IMPLEMENTATION OF SAFE AND RELIABLE COVERAGE CONTROL FOR MULTIPLE MOBILE ROBOTS

BY

RICHARD A. REKOSKE

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Systems and Entrepreneurial Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Associate Professor Dušan M. Stipanović

ABSTRACT

This thesis presents some results relating to an implementation of a safe and reliable coverage control algorithm. The application was focused on implementation on differential drive robots developed at the University of Illinois at Urbana-Champaign. Control laws for coverage, avoidance, and proximity were applied to a multi-agent system of robots. The control laws were merged to provide coverage using the system while guaranteeing inter-agent proximity, inter-agent collision avoidance, and agent-environment collision avoidance. Circular regions were considered for avoidance. The performance and limitation of the application are examined. Practical considerations for implementation are discussed. The experimental platform consisted of a motion capture system, three differential drive robots with multiple sensing capabilities, and two supporting computers for the motion capture system and data visualization.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

The control and coordination of multiple autonomous vehicles has been the focus of much study. One of the most important issues when working with multiple autonomous agents is guaranteeing the safety of the operation, i.e. avoiding any collisions between the agents or between agents and their environment. In addition to this, it is also important to be able to combine this guaranteed safety with other objectives, such as maintaining short range communication networks or sensing a compact area. It has been presented in [1] that it is possible to combine these multiple control objectives in a manner that can be applied to a network of agents of arbitrary size, so long as the agents have dynamic models that are nonlinear yet affine in control. In this thesis, an implementation of the multi-objective problem where multiple differential drive robots are tasked with sensing an area while remaining collision free and maintaining reliable short range communication is demonstrated. The agents were not strictly required to remain in the compact domain, and the domain being sensed was not required to be a square area, but for this implementation a square area was sensed.

Chapter 2 of this thesis outlines the avoidance and proximity objective functions as well as their proposed control laws used in the implementation. The control laws presented in this chapter will be applied in Chapter 5, the experimental results chapter.

Chapter 3 will focus on the coverage control aspect of the agents' objectives. It will discuss how the sensing capabilities and coverage quality of the robots was modeled, and it will show how the control law for coverage is calculated for each robot. The control law shown in this chapter will be applied to the agents' specific dynamic models in Chapter 5.

Chapter 4 discusses the experimental testbed for the implementation. The algorithm can support an arbitrary number of agents and obstacles, but three agents and two obstacles were implemented. Additionally, while the algorithm does not strictly require a square compact domain to be sensed, a square planar domain was used. The implementation used the OptiTrack motion capture system, onboard sensors on the differential drive robots, and a local wireless network utilizing UDP (User Datagram Protocol) for communication.

Chapter 5 will present the specific dynamic models for the agents, the specific control laws used, and the specific parameters used in the implementation. The robots were modeled

using a unicycle model, the control laws developed in Chapters 2 and 3 were merged for application on the robots, and design parameters, such as the sensing radii of the robots, were chosen appropriately for this experiment. Additionally, this chapter also discusses the issue of determining accurate state variables over time. Due to issues with both the motion capture system and calculating position and orientations using dead reckoning, a Kalman filter was implemented to provide much smoother and reliable state estimations for the robots. Full motivation for this filter implementation is discussed. Finally, the results of the implementation are discussed. Multiple scenarios with different assumptions about the robots will be presented in this chapter.

The final chapter discusses the conclusions drawn from the implementation of the safe and reliable coverage control algorithm in this thesis. Ideas for further implementation are also presented.

CHAPTER 2
AVOIDANCE AND PROXIMITY CONTROL

Part of the experiment was focused on the objective of inter-agent, and agent-environment collision avoidance, as well the objective of inter-agent proximity. For this experiment using differential drive robots, which are nonlinear and affine in control, we can model the agents as [2]

$$\dot{x}_i = f_i(x_i, u_i) = g_i(x_i)u_i + h_i(x_i),\ \ x_i(0) = x_{io},\ \ \forall t \in [0, +\infty),\ \ \forall i \in N \tag{1}$$

where *N* is the number of agents, $\mathrm{N} = \{1, \dots, N\}$, $x_i \in \mathbb{R}^{n_i}$ is the state, $u_i \in \mathbb{R}^{m_i}$ is the control input, and x_{io} is a given initial condition for the *i*th agent. As stated in [1], the n_i-dimensional vector functions $f_i(\cdot,\cdot), i \in \mathrm{N}$ are assumed to be continuously differentiable with respect to both arguments. The objective functions and control laws developed for these tasks of avoidance and proximity are outlined in what follows.

2.1 Avoidance Objective Functions

The objective function for avoidance between the *i*th and *j*th agents is given by [3]:

$$v_{ij}^{a}(x) = \left(\min\left\{0, \frac{\left(x_i^p - x_j^p\right)^T P_{ij}\left(x_i^p - x_j^p\right) - R_{ij}^2}{\left(x_i^p - x_j^p\right)^T P_{ij}\left(x_i^p - x_j^p\right) - r_{ij}^2}\right\}\right)^2 \tag{2}$$

where $R_{ij} > r_{ij}$ are positive scalars and P_{ij} is a positive definite matrix for the pair (i, j). Similarly, the objective function for avoidance between the *i*th agent and the *l*th obstacle is given by:

$$v_{il}^{a}(x) = \left(\min\left\{0, \frac{\left(x_i^p - x_l^o\right)^T P_{il}\left(x_i^p - x_l^o\right) - R_{il}^2}{\left(x_i^p - x_l^o\right)^T P_{il}\left(x_i^p - x_l^o\right) - r_{il}^2}\right\}\right)^2 \tag{3}$$

where $R_{il} > r_{il}$ are positive scalars and P_{il} is a positive definite matrix for the pair (i, l). The gradients of the avoidance objective functions, in terms of the number of agents *i* and the combined number of agents and obstacles *j*, is given by [3]:

$$\frac{\partial v_{ij}^{a}}{\partial x_i^p} = \begin{cases} 0, & \text{if } \left\|x_i^p - \hat{x}_j\right\|_{P_{ij}} \geq R_{ij} \\ 4\dfrac{\left(R_{ij}^2 - r_{ij}^2\right)\left(\left(x_i^p - \hat{x}_j\right)^T P_{ij}\left(x_i^p - \hat{x}_j\right) - R_{ij}^2\right)}{\left(\left(x_i^p - \hat{x}_j\right)^T P_{ij}\left(x_i^p - \hat{x}_j\right) - r_{ij}^2\right)^3}\left(x_i^p - \hat{x}_j\right)^T P_{ij}, & \text{if } R_{ij} > \left\|x_i^p - \hat{x}_j\right\|_{P_{ij}} > r_{ij} \\ \text{not defined}, & \text{if } \left\|x_i^p - \hat{x}_j\right\|_{P_{ij}} = r_{ij} \\ 0, & \text{if } \left\|x_i^p - \hat{x}_j\right\|_{P_{ij}} < r_{ij} \end{cases} \tag{4}$$

where $\hat{x}_j$ refers to x_j^p if j corresponds to an agent and x_j^o if j corresponds to an obstacle, and $\|y\|_{P_{ij}} = \sqrt{y^T P_{ij} y}$ where y is a vector. The particular choices for the constants P_{ij}, R_{ij}, and r_{ij} from (2), and the choices for constants P_{il}, R_{il}, and r_{il} from (3) for the experiment are discussed in Chapter 5, the experimental results chapter. The parameters are chosen based off of the sensing capabilities of the agents, the dynamics of the agents, and the desired safety requirements for the experiment. Also, practical considerations for implementation of the gradient are discussed in Chapter 5.

2.2 Proximity Objective Functions

As stated in [1], it is assumed that the agents exchange information wirelessly over a network. Details regarding the proximity of agents and the exchange of information will be discussed in later chapters. The objective function, which is similar to those in [4], for proximity is given in [1] by:

$$v_{ij}^{p}\left(x_i^p, x_j^p\right) = \max\left\{0, \left\|x_i^p - x_j^p\right\|^2 - \hat{R}_{ij}^2\right\}^2 \tag{5}$$

where $\hat{R}_{ij}$ is the constant maximum desired distance between the *i*th and *j*th agents for reliable communication. The gradient for this objective function is given by:

$$\frac{\partial v_{ij}^{p}}{\partial x_i^p} = 4\max\left\{0, \left\|x_i^p - x_j^p\right\|^2 - \hat{R}_{ij}^2\right\}\left(x_i^p - x_j^p\right)^T \tag{6}$$

2.3 Avoidance and Proximity Control Functions

As described in [1], it is possible to find the approximation of the maximum for use in a Liapunov-type analysis for accomplishing multiple objectives using the following equation:

$$\bar{\rho}(\delta, a) = \sqrt[\delta]{\sum_{i=1}^{N} a_i^{\delta}} \tag{7}$$

where $\delta \in \mathbb{R}_+ = (0, +\infty)$, $a = [a_1, \ldots, a_N]^T \in \mathbb{R}_+^N$, and N is a positive integer. The preceding is a δ-norm when $\delta \in [1, +\infty)$. It was also shown that the overall goal of avoidance and proximity could be formulated as $v_i(t, x) \leq \epsilon$ where

$$v_i(\cdot) = \bar{\rho}_{\delta}\left(\gamma_{i1} v_{i1}(\cdot), \ldots, \gamma_{iN_i} v_{iN_i}(\cdot)\right) \tag{8}$$

where $\gamma_{ij} \leq {}^{1}/_{\epsilon_{ij}}$ when ϵ_{ij} is chosen appropriately such that $v_{ij} \leq \epsilon_{ij}$ for the avoidance and proximity objective functions from (2), (3), and (5). Finally, it was shown in [1] that the agents' dynamics and objective function gradients could produce a control vector of the form:

$$\hat{u}_i(x) = -\hat{k}_i g_i^T(x_i) \frac{\partial v_i(x)}{\partial x_i}^T \tag{9}$$

where $\hat{k}_i$ is a positive scalar gain. This application of this control vector for the experiment, as well as its combination with the control laws produced by the third objective of coverage, will be discussed in Chapter 5, the experimental results chapter.

CHAPTER 3
COVERAGE CONTROL

In addition to the control corresponding to avoidance and proximity, we desire a control law corresponding to the coverage of a given domain. For the control of agents when their dynamics are affine in control, an error function with a modified area integral for agents with overlapping nonuniform sensing capabilities that does not require the agents to remain inside of the sensing region was developed in [1]. This was useful in the application presented in this paper due to the fact that agents were allowed, without penalty, to exit the search area in order to avoid collisions with obstacles and other agents. An overview of the modeling of the cumulative coverage, the sensing of the robots, and the coverage error and control laws is given in this chapter.

3.1 Sensing

The sensing of the robots is modeled by the following function [5]:

$$S_i(p) = \frac{M_i}{R_i^4}\max\{0, R_i^2 - p\}^2 \Rightarrow S_i'(p) = -2\frac{M_i}{R_i^4}\max\{0, R_i^2 - p\} \tag{10}$$

where M_i is the peak sensing capability of the *i*th agent, and R_i is the *i*th agent's sensing radius. The choices for the sensing capabilities and the sensing radii for the robots will be discussed in Chapter 5. Also from [5], the cumulative sensing function is

$$Q(t, \tilde{x}) = \int_0^t \left(\sum_{i=1}^{N} S_i\left(\left\| x_i^p(\tau) - \tilde{x} \right\|^2 \right) \right) d\tau \tag{11}$$

where *N* is the number of agents, $\tilde{x} = [\tilde{x}_1, \tilde{x}_2]^T \in \mathbb{R}^2$ and $x_i^p(\cdot)$ is the planar position of the *i*th agent as a function of time.

3.2 Coverage Control Functions

As shown in [1], letting $h(w) = (\max\{0, w\})^3$, so that $h'(w) = 3(\max\{0, w\})^2$ and $h''(w) = 6\max\{0, w\}$, a possible coverage error function is

$$\tilde{\tilde{e}}(t) = a_0(t) + \sum_{i=1}^{N} a_i^T(t)\dot{x}_i^p(t) \tag{12}$$

where $\dot{x}_i^p(t) = \left[\dot{x}_{i1}^p(t), \dot{x}_{i2}^p(t)\right]^T$, $a_i(t) = [a_{i1}(t), a_{i2}(t)]^T$, and

$$a_0(t) = \iint_{\mathcal{D}} h''\big(C^* - Q(t,\tilde{x})\big)\left(\sum_{i=1}^{N} S_i\big(p_i(t,\tilde{x})\big)\right)\left(S^* - \sum_{i=1}^{N} S_i\big(p_i(t,\tilde{x})\big)\right)\phi(\tilde{x})d\tilde{x}_1 d\tilde{x}_2$$

$$a_{i1}(t) = 2\iint_{\mathcal{D}} h'\big(C^* - Q(t,\tilde{x})\big)S_i'\big(p_i(t,\tilde{x})\big)\big(x_{i1}^p(t) - \tilde{x}_1\big)d\tilde{x}_1 d\tilde{x}_2$$

$$a_{i2}(t) = 2\iint_{\mathcal{D}} h'\big(C^* - Q(t,\tilde{x})\big)S_i'\big(p_i(t,\tilde{x})\big)\big(x_{i2}^p(t) - \tilde{x}_2\big)d\tilde{x}_1 d\tilde{x}_2 \tag{13}$$

where $\mathcal{D}$ is the area to be covered, C^* is a positive constant that represents the desired quality of the coverage at a certain point, $\phi(\tilde{x})$ is a nonnegative scalar function used to incorporate preferences, S^* is a positive constant that satisfies $S^* > \sum_{i=1}^{N} M_i$, and $p_i(t,\tilde{x}) = \left\|x_i^p(t) - \tilde{x}\right\|^2$. The use of the coverage error function in (12) does not demand that the agents remain inside of the coverage area. From [1], when $\dot{x}_i^p = g_i^p(x_i)u_i^c$ represents the relationship between the state position variables and the agents' corresponding partial dynamics, the proposed control law for the control vector corresponding to the state position variables is

$$u_i^c(t) = k_i^c g_i^p(x_i)^T a_i(t), i \in \{1, \dots, N\} \tag{14}$$

where k_i^c is a positive coverage control gain for agent *i* and N is the number of agents. The application of this proposed control law will be discussed in Chapter 5.

CHAPTER 4
EXPERIMENTAL TESTBED

The experiments using the applied safe and reliable coverage control were all conducted in room 302 Transportation Building of the University of Illinois in Urbana, Illinois. The room is an instructional lab containing a Natural Point OptiTrack motion capture system. Position and attitude data were collected by the motion capture system and broadcast over a local network to three differential drive robots. The robots utilized the data from the motion capture system, as well as data from onboard sensors, to localize. With these localizations, the robots shared their positions and orientations with each other over a wireless network. Each robot ran the control scheme, and then position, orientation, and coverage data was transmitted to, collected on, and visualized on a computer in real time. An overview of the flow of information between the robots and the external environment is shown in Figure 4.1.

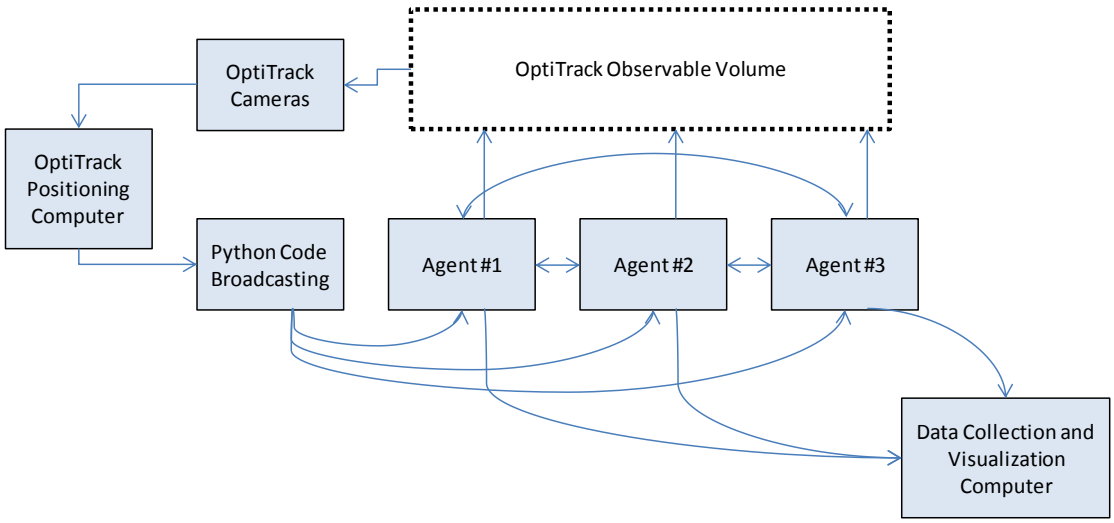

Figure 4.1: Overview of the Flow of Information between Different Entities in the Testing Environment

4.1 Natural Point OptiTrack Motion Capture System

The motion capture system utilized in the lab was an 18 camera OptiTrack system from Natural Point. The cameras used were V100:R2 cameras running at 100 Hz and are shown in Figure 4.2. At each time step, the system would calculate the centroids and the orientations of robots. The Natural Point program that processed and displayed the calculated positions and orientations of

the robots was Tracking Tools. A screenshot of the interface with the three robots is shown in Figure 4.3.

Figure 4.2: Natural Point V100:R2 Camera

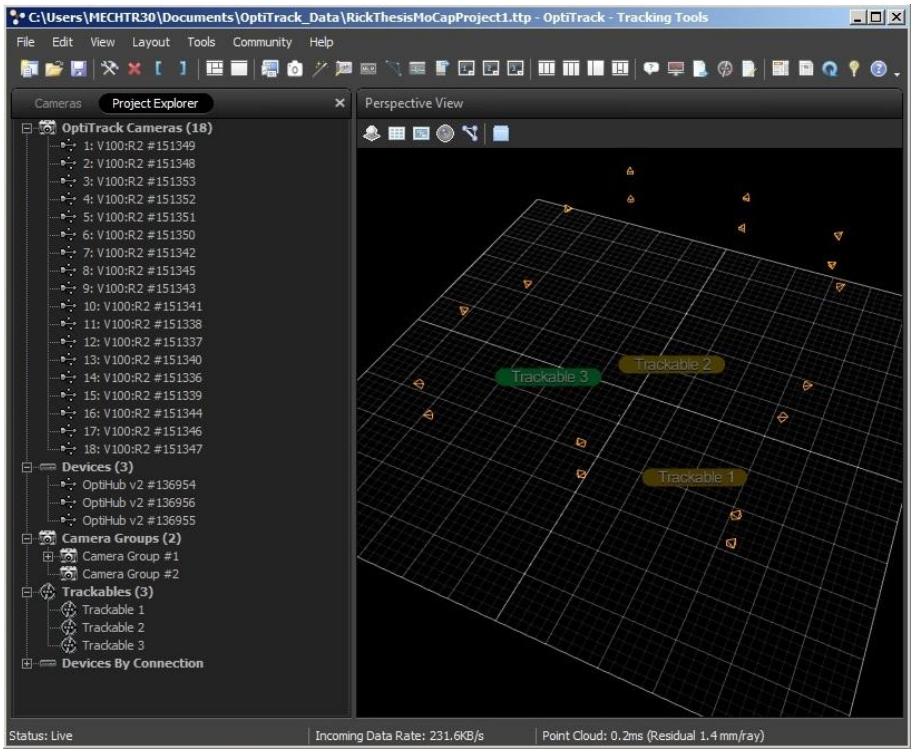

Figure 4.3: Natural Point Tracking Tools Software Interface

Each robot was defined by a unique pattern of reflective tracking balls arranged on a hard piece of foam attached to the robots. An example of the identifier template for one of the robots is visible in Figure 4.4. Due to the fact that the robots in this experiment moved within a plane and mounting space for the tracking balls was not tightly constrained, it was trivial to create a unique template for each robot. In more complicated systems where agents have more degrees of freedom, it is important to carefully construct identifier templates that will not cause aliasing within the positioning software.

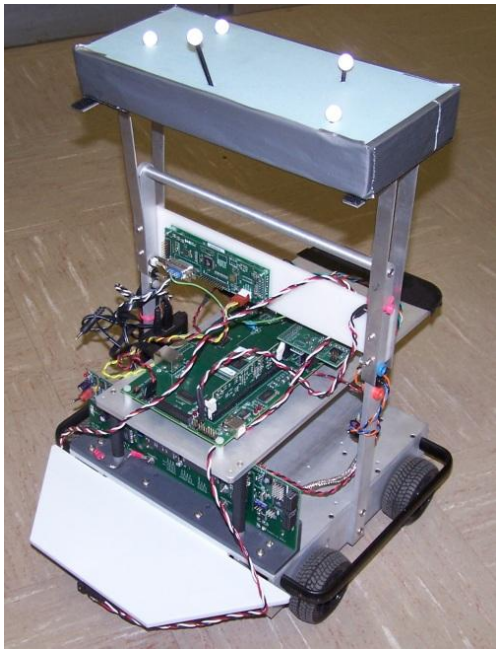

Figure 4.4: Differential Drive Robot with Tracking Ball Identifier Template

After the Tracking Tools software calculated the positions and orientations of all of the robots, a Python script would receive the data from Tracking Tools and condense the data into smaller packets containing only the necessary data. This was done to decrease transmission times. The motion capture system broadcasted far more data than was necessary to run the safe and reliable coverage control algorithm. In particular, because the robots were ground robots moving on a planar surface, the only necessary data for each robot was its coordinates in the ground plane and its heading. All of the other data, such as roll, pitch, and the positions of each individual tracking ball comprising each trackable object, were unnecessary and therefore not sent to the robots. Of course, this platform and the safe and reliable coverage control algorithm

can be expanded to accommodate agents moving in three dimensions with higher degrees of freedom.

Once the data was condensed into a smaller packet, the Python script would send the relevant data to each robot in the experiment using UDP. A Linux program running on each robot would receive the data and transfer it to the DSP (Digital Signal Processor) core on each robot. This receiving process will be discussed in more detail in the next section.

4.2 Robot Hardware and Software

The robots used in the experiment were differential drive, four wheeled, two motor robots built by Daniel Block of the College of Engineering Control Systems Lab at the University of Illinois. The robots used are pictured in Figure 4.4. Each robot has a custom designed circuit board with a Logic PD OMAP-L138 SOM-M1 (System on Module) for higher level tasks such as wireless communications and control calculations, a TMS320F28335 controlCARD for lower level tasks such as sensor data collection and motor control, a wireless module for communicating on the wireless network, various sensors such as a gyroscope on a Pololu gyroscope breakout board, and two LiPo (Lithium Polymer) batteries. A closer view of the robot with some of its important hardware labeled is shown in Figure 4.5.

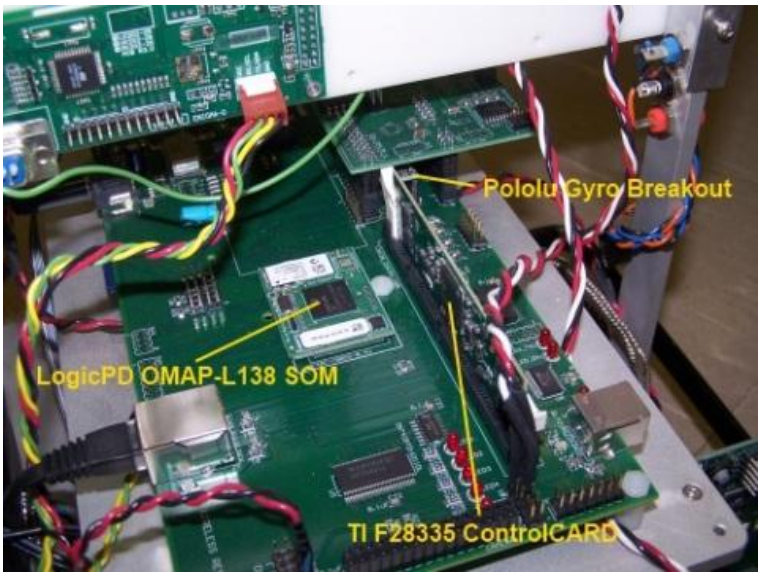

Figure 4.5: Robot Hardware with Selected Components Labeled

The OMAP processor of the robot has a DSP core to complete the control computations and to communicate with the TMS320F28335 processor that controls the motors. The OMAP also has an ARM core running an Ångström Linux distribution. Processes running in Linux on this ARM core handle the inter-robot, OptiTrack, and data visualization communications. Data is exchanged between the ARM core running Linux and the DPS core of the OMAP through shared memory. An overview of the internal flow of information on each robot is shown in Figure 4.6.

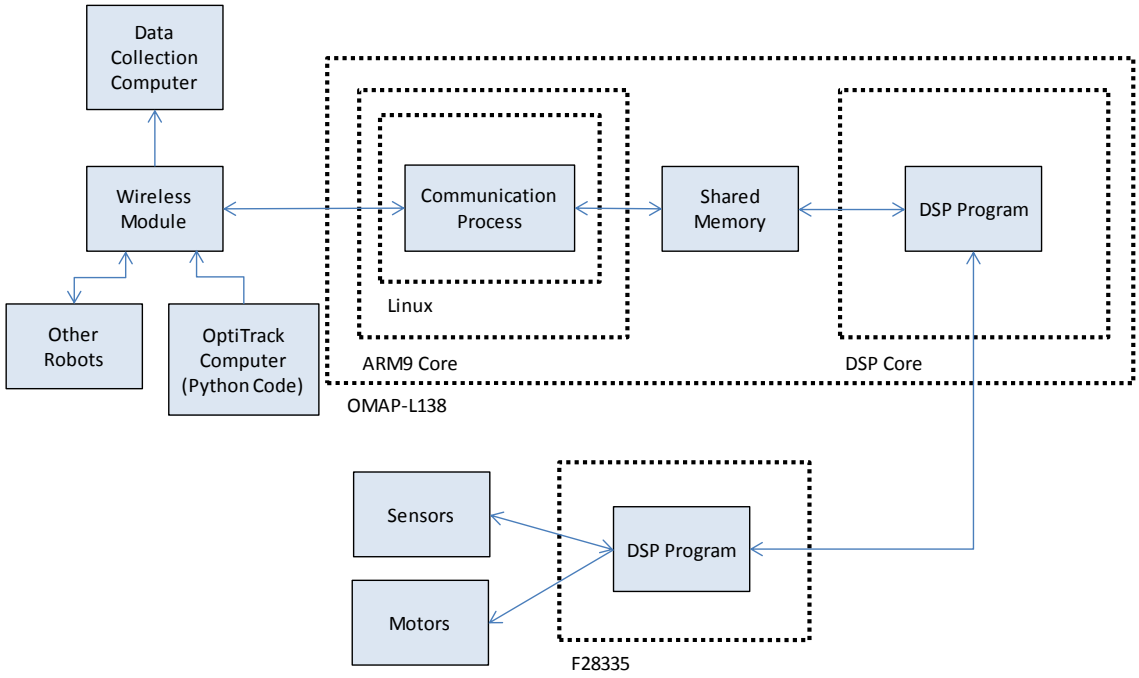

Figure 4.6: Internal Robot Information Flow

4.3 Data Collection and Visualization

At each time step of the experiment, the coverage quality data for the discretized arena was collected in addition to the positions of each of the robots and obstacles. This data was then transmitted and collected on a desktop computer running a VB6 (Visual Basic 6) application. As this data was collected, it was plotted, along with the positions of the robots, on a map of the course. The robots were represented by different colored circles, and each square of the discretized map was colored according to the quality of the coverage at that location up to that point in time. This was accomplished by converting the coverage quality to an RGB (Red,

Green, Blue) value. While this type of heat map conversion is native to other applications such as MathWorks' MATLAB, a simple version of this value to RBG encoding had to be programmed for VB6. A screenshot of the VB6 real time visualization can be seen in Figure 4.7.

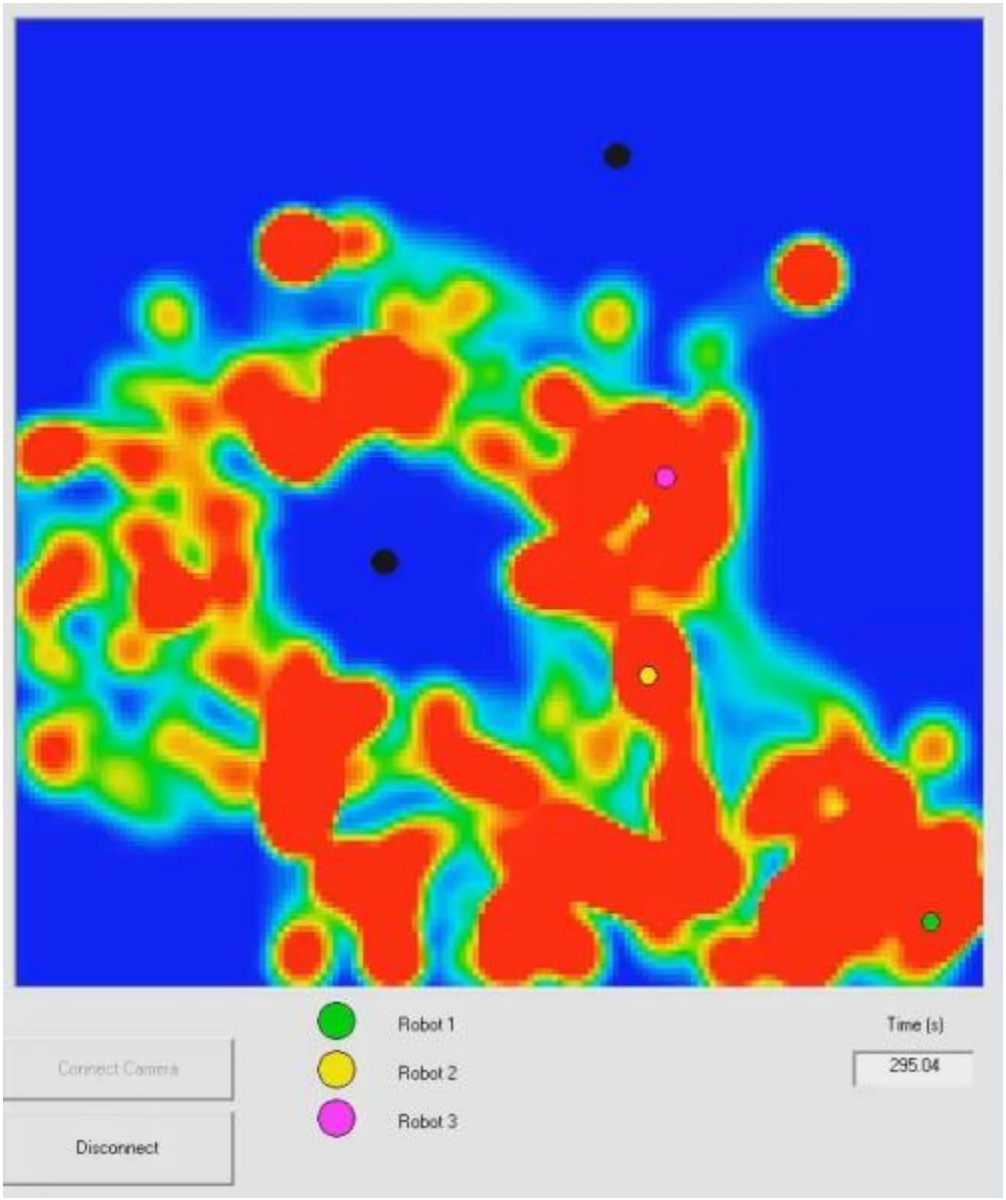

Figure 4.7: VB6 Coverage Visualization Program

CHAPTER 5
EXPERIMENTAL RESULTS

In the experiment, three robots, as described in Chapter 4, were controlled by the safe and reliable coverage control algorithm. The three robots were modeled as unicycles. Their motors were controlled by an inner PI (Proportional and Integral) control loop. Each robot was assigned gains and experimental parameters for coverage control and avoidance and proximity control. Positions and orientations of each robot were estimated with Kalman filtering using measurements from multiple sensors, including the data from the OptiTrack motion capture system. This was done so the robots were not constrained to the space viewable by the OptiTrack camera system. The aforementioned elements, as well as other assumptions and the performance and difficulties of the implementation will be discussed in this chapter.

5.1 Unicycle Model and Tracking

The three robots in the experiment are differential drive robots, so they were modeled using a unicycle model as shown in [1]. For each robot $i \in \mathrm{N} = \{1, \dots, N\}$ where N is the total number of robots, in this case three, the model is:

$$\begin{aligned} \dot{x}_{i1}^{p} &= u_{i1}\cos(x_{i3}) \\ \dot{x}_{i2}^{p} &= u_{i1}\sin(x_{i3}) \\ \dot{x}_{i3}^{p} &= u_{i2} \end{aligned} \tag{15}$$

where $x_i^p = \left[x_{i1}^p, x_{i2}^p\right]^T$ is the position state variables, x_{i3} is the orientation state, and $x_i = \left[\left(x_i^p\right)^T, x_{i3}\right]^T$ is the state vector. The state vector of each agent is determined experimentally using sensors and Kalman filtering. This will be discussed in the Kalman filtering section of this chapter. The control vector for the *i*th agent is $u_i = [u_{i1}, u_{i2}]^T$ where u_{i1} is the speed control input, and u_{i2} is the angular velocity control input. Therefore, it follows that $g_i(x_i)$ from (1) is:

$$g_i(x_i) = \begin{bmatrix} \cos(x_{i3}) & 0 \\ \sin(x_{i3}) & 0 \\ 0 & 1 \end{bmatrix} \tag{16}$$

When $b_{ij}(t,x) = \partial v_i / \partial x_{ij}, j \in \{1,2\}$ for collision avoidance and proximity with $v_i(\cdot)$ given in (8), and $b_{ij}(t,x) = a_{ij}(t)$ for coverage with $a_{ij}(t)$ given in (13), it has been shown in [1] that valid control laws for u_{i1} are:

$$u_{i1}(t,x) = c_i k_i^s \big(b_{i1}(t,x)\cos(x_{i3}) + b_{i2}(t,x)\sin(x_{i3})\big), i \in \mathrm{N} \tag{17}$$

where $c_i = -1$ for collision avoidance and proximity and $c_i = 1$ for coverage, and k_i^s are positive gains related to speed. For this experiment, a consideration had to be made in order to implement (17) on the robots. Recalling that for the cases when $\|x_i^p - \hat{x}_j\|_{P_{ij}} \leq r_{ij}$ in (4), the gradient $\partial v_{ij}^a / \partial x_i^p$ will either be undefined or equal to zero. This can be interpreted as if a pair of agents, or an agent and an object, are at or within some critical range r_{ij}, $u_{i1} = not\ defined$ or 0. This is undesirable because if two agents, or an agent and an obstacle, are too close, we would still like the avoidance objective function to continue to contribute to the overall control of the robots, even though they may have failed the safety requirements. The gradient in (4) can't realistically be used to prevent cases where $\|x_i^p - \hat{x}_j\|_{P_{ij}} \leq r_{ij}$ from ever occurring, because it is not possible to indefinitely increase the speed control input in the limit as $\|x_i^p - \hat{x}_j\|_{P_{ij}}$ approaches r_{ij} on the actual system. The speed control input and the angular velocity control inputs are both physically limited by the maximum torque of the motors, so it is possible that $\|x_i^p - \hat{x}_j\|_{P_{ij}} \leq r_{ij}$ could occur. It should also be noted that it is possible for initial conditions such that $\|x_i^p - \hat{x}_j\|_{P_{ij}} \leq r_{ij}$. Therefore, a proposed modified version of the gradient in (4) was implemented as shown below:

$$\frac{\partial v_{ij}^a}{\partial x_i^p} = \begin{cases} 0, & \text{if } \|x_i^p - \hat{x}_j\|_{P_{ij}} \geq R_{ij} \\ 4\dfrac{\left(R_{ij}^2 - r_{ij}^2\right)\left(\left(x_i^p - \hat{x}_j\right)^T P_{ij}\left(x_i^p - \hat{x}_j\right) - R_{ij}^2\right)}{\left(\left(x_i^p - \hat{x}_j\right)^T P_{ij}\left(x_i^p - \hat{x}_j\right) - r_{ij}^2\right)^3}\left(x_i^p - \hat{x}_j\right)^T P_{ij}, & \text{if } R_{ij} > \|x_i^p - \hat{x}_j\|_{P_{ij}} > r_{ij} \\ 4\dfrac{\left(R_{ij}^2 - r_{ij}^2\right)\left(\left(x_i^p - \hat{x}_j\right)^T P_{ij}\left(x_i^p - \hat{x}_j\right) - R_{ij}^2\right)}{\left(\left(x_i^p - \hat{x}_j\right)^T P_{ij}\left(x_i^p - \hat{x}_j\right) - \left(k_{ij} r_{ij}\right)^2\right)^3}\left(x_i^p - \hat{x}_j\right)^T P_{ij}, & \text{if } \|x_i^p - \hat{x}_j\|_{P_{ij}} \leq r_{ij} \end{cases} \tag{18}$$

where $k_{ij} \in \mathbb{R}_+ = (0,1)$ is a design parameter to force the gradient to a relatively large value until the agent pair, or the agent and the object, are no longer considered to be too close. From [1] it was shown that u_{i1} is optimally efficient when

$$x_{i3}^o = \varphi_i^o = {}^{\pi}/_{2} - \arctan\left(\frac{b_{i1}(t,x)}{b_{i2}(t,x)}\right) \tag{19}$$

A valid angular velocity control can then be designed as a proportional controller as follows:

$$u_{i2} = -k_i^{\omega}(x_{i3} - \varphi_i^o) \tag{20}$$

where k_i^{ω} are positive gains related to angular velocity.

In the numerical example simulation from [1], the state vectors of the agents were updated at each time step according to the dynamic model in (15). For this implementation, the control vector is applied to the differential drive motors of the robots using a PI control inner loop as shown in Figure 5.1.

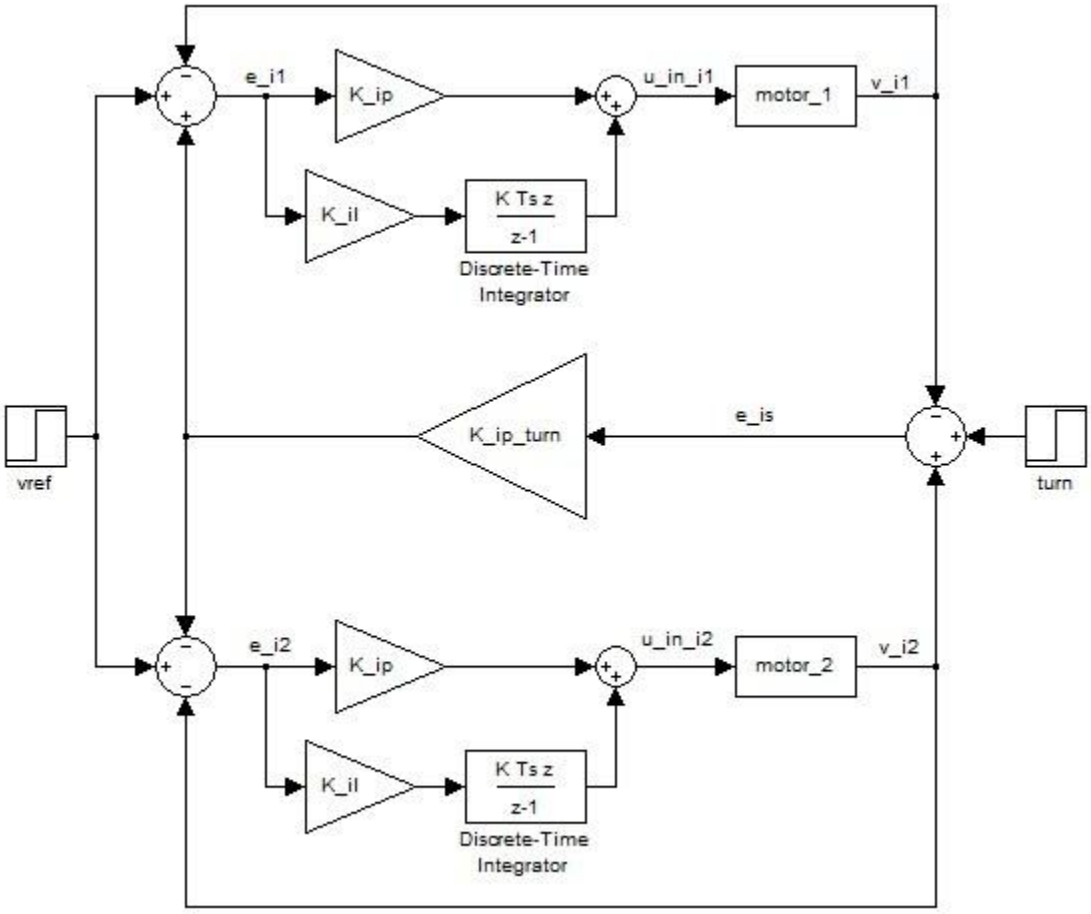

Figure 5.1: Block Diagram of Inner PI Control Loop

The error equations for this inner loop controller of each *i*th robot are shown below:

$$\begin{aligned} e_{is} &= v_{i2} - v_{i1} + turn \\ e_{i1} &= vref - v_{i1} + K_{ip_{turn}} e_{is} \\ e_{i2} &= vref - v_{i2} - K_{ip_{turn}} e_{is} \end{aligned} \tag{21}$$

where $turn$ is u_{i2} and $vref$ is u_{i1}, $K_{ip_{turn}}$is a positive design gain for each robot, and $v_{ij}, j \in \{1,2\}$ are the velocities of the left and right motors of the robots.

$$\begin{aligned} u_{inner_{i1}} &= K_{ip} e_{i1} + K_{iI} \int_0^t e_{i1} d\tau \\ u_{inner_{i2}} &= K_{ip} e_{i2} + K_{iI} \int_0^t e_{i2} d\tau \end{aligned} \tag{22}$$

5.2 Experimental Parameters

In most of the experiments conducted, three robots and two obstacles were used. As in the numerical example in [1], let indices {1,2,3} denote the robots, and let indices {4,5} denote the objects. The robots were placed arbitrarily within a 5 meter by 5 meter area centered in the visible range of the OptiTrack motion camera system. Due to the fact that the robots used in the implementation were the same design and nearly identical, similar design parameters were used for each agent in the experiments. For recording the coverage quality the robots performed over time, the 5 meter by 5 meter course was discretized into a 160x160 space. Therefore, there were 25,600 discrete square areas with their own coverage quality information. Each tile of the area to be covered was a square with size length 0.03125 m. In order to help visualize the area being covered and the sizes of the robots and obstacles relative to that area, Figure 5.2 has been included to show the robots navigating the area during one of the experiments.

The positive definite avoidance matrices P_{ij} and P_{il} from (2) and (3) were declared to be the 2-dimensional identity matrix. The circular avoidance and detection regions of the robots were defined by $R_{1j} = R_{2j} = R_{3j} = 1.0$ and $r_{1j} = r_{2j} = r_{3j} = 0.5, \text{j} \in \{1, \dots, 5\}$. The gains relating to speed control input in (9) were defined to be $\hat{k}_1 = \hat{k}_2 = \hat{k}_3 = 1.25$ for the three agents. The agents were assigned proximity distances $\hat{R}_{ij}$ from (5) in a uniform manner such that $\hat{R}_{1j} = \hat{R}_{2j} = \hat{R}_{3j} = 1.5, j \in \{1,2,3\}$. Scaling coefficients γ_{ij} from (8) for collision avoidance objective functions were set as $\gamma_{1j} = \gamma_{2j} = \gamma_{3j} = 0.025 \; \forall \, j \in \{1, \dots, 5\} : i \neq j$. Scaling coefficients γ_{ij} from (8) for proximity objective functions were set as $\gamma_{1j} = \gamma_{2j} = \gamma_{3j} =$

$0.075\ \forall\ j \in \{1,2,3\} : i \neq j$. As in [6], for one of the experiments it was decided that robot 1 would avoid obstacles but would not attempt to avoid the other agents or remain in proximity with them. In calculating the approximation in (7), $\delta = 2$ was chosen.

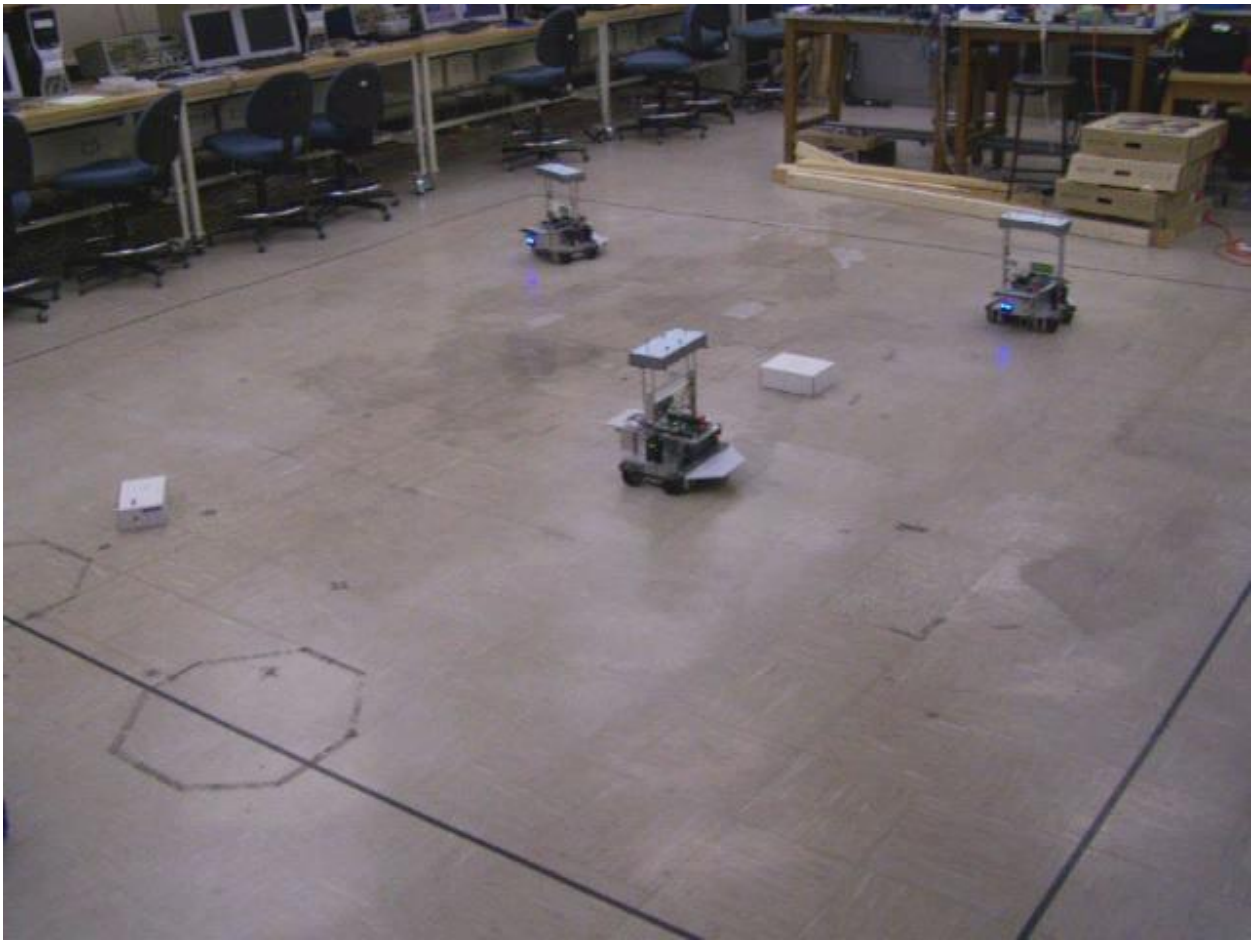

Figure 5.2: Experimental Setup with Three Robots and Two Obstacles

In the experiments, limitations on wireless communication and the transmission of cumulative coverage data were simulated. Each robot continuously estimated its own position using Kalman filtering, which will be discussed in the following section, and wirelessly sent this calculated position to the other robots. Using these positions, each robot calculated the sensing and cumulative coverage data, Q from (11), for itself and the other robots. However, even though each robot had the total cumulative coverage data for every other robot in memory at all times, the cumulative coverage data for a robot was only merged with the cumulative coverage data of other robots if the neighboring robots were determined to be within the assigned

proximity distances $\hat{R}_{ij}$. This simulation of sharing coverage data was done because it was determined to be faster to calculate the coverage data for all robots on each robot rather than broadcast the coverage data for a robot to the other robots over the wireless network. This method also avoided the problem of having to design a robust protocol that would allow for sharing of large amounts of data over the wireless network without data loss or network interference.

For coverage control, agents were also given specific parameters. In many experiments, the robots were assumed to be homogenous in terms of sensing region radius, R_i from (10), and peak sensing capability, M_i from (10). The robots were designed with $R_1 = R_2 = R_3 = 0.25$ and $M_1 = M_2 = M_3 = 12$. While all of the robots could potentially be fitted with appropriate sensors, such as LIDAR (Light Detection And Ranging), cameras, ultrasonic, etc, this was left out for simplicity and for future work. The sensing radius for each of the robots was designed so that the sensing area was approximately the same as the area covered by the base of the robots while they drove around the space. Just as with the avoidance and proximity control, speed control gains from (14) were also assigned for coverage control of each robot. These gains were defined as $k_1^c = k_2^c = k_3^c = 1.5 \cdot 10^{-4}$. The maximum coverage value C^* from (13) was chosen to be $C^* = 40$.

The total speed control input for each robot was calculated using the sum of the results of (17) when using the aforementioned defined gains of k_i^c for coverage control and k_i^s and $\hat{k}_i$for proximity and avoidance control. The heading angle control for each robot was calculating using (20). The gains for the proportional heading angle control law were set as $k_i^\omega = 1, i \in \{1,2,3\}$. The reference angle for this control was the sum of the desired angles for coverage and for avoidance and proximity calculated by (19).

5.3 Kalman Filtering

The OptiTrack motion capture system offers accurate data about the position and orientation of rigid bodies, such as the robots used in the experiment. However, it does have two issues: the updated position and orientation is only calculated by the camera system's software at 100Hz, and the camera system has a limited tracking area. The first problem, the rate at which robot data is calculated, leaves open the possibility that when the coverage and avoidance control algorithm runs, it may use position and orientation data that is up to 10 ms old. This means that

the algorithm may produce control inputs that are not suited for the robots' actual current positions and orientations. This is part of the motivation for having a better way to update the robots' states. The second problem, the limited tracking range of the cameras, is a much larger problem. If only the camera system is used and a robot ends up driving out of the range of the cameras, there is no chance for recovery. Since the algorithm doesn't require that the agents remain in the area being covered, this again provides motivation for an improved method for updating the agents' states. To avoid both of the aforementioned issues, a dead reckoned position and orientation could be used. In this situation, the data from the gyro and the encoders of the robots could be used to dead reckon the position and orientation of the robots. However, this method has well known downsides as well such as wheel slippage and gyro drift. It was decided that a Kalman filter to combine the dead reckoned position and the position from the camera system would work well for this application.

As described in [7], the Kalman filter is a set of recursive mathematical equations which aim to minimize the mean of the squared error in the estimation of the states in a given process. This filter works well for this implementation, because a precisely identified model of the system and the dynamics of the robot are not required to accurately estimate states. It is assumed that the state for any given robot, x, is governed by the linear stochastic difference equation

$$x_k = Ax_{k-1} + B_{k-1}u_{k-1} + w_{k-1} \tag{23}$$

with state measurements, z, given by

$$z_k = Hx_k + v_k \tag{24}$$

where w_k and v_k are random process and measurement noise, respectively, with assumed normal probability distributions of

$$p(w) \sim N(0, Q) \tag{25}$$

$$p(v) \sim N(0, R) \tag{26}$$

where Q and R are assumed to be constant matrices representing process noise covariance and measurement noise covariance, respectively. As demonstrated in [8], trusting the onboard dead reckoning process much more than the off-board camera measurements provides smooth state variables without the drift inherent with dead reckoning with encoders and a gyro. Therefore, the covariance of the process noise matrix, Q, was set to be orders of magnitude smaller than the covariance of measurement noise matrix, R. Additionally, these matrices were experimentally tuned so that if a robot spent a considerable amount of time outside of the trackable area of the

camera system and then returned to the trackable area, the drifted states of the robot would converge at a desirable rate to near the states measured by the cameras. These Kalman filtered state variables, updated every millisecond on the robots' DSPs, provided the state variables to be used in the control algorithms.

From [7], the Kalman filter is composed of two sets of equations. The first set, the time update equations, are intended to predict an *a priori* state estimate, $\hat{x}_{\bar{k}}$. The second set, the measurement update equations, are intended to correct the *a priori* estimate to generate a more accurate *a posteriori* state estimate $\hat{x}_k$. The equations for the time update are given as

$$\hat{x}_{\bar{k}} = A\hat{x}_{k-1} + B_{k-1}u_{k-1} \tag{27}$$

$$P_{\bar{k}} = AP_{k-1}A^T + Q \tag{28}$$

where P_k is the *a priori* state estimate error covariance matrix. The equations for the measurement update are given as

$$K_k = P_{\bar{k}}H^T(HP_{\bar{k}}H^T + R)^{-1} \tag{29}$$

$$\hat{x}_k = \hat{x}_{\bar{k}} + K_k(z_k - H\hat{x}_{\bar{k}}) \tag{30}$$

$$P_k = (I - K_kH)P_{\bar{k}} \tag{31}$$

For the next time step after a measurement update, the previous *a posteriori* state estimate is then used to predict the *a priori* state estimate.

5.4 System Performance

Testing of the safe and reliable coverage control algorithm on the platform was conducted during many experiments. Five different sets of experimental results of interest are presented in this section. However, many more experiments were conducted in order to tune certain system parameters, such as the coverage and avoidance proximity gains, so that satisfactory performance was achieved on this platform. Also, for each experiment, it is assumed parameters are equal to the values given in Section 5.2 unless otherwise noted.

Experiment 1 used the default parameters given in Section 5.2 representing three homogenous robots searching the area. The centered positions of the obstacles, fixed across all of the experiments, except Experiment 5 which had no obstacles, were (1.9, 2.2) and (3.1, 4.3). The obstacles are shown as the black circles in Figure 5.3. The figure also shows the trajectories of the robots over the duration of this experiment. The green dots and red dots show the initial and final positions of the robots, respectively. The trajectories are shown as the blue line for

agent 1, the green line for agent 2, and the magenta line for agent 3. Figure 5.4 shows the pairwise distances for the robots in the experiment over time, and Figure 5.5 shows the pairwise distances for the robots and obstacles over time. It is important to note the behavior of the robots near the desired proximity, detection, and avoidance boundaries. The normalized coverage error over time for this experiment is given in Figure 5.6. Finally, the terminal coverage quality is shown as a heat map in Figure 5.7.

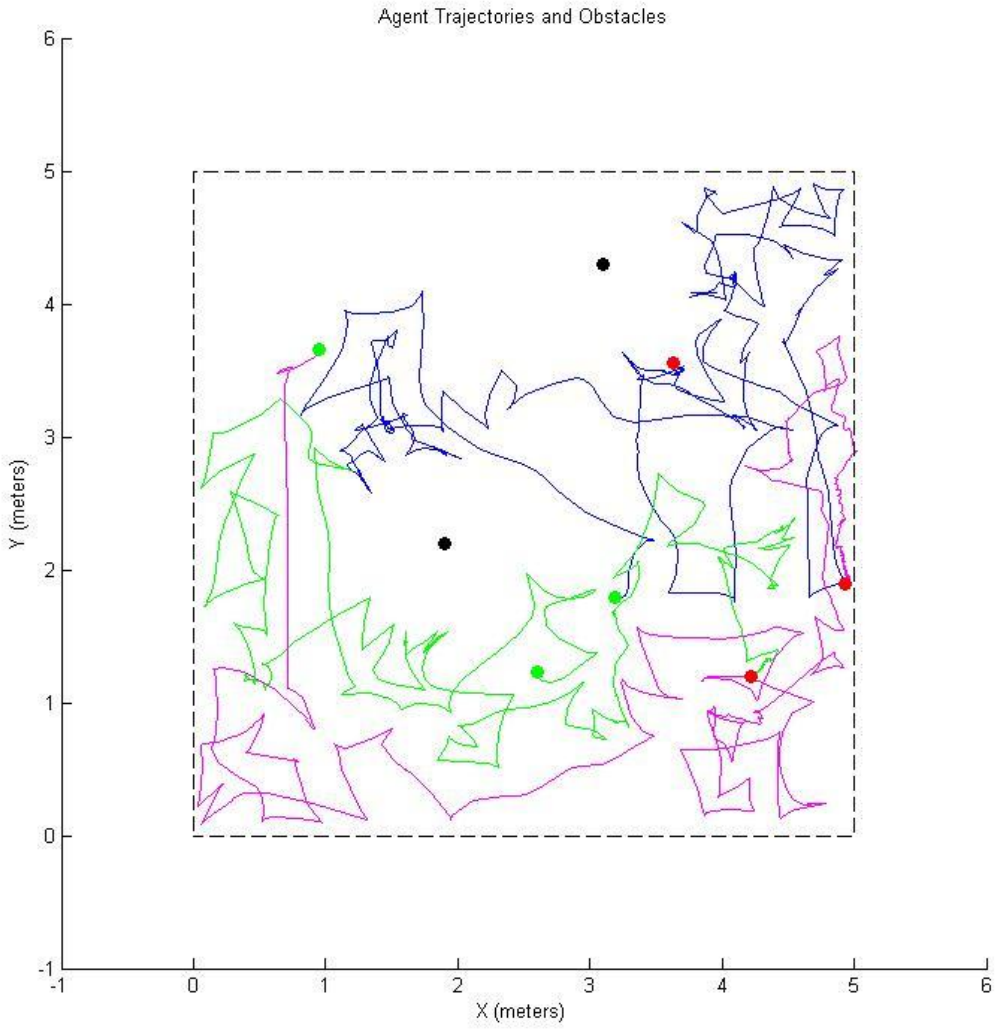

Figure 5.3: Experiment 1 Agent Trajectories and Object Positions Where Green Circles Are Start Positions and Red Circles Are Final Positions (blue = agent 1, green = agent 2, magenta = agent 3)

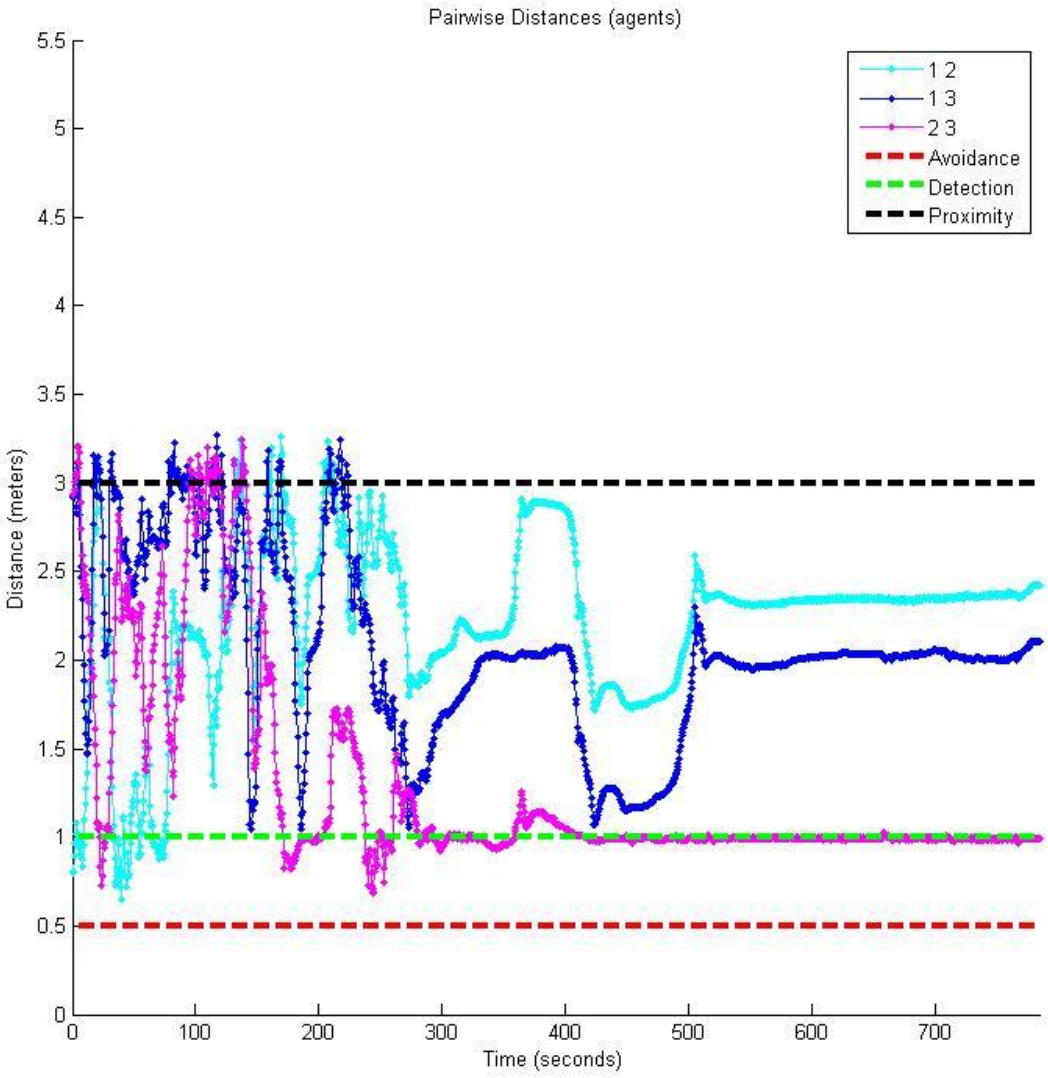

Figure 5.4: Experiment 1 Agent Pairwise Distances

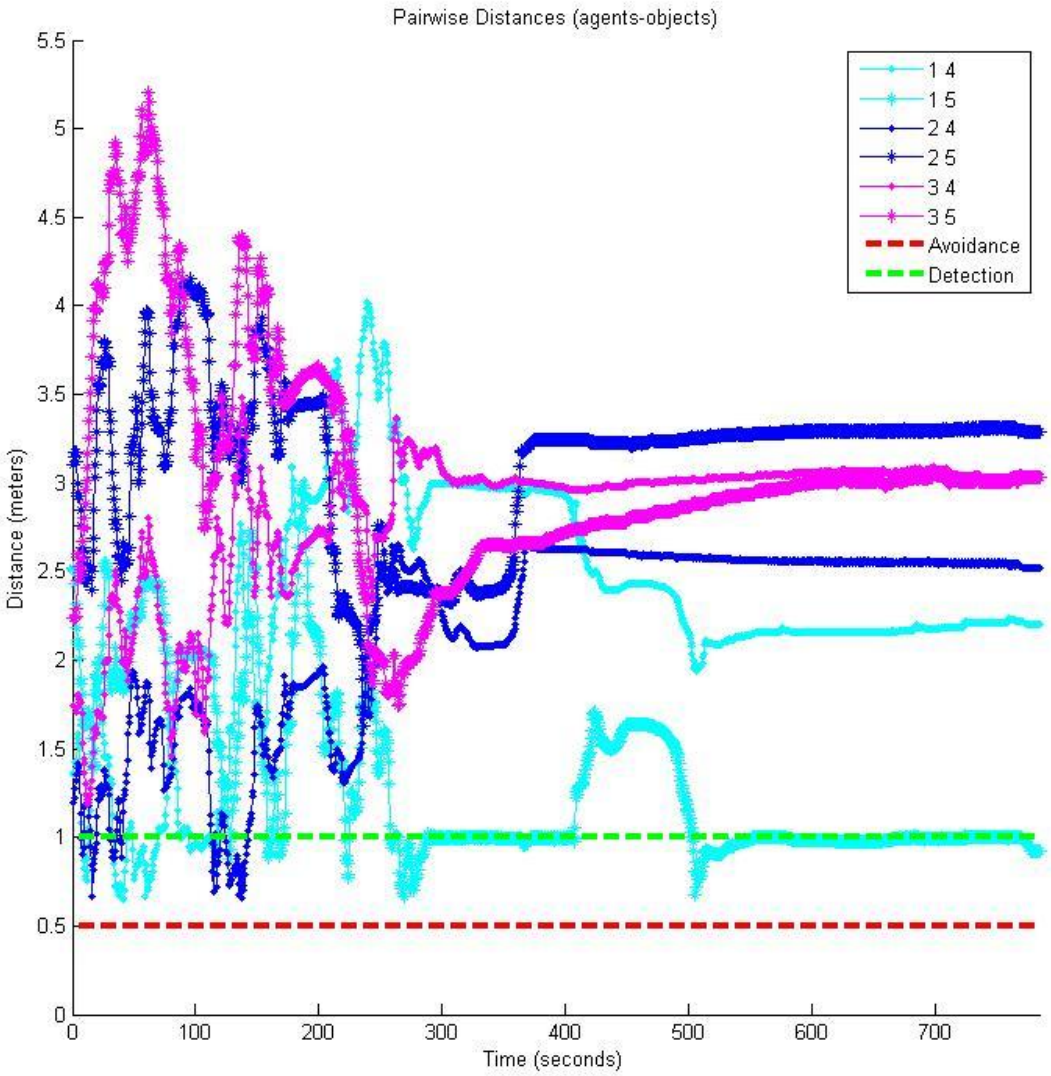

Figure 5.5: Experiment 1 Agent-Object Pairwise Distances

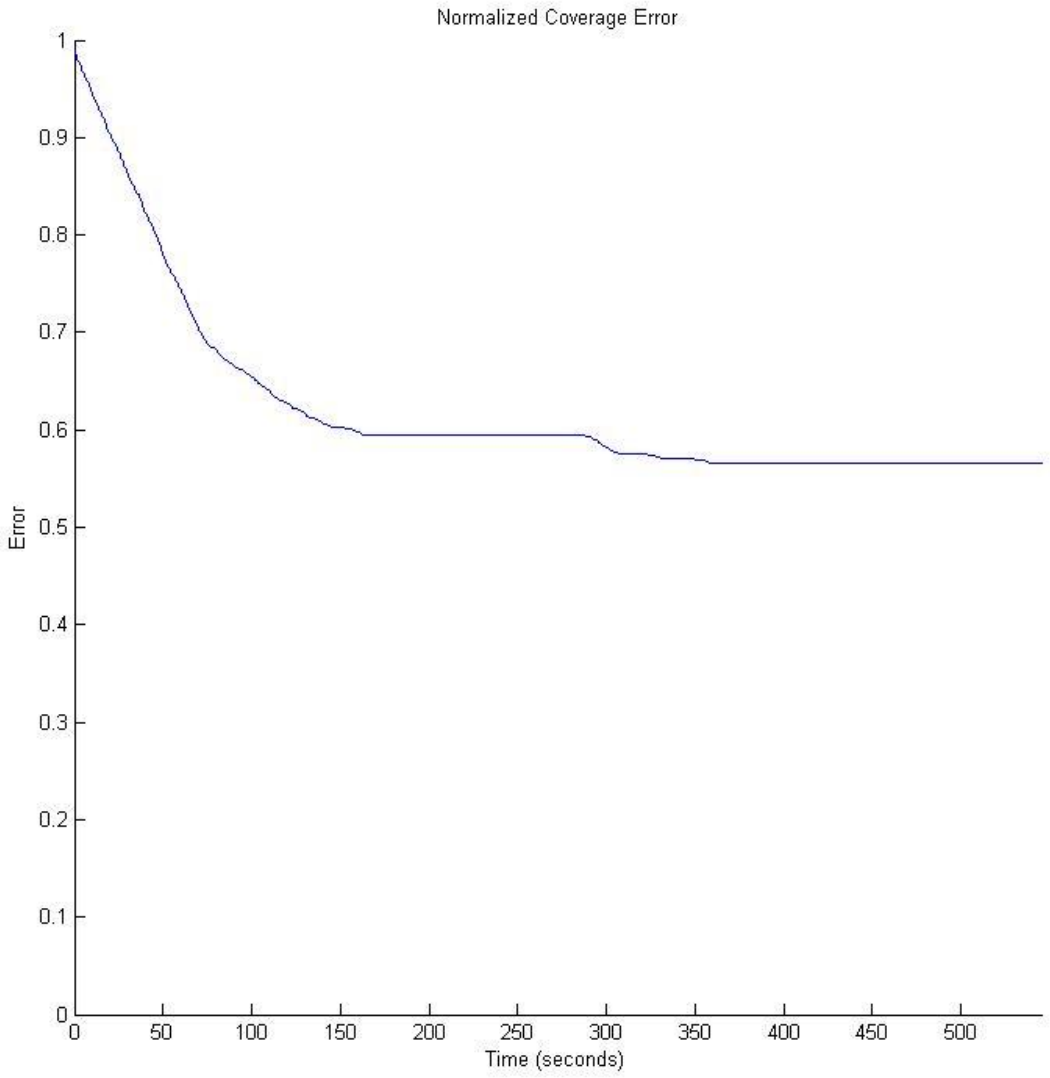

Figure 5.6: Experiment 1 Normalized Coverage Error

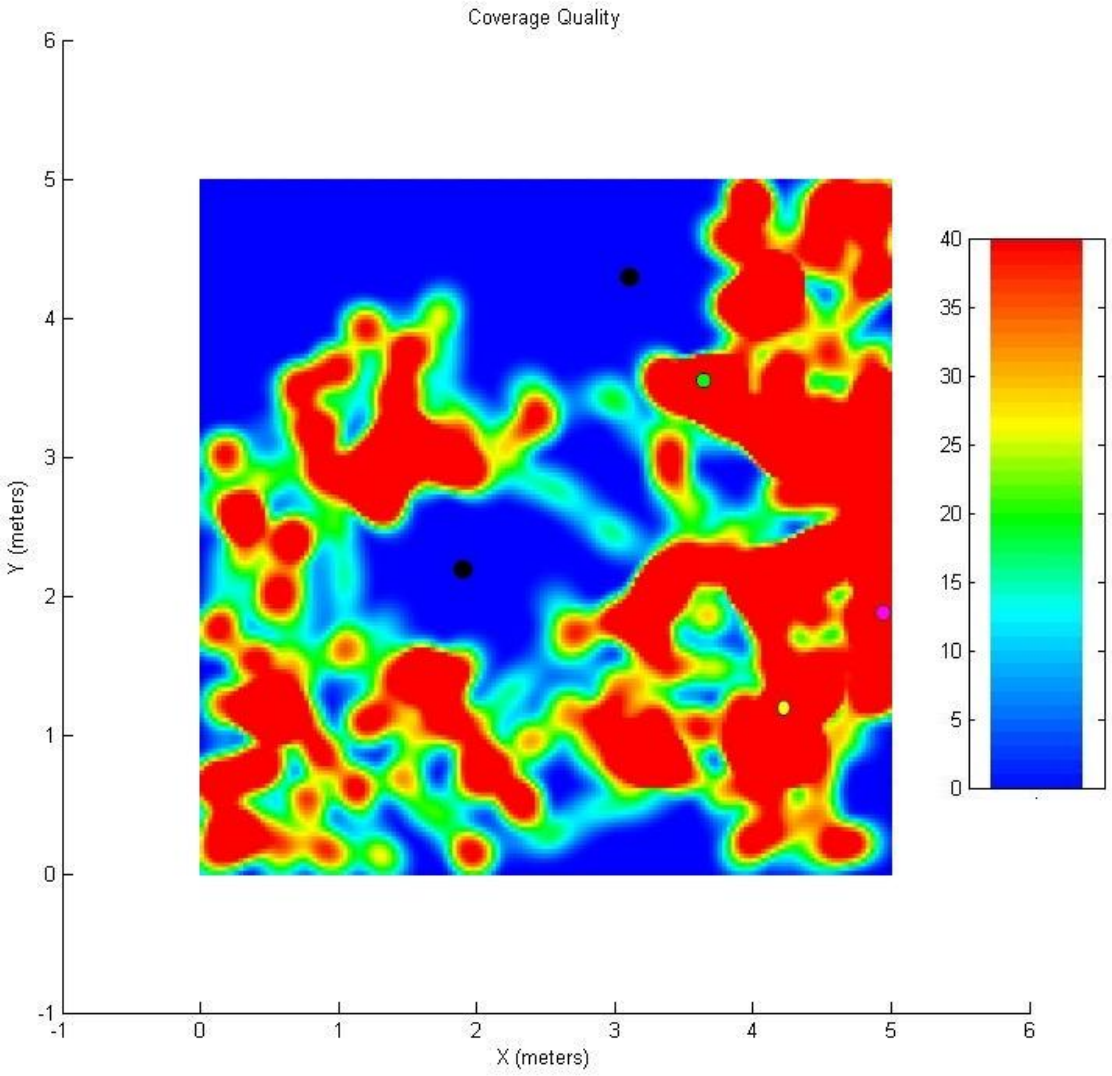

Figure 5.7: Experiment 1 Terminal Coverage Quality Heat Map (green = agent 1, yellow = agent 2, magenta = agent 3)

It becomes clear that in Experiment 1 the robots did not fully cover the search domain after becoming "stuck" in a particular arrangement. This represents the issue of combining the control laws from (17) to create the control input to each robot. In this case, it is apparent that the control contributed by the coverage control law and the control contributed by the proximity and avoidance control laws summed to yield a control input for each robot that was not great enough to actually move any of the robots. Furthermore, because the sensing regions of the robots were designed to be very local and symmetrical, representing approximately only the area

a robot occupied on the floor, the coverage control law remained fixed over time. Therefore, once the robots reached their terminal arrangement with the coverage quality that had been cumulated up to that point in time, there was very little chance that the robots would be able to escape this equilibrium. It is also worth noting that in the pairwise distance plots, the robots behaved desirably around the proximity limit, detection limit, and avoidance limit. Whenever a robot pairwise distance went over the proximity limit, the robots quickly rebounded back into proximity. Similarly, whenever a robot pairwise distance or robot-obstacle pairwise distance went under the detection limit into the detection region, the robots rebounded out of the detection region without entering the avoidance region.

In Experiment 2, an attempt at addressing the issue of the robots being trapped in an equilibrium was made by modifying the coverage gains of the robots from $k_1^c = k_2^c = k_3^c = 1.5 \cdot 10^{-4}$ to nonhomogeneous values $k_1^c = 1.5 \cdot 10^{-4}$, $k_2^c = 4.5 \cdot 10^{-4}$, and $k_3^c = 3.0 \cdot 10^{-4}$. The results for this experiment are shown in Figures 5.8 through 5.12. The changes made in Experiment 2 failed to improve the performance of the system. Much like in Experiment 1, the robots left much of the area uncovered and became trapped in an equilibrium. It is apparent that simply increasing coverage gains and reducing the amount of uniformity of coverage gains was not enough to address the problems preventing more thorough coverage.

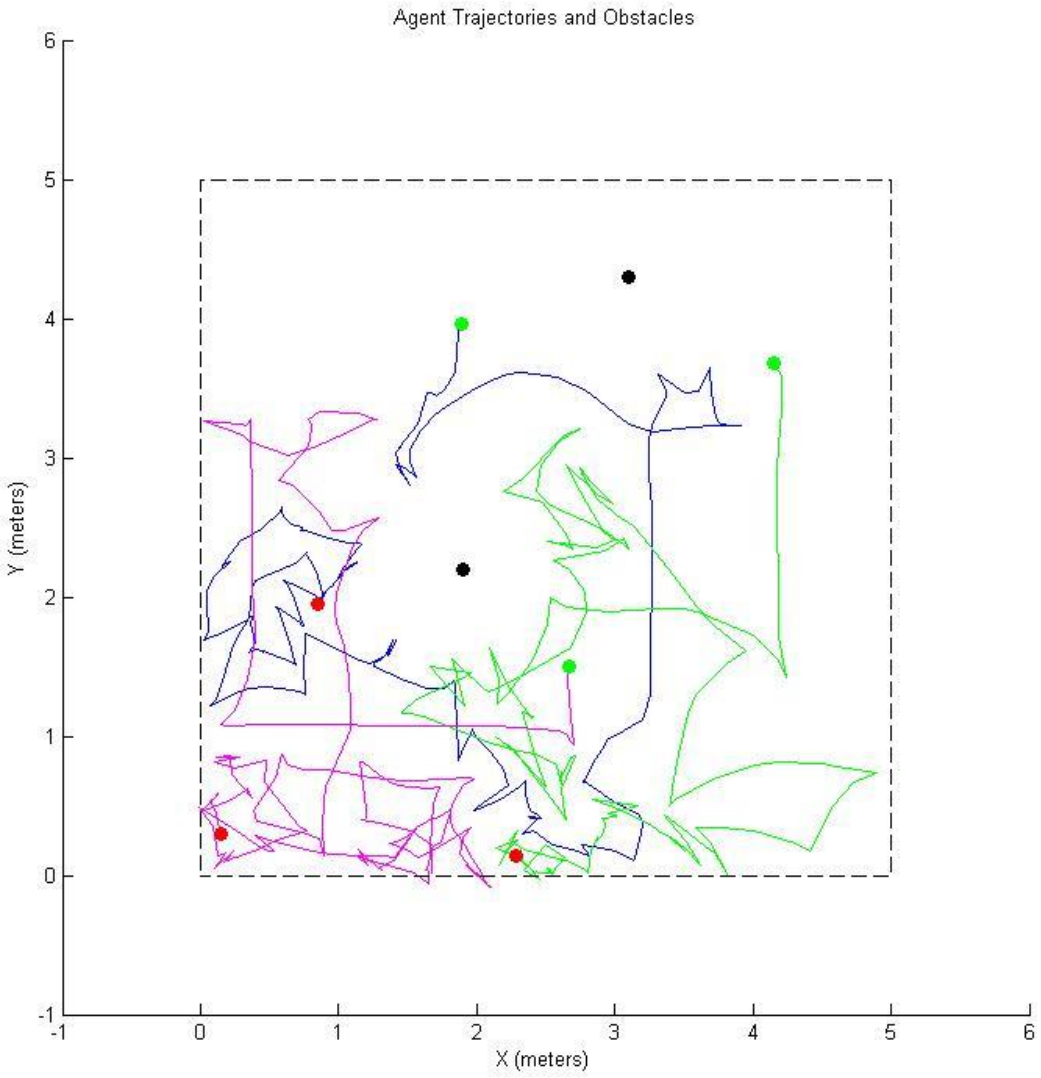

Figure 5.8: Experiment 2 Agent Trajectories and Object Positions Where Green Circles Are Start Positions and Red Circles Are Final Positions (blue = agent 1, green = agent 2, magenta = agent 3)

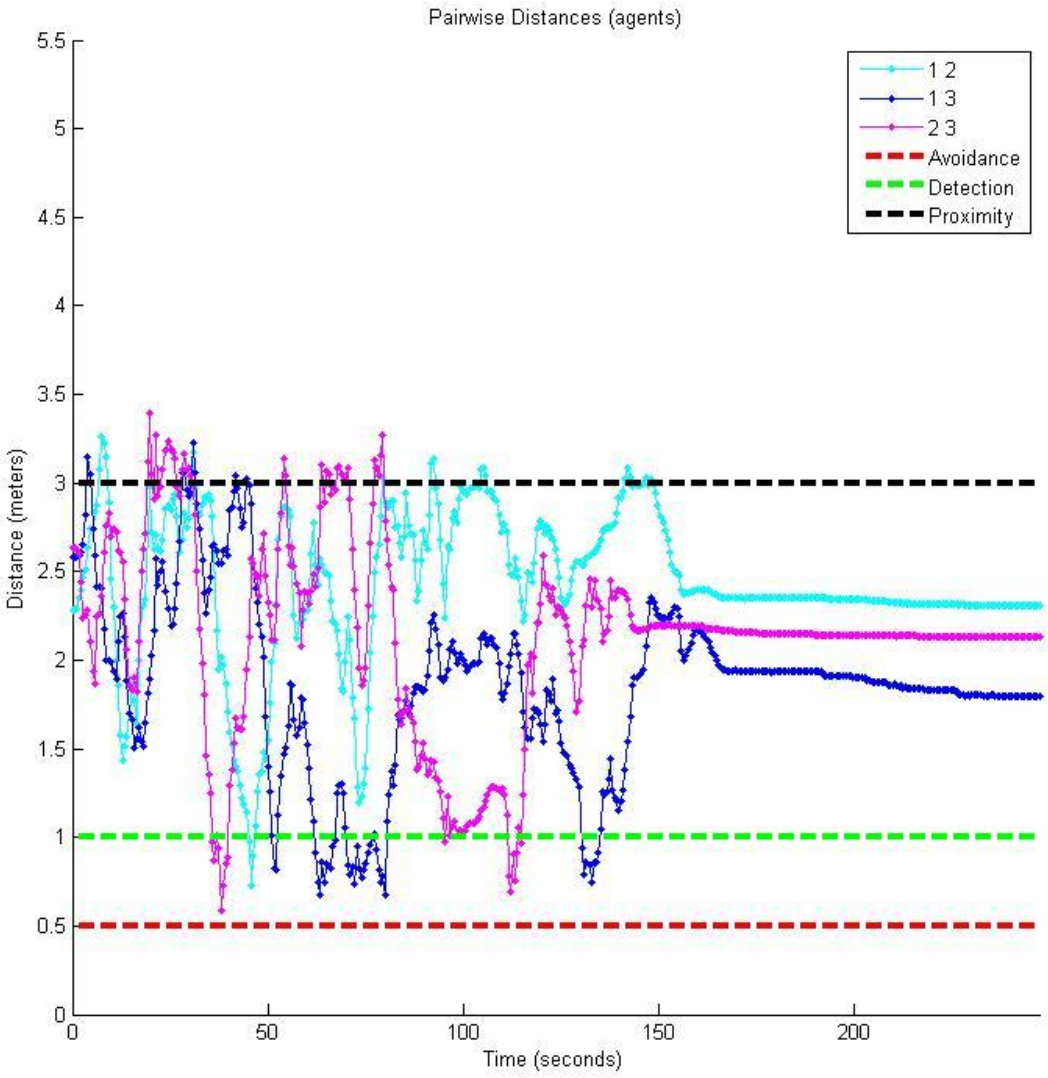

Figure 5.9: Experiment 2 Agent Pairwise Distances

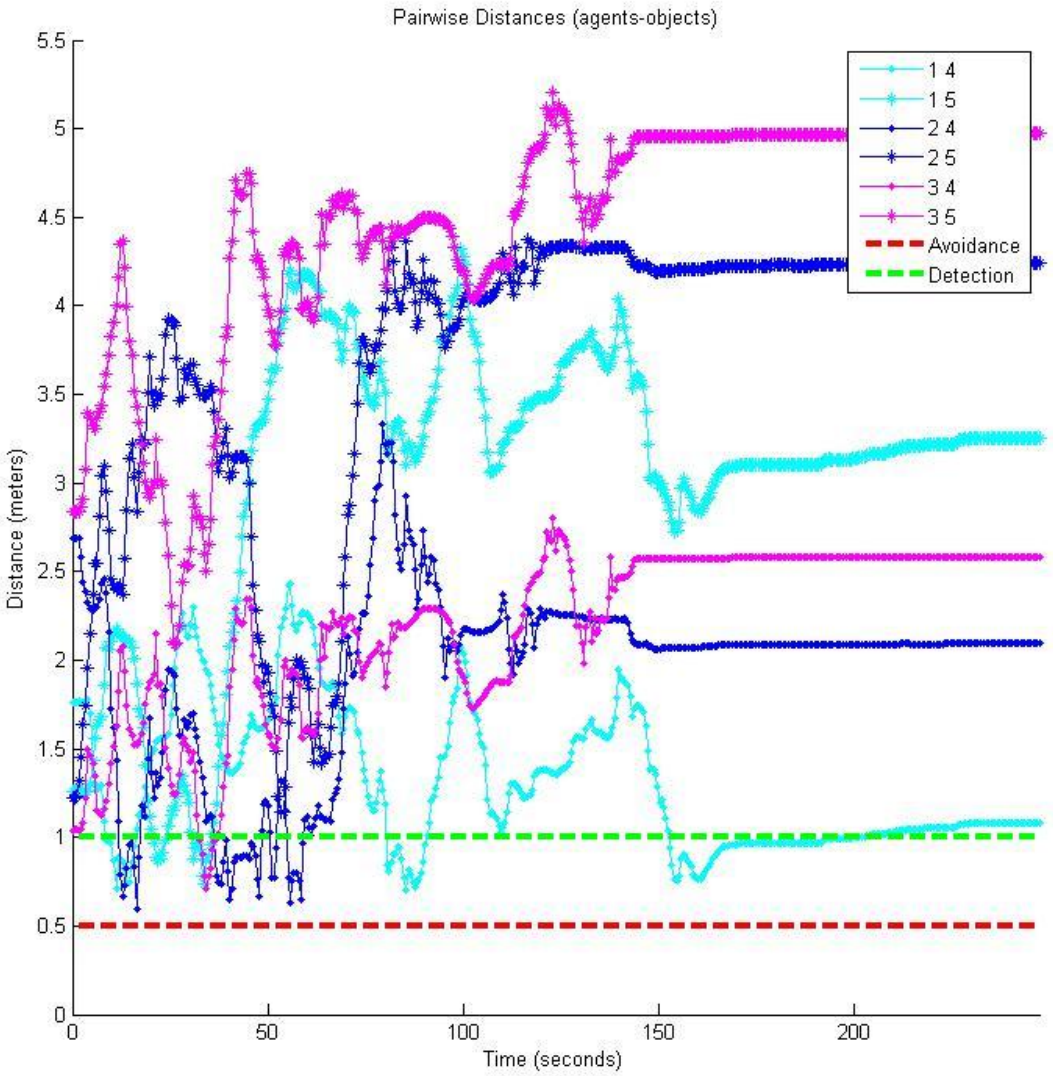

Figure 5.10: Experiment 2 Agent-Object Pairwise Distances

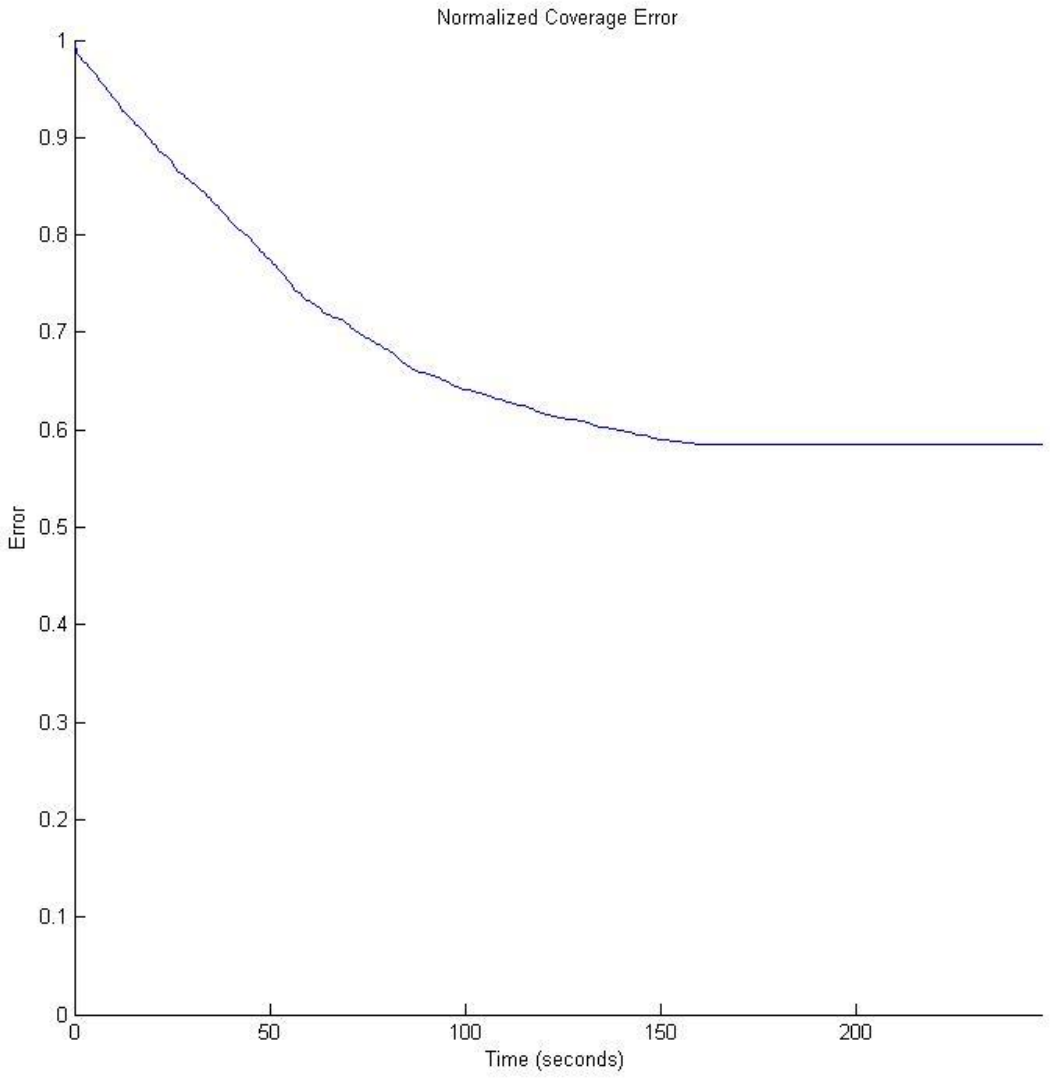

Figure 5.11: Experiment 2 Normalized Coverage Error

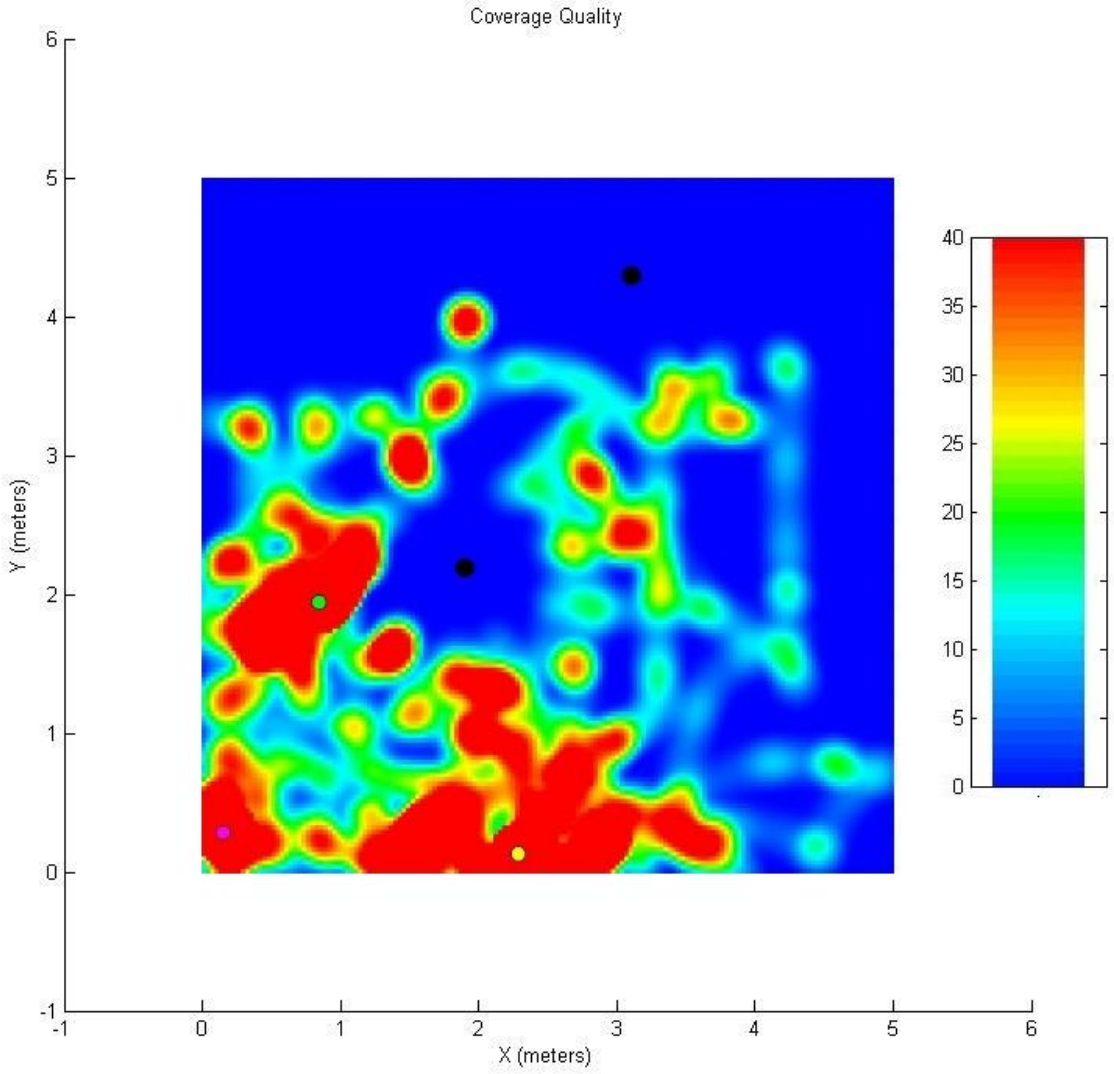

Figure 5.12: Experiment 2 Terminal Coverage Quality Heat Map (green = agent 1, yellow = agent 2, magenta = agent 3)

Experiment 3 attempts to solve the equilibrium issue by removing the proximity constraint on one of the robots. By restoring the homogenous coverage gains of $k_1^c = k_2^c = k_3^c = 1.5 \cdot 10^{-4}$ and removing the proximity constraint on robot 1, the results of Experiment 3 are shown in Figures 5.13 through 5.17. As in Experiment 2, this attempt at correcting the equilibrium issue fails. It is interesting to note that even though robot 1 makes no attempt to remain in proximity with the other robots, the other two robots are able to maintain proximity with robot 1 by

themselves. Therefore, it is apparent that the proximity constraint is likely not necessary for both robots in every unique set of two robots.

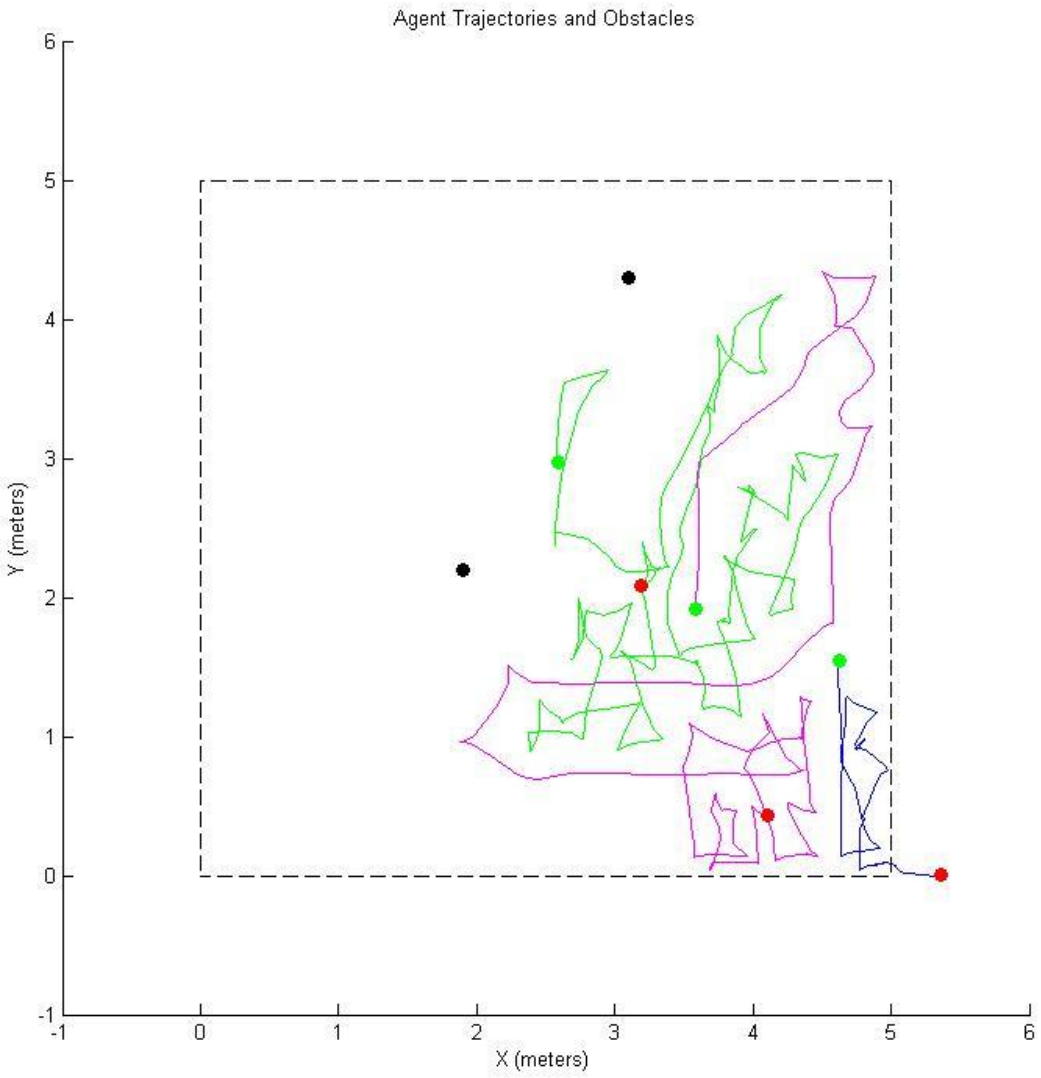

Figure 5.13: Experiment 3 Agent Trajectories and Object Positions Where Green Circles Are Start Positions and Red Circles Are Final Positions (blue = agent 1, green = agent 2, magenta = agent 3)

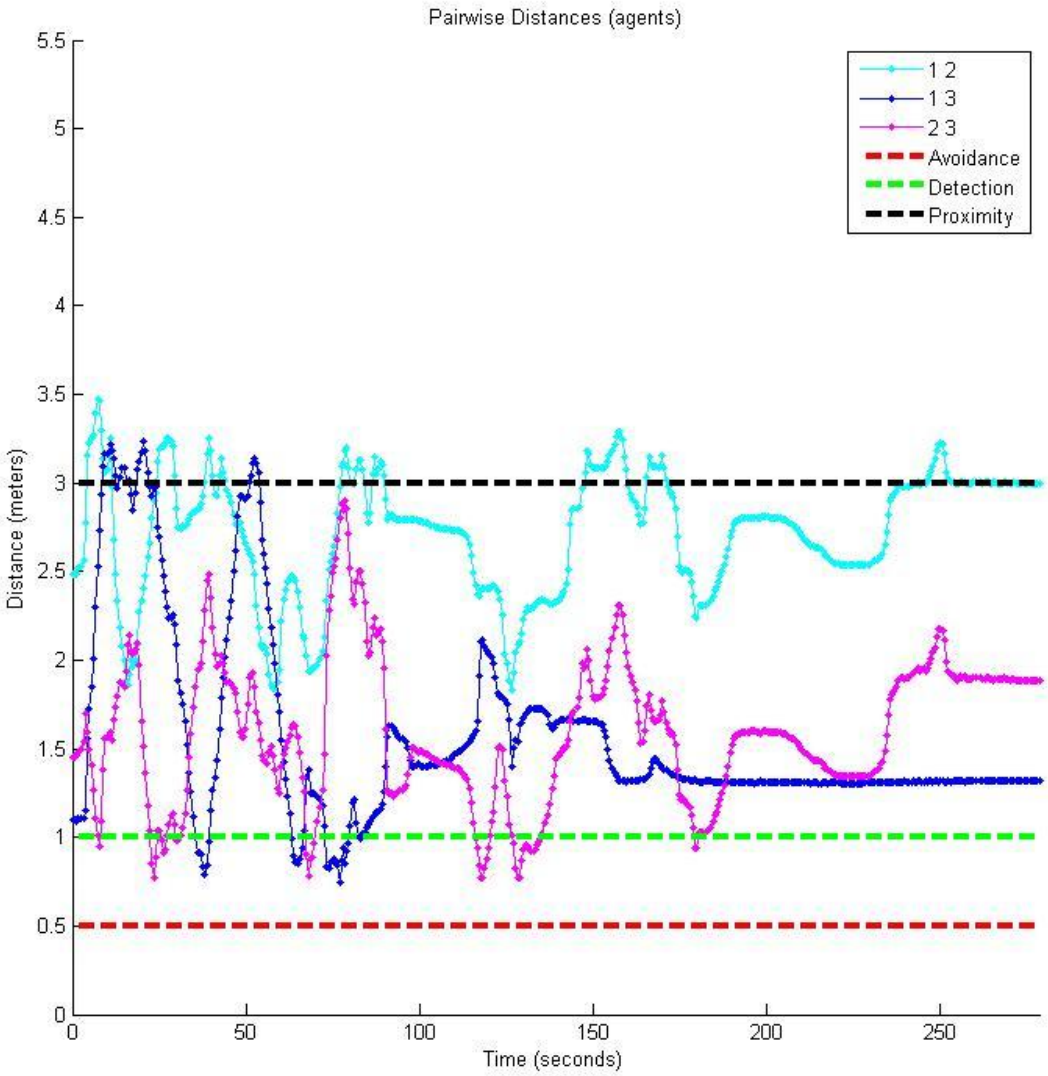

Figure 5.14: Experiment 3 Agent Pairwise Distances

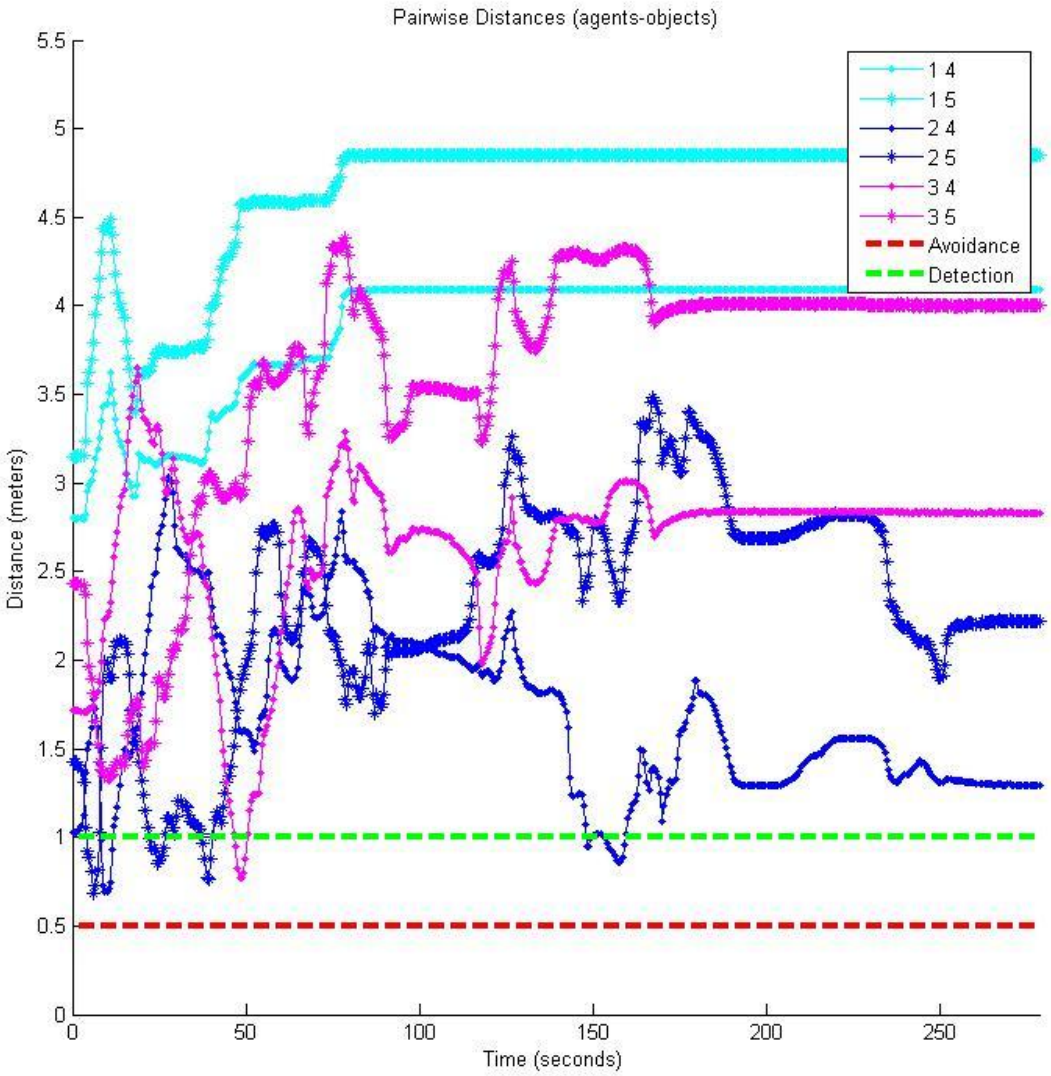

Figure 5.15: Experiment 3 Agent-Object Pairwise Distances

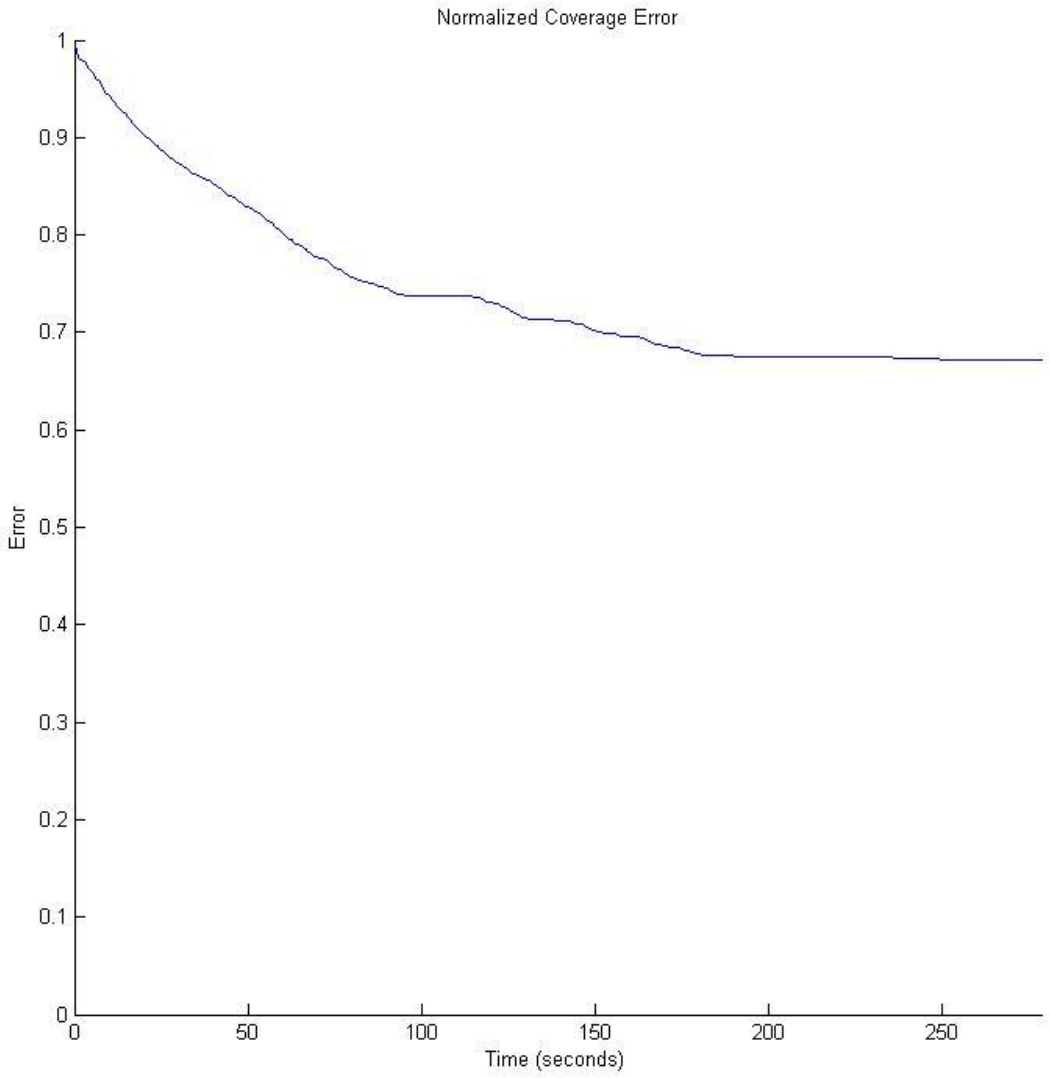

Figure 5.16: Experiment 3 Normalized Coverage Error

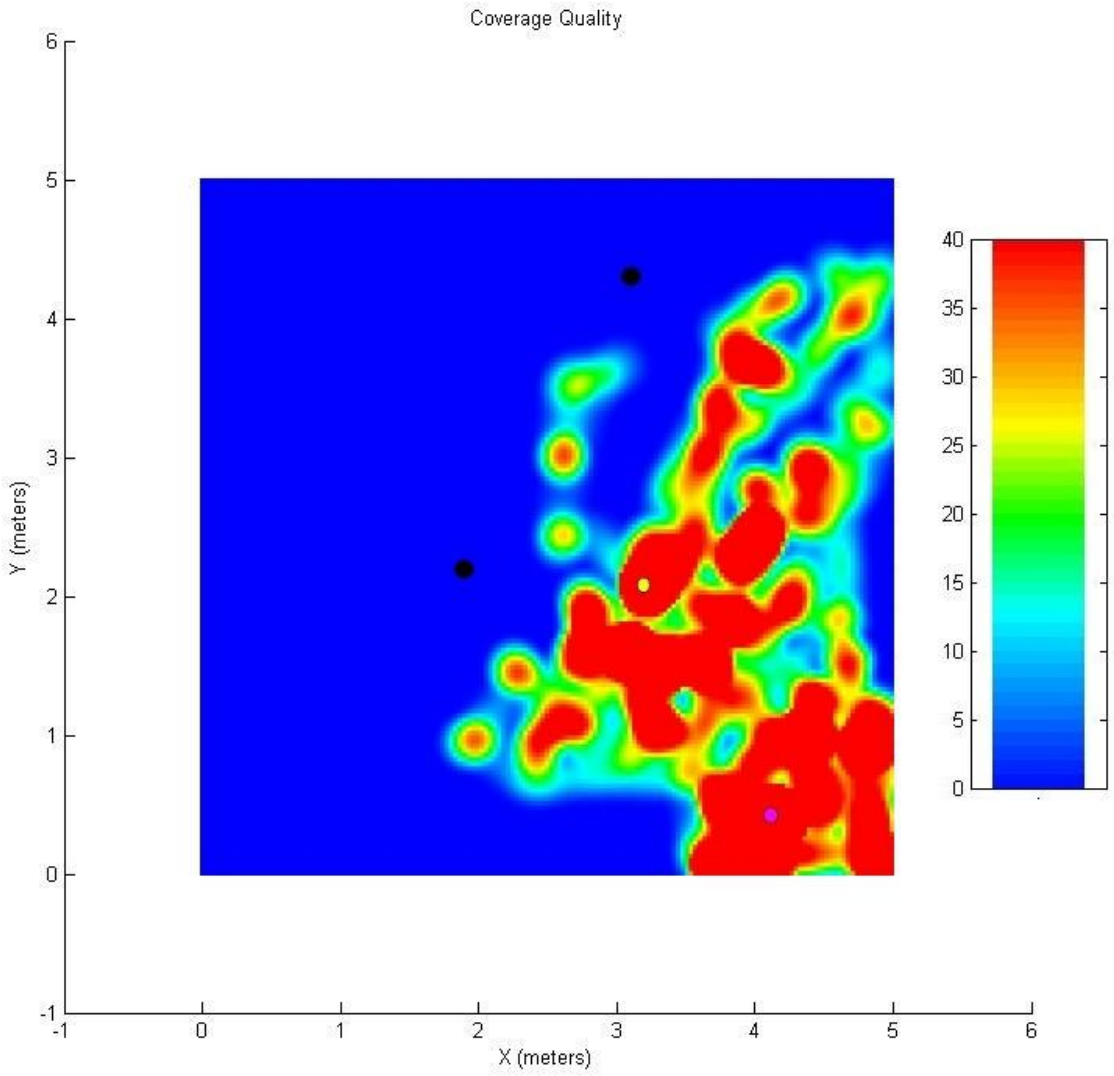

Figure 5.17: Experiment 3 Terminal Coverage Quality Heat Map (green = agent 1, yellow = agent 2, magenta = agent 3)

Experiment 4 takes a different approach at guaranteeing coverage of the entire area. Instead of assuming that all three robots have homogenous sensing regions that are constrained to very local regions centered at the center position of the individual robots, it is assumed that robot 1 has sensing capabilities over the entire domain. Robot 1 was given a sensing region radius, R_i from (10), of $R_1 = 5\sqrt{2}$ and a peak sensing capability, M_i from (10), of $M_1 = 1/5$. The other two robots, robots 2 and 3, retained the default sensing parameters with $R_2 = R_3 =$

0.25 and $M_2 = M_3 = 12$. In addition, robot 1 was free of any proximity and avoidance constraints in relation to other robots. However, it still had avoidance enabled for the obstacles. The results of Experiment 4 are shown in Figures 5.18 through 5.22. Satisfactory coverage of the entire area was reached when the coverage error function fell below 0.001 at T = 489.936 seconds. As in the previous experiments, the agents exhibited desirable proximity and avoidance. Also as in the previous experiments, the robots reached very near their terminal positions before coverage was completed. After this, robot 1 completed the coverage of the entire area relatively slowly using its relatively low peak sensing capabilities over the entire domain. Therefore, while this experiment guaranteed coverage of the entire domain, it still did not have entirely desirable characteristics.

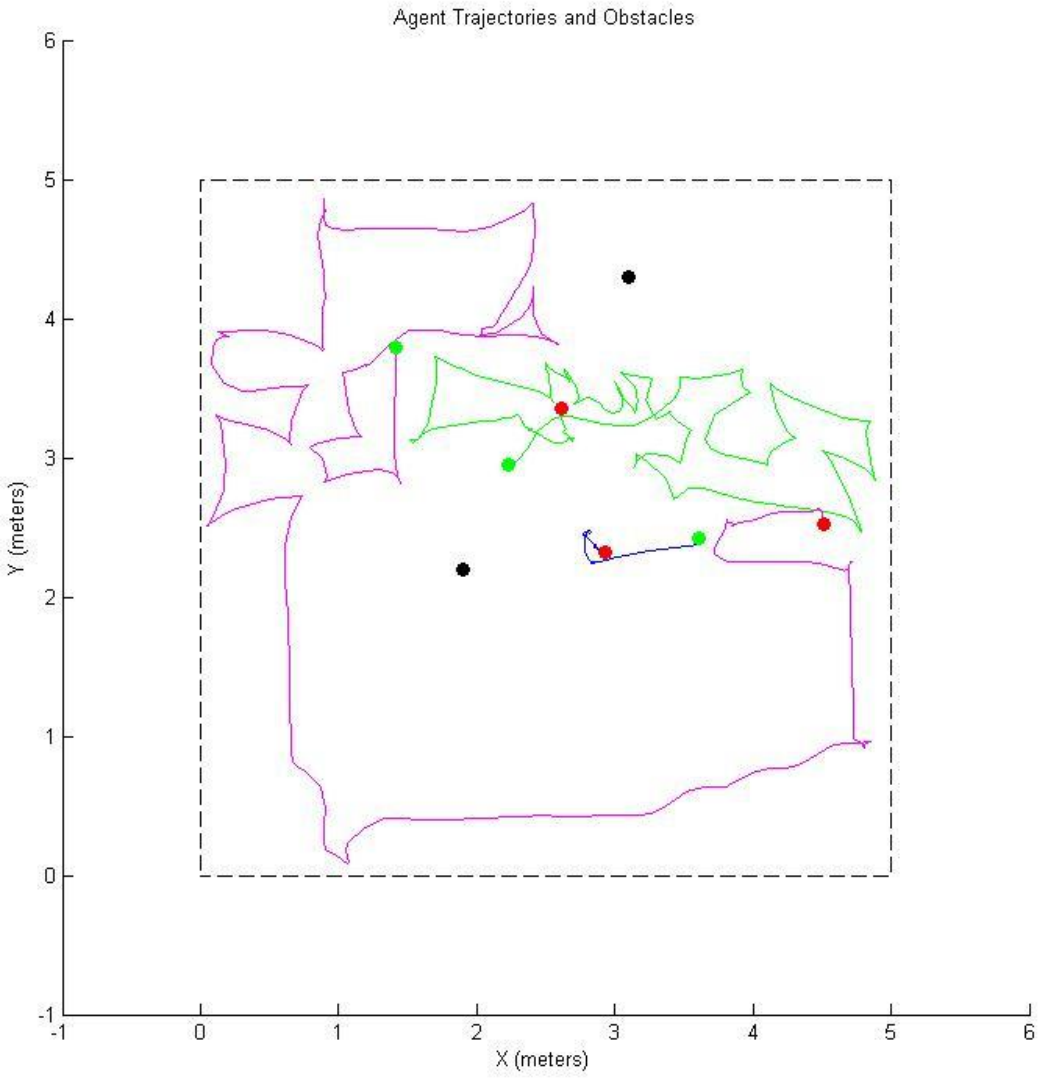

Figure 5.18: Experiment 4 Agent Trajectories and Object Positions Where Green Circles Are Start Positions and Red Circles Are Final Positions (blue = agent 1, green = agent 2, magenta = agent 3)

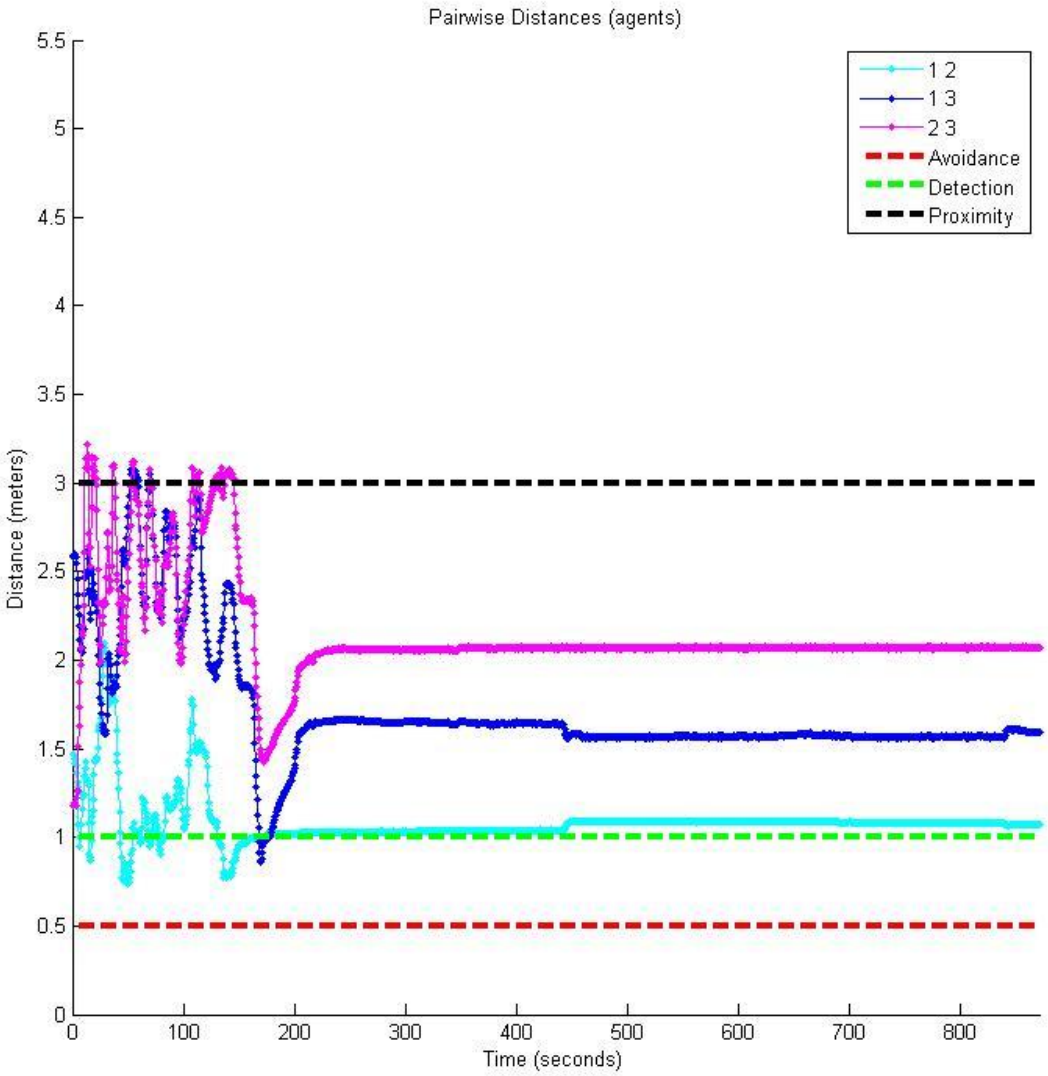

Figure 5.19: Experiment 4 Agent Pairwise Distances

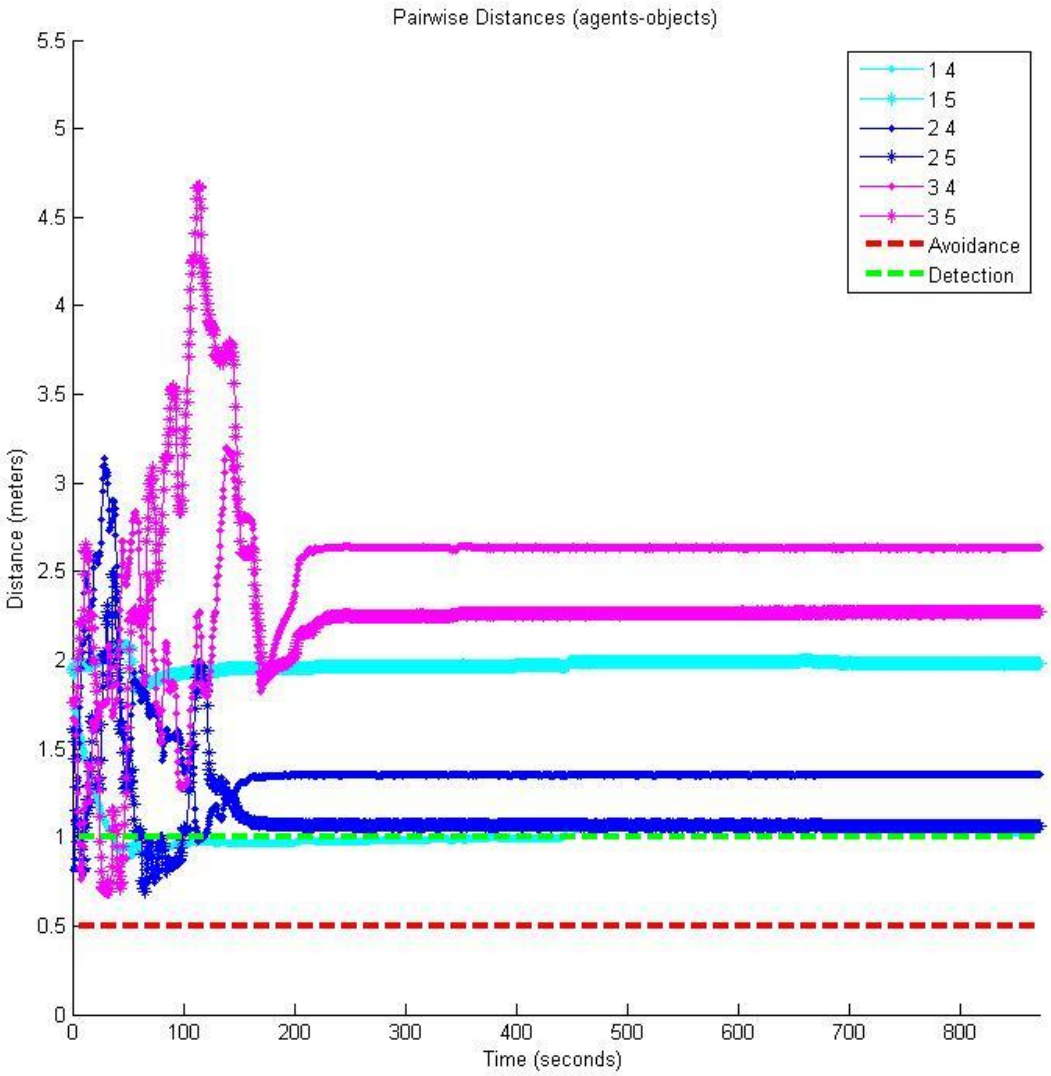

Figure 5.20: Experiment 4 Agent-Object Pairwise Distances

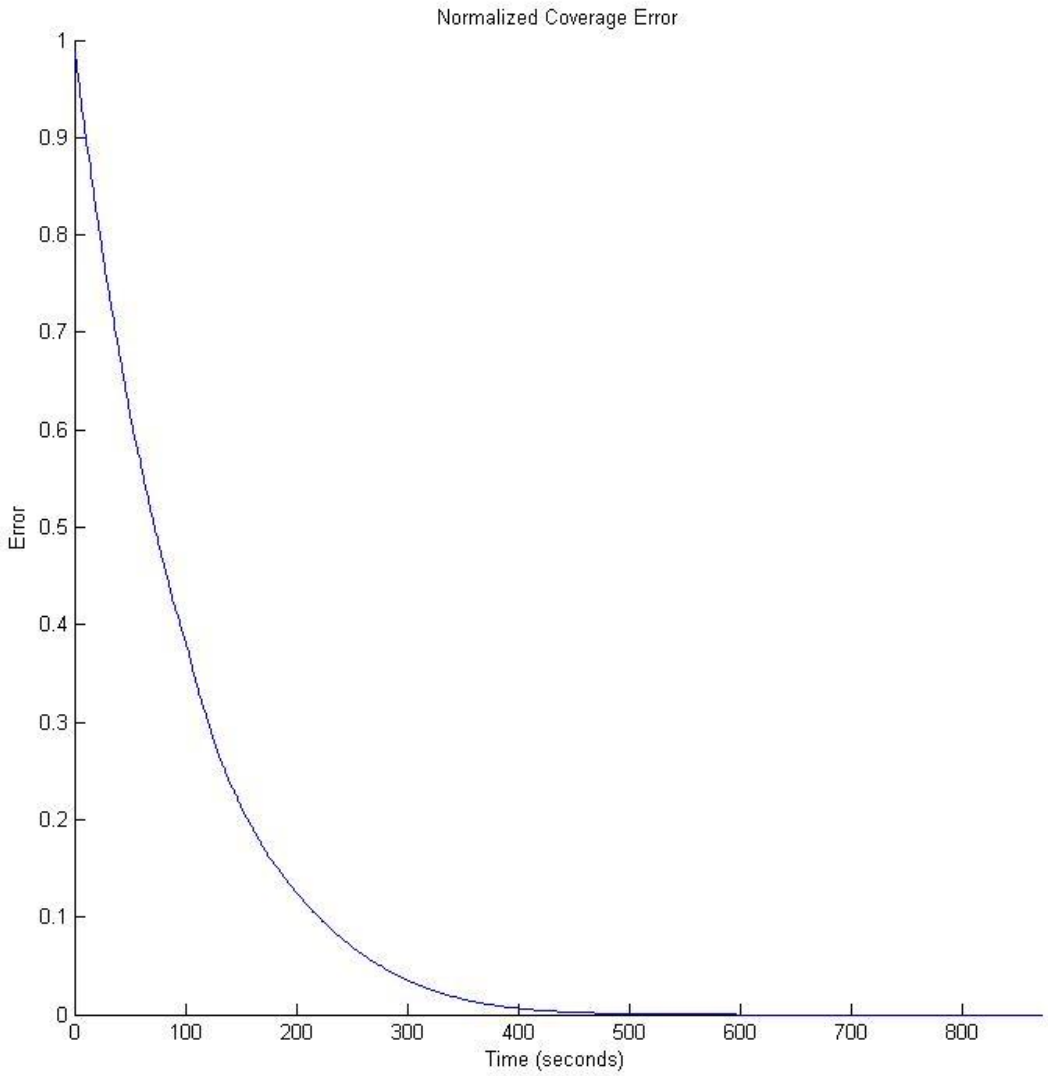

Figure 5.21: Experiment 4 Normalized Coverage Error

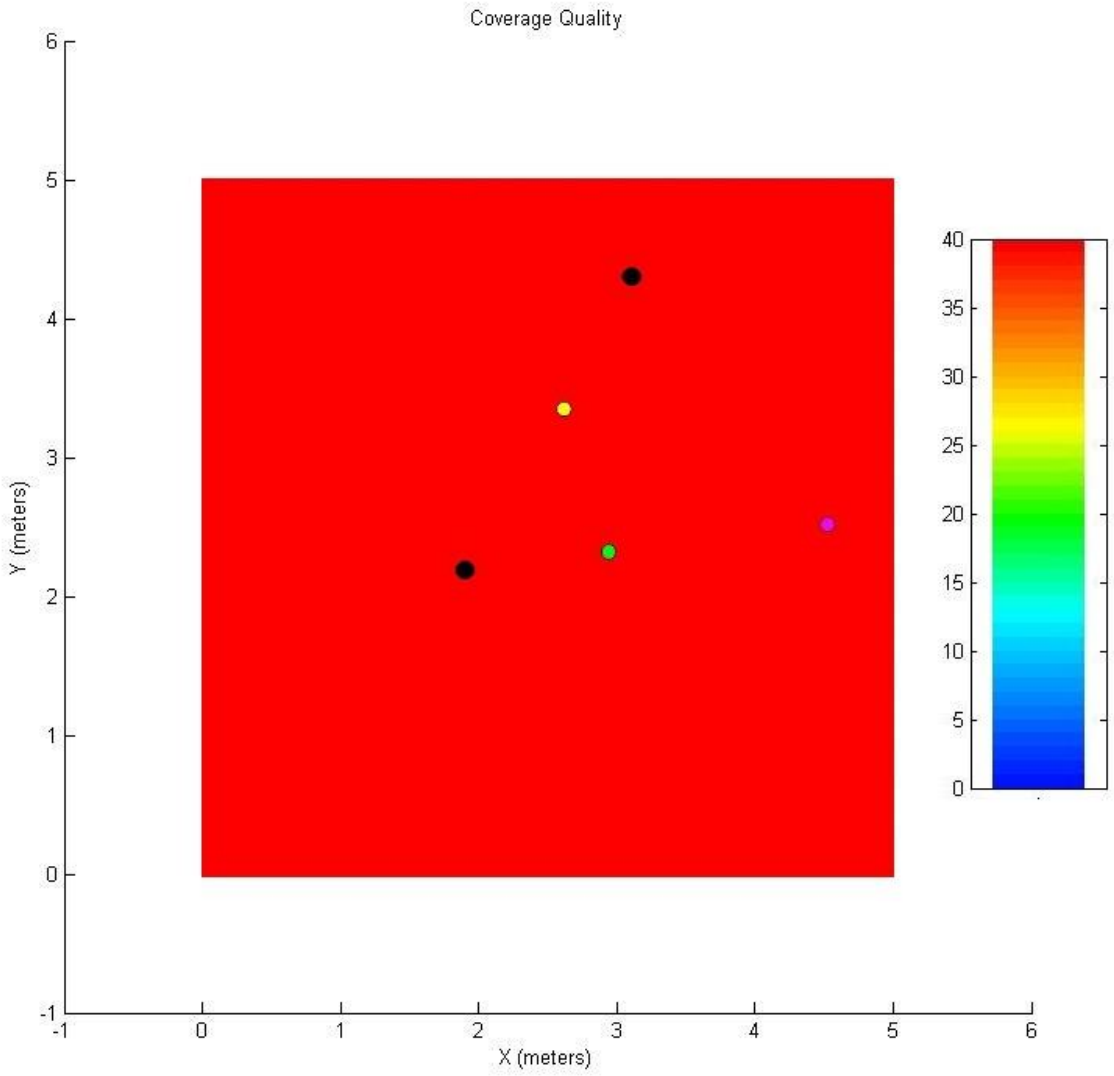

Figure 5.22: Experiment 4 Terminal Coverage Quality Heat Map (green = agent 1, yellow = agent 2, magenta = agent 3)

Experiment 5 takes a different approach to the coverage problem. In Experiment 5, the robots were once again set to have homogenous sensing parameters of $R_1 = R_2 = R_3 = 0.25$ $M_1 = M_2 = M_3 = 12$. Robot 1, instead of being controlled using the safe and reliable coverage control algorithm, was controlled by pursuing randomly generated target positions using a PI controller. Once robot 1 reached its targeted position, it was then assigned a new target position. Due to the removal of the avoidance control for robot 1, the obstacles were also removed from the experiment. Robots 2 and 3 were still controlled using the safe and reliable coverage control

algorithm with all of the default parameters from section 5.2. The results of this experiment are shown in Figures 5.23 through 5.26. The coverage error in this experiment never reached a value near zero due to the fact that one of the robots became disabled at the end of the experiment. This can be seen in Figure 5.24 when the pairwise distance between robots 1 and 2 entered deep in the avoidance region. Robot 1 continued to pursue its randomly assigned target positions, but robot 2 made no attempt at avoidance because it became disabled by a broken connection between the embedded Linux process handling communication and the Python script broadcasting the OpiTrack data. However, this experiment does show that there are alternative methods that may be used eliminate the possibility of a team of robots reaching an undesirable equilibrium. Depending on the application, additional logic or time-varying experimental parameters may enhance the performance of the system.

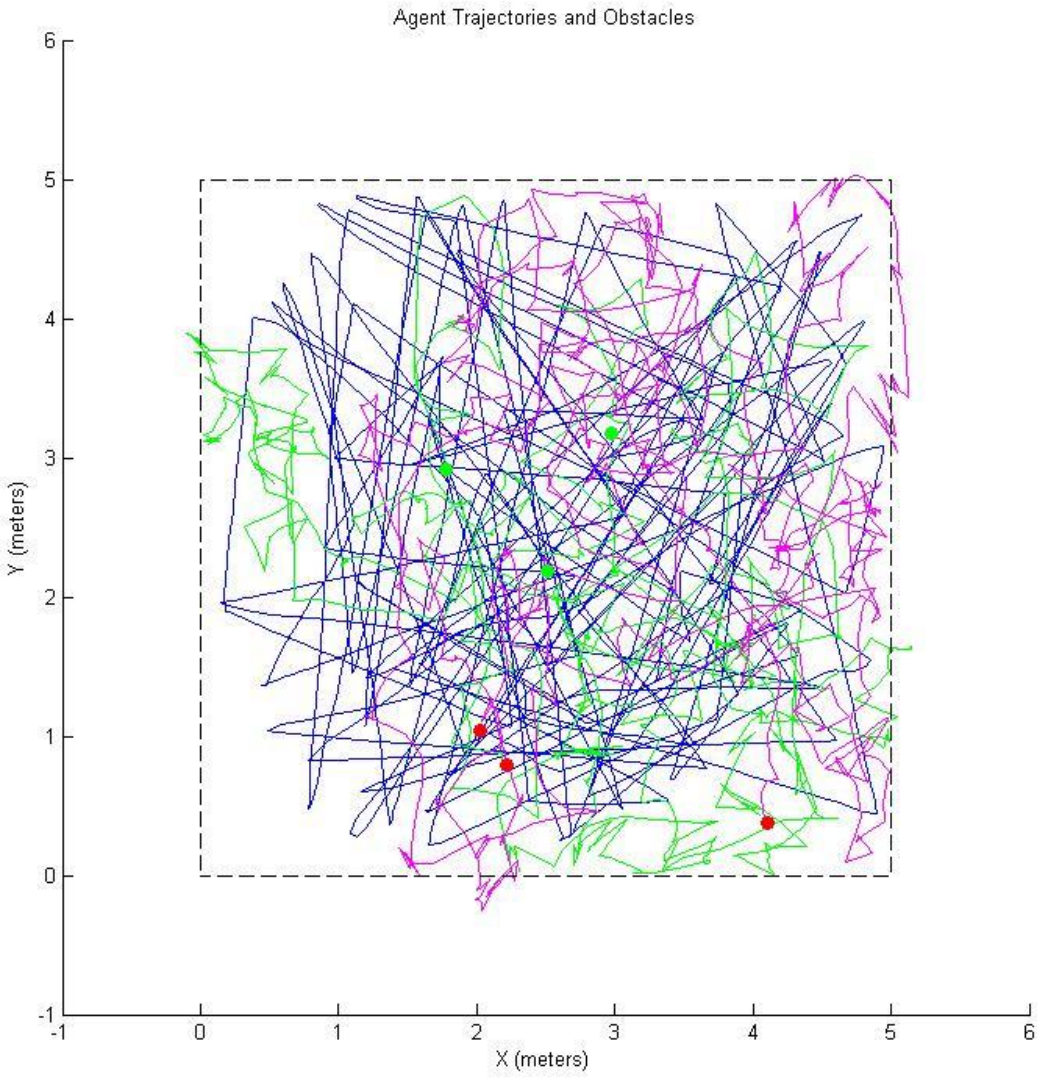

Figure 5.23: Experiment 5 Agent Trajectories and Object Positions Where Green Circles Are Start Positions and Red Circles Are Final Positions (blue = agent 1, green = agent 2, magenta = agent 3)

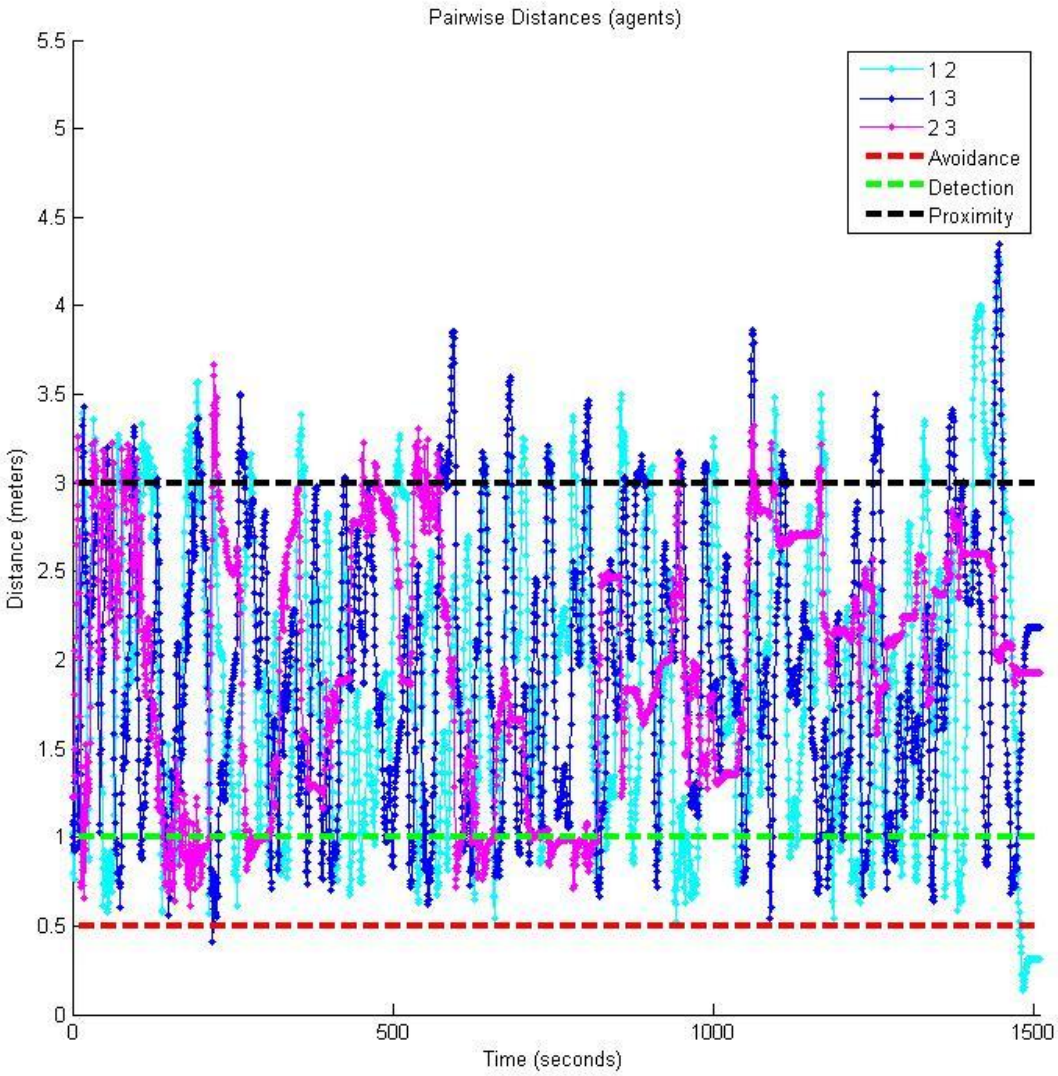

Figure 5.24: Experiment 5 Agent Pairwise Distances

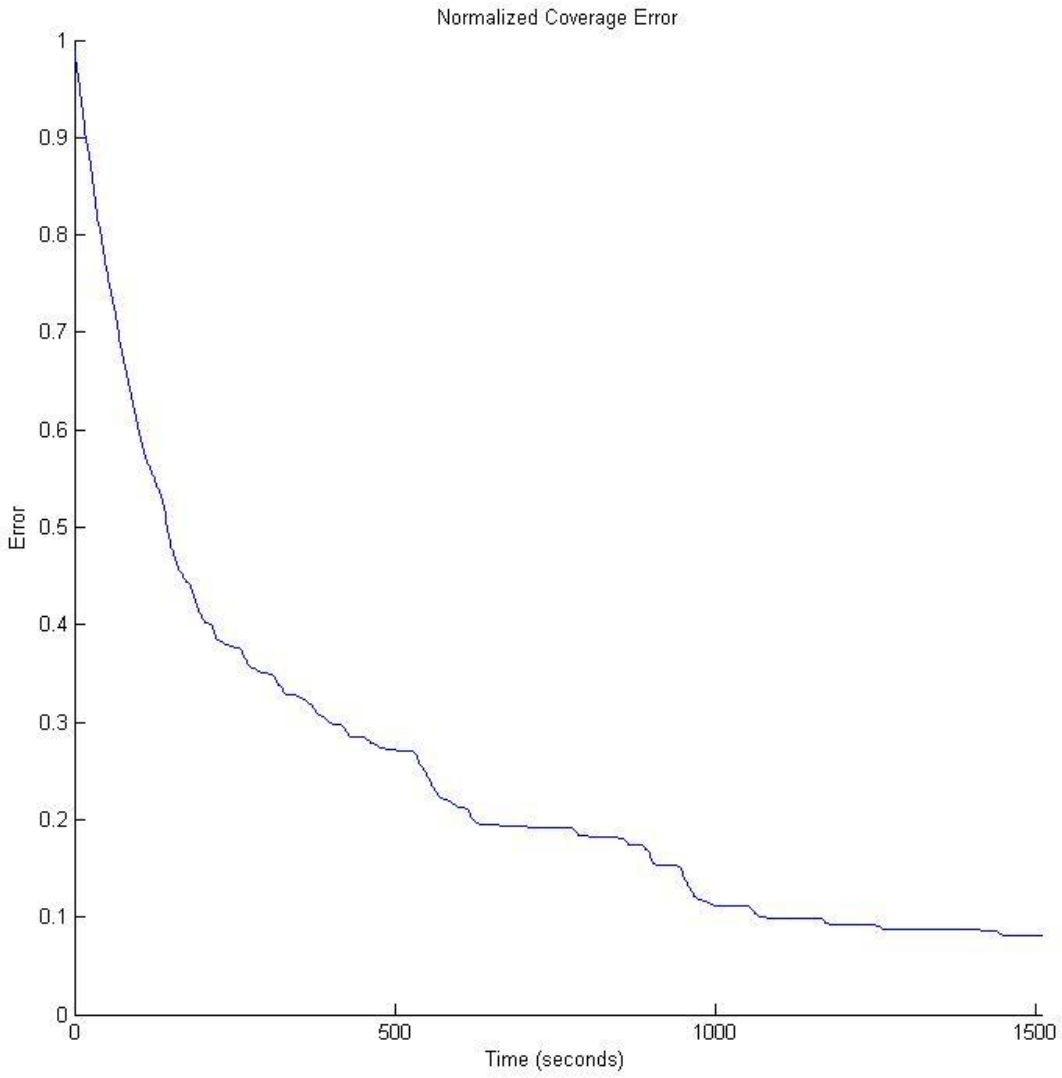

Figure 5.25: Experiment 5 Normalized Coverage Error

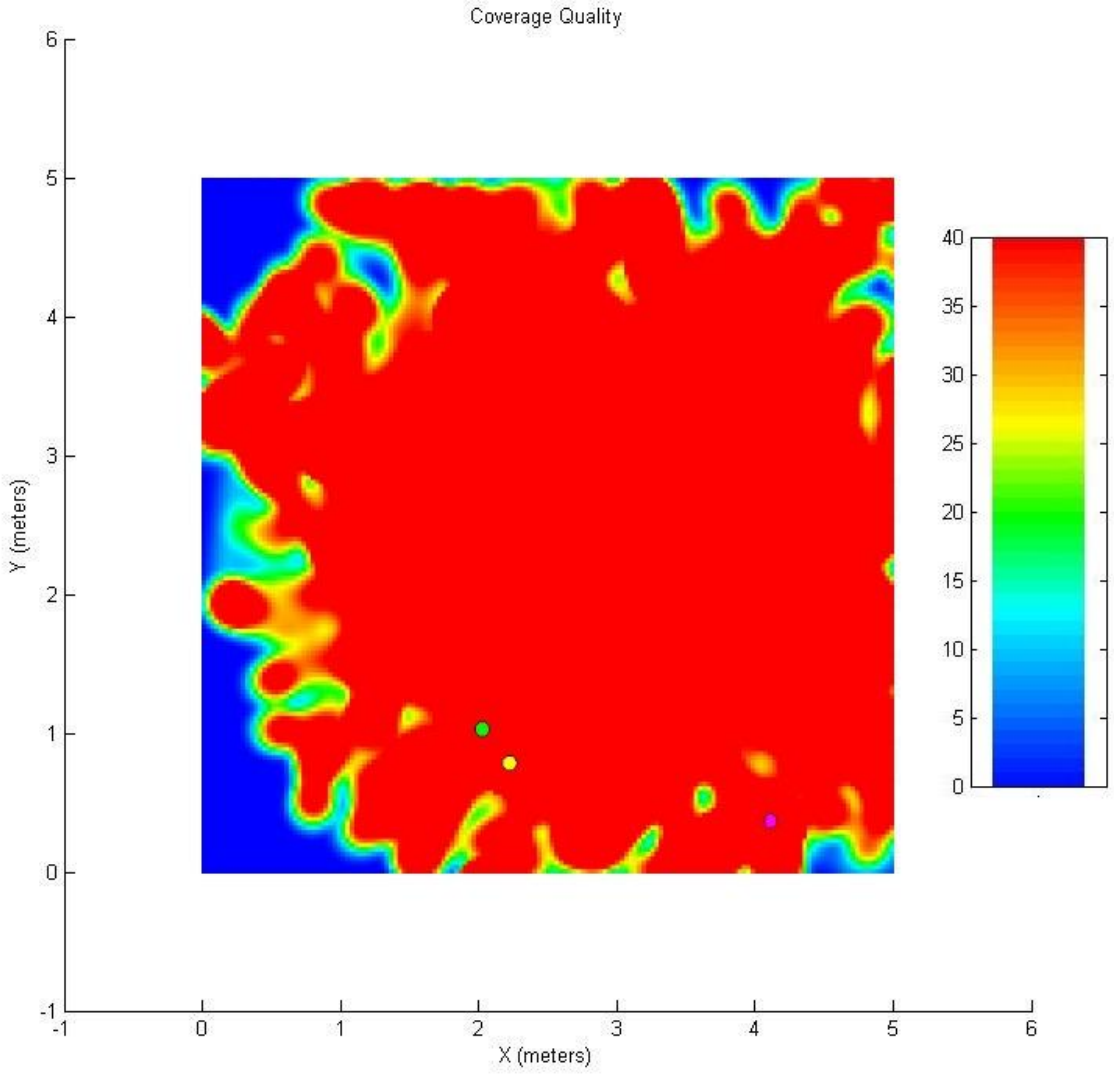

Figure 5.26: Experiment 5 Terminal Coverage Quality Heat Map (green = agent 1, yellow = agent 2, magenta = agent 3)

CHAPTER 6
CONCLUSIONS

In this thesis, a successful implementation of the safe and reliable coverage control algorithm simulated in [1] was demonstrated. A review of the methodologies in [1] for avoidance and proximity control, coverage control, and the merger of the multiple objectives was discussed. In addition, practical considerations for the avoidance and proximity control objective functions and gradients were presented. The experimental test bed, specifically the Natural Point OptiTrack motion capture system and the differential drive robots developed at the University of Illinois, were detailed. Experimental parameters that were analogous to the experimental parameters of the simulation chosen in [1] were applied to this implementation. The dynamic models of the differential drive robots were assumed to be nonlinear yet affine in control. The motivation and development of Kalman filtering for the positions and orientations of the robots was discussed. The experimental system performance of the safe and reliable coverage control algorithm was presented. While successful at preventing collisions and maintaining proximity between the robots, many of the experiments failed to guarantee coverage of the entire area defined with the assumptions made about sensing capabilities of the robots. The implementation proved to be a reliable testing platform for testing control strategies of this type involving carefully constructed objective functions for avoidance, proximity, and coverage. Additional work, such as attempts to overlay additional strategies for guaranteeing coverage, as demonstrated in the fifth experiment in Chapter 5, may offer improved system performance in terms of terminal coverage quality and the time required to reach satisfactory coverage.

REFERENCES

[1] D. M. Stipanović, C. Valicka, C. J. Tomlin, and T. R. Bewley, *Safe and Reliable Coverage Control*, Numerical Algebra, Control and Optimization, **3** (2013), 31-48.

[2] R. Siegwart, and I. R. Nourbakhsh. "Introduction to Autonomous Mobile Robots," The MIT Press, Cambridge, Massachusetts, 2004.

[3] D. M. Stipanović, P. F. Hokayem, M. W. Spong, and D. D. Šiljak, *Cooperative avoidance control for multi-agent systems*, Journal of Dynamic Systems, Measurement, and Control, **129** (2007), 699-707.

[4] I. I. Hussein and D. M. Stipanović, *Effective coverage control using dynamic sensor networks with flocking and guaranteed collision avoidance*, Proceedings of the 2007 American Control Conference, pp. 3420-3425 (2007).

[5] I. I. Hussein and D. M. Stipanović, *Effective coverage control using mobile sensor networks with guaranteed collision avoidance*, IEEE Transactions on Control Systems Technology **15** (2007), 642-657.

[6] D. M. Stipanović, C. J. Tomlin, and C. G. Valicka, *Collision free coverage control with multiple agents*, Proceedings of the RoMoCo'11 Conference, Bukowy Dworek, Poland, (2011).

[7] G. Welch and G. Bishop, *An Introduction to the Kalman Filter*, Department of Computer Science, University of North Carolina at Chapel Hill, TR 95-041, (2006).

[8] T. Baxter, *Adaptive Trajectory Tracking Control for a Nonholonomic Mobile Robot with Time Delay Compensation Through Kalman Filtering*, Master's Thesis, University of Illinois at Urbana-Champaign, (2007).

APPENDIX A

OMAPL138 DSP Code

This section includes a selected portion of the DSP code that implemented the safe and reliable coverage control algorithm on the robots. This included code does not necessarily reflect the state of the code for any one specific experiment conducted, but is provided to demonstrate the general methods of implementation.

```
#include <std.h>
#include <log.h>
#include <clk.h>
#include <gbl.h>
#include <bcache.h>

#include <mem.h> // MEM_alloc calls
#include <que.h> // QUE functions
#include <sem.h> // Semaphore functions
#include <sys.h>
#include <tsk.h> // TASK functions
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <c6x.h> // register defines


#include "projectinclude.h"
#include "c67fastMath.h" // sinsp,cossp, tansp
#include "evmomapl138.h"
#include "evmomapl138_i2c.h"
#include "evmomapl138_timer.h"
#include "evmomapl138_led.h"
#include "evmomapl138_dip.h"
#include "evmomapl138_gpio.h"
#include "evmomapl138_vpif.h"
#include "evmomapl138_spi.h"
#include "COECSL_edma3.h"
#include "COECSL_mcbsp.h"
#include "COECSL_registers.h"

#include "mcbsp_com.h"
#include "ColorVision.h"
#include "ColorLCD.h"
#include "sharedmem.h"
#include "LCDprintf.h"
#include "MatrixMath.h"
#include "xy.h"

#define NUM_OBSTACLES 2
#define NUM_AVOIDANCE (NUM_TRACKABLES+NUM_OBSTACLES)
#define AVOID_ON 1
#define PROX_ON 1
#define SPECIAL_LEADER 0  // 1 = leader sensing, no avoiding other robots
#define SPECIAL_LEADER2 0  // 1 = controller / no avoid/prox influence on utheta
#define SPECIAL_LEADER3 0  // 1 = no prox influence on leader
#define SPECIAL_LEADER4 0  // goto x,y control for leader
#define D_PROX 3 // was 1.5
#define SETTLETIME 6000 // gyro settling time
#define ProcUncert 0.0001  // Kalman filter uncertainties
#define CovScalar 10
#define MeasUncert 1 // was 10

extern unsigned long timeint;
extern float enc1;  // Left motor encoder
extern float enc2;  // Right motor encoder
extern float enc3;
extern float enc4;
```

```
extern float adcA0;  // ADC A0 - Gyro_X -400deg/s to 400deg/s  Pitch
extern float adcB0;  // ADC B0 - External ADC Ch4 (no protection circuit)
extern float adcA1;  // ADC A1 - Gyro_4X -100deg/s to 100deg/s  Pitch
extern float adcB1;  // ADC B1 - External ADC Ch1
extern float adcA2;  // ADC A2 -    Gyro_4Z -100deg/s to 100deg/s  Yaw
extern float adcB2;  // ADC B2 - External ADC Ch2
extern float adcA3;  // ADC A3 - Gyro_Z -400deg/s to 400 deg/s  Yaw
extern float adcB3;  // ADC B3 - External ADC Ch3
extern float adcA4;  // ADC A4 - Analog IR1
extern float adcB4;  // ADC B4 - USONIC1
extern float adcA5;  // ADC A5 -    Analog IR2
extern float adcB5;  // ADC B5 - USONIC2
extern float adcA6;  // ADC A6 - Analog IR3
extern float adcA7;  // ADC A7 - Analog IR4
extern float adc1_i2c;  // Not connected
extern float adc2_i2c;  // Not connected
extern float adc3_i2c;  // Not connected
extern float adc4_i2c;  // Not connected
extern float compass;
extern float switchstate;

extern volatile int new_sendtolinux_vision;
extern volatile float send_object_x;
extern volatile float send_object_y;
extern volatile int send_numpels;

extern volatile int new_sendtolinux_tcpip;
extern volatile float send_tcpip1;
extern volatile float send_tcpip2;
extern volatile float send_tcpip3;
extern volatile float send_tcpip4;

extern volatile int new_sendtolinux;
extern volatile float send_object_x;
extern volatile float send_object_y;
extern volatile int send_numpels;

extern float tcpip1;
extern float tcpip2;
extern float tcpip3;
extern float tcpip4;

extern sharedmemstruct *ptrshrdmem;
sharedmemstruct2 *ptrshrdmem2;


volatile uint32_t index;
uint8_t LinuxBooted = 0;

float temp_ang = 0;
float temp_ot = 0;
float temp_kal = 0;
int errorcheck = 1;

// wall follow variables
float ir1 = 0;
float ir2 = 0;
float Rwall = 2500;
float Fwall = 2000;
float Kpright = 0.0012;
float Kpfront = 0.002;
float Turn_vel = 0;
float Turn_sat = 3.0;
float Vel_forward = 0.5;
float front_wall_error = 0;
float right_wall_error = 0;

float vref = 0;
float turn = 0;

int otkalcount = 0;
int tskcount = 0;

char fromLinuxstring[LINUX_COMSIZE + 2];
char toLinuxstring[LINUX_COMSIZE + 2];

float VBDAC1 = 0;
float VBDAC2 = 0;
float VBDAC1_send = 0;
float VBDAC2_send = 0;
int new_VB_data = 0;
```

```
float value_tcpip1 = 0.0;
float value_tcpip2 = 0.0;
float value_tcpip3 = 0.0;
float value_tcpip4 = 0.0;

float value_object_x = 0;
float value_object_y = 0;
float value_numpels = 0;

int newnavdata = 0;
float newvref = 0;
float newturn = 0;

unsigned char controllerdata[65];

// OPTITRACK/ALGORITHM VARIABLES
float dx = 0.03125; // was 0.03125 for size = 5, 0.01875 for size = 3
float dy = 0.03125;
int size = 5;
int gridsize = 160;
int C = 40;
dataset *alldata;
float *Q_global;
int *Q_linux;
unsigned char *sharedQTmem;
float M[NUM_TRACKABLES];
float R_cov[NUM_TRACKABLES];
float R_cov2[NUM_TRACKABLES];
float M_over_R4[NUM_TRACKABLES];
int trackableID = -1;
float ax[NUM_TRACKABLES];
float ay[NUM_TRACKABLES];
float ce = 0;
float P[4] = {1,0,0,1};
float vP[NUM_TRACKABLES*NUM_AVOIDANCE];
float dvPx[NUM_TRACKABLES*NUM_AVOIDANCE];
float dvPy[NUM_TRACKABLES*NUM_AVOIDANCE];
float vA[NUM_TRACKABLES*NUM_AVOIDANCE];
float dvAx[NUM_TRACKABLES*NUM_AVOIDANCE];
float dvAy[NUM_TRACKABLES*NUM_AVOIDANCE];
float R_col[NUM_TRACKABLES];
float r_col[NUM_TRACKABLES];
float dvi_dx[NUM_TRACKABLES];
float dvi_dy[NUM_TRACKABLES];
float theta_des_ap[NUM_TRACKABLES];
float theta_des_cov[NUM_TRACKABLES];
float theta_des[NUM_TRACKABLES];
float AP_gain[NUM_TRACKABLES] = {-1.25,-1.25,-1.25};
float cov_gain[NUM_TRACKABLES] = {0.00015,0.00015,0.00015}; // k_cov = .0008*[.35 .0041 .062];  // was 0.0005
for each
float theta_gain[NUM_TRACKABLES] = {-1.0,-1.0,-1.0};
int trackableIDerror = 0;
int firstdata = 1;
volatile int new_optitrack = 0;
volatile int new_optitrack_kal = 0;
volatile float previous_frame = -1;
int frame_error = 0;
volatile float Optitrackdata[OPTITRACKDATASIZE];
volatile long currtime = 0;
volatile long prevtime = 0;
volatile long swi_time = 0;
volatile long swi_time_prev = 0;
volatile long optitrack_rectime = 0;
volatile int temp_trackableID = -1;
volatile float dt = 0;
int firstime = 1;

float gp[NUM_TRACKABLES] = {0.075,0.075,0.075};  // was 0.025
float ga[NUM_TRACKABLES] = {0.025,0.025,0.025}; // was 0.02
float gamma_prox[NUM_TRACKABLES*NUM_TRACKABLES] = {0,1,1,
                                                   1,0,1,
                                                   1,1,0};
float gamma_avoid[NUM_TRACKABLES*NUM_AVOIDANCE];

float angle_diff1 = 0;
float angle_diff2 = 0;
float remain = 0;
float theta_des_norm[NUM_TRACKABLES] = {0,0,0};
float theta_control_temp = 0.0;
float theta_des_temp = 0.0;
float temp_theta = 0.0;
```

```
volatile float xpos_ot[NUM_AVOIDANCE] = {-1.0,-2.0,-3.0,1.9,3.1};//,4.0,5.0};
volatile float ypos_ot[NUM_AVOIDANCE] = {-1.0,-2.0,-3.0,2.2,4.3};//,4.0,5.0};
volatile float theta_ot[NUM_AVOIDANCE] =  {0,0,0,0,0};//,0,0};
volatile float xpos_kal[NUM_AVOIDANCE] = {-1.0,-2.0,-3.0,1.9,3.1};//,4.0,5.0};
volatile float ypos_kal[NUM_AVOIDANCE] = {-1.0,-2.0,-3.0,2.2,4.3};//,4.0,5.0};
volatile float theta_kal[NUM_AVOIDANCE] =  {0,0,0,0,0};//,0,0};
volatile float xpos_control[NUM_AVOIDANCE] = {-1.0,-2.0,-3.0,1.9,3.1};//,4.0,5.0};
volatile float ypos_control[NUM_AVOIDANCE] = {-1.0,-2.0,-3.0,2.2,4.3};//,4.0,5.0};
volatile float theta_control[NUM_AVOIDANCE] =  {0,0,0,0,0};//,0,0};

float u_control = 0;
float utheta_control = 0;
float u_out = 0;
float utheta_out = 0;
float ce_control = 0;

float gotox = 0;
float gotoy = 0;
float gotox_control = 0;
float gotoy_control = 0;

volatile int updateControlOut = 0;
volatile int updateControlPos = 1;

// KALMAN FILTER VARIABLES
float vel1 = 0,vel2 = 0;
float vel1old = 0,vel2old = 0;
float enc1old = 0,enc2old = 0;

// SETTLETIME should be an even number and divisible by 3
int settlegyro = 0;
float gyro_zero = 0;
float gyro_angle = 0;
float old_gyro = 0;
float gyro_drift = 0;
float gyro = 0;
int gyro_degrees = 0;
float gyro_radians = 0.0;
float gyro_x = 0,gyro_y = 0;
float gyro4x_gain = 1.01;

// KALMAN FILTERING
float x_pred[3][1] = {{0},{0},{0}};                 // predicted state

//more kalman vars
float B[3][2] = {{1,0},{1,0},{0,1}};            // control input model
float u[2][1] = {{0},{0}};          // control input in terms of velocity and angular velocity
float Bu[3][1] = {{0},{0},{0}}; // matrix multiplication of B and u
float z[3][1];                              // state measurement
float eye3[3][3] = {{1,0,0},{0,1,0},{0,0,1}};   // 3x3 identity matrix
float K[3][3] = {{1,0,0},{0,1,0},{0,0,1}};      // optimal Kalman gain
float Q[3][3] = {{ProcUncert,0,ProcUncert/CovScalar},
                 {0,ProcUncert,ProcUncert/CovScalar},
                 {ProcUncert/CovScalar,ProcUncert/CovScalar,ProcUncert}};   // process noise (covariance of
encoders and gyro)
float R[3][3] = {{MeasUncert,0,MeasUncert/CovScalar},
                 {0,MeasUncert,MeasUncert/CovScalar},
                 {MeasUncert/CovScalar,MeasUncert/CovScalar,MeasUncert}};   // measurement noise (covariance of
LADAR)
float S[3][3] = {{1,0,0},{0,1,0},{0,0,1}};  // innovation covariance
float S_inv[3][3] = {{1,0,0},{0,1,0},{0,0,1}};  // innovation covariance matrix inverse
float P_pred[3][3] = {{1,0,0},{0,1,0},{0,0,1}}; // predicted covariance (measure of uncertainty for current
position)
float temp_3x3[3][3];               // intermediate storage
float temp_3x1[3][1];               // intermediate storage
float ytilde[3][1];                 // difference between predictions
float start_laser_pred[3][1];       // keeps track of the prediction when the LADAR has new data

// leader goes to least covered inits
float minX = 0.0;
float minY = 0.0;
float Qmin = 40.0;

// USED FOR GYRO CAL
// a pose (position and orientation) of the robot
typedef struct
{
    float x;        //in feet
    float y;        //in feet
    float theta;    // in radians between -PI and PI.  O radians is along the +x axis, PI/2 is the +y axis
```

```
} pose;

int statePos = 0;   // index into robotdest
int robotdestSize = 4;  // number of positions to use out of robotdest
pose robotdest[4];  // array of waypoints for the robot

float Q_average = 0;
float Q_min = 40;
float Q_average_print = 0;
int num_pels_print = 0;

/*
 *  ======== main ========
 */
Void main()
{


    int i = 0;
    int j = 0;
    int k = 0;


   // unlock the system config registers.
   SYSCONFIG->KICKR[0] = KICK0R_UNLOCK;
   SYSCONFIG->KICKR[1] = KICK1R_UNLOCK;

   SYSCONFIG1->PUPD_SEL |= 0x10000000;  // change pin group 28 to pullup for GP7[12/13] (LCD switches)

   // Initially set McBSP1 pins as GPIO ins
   CLRBIT(SYSCONFIG->PINMUX[1], 0xFFFFFFFF);
   SETBIT(SYSCONFIG->PINMUX[1], 0x88888880);  // This is enabling the McBSP1 pins

   CLRBIT(SYSCONFIG->PINMUX[16], 0xFFFF0000);
   SETBIT(SYSCONFIG->PINMUX[16], 0x88880000);  // setup GP7.8 through GP7.13
   CLRBIT(SYSCONFIG->PINMUX[17], 0x000000FF);
   SETBIT(SYSCONFIG->PINMUX[17], 0x00000088);  // setup GP7.8 through GP7.13


   //Rick added for LCD DMA flagging test
   GPIO_setDir(GPIO_BANK0, GPIO_PIN8, GPIO_OUTPUT);
   GPIO_setOutput(GPIO_BANK0, GPIO_PIN8, OUTPUT_HIGH);

   GPIO_setDir(GPIO_BANK0, GPIO_PIN0, GPIO_INPUT);
   GPIO_setDir(GPIO_BANK0, GPIO_PIN1, GPIO_INPUT);
   GPIO_setDir(GPIO_BANK0, GPIO_PIN2, GPIO_INPUT);
   GPIO_setDir(GPIO_BANK0, GPIO_PIN3, GPIO_INPUT);
   GPIO_setDir(GPIO_BANK0, GPIO_PIN4, GPIO_INPUT);
   GPIO_setDir(GPIO_BANK0, GPIO_PIN5, GPIO_INPUT);
   GPIO_setDir(GPIO_BANK0, GPIO_PIN6, GPIO_INPUT);

   GPIO_setDir(GPIO_BANK7, GPIO_PIN8, GPIO_OUTPUT);
   GPIO_setDir(GPIO_BANK7, GPIO_PIN9, GPIO_OUTPUT);
   GPIO_setDir(GPIO_BANK7, GPIO_PIN10, GPIO_OUTPUT);
   GPIO_setDir(GPIO_BANK7, GPIO_PIN11, GPIO_OUTPUT);
   GPIO_setDir(GPIO_BANK7, GPIO_PIN12, GPIO_INPUT);
   GPIO_setDir(GPIO_BANK7, GPIO_PIN13, GPIO_INPUT);

   GPIO_setOutput(GPIO_BANK7, GPIO_PIN8, OUTPUT_HIGH);
   GPIO_setOutput(GPIO_BANK7, GPIO_PIN9, OUTPUT_HIGH);
   GPIO_setOutput(GPIO_BANK7, GPIO_PIN10, OUTPUT_HIGH);
   GPIO_setOutput(GPIO_BANK7, GPIO_PIN11, OUTPUT_HIGH);

   CLRBIT(SYSCONFIG->PINMUX[13], 0xFFFFFFFF);
   SETBIT(SYSCONFIG->PINMUX[13], 0x88888811); //Set GPIO 6.8-13 to GPIOs and IMPORTANT Sets GP6[15] to
/RESETOUT used by PHY, GP6[14] CLKOUT appears unconnected

   GPIO_setDir(GPIO_BANK6, GPIO_PIN8, GPIO_OUTPUT);
   GPIO_setDir(GPIO_BANK6, GPIO_PIN9, GPIO_OUTPUT);
   GPIO_setDir(GPIO_BANK6, GPIO_PIN10, GPIO_OUTPUT);
   GPIO_setDir(GPIO_BANK6, GPIO_PIN11, GPIO_OUTPUT);
   GPIO_setDir(GPIO_BANK6, GPIO_PIN12, GPIO_OUTPUT);
   GPIO_setDir(GPIO_BANK6, GPIO_PIN13, GPIO_INPUT);

   // flag pins
   GPIO_setDir(IMAGE_TO_LINUX_BANK, IMAGE_TO_LINUX_FLAG, GPIO_OUTPUT);
   GPIO_setDir(CONTINUOUSDATA_TO_LINUX_BANK, CONTINUOUSDATA_TO_LINUX_FLAG, GPIO_OUTPUT);
   GPIO_setDir(DATA_TO_LINUX_BANK, DATA_TO_LINUX_FLAG, GPIO_OUTPUT);
   GPIO_setDir(DATA_FROM_LINUX_BANK, DATA_FROM_LINUX_FLAG, GPIO_OUTPUT);
```

```
    LinuxBooted = GET_LINUX_BOOTED;
    if (LinuxBooted == 1) {
        while ((T1_TGCR & 0x7) != 0x7) {
          for (index=0;index<50000;index++) {}  // small delay before checking again

        }
    } else {
        EVMOMAPL138_lpscTransition(PSC0, DOMAIN0, LPSC_TPCC, PSC_ENABLE);
        EVMOMAPL138_lpscTransition(PSC0, DOMAIN0, LPSC_TPTC0, PSC_ENABLE);
        EVMOMAPL138_lpscTransition(PSC0, DOMAIN0, LPSC_TPTC1, PSC_ENABLE);

        // configure the next state for psc1 modules.
        EVMOMAPL138_lpscTransition(PSC1, DOMAIN0, LPSC_EDMA3CC1, PSC_ENABLE);
        EVMOMAPL138_lpscTransition(PSC1, DOMAIN0, LPSC_TPTC2, PSC_ENABLE);

        T1_TCR = 0;  // Disable timer and set to internal Clock
        T1_TGCR = 0;  // Reset both 32bit timers
        T1_TGCR = TGCR_TIMMODE_32BIT_UNCHAINED;  // set 32bit unchained
        T1_TGCR |= (TGCR_TIM12_RESET | TGCR_TIM34_RESET);  // pull timers out of reest
        T1_TIM12 = 0;  // zero count register
        T1_TIM34 = 0; // zero count register

    }
    USTIMER_init();

    // Turn on McBSP1
    EVMOMAPL138_lpscTransition(PSC1, DOMAIN0, LPSC_MCBSP1, PSC_ENABLE);

    if (LinuxBooted == 1) {
        USTIMER_delay(4*DELAY_1_SEC);  // delay allowing Linux to partially boot before continuing with DSP
code
    }

    // init the us timer and i2c for all to use.
    I2C_init(I2C0, I2C_CLK_100K);
    init_ColorVision();
    init_LCD_mem(); // added rick

    EVTCLR0 = 0xFFFFFFFF;
    EVTCLR1 = 0xFFFFFFFF;
    EVTCLR2 = 0xFFFFFFFF;
    EVTCLR3 = 0xFFFFFFFF;

    init_DMA();
    init_McBSP();

    CLRBIT(SYSCONFIG->PINMUX[1], 0xFFFFFFFF);
    SETBIT(SYSCONFIG->PINMUX[1], 0x22222220);  // This is enabling the McBSP1 pins

    CLRBIT(SYSCONFIG->PINMUX[5], 0x00FF0FFF);
    SETBIT(SYSCONFIG->PINMUX[5], 0x00110111);  // This is enabling SPI pins

    CLRBIT(SYSCONFIG->PINMUX[16], 0xFFFF0000);
    SETBIT(SYSCONFIG->PINMUX[16], 0x88880000);  // setup GP7.8 through GP7.13
    CLRBIT(SYSCONFIG->PINMUX[17], 0x000000FF);
    SETBIT(SYSCONFIG->PINMUX[17], 0x00000088);  // setup GP7.8 through GP7.13

    init_LCD();

    // ADDED FOR OPTITRACK
    alldata = (dataset *)(CONTROL_MEM_BASE); // external memory  //    is C4000000
    Q_global = (float *)(CONTROL_MEM_BASE+CONTROL_OFFSET*2); // should be C4400000
    Q_linux = (int *)(CONTROL_MEM_BASE+CONTROL_OFFSET*3);  // should be C4600000
    ptrshrdmem2 = (sharedmemstruct2 *)(SHARED_MEM+0x2000);  // should be 80005000
    sharedQTmem = (unsigned char *)(SHARED_MEM+0x3000);

    // Initialize shared memory flags
    SET_CONTINUOUSDATA_TO_LINUX;
    CLR_DATA_FROM_LINUX;
    SET_DATA_TO_LINUX;
    SET_IMAGE_TO_LINUX;

    // Initialize sensing and avoidance variables
    for (i=0;i<NUM_TRACKABLES;i++){
        if (i==0 && SPECIAL_LEADER) {
            R_cov[i] = size*sqrtsp(2);///4.0; // was size*sqrt(2)
            M[i] = 1.0/10.0;  // was 1/75
        } else {
            R_cov[i] = 0.25;//size*sqrtsp(2);//0.25; // was 0.3125
            M[i] = 12.0;  // was 4
        }
```

```
        R_cov2[i] = R_cov[i]*R_cov[i];
        M_over_R4[i] = M[i]/((R_cov[i])*(R_cov[i])*(R_cov[i])*(R_cov[i]));
        //M_over_R4[i] = M[i]/powsp(R_cov[i],32);

        R_col[i] = 1.0;  // was 10
        r_col[i] = 0.5;  // was 5
    }

    // Initialize gammas for proximity/avoidance
    for (i=0;i<NUM_TRACKABLES;i++) {
        for (j=0;j<NUM_TRACKABLES;j++) {
            gamma_prox[i*NUM_TRACKABLES+j] = gp[i]*gamma_prox[i*NUM_TRACKABLES+j];
        }
        for (k=0;k<NUM_AVOIDANCE;k++) {
            gamma_avoid[i*NUM_AVOIDANCE+k] = ga[i]*1.0;
        }
    }

    // Initialize algorithm grid memory structure
    for (k=0;k<NUM_TRACKABLES;k++){
        for (i=0;i<gridsize;i++) {
            for (j=0;j<gridsize;j++) {
                alldata[gridsize*(gridsize*k+i)+j].Q = 0;
                alldata[gridsize*(gridsize*k+i)+j].S = 0;
                alldata[gridsize*(gridsize*k+i)+j].S_prev = 0;
                alldata[gridsize*(gridsize*k+i)+j].dS = 0;
                alldata[gridsize*(gridsize*k+i)+j].intS = 0;
            }
        }
    }

    // Initialize Algorithm Variables
    for (i=0;i<NUM_TRACKABLES;i++) {
        for (j=0;j<NUM_AVOIDANCE;j++) {
            dvAx[i*NUM_TRACKABLES+j] = 0;
            dvAy[i*NUM_TRACKABLES+j] = 0;
        }
    }
    for (i=0;i<NUM_TRACKABLES;i++) {
        for (j=0;j<NUM_AVOIDANCE;j++) {
            vP[i*NUM_AVOIDANCE+j] = 0.0;
            dvPx[i*NUM_AVOIDANCE+j] = 0.0;
            dvPy[i*NUM_AVOIDANCE+j] = 0.0;
            vA[i*NUM_AVOIDANCE+j] = 0.0;
            dvAx[i*NUM_AVOIDANCE+j] = 0.0;
            dvAy[i*NUM_AVOIDANCE+j] = 0.0;
        }
    }

    // Initialize Linux/Global coverage arrays
    for (i=0;i<gridsize;i++) {
        for (j=0;j<gridsize;j++) {
            Q_linux[gridsize*i+j] = 0;
            Q_global[gridsize*i+j] = 0;
        }
    }

    // Initialize PS3 controller data
    for (i=0;i<65;i++) {
        controllerdata[i] = 127;
    }

    // USED FOR GYRO CAL
    // TODO: defined destinations that moves the robot around and outside the course
    robotdest[0].x = 0;     robotdest[0].y = 0;
    robotdest[1].x = 5;     robotdest[1].y = 0;
    robotdest[2].x = 5;     robotdest[2].y = 5;
    robotdest[3].x = 0;     robotdest[3].y = 5;

}


// Control Algorithm Task
void Control(void) {

    TSK_sleep(100);

    while(1) {

        int i = 0;
        int j = 0;
```

```
        int k = 0;
        int l = 0;
        float max_temp = 0;
        float MRM_temp = 0;
        float Q_temp = 0;
        float p_temp = 0;
        float axD_prev = 0;
        float axD_temp = 0;
        float axDD_prev = 0;
        float axDD_temp = 0;
        float ayD_prev = 0;
        float ayD_temp = 0;
        float ayDD_prev = 0;
        float ayDD_temp = 0;
        float ceD_prev = 0;  // coverage error
        float ceD_temp = 0;
        float ceDD_prev = 0;
        float ceDD_temp = 0;
        float norm_err_const = 0;
        float D_temp = 0;
        float vcalc_temp = 0;
        float Lx = 0;
        float Ly = 0;
        float prox_diff = 0;
        float prox_max = 0;
        float ai_delta_sum = 0;
        float dai_Px = 0;
        float dai_Py = 0;
        float dai_Ax = 0;
        float dai_Ay = 0;
        float dRho = 0;
        float C_temp = 0;
        float size_temp = 0;

        C_temp = (float)C;
        size_temp = (float)size;

        while (updateControlPos == 1) {
            TSK_sleep(1);
        }

        if (firstime) {
            currtime = CLK_getltime();
            firstime = 0;
        }

        for (i=0;i<NUM_TRACKABLES;i++) {
            ax[i] = 0;
            ay[i] = 0;
        }
        ce = 0;

        prevtime = currtime;
        currtime = CLK_getltime();

        dt = (currtime - prevtime)/1000.0;

        Q_average = 0;
        Q_min = 40;

        norm_err_const = (C_temp)*(C_temp)*(C_temp)*(size_temp)*(size_temp);  // i need to typecast this
correctly

        for (k=0;k<NUM_TRACKABLES;k++){
            for (i=0;i<gridsize;i++) {
                for (j=0;j<gridsize;j++) {

                    p_temp = (xpos_control[k]-0.03125*j)*(xpos_control[k]-0.03125*j) + (ypos_control[k]-
0.03125*i)*(ypos_control[k]-0.03125*i);  // room for improvement (?)
                    //p_temp = (xpos[k])*(xpos[k]) + (ypos[k])*(ypos[k]);

                    //max temp = (R cov[k])*(R cov[k])-p temp;
                    max_temp = (R_cov2[k])-p_temp;  // no change
                    max_temp = (max_temp < 0) ? 0.0 : max_temp;

                    // +162us (all data for 3 trackables / 80 gridsize
                    alldata[gridsize*(gridsize*k+i)+j].S_prev = alldata[gridsize*(gridsize*k+i)+j].S; // save
the old values - MOVE THIS LATER

                    // adds 4.2+ ms
                    //MRM_temp = M[k]/((R_cov[k])*(R_cov[k])*(R_cov[k])*(R_cov[k]))*max_temp;
```

```
                    MRM_temp = M_over_R4[k]*max_temp; // drastic improvement
                    alldata[gridsize*(gridsize*k+i)+j].S =
MRM_temp*max_temp;//M_over_R4[k]*powsp(max_temp,16);//MRM_temp*max_temp; // need a max function for floats

                    // adds 300 us only
                    alldata[gridsize*(gridsize*k+i)+j].dS = -2.0*MRM_temp;//-
16.0*M_over_R4[k]*powsp(max_temp,15);//-2.0*MRM_temp; // make faster with one calc

                    // adds 40us only / maybe 0.5ms
                    alldata[gridsize*(gridsize*k+i)+j].intS =
0.5*(alldata[gridsize*(gridsize*k+i)+j].S_prev+alldata[gridsize*(gridsize*k+i)+j].S)*dt;  // Ti in matlab code
.005 = .01s / 2.

                    // adds 300us only
                    Q_temp = alldata[gridsize*(gridsize*k+i)+j].Q+alldata[gridsize*(gridsize*k+i)+j].intS;
                    alldata[gridsize*(gridsize*k+i)+j].Q = (Q_temp < C) ? Q_temp : C;  // Q for agent

                    if (k==0) {
                        Q_average += (alldata[gridsize*(gridsize*0+i)+j].Q);
                        if (alldata[gridsize*(gridsize*0+i)+j].Q < Q_min) {
                            Q_min = alldata[gridsize*(gridsize*0+i)+j].Q;
                        }
                    }

                    // adds 800 us
                    Q_temp = Q_global[gridsize*i+j]+alldata[gridsize*(gridsize*k+i)+j].intS;
                    Q_global[gridsize*i+j] = (Q_temp < C) ? Q_temp : C;  // Global Q - Keep these local maybe
or put in shared memory?
                    if (k == (NUM_TRACKABLES - 1)) {
                        Q_temp = C - Q_global[gridsize*i+j];
                        Q_temp = (Q_temp > 0) ? Q_temp : 0;
                        ceDD_temp = (Q_temp != 0) ? (Q_temp*Q_temp*Q_temp) : 0;
                    }

                    Q_temp = C - alldata[gridsize*(gridsize*k+i)+j].Q;
                    Q_temp = (Q_temp > 0) ? Q_temp : 0;
                    axDD_temp = (Q_temp != 0) ?
6.0*(Q_temp*Q_temp)*(alldata[gridsize*(gridsize*k+i)+j].dS)*(xpos_control[k]-0.03125*j) : 0;
                    ayDD_temp = (Q_temp != 0) ?
6.0*(Q_temp*Q_temp)*(alldata[gridsize*(gridsize*k+i)+j].dS)*(ypos_control[k]-0.03125*i) : 0;

                    if (j > 0) {
                        axD_temp += (axDD_temp+axDD_prev)/2.0;
                        ayD_temp += (ayDD_temp+ayDD_prev)/2.0;
                        if (k == (NUM_TRACKABLES - 1)) {
                            ceD_temp += (ceDD_temp+ceDD_prev)/2.0;
                        }
                    }

                    axDD_prev = axDD_temp;
                    ayDD_prev = ayDD_temp;
                    if (k == (NUM_TRACKABLES - 1)) {
                        ceDD_prev = ceDD_temp;
                    }

                }
                if (i > 0) {
                    ax[k] += (axD_temp+axD_prev)/2.0;
                    ay[k] += (ayD_temp+ayD_prev)/2.0;
                    if (k == (NUM_TRACKABLES - 1)) {
                        ce += (ceD_temp+ceD_prev)/2.0;
                    }
                }
                axD_prev = axD_temp;
                axD_temp = 0;
                axDD_prev = 0;
                ayD_prev = ayD_temp;
                ayD_temp = 0;
                ayDD_prev = 0;
                if (k == (NUM_TRACKABLES - 1)) {
                    ceD_prev = ceD_temp;
                    ceD_temp = 0;
                    ceDD_temp = 0;
                }
            }
            axD_prev = 0;
            ayD_prev = 0;
            if (k == (NUM_TRACKABLES - 1)) {
                ceD_prev = 0;
            }
```

```
        }

        // scale ax and ay for gridsize
        for (i=0;i<NUM_TRACKABLES;i++) {
            ax[i] = ax[i]*dx*dx; // works for square grid only
            ay[i] = ay[i]*dx*dx; // works for square grid only
        }
        ce = (ce*dx*dx)/norm_err_const;

        //dist_count = 1;
        for (i=0;i<NUM_TRACKABLES;i++) {
            for (j=0;j<NUM_AVOIDANCE;j++) {
                if (i != j) {
                    Lx = xpos_control[i]-xpos_control[j];
                    Ly = ypos_control[i]-ypos_control[j];
                    D_temp = sqrtsp((Lx)*(Lx)+(Ly)*(Ly));

                    // if in proximity
                    if ((j < NUM_TRACKABLES) && (i<j)) {
                        if (D_temp <= D_PROX) {
                            // merge Q values
                            for (k=0;k<gridsize;k++) {
                                for (l=0;l<gridsize;l++) {
                                    if (alldata[gridsize*(gridsize*i+k)+l].Q >
alldata[gridsize*(gridsize*j+k)+l].Q) {
                                        alldata[gridsize*(gridsize*j+k)+l].Q =
alldata[gridsize*(gridsize*i+k)+l].Q;
                                    } else {
                                        alldata[gridsize*(gridsize*i+k)+l].Q =
alldata[gridsize*(gridsize*j+k)+l].Q;
                                    }
                                }
                            }
                        }
                    }

                    if (PROX_ON) {
                        if (j < NUM_TRACKABLES) {
                            if (i == 0 && SPECIAL_LEADER3) {
                                vP[i*NUM_AVOIDANCE+j] = 0;
                                dvPx[i*NUM_AVOIDANCE+j] = 0;
                                dvPy[i*NUM_AVOIDANCE+j] = 0;
                            } else {
                                prox_diff = (D_temp)*(D_temp) - (D_PROX)*(D_PROX);
                                prox_max = (prox_diff > 0) ? prox_diff : 0;
                                vP[i*NUM_AVOIDANCE+j] = (prox_max)*(prox_max);
                                dvPx[i*NUM_AVOIDANCE+j] = 4*prox_max*Lx;
                                dvPy[i*NUM_AVOIDANCE+j] = 4*prox_max*Ly;
                            }
                        }
                    }

                    if (AVOID_ON) {

                        if (j < NUM_TRACKABLES) {
                            D_temp = sqrtsp(Lx*Lx+Ly*Ly);
                        } else {
                            D_temp = sqrtsp(Lx*(Lx*P[0]+Ly*P[2])+Ly*(Lx*P[1]+Ly*P[3])); // L'*P*L
                        }

                        vcalc_temp = (D_temp*D_temp - R_col[i]*R_col[i])/(D_temp*D_temp - r_col[i]*r_col[i]);
                        vA[i*NUM_AVOIDANCE+j] = (vcalc_temp < 0) ? vcalc_temp*vcalc_temp : 0;

                        if (i == 0 && j < NUM_TRACKABLES && SPECIAL_LEADER) {  // REMOVED <= 5/18/2012
                            dvAx[i*NUM_AVOIDANCE+j] = 0.0;
                            dvAy[i*NUM_AVOIDANCE+j] = 0.0;
                        }

                        else if (D_temp >= R_col[i]) {
                            dvAx[i*NUM_AVOIDANCE+j] = 0.0;
                            dvAy[i*NUM_AVOIDANCE+j] = 0.0;
                        }

                        else if (D_temp > r_col[i]) {
                            if (j < NUM_TRACKABLES) {
                                dvAx[i*NUM_AVOIDANCE+j] = 4*(R_col[i]*R_col[i] -
r_col[i]*r_col[i])*(D_temp*D_temp-R_col[i]*R_col[i])*
                                    (Lx)/((D_temp*D_temp - r_col[i]*r_col[i])*(D_temp*D_temp -
r_col[i]*r_col[i])*(D_temp*D_temp - r_col[i]*r_col[i]));
                                dvAy[i*NUM_AVOIDANCE+j] = 4*(R_col[i]*R_col[i] -
r_col[i]*r_col[i])*(D_temp*D_temp-R_col[i]*R_col[i])*
```

```
                                        (Ly)/((D_temp*D_temp - r_col[i]*r_col[i])*(D_temp*D_temp -
r_col[i]*r_col[i])*(D_temp*D_temp - r_col[i]*r_col[i]));;
                                } else {
                                    dvAx[i*NUM_AVOIDANCE+j] = 4*(R_col[i]*R_col[i] -
r_col[i]*r_col[i])*(D_temp*D_temp-R_col[i]*R_col[i])*
                                        (Lx*P[0]+Ly*P[2])/((D_temp*D_temp - r_col[i]*r_col[i])*(D_temp*D_temp -
r_col[i]*r_col[i])*(D_temp*D_temp - r_col[i]*r_col[i]));
                                    dvAy[i*NUM_AVOIDANCE+j] = 4*(R_col[i]*R_col[i] -
r_col[i]*r_col[i])*(D_temp*D_temp-R_col[i]*R_col[i])*
                                        (Lx*P[2]+Ly*P[3])/((D_temp*D_temp - r_col[i]*r_col[i])*(D_temp*D_temp -
r_col[i]*r_col[i])*(D_temp*D_temp - r_col[i]*r_col[i]));;

                                }
                            }

                            else if (D_temp < r_col[i]) {

                                if (j < NUM_TRACKABLES) {
                                    dvAx[i*NUM_AVOIDANCE+j] = 4*(R_col[i]*R_col[i] -
(0.2*r_col[i])*(0.2*r_col[i]))*(D_temp*D_temp-R_col[i]*R_col[i])*
                                        (Lx)/((D_temp*D_temp - (0.2*r_col[i])*(0.2*r_col[i]))*(D_temp*D_temp -
(0.2*r_col[i])*(0.2*r_col[i]))*(D_temp*D_temp - (0.2*r_col[i])*(0.2*r_col[i])));
                                    dvAy[i*NUM_AVOIDANCE+j] = 4*(R_col[i]*R_col[i] -
(0.2*r_col[i])*(0.2*r_col[i]))*(D_temp*D_temp-R_col[i]*R_col[i])*
                                        (Ly)/((D_temp*D_temp - (0.2*r_col[i])*(0.2*r_col[i]))*(D_temp*D_temp -
(0.2*r_col[i])*(0.2*r_col[i]))*(D_temp*D_temp - (0.2*r_col[i])*(0.2*r_col[i])));;
                                } else {
                                    dvAx[i*NUM_AVOIDANCE+j] = 4*(R_col[i]*R_col[i] -
(0.2*r_col[i])*(0.2*r_col[i]))*(D_temp*D_temp-R_col[i]*R_col[i])*
                                        (Lx*P[0]+Ly*P[2])/((D_temp*D_temp -
(0.2*r_col[i])*(0.2*r_col[i]))*(D_temp*D_temp - (0.2*r_col[i])*(0.2*r_col[i]))*(D_temp*D_temp -
(0.2*r_col[i])*(0.2*r_col[i])));
                                    dvAy[i*NUM_AVOIDANCE+j] = 4*(R_col[i]*R_col[i] -
(0.2*r_col[i])*(0.2*r_col[i]))*(D_temp*D_temp-R_col[i]*R_col[i])*
                                        (Lx*P[2]+Ly*P[3])/((D_temp*D_temp -
(0.2*r_col[i])*(0.2*r_col[i]))*(D_temp*D_temp - (0.2*r_col[i])*(0.2*r_col[i]))*(D_temp*D_temp -
(0.2*r_col[i])*(0.2*r_col[i])));;

                                }
                            }
                        } // end if AVOID ON
                    } // end if i != j
                } // end j<NUM_AVOIDANCE
                ai_delta_sum = 0.0;
                dai_Px = 0.0;
                dai_Py = 0.0;
                dai_Ax = 0.0;
                dai_Ay = 0.0;
                dRho = 0.0;
                if (i == 0 && SPECIAL_LEADER) { // ALL FOR delta = 2
                    for (k=0;k<NUM_AVOIDANCE;k++) {
                        if ((k != i) && (k<NUM_TRACKABLES)) {
                            ai_delta_sum +=
gamma_prox[i*NUM_TRACKABLES+k]*gamma_prox[i*NUM_TRACKABLES+k]*vP[i*NUM_AVOIDANCE+k]*vP[i*NUM_AVOIDANCE+k]; //
delta = 2
                            dai_Px +=
gamma_prox[i*NUM_TRACKABLES+k]*gamma_prox[i*NUM_TRACKABLES+k]*vP[i*NUM_AVOIDANCE+k]*dvPx[i*NUM_AVOIDANCE+k];
                            dai_Py +=
gamma_prox[i*NUM_TRACKABLES+k]*gamma_prox[i*NUM_TRACKABLES+k]*vP[i*NUM_AVOIDANCE+k]*dvPy[i*NUM_AVOIDANCE+k];
                        }
                        if (k >= NUM_TRACKABLES) { // only avoid obstacles
                            ai_delta_sum +=
gamma_avoid[i*NUM_AVOIDANCE+k]*gamma_avoid[i*NUM_AVOIDANCE+k]*vA[i*NUM_AVOIDANCE+k]*vA[i*NUM_AVOIDANCE+k]; //
delta = 2
                            dai_Ax +=
gamma_avoid[i*NUM_AVOIDANCE+k]*gamma_avoid[i*NUM_AVOIDANCE+k]*vA[i*NUM_AVOIDANCE+k]*dvAx[i*NUM_AVOIDANCE+k];
                            dai_Ay +=
gamma_avoid[i*NUM_AVOIDANCE+k]*gamma_avoid[i*NUM_AVOIDANCE+k]*vA[i*NUM_AVOIDANCE+k]*dvAy[i*NUM_AVOIDANCE+k];
                        }
                    }
                } else {
                    for (k=0;k<NUM_AVOIDANCE;k++) {
                        if ((k != i) && (k<NUM_TRACKABLES)) {
                            ai_delta_sum +=
gamma_prox[i*NUM_TRACKABLES+k]*gamma_prox[i*NUM_TRACKABLES+k]*vP[i*NUM_AVOIDANCE+k]*vP[i*NUM_AVOIDANCE+k]; //
delta = 2
                            dai_Px +=
gamma_prox[i*NUM_TRACKABLES+k]*gamma_prox[i*NUM_TRACKABLES+k]*vP[i*NUM_AVOIDANCE+k]*dvPx[i*NUM_AVOIDANCE+k];
                            dai_Py +=
gamma_prox[i*NUM_TRACKABLES+k]*gamma_prox[i*NUM_TRACKABLES+k]*vP[i*NUM_AVOIDANCE+k]*dvPy[i*NUM_AVOIDANCE+k];
                        }
```

```
                    if (k != i) { // avoids everything but itself
                        ai_delta_sum +=
gamma_avoid[i*NUM_AVOIDANCE+k]*gamma_avoid[i*NUM_AVOIDANCE+k]*vA[i*NUM_AVOIDANCE+k]*vA[i*NUM_AVOIDANCE+k]; //
delta = 2
                        dai_Ax +=
gamma_avoid[i*NUM_AVOIDANCE+k]*gamma_avoid[i*NUM_AVOIDANCE+k]*vA[i*NUM_AVOIDANCE+k]*dvAx[i*NUM_AVOIDANCE+k];
                        dai_Ay +=
gamma_avoid[i*NUM_AVOIDANCE+k]*gamma_avoid[i*NUM_AVOIDANCE+k]*vA[i*NUM_AVOIDANCE+k]*dvAy[i*NUM_AVOIDANCE+k];
                    }
                }
            }
            dRho = (ai_delta_sum == 0) ? 0 : 1.0/sqrtsp(ai_delta_sum);  // delta = 2 : ai_delta_sum^(1/delta-1)
= ai_delta_sum^(-1/2)
            dvi_dx[i] = dRho*(dai_Px+dai_Ax);
            dvi_dy[i] = dRho*(dai_Py+dai_Ay);

            theta_des_ap[i] = PI/2.0 - atan2f(dvi_dy[i],dvi_dx[i]);
            theta_des_cov[i] = PI/2.0 - atan2f(ay[i],ax[i]);
            if (trackableID == 0 && SPECIAL_LEADER2) {
                theta_des[i] = theta_des_ap[i];
            } else {
                theta_des[i] = theta_des_ap[i] + theta_des_cov[i]; // remove coverage influence for driver mode
            }


            if ((i == trackableID) && (updateControlOut == 0)) {
                u_control =
AP_gain[i]*(dvi_dx[i]*cosf(theta_control[i])+dvi_dy[i]*sinf(theta_control[i]))+cov_gain[i]*(ax[i]*cosf(theta_co
ntrol[i])+ay[i]*sinf(theta_control[i]));

                theta_control_temp = fmodf(theta_control[i],(float)(2*PI));
                theta_des_temp = fmodf(theta_des[i],(float)(2*PI));
                if (theta_control_temp < -PI) theta_control_temp+=2*PI;
                if (theta_control_temp > PI) theta_control_temp-=2*PI;
                if (theta_des_temp < -PI) theta_des_temp+=2*PI;
                if (theta_des_temp > PI) theta_des_temp-=2*PI;

                angle_diff1 = theta_des_temp - theta_control_temp;
                if (angle_diff1 > PI) angle_diff1 -= (2*PI);
                if (angle_diff1 < -PI) angle_diff1 += (2*PI);

                utheta_control = theta_gain[i]*angle_diff1;

                if ((trackableID == 0) && (SPECIAL_LEADER4)) {
                    Q_average = Q_average/25600.0;
                    Q_average_print = Q_average;

                    findLeastCovered();
                    gotox_control = gotox;
                    gotoy_control = gotoy;
                    num_pels_print = send_numpels;
                }
                ce_control = ce;
                updateControlOut = 1;
            }

        } // end i<NUM_TRACKABLES
        updateControlPos = 1;
        TSK_sleep(1);
    }
}


long timecount= 0;
int whichled = 0;



// This SWI is Posted after each set of new data from the F28335
void RobotControl(void) {

    int n = 0;
    int i = 0;

    if (0==(timecount%1000)) {
        switch(whichled) {
        case 0:
            SETREDLED;
            CLRBLUELED;
            CLRGREENLED;
            whichled = 1;
```

```
            break;
        case 1:
            CLRREDLED;
            SETBLUELED;
            CLRGREENLED;
            whichled = 2;
            break;
        case 2:
            CLRREDLED;
            CLRBLUELED;
            SETGREENLED;
            whichled = 0;
            break;
        default:
            whichled = 0;
            break;
        }
    }

    if (GET_DATA_TO_LINUX) {
        for (i=0;i<65;i++) {
            controllerdata[i] = sharedQTmem[i];
        }

        BCACHE_wb ((void *)sharedQTmem,65,EDMA3_CACHE_WAIT);

        CLR_DATA_TO_LINUX;
    }

    BCACHE_inv((void *)ptrshrdmem,sizeof(sharedmemstruct),EDMA3_CACHE_WAIT);
    if (GET_DATA_FROM_LINUX) {

        if (new_optitrack == 0) {
            for (i=0;i<OPTITRACKDATASIZE;i++) {
                Optitrackdata[i] = ptrshrdmem->Optitrackdata[i];
                temp_trackableID = ptrshrdmem->RobotID;
            }
            optitrack_rectime = CLK_getltime();
            new_optitrack = 1;
        }
        if (GET_CONTINUOUSDATA_TO_LINUX) {
            for (i=0;i<NUM_TRACKABLES;i++) {
                if (i != trackableID) {
                    xpos_kal[i] = ptrshrdmem2[i].tx;
                    ypos_kal[i] = ptrshrdmem2[i].ty;
                }
            }
            ptrshrdmem2[trackableID].tx = xpos_kal[trackableID];
            ptrshrdmem2[trackableID].ty = ypos_kal[trackableID];

            BCACHE_wb ((void *)ptrshrdmem2,(3*NUM_TRACKABLES)*4,EDMA3_CACHE_WAIT);

            CLR_CONTINUOUSDATA_TO_LINUX;
        }
        CLR_DATA_FROM_LINUX;
    }

    if (new_optitrack == 1) {
        // Check for frame errors / packet loss
        if (previous_frame == Optitrackdata[OPTITRACKDATASIZE-2]) {
            frame_error++;
        }
        previous_frame = Optitrackdata[OPTITRACKDATASIZE-2];

        // Set local trackableID if first receive data
        if (firstdata){
            CLRLED2;
            CLRLED3;
            CLRLED4;
            CLRLED5;
            trackableID = temp_trackableID;
            firstdata = 0;
            switch(trackableID) {
            case 0:
                SETLED2;
                break;
            case 1:
                SETLED3;
                break;
            case 2:
                SETLED4;
```

```
                break;
            default:
                SETLED2;
                SETLED3;
                SETLED4;
                SETLED5;
                break;
            }
        }

        // Check if local trackableID has changed - should never happen
        if (trackableID != temp_trackableID) {
            trackableIDerror++;
        }

        // Save position and yaw data
        for (i=0;i<NUM_TRACKABLES;i++){
            // subtracting 16 so everything is shifted such that optitrack's origin is the center of the arena
(while keeping all coordinates positive)
            if (isnan(Optitrackdata[i*3+0]) != 1) {
                if ((Optitrackdata[i*3+0] != 0.0) && (Optitrackdata[i*3+1] != 0.0) && (Optitrackdata[i*3+2] !=
0.0)) {
//                  xpos[i] = Optitrackdata[i*3+0]*32.0/3.6576+16;  // scaled so 32x32 unit course fits 12x12
feet tiles (OpitrackdataInMeters*32units/3.66metersIn12Feet)
//                  ypos[i] = (Optitrackdata[i*3+1]*32.0/3.6576)*-1.0+16;  // need to flip the y direction
(ground tool sets it opposite of what I want)
//                  theta[i] = Optitrackdata[i*3+2]*2*PI/360.0;  // in radians - should this not accumulate?
                    xpos_ot[i] = Optitrackdata[i*3+0]+2.5; // was 2.5 for size = 5
                    ypos_ot[i] = Optitrackdata[i*3+1]*-1.0+2.5;

                    if (i != trackableID) {
                        theta_ot[i] = Optitrackdata[i*3+2]*2*PI/360.0;
                    } else {
                        temp_theta = fmodf(theta_kal[trackableID],(float)(2*PI));//(theta[trackableID]%(2*PI));
                        if (temp_theta > 0) {
                            if (temp_theta < PI) {
                                if (Optitrackdata[i*3+2] >= 0.0) {
                                    // THETA > 0, kal in QI/II, OT in QI/II
                                    theta_ot[i] = ((int)((theta_kal[trackableID])/(2*PI)))*2.0*PI +
Optitrackdata[i*3+2]*2*PI/360.0;
                                } else {
                                    if (temp_theta > (PI/2)) {
                                        // THETA > 0, kal in QII, OT in QIII
                                        theta_ot[i] = ((int)((theta_kal[trackableID])/(2*PI)))*2.0*PI + PI +
(PI + Optitrackdata[i*3+2]*2*PI/360.0);
                                    } else {
                                        // THETA > 0, kal in QI, OT in QIV
                                        theta_ot[i] = ((int)((theta_kal[trackableID])/(2*PI)))*2.0*PI +
Optitrackdata[i*3+2]*2*PI/360.0;
                                    }
                                }
                            } else {
                                if (Optitrackdata[i*3+2] <= 0.0) {
                                    // THETA > 0, kal in QIII, OT in QIII
                                    theta_ot[i] = ((int)((theta_kal[trackableID])/(2*PI)))*2.0*PI + PI + (PI +
Optitrackdata[i*3+2]*2*PI/360.0);
                                } else {
                                    if (temp_theta > (3*PI/2)) {
                                        // THETA > 0, kal in QIV, OT in QI
                                        theta_ot[i] = ((int)((theta_kal[trackableID])/(2*PI)))*2.0*PI + 2*PI +
Optitrackdata[i*3+2]*2*PI/360.0;
                                    } else {
                                        // THETA > 0, kal in QIII, OT in QII
                                        theta_ot[i] =
(floorf((theta_kal[trackableID])/((float)(2.0*PI))))*2.0*PI + Optitrackdata[i*3+2]*2*PI/360.0;
                                    }
                                }
                            }
                        } else {
                            if (temp_theta > -PI) {
                                if (Optitrackdata[i*3+2] <= 0.0) {
                                    // THETA < 0, kal in QIII/IV, OT in QIII/IV
                                    theta_ot[i] = ((int)((theta_kal[trackableID])/(2*PI)))*2.0*PI +
Optitrackdata[i*3+2]*2*PI/360.0;
                                } else {
                                    if (temp_theta < (-PI/2)) {
                                        // THETA < 0, kal in QIII, OT in QII
                                        theta_ot[i] = ((int)((theta_kal[trackableID])/(2*PI)))*2.0*PI - PI + (-
PI + Optitrackdata[i*3+2]*2*PI/360.0);
                                    } else {
                                        // THETA < 0, kal in QIV, OT in QI
```

```
                                    theta_ot[i] = ((int)((theta_kal[trackableID])/(2*PI)))*2.0*PI +
Optitrackdata[i*3+2]*2*PI/360.0;
                                }
                            }
                        } else {
                            if (Optitrackdata[i*3+2] >= 0.0) {
                                // THETA < 0, kal in QI/II, OT in QI/II
                                theta_ot[i] = ((int)((theta_kal[trackableID])/(2*PI)))*2.0*PI - PI + (-PI +
Optitrackdata[i*3+2]*2*PI/360.0);
                            } else {
                                if (temp_theta < (-3*PI/2)) {
                                    // THETA < 0, kal in QI, OT in QIV
                                    theta_ot[i] = ((int)((theta_kal[trackableID])/(2*PI)))*2.0*PI - 2*PI +
Optitrackdata[i*3+2]*2*PI/360.0;
                                } else {
                                    // THETA < 0, kal in QII, OT in QIII
                                    theta_ot[i] = ((int)((theta_kal[trackableID])/(2*PI)))*2.0*PI +
Optitrackdata[i*3+2]*2*PI/360.0;
                                }
                            }
                        }
                    }
                    if (fabs(theta_ot[i]-theta_kal[trackableID])<(0.25)){ // 0.25rad is about 15 degrees
                        new_optitrack_kal = 1;
                    } else {
                        otkalcount++;
                    }
                }
            }
        }
    }
    new_optitrack = 0;
}

// using 400deg/s gyro
gyro = adcA3*3.0/4096.0;
if (settlegyro < SETTLETIME) {
    settlegyro++;
    if (settlegyro < (SETTLETIME/3)) {
        // do nothing
    } else if (settlegyro < (2*SETTLETIME/3)) {
        gyro_zero = gyro_zero + gyro/(SETTLETIME/3);
    } else {
        gyro_drift += (((gyro-gyro_zero) + old_gyro)*.0005)/(SETTLETIME/3);
        old_gyro = gyro-gyro_zero;
    }
    if(settlegyro%500 == 0) {
        LCDPrintfLine(1,"Cal Gyro -- %.1fSecs", (float)(SETTLETIME - settlegyro)/1000.0 );
        LCDPrintfLine(2,"");
    }

    // while calibrating set positions to optitrack data
    x_pred[0][0] = xpos_ot[trackableID]; //estimate in structure form (useful elsewhere)
    x_pred[1][0] = ypos_ot[trackableID];
    x_pred[2][0] = theta_ot[trackableID];

    for (i=0;i<NUM_TRACKABLES;i++) {
        xpos_kal[i] = xpos_ot[i];
        ypos_kal[i] = ypos_ot[i];
        theta_kal[i] = theta_ot[i];
    }

    new_optitrack_kal = 0;

    SetRobotOutputs(0,0,0,0,0,0,0,0,0,0);
} else {

    gyro_angle = gyro_angle - ((gyro-gyro_zero) + old_gyro)*.0005 + gyro_drift;   // doing to have to do
this based on time
    old_gyro = gyro-gyro_zero;
    gyro_radians = (gyro-gyro_zero)*(PI/180.0)*400.0*gyro4x_gain;

    // Kalman filtering
    vel1 = (enc1 - enc1old)/(193.0*0.001)*0.3048;   // calculate actual velocities IN METERS/s
    vel2 = (enc2 - enc2old)/(193.0*0.001)*0.3048;
    if (fabsf(vel1) > 10.0) vel1 = vel1old; // check for encoder roll-over should never happen
    if (fabsf(vel2) > 10.0) vel2 = vel2old;
    enc1old = enc1; // save past values
    enc2old = enc2;
    vel1old = vel1;
    vel2old = vel2;
```

```
        // Step 0: update B, u
        B[0][0] = cosf(theta_kal[trackableID])*0.001;
        B[1][0] = sinf(theta_kal[trackableID])*0.001;
        B[2][1] = 0.001;
        u[0][0] = 0.5*(vel1 + vel2);    // linear velocity of robot
        u[1][0] = (gyro-gyro_zero)*(PI/180.0)*400.0*gyro4x_gain;    // angular velocity in rad/s (negative for
right hand angle)

        // Step 1: predict the state and estimate covariance
        Matrix3x2_Mult(B, u, Bu);                   // Bu = B*u
        Matrix3x1_Add(x_pred, Bu, x_pred, 1.0, 1.0); // x_pred = x_pred(old) + Bu
        Matrix3x3_Add(P_pred, Q, P_pred, 1.0, 1.0); // P_pred = P_pred(old) + Q

        // Step 2: if there is a new measurement, then update the state
        if (1 == new_optitrack_kal) {
            z[0][0] = xpos_ot[trackableID]; // take in the optitrack measurement
            z[1][0] = ypos_ot[trackableID];
            z[2][0] = theta_ot[trackableID];

            new_optitrack_kal = 0;

            // Step 2a: calculate the innovation/measurement residual, ytilde
            Matrix3x1_Add(z, x_pred, ytilde, 1.0, -1.0);    // ytilde = z-x_pred
            // Step 2b: calculate innovation covariance, S
            Matrix3x3_Add(P_pred, R, S, 1.0, 1.0);                          // S = P_pred + R
            // Step 2c: calculate the optimal Kalman gain, K
            Matrix3x3_Invert(S, S_inv);
            Matrix3x3_Mult(P_pred,  S_inv, K);                              // K = P_pred*(S^-1)
            // Step 2d: update the state estimate x_pred = x_pred(old) + K*ytilde
            Matrix3x1_Mult(K, ytilde, temp_3x1);
            Matrix3x1_Add(x_pred, temp_3x1, x_pred, 1.0, 1.0);
            // Step 2e: update the covariance estimate   P_pred = (I-K)*P_pred(old)
            Matrix3x3_Add(eye3, K, temp_3x3, 1.0, -1.0);
            Matrix3x3_Mult(temp_3x3, P_pred, P_pred);
        }   // end of correction step

        // set ROBOTps to the updated and corrected Kalman values.
        xpos_kal[trackableID] = x_pred[0][0];
        ypos_kal[trackableID] = x_pred[1][0];
        theta_kal[trackableID] = x_pred[2][0];

        if (updateControlPos){
            for (n=0;n<NUM_TRACKABLES;n++){
                    xpos_control[n] = xpos_kal[n];
                    ypos_control[n] = ypos_kal[n];
                    theta_control[n] = theta_kal[n];
            }

            if (GET_IMAGE_TO_LINUX) {


                for (n=0;n<gridsize;n++) {
                    for (i=0;i<gridsize;i++) {
                        Q_linux[gridsize*n+i] = (int)(Q_global[gridsize*n+i]*1000);
                    }
                }

                for (n=0;n<NUM_TRACKABLES;n++) {
                        Q_linux[gridsize*gridsize+n*2] = (int)(xpos_control[n]*1000);
                        Q_linux[gridsize*gridsize+n*2+1] = (int)(ypos_control[n]*1000);
                }

                Q_linux[gridsize*gridsize+NUM_TRACKABLES*2] = (int)(ce_control*10000.0);
                Q_linux[gridsize*gridsize+NUM_TRACKABLES*2+1] = (int)(sqrtsp((xpos_control[0]-
xpos_control[2])*(xpos_control[0]-xpos_control[2])+(ypos_control[0]-ypos_control[2])*(ypos_control[0]-
ypos_control[2]))*1000.0);
                Q_linux[gridsize*gridsize+NUM_TRACKABLES*2+2] = (int)(sqrtsp((xpos_control[1]-
xpos_control[2])*(xpos_control[1]-xpos_control[2])+(ypos_control[1]-ypos_control[2])*(ypos_control[1]-
ypos_control[2]))*1000.0);
                Q_linux[gridsize*gridsize+NUM_TRACKABLES*2+3] = (int)(sqrtsp((xpos_control[0]-
xpos_control[3])*(xpos_control[0]-xpos_control[3])+(ypos_control[0]-ypos_control[3])*(ypos_control[0]-
ypos_control[3]))*1000.0);
                Q_linux[gridsize*gridsize+NUM_TRACKABLES*2+4] = (int)(sqrtsp((xpos_control[0]-
xpos_control[4])*(xpos_control[0]-xpos_control[4])+(ypos_control[0]-ypos_control[4])*(ypos_control[0]-
ypos_control[4]))*1000.0);
                Q_linux[gridsize*gridsize+NUM_TRACKABLES*2+5] = (int)(sqrtsp((xpos_control[1]-
xpos_control[3])*(xpos_control[1]-xpos_control[3])+(ypos_control[1]-ypos_control[3])*(ypos_control[1]-
ypos_control[3]))*1000.0);
```

```
                Q_linux[gridsize*gridsize+NUM_TRACKABLES*2+6] = (int)(sqrtsp((xpos_control[1]-
xpos_control[4])*(xpos_control[1]-xpos_control[4])+(ypos_control[1]-ypos_control[4])*(ypos_control[1]-
ypos_control[4]))*1000.0);
                Q_linux[gridsize*gridsize+NUM_TRACKABLES*2+7] = (int)(sqrtsp((xpos_control[2]-
xpos_control[3])*(xpos_control[2]-xpos_control[3])+(ypos_control[2]-ypos_control[3])*(ypos_control[2]-
ypos_control[3]))*1000.0);
                Q_linux[gridsize*gridsize+NUM_TRACKABLES*2+8] = (int)(sqrtsp((xpos_control[2]-
xpos_control[4])*(xpos_control[2]-xpos_control[4])+(ypos_control[2]-ypos_control[4])*(ypos_control[2]-
ypos_control[4]))*1000.0);

                BCACHE_wb ((void *)Q_linux,(gridsize*gridsize+2*NUM_TRACKABLES+9)*4,EDMA3_CACHE_WAIT);

                CLR_IMAGE_TO_LINUX;
            }

            updateControlPos = 0;

        }

        if (updateControlOut) {
            if ((trackableID == 0) && SPECIAL_LEADER4) {

                xy_control(&vref, &turn, 1.0, xpos_control[trackableID], ypos_control[trackableID],
gotox_control*dx, gotoy_control*dy, theta_control[trackableID], 0.0762, 0.1524);

                u_out = vref;
                utheta_out = turn;


            } else {
                u_out = u_control;
                utheta_out = utheta_control;
            }
            updateControlOut = 0;
        }

        swi_time = CLK_getltime();
        // Halt robots if Optitrack stops streaming
        if ((swi_time - optitrack_rectime) > 500) {
            u_out = 0;
            utheta_out = 0;
        }

        if (errorcheck == 0) {
            u_out = 0;
            utheta_out = 0;
        }

        if ((swi_time - swi_time_prev) > 150) {
            if (trackableID > -1) {

                if ((trackableID == 0) && SPECIAL_LEADER4) {

LCDPrintfLine(1,"%.1f,%.1f,%.1f,%d",gotox_control*dx,gotoy_control*dy,Q_average_print,num_pels_print);
                } else {

LCDPrintfLine(1,"t:%.1f,x:%.1f,y%.1f",theta_control[trackableID],xpos_control[trackableID],ypos_control[trackab
leID]);
                }

                LCDPrintfLine(2,"u:%.1f,ut:%.1f,dt:%.3f",u_out,utheta_out,dt*1000.0);

            } else {

                    LCDPrintfLine(1,"%u,%u,%u",controllerdata[0],controllerdata[1],controllerdata[2]);
                    LCDPrintfLine(2,"u:%.1f,ut:%.1f,dt:%.3f",u_out,utheta_out,dt*1000.0);
            }
            swi_time_prev = swi_time;
        }

        // Saturation
        if (u_out > 4) u_out = 4;
        if (u_out < -4) u_out = -4;
        if (utheta_out > 10) utheta_out = 10;
        if (utheta_out < -10) utheta_out = -10;

        SetRobotOutputs(u_out,utheta_out,0,0,0,0,0,0,0,0);

        timecount++;
    }
}
```