THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF MECHANICAL AND NUCLEAR ENGINEERING

DEVELOPMENT AND APPLICATIONS OF A ROBOT TRACKING SYSTEM FOR NIST TEST METHODS

HERSCHEL PANGBORN
SPRING 2013

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree in Mechanical Engineering
with honors in Mechanical Engineering

Reviewed and approved* by the following:

Sean N. Brennan
Associate Professor of Mechanical Engineering
Thesis Supervisor

Karl Reichard
Research Associate, Applied Research Laboratory
ARL Research Supervisor

H.J. Sommer III
Professor of Mechanical Engineering
Honors Adviser

* Signatures are on file in the Schreyer Honors College.

ABSTRACT

To investigate the standardization of ground robot endurance as defined by the National Institute of Standard and Technology (NIST), this thesis presents both the fabrication of a NIST testing arena and the development of a novel robot tracking system that uses several overhead cameras to record the number of laps and the distance traveled by a robot over the duration of a test. The computer algorithms employed perform four primary functions: 1) image acquisition and correction for camera barrel distortion, 2) localization of the robot through fiducial identification, 3) lap counting between user-defined "end-zones," and 4) conversion of the path traversed from pixels to real-world distances via user-conducted calibrations. Analyses of the precision and accuracy of this system, as well as expected sources of error, are provided.

Two applications relevant to robot endurance are discussed using data from three separate testing events. The first evaluation is the consistency of laps completed (the current NIST method of estimating the distance traversed), comparing distance and time across different robots and operators. The second evaluation considers trends in operator performance over time for the duration of a test.

TABLE OF CONTENTS

List of Figures iii

List of Tables iv

Acknowledgements v

Chapter 1 Introduction 1

Chapter 2 Fabrication of a Testing Arena 4

- 2.1 Overview and Modifications to Assembly Guide Instructions 5
  - 2.1.1 Endurance Bay Walls and Door 5
  - 2.1.2 Support Arches 6
  - 2.1.3 Ramps 6
- 2.2 Materials 7
  - 2.2.1 Bill of Materials 8
- 2.3 Cuts 9
- 2.4 Assembly 10
- 2.5 Lap Markers 12
- 2.6 Fabrication Time and Safety 12
- 2.7 Performance and Suggestions for Improvement 13

Chapter 3 Development of a Data Acquisition System 15

- 3.1 Hardware 15
  - 3.1.1 Camera 16
  - 3.1.2 Ethernet Router 17
  - 3.1.3 Computer 17
- 3.2 Software 17
  - 3.2.1 Software Overview 18
  - 3.2.2 Initialization 20
  - 3.2.3 Loop 29
- 3.3 Performance 41
  - 3.3.1 Frame rate 42
  - 3.3.2 Precision 43
  - 3.3.3 Accuracy 45
- 3.4 Sources of Error 47
  - 3.4.1 Error Due to Angled Camera Perspective 47
  - 3.4.2 Error Due to Inconsistent Coordinate Frames Between Cameras 49
  - 3.4.3 Error Due to Planar Distance Measurement Only 51
- 3.5 Interpreting Testing Event Data 52

Chapter 4 Applications 53

4.1 Data Presentation 55
4.2 Consistency of Laps 60
4.2.1 Different Operators 60
4.2.2 Different Robots 62
4.3 Trends in Operator Performance 62
4.4 Relationships to the Field 63

Chapter 5 Conclusions 64

5.1 Recommendations for Future Work on the Tracking System 64
5.2 Recommendations for Future Work on Applications of the System 66
5.3 Final Remarks 67

Appendix A Instructions for Collecting Data 68

A.1 Using Python Code to Save Images to File 68
A.2 Using MATLAB Code to Process Images from File 70

Appendix B Python Code 71

B.1 Collect_Test_Images.py 71
B.2 MakeVideos.sh 72
B.3 Axis_GStreamer_1.sh 72
B.4 Axis_GStreamer_2.sh 73
B.5 Axis_GStreamer_3.sh 73

Appendix C MATLAB Code 74

C.1 ScriptCalibrate.m 74
C.2 ScriptCollect_Test_Images.m 74
C.3 ScriptLap_and_Dist_Tracking.m 75
C.4 ScriptSort_Data.m 77
C.5 FcnCalcDist.m 80
C.6 FcnCalcLaps.m 81
C.7 FcnFixDistort 81
C.8 FcnFixDistort_Apply.m 82
C.9 FcnFixDistort_Rect.m 84
C.10 FcnGetCalibrations.m 86
C.11 FcnGetImage_All.m 86
C.12 FcnGetImage_Select.m 87
C.13 FcnGetImage.m 88
C.14 FcnGetPosition.m 89
C.15 FcnGetTimestamps.m 90
C.16 FcnInitCamParams.m 91
C.17 FcnInitDistortCorrection.m 92
C.18 FcnInitDistortCorrection_Calib.m 93

C.19 FcnInitDistTrack.m 95
C.20 FcnInitDistTrack_Calib.m 96
C.21 FcnInitDistTrack_Get2Pts.m 98
C.22 FcnInitEndzones.m 98
C.23 FcnInitEndzones_Calib.m 99
C.24 FcnInitTestConditions.m 100
C.25 FcnInitVars.m 101
C.26 FcnLogData.m 101
C.27 FcnMask.m 102
C.28 FcnMask_Color.m 103
C.29 FcnPlot.m 104
C.30 FcnUndisort.m 105

References 106

LIST OF FIGURES

Figure **2-1**: Graphic of continuous pitch/roll ramps apparatus 4

Figure **2-2**: Bay door from right side 6

Figure **2-3**: Bay door from left side 6

Figure **2-4:** Testing arena viewed from door 10

Figure **2-5**: Arena viewed from rear bay wall 11

Figure **2-6**: Talon robot in arena 11

Figure **2-7:** View of traffic cone pylons 12

Figure **2-8:** Stripes identifying end-zones 12

Figure **3-1**: Cameras mounted to posts across support arches 16

Figure **3-3**: Talon robot in test arena 20

Figure **3-4**: Plot at end of initialization 21

Figure **3-5**: Open CV Camera Calibrator [4] 22

Figure **3-6**: Distorted and undistorted images side-by-side 23

Figure **3-7**: Converting camera parameters between OpenCV (left) and MATLAB formats (right) 24

Figure **3-8**: User-calibrated end-zone locations overlaid on images of arena 27

Figure **3-9**: Closer look at image acquisition and fiducial identification 30

Figure **3-10**: Bounding box around Talon robot 32

Figure **3-11**: Colorspace components for RGB and HSV 34

Figure **3-12**: Intensity matrix 35

Figure **3-13**: Image mask 36

Figure **3-14**: Removing small objects and filling in holes 36

Figure **3-15**: Full mask 37

Figure **3-16**: Example of plotted camera images and overlaid information 39

Figure **3-17**: Errors due to camera perspective 48

Figure **3-18**: Measured positions for a testing event with poor pixel-to-feet calibrations 49

Figure **3-19**: Measured positions for a testing event with proper pixel-to-feet calibrations 51

Figure **3-20**: Errors due to ramp angle 52

Figure **4-1**: Talon robot [6] 54

Figure **4-2**: Bombot [7] 54

Figure **4-3**: Position measurements for Talon, Driver 1 (testing event 1) 55

Figure **4-4**: Position measurements for Talon, Driver 2 (testing event 2) 56

Figure **4-5**: Position measurements for Bombot, Driver 1 (testing event 3) 57

Figure **4-6**: Data by lap for Talon, Driver 1 (testing event 1) 58

Figure **4-7**: Data by lap for Talon, Driver 2 (testing event 2) 58

Figure **4-8**: Data by lap for Bombot, Driver 1 (testing event 3) 59

Figure **4-9**: Talon “stuck” between wall and ramps 60

LIST OF TABLES

Table **2-1**: Pitch and roll ramps bill of materials 8

Table **2-2**: Pitch and roll ramps fabrication time 13

Table **3-1**: Cluster properties 38

Table **3-2**: Frame rates 42

Table **3-3**: Standard deviations in pixels of position for a stationary fiducial 43

Table **3-4**: Standard deviation in feet of position for a stationary fiducial 44

Table **3-5**: Accuracy of position measurements 45

Table **3-6**: Accuracy of distance measurements 46

Table **4-1**: Testing event data 59

ACKNOWLEDGEMENTS

This thesis was made possible by collaboration between the Penn State Intelligent Vehicles and Systems Group (IVSG) and the Penn State Applied Research Lab (ARL). I want to thank Dr. Sean Brennan of the former for his mentorship, not only over the process of this research but also during my entire time at Penn State. I want to also thank Dr. Karl Reichard of the latter for his advice throughout the course of my undergraduate research, and for his willingness to deploy the resources, facilities, and manpower that it took to bring this work to fruition.

The input of many of the IVSG graduate students has also been an invaluable resource throughout my time in the Group. I would like to especially recognize Jesse Pentzer for the help he provided in all stages of the process, both as a peer and as a role model for what it means to conduct quality research.

Lastly, I want to thank my parents for their support of my education and passions—both in engineering and other fields.

Funding was made possible from the following source:

The Exploratory and Foundational Research Program, administered by the Applied Research Laboratory, The Pennsylvania State University.

Chapter 1

Introduction

As the capabilities of mechatronic systems develop, the potential for using emergency response robots to save lives in hazardous conditions increases. The disaster at the Fukushima Nuclear Plant, the oil spill of the Deepwater Horizon, and the Chilean Copiapó mine collapse are just a few examples of recent events in which the involvement of advanced, robust robotics could have made — or have made — a positive contribution. This opportunity has been recognized, and investment in emergency response robots has risen accordingly. For example, the 2012 DARPA Robotics Challenge consists of tasks for emergency response robots that could have enabled operators of the Fukushima Nuclear Plant to open pressure valves in a timely fashion when high radiation levels limited the ability of humans to be in the vicinity during the tsunami disaster of March 2012 [1]. Institutions participating in this challenge are eligible for hundreds of thousands of dollars in funding.

The need for methods of evaluating emergency response robots has accompanied this increase in their commercial, academic, and governmental development. With sponsorship from the Department of Homeland Security, the National Institute of Standards and Technology (NIST) is producing a set of standardized metrics to quantify and compare the capabilities of emergency response robots. Test results are to be used as an aid both in robot purchasing decisions and in understanding robot capabilities prior to deployment. As one such metric, NIST uses the number of figure-eight laps completed around an 8’x24’ arena to measure of robot endurance in time and distance [2].

Pre-existing methods of lap counting rely upon human volunteers to manually record each lap—a task that is both time consuming and subject to human error. Additionally, the consistency of this metric is unknown because different robots and operators exhibit variable lap paths. Completing the lap counting task automatically, as well as exploring the consistency of using laps completed to approximate distance traveled, was an initial motivator for the development of a digital data acquisition system for the NIST Test Methods Arena. However, it quickly became apparent that such a system would have numerous other applications. One of these is its use in developing correlations between a robot's performance in a standardized arena and its performance in the field, making its NIST testing results much more meaningful to end users who rely upon the robot to perform real world missions.

Once the decision to develop a system had been made, the method of data acquisition to be used for robot tracking within the NIST arenas was explored. Three sensors in frequent use for tracking vehicles were quickly ruled out. First, GPS receivers cannot be used because many of the NIST test methods are often conducted indoors. Second, Inertial Measurement Units (IMUs), which are a combination of accelerometers, gyroscopes, and often magnetometers, are unusable because the constant "banging around" a robot undergoes as it drives up and down 15º ramps in the testing arena creates too much sensor noise. Finally, rotary encoders placed on the vehicle's drive shaft or wheels cannot account for wheel or tread slip, nor are they easily mounted the wheels or tracks if a robot does not already have encoders. All three of these sensors also require that hardware be mounted to the robot, which can be difficult or impossible to do for robots that were not designed to transport a payload. Additionally, the extra weight of this hardware is likely to negatively affect the robot's performance.

Using cameras to track the motion of the robot does not have the detractions exhibited by other methods. For fiducial identification (locating an object in a digital image using its color), the only modification to the robot required is to tape a bright piece of cardboard to its top surface.

More advanced object identification techniques would require no modification to the robot at all. Furthermore, if the system is designed properly, the banging around of the robot within the ramps does not affect the ability to track its position. This method also allows the position of the robot relative to that of elements in the arena to be easily found. The cameras can literally "see" if the robot has crossed into the end-zones at each end of the arena. This sensor also allows for concurrent measurement of distance traveled and laps completed, enabling their direct comparison as metrics of robot endurance.

Many of the visual tracking and image processing algorithms that were assembled to form the data acquisition system are well documented throughout computer vision literature. However, implementation of these specifically for use in the NIST testing arenas posed a unique problem, particularly with regard to the use of multiple cameras in order to view the entire arena and the development of methods for converting from pixels in the camera images to a real-world coordinate frame.

Chapter 2 of this document details the fabrication of a NIST testing apparatus in a facility of the Penn State Applied Research Lab. Chapter 3 follows the development of the data acquisition system, including detailed descriptions of the algorithms and analysis of its performance. Chapter 4 presents the system's use for several different applications of interest, and Chapter 5 provides final conclusions and suggestions for future work. MATLAB and Python code for collecting and processing data, as well as brief instructions for using this code, are provided in the appendices.

Chapter 2

Fabrication of a Testing Arena

The NIST Apparatus Assembly Guide for Standard Test Methods defines a suite of terrains, targets, and tasks that can be used to evaluate the performance and capabilities of emergency response mobile robots. One terrain element used to measure the endurance of a robot is a series of upwards and downwards sloping ramps arranged into an 8'x24' arena, termed "continuous pitch/roll ramps," and pictured in Figure **2-1**. Within this arena, the number of figure-eight laps around pylons sixteen feet apart that a robot can complete on one full charge of batteries is used to define the endurance of that robot.

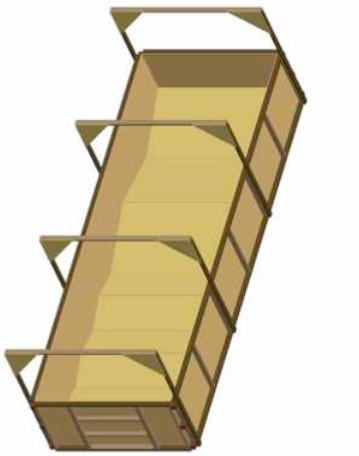


Figure **2-1**: Graphic of continuous pitch/roll ramps apparatus

In order to receive credit for completing a full lap, the robot must cross into the "end-zones" on either end of the arena, which are four feet deep. Before a data acquisition system for the apparatus could be constructed, an arena itself had to be fabricated in-house. The following sections detail this process.

2.1 Overview and Modifications to Assembly Guide Instructions

The apparatus is composed of three basic components: the endurance bay walls and door, the support arches, and the interior ramps. While the guidelines laid out in the February 2011 version of the NIST Apparatus Assembly Guide were generally followed, a few small alterations were made, primarily to the construction of the bay door. Since the construction of the ramps in Summer 2012, a new version of the NIST Apparatus Assembly Guide (dated March 2013) has been released. It introduces one additional modification to the pitch/roll ramps arena, made to the support structure of the interior ramps, which was not a part of this build. This new version also includes details about the pylons and end-zone locations used to define the figure-eight pattern constituting a lap, which were incorporated.

2.1.1 Endurance Bay Walls and Door

The endurance bay walls enclose the ramps and prevent a robot from falling out of the arena if it drives to the edge of the ramps. The walls are made from 7/16" thick oriented strand board (OSB) panels framed with 2x4 posts. At one end of the walls, an 8' wide bay door opens to allow robots to be loaded into and out of the arena. In the NIST Assembly Guide, this is designed to swing downward like an oven door and be secured in place entirely with slide latches. For this fabrication, the bay door was instead hinged so that it could swing outward like an ordinary house door. This makes it much easier and faster to open and close the door. In order to make this modification, an overhang of OSB beyond the 2x4 frame of the door called for in the NIST Apparatus Assembly Guide was removed. One slide latch was installed on the top of the bay walls on the opposite side of the door from the hinges so that the door could be secured in place when closed. Figures **2-2** and **2-3** below show the completed door.

(LEFT) Figure **2-2**: Bay door from right side
(RIGHT) Figure **2-3**: Bay door from left side

2.1.2 Support Arches

Support arches stabilize the bay walls. They are constructed from 2x4 posts and supported at joints with triangles of 7/16" thick OSB paneling. These arches ultimately served not only as stabilizers for the bay walls, but also as beams from which to hang the overhead cameras used to measure robot position.

2.1.3 Ramps

The ramps are placed within the bay walls, and form the surface on which robots drive. The ramps are constructed from 3/4" thick OSB panels, and supported underneath by 4x4 posts. Termed "half" ramps, these are 24.625" long and 4' wide. To save material, the length of each of the "half" ramps was shortened by 0.625", to be an even 24". This made it possible to cut four ramps from one 8'x 4' OSB panel, but also shortened the overall size of the arena by several inches along its 24' length, causing a small gap between the bay wall on one end and the ramps. This gap is negligible enough in size that it does not jeopardize the validity of the fabricated arena as a reproduction of the NIST standard. The March 2013 version of the NIST Apparatus

Assembly Guide has introduced one modification to the ramps that is not reflected in this discussion—the addition of support triangles to the sides of the ramps that add extra strength and stability for tests involving heavy robots. Because no robots heavy enough to require this feature were used in this study, and because the driving surface of the ramps is not affected by the change, the absence of the support triangles in the fabricated arena was not of consequence.

2.2 Materials

The first step in the fabrication process was to determine what materials and how much of each would be required. All of the arena's components are composed of essentially four different materials. These are:

1. Oriented strand board (OSB) panels
2. Wood posts
3. Screws
4. Hardware for the bay door

The parts lists provided in the NIST Apparatus Assembly Guide were compiled into one master list, and adjusted to account for the modifications described above to ramp size, as well as to the bay door.

1. Oriented strand board (OSB) plywood panels
    a. [11] 8' x 4' x 7/16"
    b. [6] 8' x 4' x 3/4"
2. Solid wood posts (non pressure treated)
    a. [41] 2" x 4" x 8'
    b. [4] 2" x 4" x 10'
    c. [7] 4" x 4" x 8'

3. Screws
    a. 1.5”
    b. 1”
4. Hardware for the bay door
    a. [1] Slide Latch
    b. [2] Hinge

2.2.1 Bill of Materials

Prior to purchasing the raw materials for the ramps, the online catalogue for Lowes was used to estimate the cost of these materials. Table **2-1** presents the results. The total materials cost was just under $500.

Table **2-1**: Pitch and roll ramps bill of materials

ITEM	QTY NEEDED	COSTEACH ($)	NET COST ($)
OSB			
8' x 4' x 7/16”	11	9.47	104.17
8' x 4' x 3/4”	6	16.17	97.02
POSTS			
2” x 4” x 8'	41	3.77	154.57
2” x 4” x 10'	4	4.22	16.88
4' x 4' x 8'	7	7.97	55.79
SCREWS			
1-1/2”	2 boxes	10.57	21.14
1”	2 boxes	13.97	27.94
HARDWARE			
Slide Latch	1	7.55	7.55
Hinge	2	2.78	5.56
		TOTAL COST ($) =	490.62

2.3 Cuts

Following purchase and delivery of the raw materials, the posts and OSB panels were cut to the necessary dimensions using a chop saw and a circular saw. Extensive use of jigging greatly reduced the time required to cut the elements, and aided in the standardization of part sizes. Pieces of the following dimensions and quantities were cut:

Oriented strand board (OSB) plywood panels:

[11] 8’ x 4’ x 7/16”

[7] 8’ x 4’ x 7/16” – Wall Panel

[1] 4’ x 4’ x 7/16” – Door

[2] 4’ x 2’ x 7/16” – Front Wall Panel

[8] 2’ x 2’ x 7/16” – Support Triangle*

[6] 8’ x 4’ x 3/4”

[24] 24’ x 48’ x 3/4”– Ramp Surface

Solid wood posts (non pressure treated):

[41] 2” x 4” x 8’

[2] 2” x 4” x 24” – Support Stud Front

[28] 2” x 4” x 45” – Support Studs

[2] 2” x 4” x 45” – Side Plate

[16] 2” x 4” x 96” – Stud Plate

[8] 2” x 4” x 96” – Arch Support Stud

[1] 2” x 4” x 48” – End Cap

[4] 2” x 4” x 10’

[4] 2” x 4” x 107” – Support Arch

[7] 4” x 4” x 8’

[72] 4” x 4” x 6.375” – Ramp Support (15°cut)

[12] 4” x 4” x 3.5” – Connection Block

*Note that the dimensions given for the support triangle are of the base and height of the right triangle shape into which those panels should be cut.

2.4 Assembly

Figure **2-4** presents a view of the assembled arena as viewed from the bay door. Assembly steps are much more extensively described in the NIST Apparatus Assembly Guide for Standard Test Methods.

Figure **2-4:** Testing arena viewed from door

The only tools required for assembly were an electric hand drill and a Phillips head bit. The ramps were assembled first because they were perceived to be the easiest part of the build. Three screws were used to attach each ramp support post to a ramp surface OSB panel. Next, the bay walls and door were assembled. Two screws were used at joints between 2x4 posts in the frame for each segment of the wall, and then an OSB panel was screwed onto each frame. Once these individual parts had been built, the wall panels were connected to form the boundary of the track using 4x4 connection blocks, and the ramps were placed into position. In order to reduce the presence of gapping between the ramps and bay walls, one screw was put through the OSB panels of the bay walls into each of the outer posts of the half ramps

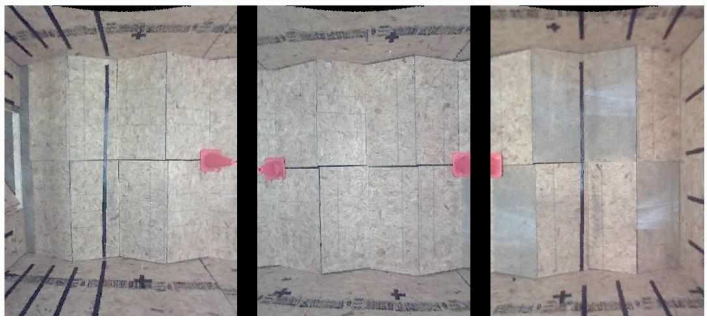
The next step in the build process was to assemble and install the four arches that stabilize the bay walls. The NIST assembly steps state that the vertical arch posts should be installed on the bay walls first and then the horizontal beams and support triangles should be added. To avoid having to stand on a ladder and hold the pieces in place while screwing them together, the team chose to instead completely assemble the arches independently, and then just attach them to the bay walls.

Figures **2-5** and **2-6** present additional views of the completed arena.

(LEFT) Figure **2-5**: Arena viewed from rear bay wall
(RIGHT) Figure **2-6**: Talon robot in arena

2.5 Lap Markers

The March 2013 version of the Assembly Guide calls for pylons to be placed along the midline of the arena, 8’ from either end. It also defines the use of a 4’ deep end-zones at both ends of the arena, identified by painting black and white strips on the portion of the bay walls that contains the end-zones. In order to receive credit for the completion of a lap, a robot must encircle each of the pylons and cross into each endzone during its trip around the arena. The latter criterion was added to prevent nimble robots from cutting sharp corners around the pylons, driving several feet less per lap than their less maneuverable counterparts. In the fabricated arena, orange traffic cones were screwed into the ramps to serve as its pylons. These can be seen in Figure **2-7**. Prior to testing, the endzones were identified with strips of duct tape, as is pictured in Figure **2-8**, an overhead view of the arena from three different cameras.



(LEFT) Figure **2-7:** View of traffic cone pylons
(RIGHT) Figure **2-8:** Stripes identifying end-zones

2.6 Fabrication Time and Safety

The exterior dimensions of the arena are 8’ wide feet by 24’ long by 8’ tall, with an additional 4.5’ of clearance needed on one end so that the bay door can swing open completely. The forty-two man-hours required to fabricate the apparatus are detailed in Table **2-2**.

Table **2-2**: Pitch and roll ramps fabrication time

TASK	# OF WORKERS	# OF HOURS TO COMPLETE	MAN HOURS/TASK
Purchasing Materials	2	1	2
Cutting Wood to Size	3	3	9
Assembling Ramps	1	4	4
Assembling Bay Wall Panels & Door	2	6	12
Assembling Arches	3	1	3
Installing Bay Wall Panels & Door	3	3	9
Installing Arches & Arch Bridge	3	1	3
	TOTAL MAN HOURS: 42		

Beyond following standard safety procedures for woodworking during the fabrication of the arena, there are no additional safety concerns to discuss. It is highly recommended that work gloves, steel-toed shoes, and eye protection be worn at all times during the build process.

The arena has proven to be relatively portable in terms of the time investment required for relocation. A team of three workers was able to take apart the arches and disassemble the bay wall into its component panels in about an hour. After spending a half hour relocating these parts to an adjacent room, it took another hour to reassemble the arena.

2.7 Performance and Suggestions for Improvement

The constructed arena has proven to be very durable after extensive use. The modified bay door can be easily opened and closed to load and unload robots, and when closed forms a structurally sound component of the bay walls. However, the weight of the door causes it to sag slightly on its hinges when open, requiring it to be lifted onto the 2x4 that lies underneath as it is swiveled into the closed position. This could be addressed by placing a wheel between the door and the floor to prevent it from sagging when open. This wheel should be cantilevered from the side of the door, because if it were directly underneath it would interfere with the 2x4.

Some slight gaps (approximately 1” wide) between the ramps along the centerline of the track can emerge during a trial as a robot traverses through the course. Using a hammer to bang on the walls makes it easy to align the ramps back into their proper places.

Overall, the fabricated arena is an excellent reproduction of the standard introduced by NIST, and a suitable environment for which to develop a data acquisition system to augment the research that can be conducted using NIST Test Methods.

Chapter 3

Development of a Data Acquisition System

One element of robot testing procedures not covered by the NIST Apparatus Assembly Guide is the method of data collection used for endurance tests. In previous testing events, this has been a human assistant, using either a mechanical tally counter or pen and paper to log the number of laps completed by a robot in an arena. The original inspiration to develop camera-based data acquisition system was twofold. First, the capacity to conduct digital lap counting eliminates both the need for a human agent to engage in the mundane task of tallying laps for hours at a time and the susceptibility to human error of this manual method of data acquisition. Second and more importantly, a camera system can be used to compute the actual distance traveled and confirm or disprove the validity of using laps completed as a measure of a robot's total travel distance. This is necessary because the consistency of the distance traveled from lap to lap, as well as for different robots and operators, has been observed to be variable.

Beyond examining the metrics of the NIST Standard Test Methods, the camera system has many additional research applications. Explored in Chapter 4, these include its use in determining trends in operator performance with time, and in developing correlations between performance in the testing arenas and in the field.

3.1 Hardware

The camera system includes three types of hardware: 1) the cameras themselves, 2) an Ethernet switch, and 3) a computer to collect and process the camera images.

3.1.1 Camera

Three AXIS 216MFD network cameras were used for the camera system. They were mounted to 2x4 posts bridging the centerline of the testing arena's support arches, approximately 8' above the surface of the ramps, and pointed straight down. Orienting the cameras this way eliminated the need to perform perspective transformations to correct for an angled view of the ramps in order to collect reasonably accurate data. Figure **3-1** shows the AXIS cameras mounted above the testing arena.

Figure **3-1**: Cameras mounted to posts across support arches

The cameras provide an excellent field of view of 100° horizontally, which is sufficiently wide that the entire width and length of the arena can be viewed with three cameras spaced evenly along its length. They also operate with 20 fps at mid-level resolutions. The AXIS branded cameras are easily interfaced with many programming languages, such as MATLAB and Python, which greatly assisted with the development of algorithms for lap counting and distance tracking. Lastly, these cameras are Power Over Ethernet (PoE) equipped, which allows them to be powered by a compatible switch via Ethernet cables only, eliminating the need for additional power supply cables.

3.1.2 Ethernet Router

In order to take advantage of the PoE feature of the cameras, a PoE-equipped Ethernet switch was required. A TRENDnet TPE-S44 was used. This device has eight ports, half of which are PoE compatible. The switch was mounted to the top of the bay walls, about halfway along the arena's length.

3.1.3 Computer

Any computer capable of running the image acquisition and processing algorithms was suitable for use with the camera system, but faster processors and internal hardware capable of handling faster data transmission rates are obviously more desirable. Over the course of testing, several different machines were used, including an Apple MacBook Pro running OS X 10.8, a Dell desktop running Windows 7, and a Panasonic Toughbook running Ubuntu 12.04. All three machines demonstrated comparable performance.

3.2 Software

Developing algorithms capable of identifying a robot within the testing arena, computing the laps completed, and measuring the distance traveled over the course of a testing event is not trivial. In fact, over 80% of the total project time was spent on this task. While commercial products exist to track robots in a 3-dimensional space, these are quite costly and far beyond the budget allowances for a simple test arena [3]. At the core of the code is the use of standard image processing techniques, facilitated by functions built into the MATLAB Image Processing Toolbox. These techniques include fiducial identification (the process of using color to locate an object in a camera's field of view), and corrections for barrel distortion (warping introduced by

camera lenses that causes the picture to appear radially bent around the center of the image). In addition, methods were developed to allow users to input the boundaries of end-zones used in defining the completion of a lap, and to calibrate the system to convert between pixels in the camera image and real-world positions. Computational efficiency was given a great deal of attention throughout the development, to achieve the highest frame rates possible when collecting and processing data. MATLAB was chosen as the programming language in which to write the scripts, both for its ease of use and for the prevalence of embedded functions for image processing. However, this MATLAB-based approach was prohibitively slow when both obtaining and processing images from the cameras in real time, so Python scripts were written to only load synchronized images from the three cameras and save them to file. These images were then post-processed in MATLAB to extract the laps completed and distance travelled. This two-step approach had the added benefit of allowing camera images from testing events to be "played back" as the MATLAB processing code was debugged. All MATLAB scripts and functions used are included in Appendix 1, while the Python scripts used are included in Appendix 2.

3.2.1 Software Overview

The flow chart on the following page, Figure **3-2**, presents a top-level overview of the algorithms developed for the camera system. The code can be roughly divided into two stages: 1) an initialization procedure that loads all the necessary conditions, parameters, variables, and calibration data, finds the starting position of the robot, and plots the images from the cameras so the user can verify that the system is operating properly, and 2) a loop in which new images from the cameras are continually obtained and processed, the number of laps completed and distance traveled are updated, and relevant data is logged. The user may set the loop to continuously plot images from the cameras to verify long-term operation of the system; however, this plotting

activity results in a more than tenfold reduction in frame rate and therefore is recommended only for debugging.

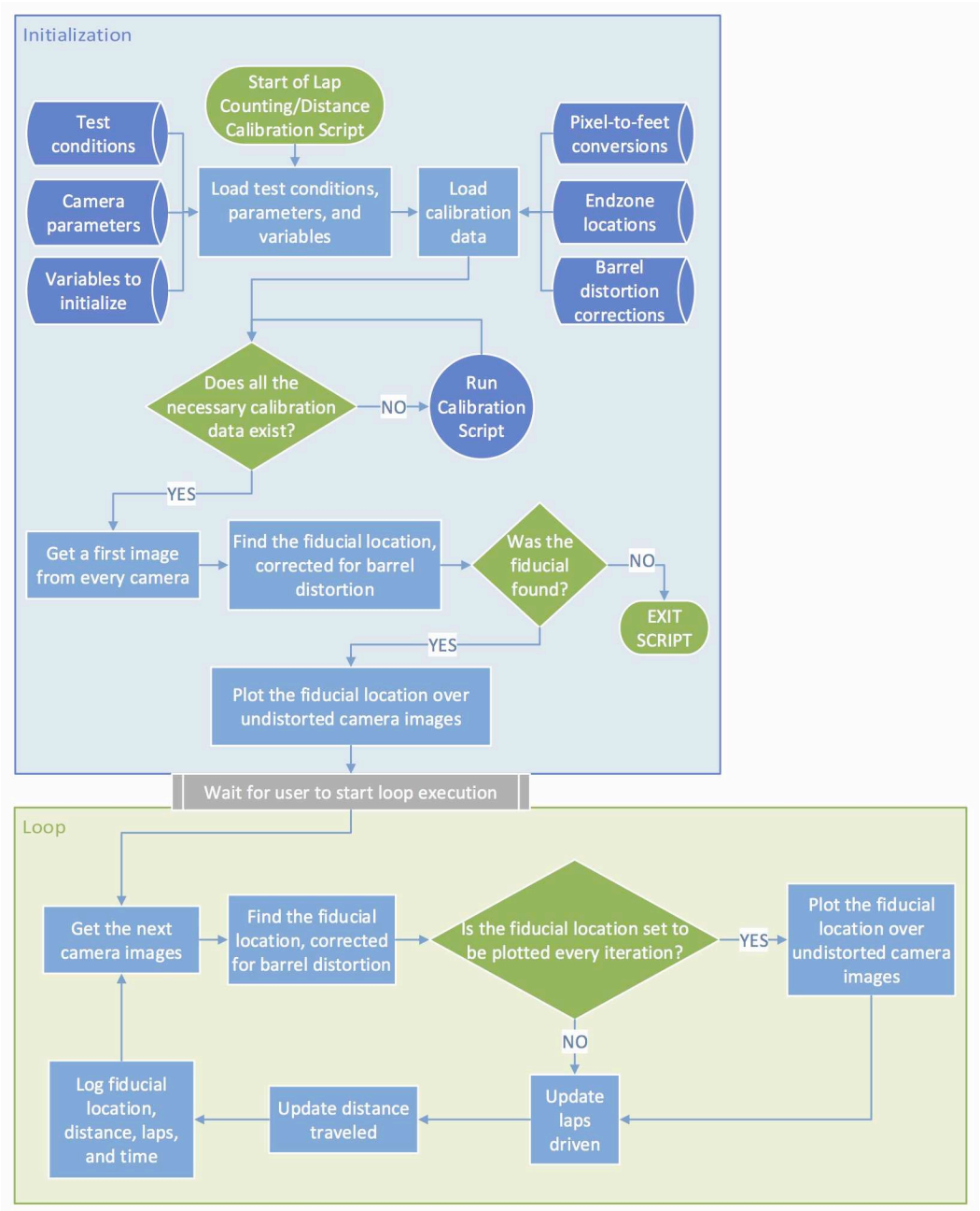


Figure **3-2**: Algorithm flow chart

In order to efficiently and robustly localize the robot's position, a circular piece of bright green cardboard, about 1' in diameter, is taped to the robot's top, along the geometric center left-

to-right and front-to-back. This object serves as a fiducial, identifiable in the camera images primarily by the uniqueness of its color, but also by the known thresholds on its size and eccentricity. Bright green was chosen for the fiducial because of its dissimilarity to the color of the arena surfaces, of the robots to be driven in the arena, and of the traffic cones used as end-zone markers. Figure **3-3** presents an unprocessed image obtained from one of the cameras during testing, in which the fiducial (attached to the Talon robot) clearly stands out. The following discussion presents more specific details of the algorithms developed for the camera system, in roughly the order in which they occur in Figure **3-2**.

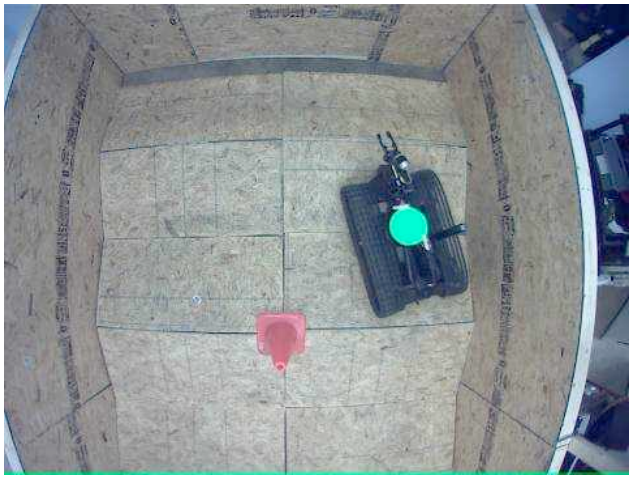


Figure **3-3**: Talon robot in test arena

3.2.2 Initialization

The initialization stage is responsible for loading all the necessary conditions, parameters, variables, and calibration data for the algorithms. If any of the calibration data is missing, this stage provides users with the opportunity to perform those calibrations. It also finds the starting position of the fiducial, ensuring that it is in the field of view of the cameras. If the fiducial is not found in this first attempt, the user is notified and execution of the code is stopped. Lastly, the initialization stage plots the first set of images from the cameras, overlays the fiducial

and end-zone locations, and waits for user confirmation before starting the loop stage. The resulting plot of this process is shown in Figure **3-4**. The blue and red lines represent the left and right endzones, respectively. The fiducial is outlined in yellow, and a crosshair is placed at its centroid. The text next to the fiducial gives its current location in feet, with the origin at top left intersection of the bay walls. Pressing the "START" button begins the loop stage. The initialization stage is explained in greater detail below.

Figure **3-4**: Plot at end of initialization

3.2.2.1 Load test conditions, prameters, and variables

The first step of the initialization is to load a series of test conditions, camera parameters, and variables required throughout the algorithms.

Test conditions

The test conditions include physical information about the system, such as the number of cameras being used, and information about how the system is intended to be used, such as whether camera images are to be pulled live from the cameras or from loaded files stored

internally on the computer or on an external disk. The latter set of conditions allows the algorithms to run in real time or be used to post-process images saved from prior testing events.

Camera parameters

Next, the script loads parameters for each camera in the system. These include their focal length, skew coefficients, and distortion coefficients. These parameters are necessary for correcting the images for barrel distortion, and they vary not only between cameras, but also with the zoom and focus positions of each camera's lens. Two methods by which the parameters can be measured are the Camera Calibration Toolbox for MATLAB and the OpenCV Camera Calibrator. Both procedures involve having the user hold a checkerboard pattern in view and automatically measuring the size and curvature of features in the pattern. Figure **3-5** presents a sample calibration image from the OpenCV Camera Calibrator documentation, with the corners of the checkerboard as extracted by the algorithms.

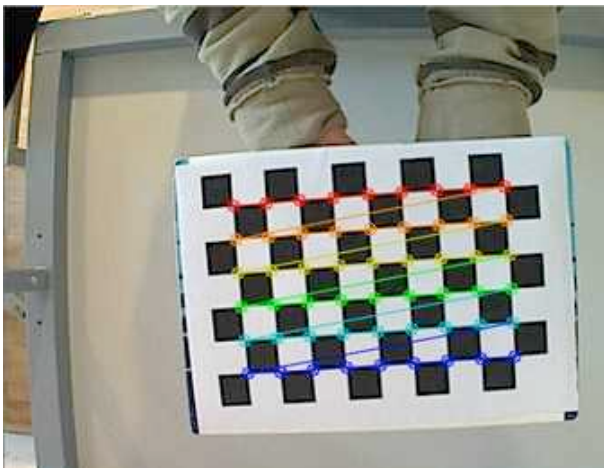


Figure **3-5**: Open CV Camera Calibrator [4]

The OpenCV method was used to find camera parameters in favor of the Camera Calibration Toolbox for MATLAB. It was preferred both due to its much more user-friendly

interface and the observation that it appeared to calculate much more accurate values. The latter conclusion was made after extensive experimentation with each method, in which the OpenCV method repeatedly generated parameters that removed more of the distortion in images than the MATLAB method. Once the parameters were found using OpenCV, distorted images could be run though embedded functions in the Camera Calibration Toolbox for MATLAB to be corrected for distortion. Figure **3-6** presents a distorted image of the testing apparatus next to the corrected image generated by use of these functions.

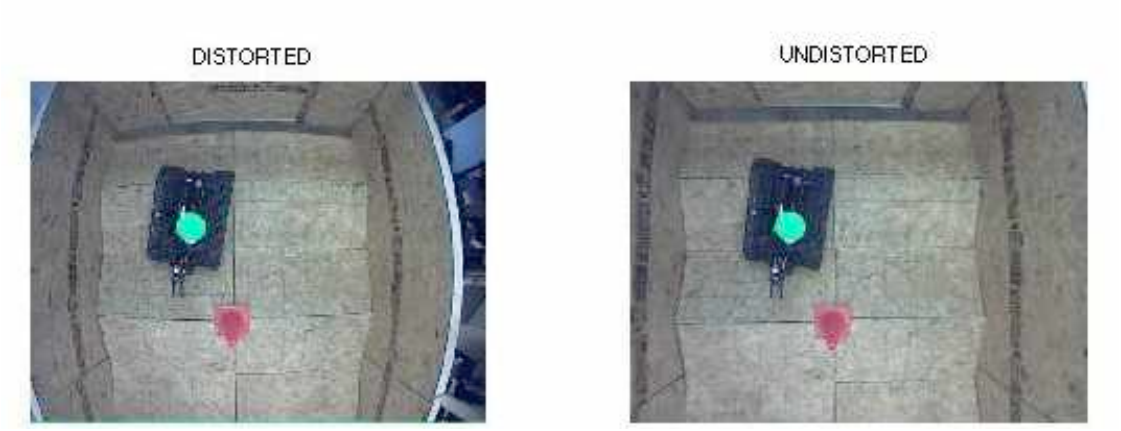


Figure **3-6**: Distorted and undistorted images side-by-side

The effect of the correction is especially observable in the straightening of the bay walls. The MATLAB distortion correction functions are not capable of operating at high frame rates, and therefore were only used in pre-computing transformation matrices capable of performing much faster distortion corrections. This process is part of the calibration script, and is described in more detail later on. It is worth noting that the OpenCV method reports camera parameters slightly differently from its MATLAB equivalent. Figure **3-7** presents a color-coded example of how parameters from OpenCV can be put into the format required by MATLAB

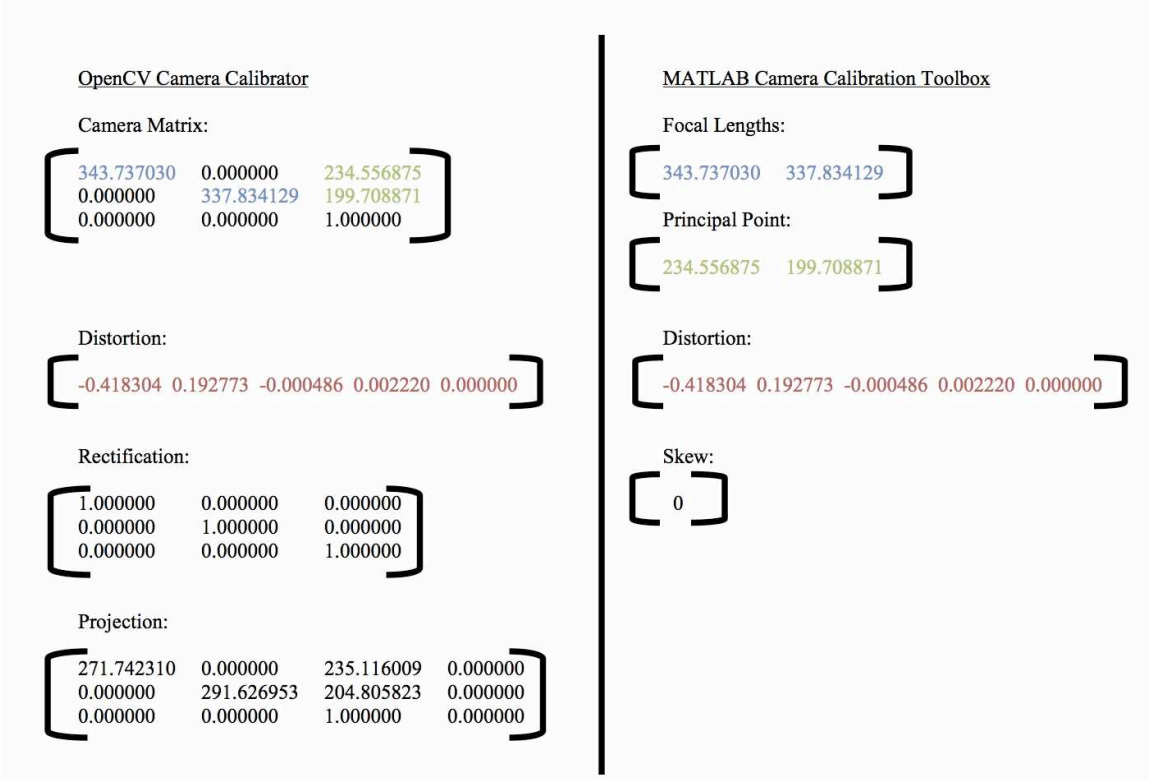


Figure **3-7**: Converting camera parameters between OpenCV (left) and MATLAB formats (right)

Variables

The next step of the initialization stage is to allocate memory for a number of variables that are continuously changed as the algorithms iterate over new images from the cameras. These include the current position of the fiducial, the number of laps completed, and the distance traveled.

3.2.2.2 Load calibration data

The next major process of the initialization stage is to load three sets of calibration data, each of which is stored as a .mat file in the current directory. The first file facilitates conversions from pixels to feet for both axes of every camera image. The second defines the locations of the

arena's end-zones. The third includes several transformation matrices used to quickly remove distortion from images. If any of these files is missing, a calibration script can be used to create them. The process of generating each calibration, as well as the way in which each is packaged in a matrix, is described below.

Pixel-to-feet conversions

In order to report the true distance traveled by the robot, the distance calibration algorithms must include some way to convert between pixel location and feet for any point in each camera's image. One way to do this would be to simply measure the real-world size of the fiducial used to track the robot, and use its pixel dimensions to generate a conversion factor, but because the fiducial is constantly changing elevation and orientation as the robot ascends and descends the ramps, this would not be an accurate method. Users could also place a stationary object of known length, such as a yard stick, in view of each camera and automatically detect and measure its length in pixels, but the accuracy of the pixel measurement would only be as good as the ability to exactly define the boundaries of that object using fiducial identification, which varies with lighting conditions, the color settings of the cameras, and other factors. A third option would be to draw a grid on the surface of the ramps and identify its features, similarly to how the MATLAB and OpenCV camera calibration tools use a checkerboard pattern to obtain distortion parameters. However, such a system would involve intensive development time and require visual modifications to the standardized testing arena defined by NIST. This would also make it more difficult to install and reinstall the arena because the ramps might no longer be interchangeable.

The best solution for this system, and the one ultimately used to develop calibrations between pixels and real-world distance, was to plot the image from each camera on the monitor

and have users define a line segment parallel to each axis of each camera by clicking two points on the image and manually typing the real-world coordinate location of each point. After both the pixel and real-world locations of just two points along each axis of each camera have been input and stored, linear interpolation can be used to define a horizontal and vertical real-world position for every pixel in the image. The real-world coordinate frame chosen is irrelevant as long as it is consistent across all the cameras, since only the distance between points is ultimately necessary to calculating the distance traveled by the robot.

This method results in highly accurate measurement of the real-world positions in the arena, which enabled the real-world distance traversed by a robot to be calculated. More details about the accuracy of the system are given in Section 3.4. As shown in that section, the accuracy is greatly improved when the points selected in calibrations relate to positions at the mean height of the fiducial when attached to the robot to be used for testing, rather than at the height of the surface of the ramps. This is due to the fact that the view of the arena from all points but directly below the cameras is of an angled perspective.

The matrix used to store distance calibration data for each camera consists of two, one-column vectors, with lengths equal to the vertical and horizontal resolution of the camera image respectively. The row index of each column corresponds to a pixel number in its respective axis, and the number stored in the element is its corresponding real-world location along that axis. A third dimension allows data to be stored for each camera in the system into one matrix. These matrices are computed as part of the calibration script, and are saved to file for access by the lap counting and distance tracking scripts. This calibration process needs to be repeated between testing events only if the orientation, position, or focus of the cameras is changed.

End-zone locations

Another task of the calibration procedure is to define the end-zones into which the robot must cross at each end of the track in order to receive credit for completing one full lap. Again, these could be marked physically on the arenas and identified automatically with image processing techniques, but having them be user-defined is much easier to develop and reduces the need to visually modify the NIST standard testing arena. After plotting images from each camera and arranging them end-to end so that the entirety of the arena is visible, the end-zone calibration algorithm has users click two points on each end-zone line. The horizontal zero-intercept and slope of the lines created by connecting each set of points are stored to file, for use not only in conducting lap counting, but also in visually demonstrating the location of the end-zones to the user when plotting images. Figure **3-8** shows the combined images from all three cameras in the arena, with the end-zone lines overlaid in blue on the left and in red on the right.

Figure **3-8**: User-calibrated end-zone locations overlaid on images of arena

Barrel distortion corrections

The final calibration data loaded in the initialization stage is a set of transformation matrices that allow entire camera images, as well as individual pixels in the images, to be quickly

corrected for barrel distortion. These matrices are created by tracking individually the movement of every pixel in the camera images when using the embedded distortion removal functions in the Camera Calibration Toolbox for MATLAB. One by one for every pixel, a matrix is created of the same dimensions as the image with zeros in every element except for that of the pixel to be tracked. The matrix is then run through the distortion removal functions, and the new location of the nonzero entry is found. These locations are stored into a one-dimensional vector, so that every element's index refers to a pixel location in the distorted image, and the value of the element gives the new index location of that same pixel following the distortion removal process. The vector is used as a look-up table when a single point is to be undistorted (namely, the centroid of the fiducial). The vector is then flipped, so that the indexes refer to pixels in the undistorted image, and the value of the elements gives the index locations of distorted pixels. This vector is used to remove distortion from an entire image from the camera, via the MATLAB "reshape" function. Calculating the transformation matrices in the calibration script can take several hours for each camera, but this investment in advance of testing events pays dividends in performance when processing images. The embedded distortion removal functions of the Camera Calibration Toolbox for MATLAB take around 0.3 seconds to process just one image. Using the transformation matrices produces an essentially identical undistorted image in just 0.015 seconds. Removal of distortion on a single pixel is even faster.

3.2.2.3 Initial fiducial position

Once all the necessary matrix data has been successfully loaded, the next step in the initialization is to obtain a first set of camera images, find a starting position for the fiducial, and plot the camera images with the end-zones and position of the fiducial overlaid. These algorithms are identical to those detailed below in the discussion of the loop stage, with two exceptions.

First, if the fiducial is unable to be identified in this first attempt, execution of the script is stopped, because this is a clear indication that there is a fault in either the physical set-up of the test or in the algorithms used to identify the fiducial. Second, regardless of whether the code has been set to plot the camera images during each iteration of the data-collection loop, they are always plotted as part of the initialization process. This allows users to verify that the fiducial was correctly identified. An example of the plot visible following the initialization procedure was presented previously in Figure **3-4**.

3.2.3 Loop

After the fiducial has been identified and plotted in the initialization phase, a virtual button appears that can be pressed to begin execution of the loop. The loop stage, as its name implies, runs continuously until the user manually halts execution, or until the last set of images from each camera has been processed if images are being read from file. The first steps of the loop are to acquire images, find the fiducial's real-world location, and under some conditions also plot the camera images. Next, the number of laps completed and distance traveled is updated. Finally, the data collected is printed to the command window and logged to file.

3.2.3.1 Image acquisition and fiducial identification

Figure **3-9** presents an in-depth flow chart of the algorithms used to acquire images and identify the fiducial location, both in the initialization procedure and the loop. This process is carefully designed to be as computationally efficient as possible. Individual steps are discussed in more detail below.

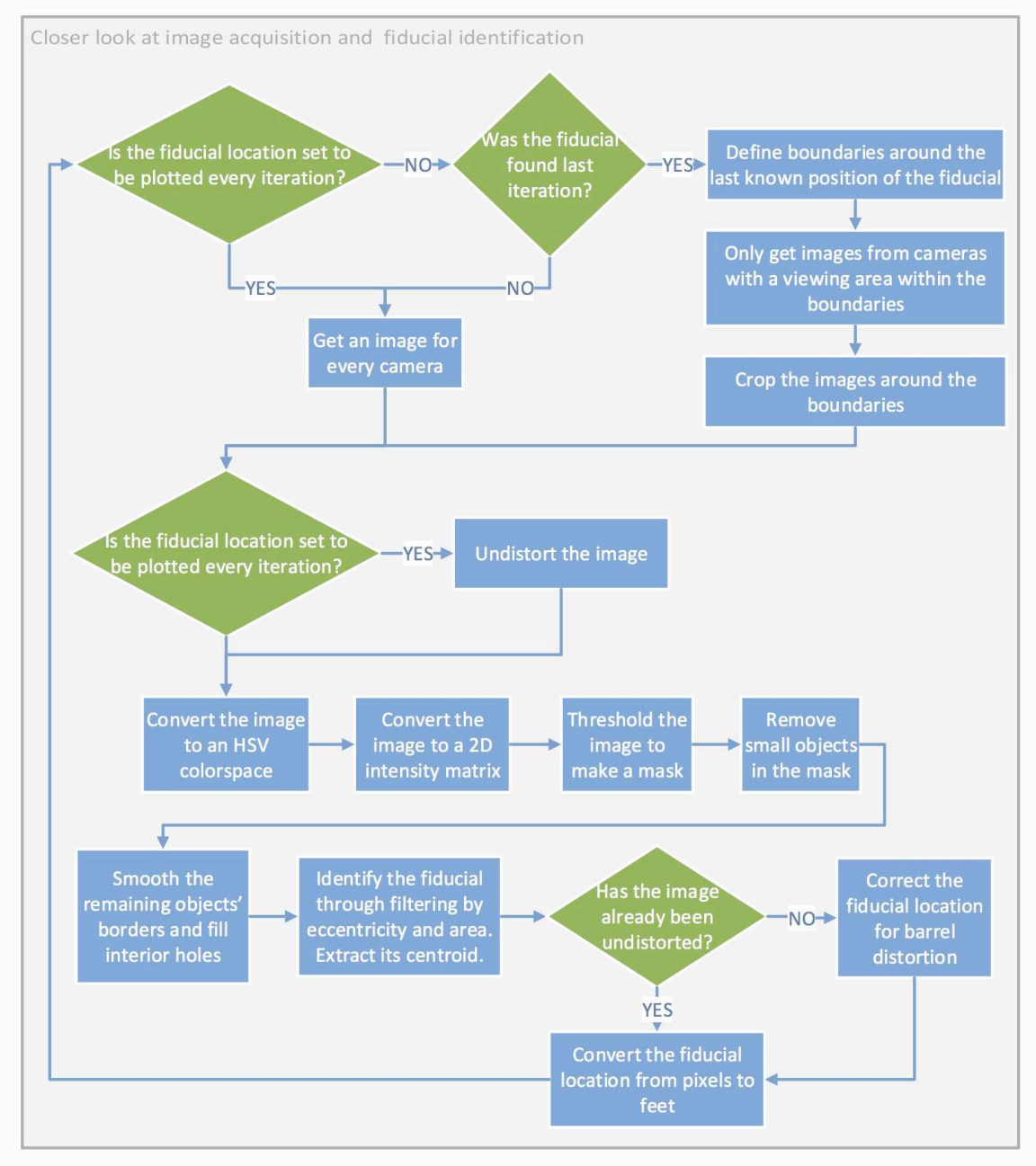


Figure **3-9**: Closer look at image acquisition and fiducial identification

Get the next camera images

The scripts allow images to either be taken live from the cameras or loaded from file. Flags associated with the test conditions loaded during the initialization stage allow the method of image acquisition to be chosen. Images to be loaded from file can be generated with an independent Python script run during testing events, capable of loading and storing images from all three cameras at up to 15 fps. The same task was attempted first with an independent MATLAB script, but it ran at less than 2 fps. Images from each camera are stored in their own folder, and named with the elapsed time since the loop stage began. When these images are loaded for processing, the filenames are extracted and used as timestamps for the images. When images are being loaded in real time from the cameras, the time since the loop began to run is used as a timestamp for the images.

The first step of the algorithms is to check a flag that tracks whether the fiducial was found in the last iteration. If it was, there is no need to process the entirety of every image from every camera, since we know the fiducial must be in the vicinity of its last location. In this case, a bounding box is established around the last known position of the fiducial, the dimensions of which represent the maximum distance that the robot is likely to have traveled since the last iteration. Unless the code has been set to plot the camera images continuously, the algorithm only loads an image from a camera if its view includes a portion of the bounding box, and the loaded images are further cropped along the boundaries of that box. This step is critical to the computational efficiency of the algorithms. If the code is set to plot the camera image continuously, the images are also undistorted as this time using the transformation matrix for removing distortion of the entire image generated during calibrations. Figure **3-10** presents an example of a bounding box created while processing images of the Talon robot driving in the arena.

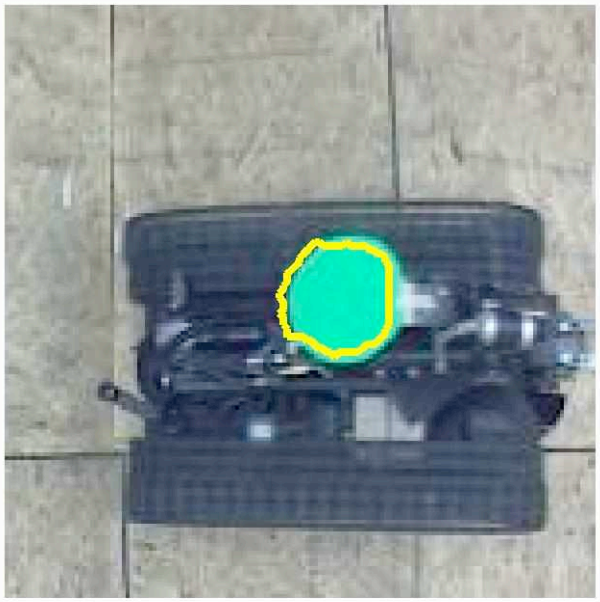


Figure **3-10**: Bounding box around Talon robot

The length used for each edge of the box is 150 pixels (75 pixels on either side of the last fiducial location). This was determined experimentally using data collected while driving the Talon around the arena so as to have as small a box as possible while still guaranteeing that the robot cannot drive fast enough to escape the box between frames. This length of the box can be easily adjusted in the algorithms for use with faster or slower robots.

Regardless of whether a bounding box is formed, the function that obtains camera images does crop some of the edges of the images to remove any area outside the walls of the arena that is in view, as well as to reduce overlap between the camera images. The latter results in the appearance of vertical black bars at the intersection of the images, visible in Figure **3-8.** The width and placement of these can be easily adjusted in code to facilitate a smooth transition between cameras when a robot being tracked crosses between their views.

If camera images are being plotted every iteration, they are corrected for distortion immediately after acquisition using the transformation matrices created by the initialization script. Otherwise only the point location of the fiducial is corrected for distortion in a later step.

Convert to an HSV colorspace

The next step in the process is to convert the image from an RGB to an HSV colorspace. The term colorspace refers to the way in which an image is broken up into layers. This practice traces its roots to the beginnings of color printing. A printer cannot have a different dye for every color in every image it prints, so instead it breaks each color down into a combination of just a few primary colors. For example, purple can be made by layering a 61% dilution of red, a 19% dilution of green, and 100% of blue. This use of individual red, green, and blue layers defines a colorspace called RGB, in which every pixel in the image is assigned a value between 0 and 255 for each layer. A pixel containing lots of red, very little green, and a medium amount of blue might be expressed numerically by (230, 12, 100), in which 230, 12, and 100 are the values of the pixel in the red, green, and blue layers respectively.

The layers of a colorspace can also describe properties other than color. An HSV image is broken into layers of hue (color), saturation (purity), and value (brightness), and is much more useful for fiducial tracking because well-chosen fiducials tend to stand out more from their background in each layer than they do in the layers of RGB images. This is exhibited by Figure **3-11**, which shows the individual components of an image in both RGB and HSV colorspaces. It can clearly be seen that the fiducial (the green circle attached to the robot) tends to stand out much more from the background in the components of the HSV colorspace than the RGB, specifically in the saturation and value components. Additionally, the separation of colors into HSV for the fiducial-detection process gives an inherent immunity to lighting conditions, as lighting changes typically only affect the value (V) space and not hue (H) or saturation (S).

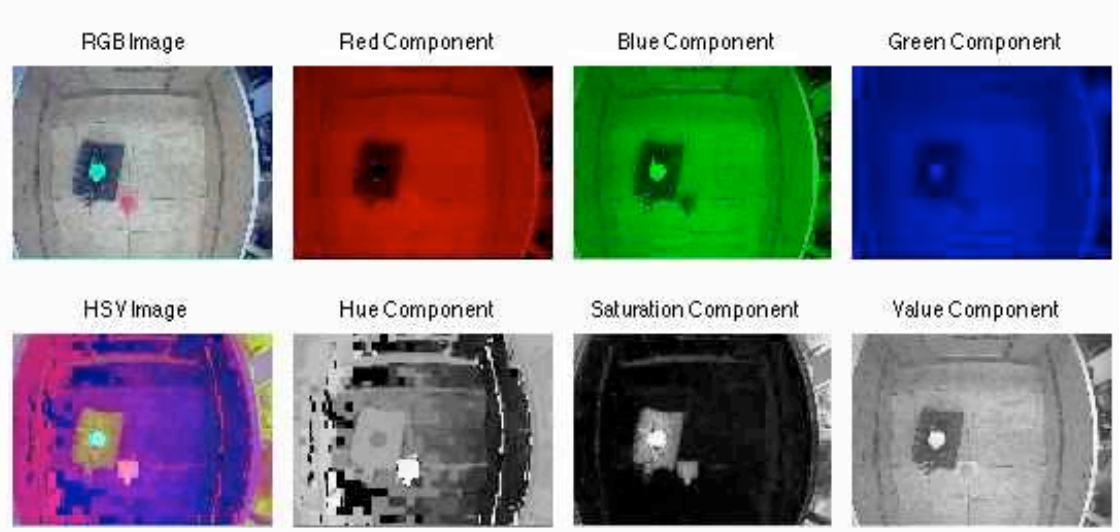


Figure **3-11**: Colorspace components for RGB and HSV

Convert to a 2D binary intensity matrix

Now that the fiducial has been made to stand out as much as possible in each layer of the image, the layers are fused together into a single layer, called an intensity image. This conversion of three layers into one increases the computational efficiency of the process by reducing the number of data points that must be analyzed in later steps by two-thirds. This step also makes those pixels belonging to the fiducial stand out even more from the background.

The intensity matrix is formed by multiplying, adding, and subtracting together the individual layers of the HSV image. For example, in Figure **3-12,** every pixel value in the saturation component has been multiplied by 20, and from each, 10 times the value of the pixel in the same location of the hue component has been subtracted. Representing this example as an equation: I = 0.*H + 20.*S-10.*V, where I is the intensity matrix, and H, S, and V represent the hue, saturation, and value components of the HSV colorspace. The ".*" operator represents an element-by-element product of two matrices. The coefficients in this equation were determined by trial and error, using careful observation of the components of the HSV image as a basis from

which to choose values to try. Figure **3-12** is plotted using a standard blue-to-red colormap, in which pixel values have been scaled to the full colormap range.

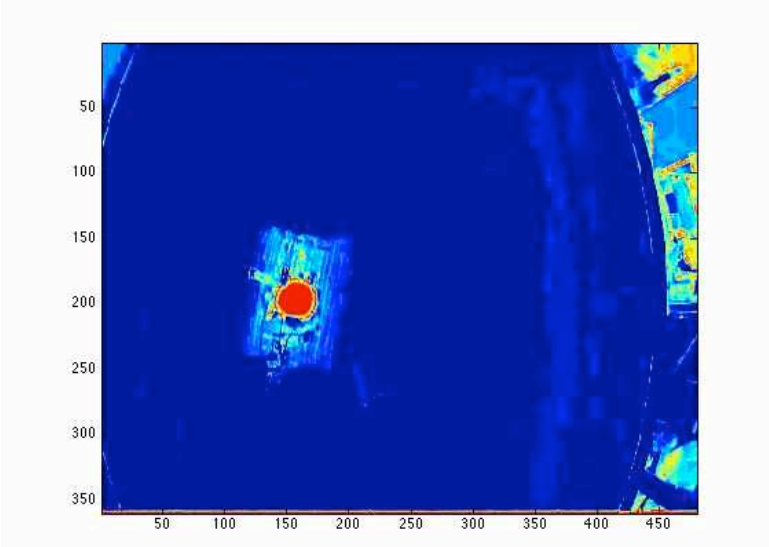


Figure **3-12**: Intensity matrix

Threshold to make a mask

The next step of the process converts the intensity matrix into a binary mask. The mask is a two-dimensional matrix with a 1 at any pixel location that likely belongs to the fiducial and a 0 everywhere else. To create the mask, a threshold is applied to the intensity image, in which only pixels of the highest values (depicted with redder colors in Figure **3-12**) are assigned a 1. The cutoff used for this threshold is carefully selected to preserve robustness. If the cutoff is too high, pixels belonging to the fiducial could be improperly removed from consideration. If it is too low, pixels that are not the fiducial will remain in consideration.

Figure **3-13** presents a binary mask, in which all pixels with a value less than or equal to 65% of the maximum possible have been filtered out. Represented as an equation,
Mask = (logical) I ./20 > 0.35, where I is the intensity matrix and the "./" operator represents the element-by-element division of two matrices.

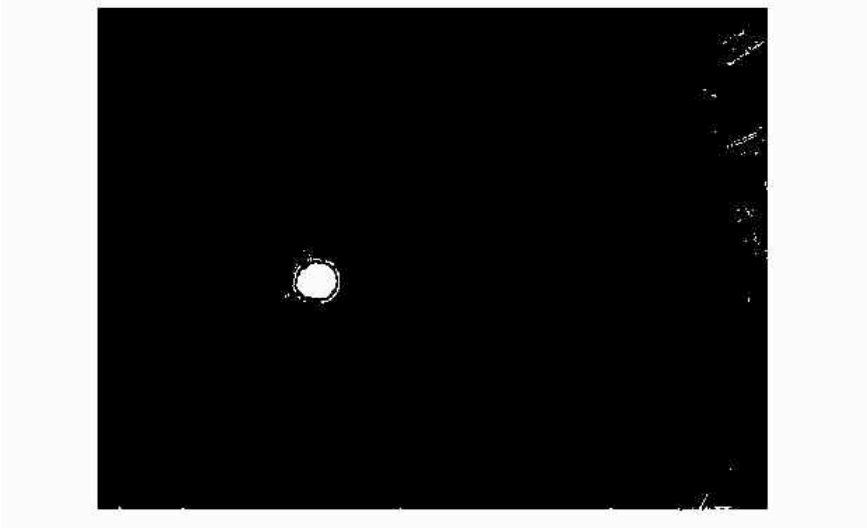


Figure **3-13**: Image mask

Remove small objects and fill holes in remaining objects

Next, clusters of white pixels that are too small to be the fiducial are removed, and interior holes in the remaining clusters are filled. Figure **3-14** presents the effects of the step on the portion of the mask containing the fiducial. To aid in the demonstration, a square hole has been artificially added to the mask near the center of the cluster of pixels corresponding to the fiducial. The initial mask is given in figure **3-14(a)**.

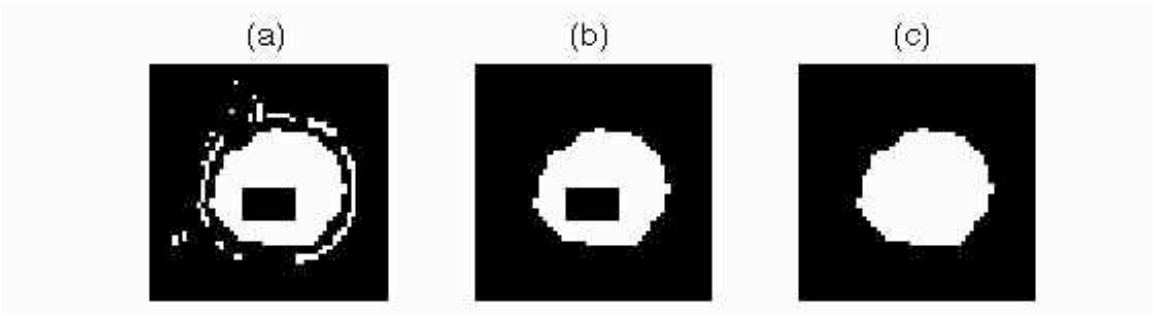


Figure **3-14**: Removing small objects and filling in holes

In **3-14(b)**, any instances of less than sixty connected pixels have been removed from the image, using the MATLAB "bwareaopen" function. This is visible by the removal of the spots

around the fiducial. If too small a number is used, the clusters of pixels around the fiducial will not be removed. If too large a number is used, the fiducial itself could be removed.

In **3-14(c)**, interior holes in the remaining objects have been filled using "imfill" in MATLAB. It is easy to see that the square hole has been filled in. In Figure **3-15**, the full area of the mask is visible. It is clear that all of the white clusters from the top right corner of Figure **3-13** have been eliminated as well as a result of using the "bwareopen" function. However, some clusters, just barely visible along the bottom edge of the mask, persist.

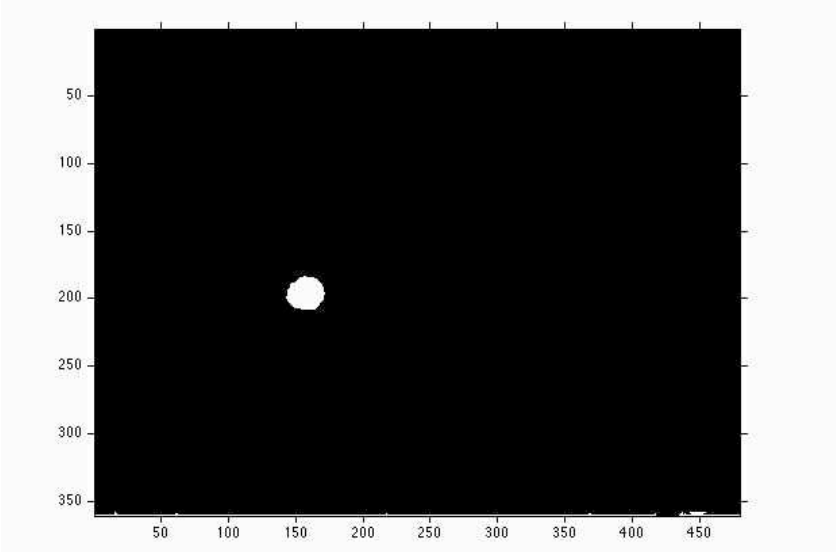


Figure **3-15**: Full mask

Filter by eccentricity and area

Using the MATLAB "regionprops" function, we can identify the three remaining pixel clusters in the mask of Figure **3-15**, and obtain the area, eccentricity, and centroid of each. Table **3-1** presents this data. Even without using the centroid information, we can tell that cluster 2 is the fiducial because of the combination of a small eccentricity (meaning that the component is very circular) and a large area. Filtering by these two criteria allows the algorithms to make the same conclusion, and assign the centroid of the correct cluster as the location of the centroid of

the fiducial. The specific filter criteria used in the algorithms are an eccentricity less than 0.8 and area less than 2000 pixels.

As is discussed in Section 3.3.2, a new fiducial location is only posted if it is further than one standard deviation from the stationary fiducial noise along each axis of the image. Otherwise, the location used in the preceding iteration is carried through.

Table **3-1**: Cluster properties

CLUSTER NUMBER	1	2	3
AREA (pixels)	426	542	71
CENTROID (pixels)	(208 , 360)	(157 , 197)	(454 , 359
ECCENTRICITY	1.000	0.4831	0.9980

Correct for barrel distortion and convert to feet

Unless the images are being plotted every iteration and thus have already been corrected for barrel distortion, the location of the fiducial itself needs to be corrected. In either case, the location needs to be converted from pixels into a real-world coordinate frame. Correcting the centroid location for barrel distortion and converting it to a real-world coordinate frame is as simple as using the lookup tables generated for each purpose in the calibration script. For the distortion correction, the index of the distorted centroid pixel corresponds to a row in the table, the element of which gives the undistorted location in pixels. This is in turn used as an index in the rows for pixel-to-feet conversions to find the real-world location of the point along both the vertical and horizontal axes.

Plot the fiducial location

If the option to plot the fiducial location is turned on, or the function is being run as part of the initialization phase, the image from each camera is plotted, and the end-zone locations are overlaid, as well as the outline and centroid of the fiducial. Lastly, the real-world location of the centroid in feet is displayed in text. Figure **3-16** shows an example of such a plot.

Figure **3-16**: Example of plotted camera images and overlaid information

3.2.3.2 Update laps completed

From the start of the code, the lap counting algorithm checks the fiducial position each iteration to determine whether one of the end-zones has been entered. After this occurs, the end-zone entered is stored as the "last end-zone," and the code only checks whether the fiducial has crossed into the opposing end-zone. When it does, a half lap is added to the total laps completed. This method ensures that a robot does not receive credit for multiple laps by crossing back and forth on the boundary of just one end-zone. Whether an end-zone has been crossed is computed by plugging the vertical pixel centroid coordinate into the slope-intercept form of the straight line equation, using the values of zero horizontal intercept and slope calculated in the end-zone calibrations. The output of this equation is compared to the horizontal pixel centroid coordinate

to determine on which side of the end-zone line the centroid lies.

3.2.3.3 Update distance traveled

The distance traveled by the fiducial is calculated using the distance formula between the current and prior real-world positions of the fiducial, and adding this to the previous total.

3.2.3.4 Log Data and Print to Command Window

For robustness, data is logged in two different formats. Described in more detail below, these formats are: 1) A series of .mat files, and 2) A single .txt file. This data includes the current iteration, lap number, fiducial location in pixels, fiducial location in feet, elapsed time since the start of the test, and distance traveled since the start of the test. This information is also printed to the MATLAB command window every iteration, along with notice if the fiducial is not located in a particular iteration. By adjusting a flag in the "Test Conditions" function, the plotted camera images can also be saved to file (although doing so significantly slows down the processing time per iteration) and later used in movie editing software such as iMovie to create stop-motion videos showing the tracking of the robot.

.mat files

The primary format of storing data is via .mat files. To prevent complete loss of data in the event of computer crashes, data from batches of a preset number of iterations (this number is easily adjustable using the "Test Conditions" function") are saved to file in sequentially named

.mat files. Whenever the number of iterations that have elapsed since the last save equals this preset number, the most recent batch of data is saved to file.

.txt files

As a backup format by which to save data, the results of each iteration are appended to a .txt file stored in MATLAB's current directory. Text files are useful in the rare situations where the MATLAB software crashes during operation; in a crash, the 'mat' files are not recoverable, whereas the text files usually are. Thus, one would not lose an entire test run – which takes days to prepare and hours to run – if the software were to crash mid-test.

3.3 Performance

The camera system was evaluated for performance in terms of its frame rate, precision, and accuracy. Frame rate can pertain to both the speed at which images can be acquired and the speed at which they are processed, although in some configurations these tasks are performed sequentially at every frame. Precision is measured by the variance in measurement for a stationary position. Accuracy is measured through two means. They are: 1) The mean real-world position measurement of the fiducial as compared to a hand measurement of the same, and 2) Distance measurements for straight traverses along both axes of the arena as compared to the known distances of those traverses. The results of this performance analysis are discussed in detail below. Suggestions of sources for the error that was measured are provided, along with the steps taken to account for this error in code.

3.3.1 Frame rate

Table **3-2** presents an approximate frame rate for the code when running in several different configurations, including when only loading and storing images from all three cameras using Python or MATLAB, and when processing images in real time or from storage using MATLAB. The fastest method is clearly to save images using Python during testing events, and then post-process them from file using MATLAB later on. The only disadvantage of this method is that it does not allow the laps completed and distance traveled to be conveyed to users in real time during a testing event. However, by plugging two computers into the Ethernet switch, MATLAB and Python scripts can be run simultaneously, each on its own machine, without interference. This provides users with real time estimates via the live processing of low frame rate images by MATLAB, which can be updated when there is time to post-process the higher frame rate images collected and stored to file by Python.

Table **3-2**: Frame rates

SAVING IMAGES TO FILE ONLY	**FRAME RATE (fps)**
MATLAB	1.5
Python	15
PROCESSING IMAGES LIVE	
MATLAB (not plotting every iteration)	3.0
POST-PROCESSING IMAGES FROM FILE	
MATLAB (not plotting every iteration)	30
MATLAB (plotting every iteration)	0.9

3.3.2 Precision

The precision of the system is measured by the variability in measurements of a stationary fiducial position. Early on in the evaluation of the system, it was observed that the noise in measured position of a stationary fiducial resulted in increasing calculations of distance traversed when in reality the robot was not moving.

In order to remove the noise, the standard deviation of both axes for approximately 50 position measurements of a stationary fiducial was found at about a dozen different locations across the arena, and the mean of these was used as a threshold against noise. The standard deviations are provided in Table **3-3**. The mean standard deviation of each axis is about a quarter of a pixel.

Table **3-3**: Standard deviations in pixels of position for a stationary fiducial

MEASURED POSITION (pixels)		STANDARD DEVIATION (px)	
X-axis (Length)	Y-axis (Width)	X-axis (Length)	Y-axis (Width)
411.2034	389.6780	0.4060	0.4713
698.0877	397.1053	0.2854	0.3096
477.0000	325.1818	0.0000	0.3892
933.8246	303.0000	0.3837	0.0000
182.0000	235.0000	0.0000	0.0000
561.5161	241.0000	0.5038	0.0000
891.4677	242.1774	0.9002	0.8782
1042.7000	239.0192	0.4660	0.1387
773.0000	200.0000	0.0000	0.4572
235.0000	194.7089	0.0000	0.4572
83.0000	118.0000	0.0000	0.0000
548.0000	120.7308	0.0000	0.0000
895.0000	75.0141	0.0000	0.1187
MEAN STANDARD DEV:		**0.2265**	**0.2477**

To prevent this noise from inflating the distance measurement when the fiducial is stationary, a given iteration's measured fiducial position is only used to update the current location estimate if its distance from the previous position is more than three standard deviations (about 0.75 pixels) in both axes from the position at the preceding iteration.

For tall robots (over 12” in height) this noise increases significantly because the fiducial is at a greater elevation and therefore closer to the cameras. Enlarging the size of the fiducial increases the variance in the identification of its boundaries, also increasing the noise in the centroid positions found. The thresholds against noise were increased for testing with the Talon robot until the noise in the stationary fiducial position measurements no longer resulted in inflated measurements of traveled distance. A suitable threshold when testing with tall robots was found to be approximately two pixels for each axis, which corresponds to about half an inch in position.

The standard deviations in position measurements of stationary fiducial can also be expressed in terms of the measured real world coordinates generated from pixel-to-feet conversions. Table **3-4** presents data from the same testing events as Table 3-3, but using the real world positions calculated instead of image pixel locations. The average of the standard deviations for each location happened to be equal, at around 1/16” (note that the units in Table **3-4** are feet).

Table **3-4**: Standard deviation in feet of position for a stationary fiducial

MEASURED POSITION (ft)		STANDARD DEVIATION (ft)	
X-axis (Length)	Y-axis (Width)	X-axis (Length)	Y-axis (Width)
8.0127	0.2943	0.0113	0.0116
16.0302	0.1110	0.0080	0.0076
9.8515	1.8858	0.0000	0.0096
21.1801	2.5658	0.0099	0.0000
4.3895	4.2562	0.0000	0.0000
12.2171	3.9630	0.0141	0.0000
20.0919	3.9866	0.0231	0.0205
23.9770	4.0604	0.0120	0.0032
17.0483	4.9719	0.0000	0.0000
5.8710	5.2769	0.0000	0.0116
1.6221	7.2202	0.0000	0.0000
11.8357	6.9307	0.0000	0.0111
20.1826	7.8917	0.0000	0.0028
MEAN STANDARD DEV:		**0.0060**	**0.0060**

3.3.3 Accuracy

As previous stated, accuracy was measured in two ways. These are: 1) The mean real-world position measurement of the fiducial as compared to a hand measurement of the same, and 2) Distance measurements for straight traverses along both axes of the arena as compared to the known distances of those traverses. While the latter aligns more closely with the actual use of the testing system, the former is still a good metric by which to judge whether the system functions properly.

3.3.3.1 Position Measurement Accuracy

The same testing data that was used to measure the system's precision was applied to evaluate the accuracy of measured positions. Table **3-**5 presents the results. The mean of the error at every location was about 1.5" in both axes.

Table **3-5**: Accuracy of position measurements

TRUE POSITION (ft)		**MEAN MEASURED POSITION (ft)**		**MEAN MEASUREMENT ERROR (ft)**	
X-axis (Length)	Y-axis (Width)	X-axis (Length)	Y-axis (Width)	X-axis (Length)	Y-axis (Width)
8	0	8.0127	0.2943	0.0127	0.2943
16	0	16.0302	0.1110	0.0302	0.1110
10	2	9.8515	1.8858	0.1485	0.1142
21	2.5	21.1801	2.5658	0.1801	0.0658
4.67	4	4.3895	4.2562	0.2805	0.2562
12.25	4	12.2171	3.9630	0.0329	0.0370
20	4	20.0919	3.9866	0.0919	0.0134
24	4	23.9770	4.0604	0.0230	0.0604
17	5	17.0483	4.9719	0.0483	0.0281
6	5	5.8710	5.2769	0.1290	0.2769
2	7	1.6221	7.2202	0.3779	0.2202
12	7	11.8357	6.9307	0.1643	0.0693
20.17	8	20.1826	7.8917	0.0126	0.1083
	MEAN MEASUREMENT ERROR FOR ALL POSITIONS (ft):			**0.1178**	**0.1273**

3.3.3.2 Distance Measurement Accuracy

The second method of evaluating the accuracy of the system is to drive a robot back and forth parallel to axes of the arena for a known distance and compare this to the distance measured by the system. Because the "true" distance only took into account the distance traveled along one axis at a time, the method of calculating distance in the system's code was modified to do the same, so that any lateral changes in position undergone by the robot did not contribute to the calculated distance. The results of this testing, performed at several locations in the arena, are presented in Table **3-6**.

Table **3-6**: Accuracy of distance measurements

PATH DRIVEN	**TRUE DISTANCE (ft)**	**MEASURED DISTANCE (ft)**	**ERROR (%)**
LENGTH AXIS			
Along left wall	133.65	130.10	-2.66
Along center	133.65	129.33	-3.23
Along right wall	133.65	130.10	-2.66
		MEAN % ERROR:	**-2.85%**
WIDTH AXIS			
Along 2nd Ramp	45.5	49.26	+8.26
Along 5th Ramp	45.5	45.74	+0.53
Along 11th Ramp	45.5	46.71	+2.66
		MEAN % ERROR:	**+3.82%**

The distance accuracy testing results are very promising, and certainly more accurate than any method of estimating the distance traveled from only the number of laps completed. Measured distances along the length of the arena were about 3% lower than the true values, while those along the width were about 4% high. As expected, this percent error in distance is larger than that of the position measurements. Because the distance traversed is calculated from the

accumulation of many position measurements, the error accumulates as well. While there does appear to be consistent bias in each axis direction, neither is significant enough to merit the addition of a "fudge factor" or other corrective measures. Potential sources of the accuracy error observed are discussed below.

3.4 Sources of Error

Three major sources of error are believed to exist within the testing system. These are: 1) errors due to the angled perspective of the cameras, 2) errors due to inconsistent real world coordinate frames between cameras, and 3) errors due to measurement of only planar traversal of the ramps. These are discussed in detail below.

3.4.1 Error Due to Angled Camera Perspective

One of the largest errors expected is a result of the angled perspective of the cameras relative to all locations in the arena except those directly underneath one of the cameras. If the cameras are calibrated to convert from pixels to feet using points at the height of the surface of the ramps, Figure **3-17** shows how the height of the fiducial off the ground creates a discrepancy between the position of the fiducial perceived by the camera and its actual location. For a camera angle of just 15º and a mean fiducial height of 15" (an approximate height when attached to the Talon robot), the difference between the perceived and actual position of the fiducial is about 4".

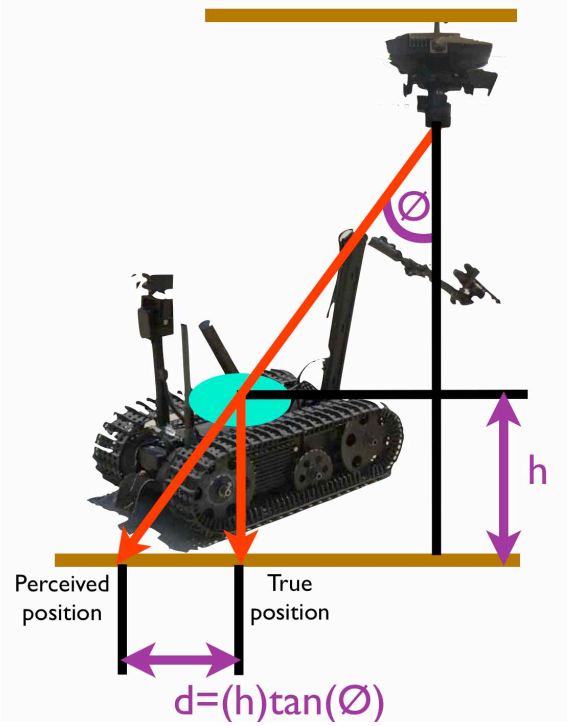


Figure **3-17**: Errors due to camera perspective (robot image from [6])

One way to mitigate this source of error is to perform the pixel-to-feet calibrations using markers placed at the mean height of the fiducial on the robot in the arena, effectively calibrating the system to a plane at the mean height of the fiducial. However, because scaling the arena's ramps causes the robot to rise and fall by 6" (or 3" on either side of its mean position), this error cannot be completely removed with the current level of sophistication of the system. The rising and falling of the robot and fiducial from their mean position is believed to be responsible for most of the overall error found in the previous section. This also explains why measured distances along the "width" axis of the arena are found to be greater than those along the "length" axis relative to their true values.

A simple way to fix this error is to calculate the fiducial's likely orientation and height based on its position relative to the ramps. This would require position-sensitive corrections and the development of a grid used to map the location of each ramp. While this would be relatively

simple to implement, it was not incorporated into the system because it was felt that the error was small enough even without this calibration step.

3.4.2 Error Due to Inconsistent Coordinate Frames Between Cameras

A second source of error results from instances of inconsistent coordinate frames between cameras when calibrations for pixel-to-feet conversions are conducted. This can cause the real world coordinate of the fiducial to seemingly "jump" in position when the robot crosses between two cameras' fields of view. Figure **3-18**, shows the path created by connecting fiducial locations measured for a test in which a robot drove back and forth in a straight line along the length of the arena.

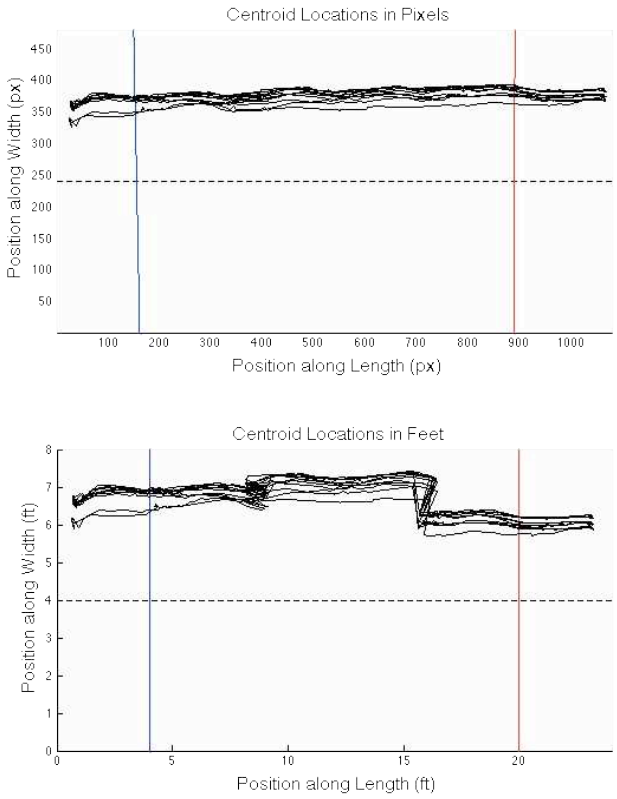


Figure **3-18**: Measured positions for a testing event with poor pixel-to-feet calibrations

The top plot shows the locations in pixels, and the lower in feet. While ripples in this plot can be seen at each ramp due to the camera perspective error discussed previously, there appears to be a smooth, essentially imperceptible handoff in pixel locations measured when the fiducial transitions from the view of one camera to that of another. However, in the lower plot, the fiducial measurement in real world coordinates jumps sharply every time the fiducial passes between camera views. This occurs because the mapping between pixels and a real world coordinate frame must be calculated individually for each camera. The real world locations manually entered during calibrations for each camera must refer to a consistent coordinate frame across all cameras. While careful calibration of each camera's pixel-to-position can greatly reduce this error, it cannot be completely eliminated. Figure **3-19** shows an example of data for a similar testing event to that used in Figure **3-18**, but with a more consistent real world coordinate frame.

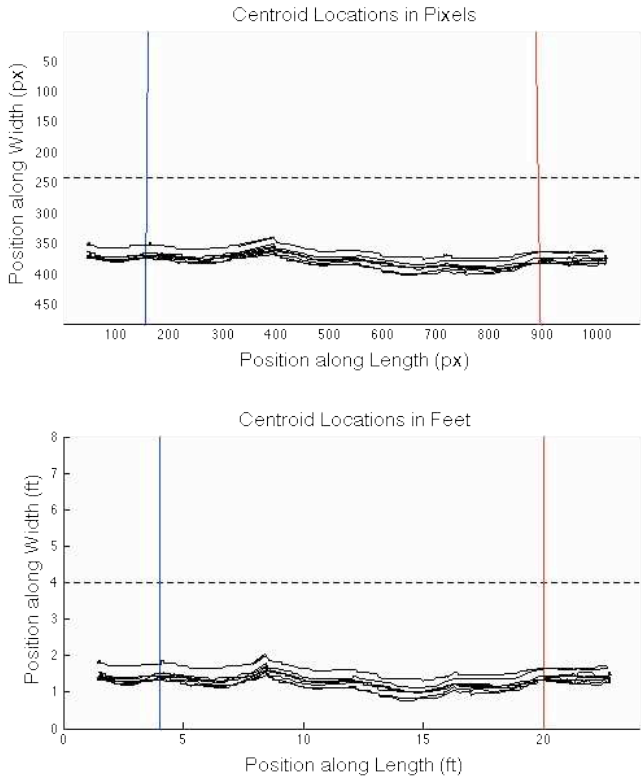


Figure **3-19**: Measured positions for a testing event with proper pixel-to-feet calibrations

3.4.3 Error Due to Planar Distance Measurement Only

A third source of error between the true distance traveled by the robot and the measured distance stems from the fact that the overhead nature of the cameras only enables them to view the planar distance traveled. As depicted in Figure **3-20**, although the robot is traversing the hypotenuse of the triangle formed by the ramps surface and the actual floor, only the base of this triangle can be measured from overhead. Due to the relatively small angle of the ramps of 15º, the distance measured as a result of this error is only approximately 3.5% less than what it would be were the error not present. However, this error does not play a role in the accuracy calculations presented in the previous section, because the "true" distances used also only took into account planar distance traveled.

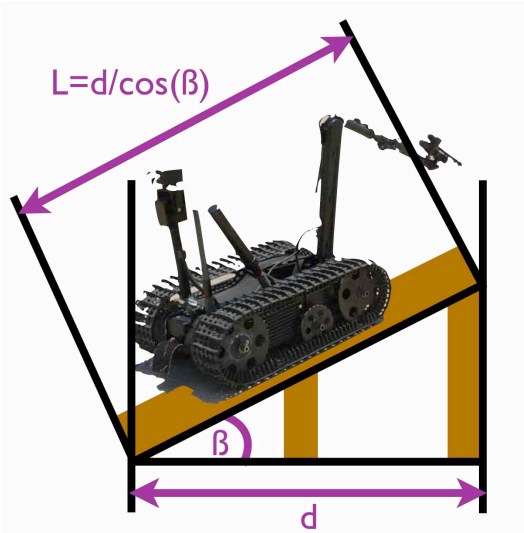


Figure **3-20**: Errors due to ramp angle (robot image from [6])

3.5 Interpreting Testing Event Data

To facilitate in the interpretation of data, a MATLAB script was written that loads the data collected for each camera image (the loop iteration, lap number, fiducial location in pixels, fiducial location in feet, elapsed time since the start of the test, and distance traveled since the start of the test). This data is indexed by laps, providing the distance traversed and time taken to complete each. Plotting this information allows trends in distance per lap and time per lap to be easily viewed. The script also plots the measured fiducial location in both pixels and feet for every iteration, as shown in Figure **3-18** and **3-19**. This allows users to inspect the results of image processing visually and to verify that the pixel-to-feet conversions refer to a consistent coordinate frame across all three cameras (see the source of error discussed in Section **3.4.2**). Figures throughout Chapter 4 show more examples of the plots generated by this script for data from robots driving in a figure-eight pattern around the ramps (as per the NIST Standard Test Methods).

Chapter 4

Applications

The applications currently envisioned for the data acquisition system fall into three general themes. These are: 1) the consistency of laps (the current NIST Robot Test Methods metric for endurance) across different robots and operators, 2) trends in operator performance with time, and 3) relationships between robot performance in the arena and in the field. Because enough data has not yet been collected to make statements about these applications with statistical significance, the discussion here intends only to show that the camera tracking system can be used to investigate these themes, and to prove that, based on the data that has been collected thus far, there is intellectual merit in investigating further.

Analysis is made using the data from three testing events, conducted in accordance with the NIST Test Methods. The first two events were conducted with the Talon robot, originally developed by Foster-Miller (which has since become a subsidiary of Qinetic North America). Pictured in Figure **4-1**, the Talon is a tracked robot of floor area approximately 3'x1.5' and weight of about 150 lbs [5]. Data was taken for two different drivers, (referred to as "Driver 1" and "Driver 2" in this document) operating the Talon.

Figure **4-1**: Talon robot [6]

The third test was conducted with the Bombot, sold by Azimuth, Inc. Pictured in Figure **4-2**, the Bombot is a front-wheel steered, four-wheel drive robot of floor area approximately 1.5'x1' and weight of about 30 lbs [5]. It is much more agile than the Talon in terms of acceleration, but cannot execute the zero-radius turns of a tracked robot, and therefore must take corners in much wider arcs. Data was taken for Driver 1 operating the Bombot.

Figure **4-2**: Bombot [7]

4.1 Data Presentation

Using the script discussed in Section 3.5, measurements of robot position, taken at an average of around 12 Hz, were plotted for all three testing events in both pixels and feet. Testing event 1 consists of 27 laps of the Talon being directed in a figure-eight pattern within the arena by Driver 1, and is shown in Figure **4-3**.

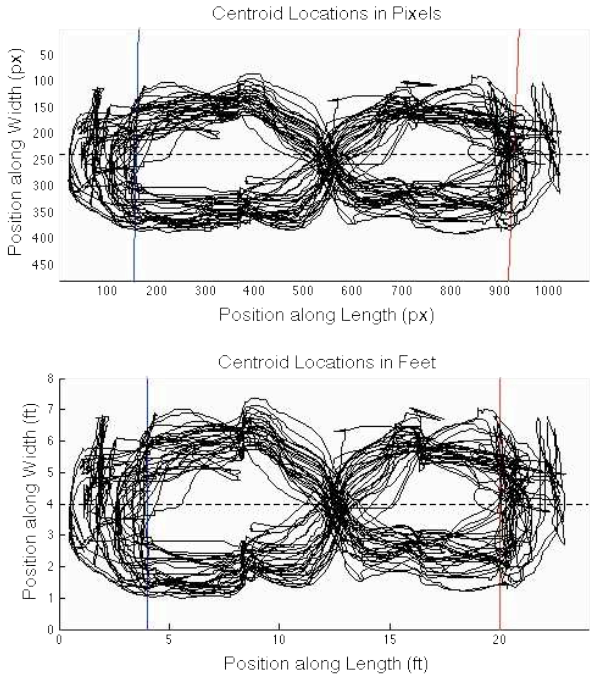


Figure **4-3**: Position measurements for Talon, Driver 1 (testing event 1)

Testing event 2 consists of 21 laps of the Talon being directed in a figure-eight pattern around within arena by Driver 2, and is shown in Figure **4-4**.

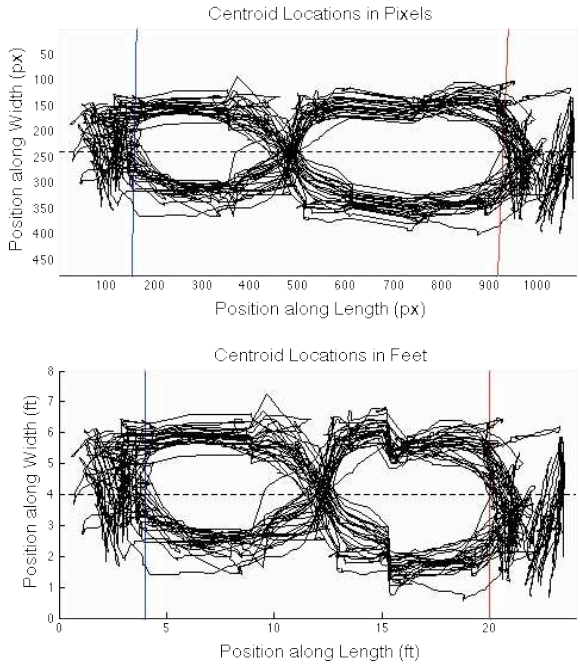


Figure **4-4**: Position measurements for Talon, Driver 2 (testing event 2)

Testing event 3 consists of 25 laps of the Bombot being driven in a figure-eight pattern within the arena by Driver 1, and is shown in Figure **4-5**.

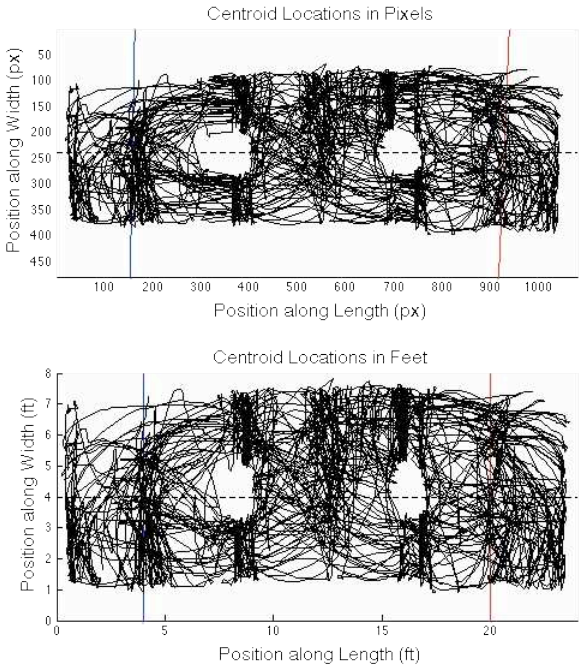


Figure **4-5**: Position measurements for Bombot, Driver 1 (testing event 3)

The distance and time data from each testing event can also be organized by lap. This is presented in Figures **4-6**, **4-7**, and **4-8**. A linear best-fit regression is overlaid on each plot.

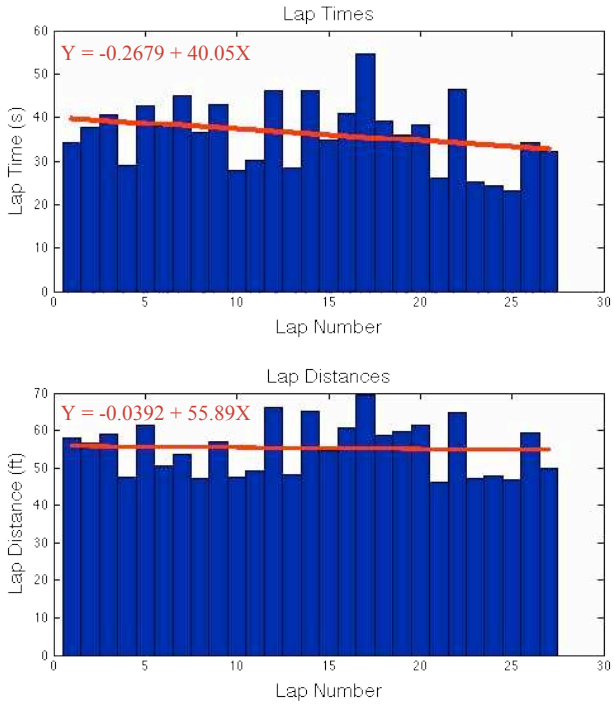


Figure **4-6**: Data by lap for Talon, Driver 1 (testing event 1)

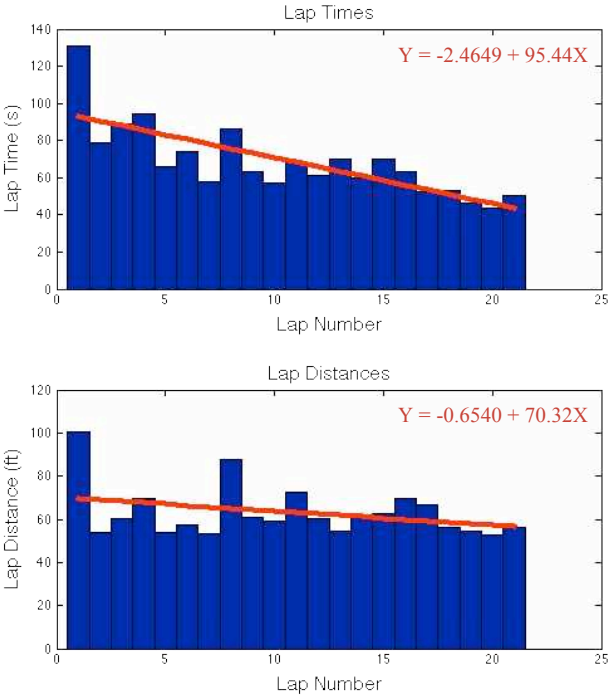


Figure **4-7**: Data by lap for Talon, Driver 2 (testing event 2)

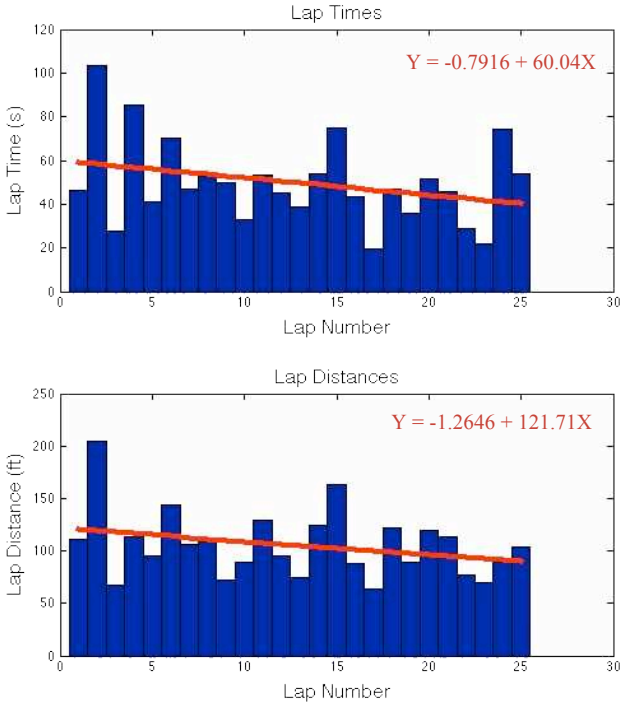


Figure **4-8**: Data by lap for Bombot, Driver 1 (testing event 3)

The data collected for all three testing events is summarized in Table **4-1**, below, along with some additional statistics.

Table **4-1**: Testing event data

TESTING EVENT	1	2	3
Robot	Talon	Talon	Bombot
Driver	1	2	1
Lap Count	27	21	25
Average Distance/Lap (ft)	55.34	63.13	105.27
Average Time/Lap (s)	36.30	68.32	49.75
Average Speed (mph)	1.039	0.630	1.443
R-value between lap time and lap distance	0.8634	0.7467	0.8133
R^2 value between lap distance and linear best fit	0.0385	0.1157	0.0275
R^2 value between lap times and linear best fit	0.0004	0.5987	0.1216

The sections below explain how this data can be applied to learn about the nature of the NIST test methods, robot operators, and the robots themselves.

4.2 Consistency of Laps

The consistency of laps as a metric of robot endurance can be explored in terms of its consistency both across different operators and across different robots.

4.2.1 Different Operators

Under the operation of Driver 2, the Talon traversed an average of about 8' more per lap than under Driver 1, but this could be explained through measurement and statistical error. It is more telling to compare the paths taken by each driver (Figures **4-3** and **4-4**). One significant difference between the two figures occurs along the right edge of each, where Driver 2 (Figure **4-4**) exhibits much higher activity, appearing to have traversed in straight lines back and forth along the width of the track many times. This occurred because Driver 2 had much less prior experience, both with driving robots within the arena and with the Talon specifically, than Driver 1. Under Driver 2's direction, the robot on several occasions became "stuck" between the bay wall and the upward slope of the ramps at one end of the arena. Figure **4-9** provides a camera image from one such instance.

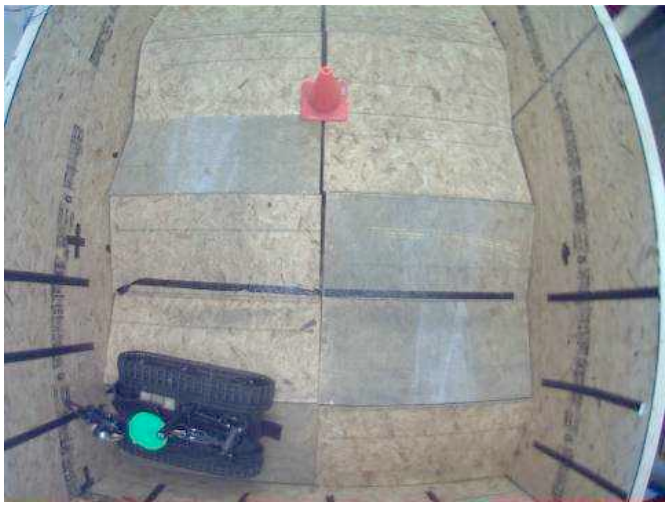


Figure **4-9**: Talon "stuck" between wall and ramps

From this position within the arena, attempting to turn the Talon to the right results in its back end hitting the bay wall, causing the tracks to slip on the surface of the upward-angled surface of the ramp. Escaping this position required the driver to drive forward and backwards along the width of the arena several times, increasing both the distance and time required to complete the lap. Relying on previous experiences operating the Talon in the arena, Driver 1 was able to avoid this occurrence entirely.

There is a much more significant difference in the average time to complete a lap between the two operators. Driver 1, at a much higher level of familiarity than Driver 2, drove both at higher speeds and in a smoother pattern, correcting the angle of the robot without reducing the throttle. Driver 2 often stopped the robot completely in order to adjust its trajectory, which slows the traversal. Therefore, this difference between operators is also largely a consequence of their experience level.

In summary, the consistency of laps across different drivers likely depends closely upon the familiarity of those operators with the robot and the arena. Because most formal NIST testing is conducted with "expert" drivers at the controls, NIST data for different operators of the same robot should be fairly consistent, especially in terms of the distance traversed per lap, which is treated by NIST as a much more valuable measurement of robot capability than the time per lap. Aiding in this consistency is the requirement that robots must enter the endzone at each end of the arena in order to receive credit for a lap, which is a relatively recent addition to the NIST Test Methods meant to prevent drivers from turning sharply around the pylons in order to reduce the robot's energy consumption per lap and therefore record more laps on one charge of batteries. Implementation of the distance tracking system could eventually eliminate the need for this requirement, since lap counting would no longer be the primary method of estimating the distance traversed.

4.2.2 Different Robots

Comparing testing events 1 and 3, in which the same operator drove the Talon and Bombot respectively for 25 or more laps in the arena, allows the consistency of laps as a measure of distance traveled to be explored across different robots. The data suggests that different robots may in fact travel very different distances in order to complete a lap. The Bombot, without the advantage of zero-radius turns, was not as maneuverable within the confines of the arena, and often had to be guided forward and back through lengthy multi-point turns. The robot's light weight, high center of gravity, and low torque at the steering motor causes interaction with the ramps to often reorient its direction of travel, which requires the traversal of extra distance to correct. Despite having a comparable average time per lap relative to testing with the Talon due to its much greater acceleration and top speed, the Bombot drove almost twice as far per lap on average than the Talon.

While data collected with more robots and operators should be conducted in order to validate these findings, based off just three testing events, early indications are that the distance traveled per lap is inconsistent across different robots.

4.3 Trends in Operator Performance

Although the data shows operators performing better with time, both in terms of distance traversed and time per lap, as exhibited by the negative slope of the trend lines in Figure **4-6**, **4-7**, and **4-8**, the R^2 values between the data and these trend lines tend to be very small, suggesting low likelihood of measurable improvement in performance with laps. However, this may be because both Drivers (especially Driver1) had some level of prior experience with the robots and arena. It is possible that data for completely "green" operators would show significant

improvement with time initially over the course of a testing event, and eventually converge to an essentially constant level of performance.

One intuitive observation that the data does support is that the time and distance taken to complete laps are correlated. This is supported by the relatively high R-values found in each testing event between the time to complete and distance to traverse each lap. In other words, if an operator takes a relatively long time to direct a robot around a particular lap, the robot probably also traversed a relatively long distance over the course of that lap.

4.4 Relationships to the Field

Imagine that an emergency response robot technician would like to use a particular robot to inspect a nuclear plant. To do this, the robot must be able to drive for one mile round trip over level pavement on a full charge of batteries. If NIST test results indicate that the robot can complete 100 laps around the endurance arena with fully charged batteries, can it complete this task? Although the development of the camera tracking system facilitates a significant step in answering this question (the conversion from laps to distance as a metric), a conversion factor would be required that allows distance traversed across the ramps of the arena to be used to estimate the distance traversable on level pavement, on stairs, etc. As discussed in Chapter 5, further development of this application for the tracking system forms the most significant need for future work.

Chapter 5

Conclusions

This thesis presents the fabrication of a NIST test arena, development of a tracking system for robots within the arena, and applications for that system. Although measurements of distances traversed by the system are accurate to within a few percent of their true value, there are still several improvements that could be made to increase its quality and capabilities. Even more importantly, more work is required in applying the system to learn about the performance of emergency response robots and their operators.

5.1 Recommendations for Future Work on the Tracking System

One current issue with the algorithms that needs to be addressed is that the measured position of the fiducial can vary slightly with the mode of operation of the code. When images are set to be plotted every iteration (the mode used when processing the data presented previously), entire images are corrected for camera distortion before the centroid of the fiducial is found. When images are not to be plotted (the most computationally efficient method of running the code), entire images are not corrected for distortion—only the individual pixel at the centroid of the fiducial is corrected. However, because barrel distortion corrections slightly change the shape of the fiducial (and therefore the location of its centroid), these two methods are not exactly the same. A better way to compute fiducial locations when not plotting would be to correct the part of the image immediately surrounding the last known location of the fiducial for distortion before searching for its new location. The addition of distortion corrections on a segment of an image (not just one pixel) would slow the speed of the algorithms slightly; however, this decrease

in computational efficiency would accompany an increase in the accuracy and precision of the system when users want to process data with speed.

While the system does enforce that the robot enters the endzones on each side of the arena in order to receive credit for a lap, it does not currently check whether the robot encircles the pylons that define the required figure-eight pattern. Verifying that the operator directs the robot in accordance with the test requirements would make the system even more useful as an automated way of conducting data collection during NIST robot testing events.

Another way in which the tracking system could be improved is to model more of the geometry of the arena in order to correct measurements for camera perspective. As discussed in section 3.4, if the locations of each ramp segment could be found, the height of the fiducial could be estimated and its measured position corrected to account for the angled perspective of the cameras. Mapping the ramps could also allow the algorithms to impose a more consistent coordinate frame between cameras, putting less dependency upon users to conduct proper calibrations in order to have high levels of accuracy.

The testing system could be made easier to set up and less intrusive to robots by eliminating the need to use a fiducial. Using a technique called background subtraction, the location of a robot within an arena could be found by detecting differences between the current images and a reference image. The reference image would be of the arena with no robots loaded in, so that the only difference between the two images is the presence of a robot.

One final improvement to the testing system is to enable it to track robots in the dark. One of the variable conditions imposed by NIST test methods is the presence of light in the arena, in the absence of which robots and their drivers must use night vision technology to operate. The tracking system could be adapted to accommodate these conditions by locating robots using infrared (IR) light as a fiducial, or by background subtraction using robots' on-board IR headlights.

5.2 Recommendations for Future Work on Applications of the System

As stated throughout Chapter 4, data from many more testing events must be collected in order to support the existing evidence that robot operators of "expert" skill level are likely to record similar distances traversed and times per lap, but that different robots exhibit very dissimilar distances traversed and times per lap. More data, especially of individuals with no prior experience operating robots in the arena, might also reveal trends in operator performance over the duration of a test. Tests of long duration might reveal not only the learning curve for effective driving, but also whether operators decrease in performance after extended periods of time. This could suggest an optimal work shift time for emergency response robot technicians.

Perhaps the most significant future application for the tracking system is its use in connecting a robot's arena endurance data to its field performance. This would allow data collected in the arenas to comment on performance outside the arena walls, facilitating a goal of the NIST Test Methods to improve the understanding of robot deployment capabilities. Researchers at the Penn State Applied Research Lab are already incorporating power consumption data into the measurements taken of robots within the NIST arena, but testing in which both power and distance measurements are made simultaneously has yet to be conducted. Combining these two measurements would allow the distance per unit energy to be measured at variable robot speeds, which could potentially serve as a scale factor for translating the results of NIST testing to performance in the field.

5.3 Final Remarks

Although there is much work still to be done, the goals of this thesis—to develop an accurate robot tracking system for the NIST Robot Endurance Test Method and demonstrate its applicability to several areas of interest in the emergency response robot domain—have been achieved. These contributions will advance our understanding of robot technology, taking additional steps on the path to a future in which robots can perform with increasing effectiveness hazardous tasks that humans cannot safely undertake.

Appendix A

Instructions for Collecting Data

A.1 Using Python Code to Save Images to File

1. Turn on the Ubuntu computer and verify that the correct software is installed. The code was developed for Unbuntu 12.04 running Python 2.7. GStreamer and v4l2loopback must also be installed.
2. Set the IP configuration of the computer to network with the cameras (camera IP addresses are 172.16.1.1, 172.16.1.2, and 172.16.1.3).
2. Plug into the Ethernet switch connected to the cameras and verify proper communications by visiting each camera's IP address in a web browser.
3. Open "Collect_Test_Images.py" -- there are two variables that can be edited:

    Freq: The approximate frame rate (in fps) at which you want to record. You can also comment the sleep function at the end of the script to record at the maximum possible frame rate.

    BasePath: The directory to which images will be saved. The script automatically creates a file for each camera at the end of this path.

4. Open a new terminal window.
5. Navigate to the current directory (the location of the code files) by typing:

    "cd <path to the directory containing the files>" (ex. cd /home/mycomputer/Desktop), and pressing enter.

6. If running for the first time you may also need to make the scripts executable. Do this by running the command: "chmod +x <filename>

7. Run the command: "sudo ./MakeVideos.sh"
8. Run the command "sudo ./Axis_GStreamer_1.sh"
9. Open a new terminal window, navigate to the current directory (see Step 5) and run the command: "sudo ./Axis_GStreamer_2.sh"
10. Open a new terminal window, navigate to the current directory (see Step 5) and run the command: "sudo ./Axis_GStreamer_3.sh"
11. Open a new terminal window, navigate to the current directory (see Step 5) and run the command: "python Collect_Test_Images.py fps"
12. After a brief pause, images will begin to be saved to file. The frame rate at which images are saved is periodically printed to the terminal window. To stop collecting images press “control” and “c” on the keyboard at the same time.

A.2 Using MATLAB Code to Process Images from File

1. Move the folders with images from each camera into a directory named “images_PY” in the MATLAB current directory
2. Open FcnInitTestConditions.m and check the value of the three flag variables. To load images from file, FlagLive must be 0.
3. If running for the first time, open ScriptCalibrate.m and follow the on-screen directions to create the necessary calibration files. Barrel distortion parameters must be stored in FcnInitCamParams.m in order to conduct distortion calibrations. Also, distorted images from each camera must be in the current directory and titled “calib_im_1” , “calib_im_2” , etc.
4. If the fiducial is not found, adjust the formula used to make the mask in FcnMask_Color.m
5. If the camera images need to be cropped differently to remove overlap, this can be adjusted at the bottom of FcnUndistort.m if FlagPlot =1, and in FcnGetImage_Select.m if FlagPlot = 0.

Appendix B

Python Code

B.1 Collect_Test_Images.py

```
#!/usr/bin/env python

import cv
from time import time, sleep
import sys
import os

cv.NamedWindow('Fixed', cv.CV_WINDOW_NORMAL)

    sleep(1.5)

    Freq = 10.0
    BasePath = "/media/UNTITLED/GStreamer/"

    pathname_1 = BasePath + "cam_1/"
    if not os.path.exists(pathname_1):
        os.mkdir(pathname_1)

    pathname_2 = BasePath + "cam_2/"
    if not os.path.exists(pathname_2):
        os.mkdir(pathname_2)

    pathname_3 = BasePath + "cam_3/"
    if not os.path.exists(pathname_3):
        os.mkdir(pathname_3)

    #@param argv[1]: fps to show frame per second
    if __name__ == '__main__':
        show_fps=False
        if len(sys.argv) >=2 and sys.argv[1] == "fps":
            show_fps = True

        #See if frame rate is shown
        if show_fps:
            frame=0
            start = time()

        #Create video capture devices
        capture_1 = cv.CaptureFromCAM(0)
        capture_2 = cv.CaptureFromCAM(1)
```

```
41.         capture_3 = cv.CaptureFromCAM(2)
42.
43.         Start_Time = time()
44.         #Run publishing image until this package is shot down.
45.         while 1:
46.
47.             Current_Time = time() - Start_Time
48.             Current_Time_Str = "%f" % Current_Time
49.
50.             New_Image = cv.QueryFrame(capture_1)
51.             cv.SaveImage(pathname_1+Current_Time_Str+".jpg",New_Image)
52.
53.             New_Image = cv.QueryFrame(capture_2)
54.             cv.SaveImage(pathname_2+Current_Time_Str+".jpg",New_Image)
55.
56.             New_Image = cv.QueryFrame(capture_3)
57.             cv.SaveImage(pathname_3+Current_Time_Str+".jpg",New_Image)
58.
59.             #Show frame rates
60.             if show_fps:
61.                 frame=frame+1
62.                 end = time()
63.                 if (end-start) >= 1.0:
64.                     print frame/(end-start)
65.                     frame=0
66.                     start=time()
67.
68.             sleep(1/Freq)
```

B.2 MakeVideos.sh

```
modprobe v4l2loopback devices=3
```

B.3 Axis_GStreamer_1.sh

```
gst-launch -vet  souphttpsrc location=http://172.16.1.1/axis-
cgi/mjpg/video.cgi?resolution=480x360 timeout=5 ! jpegdec ! v4l2sink device=/d
ev/video0
```

B.4 Axis_GStreamer_2.sh

```
gst-launch -vet  souphttpsrc location=http://172.16.1.2/axis-
cgi/mjpg/video.cgi?resolution=480x360 timeout=5 ! jpegdec ! v4l2sink device=/d
ev/video1
```

B.5 Axis_GStreamer_3.sh

```
gst-launch -vet  souphttpsrc location=http://172.16.1.3/axis-
cgi/mjpg/video.cgi?resolution=480x360 timeout=5 ! jpegdec ! v4l2sink device=/d
ev/video2
```

Appendix C

MATLAB Code

C.1 ScriptCalibrate.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);

% 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);
```

C.2 ScriptCollect_Test_Images.m

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

% There may be some problems with the string length of the image names
% generated by this script when they are read in FcnGetTimestamps.  This is
% because that function was designed to process images generated using
% Python.


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
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 every camera
            for CamNum=1:length(IP)

                % Get an 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
```

C.3 ScriptLap_and_Dist_Tracking.m

```
% This script conducts lap counting and distance tracking for NIST robot testing methods
on a green 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, 2013, 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 ] = 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 );

% 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 );
            % 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
```

C.4 ScriptSort_Data.m

```
% This script sorts data saved to file by ScriptLap_and_Dist_Tracking and
% displays some useful statistics

clear
clc

%% Load the data

% Data is stored in this order:
% 1) Iter, 2+3) CentroidPX_Current 4+5) CentroidFT_Current 6) TotalTime 7) TotalLaps 8)
TotalDist

% 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);

% Get variables from file
listing = dir('DataLog*.mat');

% Variable for the iteration we're currently loading
Iter = 1;

% For every data file
for n=1:size(listing)

    % Load the data
    load(listing(n).name)

    % Add its contents to a list
    size = length(DataLog);
    DataLog_All(Iter:Iter+size-1,1:8) = DataLog;

    % Update the iteration tracker
    Iter = Iter+size;
end

%% Plot the fiducial location for every iteration in pixels and feet

MaxIter = max(DataLog_All(:,1));

% Prep endzone data
load('DataCalibEndzones.mat')

% 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:CamRes(1);
Lbound = Lm*X+Lb;
Rbound = Rm*X+Rb;

% Plot the data in pixels
figure(1)
subplot(2,1,1)
hold on
emptyim = imrotate(ones(CamRes(2)*NumCams,CamRes(1),3),90);
imagesc(emptyim)
plot( [0,CamRes(2)*NumCams],[CamRes(1)/2,CamRes(1)/2],'--k'); % Plot the midline of the
image
plot( Lbound,X,'b' ); % Plot the endzone locations
plot( Rbound,X,'r') ;
plot(DataLog_All(1:MaxIter,2),DataLog_All(1:MaxIter,3),'k') % Plot the data itself
set(gca,'YDir','reverse')
axis([1 CamRes(2)*NumCams 1 CamRes(1)])
title('Centroid Locations in Pixels','Fontsize',15)
xlabel('Position along Length (px)','Fontsize',15)
ylabel('Position along Width (px)','Fontsize',15)

% Plot the data in feet
subplot(2,1,2)
hold on
plot([0 24],[4,4],'--k') % Plot the midline
plot([4 4],[0 24],'b') % Plot the left endzone
plot([20 20],[0 24],'r') % Plot te right endzone
plot(DataLog_All(1:MaxIter,4),DataLog_All(1:MaxIter,5),'k') % Plot the data itself
axis([0 24 0 8])
title('Centroid Locations in Feet','Fontsize',15)
xlabel('Position along Length (ft)','Fontsize',15)
ylabel('Position along Width (ft)','Fontsize',15)
```

```
%% For finding stationary fiducial error

% Get some statistics
X_PX_mean = mean(DataLog_All(1:MaxIter,2))
Y_PX_mean = mean(DataLog_All(1:MaxIter,3))

X_PX_std = std(DataLog_All(1:MaxIter,2))
Y_PX_std = std(DataLog_All(1:MaxIter,3))

X_FT_mean = mean(DataLog_All(1:MaxIter,4))
Y_FT_mean = mean(DataLog_All(1:MaxIter,5))

X_FT_std = std(DataLog_All(1:MaxIter,4))
Y_FT_std = std(DataLog_All(1:MaxIter,5))

%% Sort the data by time and distance of each lap

% Get the first lap count
LastLap = DataLog(1,7);
FlagFirstLap = 0;

% Counter for rows
m = 1;

% For every iteration
for n=2:length(DataLog_All)

    % When the first lap is found
    if DataLog_All(n,7) == 1 && FlagFirstLap == 0

        % Store starting time and distance
        LastLap = 1;
        LastTime = DataLog_All(n,6);
        LastDist = DataLog_All(n,8);
        FlagFirstLap = 1;

    end

    % If a new lap has been completed
    if DataLog_All(n,7) > LastLap && rem(DataLog_All(n,7),1) == 0

        % Get elapsed time and distance
        DataLog_Laps(m,1) = LastLap;
        DataLog_Laps(m,2) = DataLog_All(n,6)-LastTime;
        DataLog_Laps(m,3) = DataLog_All(n,8)-LastDist;

        % Update row counter
        m = m+1;

        % Replace as last time and distance
        LastTime = DataLog_All(n,6);
        LastDist = DataLog_All(n,8);
        LastLap = DataLog_All(n,7);
    end
end

%% Plot data by laps

figure(2)
subplot(2,1,1)
bar(DataLog_Laps(:,2),'hist');
title('Lap Times','Fontsize',15)
xlabel('Lap Number','Fontsize',15)
ylabel('Lap Time (s)','Fontsize',15)
axis([0 length(DataLog_Laps) 0 140])

subplot(2,1,2)
bar(DataLog_Laps(:,3),'hist');
title('Lap Distances','Fontsize',15)
xlabel('Lap Number','Fontsize',15)
```

```
ylabel('Lap Distance (ft)','Fontsize',15)
axis([0 length(DataLog_Laps) 0 140])

% Plot best fit lines
subplot(2,1,1)
hold on
p1 = polyfit(DataLog_Laps(:,1),DataLog_Laps(:,2),1);
y1 = DataLog_Laps(:,1)*p1(1,1) + p1(1,2);
plot(DataLog_Laps(:,1),y1,'r','linewidth',3)

subplot(2,1,2)
hold on
p2 = polyfit(DataLog_Laps(:,1),DataLog_Laps(:,3),1);
y2 = DataLog_Laps(:,1)*p2(1,1) + p2(1,2);
plot(DataLog_Laps(:,1),y2,'r','linewidth',3)


%% Report some useful numbers

TotalDist = DataLog_All(MaxIter,8)
TotalLaps = max(DataLog_Laps(:,1))

corr_laps_dist = corr(DataLog_Laps(:,2),DataLog_Laps(:,3))
mean_lap_time = mean(DataLog_Laps(:,2))
mean_lap_dist = mean(DataLog_Laps(:,3))

average_speed = sum(DataLog_Laps(:,3)) / sum(DataLog_Laps(:,2))
```

C.5 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
```

C.6 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
```

C.7 FcnFixDistort

```
function [ im ] = FcnFixDistort( im,Cam )

% This function corrects for camera barrel distortion.  The algorithm used is taken
% directly from the Matlab Camera Calibration Toolbox.


fc = Cam.fc;
alpha_c = Cam.alpha_c;
cc = Cam.cc;
kc = Cam.kc;

KK = [fc(1) alpha_c*fc(1) cc(1);0 fc(2) cc(2) ; 0 0 1];

I1 = im(:,:,1);
```

```
I2 = im(:,:,2);
I3 = im(:,:,3);

[I11] = FcnFixDistort_Rect(double(I1),eye(3),fc,cc,kc,alpha_c,KK);

[I22] = FcnFixDistort_Rect(double(I2),eye(3),fc,cc,kc,alpha_c,KK);

[I33] = FcnFixDistort_Rect(double(I3),eye(3),fc,cc,kc,alpha_c,KK);

im(:,:,1) = I11;
im(:,:,2) = I22;
im(:,:,3) = I33;

im=uint8(im);

end
```

C.8 FcnFixDistort_Apply.m

```
function [xd,dxddk] = FcnFixDistort_Apply(x,k)

% This function comes directly from the Matlab Camera Calibration Toolbox
% and is used in the correction of barrel distortions.
% It is called by FcnFixDistort_Rect.


% Complete the distortion vector if you are using the simple distortion model
length_k = length(k);
if length_k <5 ,
    k = [k ; zeros(5-length_k,1)];
end;


[m,n] = size(x);

% Add distortion:

r2 = x(1,:).^2 + x(2,:).^2;

r4 = r2.^2;

r6 = r2.^3;


% Radial distortion:

cdist = 1 + k(1) * r2 + k(2) * r4 + k(5) * r6;

if nargout > 1,
    dcdistdk = [ r2' r4' zeros(n,2) r6'];
end;


xd1 = x .* (ones(2,1)*cdist);

coeff = (reshape([cdist;cdist],2*n,1)*ones(1,3));

if nargout > 1,
    dxd1dk = zeros(2*n,5);
    dxd1dk(1:2:end,:) = (x(1,:)'*ones(1,5)) .* dcdistdk;
    dxd1dk(2:2:end,:) = (x(2,:)'*ones(1,5)) .* dcdistdk;
end;


% tangential distortion:
```

```
a1 = 2.*x(1,:).*x(2,:);
a2 = r2 + 2*x(1,:).^2;
a3 = r2 + 2*x(2,:).^2;

delta_x = [k(3)*a1 + k(4)*a2 ;
   k(3) * a3 + k(4)*a1];

aa = (2*k(3)*x(2,:)+6*k(4)*x(1,:))'*ones(1,3);
bb = (2*k(3)*x(1,:)+2*k(4)*x(2,:))'*ones(1,3);
cc = (6*k(3)*x(2,:)+2*k(4)*x(1,:))'*ones(1,3);

if nargout > 1,
    ddelta_xdk = zeros(2*n,5);
    ddelta_xdk(1:2:end,3) = a1';
    ddelta_xdk(1:2:end,4) = a2';
    ddelta_xdk(2:2:end,3) = a3';
    ddelta_xdk(2:2:end,4) = a1';
end;

xd = xd1 + delta_x;

if nargout > 1,
    dxddk = dxd1dk + ddelta_xdk ;
    if length_k < 5,
        dxddk = dxddk(:,1:length_k);
    end;
end;


return;

% Test of the Jacobians:

n = 10;

lk = 1;

x = 10*randn(2,n);
k = 0.5*randn(lk,1);

[xd,dxddk] = apply_distortion(x,k);


% Test on k: OK!!

dk = 0.001 * norm(k)*randn(lk,1);
k2 = k + dk;

[x2] = apply_distortion(x,k2);

x_pred = xd + reshape(dxddk * dk,2,n);


norm(x2-xd)/norm(x2 - x_pred)
```

C.9 FcnFixDistort_Rect.m

```
function [Irec] = FcnFixDistort_Rect( I,R,f,c,k,alpha,KK_new )

% This function comes directly from the Matlab Camera Calibration Toolbox
% and is used in the correction of barrel distortions.
% It is called by FcnFixDistort and calls FcnFixDistort_Apply


if nargin < 5,
   k = [0;0;0;0;0];
   if nargin < 4,
      c = [0;0];
      if nargin < 3,
         f = [1;1];
         if nargin < 2,
            R = eye(3);
            if nargin < 1,
               error('ERROR: Need an image to rectify');
            end;
         end;
      end;
   end;
end;


if nargin < 7,
   if nargin < 6,
        KK_new = [f(1) 0 c(1);0 f(2) c(2);0 0 1];
   else
    KK_new = alpha; % the 6th argument is actually KK_new
   end;
   alpha = 0;
end;



% Note: R is the motion of the points in space
% So: X2 = R*X where X: coord in the old reference frame, X2: coord in the new ref frame.


if ~exist('KK_new'),
   KK_new = [f(1) alpha*f(1) c(1);0 f(2) c(2);0 0 1];
end;


[nr,nc] = size(I);

Irec = 255*ones(nr,nc);

[mx,my] = meshgrid(1:nc, 1:nr);
px = reshape(mx',nc*nr,1);
py = reshape(my',nc*nr,1);

rays = inv(KK_new)*[(px - 1)';(py - 1)';ones(1,length(px))];


% Rotation: (or affine transformation):

rays2 = R'*rays;

x = [rays2(1,:)./rays2(3,:);rays2(2,:)./rays2(3,:)];
```

```
% Add distortion:
xd = FcnFixDistort_Apply(x,k);


% Reconvert in pixels:

px2 = f(1)*(xd(1,:)+alpha*xd(2,:))+c(1);
py2 = f(2)*xd(2,:)+c(2);


% Interpolate between the closest pixels:

px_0 = floor(px2);


py_0 = floor(py2);
py_1 = py_0 + 1;


good_points = find((px_0 >= 0) & (px_0 <= (nc-2)) & (py_0 >= 0) & (py_0 <= (nr-2)));

px2 = px2(good_points);
py2 = py2(good_points);
px_0 = px_0(good_points);
py_0 = py_0(good_points);

alpha_x = px2 - px_0;
alpha_y = py2 - py_0;

a1 = (1 - alpha_y).*(1 - alpha_x);
a2 = (1 - alpha_y).*alpha_x;
a3 = alpha_y .* (1 - alpha_x);
a4 = alpha_y .* alpha_x;

ind_lu = px_0 * nr + py_0 + 1;
ind_ru = (px_0 + 1) * nr + py_0 + 1;
ind_ld = px_0 * nr + (py_0 + 1) + 1;
ind_rd = (px_0 + 1) * nr + (py_0 + 1) + 1;

ind_new = (px(good_points)-1)*nr + py(good_points);



Irec(ind_new) = a1 .* I(ind_lu) + a2 .* I(ind_ru) + a3 .* I(ind_ld) + a4 .* I(ind_rd);



return;


% Convert in indices:

fact = 3;

[XX,YY]= meshgrid(1:nc,1:nr);
[XXi,YYi]= meshgrid(1:1/fact:nc,1:1/fact:nr);

%tic;
Iinterp = interp2(XX,YY,I,XXi,YYi);
%toc

[nri,nci] = size(Iinterp);


ind_col = round(fact*(f(1)*xd(1,:)+c(1)))+1;
ind_row = round(fact*(f(2)*xd(2,:)+c(2)))+1;

good_points = find((ind_col >=1)&(ind_col<=nci)&(ind_row >=1)& (ind_row <=nri));
```

C.10 FcnGetCalibrations.m

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

C.11 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
```

C.12 FcnGetImage_Select.m

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

% 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 = 150;  % 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 on)
    switch Cams(CamIndex)
        case 1
            newim(1:480,330:360,:) = 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 compiled
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
```

C.13 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);
    case 4
        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
```

C.14 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 )

% Ths 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 );
    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 zeroth iteration OR the plot flag is ON...
if Iter==0 || FlagPlot == 1
    % Undistort the entire image
    im = FcnUndistort(im,DistortionMapping,NumCams,CamRes);
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(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(1)
        end

        % If the object is found, plot it
        FcnPlot(
im,Mask,CalibEndzones,CentroidPX_Current,CentroidFT_Current,CamRes(2)*NumCams,CamRes(1),F
lagObjFound );
    end
end
```

C.15 FcnGetTimestamps.m

```
function [ TimeStamps ] = FcnGetTimestamps()

% This function extracts 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
```

C.16 FcnInitCamParams.m

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

% This function initializes 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 = [343.737030 ; 337.834129]; % Focal lengths for each axis in pixels
cc1 = [234.556875 ; 199.708871]; % Image center for each axis
kc1 = [-0.418304 ; 0.192773 ; -0.000486 ; 0.002220 ; 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 = [440.187719 ; 392.214480]; % Focal lengths for each axis
cc2 = [225.518547 ; 176.499447]; % Image center for each axis
kc2 = [-0.540728 ; 0.250023 ; 0.010247 ; -0.006886 ; 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 = [335.882994 ; 305.973338]; % Focal lengths for each axis
cc3 = [244.737794 ; 153.294929]; % Image center for each axis
kc3 = [-0.375910 ; 0.149280 ; 0.008296 ; -0.010249 ; 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
```

C.17 FcnInitDistortCorrection.m

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

% 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(CamParam,NumCams); % Take new calibrations
end

end
```

C.18 FcnInitDistortCorrection_Calib.m

```
function FcnInitDistortCorrection_Calib(CamParam,NumCams)

% 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));

        if 1==0 % 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')


    % Flip the mapping
    for i=1:length(newlocation)
        DistortionMapping(newlocation(i,CamNum),CamNum) = i;
    end

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

    %% 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 = round(rand*(length(indices_to_chose_from)-1))+1;
            DistortionMapping(i,CamNum) =
DistortionMappingSparse(indices_to_chose_from(value),CamNum);
        end
    end

    %% 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
```

C.19 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
```

C.20 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);
            imlayer = reshape(imlayer(DistortionMapping(:,CamNum)),CamRes(2),CamRes(1));
            im(:,:,Dimension) = imlayer;
        end
        % Re-rotate the image segment
        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
```

C.21 FcnInitDistTrack_Get2Pts.m

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

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


% Prepare the figure
hold on
figure(1)
imagesc(im)
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
```

C.22 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
```

C.23 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
```

C.24 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 te 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
% Make this really big (~1 million) if not running live so the variable
% does not run out of space for storing data, otherwise ~300 should be good
Data2File = 1000000;

end
```

C.25 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;
else
    TimeStamps = 0;
end

end
```

C.26 FcnLogData.m

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

% This function saves data to 'DataLog'
% If running live, it does this 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);
```

C.27 FcnMask.m

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

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


% Define the minimum pixel area of the fiducial expected
MinSize = 180;

% 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 < 2000
        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
```

C.28 FcnMask_Color.m

```
function Mask = FcnMask_Color(im)

% This function creates a mask used to find for a pink fiducial
im = rgb2hsv(im);

% Good for low light
%I = -20*(im(:,:,1)-10*im(:,:,2));
%Mask = I/160>.80;

% Good for bright light
I = -5*(im(:,:,1))+(8*im(:,:,2))+5*(im(:,:,3));
Mask = (I/8)>0.90;
```

C.29 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(1)
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
```

C.30 FcnUndisort.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);
            case 4
                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);
            case 4
                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 on)
im(1:480,340:360,:) = 0;
im(1:480,620:720,:) = 0;


end
```

References

[1] E. Guizzo and E. Ackerman, "DARPA Robotics Challenge," *IEEE Spectrum*, Apr. 10, 2012. [Online]. Available: http://spectrum.ieee.org

[2] E. Messina et al., "Apparatus Assembly Guide for Standard Test Methods," NIST Intelligent Systems Division, Gaithersburg, MD, March 2013.

[3] "OptiTrack." Internet: http://www.naturalpoint.com/optitrack, [Apr. 7, 2013].

[4] *"Camera Calibration Chessboard."* [Online]. Available: http://docs.opencv.org

[5] D. Logan, "Optimization of hybrid power sources for mobile robotics through the use of allometric design principles and dynamic programming," Master's thesis, Dept. of Mech. and Nuclear Eng., The Pennsylvania State Univ., University Park, PA, 2010.

[6] "*Talon Robot.*" [Online]. Available: http://www.qinetiq.com

[7] (2007, Sept. 26). "*Bombot*." [Online]. Available: http://www.strategypage.com

ACADEMIC VITA

Herschel Pangborn

322 E. Irvin Ave.
State College, PA 16801
hpangborn@icloud.com

Education

B.S., Mechanical Engineering, 2013, The Pennsylvania State University, University Park, PA
Minor in Music Performance
International Engineering Certificate

Honors and Awards

- Dr. John P. Karidis Department Head's Award for Research Achievement in Mechanical Engineering 2013
- Louis A. Harding Memorial Scholarship, 2011-Present, Department of Mechanical Engineering
- Schreyer Honors Scholar, 2009-Present
- Dean's List, 2009-Present

Association Memberships

- Tau Beta Pi National Honors Society

Professional Experience

Undergraduate Researcher, Applied Research Laboratory Summer, 2011-Spring 2013
- Conducted robotics research for Penn State's Navy-sponsored Applied Research Laboratory under the supervision of Dr. Karl Reichard, as a member of the Intelligent Vehicles and Systems Group maintained by Dr. Sean Brennan

Teaching Intern, Dept. of Mechanical & Nuclear Engineering, Fall 2012
- Provided instructional aid to two class sections of ME 300: Engineering Thermodynamics, including holding regular office hours and exam review sessions
- Participated in a weekly seminar designed to help teaching interns develop the ability to provide engineering instruction and make ethically sound decisions within academia

Research Interests

- Control Systems
- Mechatronics