocation DistortionMappingSparse
DistortionMapping info_newlocation

    disp(strcat('Cam ',CamNum,' Complete!'))
```

```
end
```

A.28 **FcnInitDistTrack.m**

```
function [] = FcnInitDistTrack( IP,FlagLive,TimeStamps,Iter,NumCams,CamRes )

% This function initializes the process of calculating or loading distance tracking
calibrations.


% Use a dialogue to ask whether the user wants to create new calibrations
choice = questdlg('Load last distance tracking calibrations or create new ones?', ...
    'Distance Tracking Calibrations', ...
    'Use Last','Create New','Create New');
% Handle response
switch choice
    case 'Create New'
        CalibDistTrack =
FcnInitDistTrack_Calib(IP,FlagLive,TimeStamps,Iter,NumCams,CamRes); % Take new
calibrations
        save('DataCalibDistTrack.mat','CalibDistTrack')
end

end
```

A.29 **FcnInitDistTrack_Calib.m**

```
function [ CalibDistTrack ] = FcnInitDistTrack_Calib(
IP,FlagLive,TimeStamps,Iter,NumCams,CamRes )

% This function allows the user to conduct calibrations for distance tracking.

% Load images from each camera, one at at time
for CamNum= 1:3
    calibcheck = 'n';
    while calibcheck ~= 'y'
        % Read in a camera image
        im = FcnGetImage( IP,FlagLive,TimeStamps,Iter,CamNum );
        % Undistort the image
        load DataCalibCamDistort.mat
        % Flip the image segment back to how it was originally
%         switch CamNum
%             case 1
%                 im = imrotate(im,90);
%             case 2
%                 im = imrotate(im,90);
%             case 3
%                 im = imrotate(im,90);
%             case 4
%                 im = imrotate(im,90);
%         end
        for Dimension = 1:3
            imlayer = im(:,:,Dimension);
            switch CamNum
                case 1
                I = imrotate(imlayer,90);
                XPixRight = 9;
                YPixDown = -20;
                RotDegCCW = 0.5;
                K = -0.4;
                input_points =   [121.1789  318.0222
                  120.2525   99.8560
                  354.6307  320.3382
                  361.1155   97.5400];
```

```
                base_points =  [120.7157  318.4854
                  120.7157  100.7824
                  360.6523  319.4118
                  361.1155   98.9296];
                imlayer =
FcnUndistort_Transform_Ind(I,XPixRight,YPixDown,RotDegCCW,K,input_points,base_points);
                imlayer = imrotate(imlayer,-90);
                case 2
                I = imrotate(imlayer,90);
                XPixRight = 16;
                YPixDown = 0;
                RotDegCCW = 1;
                K = -0.4;
                input_points =  [120.5209  302.0057
                  125.6589   75.9348
                  371.3475  301.0715
                  365.2753   72.1981];
                base_points =  [126.5930  302.0057
                  126.1259   76.4019
                  364.8082  300.6044
                  364.8082   76.4019];
                imlayer =
FcnUndistort_Transform_Ind(I,XPixRight,YPixDown,RotDegCCW,K,input_points,base_points);


                imlayer = imrotate(imlayer,-90);
                case 3
                I = imrotate(imlayer,90);
                XPixRight = 22;
                YPixDown = 0;
                RotDegCCW = -.5;
                K = -0.3;
                input_points = [96.1662  299.0311
                   99.8718   32.2291
                  381.9594   32.2291
                  391.6865  299.9575];
                base_points = [96.6294  298.1047
                   97.0926   31.7659
                  393.0761   31.7659
                  394.0025  298.1047];
                imlayer =
FcnUndistort_Transform_Ind(I,XPixRight,YPixDown,RotDegCCW,K,input_points,base_points);
                imlayer = imrotate(imlayer,-90);
            end
            %imlayer = reshape(imlayer(DistortionMapping(:,CamNum)),CamRes(2),CamRes(1));
            im(:,:,Dimension) = imlayer;
        end
        % Re-rotate the image segment
        % Note: Image rotation dependant on camera setup!
%         switch CamNum
%             case 1
%                 im = imrotate(im,-90);
%             case 2
%                 im = imrotate(im,-90);
%             case 3
%                 im = imrotate(im,-90);
%             case 4
%                 im = imrotate(im,-90);
%         end

        commandwindow
        % Get two vertical points
        fprintf('Select two points in a vertical line.\n')
        [pointPX_Vert,pointFT_Vert] = FcnInitDistTrack_Get2Pts(im);

        commandwindow
        % Get two horizontal points
        fprintf('Select two points in a horizontal line.\n')
        [pointPX_Horiz,pointFT_Horiz] = FcnInitDistTrack_Get2Pts(im);
```

```
        % Get a scale factor of pixels/feet
        FTperPX_Vert  = ( pointFT_Vert(1,2)-pointFT_Vert(2,2)   ) / ( pointPX_Vert(1,2)-
pointPX_Vert(2,2)   );
        FTperPX_Horiz = ( pointFT_Horiz(2,1)-pointFT_Horiz(1,1) ) / ( pointPX_Horiz(2,1)-
pointPX_Horiz(1,1) );

        % Get the length of the longest axis
        Length=length(im);

        % Get the size of the image
        Res_Vert  = size(im,1);
        Res_Horiz = size(im,2);

        % Initialize 1D arrays in which to store real-world pixel locations
        FT_Vert  = zeros(Length,1);
        FT_Horiz = zeros(Length,1);

        % Initialize 'CalibDistTrack' on the first iteration
        if CamNum == 1
            CalibDistTrack=zeros(Length,2,NumCams);
        end

        % Calculate the real-world location of every vertical pixel
        for Res = 1:Res_Vert
            FT_Vert(Res)  = pointFT_Vert(1,2)  + FTperPX_Vert  * (Res - pointPX_Vert(1,2)
);
        end

        % Calculate the real-world location of every horizontal pixel
        for Res = 1:Res_Horiz
            FT_Horiz(Res) = pointFT_Horiz(1,1) + FTperPX_Horiz * (Res -
pointPX_Horiz(1,1) );
        end

        figure(1)

        % Set the axes so that the text about to be plotted will be visibile
        axis([-200 Res_Horiz+200 -200 Res_Vert+200])

        % Identify some pixel locations at which to plot the calibrated real-world points
        PointsToPlot =
[1,1;1,Res_Vert;Res_Horiz,1;Res_Horiz,Res_Vert;round(Res_Horiz/2),round(Res_Vert/2)];

        % Plot and label the real-world points on the image
        for n=1:size(PointsToPlot,1)

            plot( PointsToPlot(n,1),PointsToPlot(n,2),'black.-','markersize', 30 );
            plot( PointsToPlot(n,1),PointsToPlot(n,2),'red+','markersize', 10 );
            text(PointsToPlot(n,1), PointsToPlot(n,2),horzcat(...
                '  ',num2str(FT_Horiz(PointsToPlot(n,1))),' , ',...
                '  ',num2str(FT_Vert (PointsToPlot(n,2)))), 'FontSize',18);
        end

        % Verify with the user that the calibration for this camera is okay
        commandwindow
        calibcheck = input('Calibration okay (y/n)?  ', 's');

        if calibcheck ~= 'y'
            fprintf('Restarting calibration for this camera...\n')
        end

        % Clear the command window and close the figure
        close(1)
    end

    clc

    %Package and return CalibDistTrack
```

```
    CalibDistTrack(:,:,CamNum)= [FT_Horiz,FT_Vert];
end
```

A.30 **FcnInitDistTrack_Get2Pts.m**

```
function [ pointPX,pointFT ] = FcnInitDistTrack_Get2Pts( im )

% This function allows the user to select two locations on the image and
% enter their real-world locations.


% Prepare the figure
figure(1)
clf(1)
imagesc(im)
hold on
axis tight

for PointNum=1:2

    pointcheck = 'n';

    while pointcheck ~= 'y'

        figure(1)

        % Have user input a point
        pointPX(PointNum,:) = ginput(1);

        % Show the point on the figure
        h(1) = plot( pointPX(PointNum,1),pointPX(PointNum,2),'black.-','markersize', 30
);
        h(2) = plot( pointPX(PointNum,1),pointPX(PointNum,2),'red+','markersize', 10 );

        commandwindow

        % Verify that the point is okay
        pointcheck = input('Point okay (y/n)?  ', 's');

        if pointcheck ~= 'y'
            delete(h(1));
            delete(h(2));
            fprintf('Select a new point.\n')
        end
    end

    % Obtain and store the real-world point locations
    pointX = str2num( input('Enter X location (ft):  ', 's') );
    pointY = str2num( input('Enter Y location (ft):  ', 's') );
    pointFT(PointNum,:)=[pointX,pointY];

end

delete(h(1));
delete(h(2));

end
```

A.31 **FcnInitEndzones.m**

```
function [] = FcnInitEndzones( IP,CamRes,FlagLive,TimeStamps,Iter,NumCams )

% This function initializes the process of calculating or loading endzone calibrations.
```

```
% Use a dialogue to ask whether the user wants to create new calibrations
choice = questdlg('Load last endzone calibrations or create new ones?', ...
    'Endzone Calibrations', ...
    'Use Last','Create New','Create New');
% Handle response
switch choice
    case 'Create New'
        CalibEndzones =
FcnInitEndzones_Calib(IP,CamRes,FlagLive,TimeStamps,Iter,NumCams); % Take new
calibrations
        save('DataCalibEndzones.mat','CalibEndzones')
end

end
```

A.32 **FcnInitEndzones_Calib.m**

```
function [ CalibEndzones ] = FcnInitEndzones_Calib(
IP,CamRes,FlagLive,TimeStamps,Iter,NumCams )

% This function allows the user to conduct calibrations for lap tracking.


% Get an image for each camera
im = FcnGetImage_All( IP,CamRes,FlagLive,TimeStamps,Iter,NumCams);

% Undistort the images
load DataCalibCamDistort.mat
im = FcnUndistort(im,DistortionMapping,NumCams,CamRes);

figure (1)
imshow(im)

hold on;

Lpoints = zeros(2,2);
Rpoints = zeros(2,2);

% Draw a line between two points on the screen selected by the user
% (store this as the left endzone for now)
for i = 1:2
    Lpoints(i,:) = ginput(1);
    plot(Lpoints(1:i,1),Lpoints(1:i,2),'b-')
    drawnow
end

% Draw the slope and intercept for this endzone
Lm = ( Lpoints(2,1) - Lpoints(1,1) ) / ( Lpoints(2,2) - Lpoints(1,2) );
Lb = Lpoints(1,1) - Lm*Lpoints(1,2);

% Superimpose this endzone on the image
for x=1:size(im,1);
    plot(Lm*x+Lb,x)
end

% Connect a line between two points on the screen selected by the user
% (store this as the right endzone for now)
for i = 1:2
    Rpoints(i,:) = ginput(1);
    plot(Rpoints(1:i,1),Rpoints(1:i,2),'r-')
    drawnow
end

% Calculate the slope and intercept for this endzone
Rm = ( Rpoints(2,1) - Rpoints(1,1) ) / ( Rpoints(2,2) - Rpoints(1,2) );
Rb = Rpoints(1,1) - Rm*Rpoints(1,2);
```

```
% Superimpose this endzone on the image
for x=1:size(im,1);
    plot(Rm*x+Rb,x,'r-')
end


% Ensure that the left and right endzones are actually located on the left and right
respectively

yL = 250*Lm + Lb; % Calculate the y value of the LEFT endzone line for an x value of 250
yR = 250*Rm + Rb; % Calculate the y value of the RIGHT endzone line for an x value of 250

% Compare the y values
if yR > yL % Endzones are are correct
    CalibEndzones=[ Lm,Lb;Rm,Rb ];
else % Endzones are switched, so reverse them when forming the matrix
    CalibEndzones=[ Rm,Rb;Lm,Lb ];
end

close(1)

% For debugging the above code - allows you to observe the endzones being switched

% Lm=CalibEndzones(1,1);
% Lb=CalibEndzones(1,2);
% Rm=CalibEndzones(2,1);
% Rb=CalibEndzones(2,2);
%
% hold off
% imagesc(im);
% hold on
%
% for x=1:size(im,1);
%     plot(Lm*x+Lb,x,'b-')
% end
%
% for x=1:size(im,1);
%     plot(Rm*x+Rb,x,'r-')
% end

end
```

A.33 **FcnInitTestConditions.m**

```
function [ FlagLive,FlagPlot,FlagSavePlot,NumCams,Data2File ] = FcnInitTestConditions

% This function initializes all necessary variables for the lap counting and distance
tracking that users may need to change.

% Flag for image collection
% --> 1 to collect data in real-time
% --> 0 to load images from file (the default basenames are: im_ 1_, im_ 2_ and im_ 3)
% REMEMBER THAT YOU HAVE TO MANUALLY SAVE DATA FOR THE LAST int (Iter/Dat2File)
% ITERATIONS WHEN RUNNING LIVE
FlagLive = 0;

% Flag for continuously plotting the camera images
% --> 1 to plot (better for debugging)
% --> 0 to NOT plot (runs faster)
FlagPlot = 1;

% Flag for saving the plots of camera images
% --> 1 to save
% --> 0 to NOT save (runs faster)
FlagSavePlot = 0;
```

```
% Number of cameras to be used for data collection
NumCams = 3;

% Script saves data to file every 'Data2File' iterations if we are running live
% IF we are not running live, it saves data to file once all images from file have been
processed
Data2File = 1000000;

% NOTE: To crop out overlap in the images:
% If FlagPlot is 1, use FcnUndistort
% If FlagPLot is 0, use FcnGetImage_select

end
```

A.34 **FcnInitVars.m**

```
function [
Iter,CentroidFT_Last,CentroidPX_Last,CentroidPX_Last_Raw,TotalLaps,TotalDist,LastZone,Fla
gObjFound,DataLog,TimeStamps] = FcnInitVars( Data2File,FlagLive )

% This function initializes all necessary variables for the main script that users don't
need to change.

Iter = 0;                % Counter for the number of iterations
CentroidPX_Last = [0,0]; % Pixel location of the fiducial at previous iteration
CentroidFT_Last = [0,0]; % Real-world location of the fiducial at previous iteration
CentroidPX_Last_Raw = [0,0]; %This is used in a trehshold against noise when FlagPlot = 0
TotalLaps = 0;           % Number of laps completed
TotalDist = 0;           % Distance traveled

% Last endzone the fiducial was in
LastZone = 0;
% --> 0 before the object ever enters an endzone
% --> 1 if the object was last in the right endzone
% --> 2 if the object was last in the left endzone

% Variable for whether we know where the fiducial is
FlagObjFound = 0;
% --> 1 if we know where the object is
% --> 0 if we don't know where the object is

% Variables for storing data
DataLog = zeros(Data2File,8);

% If loading images from file, extract timestamps from the filenames
if FlagLive == 0
    TimeStamps = FcnGetTimestamps;
    %TimeStamps_Ref = FcnGetTimestamps_Ref;
else
    TimeStamps = 0;
    %TimeStamps_Ref = 0;
end

end
```

A.35 **FcnLensDistort.m**

```
function I2 = FcnLensDistort(I, k, varargin)
%LENSDISTORT corrects for barrel and pincusion lens abberations
%   I = LENSDISTORT(I, k)corrects for radially symmetric distortions, where
%   I is the input image and k is the distortion parameter. lens distortion
%   can be one of two types: barrel distortion and pincushion distortion.
%   In "barrel distortion", image magnification decreases with
```

```
%   distance from the optical axis. The apparent effect is that of an image
%   which has been mapped around a sphere (or barrel). In "pincushion
%   distortion", image magnification increases with the distance from the
%   optical axis. The visible effect is that lines that do not go through the
%   centre of the image are bowed inwards, towards the centre of the image,
%   like a pincushion [1].
%
%   I = LENSDISTORT(...,PARAM1,VAL1,PARAM2,VAL2,...) creates a new image image,
%   specifying parameters and corresponding values that control various aspects
%   of the image distortion correction. Parameter names case does not matter.
%
%   Parameters include:
%
%   'bordertype'            String that controls the treatment of the image
%                           edges. Valid strings are 'fit' and 'crop'. By
%                           default, 'bordertype' is set to 'crop'.
%
%   'interpolation'         String that specifies the interpolating kernel
%                           that the separable resampler uses. Valid
%                           strings are 'cubic', 'linear' and 'nearest'. By
%                           default, the 'interpolation' is set to 'cubic'
%
%   'padmethod'             string that controls how the resampler
%                           interpolates or assigns values to output elements
%                           that map close to or outside the edge of the input
%                           array. Valid strings are 'bound', circular',
%                           'fill', 'replicate', and symmetric'. By
%                           default, the 'padmethod' is set to 'fill'
%
%   'ftype'                 Integer between 1 and 4 that specifies the
%                           distortion model to be used. The models
%                           available are
%
%                           'ftype' = 1:    s = r.*(1./(1+k.*r));
%
%                           'ftype' = 2:    s = r.*(1./(1+k.*(r.^2)));
%
%                           'ftype' = 3:    s = r.*(1+k.*r);
%
%                           'ftype' = 4:    s = r.*(1+k.*(r.^2));
%
%                           By default, the 'ftype' is set to 4.
%
%   Class Support
%   -------------
%   An input intensity image can be uint8, int8, uint16, int16, uint32,
%   int32, single, double, or logical. An input indexed image can be uint8,
%   uint16, single, double, or logical.
%
%   Examples
%   --------
%       % read image
%       I = imread('cameraman.tif');
%
%       % Distort Image
%       I2 = lensdistort(I, 0.1);
%
%       % Display both images
%       imshow(I), figure, imshow(I2)
%
%   References
%   --------------
%   [1] http://en.wikipedia.org/wiki/Distortion_(optics), August 2012.
%
%   [2] Harri Ojanen, "Automatic Correction of Lens Distortion by Using
%       Digital Image Processing," July 10, 1999.
%
%   [3] G.Vassy and T.Perlaki, "Applying and removing lens distortion in post
%       production," year???
```

```
%
%   [4] http://www.mathworks.com/products/demos/image/...
%       create_gallery/tform.html#34594, August 2012.
%
%   Created by Jaap de Vries, 8/31/2012
%   jpdvrs@yahoo.com
%
%-------------------------------------------------------------------------%

%-------------------------------------------------------------------------
% This part of the codes creates variable input parameters using the input
% parser object
p = inputParser;
%   Make input string case independant
p.CaseSensitive = false;

%   Specifies the required inputs
addRequired(p,'I',@isnumeric);
addRequired(p,'k',@isnumeric);

%   Sets the default values for the optional parameters
defaultFtype = 4;
defaultBorder = 'crop';
defaultInterpolation = 'cubic';
defaultPadmethod = 'fill';

%   Specifies valid strings for the optional parameters
validBorder = {'fit','crop'};
validInterpolation = {'cubic','linear', 'nearest'};
validPadmethod = {'bound','circular', 'fill', 'replicate', 'symmetric'};

%   Funtion handles to determine wheter a proper input string has been used
checkBorder = @(x) any(validatestring(x,validBorder));
checkInterpolation = @(x) any(validatestring(x,validInterpolation));
checkPadmethod = @(x) any(validatestring(x,validPadmethod));

%   Create optional inputs
addParamValue(p,'bordertype',defaultBorder,checkBorder);
addParamValue(p,'interpolation',defaultInterpolation,checkInterpolation);
addParamValue(p,'padmethod',defaultPadmethod,checkPadmethod);
addParamValue(p,'ftype',defaultFtype,@isnumeric);

%   Pass all parameters and input to the parse method
parse(p,I,k,varargin{:});

%-------------------------------------------------------------------------
% This determines wether its a color (M,N,3) or gray scale (M,N,1) image
if ndims(I) == 3
     for i=1:3
        I2(:,:,i) = imdistcorrect(I(:,:,i),k);
     end
elseif ismatrix(I)
    I2 = imdistcorrect(I,k);
else
    error('Unknown image dimensions')
end

%-------------------------------------------------------------------------
% Nested function that perfoms the transformation
    function I3 = imdistcorrect(I,k)
    % Determine the size of the image to be distorted
    [M N]=size(I);
    center = [round(N/2) round(M/2)];
    % Creates N x M (#pixels) x-y points
    [xi,yi] = meshgrid(1:N,1:M);
    % Creates converst the mesh into a colum vector of coordiantes relative to
    % the center
    xt = xi(:) - center(1);
    yt = yi(:) - center(2);
```

```
    % Converts the x-y coordinates to polar coordinates
    [theta,r] = cart2pol(xt,yt);
    % Calculate the maximum vector (image center to image corner) to be used
    % for normalization
    R = sqrt(center(1)^2 + center(2)^2);
    % Normalize the polar coordinate r to range between 0 and 1
    r = r/R;
    % Aply the r-based transformation
    s = distortfun(r,k,p.Results.ftype);
    % un-normalize s
    s2 = s * R;
    % Find a scaling parameter based on selected border type
    brcor = bordercorrect(r,s,k, center, R);

    s2 = s2 * brcor;


    % Convert back to cartesian coordinates
    [ut,vt] = pol2cart(theta,s2);

    u = reshape(ut,size(xi)) + center(1);
    v = reshape(vt,size(yi)) + center(2);
    tmap_B = cat(3,u,v);
    resamp = makeresampler(p.Results.interpolation, p.Results.padmethod);
    I3 = tformarray(I,[],resamp,[2 1],[1 2],[],tmap_B,255);
    end

%--------------------------------------------------------------------------
% Nested function that creates a scaling parameter based on the
% 'bordertype' selected
    function x = bordercorrect(r,s,k,center, R)
        if k < 0
            if strcmp(p.Results.bordertype, 'fit')
               x = r(1)/s(1);
            end
            if strcmp(p.Results.bordertype,'crop')
               x = 1/(1 + k*(min(center)/R)^2);
            end
        elseif k > 0
            if strcmp(p.Results.bordertype, 'fit')
               x = 1/(1 + k*(min(center)/R)^2);
            end
            if strcmp(p.Results.bordertype, 'crop')
               x = r(1)/s(1);
            end
        end
    end

%--------------------------------------------------------------------------
% Nested function that pics the model type to be used
    function s = distortfun(r,k,fcnum)
        switch fcnum
        case(1)
            s = r.*(1./(1+k.*r));
        case(2)
            s = r.*(1./(1+k.*(r.^2)));
        case(3)
            s = r.*(1+k.*r);
        case(4)
            s = r.*(1+k.*(r.^2));
        end
    end


end
```

A.36 **FcnLogData.m**

```
function [DataLog] = FcnLogData( Iter, FlagLive ,TotalLaps, TotalDist, TotalTime,
CentroidPX_Current, CentroidFT_Current, DataLog, Data2File,TimeStampLength )

% This function saves data to 'DataLog' and saves data to file every 'Data2File'
iterations.


% Get the number of iterations since the last time 'DataLog' was saved to file
Line = rem(Iter,Data2File);

if Line == 0
    Line = Data2File;
end

% Update 'DataLog'
DataLog(Line,:) = [
Iter,CentroidPX_Current,CentroidFT_Current,TotalTime,TotalLaps,TotalDist ];

% If 'DataLog' is full and we are obtaining images live
if FlagLive == 1 && Line == Data2File

    % Save 'DataLog' to file with a unique postscript
    NameExtension = num2str(Iter/Data2File);
    Name=strcat( 'DataLog_',NameExtension,'.mat' );
    save( Name,'DataLog' );

    % Empty the 'DataLog' matrix so we can start filling it all over again
    DataLog = zeros(Data2File,8);
end

% If we are obtaining images from file and have reached the last one
if FlagLive == 0 && Iter == TimeStampLength

    % Save 'DataLog' to file with a unique postscript
    Name=strcat( 'DataLog','.mat' );
    save( Name,'DataLog' );

end

    % Also create one big file with all the data in one, named 'DataLog_Continuous'
    myformat = '%7d %4.4f %4.4f %3.2f %3.2 %10.2f %5d %10.2f\n';
    fid = fopen('DataLog_Continuous.txt','a');
    fprintf(fid,
myformat,[Iter,CentroidPX_Current,CentroidFT_Current,TotalTime,TotalLaps,TotalDist]);
    fclose(fid);
```

A.37 **FcnMask.m**

```
function [ Mask,UpdatedFlagObjFound,CentroidPX_Current ] = FcnMask( im,CentroidPX_Current
)

% This function makes a mask based on color and object size in LAB space for a pink
fiducial.


% Define the minimum pixel area of the fiducial expected
MinSize = 30;  % Default around 180 for green disk fiducial
               % Default around 50 for LED fiducial

% Make a mask based on color only
Mask = FcnMask_Color( im );

% Filter out small objects
Mask = bwareaopen( Mask,MinSize );
```

```
% Smooth the border using a morphological closing operation
structuringElement = strel( 'disk', 4 );
Mask = imclose( Mask, structuringElement );

% Fill in holes
Mask = uint8( imfill(Mask, 'holes') );

% Get region properties for all components
CC = bwconncomp(Mask);
props = regionprops( CC,Mask,'Area','Centroid','Eccentricity' );

% Use the below for debugging the mask
%figure(3)
%imagesc(Mask)

% Find the fiducial out of the existing components
for n=1:size(props,1)
    if props(n).Eccentricity < 0.95 && props(n).Area < 10000
        ObjectIndex = n; %store the index of the object we want

        ObjectArea = props(n).Area;

        % Use the below for debugging the mask
        %disp(props(n).Eccentricity)
        %disp(props(n).Area)

        % Update FlagObjFound since we know where the object is
        UpdatedFlagObjFound = 1;
    end
end

% If the object was not found
if exist('ObjectIndex','var') == 0
        UpdatedFlagObjFound = 0;
        return
end

% Remove objects smaller than the size of the largest object
Mask = bwareaopen(Mask,ObjectArea-1);

% Store the location of centroid
CentroidPX_Current(1) = props(ObjectIndex).Centroid(1);
CentroidPX_Current(2) = props(ObjectIndex).Centroid(2);

% Overlay the mask - useful for debugging
%mask = cast(mask, class(im));
%maskr = mask.*im(:,:,1);
%maskg = mask.*im(:,:,2);
%maskb = mask.*im(:,:,3);
%maskedim = cat(3,maskr,maskg,maskb);

% Plot the mask
%imagesc(maskedim);

end
```

A.38 **FcnMask_Color.m**

```
function Mask = FcnMask_Color(im)
% This function creates a mask used to find for a green fiducial

% Background subtraction
im2 = abs(im);
% Fiducial mask
% im3 = rgb2hsv(im2);
```

```
% Extract the highest values from the second dimension
I = im2(:,:,3);
%I = im2(:,:,1) - im2(:,:,2) - im2(:,:,3);
% Add contrast
%I = imadjust(I,[.07 .36],[]);
% Normalize
I = double(I);
I = I/max(max(I));
% Apply threshold
Mask = I>.6;
end
```

A.39 **FcnPathDev.m**

```
function [ C, Ridge, PathDev, DataLogInterp ] = FcnPathDev( DataLog )

% This script plots a 3D histogram of robot path data.
% The histogram is then used to determine the most common path.
% Deviation from the optimum path for every position is calculated.

%% Fill out the DataLog position data through interpolation (make function?)

InterSize = 10; % How many points to interpolate between position entries

DataLogInterp = [];
DataLogInterp(1,:) = DataLog(1,:);
for i = 2:length(DataLog)
    DataLogInterp(InterSize*(i-1)+1,:) = DataLog(i,:);

    for column = 2:8
    % Generating linear interpolation
    Proto = linspace(DataLog(i-1,column),DataLog(i,column),InterSize+1);
    Proto = Proto(2:end); % taking off first (from linspace)
    DataLogInterp(InterSize*(i-1)-8:InterSize*(i-1)+1,column) = Proto;
    end

end

%% Bounding box for histogram

X_Pos = DataLogInterp(:,4);
Y_Pos = DataLogInterp(:,5);

Min_X = min(X_Pos);
Max_X = max(X_Pos);
Min_Y = min(Y_Pos);
Max_Y = max(Y_Pos);

Min_X = floor(Min_X) - 2;
Max_X = ceil(Max_X) + 2;
Min_Y = floor(Min_Y) - 2;
Max_Y = ceil(Max_Y) + 2;

% right now, steps should be divisors of feet
step_x = 1/12; % width of each bin in x dim (ft)
step_y = 1/12; % width of each bin in y dim (ft)

X_Grid = [Min_X:step_x:Max_X];
Y_Grid = [Min_Y:step_y:Max_Y];

% Edges for histogram
CTRS{1} = X_Grid(1:end-1) + step_x/2;
CTRS{2} = Y_Grid(1:end-1) + step_y/2;

Coord = [X_Pos,Y_Pos];
```

```
% Initialize sparse coordinates and bounding box
Coord_Sparse(1,:) = Coord(1,:);
X_Low = X_Grid(max(find(X_Grid<Coord(1,1))));
X_High = X_Grid(min(find(X_Grid>Coord(1,1))));
Y_Low = Y_Grid(max(find(Y_Grid<Coord(1,2))));
Y_High = Y_Grid(min(find(Y_Grid>Coord(1,2))));

% Loop time
% Iterate every position coordinate
count = 2;
for i = 2:length(Coord)
    % If robot has left box, new entry stored, otherwise entry forgotten
    if Coord(i,1) < X_Low || Coord(i,1) >= X_High || Coord(i,2) < Y_Low || Coord(i,2) >=
Y_High
        Coord_Sparse(count,:) = Coord(i,:);
        count = count + 1;
        % Defining new bounding box
        X_Low = X_Grid(max(find(X_Grid<Coord(i,1))));
        X_High = X_Grid(min(find(X_Grid>Coord(i,1))));
        Y_Low = Y_Grid(max(find(Y_Grid<Coord(i,2))));
        Y_High = Y_Grid(min(find(Y_Grid>Coord(i,2))));
    end
end

figure
hist3(Coord_Sparse,CTRS)
xlabel('x-axis (ft)')
ylabel('y-axis (ft)')
set(gcf,'renderer','opengl'); % colors histogram by magnitude
set(get(gca,'child'),'FaceColor','interp','CDataMode','auto');
%axis equal

%% Surface plot
[N,C] = hist3(Coord_Sparse,CTRS);
SurfLaps = N';
% figure
% surf(C{1},C{2},SurfLaps)
% xlabel('x-axis (ft)')
% ylabel('y-axis (ft)')

% Watershed (no segmentation)
% figure
L = watershed(SurfLaps);
% extracting the ridgeline
SurfMax = max(max(SurfLaps));
H = 1.0*SurfMax*double(~L);
% mesh(C{1},C{2},H)
% hidden('on')
% shading('interp')

%% Segmentation
% Imaging processing tutorial found at this location:
% http://www.mathworks.com/help/images/examples/marker-controlled-watershed-
segmentation.html

% Normalizing surface plot (treat as grayscale image)
SurfNorm = SurfLaps/SurfMax;
I = flipud(SurfNorm);
%figure,imshow(I,'Border','loose','InitialMagnification',1000)

% Blurring image
GausFilter = fspecial('gaussian',[3 3], 1);
IBlur = imfilter(I, GausFilter, 'replicate');
% Blur again
IBlur2 = imfilter(IBlur, GausFilter, 'replicate');
%figure,imshow(IBlur2,'Border','loose','InitialMagnification',1000)

% Opening and Closing
```

```
se = strel('disk',3); % VERY IMPORTANT TO CHANGE FOR RESOLUTION
se2 = strel('disk',10); % VERY IMPORTANT TO CHANGE FOR RESOLUTION
% Opening
IOpen = imopen(IBlur2, se);
%figure, imshow(IOpen,'InitialMagnification',1000)
% Closing
IClose = imclose(IOpen, se2);
%figure, imshow(IClose,'InitialMagnification',1000)

% Contrast
ICon = imadjust(IClose,[.07 .36],[]);
%figure,imshow(ICon,'Border','loose','InitialMagnification',1000)


%% Watershed (Segmentation)
W = watershed(flipud(ICon));
figure
hold on
surf(C{1},C{2},SurfLaps)
xlabel('x-axis (ft)')
ylabel('y-axis (ft)')
Ridge = 1.5*SurfMax*double(~W);
surf(C{1},C{2},Ridge)
hidden('on')

%% Deviation
% For every iteration of test, calculates shortest distance from current
% position to ridgeline (most common path)

PosXYZ = [DataLog(:,4),DataLog(:,5),ones(length(DataLog),1)*max(max(Ridge))];
[Xmesh,Ymesh] = meshgrid(C{1},C{2});
RidgeXYZ = [Xmesh(:),Ymesh(:),Ridge(:)];

[Indices,PathDev] = dsearchn(RidgeXYZ,PosXYZ);

end
```

A.40 **FcnPlot.m**

```
function [] = FcnPlot(
im,mask,CalibEndzones,CentroidPX_Current,CentroidFT_Current,Xres,Yres,FlagObjFound )

% This function plots the image from the cameras and highlights the
% location of the fiducial by enclosing it with a green line and placing a
% crosshair at the centroid.  It also shows the locations of the left and
% right endzones.

% Extract endzone slope and intercept
Lm = CalibEndzones(1,1);
Lb = CalibEndzones(1,2);
Rm = CalibEndzones(2,1);
Rb = CalibEndzones(2,2);

% Calculate some points for plotting
X = 1:Yres;
Lbound = Lm*X+Lb;
Rbound = Rm*X+Rb;

% Show the image
figure(2)
imshow(im);
hold on

% If the object was found
if FlagObjFound == 1
```

```
% Plot the boundary of the object
Boundaries = bwboundaries(mask);
NumberOfBoundaries = size(Boundaries);
for k = 1 : NumberOfBoundaries
    ThisBoundary = Boundaries{k};
    plot( ThisBoundary(:,2), ThisBoundary(:,1), 'y', 'LineWidth', 4 );
end

% Place a crosshair on the centroid of the object
plot( CentroidPX_Current(1),CentroidPX_Current(2),'k.-','markersize', 30 );
plot( CentroidPX_Current(1),CentroidPX_Current(2),'r+','markersize', 10 );

% Display the fiducial location in ft
text(CentroidPX_Current(1)+40, CentroidPX_Current(2),horzcat(...
    '  ',num2str(CentroidFT_Current(1)),2,' , ',...
    '  ',num2str(CentroidFT_Current(2)),2),'FontSize',14,'BackgroundColor',[.7 .9 .7],...
    'Margin',3);

end

% Plot the midline of the image
plot( (1:Xres),(Yres/2:Yres/2) );

% Plot the endzone locations
plot( Lbound,X );
plot( Rbound,X,'r') ;

end
```

A.41 **FcnPowerLog.m**

```
function [PowerLog] = FcnPowerLog()

% This function loads the power data into Matlab from CSV files in the
% specified folder.

% Get the filenames of the data
Listing = dir('G:\ARL_New\MATLAB_3\power_logger\F201*.CSV');

% Get the number of files of data
NumFiles = length(Listing);

% Initialize 'RawData'
RawData = [];

% For every file...
for n=1:NumFiles

    % Get the data from the current file
    Import = CSVread(strcat('G:\ARL_New\MATLAB_3\power_logger\',Listing(n,1).name),10,0);

    % Add the data to 'RawData'
    RawData(length(RawData)+1:length(RawData)+length(Import),:) = Import;
end

% Produce powerlog file with voltage, current, power

PowerLog(:,1) = RawData(:,2); % voltage
PowerLog(:,2) = RawData(:,3); % current
PowerLog(:,3) = PowerLog(:,1).*PowerLog(:,2); % power
PowerLog(:,4) = RawData(:,1); % iteration

end
```

A.42 **FcnUndistort.m**

```
function im = FcnUndistort( im,DistortionMapping,NumCams,CamRes )

% This function corrects  an image for barrel distorted using a
% pre-computed distortion matrix

% For all three dimensions
for Dimension = 1:3

    % Extract a dimension of the image
    imlayer = im(:,:,Dimension);

    % For all the cameras
    for CamNum = 1:NumCams

        % Get the segment of the image to undistort
        imsegment = imlayer(1:CamRes(1),1+CamRes(2)*(CamNum-1):CamRes(2)*CamNum);

        % Flip the image segment back to how it was originally
        switch CamNum
            case 1
                imsegment = imrotate(imsegment,90);
            case 2
                imsegment = imrotate(imsegment,90);
            case 3
                imsegment = imrotate(imsegment,90);
        end

        % Undistort the image segment
        imsegment = reshape(imsegment(DistortionMapping(:,CamNum)),CamRes(2),CamRes(1));

        % Re-rotate the image segment
        switch CamNum
            case 1
                imsegment = imrotate(imsegment,-90);
            case 2
                imsegment = imrotate(imsegment,-90);
            case 3
                imsegment = imrotate(imsegment,-90);
        end

        % Place the image segment back in the matrix
        imlayer(1:CamRes(1),1+CamRes(2)*(CamNum-1):CamRes(2)*CamNum) = imsegment;
    end

    im(:,:,Dimension) = imlayer;

end

% Crop out overlap in the images
% (this is done in FcnGetImage_Select when FlagPlot is 0
%im(1:480,340:360,:) = 0;
%im(1:480,720:740,:) = 0;
%im(430:480,:,:) = 0;
%im(1:50,:,:) = 0;

% Crop out the orange cone if it is messing up the fiducial identification
%im(213:259,280:355,:) = 0;

end
```

A.43 **FcnUndistort_Transform.m**

```
function im = FcnUndistort_Transform( im,NumCams,CamRes)
```

```
% This function corrects  an image for barrel distorted using original
% transformation. Slow, but necessary to generate and benchmark
% computed distortion matrix table.

for Dimension = 1:3

    % Extract a dimension of the image
    imlayer = im(:,:,Dimension);

    % For all the cameras
    for CamNum = 1:NumCams

        % Get the segment of the image to undistort
        imsegment = imlayer(1:CamRes(1),1+CamRes(2)*(CamNum-1):CamRes(2)*CamNum);
        imsegment = imrotate(imsegment,90);

        % Calls functions where individual camera transformations take place
        imsegment = FcnUndistort_Transform_Inputs(imsegment,CamNum);

        % Place the image segment back in the matrix
        imsegment = imrotate(imsegment,-90);
        imlayer(1:CamRes(1),1+CamRes(2)*(CamNum-1):CamRes(2)*CamNum) = imsegment;
    end

    im(:,:,Dimension) = imlayer;

end

% Crop out overlap in the images
% (this is done in FcnGetImage_Select when FlagPlot is 0
% im(1:480,340:360,:) = 0;
% im(1:480,720:740,:) = 0;
% im(430:480,:,:) = 0;
% im(1:50,:,:) = 0;

% Crop out the orange cone if it is messing up the fiducial identification
% im(213:259,280:355,:) = 0;

end
```

A.44 **FcnUndistort_Transform_Calib.m**

```
function [I6,input_points,base_points] =
FcnUndistort_Transform_Calib(I,XPixRight,YPixDown,RotDegCCW,K)

% This function undistorts an image through translation, rotation, barrel
% distortion correction, skew correction, and cropping

%shift image
T = maketform('affine', [1 0 0; 0 1 0; XPixRight YPixDown 1]);
I2 = imtransform(I, T, 'XData',[1 size(I,2)], 'YData',[1 size(I,1)]);
%rotate image
I3 = imrotate(I2,RotDegCCW);
%barrel distortion correction
I4 = FcnLensDistort(I3,K); % K is distortion parameter
%skew distortion correction
transformtype = 'projective';
imshow(I4)
disp('Enter 4 skew corners as they are.')
disp('Enter 4 skew corners as you want them to be.')
input_points = ginput;
base_points = ginput;
tform = cp2tform(input_points,base_points,transformtype);
I5 = imtransform(I4,tform);
%trim image
I6 = I5(1:360,1:480,:);
```

```
end
```

A.45 **FcnUndistort_Transform_Ind.m**

```
function [I6] =
FcnUndistort_Transform_Ind(I,XPixRight,YPixDown,RotDegCCW,K,input_points,base_points)

% This function undistorts an image through translation, rotation, barrel
% distortion correction, skew correction, and cropping

%shift image
T = maketform('affine', [1 0 0; 0 1 0; XPixRight YPixDown 1]);
I2 = imtransform(I, T, 'XData',[1 size(I,2)], 'YData',[1 size(I,1)]);
%rotate image
I3 = imrotate(I2,RotDegCCW);
%barrel distortion correction
I4 = FcnLensDistort(I3,K); % K is distortion parameter
%skew distortion correction
transformtype = 'projective';
tform = cp2tform(input_points,base_points,transformtype);
I5 = imtransform(I4,tform);
%trim image
I6 = I5(1:360,1:480,:);
end
```

A.46 **FcnUndistort_Transform_Inputs.m**

```
function imsegment = FcnUndistort_Transform_Inputs(imsegment,CamNum)


        % Flip the image segment back to how it was originally & run
        % undistortion
        switch CamNum
            case 1
                %imsegment = imrotate(imsegment,-90);
                XPixRight = 9;
                YPixDown = -20;
                RotDegCCW = 0.5;
                K = -0.4;
                input_points =   [121.1789  318.0222
                  120.2525   99.8560
                  354.6307  320.3382
                  361.1155   97.5400];
                base_points =  [120.7157  318.4854
                  120.7157  100.7824
                  360.6523  319.4118
                  361.1155   98.9296];
                imsegment =
FcnUndistort_Transform_Ind(imsegment,XPixRight,YPixDown,RotDegCCW,K,input_points,base_poi
nts);
            case 2
                XPixRight = 16;
                YPixDown = 0;
                RotDegCCW = 1;
                K = -0.4;
                input_points =  [120.5209  302.0057
                  125.6589   75.9348
                  371.3475  301.0715
                  365.2753   72.1981];
                base_points =  [126.5930  302.0057
                  126.1259   76.4019
                  364.8082  300.6044
                  364.8082   76.4019];
```

```
                imsegment =
FcnUndistort_Transform_Ind(imsegment,XPixRight,YPixDown,RotDegCCW,K,input_points,base_poi
nts);
            case 3
                XPixRight = 22;
                YPixDown = 0;
                RotDegCCW = -.5;
                K = -0.3;
                input_points = [96.1662  299.0311
                   99.8718   32.2291
                  381.9594   32.2291
                  391.6865  299.9575];
                base_points = [96.6294  298.1047
                   97.0926   31.7659
                  393.0761   31.7659
                  394.0025  298.1047];
                imsegment =
FcnUndistort_Transform_Ind(imsegment,XPixRight,YPixDown,RotDegCCW,K,input_points,base_poi
nts);
        end

%       % Undistort the image segment
%       imsegment = reshape(imsegment(DistortionMapping(:,CamNum)),CamRes(2),CamRes(1));

end
```

A.47 **FcnVelocity.m**

```
function [Velocity,Velocity_Filt] = FcnVelocity(DataLog)

% This function produces velocity of robot from position data.

% Method:
% Calculates velocity every iteration (ft/s), from distance traveled and time
% Differencing technique: average of backward and forward difference at
% each point. (Just forward at first point, just backward at last.)
for i = 1:length(DataLog)
    if i == 1; % forward differencing at first
        Velocity(i) = (DataLog(i+1,8) - DataLog(i,8))/(DataLog(i+1,6) - DataLog(i,6));
    elseif i == length(DataLog) % backward differencing at last
        Velocity(i) = (DataLog(i,8) - DataLog(i-1,8))/(DataLog(i,6) - DataLog(i-1,6));
    else % average forward and backward differencing for rest
        Backward = (DataLog(i,8) - DataLog(i-1,8))/(DataLog(i,6) - DataLog(i-1,6));
        Forward = (DataLog(i+1,8) - DataLog(i,8))/(DataLog(i+1,6) - DataLog(i,6));
        Velocity(i) = (Forward + Backward)/2;
    end
end

% Perform filtering on the data

% Define a filter as a 2nd-order Butterworth low-pass filter, with
% bandwidth of 0.01. The 0.01 part was just a guess, since sampling rate is
% not clear from above data... make this number smaller for more smooth,
% like 0.001, and bigger (like 0.1) for more noise but better "tracking" of
% raw data.
[B,A] = butter(2,0.1);
% Perform a forward/backward (noncausal) filtering of data
Velocity_Filt = filtfilt(B,A,Velocity);

end
```

References

[1] "Guide for Evaluating, Purchasing, and Training with Response Robots Using DHS-NIST-ASTM International Standard Test Methods," NIST.

[2] "Apparatus Assembly Guide for Standard Test Methods," NIST, March 2013.

[3] H. Pangborn, "Development and Applications of a Robot Tracking System for NIST Test Methods," B.S. honors thesis, Pennsylvania State University, 2013.

[4] "Talon," [Online]. Available: https://www.qinetiq-na.com. [Accessed 28 March 2014].

[5] "BT-70791A (BB-2590/U)," Bren-Tronics, Inc., [Online]. Available: http://www.bren-tronics.com/. [Accessed 5 April 2014].

[6] "Enter the BomBot," 13 June 2006. [Online]. Available: http://www.defensetech.org. [Accessed 28 March 2014].

[7] "BomBot," 26 September 2007. [Online]. Available: https://www.strategypage.com/. [Accessed 28 March 2014].

[8] "BB-390B/U," Maxa Vision Technologies, [Online]. Available: http://www.maxavision.net/. [Accessed 5 April 2014].

[9] J. Vries, "barrel and pincushion lens distortion correction," Mathworks File Exchange, 31 August 2012. [Online]. Available: www.mathworks.com. [Accessed 6 April 2014].

[10] R. Siegwart, I. Nourbakhsh and D. Scaramuzza, Introduction to Autonomous Mobile Robots, 2nd ed., Massachusetts Institute of Technology, 2011.

[11] L. Vincent and P. Soille, "Watersheds in Digital Spaces," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 13, no. 6, pp. 583-598, June 1991.

[12] B. Hayes, "Dividing the Continent," vol. 88, no. 6, p. 481, November 2000.

ACADEMIC VITA

Adam Crimboli

Campus Address:
67 Atherton Hall
University Park, PA 16802
Phone: (724) 610 - 8581

Permanent Address:
549 Austin St.
Greensburg, PA 15601
Email:
aac5230@psu.edu

Career Interest

To research and work on projects relevant to advancing the fields of robotics, automation, control systems, and mechatronics.

Education

BS - Mechanical Engineering

BS - Nuclear Engineering

The Schreyer Honors College at The Pennsylvania State University
University Park, PA 16802

Graduation Date: May 2014

Relevant Courses

Microcomputer Interfacing
Industrial Robot Applications
Aerospace Control Systems
Modeling of Dynamic Systems
Instrumentation & Measurement

Computer Skills

MATLAB
Simulink
Arduino
Autodesk
ANSYS
SolidWorks

Scholarship

Awards for Academic Excellence in Engineering

- Vollmer-Kleckner Scholarship in Engineering
- Jospeh B. Wharton Memorial Scholarship
- Gabron Scholarship in Engineering
- John J. Brennan Excellence in Nuclear Engineering Award
- Louis Harding Memorial Scholarship

Work Experience

Researcher: Applied Research Laboratory:
The Pennsylvania State University
Embedded Hardware/Software Systems and Applications Dept.
State College, PA

5/2013 – Present

- Conduct mobile ground robot operator variability and power consumption experiments.
- Collect and process image capture and power logger data.
- Analyze and present data using MATLAB.

Intern: Westinghouse Electric Company, LLC
Steam Generator Design & Analysis Dept.
Madison, PA

5/2012 – 8/2012

- Took on more autonomy and responsibilities as a returning intern.
- Processed steam generator corrosion data.
- Created corrosion data plots for management.
- Prepared technical reports for customers.

5/2011 – 8/2011

- Performed 20-year update and digitization of Westinghouse steam generator reference manual.
- Built and stress-tested virtual models of nuclear reactor components using computer analysis software ANSYS.

Affiliated Organizations

American Society of Mechanical Engineers,
PSU Chapter: Treasurer: Fall 2013 – Spring 2014
Alpha Nu Sigma – Nuclear Honor Society,
PSU Chapter
Leonhard Engineering Scholars Program, Penn State
Penn State Ballroom Dance Team