UNIVERSITY OF OTTAWA

OTTAWA-CARLETON INSTITUTE FOR MECHANICAL AND AEROSPACE ENGINEERING

Design and Development of a Hardware Test Platform for Multi-Agent Control Algorithms Testing and Validation

Author:
Sarmad Tanveer

Supervisor:
Davide Spinello

A thesis submitted in partial fulfillment of the requirements for the Master of Applied Science degree
in
Mechanical Engineering

uOttawa

Design and Development of a Hardware Test Platform for Multi-Agent Control Algorithms Testing and Validation

Sarmad Tanveer

Abstract

Multi-robot networks are used in a variety of military and civilian applications such as harbour protection, perimeter surveillance, search & rescue missions and cooperative estimation, among others. In order to develop functional multi-robot networks to achieve these tasks, a combination of theoretical and experimental work is required. Theoretical research aims to model the open and closed loop dynamics of multi-robot systems and to develop corresponding control algorithms for tackling the previously mentioned tasks. Experimental work focuses on the hardware or simulated implementations to test and evaluate the algorithms' performance, and eventually inform refinements of theoretical algorithms to adapt to hardware imposed intrinsic constraints.

As theoretical and algorithmic research in the field of multi-robot networks matures, a need for experimental validation of a variety of sophisticated algorithmic tools becomes apparent, with the two aspects co-developing to inform theoretical refinements that account for hardware induced constraints, and possible technological advances suggested by theoretical predictions. This thesis contributes a design for a hardware test platform for evaluating and implementing algorithms in the field of multi-robot networks. The test platform design implements an off the market robot solution for the robotic agents, discussing various additional embedded frameworks that allow for capabilities such as indoor agent localization, inter-agent wireless communication and agent locomotion, with the goal of understanding if a combination of existing market and academic technologies can be used to develop a cost effective hardware multi-agent test platform.

The proposed hardware design is then validated on previously developed multi-agent area coverage control and optimization algorithms. More specifically, the hardware test platform is used to implement various optimal coverage problems with time invariant risk density distributions. Both uniform and nonuniform risk density distribution scenarios are considered. These experimental results are com-

pared with simulations to assess if the proposed hardware test platform design can plausibly reproduce behaviours that are consistent with theoretical predictions of area coverage control algorithms. Future work will include the extension of the testing capabilities of this test platform to a larger class of multi-agent control algorithms.

Acknowledgments

I would like to express my sincere gratitude to Dr. Davide Spinello for his supervision, guidance, support and advice throughout my graduate studies and this thesis.

I would also like to thank my family for supporting me through my studies and for encouraging my desire to learn.

Finally I would like to thank my fiancé, Fatima, for her consistent support for my goals and appreciation for my achievements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Wireless sensor networks are comprised of interconnected mobile sensors that are capable of decentralized wireless communication. Technological and computational improvements allow for advancements in the field of wireless sensor networks, expanding the applications and democratizing the hardware systems that start to be widely available at low cost with user friendly interfaces [1]. This allows for applications involving distributed environmental sensing of various quantities such as temperature, pressure, sound or motion [2]. Wireless sensor networks in the form of multi-robot networks have additional applications in a variety of military and civilian tasks such as harbour protection [3, 4], perimeter surveillance [5, 6], search & rescue missions [7] and cooperative estimation [8].

In order to develop functional multi-robot networks to achieve these tasks, a combination of theoretical and experimental work is required. Theoretical research aims to model the open and closed loop dynamics of multi-robot networks and to develop corresponding control algorithms for tackling the previously mentioned tasks, among others. Experimental work focuses on the hardware or simulated implementations to test and evaluate the algorithms' performance, and to inform algorithmic refinements that incorporate hardware induced constraints. Hardware implementation is also crucial to understand limitations and to devise new applications, eventually inducing theoretical investigations and continuing the co-development loop.

This thesis focuses primarily on the design and hardware implementation of a multi-robot test platform to be used to test and evaluate different cooperative control and estimation algorithms in the multi-robot network domain.

1.1 Motivation

As research in the field of multi-robot networks matures and a variety of theoretical work is being developed, a need for experimental validation of this theoretical work becomes apparent. As such, this thesis aims to present the design and development of a physical test platform for evaluating multi-agent system motion control and estimation algorithms, with the goal of understanding if a combination of existing market and academic technologies can be used to develop a cost effective hardware multi-agent test platform.

As a resulting benefit, the successful development of a cost effective testing platform will enable my research group to have a more complete approach for research in the field of multi-agent systems, complementing the past and present theoretical modelling and algorithm development [9–11], which do not always translate smoothly to practical implementation and realistic performance. This can be due to a variety of factors such as lack of availability of required hardware, latency caused by hardware components, cost of parts or the presence of complex parameters that are difficult to model theoretically.

For this reason, a reliable, robust, and multi-task test platform can help ensure that theoretical predictions and algorithmic tools can accurately and efficiently be translated into practical implementation, and refined to account for constraints imposed by the practical implementation.

1.2 Thesis Objectives and Contributions

This work contributes a design and hardware realization for a general purpose multi-agent algorithm testing platform constituted of terrestrial low cost robots. This is the first iteration of the multi-agent algorithm testing platform and as such, for the sake of simplicity and feasibility, the design is limited to terrestrial low cost robots. Future work will aim to extend the features of the test platform, transitioning toward a more agnostic design. The proposed design addresses the following requirements:

- Reliable inter-robot wireless communication capabilities
- Reliable localization of robotic agents within the work space
- Terrestrial locomotion agents in the form of mechatronic differential drive robots

- Locomotion of each robotic agent within the environment

The proposed design was constructed through the use of a variety of technologies. These technologies will be introduced and discussed further in Chapter 3. The proposed design for the hardware test platform is intended to be sufficiently flexible to be used for a variety of multi-agent control and estimation algorithms. However, for the scope of this thesis, the hardware test platform's operation is illustrated and tested on well established algorithms in the domain of multi-agent area coverage control [9–13]. These experimental results are compared with simulations to assess if the proposed hardware test platform design can plausibly reproduce behaviour that is consistent with theoretical predictions of area coverage control algorithms. With the eventual goal being to generalize the testing capabilities of this test platform to a larger domain of multi-agent control algorithms.

1.3 Thesis Outline

This thesis is structured as follows. Chapter 2 provides a literature review on fundamental concepts in the realm of multi-robot networks, multi-agent coverage control and optimization algorithms, wireless communication protocols, indoor localization frameworks, machine learning and mechatronic devices available on the market.

Chapter 3 describes the proposed design of the hardware test platform in detail. This chapter introduces all the different components of the hardware test platform and explains how these pieces come together to make a functional system. Hardware and software details are provided. This chapter also discusses the implementation of area coverage control and optimization algorithms using the proposed design for the hardware test platform. The principles behind a simulation of area coverage control and optimization algorithms are also discussed.

Chapter 4 presents the experimental results of area coverage control and optimization algorithms using the hardware test platform. These results are compared to the results obtained from a simulation of area coverage control and optimization algorithms to evaluate if the hardware test platform can plausibly reproduce behaviour that is consistent with theoretical predictions of area coverage control algorithms. A discussion of the results is presented and future steps are discussed.

Chapter 5 summarizes the objectives, results and conclusions of the thesis. Finally, all the code used for the software portion of this thesis, the raw data results,

along with any other supplemental material can be found in the Appendix.

Chapter 2

Literature Review and Background Theory

This chapter conducts a literature review and provides relevant background theory relating to multi-robot networks. Additionally, since the aim of this thesis is to design and build a hardware testing platform for multi-agent systems motion control algorithms, the literature review also includes the discussion of the technologies used for enabling localization and wireless communication of robotic agents. Finally, since the operation of the hardware test platform is evaluated and illustrated specifically by implementing multi-agent coverage control and optimization algorithms, some relevant background theory in this domain is provided.

The literature review is structured as follows. Section 2.1 discusses multi-robot networks and introduces some literature on the various types and architectures of multi-robot networks. Section 2.2 introduces the problem of area coverage control and optimization and discusses some useful concepts that are relevant for tackling this problem. These concepts include Voronoi diagrams, the Lloyd algorithm and the notion of environments with time invariant risk density fields. Section 2.3 brings focus back to the objective of this thesis, which is the design and development of a testing platform for multi-agent systems motion control algorithms. This section discusses the required hardware capabilities of the test platform. These requirements include effective real time inter-agent communication and precise localization of agents.

2.1 Multi-Robot Networks

Multi-robot networks consist of a group of robots that work together towards the completion of a task. For certain applications the process of working together can be intentional and cooperative. For other applications, the process of working together requires no cooperation and is simply the result of the robots in the networks following local control laws and relying on spatial interactions with neighbouring robots, that are spatially defined, to exhibit a more complex sense of group behaviour [14].

The robotic agents used in these networks can vary drastically in terms of complexity and capability based on application related specifics. A combination of a variety of technologies can be used to develop the optimal robotic agents for a given task. Some of these technology domains include environmental sensing, actuation, perception, computer vision, machine learning, data processing, localization, mobility and communication [14].

Recent advancements in these domains have resulted in increased opportunity for the implementation of multi-robot networks. Some general domains of application for multi-robot networks include surveillance tasks [15–17], search and rescue missions [18, 19], foraging tasks [20–22], formation control [23–25], exploration tasks [26–28] and cooperative manipulation tasks [29–31]. Robots in of themselves are quite capable and in certain cases a single robot can perform desired tasks quite effectively. However, there are some tasks that require the use of multi-robot networks as they have the following benefits over single robot systems [14]:

- Distributed allocation of tasks allows multi-robot networks to complete tasks more quickly.
- Multi-robot networks can consist of a variety of heterogeneous robotic agents, each with its own specialization. This means that multi-robot networks can work together to tackle complex tasks effectively.
- Compared to single robot systems, multi-robot networks can cover larger work spaces, allowing for completion of tasks that are not tightly localized.
- Multi-robot networks are more versatile and adaptable, as they can continue functioning even if some members of the network fail or stop functioning altogether in distributed implementations.

- Inter-agent communications can allow for distributed information sharing and decision making architectures, which can be used to introduce robustness and resilience of the system, by removing the dependence on a crucial agent or unit to function.

2.1.1 Types of Multi-Robot Networks

There are mainly two types of multi-robot networks and the distinction between these two types of networks relates to the level of communication and the intention of cooperation between robots in each network type. These two network types are [14]:

2.1.1.1 Collective Swarm Systems

These types of multi-robot networks typically consist of a large number of robotic agents. Another characteristic of these types of networks is that there is usually minimal communication required between different agents. Instead, the robots in these networks follow local control laws and rely on spatial interactions with neighbouring robots that are spatially defined to exhibit a more complex sense of group cooperation.

2.1.1.2 Intentionally Cooperative Systems

On the other hand, within this second type of multi-robot network, the robotic agents typically have strong communication capabilities. These agents work together to achieve tasks and typically take into consideration the state and actions of other agents in the network to determine their own behaviour.

Intentionally cooperative systems can be further split into two categories, based on the degree to which the agents in these systems cooperate with their colleagues. The two distinctions are [14]:

- **Strongly Cooperative:** In these types of systems, agents rely on sophisticated and accurate communication with the other agents in the network. They require up to date information regarding the state of the system and they work closely with other agents to achieve a specific task.
- **Weakly Cooperative:** These systems exhibit a more relaxed sense of cooperation between agents in the network. Typically communication in these

networks is not required to be as accurate or up to date. Agents in these networks typically perform individual tasks independently, and communicate with other agents in the network occasionally for updates on state of the system or simply for updates on the distribution of workload within the network.

2.1.2 Multi-Robot Network Communication Architectures

A variety of multi-robot network communication architectures can be implemented. Communication structures underlying multi-agent systems are suitably described by graph theoretical tools, in which agents are represented by nodes and communication links are represented by edges. Different communication network topologies have significant influence on the operation of a multi-robot network, and therefore they must be selected carefully based on the requirements or limitations of the application. The four main types of architectures are [14]:

2.1.2.1 Centralized

Centralized multi-robot networks have a centralized unit that processes the information about the state of the system, fuses it, and broadcasts it for the purpose of controlling the actions of all agents in the network. The benefit of this type of architecture is that it can be quite simple to setup and it ensures synchronization among all agents in the network. Furthermore, a centralized entity can be designed to handle more complex data processing than what could be handled by individual agents due to hardware limitations. Scenarios in which communication capabilities are limited would also benefit from the adoption of a centralized architecture. This architecture however lacks scalability and reliability. Scalability is difficult because as the size of the network grows, the centralized entity may become less capable of handling the computational workload. Reliability is also an issue due to the fact that if the centralized entity fails, then the whole network fails as well. In certain applications, this single point vulnerability can be insufficient.

2.1.2.2 Hierarchical

Hierarchical architectures exhibit a layered structure in which control is passed down through each layer, similar to how a corporate environment is set up. The structure would begin with a particular agent that controls a group of other agents

in the network. These agents then each control their own group of agents. This structure has the effect of distributing control and responsibility. Compared to the centralized architecture, this architecture type is more scalable as computational load can be distributed down the layers. However, this architecture faces the same issue of reliability. Any failure in the top levels of control will trickle down to the remaining layers. This means the function of the architecture is particularly vulnerable to any errors at the top layers.

2.1.2.3 Decentralized

This architecture has no central control entity or entities. Any action that an agent takes is decided on fully by that particular agent, based on all the information available to the agent at the time. This method is both scalable and reliable, as the failure of any one unit has almost no influence on other agents in the network. Computing demands are also distributed across the entire network, so an increase in size of the network is easy to handle. This architecture does however have its downsides as well. A decentralized architecture means that the individual agents must also compute any necessary parameters for the task at hand. This is not always feasible depending on the computational capability of each individual agent. Furthermore, it can be difficult to have agents work in a synchronous and timely manner with this architecture. Decentralized architectures can make use of reliable inter-agent communication protocols for distribution of information across the network as well, when required.

2.1.2.4 Hybrid

The fourth type of architecture is essentially some form of a hybrid of the other three architectures. This architecture allows for agents to make local control decisions, and also has some form of hierarchical control, where control actions are fed downwards through layers of the network. This has the benefit of combining the scalability and reliability of decentralized architectures, with the computation capability and precise cooperation provided by centralized architectures.

2.2 Multi-agent Area Coverage Control and Optimization

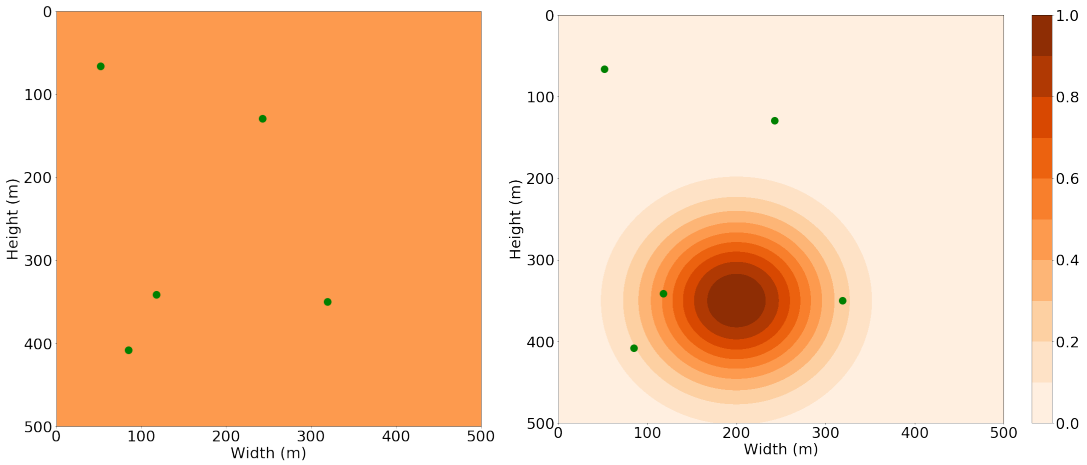
This thesis provides a design for a hardware test platform that is sufficiently flexible to be used for a variety of multi-agent control and estimation algorithms. However, for the scope of this thesis, the proposed design for the hardware test platform is evaluated on algorithms in the domain of multi-agent area coverage control and optimization as they have been developed by my research group [9–11]. Therefore, some relevant background theory in the field of area coverage control and optimization is presented in this section.

2.2.1 The Problem of Area Coverage Control and Optimization

There are a variety of problems that can be addressed using a multi-robot networks framework. In this general domain of problems, one specific problem of particular interest is area coverage control and optimization, which can be classified as a resource allocation problem.

In this problem, the objective is to determine the spatial distribution of a set of mobile agents that maximizes an index called coverage metric, which encodes the agents' sensing of the environment weighted by a scalar field called risk. The risk is distributed in the work space and encodes the relative importance of a given point [10, 11].

In the simplest case, the risk distribution is uniform. In the case of uniform risk distribution, the risk is constant throughout the environment, meaning that all points in the environment are of equal importance as represented in Fig 2.1a. In the case of nonuniform risk distribution, the risk is not constant throughout the environment. This means that some points in the environment are of higher importance than others. This is visualized in Fig 2.1b, where color intensity is correlated with the importance of a given point in the two dimensional planar environment. Fig 2.2 shows an alternative plot for visualizing nonuniform risk distribution in a given environment. The Z-axis is used here to visualize the relative importance of points on the XY plane. Higher risk density indicates more importance.



(a) Uniform risk distribution, where all points in the work space are of equal importance

(b) Nonuniform risk distribution, where some points have more importance than others (visualized by color intensity)

Figure 2.1: Comparison of uniform and nonuniform risk distribution (agents are visualized as green dots)

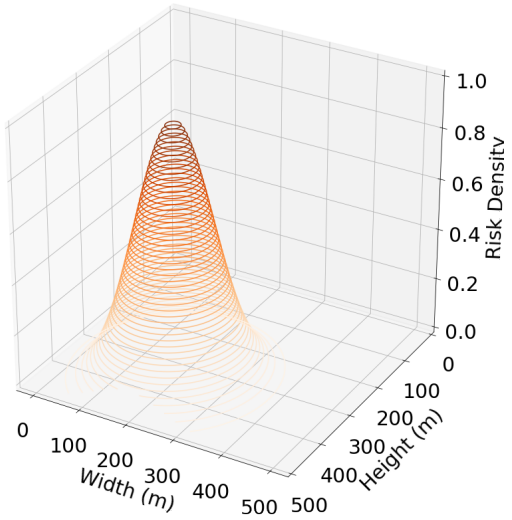
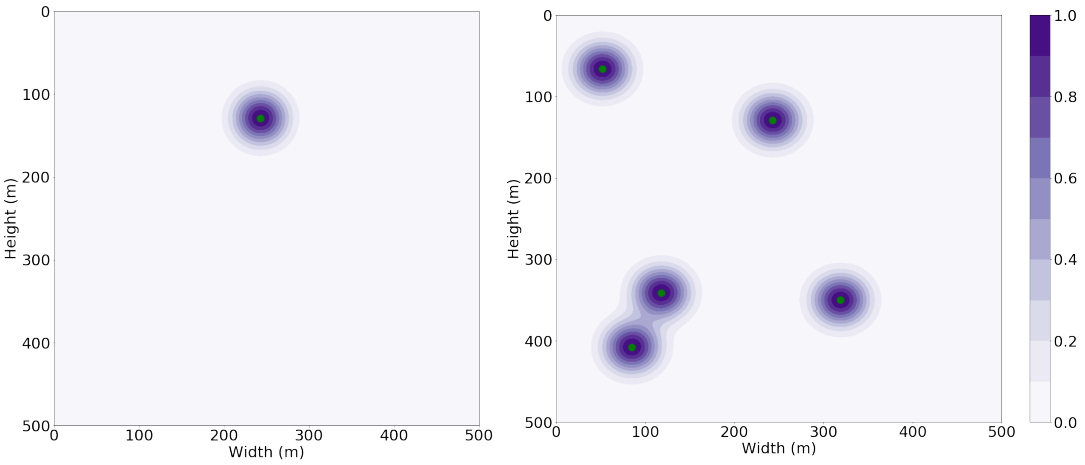


Figure 2.2: 3D plot of nonuniform risk distribution, where higher elevation indicates higher relative importance of a given point

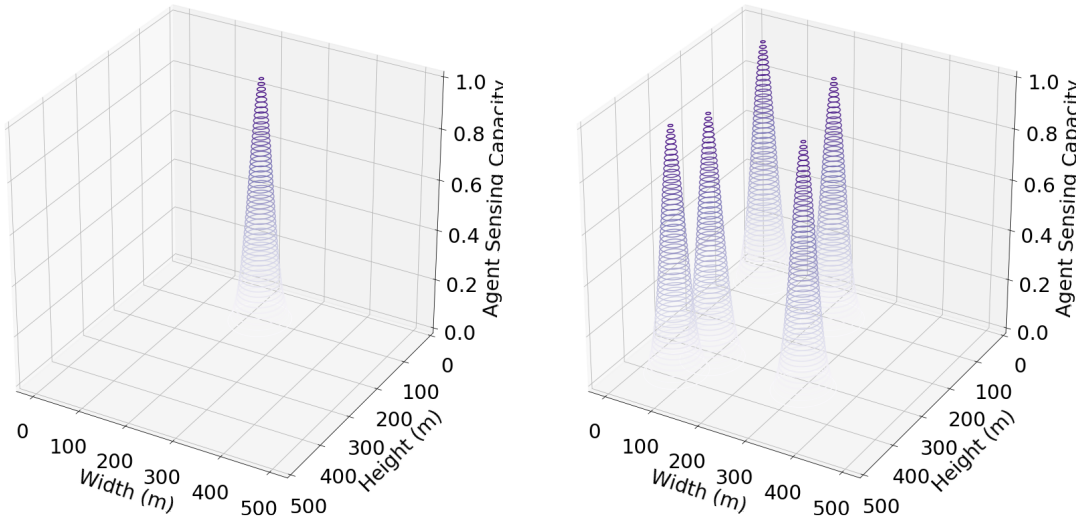
Agent sensing performance is another important component of the coverage metric. Agent sensing performance is typically modelled as a bell shaped surface that is centered at the position of a given agent. Typically an agent's sensing performance is highest at the position of the agent and decreases with the distance from the agent. Figures 2.3a and 2.4a show a visualization of this sensing performance around a single agent. As additional agents are added to an environment,

Figures 2.3b and 2.4b show the environmental sensing capacity around each of the five agents.



(a) Single agent (b) Five agents

Figure 2.3: Visualization of environmental sensing capabilities of agents (agents are visualized as green dots)



(a) Single agent (b) Five agents

Figure 2.4: 3D visualization of environmental sensing capabilities of agents. An agents' sensing capabilities decrease as distance from the agent increases.

The coverage metric calculation makes use of the risk density distribution in the work space along with the sensing capability of agents in the environment. The formal definition of the coverage metric is discussed in Section 2.2.4.1. The

objective of an area coverage control algorithm is to maximize the coverage metric. This results in optimal coverage and sensing policies with respect to the work space of interest, inducing optimal spatial configurations of the agents with respect to the coverage performance index.

2.2.2 Voronoi Diagrams

Voronoi diagrams emerge as solutions of area coverage control problems in describing the partition of a given spatial domain among the agents deployed in it, when the objective is the maximization of the area coverage. A Voronoi diagram is named after Georgy Voronoy and is sometimes referred to by other names such as a Voronoi tessellation, Voronoi decomposition, Voronoi partition or a Dirichlet tessellation [32, section 2.8].

Given a two dimensional planar environment with **N** sites, points, or seeds, a Voronoi diagram partitions the two dimensional planar environment into **N** regions. In the Voronoi diagram each site has a corresponding region. A region belonging to a particular site is defined by the collection of all points in the environment that are closer to that particular site than all other sites [32, section 2.8]. Moving forward these sites will be referred to as Voronoi generators, and the region corresponding to each site will be referred to as a Voronoi cell.

2.2.2.1 Formal Definition

Let Ω be a metric space with a distance function d. Let the expression $d(x, y)$ be the distance between point x and point y in the space Ω. Let P be a set of indices, that is $P = \{p_1, p_2, ..., p_n\} \in \Omega$, where $2 \leq n \leq \infty$. Given one of these indices p_i as a generator site, the corresponding Voronoi cell $\mathcal{V}_i$ is defined as the set of all points in Ω whose distance to p_i is less than or equal to their distance to other generators p_j, where j is any index different from i [33, chapter 3]. That is:

$$\mathcal{V}_i = \{q \in \Omega \mid d(q, p_i) \leq d(q, p_j), \forall j \ni i \neq j\} \tag{2.1}$$

The Voronoi diagram V of the set P is then defined as the following set [33, chapter 3]:

$$V = \{\mathcal{V}_1, \mathcal{V}_2, ..., \mathcal{V}_n\} \tag{2.2}$$

An interesting point to note is that the formal definition allows for an infinite

amount of generator sites to exist. However, most use cases only consider scenarios with finite numbers of generator sites.

2.2.2.2 Voronoi Diagrams in Euclidean Space

The use of Voronoi diagrams in Euclidean space is a popular and common implementation. The work of this thesis will also focus on Voronoi diagrams in the context of Euclidean spaces. When discussing Voronoi diagrams in this context, we treat each generator as a point in an Euclidean space. The collection of these generator points is both finite and unique. In this case, the resulting Voronoi cells are convex polytopes.

2.2.2.3 Example in Two-Dimensional Euclidean Space

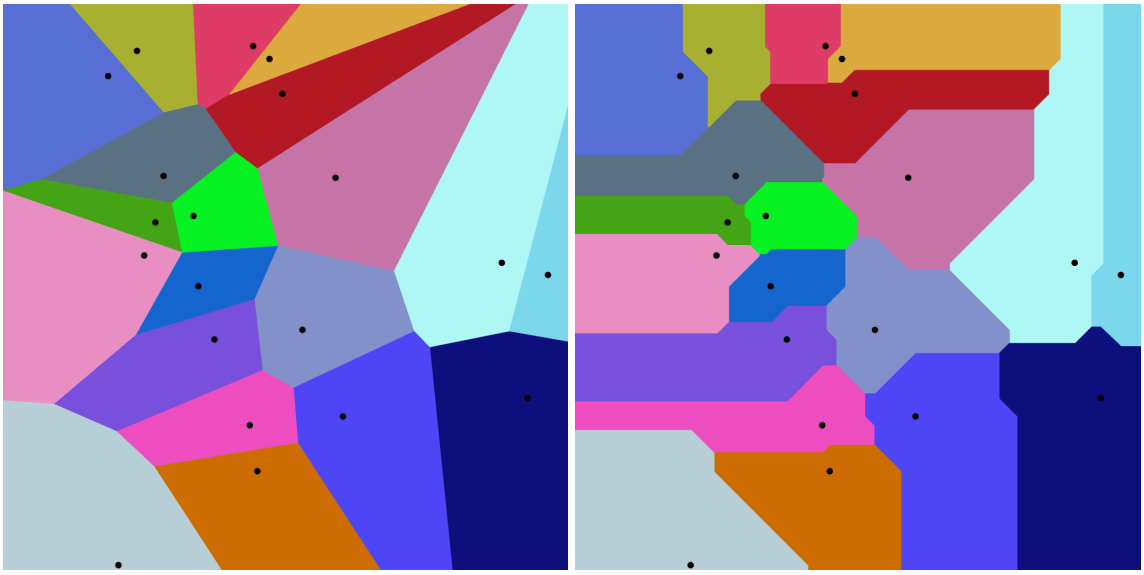
Consider that you work at a data analysis firm and are tasked with estimating the relative number of users that go to different gyms spread across the city. Making the assumption that all gyms provide the same level of quality, service, cleanliness and amenities, you decide to reason that users go to their gym of choice based entirely on proximity. That is to say that a user will always go to the gym that is closest to where they live.

You can view your city from above to obtain a two-dimensional Euclidean representation of it. Each gym in the city can then be treated as a Voronoi generator. This allows for the creation of a Voronoi cell for each gym in the city. An example of a Voronoi diagram consisting of 20 Voronoi generators is presented in Fig 2.5. Assuming each Voronoi generator represents a gym in the city, you can now easily visualize which gyms attract more users based on proximity.

As apparent in the formal definition presented above for Voronoi diagrams, the particular distance function that is chosen can have an impact on the final structure, properties and shape of the Voronoi diagram. This can be seen in action by first defining the following two points, $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$. Fig 2.5a shows the resulting Voronoi diagram when a Euclidean distance function is used. Euclidean distance defines the distance between P_1 and P_2 as:

$$d(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{2.3}$$

Another option is to use a Manhattan distance based function. Manhattan distance defines the distance between P_1 and P_2 as:



(a) Voronoi diagram based on a Euclidean distance function

(b) Voronoi diagram based on a Manhattan distance function

Figure 2.5: Voronoi diagram with 20 Voronoi generators

$$d(P_1, P_2) = |x_1 - x_2| + |y_1 - y_2| \tag{2.4}$$

The resulting Voronoi diagram in this case can be seen in Fig 2.5b. It is possible to see that these two figures have distinct appearances.

2.2.3 Centroidal Voronoi Diagrams and Lloyd's Algorithm

Centroidal Voronoi diagrams represent a special subset of Voronoi diagrams. A centroidal Voronoi diagram is one where the Voronoi generators of the diagram are also the mass centroids of their respective Voronoi cells in relation to a given density function [33, chapter 7]. An interesting note is that for any given environment, set of generators and an underlying density function for that environment, a centroidal Voronoi diagram is typically not unique. This means that there are usually multiple arrangements that the generators can be placed in to obtain a centroidal Voronoi diagram assuming the environment and its density function remain constant [33, chapter 7].

Centroidal Voronoi diagrams represent optimal coverage placements in the context of multi-agent area coverage [34]. For this reason they are a useful tool for multi-robot networks. Lloyd's algorithm is an iterative technique that can be used to find centroidal Voronoi diagram arrangements [33, chapter 7].

2.2.3.1 Lloyd's Algorithm

In the context of centroidal Voronoi diagram construction the algorithm begins by placing an n number of generators in the two dimensional euclidean space. After this, the algorithm becomes iterative, taking the following three steps repeatedly until a centroidal Voronoi diagram arrangement is found [33, chapter 7]:

- First the Voronoi cell, $\mathcal{V}$, of each Voronoi generator is calculated
- Next the mass centroid of each Voronoi cell is calculated
- In the third step, the position of each Voronoi generator is moved to coincide with the position of the mass centroid of its respective Voronoi cell

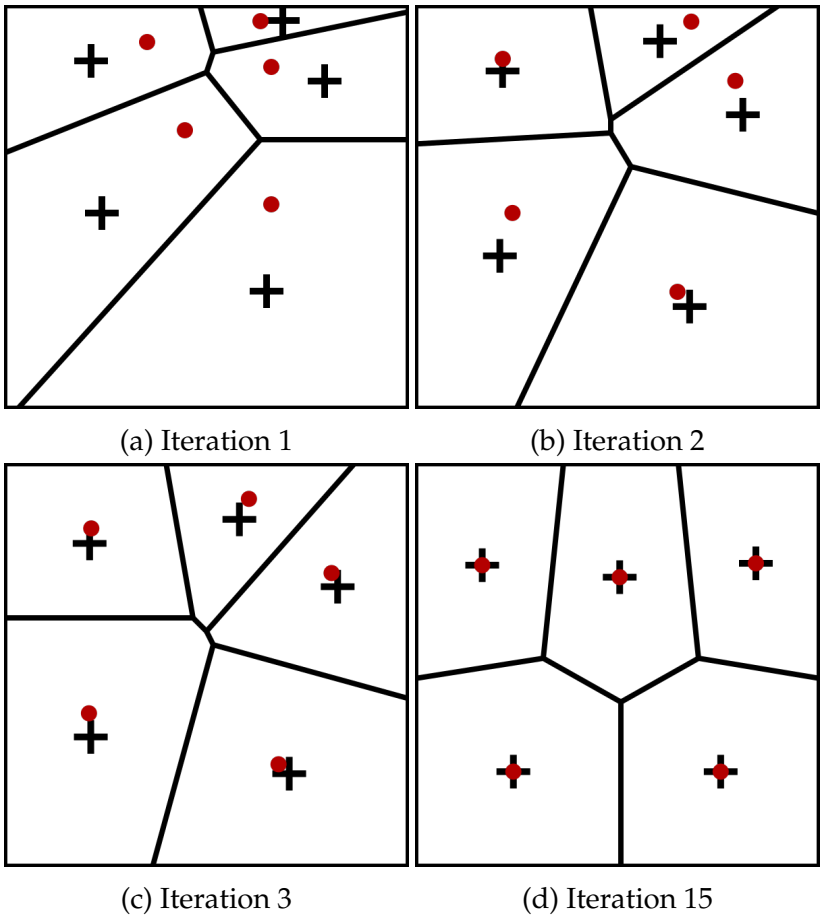
These three steps are repeated until a centroidal Voronoi diagram is achieved. An example of this process can be seen in Fig 2.6. In this particular example, convergence of the algorithm can be seen after 15 iterations, at which point the arrangement of the five generators results in a centroidal Voronoi diagram.

This algorithm converges effectively in one-dimensional scenarios, however has more limited convergence in higher order cases. In higher dimensions, convergence can occasionally take an unpractical amount of time, or instead can occasionally never occur fully. This can be attributed to lack of numerical precision in certain situations. For this reason practical applications of this algorithm do not always look for complete convergence [13]. Instead they set threshold values for convergence, which when met provide generator arrangements that result in nearly optimal centroidal Voronoi diagrams.

2.2.4 Area Coverage with Time Invariant Density Function

In the case of uniform density across the environment, it is assumed that all points in the environment hold equal weight with respect to area coverage. When a non-uniform density exists in the environment, it is typically identified by a function ϕ. In this scenario, the points in the environment no longer hold equal weight and are instead given value based on this density distribution. The function $\phi(q)$ denotes a time invariant density function as the density distribution is not a function of time and therefore remains constant in its form [12].

Given a two dimensional planar space Ω, and given a partition of this planar space $\mathcal{V}$, the following basic quantities are defined [12]:



(a) Iteration 1 (b) Iteration 2

(c) Iteration 3 (d) Iteration 15

Figure 2.6: Demonstration of Lloyd's algorithm

- Mass of $\mathcal{V}$:

$$M_{\mathcal{V}} = \int_{\mathcal{V}} \phi(q) dq \tag{2.5}$$

- Centroid (center of mass) of $\mathcal{V}$:

$$C_{\mathcal{V}} = \frac{1}{M_{\mathcal{V}}} \int_{\mathcal{V}} q\phi(q) dq \tag{2.6}$$

where $q \in \Omega$ and $\phi(q)$ is the time invariant density function across Ω.

2.2.4.1 Coverage Metric

Coverage metric is a scalar index which is used to quantify the area coverage performance of a group of mobile robots.

Consider a bounded environment Ω, where $\Omega \in \mathbb{R}^N$, containing a distribution of a finite number n of agents. The distribution of these agents can be described by introducing the collection $P = \{p_1, \ldots, p_n\}$, with the position of the $i-th$ agent in the bounded environment given by p_i. The coverage metric is then defined as [12]:

$$\mathcal{H}(P) = \int_{\Omega} \max_{i \in \{1,\ldots,n\}} f(d(q, p_i))\phi(q) d\Omega \tag{2.7}$$

where $\phi(q)$ is the time invariant density function of the environment. This density function essentially indicates the relative importance of different points $q \in \Omega$. The function d calculates the distance between q and p_i which is the position of the $i-th$ agent. The function f represents the sensing performance of each agent with respect to a point in the work space Ω. The function f is required to be continuously differentiable and monotonic with respect to d [12].

By using the sensing function f, we can generalize Equation 2.1 and define the Voronoi partition with respect to f [12]:

$$\mathcal{V}_i = \{q \in \Omega \mid f(d(q, p_i)) \geq f(d(q, p_j)), \forall j \ni i \neq j\} \tag{2.8}$$

Introducing this definition of a Voronoi cell provided in Equation 2.8 into Equation 2.7, the following expression can be obtained [35]:

$$\mathcal{H}(P) = \sum_{i=1}^{n} \mathcal{H}(p_i) = \sum_{i=1}^{n} \int_{\mathcal{V}_i} f(r_i)\phi(q) d\Omega \tag{2.9}$$

where $r_i = d(q, p_i)$ is the distance between q and p_i. Given the coverage metric

defined above, the problem of area coverage optimization essentially becomes an optimization problem defined as follows [35]:

$$\max_P \mathcal{H}(P) = \max_P \sum_{i=1}^{n} \int_{\mathcal{V}_i} f(r_i)\phi(q)d\Omega \tag{2.10}$$

If the distance function d is the Euclidean distance, the gradient of the coverage metric with respect to P gives [10]:

$$\frac{\partial \mathcal{H}}{\partial p_i} = \int_{\mathcal{V}_i} -2\frac{\partial f(r_i)}{\partial r_i^2}(q - p_i)\phi(q)d\Omega \tag{2.11}$$

Upon introduction of the the generalized density function [10]:

$$\tilde{\phi}(p_i, q) = -2\phi(q)\frac{\partial f(r_i)}{\partial (r_i^2)} \tag{2.12}$$

it is possible to generalize the definitions of mass and centroid in Equations 2.5 and 2.6 respectively, defining the generalized mass and generalized centroid as [10]:

$$\tilde{M}_{\mathcal{V}_i} = \int_{\mathcal{V}_i} \tilde{\phi}(p_i, q)dq \tag{2.13}$$

$$\tilde{C}_{\mathcal{V}_i} = \frac{1}{\tilde{M}_{\mathcal{V}_i}} \int_{\mathcal{V}_i} q\tilde{\phi}(p_i, q)dq \tag{2.14}$$

which results in the following form for Equation 2.11:

$$\frac{\partial \mathcal{H}}{\partial p_i} = \tilde{M}_{\mathcal{V}_i}(\tilde{C}_{\mathcal{V}_i} - p_i) \tag{2.15}$$

Lloyd's algorithm is an iterator to find local optima of the coverage metric by driving the agents to centroidal Voronoi tessellations. Assuming that the agents are kinematically actuated according to

$$\dot{p}_i = u_i \tag{2.16}$$

Lloyd's algorithm drives the system of agents to optimal configurations (maximizing the coverage metric) through the feedback [36]:

$$u_i = \gamma \frac{\partial H}{\partial p_i} = \gamma \tilde{M}_i(\tilde{C}_{\mathcal{V}_i} - p_i) \tag{2.17}$$

where $\gamma > 0$ is a gain. Optimality is implied by the fact that the coverage metric monotonically increases along the trajectories generated by the algorithm

$$\dot{H} = \sum_i \frac{\partial H}{\partial p_i} \cdot \dot{p}_i = \gamma \sum_i \tilde{M}_i^2 \left\| \tilde{C}_{\mathcal{V}_i} - p_i \right\|^2 \geq 0 \tag{2.18}$$

with the equality holding at centroidal configurations $\tilde{C}_i = p_i$.

2.3 Desired Features of a Multi-robot Network Test Platform

This section focuses on the design of the multi-agent algorithm test platform. More specifically, background theory and literature review is presented which is relevant to the desired features and capabilities of the hardware testing platform along with the capabilities of the robotic agents within the test platform. The required general capabilities that will be discussed are:

- Reliable inter-agent communication protocol.
- Accurate indoor localization of robotic agents with respect to a global environment.

Inter-agent communication is a crucial requirement for the hardware test platform as communication is at the core of strongly cooperative multi-robot networks with decentralized or centralized architecture. A communication protocol allows for flow of information to and from each agent in the network. This provides a method of wirelessly sending control actions to robotic agents in a centralized architecture. Using wired connections to achieve the same task has a significant impact on the kinematics of the robots in the test platform and introduces additional challenges in the lab setting. Communication between agents is also crucial for tasks that require coordination in decentralized architectures. Additionally, environment related information can be shared among agents as well, which is important for cooperative estimation [8] or cooperative manipulation tasks [29–31]. The notion of reliable communication is important for optimal function of the hardware test platform as it ensures that data packet drops do not occur.

The hardware test platform is to be designed in an indoor lab environment, and therefore a method for indoor localization of the robotic agents is necessary.

This framework will identify the positioning of each agent in the environment, which is crucial knowledge for multi-agent algorithm testing. The goal is to find a framework that can localize agents in the environment to within 1 cm. This level of accuracy will result in a more practical and effective hardware test platform as localization errors will be minimized.

For the reasons discussed above, an inter-agent communication protocol and an indoor localization framework are both crucial for the design of a multi-agent algorithm testing platform that is sufficiently flexible to be used for a variety of multi-agent control and estimation algorithms.

2.3.1 Inter-agent Communication Protocols

This section presents background theory with respect to existing technologies that are used for inter-agent wireless communication in multi-robot networks. Wireless communication capability is an important component of any strongly cooperative multi-robot network and allows for the distribution of information across the network. Beyond just as a method of sharing information within the network, in the case of a centralized multi-robot network architecture, communication is an important tool for providing control inputs from the central controller to the nodes of the network. Furthermore, as will be discussed in Section 2.3.2, wireless communication can be used for localization purposes as well.

While a variety of wireless communication protocols are available for use, this discussion will limit its focus to low power protocols. Since the robotic agents will be standalone units running on small batteries, a low power protocol will be able to perform effectively with a limited power supply. Low power protocols typically have lower bandwidth, or rate of data transfer, when compared to high power alternatives. This can be an issue for applications that require high bandwidth, however, high bandwidth is not a requirement for the objectives of this thesis. Specific bandwidth information for some low power protocols is provided in Table 2.2. Some protocols that fit the low power criteria include [37–39]:

- Bluetooth Low-Energy (BLE)
- ANT/ANT+
- ZigBee
- WiFi HaLow (IEEE 802.11ah)

- NIKE+
- Infrared Data Association (IrDA)
- Near Field Communication (NFC)

When attempting to choose the proper wireless communication protocol, it is important to consider the capabilities and limitations of each option that is available to you. The following text will investigate the architecture compatibility, range capability, throughput rate, latency, power requirements and finally the cost of each of these protocols to arrive at the optimal choice.

2.3.1.1 Wireless Communication Architecture Compatibility

Some common wireless communication architectures include [37, 38]:

- **Broadcast:** A transmitter transmits a message that is not for a specific target receiver. Any receiving modules within range will receive this message, without providing any acknowledgement of receiving the message.
- **Peer to Peer:** In the case of peer to peer communication, two transmitters are able to communicate bidirectionally and provide acknowledgement to each other when they receive a message.
- **Practical Mesh:** Mesh arrangements consist of a group of networks. Typically each network has a central transmitter (hub) that has bidirectional communication capability with other receivers in the network. Each network also contains relay units which have bidirectional communication capabilities with relay units in other networks. Through these relay units, the entire group of networks can communicate effectively.
- **Star:** A star arrangement consists of a single central transmitter which has bidirectional communication capabilities with all other receivers in the network. These receivers are not able to communicate with other receivers in the arrangement.
- **Scanning Mode:** Scanning mode refers to an arrangement where a central receiver is continuously listening for messages from any transmitters that are within range. Communication in this arrangement is only one way, from transmitters to receiver.

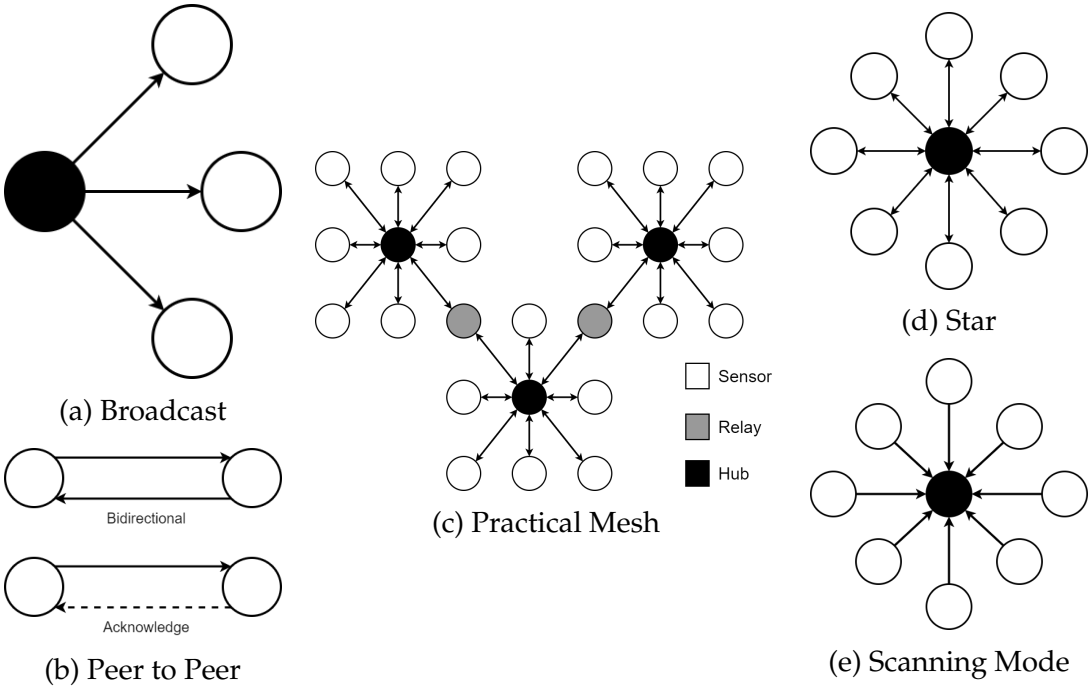


Figure 2.7: Visualization of Wireless Communication Architectures [37]

These five arrangements are visualized in Fig 2.7. For the application of a multi-robot network, a more versatile communication architecture is more appropriate. From the architectures discussed above, the practical mesh is the most suitable option for the objective of this thesis. With a practical mesh arrangement for wireless communication, a variety of multi-robot network architectures can be modelled (centralized, hierarchical, decentralized or hybrid).

Table 2.1 shows a summary of the compatibility between various communication protocols and various communication architectures. As the desired architecture is the mesh structure, the scope of this analysis becomes limited to Bluetooth Low-Energy (BLE), ANT/ANT+ and ZigBee protocols.

Architecture	BLE	ANT/ANT+	ZigBee	WiFi	NIKE+	IrDA	NFC
Broadcast	✓	✓					
Peer to Peer	✓	✓	✓	✓	✓	✓	✓
Star	✓	✓	✓	✓			
Scanning	✓	✓	✓		✓		
Mesh	✓	✓	✓				

Table 2.1: Architecture Compatibility for Wireless Communication Protocols [37]

2.3.1.2 Range, Throughput, Latency, Power Requirements and Cost

Some additional protocol characteristics that are important to consider include the range capability, throughput rate, latency, power requirements and cost [37, 38, 40]. Having limited the discussion to three protocols of interest, this analysis will now dive deeper into these additional requirements.

The concept of range is an important consideration when discussing wireless communication protocols. Given that agents in the multi-robot network will require effective communication across distances of a few meters ($\sim$ 10 meters) in the lab setting, it is important to choose a protocol that meets this requirement.

Throughput rate refers to the rate of data transfer (bandwidth) that these protocols are capable of. In a general sense, a higher throughput rate allows for faster and more complete communication. There are no baseline requirements for this parameter in the application of this thesis, however typically a higher throughput rate is more desirable.

Latency refers to the time delay between the moment when a message is sent from a transmitter to the moment when a message is received by a receiver. In the case of a multi-robot network algorithm testing platform, an extremely low latency is not a major requirement, and a generally low latency ($<$ 50 milliseconds) is sufficient. This is a direct consequence of the fact that the robotic agents that will be used in the hardware test platform have relatively slow kinematics. The proposed robotic agents (Section 2.4) require a time scale of $\sim$ 2000 milliseconds to execute typical control actions. Relative to these time scales, a generally low latency ($<$ 50 milliseconds) is minimal and is found to be sufficient for effective communication between agents.

Power requirement is an important consideration in a multi-robot network as it dictates the battery life of the agents in the system. In the case of a test platform for a lab setting, lower power requirements allow for more tests per charging cycle and generally provide a more friendly user experience.

Finally minimizing cost is an important parameter for scaling the size of the multi-robot network. As each additional agent that is added to the multi-robot network will require additional communication hardware, a lower cost protocol will allow more room for scalability.

A summary of all these parameters for the Bluetooth Low-Energy (BLE), ANT/ANT+ and ZigBee protocols is provided in Table 2.2. Values provided in Table 2.2 are estimates and can vary based on application and availability.

The ANT/ANT+ protocols have the least desirable characteristics for the ap-

Characteristic	BLE	ANT/ANT+	ZigBee
Range (m)	100	30	100
Throughput (Kbits/s)	305 - 1000	10 - 20	200 - 250
Latency (ms)	2.5	negligible	20
Power Requirement (nJ/bit)	75	715	360
Cost ($/100 chips)	3.99	4.09	3.62

Table 2.2: Summary of Characteristics for the Bluetooth Low-Energy (BLE), ANT/ANT+ and ZigBee Protocols [37, 40].

plication of this thesis. While they provide low latency, they have the lowest range capability, lowest throughput rate, highest power requirements and highest cost. In this situation the low latency is not worth the trade off.

The remaining two protocols have similar characteristics. They have 100 meters of range capability. They both have high throughput rates (enough to satisfy the requirements for this thesis). They have low latency (< 50 milliseconds). The Bluetooth Low-Energy protocol has superior performance when it comes to power requirement, with a more efficient power requirement per bit of data sent. However, ZigBee technology is found to generally be the cheaper alternative. Important to note is that the prices provided in Table 2.2 provide a general relative pricing for these protocols. In reality, these chips can be priced higher based on the specific application and the manufacturer.

To comment further on the range capability of these protocols, 100 meters of range is more than sufficient for the requirements of this thesis ($\sim$ 10 meters), due to the small size of the testing setup in the lab. However, it should be noted that for some applications, especially outdoor applications, a larger than 100 meter range of communication can be desirable or even necessary for reliable communication within a large testing space.

The lower general price of the ZigBee protocol combined with its satisfactory performance with respect to range, throughput, latency and power requirements, makes it the optimal choice for a cost effective multi-robot network algorithm testing platform.

2.3.2 Indoor Localization of Mobile Terrestrial Robots

While the problem of outdoor localization is typically addressed using GPS, the lack of access to GPS technology indoors makes the problem of indoor localization less trivial. The aim of this discussion is to explore methods of localizing mobile

agents in an unchanging environment with low error ($\sim$ 1 cm). It is also assumed that the environment is already mapped and the location of the mobile agents in the environment must be estimated. The following are some existing solutions in literature for solving this problem.

2.3.2.1 Wireless Communication Based Localization

Wireless communication based localization is compatible with a variety of communication protocols such as Wi-Fi, Ultra-Wideband, Bluetooth, RFID, Ultrasound and even ZigBee. Based on the application, some of these protocols can provide better performance that others in terms of precision and reliability. The following methods are commonly used to localize a target node using principles related to wireless communication protocols [41, 42]:

Received Signal Strength Indicator (RSSI): The received signal strength indicator method for localization is quite straightforward. In a basic sense, a signal is sent from a transmitter to a target receiver. The relationship between the strength of this signal and the distance that this signal has travelled is known which allows for calculation of the absolute distance between the transmitter and receiver. As an example, the following equation calculates the received signal strength indicator value for a simple path-loss propagation model [41].

$$\text{Received Signal Strength Indicator} = -10n\log_{10}(d) + A \tag{2.19}$$

where n is the path loss exponent which can vary based on the environmental conditions and A represents the received signal strength indicator value at a known reference distance between the transmitter and receiver (Fig 2.8a).

Angle of Arrival (AoA): Angle of arrival makes use of antenna array structures (a single receiver and transmitter unit that consists of multiple antennas) to determine the angle of incidence of an incoming signal. When an antenna array receiver receives a signal, the time difference of arrival is calculated for all the individual antennas that make up the larger antenna array. The time difference of arrival values calculated can then be used to determine an estimate for the angle of origin for the signal that was received [41]. This is explained in Fig 2.8b.

Time of Flight (ToF): Time of flight techniques calculate the distance between a transmitter and a target receiver by measuring the time required for a signal to travel from the transmitter to the target. The speed of the signal is typically assumed to be equal to the speed of light, $c = 3\ x\ 8^{10}\ m/s$, which multiplied by the time of flight of a signal will provide an estimated distance value. When using this method, it is important to ensure that the transmitting and receiving devices are synchronized in terms of time measurement. Often times the structure of the packet being sent from the transmitter to the receiver will also contain a time stamp in order to allow for calculation of the time of flight. On the receiving side, a high sampling rate is important for accurate measurement as it allows for acknowledgement of a signal the moment a signal arrives. If sampling rate is low, a signal might arrive in between two sampling points, impacting the accuracy of the time of flight calculation (Fig 2.8c) [41].

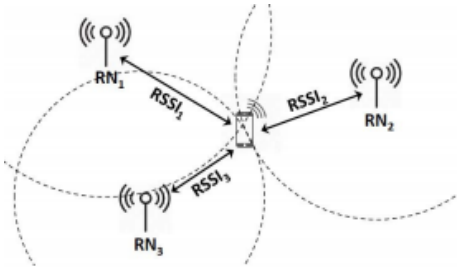
Time Difference of Arrival (TDoA): When using time difference of arrival, the goal is to localize a target transmitting device. This target device transmits a signal and unlike in the time of flight technique, the time at which this signal is sent is not important. Instead the focus here is to determine the time at which this signal arrives at various receiver sites with known locations. The difference in time required for this signal to arrive at various receiver sites can then be used to localize the transmitting device. Using the speed of the signal (typically set to $c = 3\ x\ 8^{10}\ m/s$) and the time difference of arrival between receivers i and j, the following distance relationship is obtained [41]:

$$L_{D(i,j)} = \sqrt{(X_i - x)^2 + (Y_i - y)^2 + (Z_i - z)^2} - \sqrt{(X_j - x)^2 + (Y_j - y)^2 + (Z_j - z)^2} \tag{2.20}$$

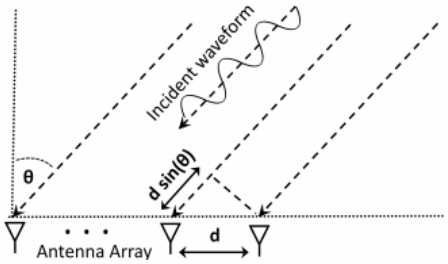
Where (X_i, Y_i, Z_i) and (X_j, Y_j, Z_j) refer to the positions of receivers i and j respectively and (x, y, z) refers to the position of the target transmitter. This equation models a hyperboloid and the target is located somewhere on the surface of this hyperboloid. Using this time difference of arrival calculation with three or more receiving sites is required for precise localization of the target transmitter (Fig 2.8d) [41].

Table 2.3 provides a summary of localization characteristics for some common wireless communication protocols.

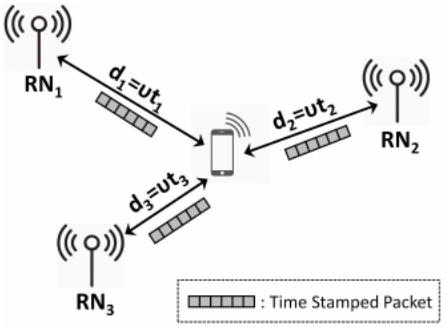
Section 2.3.1 discusses a variety of low power wireless communication protocols for inter-agent communication. In this section, the ideal protocol was found



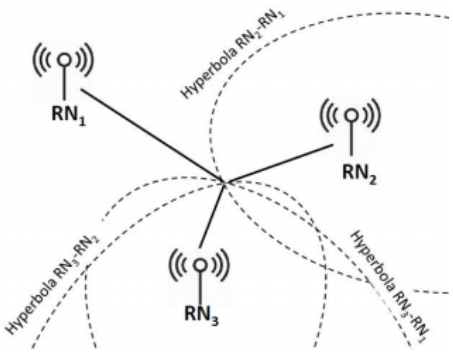
(a) Triangulation Using Recevied Signal Strength Indicator



(b) Angle of Arrival Schematic



(c) Triangulation Using Time of Flight



(d) Visualizing Time Difference of Arrival

Figure 2.8: Methods for Localization Using Wireless Communication [41]

to be the ZigBee protocol for its reliable characteristics and lower price range. Ideally, this same protocol could be used for the purpose of localization as well. This would significantly reduce hardware requirements. However, the localization accuracy provided by the ZigBee protocol is not precise enough. An accuracy closer to the range of $\sim$1cm is required for a multi-robot network test platform within a lab setting. The Ultra-Wideband and the ultrasound protocols seem to provide a higher degree of accuracy, however their low scalability and high cost make them unsuitable options. The infrared protocol has the potential to provide localization accuracy within the centimeter range at a lower cost than the Ultra-Wideband and ultrasound protocols. The following sections will introduce additional localization techniques that do not involve the use of wireless communication protocols.

2.3.2.2 Dead Reckoning Localization

The dead reckoning localization technique estimates the current position of a target object based on a known previous position. The change in position from the previous position to the current position is estimated based on estimated values for velocity and heading of the target object. The velocity and heading of the tar-

System	Methods	Accuracy	Scalability	Cost
WiFi	RSSI, AoA, ToF, TDoA	1m - 10m	high	low
Ultra-Wideband	RSSI, AoA, ToF, TDoA	1cm - 1m	low	high
Bluetooth	RSSI, ToF	1m - 5m	high	low
RFID	RSSI	1m - 5m	medium	low
ZigBee	RSSI	1m - 10m	low	medium
Infrared	ToF	1cm - 5m	low	medium
Ultrasound	ToF, TDoA	1cm - 1m	low	high

Table 2.3: Summary of Localization Characteristics for Wireless Communication Protocols [43]

get object can be estimated through the use of a variety of sensors or odometers [43]. This can include accelerometers, gyroscopes, compasses and wheel encoders to name a few. The main benefit of dead reckoning localization is that it requires no external infrastructure for reference, as all localization can be done locally on the robotic agent. However, the main drawback of this technique is that error values can compound over each iteration and over time the error value can grow to a point where the technique becomes obsolete [43]. This drawback can be tackled partially through the implementation of filters (complementary filter, kalman filter) on the sensor readings.

2.3.2.3 Video Scene Analysis Localization

Video scene analysis focuses on processing video information collected from the environment for the purpose of obtaining localization information.

On one hand, this can be done locally at the level of the robotic agents in the multi-robot network. In this situation, the robotic agents themselves collect visual information about the environment and use pattern matching techniques to localize themselves. The online visual information collected can be compared with an offline database containing visual information and localization information pairs. In a more specific example, QR codes or barcodes can be placed around the environment of interest. These codes contain localization information and act as landmarks for the environment of interest. When an agent detects a code and processes the relevant localization or identification information, it is able to localize itself in the environment of interest [43].

A different method of using video scene analysis is by creating an infrastructure that is external to the agents within the multi-robot network. This infrastructure can then collect visual information from the environment and can process this

data in a way to localize target objects (robotic agents) within the environment. For example, an environment can be covered by an overhead camera. Identification barcodes or QR codes can then be placed on the target agents that need to be localized. This external camera can then track each agent in the environment using the respective identifier codes. This can also be implemented using pattern matching techniques or machine learning techniques for object identification [43].

Video scene analysis localization methods are capable of providing high accuracy localization ($\sim$ 1 cm error). Depending on the application, these methods can be implemented at low costs as they reduce hardware requirements. These methods require a good understanding of the target environment and are more environment specific than dead reckoning localization methods [43]. However, more recent advancements in machine learning are helping make these localization methods more versatile and generalized.

2.4 Options for Hardware Robotic Systems

This section presents some options for mechatronic systems that can be used as robotic agents in the multi-agent algorithm test platform. In recent years the field of robotics has seen significant advancements due to progress in related fields such as computation, manufacturing and quick prototyping. As such, a variety of mechatronic systems exist both in literature and on the market for the purpose of assembling multi-agent systems. The main criteria for selection were cost effectiveness, simplicity, accessibility, customizability, modularity, and finally the satisfaction of the basic requirements laid out in Section 2.3. The following options were selected as potential viable solutions.

2.4.1 Kilobot

Figure 2.9: Kilobot robot [44]

The Kilobot (Fig 2.9) is a low cost, relatively simply robot that is designed specifically for use in swarm systems. It is a small robot, with a circular shape of 33mm diameter and a height of 34mm. The robot has no wheels for locomotion and instead it rests on three legs as support structures. It uses two coin shaped vibration motors, one attached on either side of the robot for locomotion. Activation of one of these motors generates centripetal forces that cause a forward force at the location of the motor on the Kilobot. This principle is referred to as the slip-stick principle [45]. By independently controlling the vibration of the two motors, the robot can be actuated to rotate or move forwards. The cost of the Kilobot is ∼160 CAD per unit. The robot itself includes recharging capabilities and has environmental sensing capability in the form of being able to measure the brightness of the ambient light that is incident on its upper surface [44].

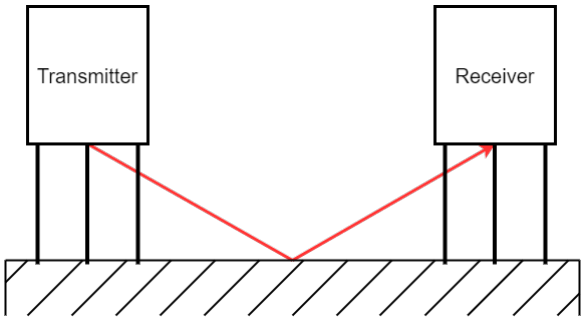


Figure 2.10: Infrared communication between two Kilobots

Communication between two Kilobots is achieved using infrared signals. A transmitting robot sends an infrared signal which reflects off the ground and is received by a receiving agent (Fig 2.10). Using this method of communication, Kilobot receivers can estimate the distance to the Kilobot transmitter using the measured received signal strength indicator. This can be useful for localization purposes. The main drawback of the Kilobot is that it can only communicate with a neighbouring Kilobot within a range of 7cm [44]. This range is relatively small for the purpose of a multi-agent algorithm testing platform. This small radius of communication limits the complexity of wireless communication architectures that can be achieved.

2.4.2 Alphabot and Alphabot2

The Alphabot and Alphabot2 models are both viable options as well. Both models can be seen in Fig 2.11. They have similar functionality, as they are both differential

(a) Alphabot

(b) Alphabot2

Figure 2.11: Alphabot robots with arduino compatibility [46, 47]

drive robots. They both provide an ultrasonic proximity sensor for distance measuring and obstacle avoidance. They both contain two additional infrared proximity sensors for improved obstacle avoidance capability. Both of these robots also have a line tracking module for line detection and line tracking. They are also both compatible with the popular Arduino micro-controller, which makes them easier to setup and use. Finally, due to the Arduino based controller, both these robots are compatible with the XBee wireless communication protocol [46, 47]. The XBee protocol is built on the ZigBee protocol discussed in Section 2.3.1, which was found to be an optimal option for the application of this thesis due to its reliable characteristics and lower price range. Additional details on this finding are discussed in Section 2.3.1. The cost of the Alphabot and Alphabot2 is ∼80 CAD per unit, which is about half the cost of the Kilobot.

The Alphabot is unique in that it provides optical wheel encoders, allowing for estimation of the robots speed. This can be useful for any dead reckoning based localization techniques. On the other hand, the Alphabot2 is more integrated and as a result is easier to assemble. It is smaller in size and less bulky. It also has built in charging capability allowing for ease of use in the lab setting when multiple units are being used [46, 47].

2.4.3 UCTRONICS K0072

The K0072 robot from UCTRONICS (Fig 2.12) is another Arduino compatible option which has a cost of ∼70 CAD per unit. It provides some similar features to the Alphabot robots such as ultrasonic sensing and line tracking. The Arduino microcontroller is compatible with the XBee protocol. The K0072 robot also comes with

built in compatibility with an HC-05 bluetooth module, allowing for bluetooth communication between different agents. However, this wireless communication protocol does not lend itself well to a practical mesh arrangement as seen in Section 2.3.1.

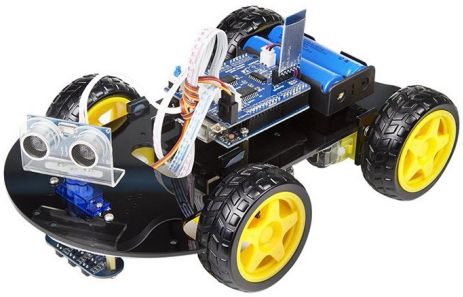


Figure 2.12: UCTRONICS K0072 robot with arduino compatibility [48]

Finally, the K0072 is a four wheeled robot as opposed to the other options which are differential drive. This four wheel design provides more power and speed, however also results in additional complexity with regard to control action. The presence of four independent motors can also increase the chances of less precise motion [48].

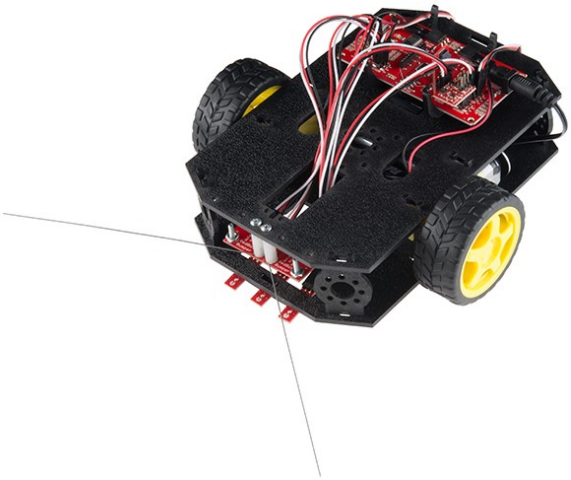


Figure 2.13: SparkFun RedBot robot with arduino compatibility [49]

2.4.4 SparkFun RedBot

The RedBot (Fig 2.13) comes in at a higher price of ∼160 CAD and has a two wheel, differential drive design. The RedBot has a line tracking module, and also comes

with two magnetic wheel encoders for speed estimation. With regard to dead reckoning localization techniques, the RedBot is fitted with an accelerometer which can be used to estimate speed and direction of motion of the robot. The RedBot uses mechanical bumpers for obstacle detection and avoidance, which require contact to function properly, as opposed to the ultrasonic sensors used by the other robots. This robot is also Arduino compatible and therefore can make use of the XBee protocol for wireless communication. However, the higher price of this model makes this option less viable when compared to the cheaper Alphabot robots and the UCTRONICS robot [49].

2.4.5 Summary of Options for a Hardware Robotic System

Table 2.4 summarizes some key characteristics for the five hardware robotic systems discussed above.

System	Kilobot	Alphabot	Alphabot2	UCTRONIC K0072	SparkFun RedBot
Cost/Unit (CAD)	160	80	80	70	160
Locomotion	Motor Vibration	Differential Drive	Differential Drive	Four-Wheel Drive	Differential Drive
Inter-agent Communication	Infrared	XBee or ZigBee	XBee or ZigBee	XBee, ZigBee or Bluetooth	XBee or ZigBee
Control Board	Atmega 168	Arduino Uno	Arduino Uno	Arduino Uno	Arduino Uno
Obstacle Avoidance		✓	✓	✓	✓
Wheel Encoder		✓			✓
Accelerometer					✓
Rechargeable	✓		✓		
Line Tracking		✓	✓	✓	✓
Ambient Light Sensing	✓				

Table 2.4: Summary of Characteristics for Hardware Robotic Systems

With respect to making a design choice about the hardware robotic system to be used for the test platform from among the various designs mentioned in Section 2.4, the Kilobot was found to be insufficient due to its price and lack of range for wireless communication. The RedBot was also eliminated as an option due to high cost per unit compared to alternative robots. The Alphabots were chosen as more optimal options when compared with the UCTRONICS K0072 four wheeled robot. This was due to the fact that typical theoretical frameworks for multi-agent systems assume the agents themselves to be point agents. Translating these point agents into hardware implementation is a more simple and reliable task with a differential drive robot as the four wheeled K0072 will introduce unnecessary complexity and error to the system.

When deciding between the Alphabot and the Alphabot2, the Alphabot2 was chosen as the more desirable option due to a more integrated design, improved build quality and a smaller overall frame.

Chapter 3

Multi-agent Algorithm Test Platform Design and Implementation

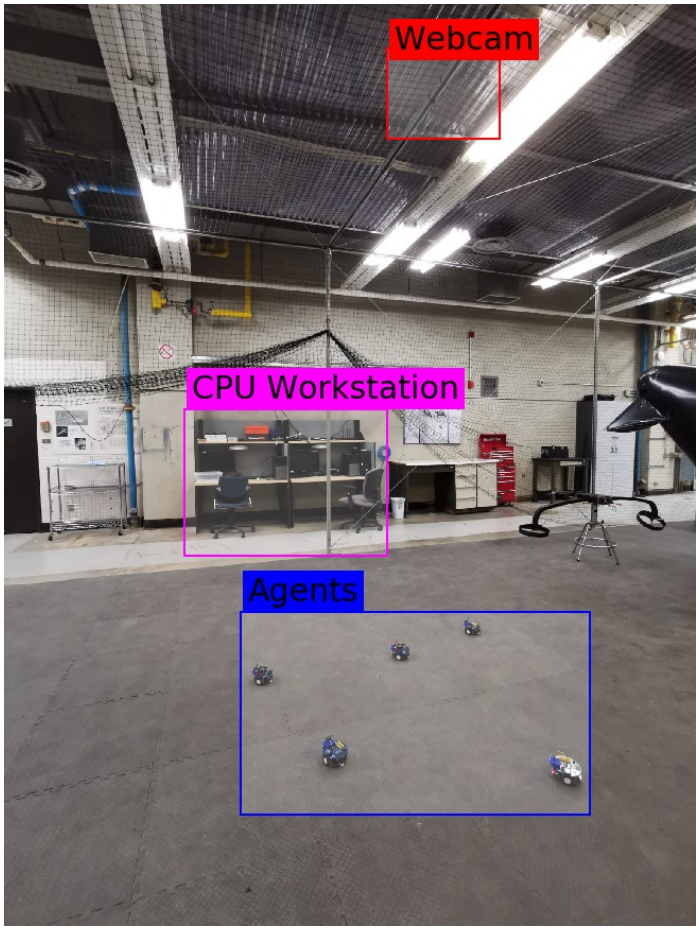
This chapter will discuss the proposed design for the multi-agent hardware test platform. The test platform is meant to be for general purpose algorithm testing in the domain of multi-agent systems. The sections of this chapter will discuss the various design choices and will provide a more detailed summary of the methods used to implement the design choices that were made.

As a method of verification, the testing platform will be used to verify existing algorithms in the domain of multi-agent coverage control and optimization. As such, part of this chapter will discuss the implementation of existing coverage control and optimization algorithms in hardware.

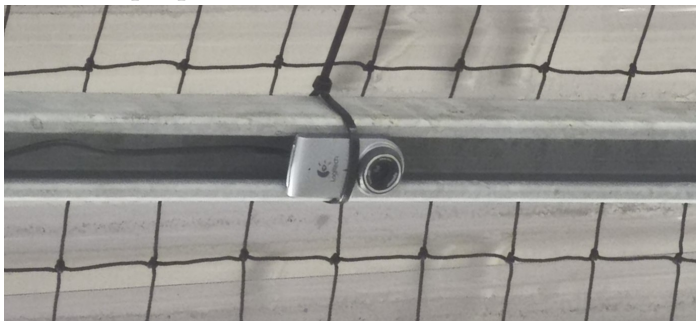
3.1 Proposed Design

This section introduces the proposed design and experimental setup. Fig 3.1a shows the full lab setup and indicates the location of the overhead webcam, the position of the agents and the location of the workstation where the central processing unit is located. The webcam is attached to a metal beam that is located above the work space and is pointing downwards as seen in Fig 3.1b. A general schematic overview of the full design is shown in Fig 3.2.

A centralized architecture is visualized in Fig 3.2a since the central processing unit is responsible for all computation, and it then sends control actions to the robots in the network. This centralized architecture is the one that is tested within the scope of this thesis due to initial simplicity of the setup along with easy syn-



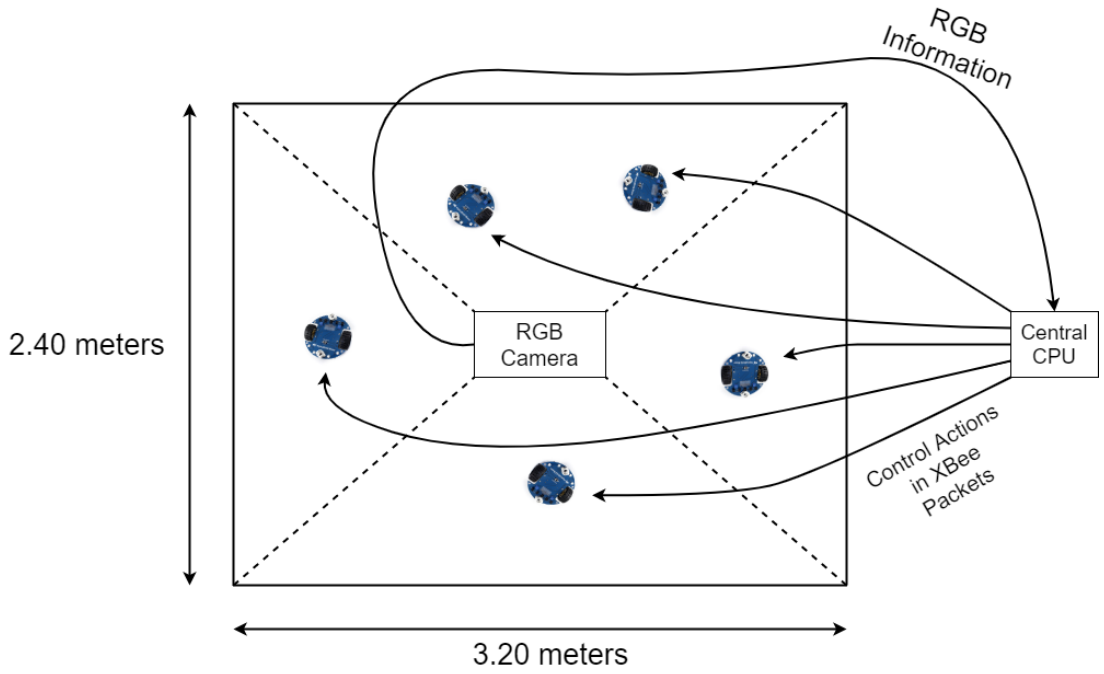
(a) Lab setup showing the webcam location in red, agents in blue and the central processing unit workstation in purple



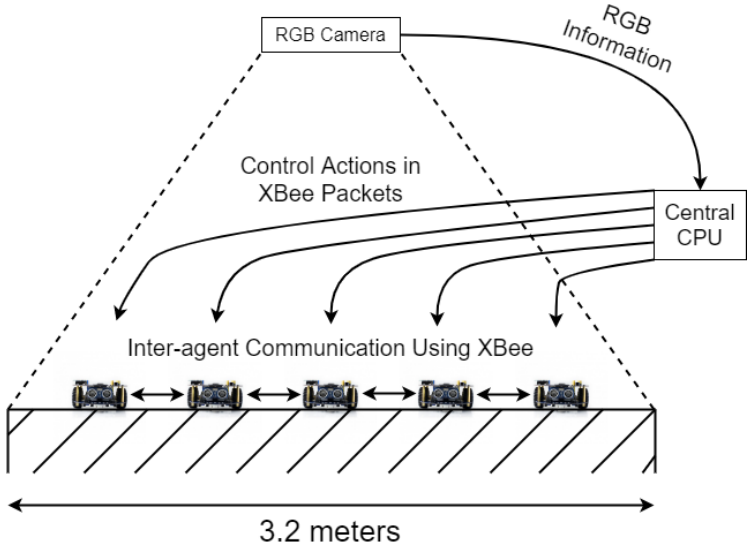
(b) Webcam attached to metal beam above the robotic agents

Figure 3.1: Lab setup showing the central processing unit workstation, the location of the webcam and the agents on a planar environment

chronization of all robots in the network. However, it is important to note that the XBee protocol allows for inter-agent communication as well, as represented



(a) Top view of proposed design



(b) Front view of proposed design

Figure 3.2: Full schematic of the proposed design for the hardware test platform

in Fig 3.3. These capabilities allow for the implementation of a decentralized architecture where computation occurs at the local level of each robot and information is distributed within the network using inter-agent communication. While the decentralized architecture is not tested or implemented in this thesis, the XBee protocol allows these capabilities to exist and ongoing work is being dedicated to

implement this decentralized architecture.

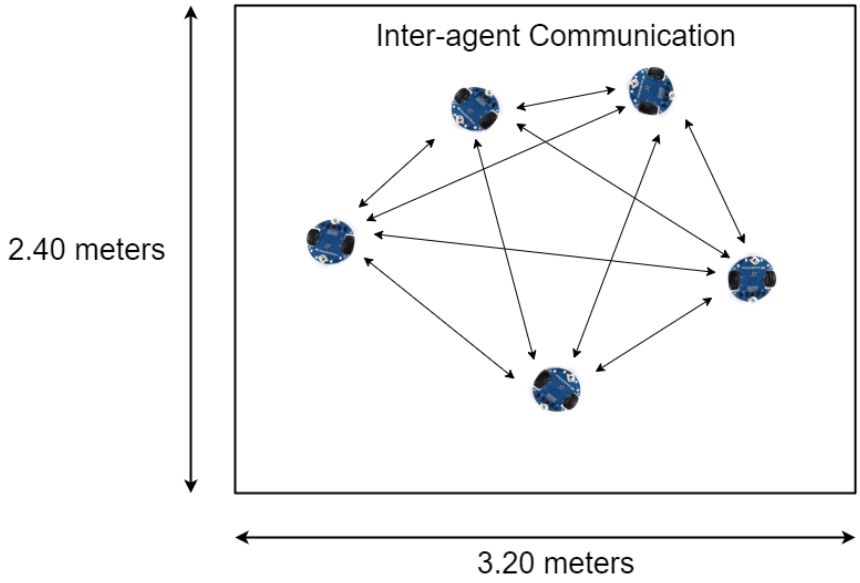


Figure 3.3: Inter-agent communication capabilities

The following sections explain the working of the major components of the test platform such as the type and architecture of the test platform, the choice for mechatronic agents, the wireless communication protocol being implemented and finally the method of localization being implemented.

3.1.1 Type and Architecture of Test Platform

Before making any hardware related decisions, it is important to make decisions informed by design constraints, as they relate to the test platform and the desired type and architecture as discussed in Section 2.1. Due to the fact that this test platform should be able to implement a variety of multi-agent algorithms, the goal will be to develop an intentionally cooperative framework that is capable of strong cooperation as referenced in Section 2.1.1. Within this context, strong cooperation refers to each agent in the network acting based on up to date and accurate information on the state of all other agents in the network. In the case of centralized architecture, the central processing unit considers the state of all robots in the network and provides appropriate control actions to each agent based on this information, which is a form of strong cooperation. This strong cooperation is crucial in many multi-agent algorithms, including the multi-agent area coverage control algorithms that will be used to evaluate the proposed hardware test platform de-

sign.

In terms of the architecture of the test platform, the hybrid approach introduced in Section 2.1.2 is the most versatile, allowing for a variety of multi-agent scenarios to be tested. This means that while the test platform will have a central control point, the robots of the test platform will also have some autonomy of their own. This means that some control actions will be computed locally by the agents themselves, while other actions will be provided by the central control point. In the current state of the proposed design, the test platform has hybrid architecture that has a strong centralized component and a minor decentralized component. The main control actions are computed at the central processing unit and are sent to each robot in the network. The robots are able to make minor obstacle avoidance decisions at the local level which introduces a decentralized component to the test platform (Section 3.5.5). Research is being dedicated to develop the decentralized component of the test platform by making use of inter-agent communication capabilities, migrating some computation to the local level of the robots and by improving environmental sensing capabilities of the robots at the local level.

As visualized in Fig 3.2, the structure of the proposed design aims to provide a strongly cooperative and hybrid test platform (with a major centralized component). The test platform consists of a rectangular environment that measures 3.2 by 2.4 meters and is populated by five circular robotic agents which have a diameter of 0.11 meters [47]. The environment itself is covered fully by an overhead RGB camera that collects visual information of the environment. This information is then fed directly to a central processing unit, where various processing techniques are applied to the data based on the specific multi-agent algorithms being tested. This central processing unit also processes data for the purpose of extracting localization information of the robotic agents in the environment. The relevant control actions are then sent from the central processing unit to each respective robotic agent.

At the local level, the agents have ability to communicate directly with other agents in the environment. They also have various environmental sensing capabilities in order to navigate their environment, such as proximity sensors. Finally, the robots are supplemented with additional gyroscopic sensors and accelerometers for improved localization capabilities through the use of dead reckoning techniques.

3.1.2 Inter-agent Communication Protocol

As concluded in 2.3.1, the ZigBee protocol was chosen to be the optimal low power protocol for meshed wireless communication architectures. As such, the XBee protocol which is based on the ZigBee protocol was chosen for creation of the wireless communication system. The XBee protocol is readily available for Arduino based micro-controllers, as found on the Alphabot2, and was therefore the protocol that was chosen. It should be noted that while ZigBee is an open source framework, XBee is not. However, certain XBee modules are built on top of the ZigBee framework, providing the same beneficial characteristics. While the open source nature of ZigBee may be preferable, the XBee modules were chosen for this thesis due to ease of availability and improved compatibility with micro-controllers such as the Arduino Uno.

More details about this protocol are discussed in Section 3.3.3.

3.1.3 Localization

For localization of the robotic agents within the environment, wireless communication based localization techniques did not provide the desired accuracy given the size of the environment (3.2 meters by 2.4 meters). As such, a combination of video scene analysis and dead reckoning techniques are used for localization purposes. The given environment is covered by an RGB camera which sends visual RGB data to the central processing unit. At the central processing unit, machine learning techniques are used for object detection and tracking. Using these techniques, the two dimensional position of the robotic agents can be determined and tracked. Using frame by frame analysis, the heading of the robots can be determined as well.

At the local level of the robotic agents, an MPU6050 (triple axis accelerometer and gyroscope) is used for an attempt at precise navigation of the robotic agents. The data provided by the MPU6050 is used in a dead reckoning fashion to ensure the robotic agents navigate with guidance (straight line motion, rotation).

3.1.4 Environmental Sensing Capabilities

With regard to environmental sensing capabilities, the Alphabot2 comes built in with an ultrasonic sensor and two infrared sensors for proximity measurement and it comes with a line tracking module. Additionally, the Alphabot2 is Arduino

based and the Arduino framework is quite modular, allowing for a variety of additional environmental sensors to be implemented on the robotic agents as required by future applications.

3.2 Alphabot2 Robots in Detail

The Alphabot2 is a relatively small scale robot (diameter of 0.11 meters) with tightly integrated features and a simple process for assembly. It consists mainly of two major components, the lower base component and the upper Arduino interface component. The assembly process requires the attachment of these two components together using four nuts and bolts. The diagrams below label the components of the Alphabot2 frame [47].

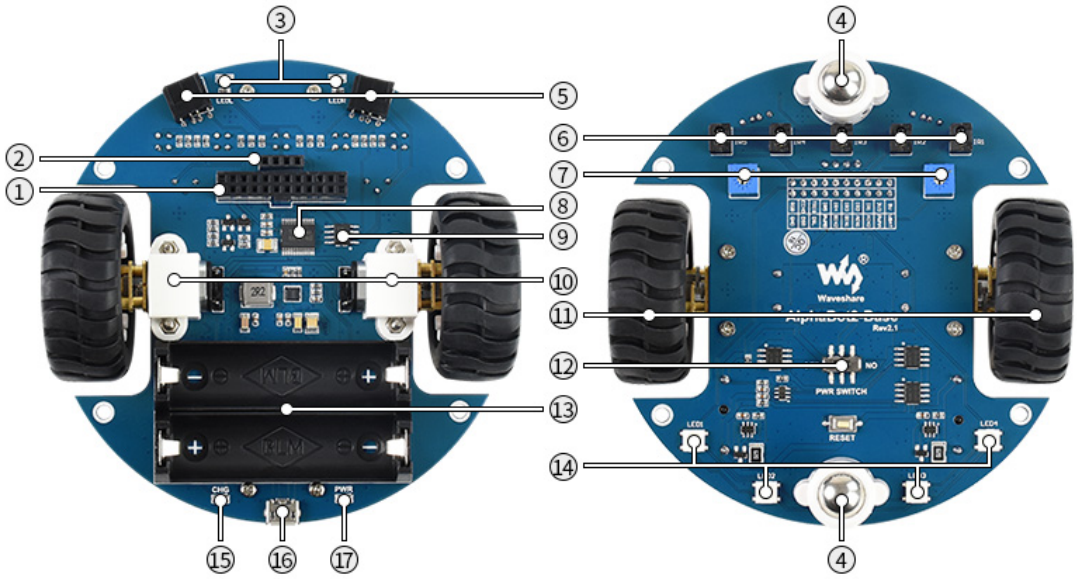


Figure 3.4: Alphabot2 base component [47]. Labels in Table 3.1

Table 3.1 labels all the components found in Figure 3.4. While Table 3.2 labels all the components found in Figure 3.5.

Label	Base Component
1	AlphaBot2 control interface
2	Ultrasonic module interface
3	Obstacle avoiding indicators
4	Omni-direction wheel
5	ST188: reflective infrared photoelectric sensor
6	ITR20001/T: reflective infrared photoelectric sensor
7	Potentiometer
8	TB6612FNG dual H-bridge motor driver
9	LM393 voltage comparator
10	N20 micro gear motor reduction rate 1:30, 6V/600RPM
11	Rubber wheels diameter 42mm, width 19mm
12	Power switch
13	Battery holder
14	WS2812B: true-color RGB LEDs
15	Battery charging indicator
16	5V USB battery charging port
17	Power indicator

Table 3.1: Sub-components of the Base Component of the Alphabot2 [47]

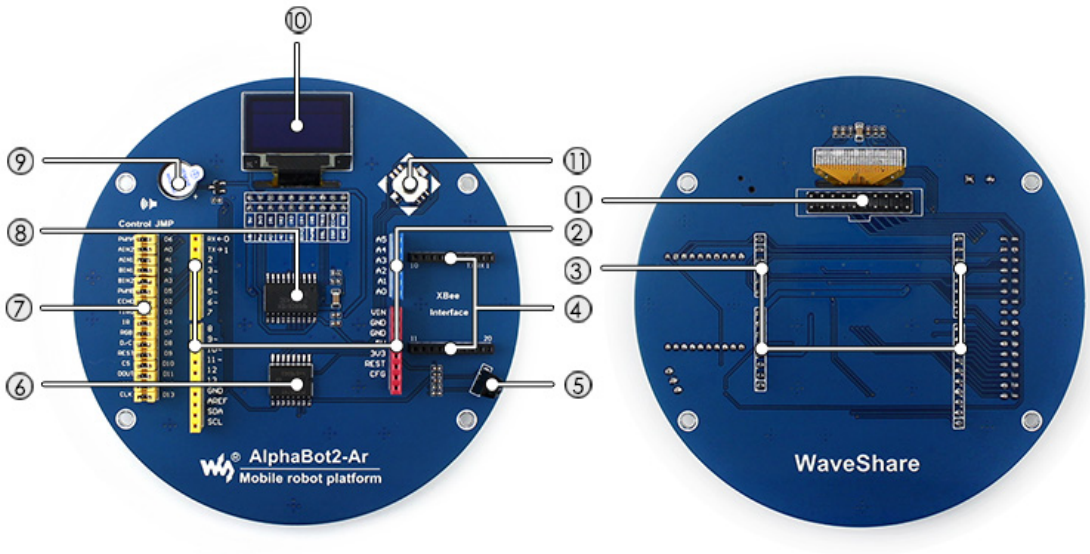


Figure 3.5: Alphabot2 top component with Arduino interface [47]. Labels in Table 3.2

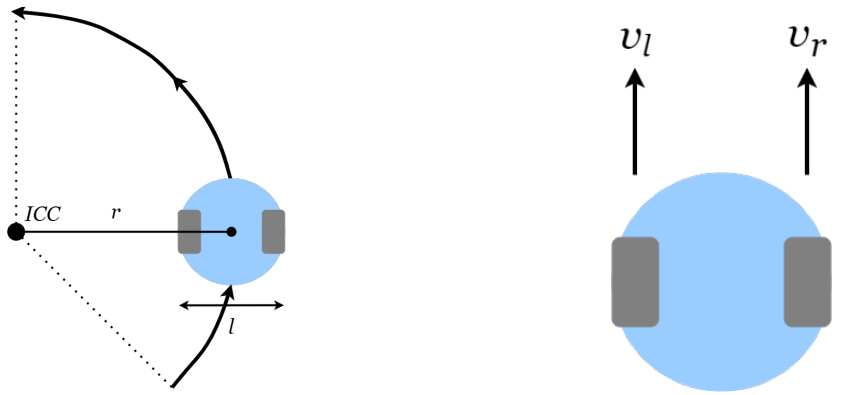
Label	Top Component
1	AlphaBot2 control interface
2	Arduino expansion header
3	Arduino interface
4	Xbee connector
5	IR receiver
6	PC8574: I/O expander, SPI interface
7	Arduino peripheral jumpers
8	TLC1543: 10-bit AD acquisition chip
9	Buzzer
10	0.96 inch OLED SSD1306 driver, 128x64 resolution
11	Joystick

Table 3.2: Sub-components of the Top Component of the Alphabot2 [47]

3.2.1 Differential Drive

The Alphabot2 is a differential drive robot with two powered wheels which lie on a collinear axis. The Alphabot2 also has two passive wheels which keep it from tipping forwards or backwards. In a basic sense this type of robot is capable of three distinct types of motion. When both wheels spin in the same direction and at the same speed, the robot simply travels in a straight line. Instead, when one wheel spins at a slower speed than the other wheel, the robot will move in an arc that curves towards the direction of the slower wheel. Finally, when the two wheels spin at equal speed but in opposite direction, the robot will simply spin in place. This can be understood more clearly through formal discussion of differential drive kinematics [50, 51].

To begin this discussion, an instantaneous center of curvature is defined, which is the point located at the center of the curved trajectory of a differential drive robot (Fig 3.6a). The term r represents the radius between the instantaneous center of curvature and the center of the body of the robot. The term l represents the distance between the two wheels of the robot. Finally, v_l and v_r refer to the velocities of the left wheel and right wheel respectively.



(a) Spatial parameters (b) Wheel velocities

Figure 3.6: Differential drive kinematics

Given these definitions, the following relationships exist between the velocity of each wheel and the angular velocity of the robot around the instantaneous center of curvature, ω [51].

$$v_r = \omega \left(r + \frac{l}{2} \right) \tag{3.1}$$

$$v_l = \omega \left(r - \frac{l}{2} \right) \tag{3.2}$$

By subtracting Equation 3.2 from Equation 3.1 and simplifying, the following equation is obtained [51]:

$$\omega = \frac{v_r - v_l}{l} \tag{3.3}$$

Alternatively, by adding Equation 3.2 with Equation 3.1 and then substituting Equation 3.3, the following equation is obtained [51]:

$$r = \frac{l(v_r + v_l)}{2(v_r - v_l)} \tag{3.4}$$

From these equations, it can be seen that the angular velocity of the robot is related to the difference in speed of the two wheels. The robot will tend to turn towards the wheel with the slower velocity. Additionally, when $v_r = v_l$, the value for angular velocity becomes zero, indicating that the robot is moving in a straight line. In the third case when $v_r = -v_l$, the value for r becomes zero, indicating that the robot is spinning in place.

3.2.1.1 Robot Pose and the Inverse Kinematics Problem

The position of a robot in a two dimensional planar environment can be defined by global rectangular Cartesian coordinates. Additionally, the orientation of the robot can be defined by θ, the angle between the heading of the robot and the positive x-axis. Together these three parameters define the pose of the robot (x, y, θ) [51]. This is visualized in Fig 3.7.

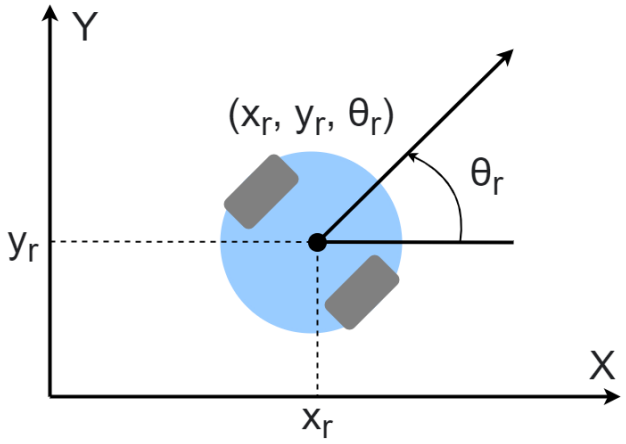


Figure 3.7: Pose of robot at (x_r, y_r, θ_r)

The inverse kinematics problem deals with the situation where given an initial pose (x_1, y_1, θ_1) and a final pose (x_2, y_2, θ_2), the proper values for v_l and v_r must be found to move the robot from the initial pose to the final pose. This problem is important to address for proper implementation of a hardware based multi-agent test platform, as each agent in the test platform will face this problem individually. In a variety of multi-agent algorithms the states of all agents in the environment is known. Additionally the desired future state of all the agents in the environment is typically also known or can be calculated. The solution to the inverse kinematics problem is then what enables these agents to move from their current state to their future desired state using an optimized control sequence.

Without any additional constraints, the inverse kinematics problem in the context of differential drive robotics is over-determined as an infinite variety of different paths can be taken from the initial pose to the final pose. Meaning that an infinite variety of options exist for values for v_l and v_r that would take the robot from the initial pose to the final pose [51].

However in the case of an environment which has no obstacles in it, certain constraints can be applied to make this problem more simple to solve. For example, by using the following constraint where $v_l = v_r$, the robot becomes limited to

straight line motion, allowing it to change its x and y position while maintaining a constant orientation (θ). Similarly, if the following constraint is introduced where $v_l = -v_r$, the robot becomes limited to spinning in place. This allows the robot to change its orientation (θ) while maintaining a constant x and y position [51].

On their own these constraints are limited, however when combined in sequence they can provide a direct path for the robot to go from the initial pose to the final pose. Initially, the second constraint can be applied to orient the robot towards the desired x and y coordinate. Next, the first constraint can be applied to move the robot from its initial x and y position straight to its final x and y position. Finally, the second constraint can be applied once more to change the orientation of the robot to match the desired final pose. This is a simple yet effective method for solving the inverse kinematics problem for a differential drive robot, and works in environments where no obstacles are present [51].

3.3 Additional Hardware Components Used

This section describes some additional hardware components that were used in the test platform apart from just the base Alphabot2 device.

3.3.1 Webcam for Object Detection

The overhead webcam used for covering the environment in the lab setup is a typical webcam (Fig 3.1b) with relatively low frame resolution and low frame rate. The webcam video resolution is 640 by 480 and the frame rate is 6 frames per second. Given intrinsic latencies and computational limitations in the central processing unit, a relatively slow camera with low resolution is sufficient. The field of view of the camera covers an area of 3.20 meters by 2.4 meters. Fig 3.8 shows a sample image of the environment where two Alphabot2 agents can be seen as well.

3.3.2 MPU6050 Inertial Measurement Unit

The MPU6050 is a small 4x4x0.9mm chip that contains a triple axis accelerometer, a triple axis gyroscope and a digital motion processor. The gyroscope provides information about the per second change in orientation of a device in three dimensional space. While the accelerometer provides information about the acceleration of a device in three dimensional space. The accelerometer also provides informa-

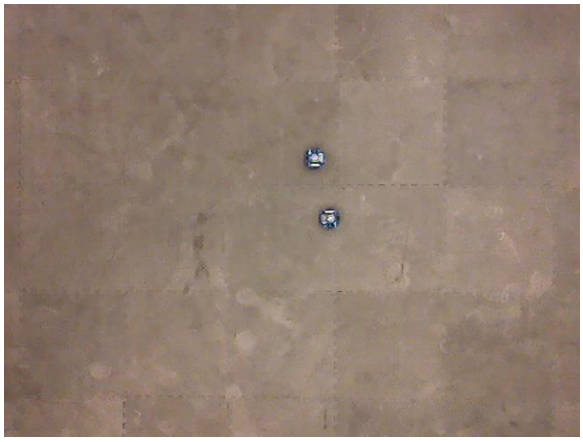


Figure 3.8: Sample image of lab environment with two Alphabot2 agents

tion about the orientation of a device relative to the static gravity vector. The digital motion processor is an onboard processor that collects and combines data from both the gyroscope and the accelerometer as an alternative method of collecting data from the MPU6050 [52, chapter 3].

The combination of these devices into one small package allows for a variety of interesting use cases. The MPU6050 has applications in augmented reality, panoramic photo capture, motion tracking and device localization [52, chapter 3]. The MPU6050 is Arduino compatible and can be found in the form of an Arduino compatible breakout board (Fig 3.9). This means the MPU6050 easily fits into the Alphabot2 system.

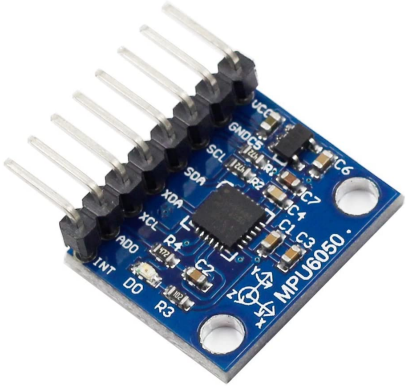


Figure 3.9: MPU6050 breakout board for Arduino

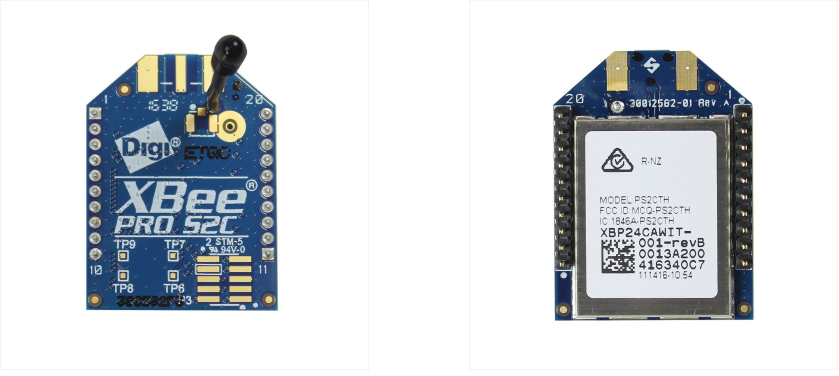
In the context of this thesis, an MPU6050 unit is attached to each Alphabot2 robot in order to keep track of the three dimensional orientation of the robot using dead reckoning. While global localization is achieved using the overhead camera and video scene analysis techniques, the localization information provided by the

MPU6050 is instead used for improved precision of locomotion capabilities. For example, if a robot is given a control action to rotate 45^o clockwise, it can use data from the MPU6050 to obtain its current heading and can actuate accordingly until a 45^o clockwise rotation is obtained. Similarly, if a robot needs to travel straight, it can use MPU6050 data to obtain its current heading and it can then actuate appropriately to ensure that the desired heading is maintained. This is discussed and implemented in more detail in Section 3.5.

3.3.3 XBee Module

The XBee module is provided by Digi International. The term XBee simply refers to a large collection of modules that are provided by Digi International. While certain XBee modules make use of alternative protocols, a large portion of these devices are built on top of the ZigBee protocol which is a low power and cost effective wireless communication protocol that is suitable for this project. Specifically, XBee modules used in the proposed design for this thesis are built on top of the 802.15.4/ZigBee protocol.

The benefit of using the XBee modules provided by Digi International is that they come fully equipped in a standardized form factor. This means they are compatible with popular micro-controllers such as the Arduino micro-controller. Digi International also provides free software that can be used for quick setup and easy implementation of these low power wireless communication modules [53]. Python development libraries also exist for easy implementation of mesh networks with XBee modules [54].

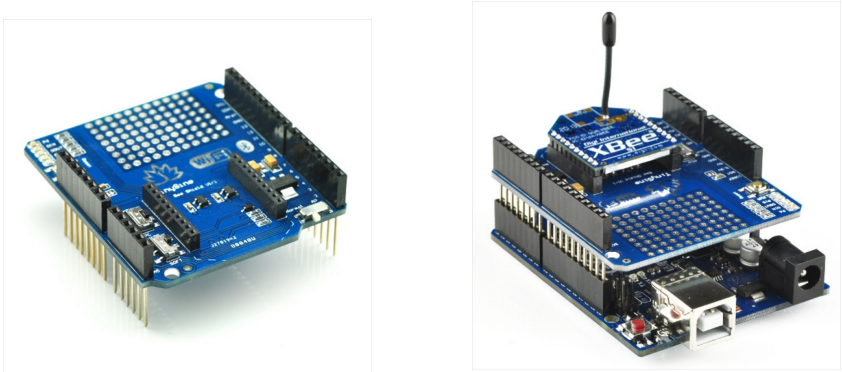


(a) Top view of module (b) Bottom view of module

Figure 3.10: XBee Series 2C Pro Module

Digi International provides a variety of 802.15.4/ZigBee based devices to choose

from. The main distinction being the series 1, series 2 and series 3 modules. Series 1 modules are otherwise useful, however are limited to the 802.15.4 protocol, meaning that they can form point to point networks but are not capable of creating ZigBee mesh networks. Series 2 modules instead are capable of making use of the ZigBee mesh protocol, meeting the use case requirements for this project. Series 3 modules provide some improved performance in terms of range and reductions in power consumption, however typically come at higher cost [55]. Series 3 modules also add additional functionality including MicroPython, Digi TrustFence Security and an integrated Bluetooth Low-Energy framework. However, the series 2 modules are a good balance between performance, functionality and cost. For this reason the series 2C Pro XBee modules are used in this thesis. Fig 3.10 shows a top and bottom view of these modules. The series 2C Pro modules provide a 2.4 GHz transceiver, a serial packet data transfer rate of 1 Mbps and an indoor communication range of 90 meters [56, pages 17 - 18][57].

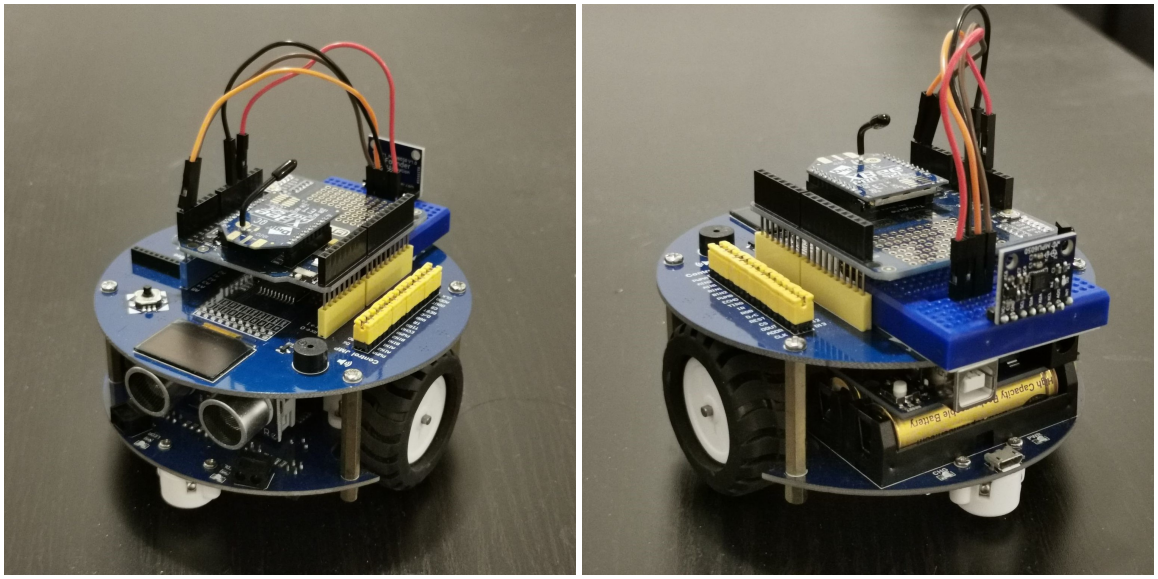


(a) XBee module shield

(b) Attachment of Arduino Uno, XBee shield and XBee module

Figure 3.11: XBee module shield for Arduino

Connecting an XBee module to an Arduino Uno micro-controller requires the use of a specific Arduino shield (Fig 3.11a). The attachment of the Arduino Uno, the XBee shield and the XBee module can be visualized in Fig 3.11b. Each Alphabot2 is fit with an XBee module shield and an XBee module, creating a wireless mesh network between all five agents in the test platform. This complete assembly of the Alphabot2, Arduino Uno, XBee shield, XBee module and MPU6050 module can be visualized in Fig 3.12.



(a) Front of complete assembly (b) Back of complete assembly

Figure 3.12: Complete assembly of Alphabot2 with XBee shield, XBee module and MPU6050

3.4 XBee Module Configuration

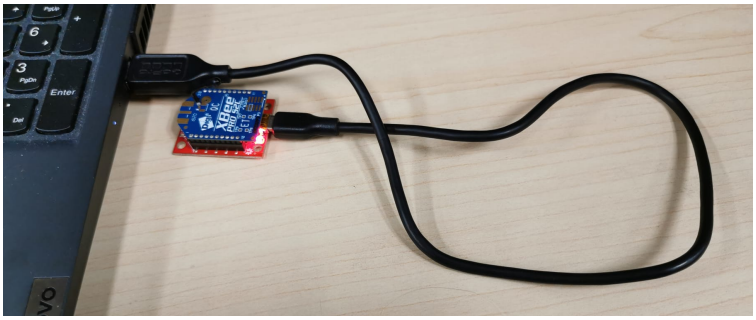


Figure 3.13: XBee module (blue) connected to a central processing unit using the XBee explorer module (red)

An XBee explorer module allows an XBee module to be connected to a laptop or desktop computer (Fig 3.13). On the laptop or desktop, XBee modules are configured using the free XCTU software offered by Digi International. In XCTU, certain parameters of the XBee can be configured to form a mesh network between a group of XBee modules. These parameters include the role, 64 bit address, channel, personal area network (PAN) ID, interface data rate and the operating mode of an XBee module.

- **Role:** Within a ZigBee mesh, each node of the network is assigned one of

three roles. Each network typically has a single coordinator, which initializes the network and allows other XBee devices to join the network. The network also consists of routers, which can send, receive and route data to other nodes in the network. Finally, nodes can also act as end devices which have more limited functionality and are able to send or receive data but are not able to route it. The lab setup for this thesis can be visualized in Fig 3.14 and consists of a single coordinator which is connected to the central processing unit, and five routers which are connected to their respective Alphabot2 base [58, page 35].

- **64 bit address:** Each XBee module has a unique 64 bit address which is used as an identifier for sending and receiving messages. This parameter cannot be modified in the XCTU interface and comes built in with an XBee module [58, page 54].

- **Channel:** In order to form a mesh network, all XBee modules in the network must be configured to share the same channel [58, page 36]. For this thesis, they were all configured to channel "C".

- **PAN ID:** Additional communication groups can be created by using the PAN ID. Within a single channel, XBee modules must share the same PAN ID in order to communicate with each other. In the case of this thesis as all agents must be able to communicate with each other, all XBee modules were configured to share the same channel and PAN ID [58, page 36]. For this thesis, they were all configured to channel "C" with a PAN ID of "3020".

- **Interface data rate:** This rate represents the rate of serial data transfer between XBee modules. In a single network, all modules must be configured to the same interface data rate to allow for communication. The recommended rate of 9600 bits/second was used [58, page 163].

- **Operating mode:** The default operating mode of an XBee module is transparent mode or AT mode. In this mode a message sent from a transmitting XBee to a receiving Xbee is sent as is, without any additional information attached. When in AT mode, the transmitting XBee must be reconfigured every time it needs to send a message to a new receiving device. As such, AT mode is useful in point to point communication between two devices, or when a transmitting device has a static destination address. However, AT mode has

limitations when a transmitter needs to transmit to a variety of destination addresses in a more complex mesh arrangement [58, pages 27 - 29].

This is where the application programming interface (API) mode is used. API mode packages can be more difficult to use, however this mode comes with additional flexibility. In API mode, transmitting devices can send structured packages of data that contain the original message being sent along with additional information. API packages can contain the source address of each message and also can be used to calculate received signal strength. When sending an API package, transmitting devices can also receive a success or failure indicator back, depending on if the message was delivered to the intended receiving device or not. Finally, in API mode a transmitting device can send messages to multiple destination addresses which is a core requirement of the mesh architecture seen in Fig 3.14. In the case of this thesis, API mode was used for the coordinator XBee module which is attached to the central processing unit as it needs to send messages to multiple destination devices [58, pages 27 - 29]. Router modules were kept on AT mode as for the scope of this thesis they were only used for receiving data and did not need to send information at any point. With on going work dedicated to a more decentralized architecture, API mode can enabled on the router modules as well to enable inter-agent communication capabilities and allow for decentralization of the test platform.

3.5 Arduino Software Implementation

The Alphabot2 is compatible with the Arduino micro-controller, and therefore an Arduino Uno is used as the main control board for all the Alphabot2 robots in the testing platform. The Arduino IDE is used for programming all the local logic onboard the Arduino Uno controller.

3.5.1 Receiving Data

Wireless communication takes place through the use of the XBee module attached to the Arduino. Each local agent is equipped with an XBee module that receives control instructions in the form of serial data packets that are sent to the serial line of the Arduino Uno. These data packets are sent from the central XBee module that

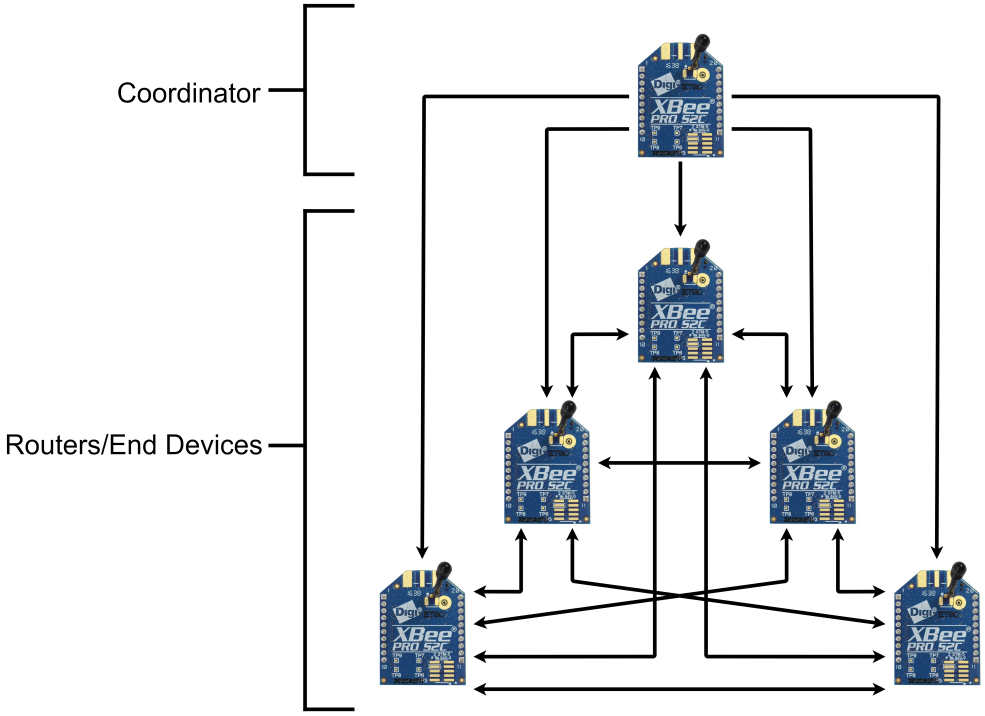


Figure 3.14: XBee mesh architecture

is connected to the central processing unit. Packets are programmed to be sent in the following format:

$$< b_1, b_2, b_3, ..., b_n > \tag{3.5}$$

where $<$ and $>$ indicate the start and end delimiters of the packet respectively, n represents the total number of bytes in the packet and $b_1 \rightarrow b_n$ represents the sequence of bytes in the packet.

When XBee modules receive a serial packet, they read it one byte at a time, which is why this structure of packet is used. The use of the $<$ and $>$ symbols as delimiters allows for the separation of the core message from additional information that is attached to the message such as received signal strength or source address data in the case of API data packets. When the $<$ symbol is read by the Arduino, it begins to store the following bytes of data. When $>$ is read, the Arduino stops storing additional bytes and concatenates the bytes that it has read into a full message.

3.5.1.1 Control Space

After the incoming bytes of the core message are stored and concatenated, the received message is processed and the received control actions are carried out.

As discussed in Section 3.2.1, by applying constraints to the velocity space of the Alphabot2, a simple solution can be found for the inverse kinematics problem for a differential drive robot. This solution only requires three main forms of locomotion from the robotic agent. The robot must be able to move in a straight line (forwards or backwards). This is achieved by spinning both wheels in the same direction at the same speed. The robot must be able to rotate in place to change its orientation. This is achieved by spinning both wheels at the same speed but in an opposite direction. Finally, the robot needs to be able to stop and stand still when it arrives at the desired location, which is achieved by removing power from both wheels.

While theoretically this control space is relatively simple to achieve, experimental implementation is not as trivial. Due to factors such as friction between the wheels and the ground, variances in voltage supply to each wheel motor or battery levels on the robot, exact straight line motion or precise rotation can be difficult to achieve. This is why an inertial measurement unit, more specifically the MPU6050, is used in an attempt to achieve precise localization for locomotion.

3.5.2 Inertial Measurement Unit

The MPU6050 is capable of providing information related to precise changes in orientation (yaw, pitch, roll) of the Alphabot2. This information is used to then ensure that the Alphabot2 is moving in a straight line (yaw remains constant) or is rotating a precise amount (yaw changes to a desired value). This section discusses how MPU6050 data can be collected and processed to obtain accurate measurements of orientation.

The MPU6050 comes packaged with a three axis accelerometer and three axis gyroscope. Both the accelerometer and gyroscope make use of micro electro-mechanical systems.

3.5.2.1 Capacitive MEMS Gyroscope

The use case for this thesis simply requires the measurement of the yaw of the robot. This can be achieved by the gyroscopic capabilities of the MPU6050 alone. The capacitive micro electro-mechanical system gyroscope makes use of Coriolis accelerations to obtain angular velocity measurements [59]. The general mechanical structure used in these types of gyroscopes resembles the structure seen in Fig 3.15.

The structure consists of a resonating mass that is attached to springs. This

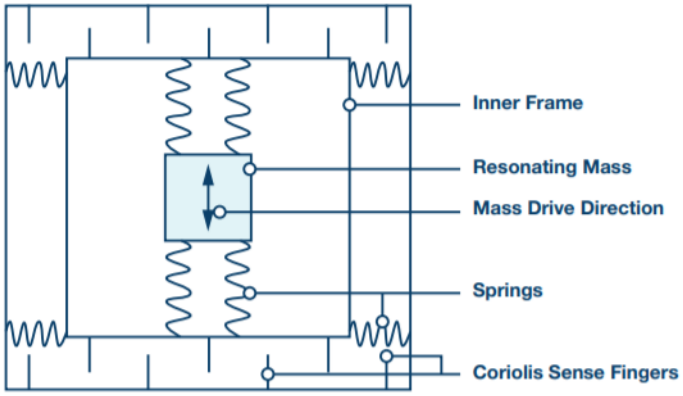


Figure 3.15: Mechanical structure designed to relate Coriolis acceleration to angular velocity [59]

mass is actuated to oscillate in the directions indicated by the mass drive direction arrows. This inner mass structure is attached to an inner frame. The inner frame itself is attached to an outer frame by springs that allow motion perpendicular to the direction indicated by the mass drive direction arrows. As the inner frame oscillates side to side, the distance between the electrodes on the inner frame and the Coriolis sense fingers on the outer frame changes, causing variance in capacitance between the two structures [59].

Fig 3.16 shows a clockwise rotating disc with this mechanical structure embedded inside of it. As the resonating mass oscillates along the direction of the mass drive actuation, the resulting Coriolis acceleration from the rotation of the disc causes the entire inner frame to oscillate side to side. The magnitude of this side to side oscillation motion is measured by the change in capacitance between the outer electrodes and the fixed Coriolis sensing fingers. This variance in capacitance is used to determine the corresponding angular velocity of the disc (which is what initially caused the oscillation of the inner frame) [59]. A triple axis gyroscope has three of these described structures, to measure angular velocity for each axis of rotation separately.

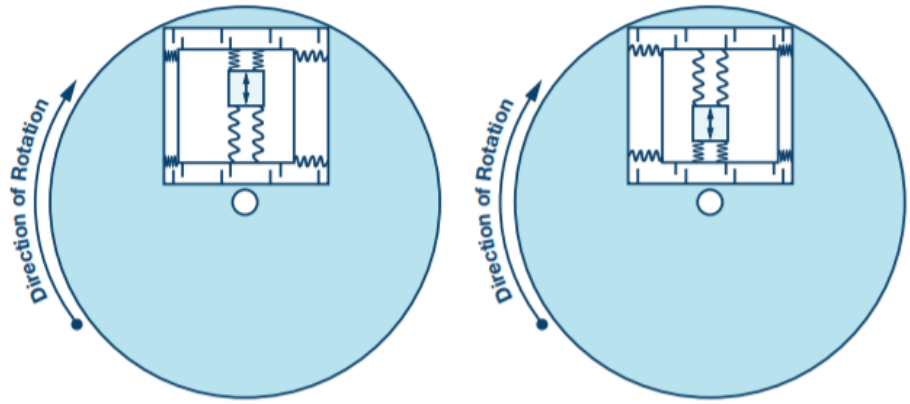


Figure 3.16: Mechanical structure designed to relate coriolis acceleration to angular velocity [59]

3.5.2.2 MPU6050 Data Collection

When connecting the MPU6050 to the Arduino Uno, the circuit seen in Fig 3.17 is used.

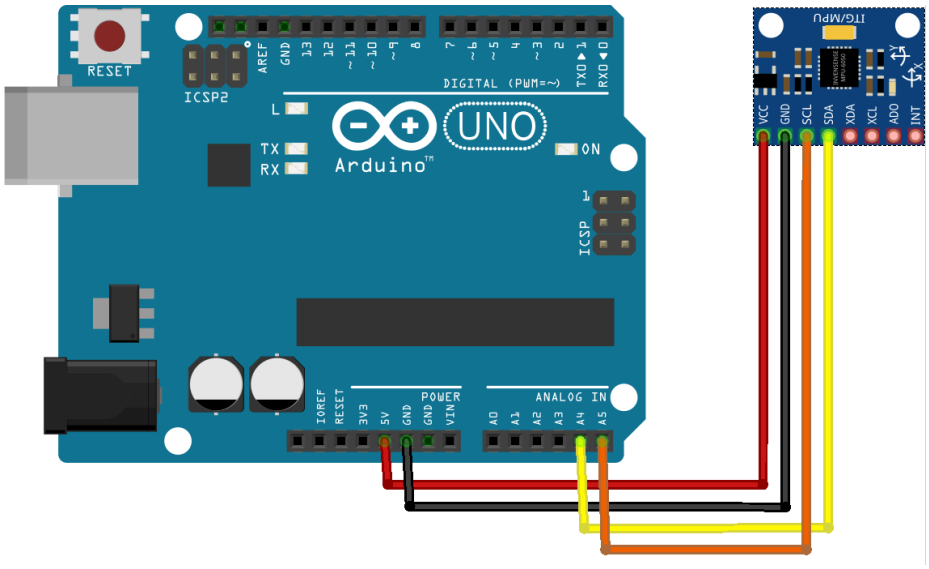


Figure 3.17: Circuit diagram for MPU6050 and the Arduino UNO

Communication between the Arduino Uno and the MPU6050 takes place using the I^2C protocol which requires the use of the Arduino wire library. In the setup function of the Arduino code, the MPU6050 registers are reset and the MPU6050 is configured to run at its default sensitivity values. For the gyroscope, this means a sensitivity of $+/-250$ degrees per second. The gyroscope can also be configured

to sensitivity levels of $+/-500$, $+/-1000$ and $+/-2000$ degrees per second [52, chapter 5].

Within the setup function of the Arduino code, gyroscopic error measurements are also calculated. For this process the Alphabot2 and the attached MPU6050 are kept stationary. The Arduino then collects 200 unique measurements from the MPU6050 gyroscope. Since the Alphabot2 is held stationary, the expected value for the angular velocity is zero for all 200 measurements. Any deviation from this expected value of zero is considered a measuring error. These errors are summed up and divided by 200 to obtain the average gyroscopic measurement error. This error value is then accounted for in any future measurements of angular velocity.

The process of collecting data from the MPU6050 involves accessing specific data registers on the MPU6050. According to the official MPU6050 register data sheet, gyroscopic data can be found on registers 0x43 - 0x48. The gyroscopic data for each axis of rotation comes in the form of twos complement 16-bit values. Registers 0x43 and 0x44 provide the most recent measurements for angular velocity around the x-axis. Similarly, registers 0x45 and 0x46 hold measurements for angular velocity around the y-axis. Finally, registers 0x47 and 0x48 hold measurements for angular velocity around the z-axis. These registers can simply be accessed by initiating communication between the Arduino and register 0x43 on the MPU6050. From there, data from the six registers mentioned above can be requested [60, chapter 4].

When using a particular sensitivity for the gyroscope, $+/-250$ degrees per second in our case, the value output by the gyroscope registers must be multiplied by the corresponding sensitivity scale factor. The MPU6050 data sheet indicates that for a gyroscopic sensitivity of $+/-250$, the sensitivity scale factor is 131 $LSB/^o/s$. This value can be calculated by dividing the maximum value of a twos complement 16-bit, which is 32767, by the sensitivity value of $+/-250$ [52, chapter 6].

The angular position of the Alphabot2 agents can be obtained by integrating over the angular velocity values that are output by the MPU6050 gyroscope. In terms of Arduino code this is achieved by multiplying the measured angular velocity in a given direction, by the time elapsed between the previous measurement and the current measurement.

The MPU6050 is aligned to the Alphabot2 such that the yaw of the Alphabot2 is represented by rotation around the x-axis of the MPU6050. As such the x-axis gyroscopic data is used to determine the yaw of the Alphabot2 agent at any given

time.

3.5.3 Straight Line Motion

In order to move straight, the robot simply needs to maintain a constant heading for the duration of the straight line motion. Heading information from the MPU6050 allows the robot to recognize when it is no longer moving straight. If the heading of the robot moves to the left of the desired heading, the left wheel speeds up relative to the right wheel in order to correct the heading. Similarly, if the heading of the robot moves to the right of the desired heading, the right wheel speeds up relative to the left wheel in order to correct the heading.

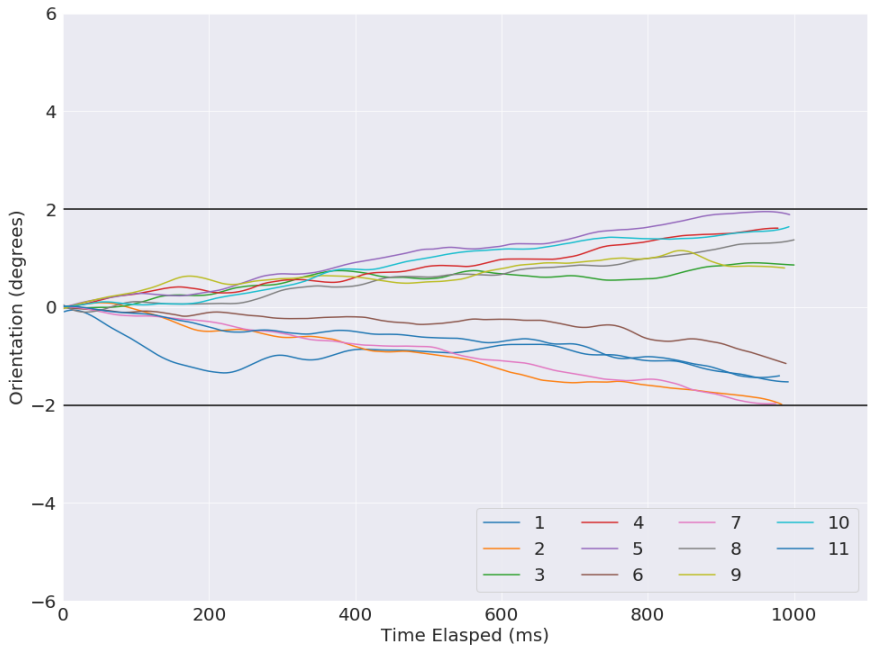


Figure 3.18: Sample responses from the Alphabot2 system with a PD controller for straight motion over 1000 milliseconds ($k_p = 0.05$, $k_d = 10$)

This control action is achieved through the use of a PD controller, where the error term is defined as the difference between the current heading and the desired heading. An integral gain is not introduced as in practical implementation, the discrete forward motion of the robot happens so quickly that the integral term was found to provide no benefit towards stability. Through visual trial and error, the heuristically tuned proportional gain was found to be 0.05. Values higher than

this would result in the robot oscillating from side to side as it moved forwards. Values lower than this would limit the robots ability to correct its heading. A higher differential gain of 10 was used. This higher differential gain ensured that any deviation from the desired heading was quickly corrected.

Fig 3.18 shows 11 sample responses from this system with the PD controller being implemented. It can be seen that over the 1000 milliseconds that the robot travels forward, the heading of the robot remains relatively constant, only swaying by $\pm$ 2 degrees. This was found to be sufficient for the purpose of the thesis, but not optimal as discussed in Chapter 4 and future work will be dedicated to improving these results.

3.5.4 Rotation

An attempt at rotational motion of the agent is achieved by obtaining the current heading of the robot and then actuating the robot in the desired direction until the desired heading is obtained. An example of this process is visualized in Fig 3.19. Assuming the robot measures a heading of $+90^o$ with respect to the positive x-axis, and the desired heading is $+180^o$ with respect to the positive x-axis. The velocity vectors of the two wheels would have opposite direction and equal magnitude to allow the robot to spin in place to the desired heading.

A PID controller is used to determine the appropriate actuation values required to stabilize the robot at the desired heading given the measured heading as a starting point. The PID control gains used were $k_p = 0.024$, $k_i = 0.002$ and $k_d = 10$. Where the error was modelled to be the difference between the current heading of the robot and the desired heading of the robot.

For initial tuning of the PID controller, the second method of the Ziegler-Nichols tuning rules was used [61, section 8.2]. When using this method for a PID controller, all gains are first set to zero. The proportional gain (k_p) is then increased until the system response exhibits sustained oscillations. After trial and error, this k_{cr} value was found to be 0.04 for the Alphabot2 system. The period of these sustained oscillations is defined by p_{cr} and this period was found to be 1000 milliseconds.

Based on these values for k_{cr} and p_{cr}, the following formulas are used to calcu-

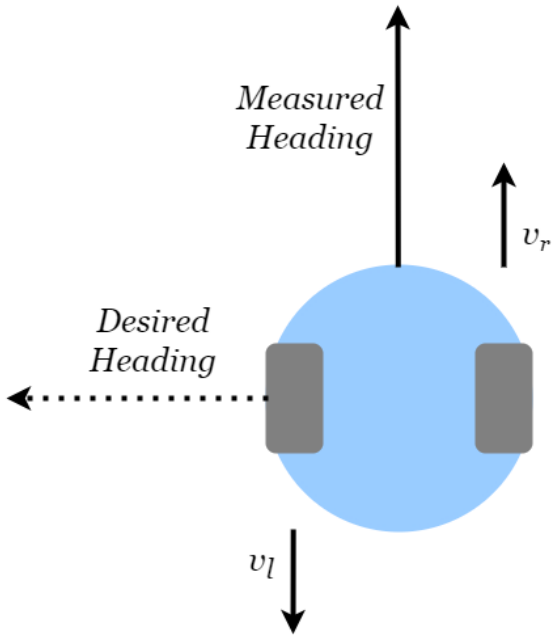


Figure 3.19: Actuation required to rotate the robot in the counter-clockwise direction

late the k_p, k_i and k_d values as per Ziegler-Nichols [61, section 8.2]:

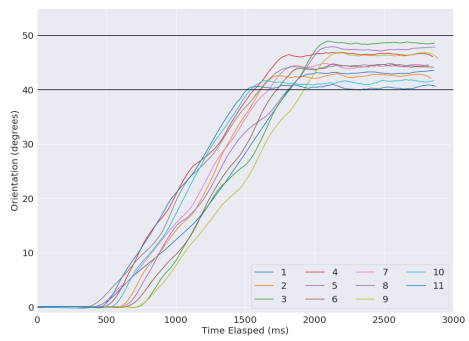
$$k_p = 0.6k_{cr} = 0.024 \tag{3.6}$$

$$k_i = \frac{1}{0.5p_{cr}} = 0.002 \tag{3.7}$$

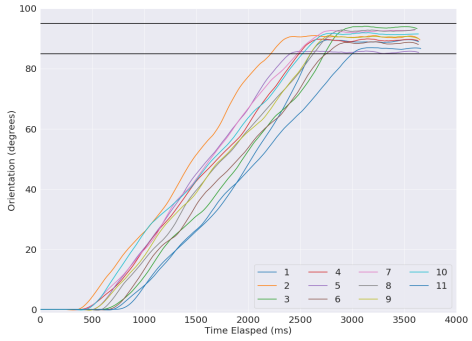
$$k_d = \frac{1}{0.125p_{cr}} = 0.008 \tag{3.8}$$

Implementing these gain values resulted in a stable response by the Alphabot2, however it was found that the system response would exhibit undesirable overshoot. The system would also oscillate around the desired value, resulting in longer settling times. Both of these issues were addressed by increasing the differential gain of the controller until a satisfactory response was obtained. The heuristically tuned differential gain was found to be 10.

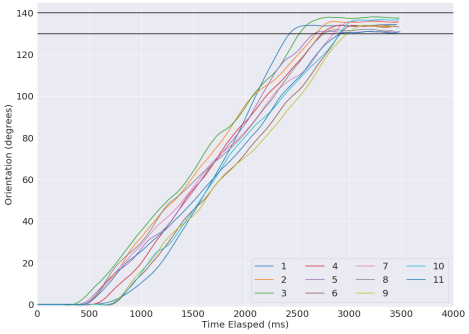
Fig 3.20 shows a plot of sample responses from this system with the PID controller being implemented. These sample responses represent cases involving varying magnitudes of rotation. In all four cases the system is able to get relatively close to the target orientation. In general the system was able to exhibit rotation accuracy within a range of ± 5 degrees. This was found to be sufficient for the purpose of the thesis, but not optimal as discussed in Chapter 4 and future work will be



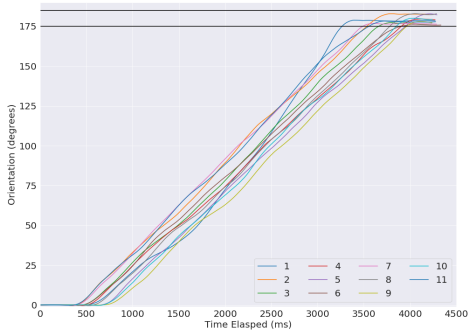
(a) Eleven trials of 45 degree rotation



(b) Eleven trials of 90 degree rotation



(c) Eleven trials of 135 degree rotation



(d) Eleven trials of 180 degree rotation

Figure 3.20: Sample responses from the Alphabot2 system with a PID controller for various magnitudes of rotation ($k_p = 0.024$, $k_i = 0.002$, $k_d = 10$)

dedicated to improving these results. Each of these plots show system responses from 11 trials. It can be noted that these trials have varying rise times and temporal inconsistency. This inconsistency can be due to the fact that the system responses were collected from five different Alphabot2 agents, and minor differences between these agents with respect to their electrical and mechanical components are hypothesized to be the cause for the temporal inconsistency.

3.5.5 Obstacle Avoidance

In order to avoid inter-agent collisions, the built in ultrasonic sensor (Fig 3.21) on the Alphabot2 robots is used. This sensor allows for proximity detection and distance estimation to external obstacles (other Alphabot2 robots). The ultrasonic sensor has a trigger pin which is activated to emit an ultrasonic energy signal. The ultrasonic sensor also has a corresponding echo pin which is used to collect

the reflected ultrasonic signal in the event that the signal hits an obstacle and is reflected back.

Figure 3.21: Ultrasonic sensor used on Alphabot2

Once the echo pin collects the reflected signal, it will output the duration of time that was required for the signal to be sent out, reflected back and recorded. This is referred to as the time of flight (ToF) of the signal. Knowing that the ultrasonic signal travels at the speed of sound (0.034cm / μs), the distance to the obstacle off which the signal was reflected can be calculated. The following formula is used:

$$\text{Distance} = ToF \times \frac{0.034cm/\mu s}{2} \approx ToF \times \frac{1cm}{59\mu s} \tag{3.9}$$

If the Alphabot2 detects an obstacle (or another Alphabot2) in its path within a distance of 8cm, it is programmed to stop and wait until the obstacle can no longer be detected within that range and will then continue moving forward. This ensures that no collisions occur between any of the Alphabot2 robots and the entire multi-agent test platform works efficiently.

3.6 Python Software Implementation

A large portion of the software implementation of this thesis is written in the Python programming language. Common Python libraries such as Numpy and OpenCV are used throughout the code for processing of data and visualization of images. The TensorFlow library is used throughout the implementation for machine learning related tasks, primarily object detection and tracking. The TensorFlow Object Detection API is also used for access to pretrained image classification and object detection models [62, 63]. The Python XBee library is used for generating and transmitting serial data packets using XBee modules.

3.6.1 Machine Learning

As mentioned in Section 3.1.3, a machine learning based object detection approach is used to localize the agents within the covered environment. For this portion of the thesis, the TensorFlow [64] framework is used. This section discusses the theory behind machine learning, object detection, the machine model training process and finally the implementation of the machine learning model.

3.6.1.1 Deep Neural Networks

Deep neural networks are the core machine learning component of this project. A deep neural network is implemented to classify and localize the Alphabot2 agents within the field of view of the overhead camera.

Neural networks in a general sense are capable of recognizing patterns and classifying data points within a given data set. They can come in a range of complexity, and are versatile enough to use on a variety of data set types (visual, audio, numeric). In a basic sense, neural networks are great at learning to connect an input value to an output value, even when the relationship between the input and output might not be clearly defined or known [65, chapter 1].

3.6.1.2 Perceptron and Activation Functions

Having briefly discussed the capabilities of neural networks, and the methods through which they learn, the next important step is to discuss the network architecture. Figure 3.22 shows an image of a perceptron, the most basic unit in any neural network [65, chapter 1].

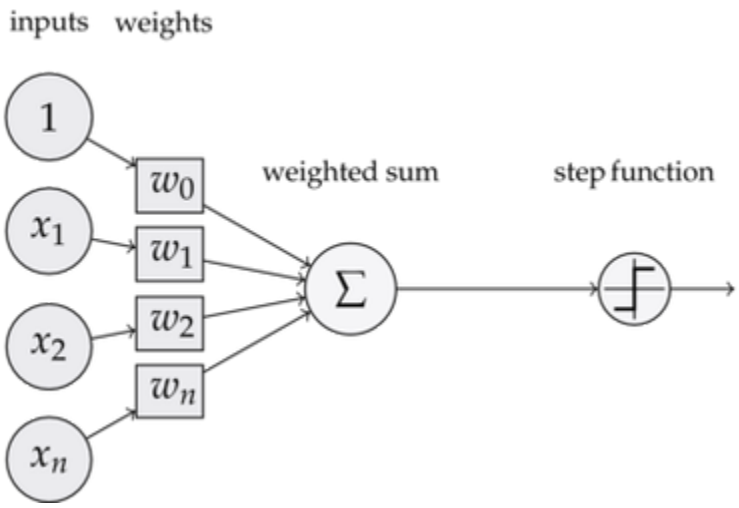


Figure 3.22: Perceptron Architecture [66]

A perceptron is a two layer network, with an input layer of nodes and a single output layer. The input layer consists of n nodes, with an additional bias node, denoted with the number "1" inside. As Figure 3.22 shows, each node is connected to the output node in the subsequent layer with a certain weight value. The feed forward process of the network is that each input value, is multiplied by the weight value of its respective node. The weighted sum of all these values is then fed into the output node [65, chapter 1]. The output node, contains a specific activation function that takes the weighted sum as input, and gives an output value. Some activation functions are discrete, and will fire a zero if their activation threshold is not met, or a value of one if their activation threshold is met by the input value. Some activation functions provide continuous output as a function of the input that they receive. Some common activation functions include the Sigmoid function and the ReLU function. Here x denotes the input to the activation function [65, chapter 1].

Sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.10}$$

ReLU:

$$max(0, x) \tag{3.11}$$

Many different types of activation functions exist, with each one having benefits and drawbacks.

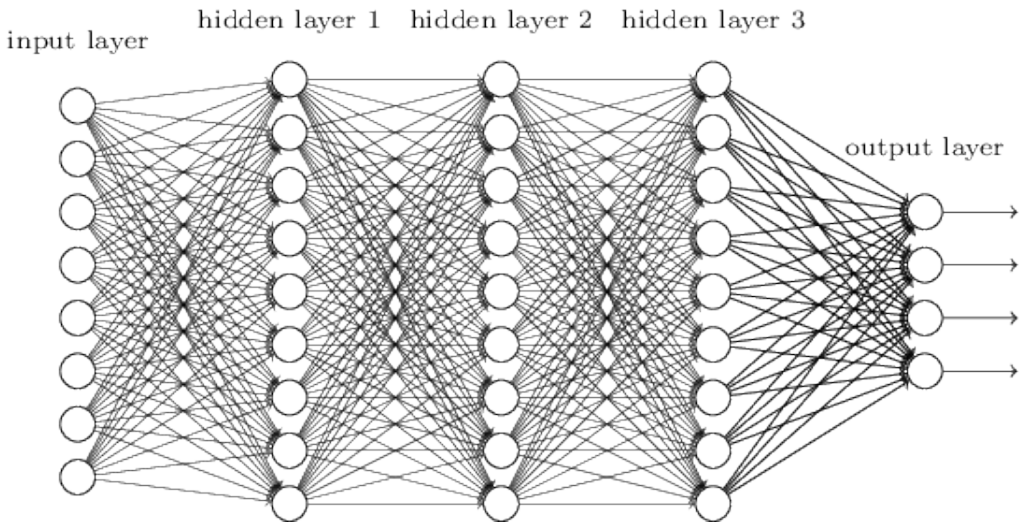


Figure 3.23: Deep Neural Network [67]

3.6.1.3 Hidden Layers

The perceptron itself is relatively simple to understand. These neural network structures can become more complex when additional "hidden layers" are added to the architecture.

Figure 3.23 shows a deep neural network [65, chapter 3], with additional hidden layers on top of the regular input and output layers. It can also be seen that the output layer in this network has multiple nodes, which can be useful for multi-class classification problems. The layers seen here are referred to as "fully connected" layers, in which all nodes in one layer are connected to all nodes in the following layer [65, chapter 3]. As each of these connections has its own weight value, the increase in complexity is evident. Every node in the hidden layers has its own respective activation function as well, which determines how the input values are fed forward to the output of the network.

3.6.1.4 Convolutional Neural Networks and the Convolutional Layer

Convolutional neural networks are another type of neural network. They specialize in dealing with visual data sets and are excellent at extracting important features from images. This project makes use of convolutional neural networks, as the data set used is entirely visual. At the core of all convolutional networks is the convolutional layer [65, chapter 8].

A convolutional layer works by taking an image as an input. In the context of Python, a normal RGB image is simply a three dimensional array. The convolu-

tional layer applies filters to this image, looking for specific feature patterns. This process can be seen in Figure 3.24 [65, chapter 8].

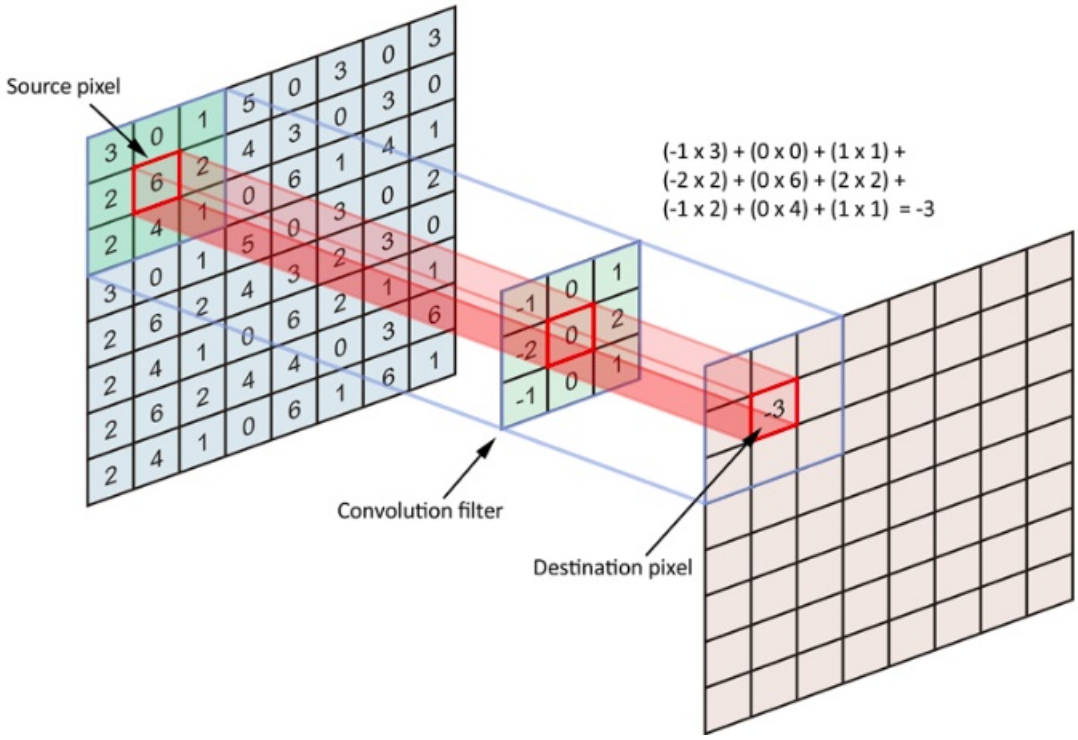


Figure 3.24: Convolutional Layer [68]

The grid on the left side of the image shows a two dimensional array, essentially just an image with a single colour channel (gray scale). The convolutional filter, also referred to as a convolutional kernel, is then applied to this image. The filter is centered on the source pixel. The center of the filter is multiplied by the source pixel, and similarly all other squares in the filter are multiplied by the respective pixel that they are covering. This weighted sum is then fed into the destination pixel in the output of the layer. Due to the nature of the process, it can be seen that if the specific portion of the image under the filter matches the structure of the filter, then the value fed into the destination pixel will be higher [65, chapter 8]. This implies that the desired feature that this filter is looking for is present in the image. In this case the size of the filter is 3 x 3, however this size can be changed as required by the application.

The filter then moves right horizontally, by a defined step size of pixels known as a stride and repeats the process to reconstruct the feature map of the input image. After hitting the extreme horizontal right, it will move down by a defined step size of pixels and repeat the left to right scanning motion. Based on the filter

size and the defined step sizes or strides, the dimensions of the feature map can be much smaller than the dimensions of the initial input image [65, chapter 8].

Each convolutional layer can have multiple filters, that each provide an output when an image is input into the convolutional layer. Each filter in a convolutional layer has a weight value associated with each of its pixels. These weight values update throughout the model training process, based on what features are found to be important for decreasing the loss function of the neural network [65, chapter 8].

3.6.1.5 Object Classification and Feature Extraction

Object classification is the task of classifying the full contents of an image into some predefined labels. Convolutional neural networks are important tools for tackling this task as they are effective at extracting features from input images as shown above. Throughout the training process, various optimization techniques are implemented to optimize feature extraction for a specific task. As an example, a neural network that is designed to classify different species of cats in RGB images will learn to extract different features than a neural network that is designed to classify different car models in RGB images.

Object classification is a sub-problem of the task of object detection and as such a convolutional neural network based architecture is used in this thesis for feature extraction. This architecture is referred to as the MobileNetV2 and its structure can be seen in Fig 3.25.

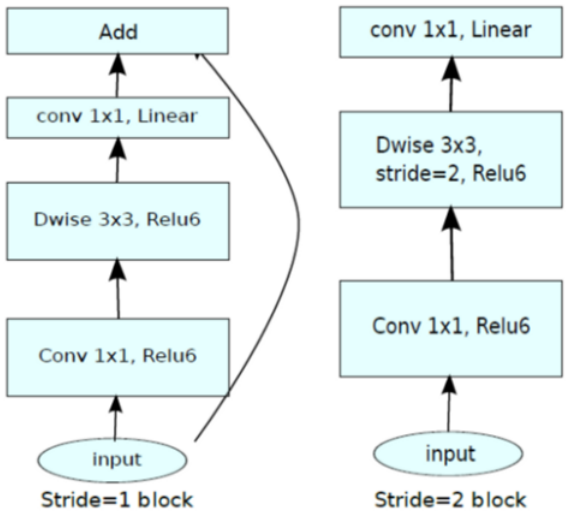


Figure 3.25: MobileNetV2 Architecture [69]

The MobileNetV2 architecture makes adjustments to the typical convolutional neural network structure. These additional features are aimed at making the MobileNetV2 infrastructure more lightweight for faster feature extraction. These features include the introduction of:

- Depthwise Separable Convolutions:
- Inverted Residuals
- Linear Bottlenecks

The detailed impact of these additional features is explained in the original paper introducing the MobileNetV2 architecture [69].

3.6.1.6 Object Detection

While object classification deals with the labelling of the contents of an image, object detection deals with the localization and labelling of the contents within an image using bounding boxes. The comparison between these two techniques is observed in Fig 3.26.

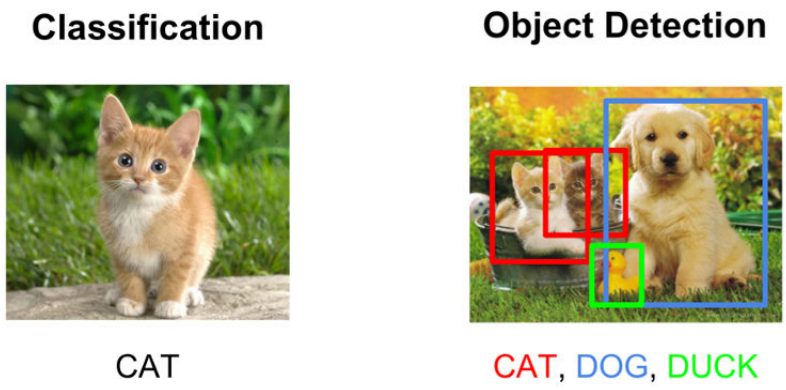
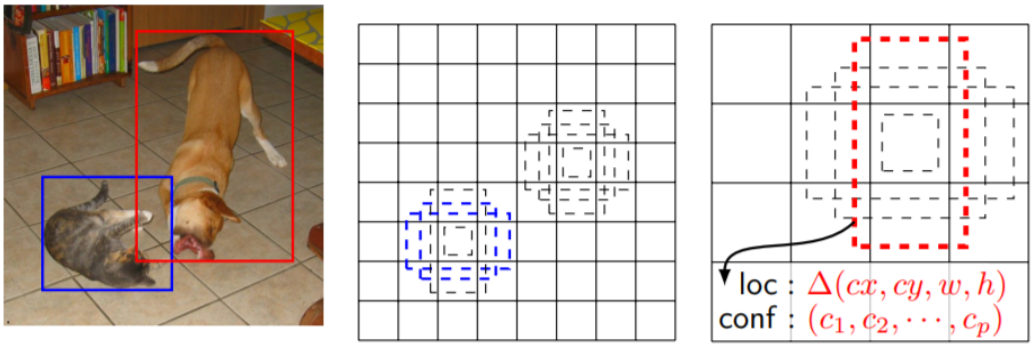


Figure 3.26: Comparing object classification and object detection techniques [70]

An effective and lightweight solution for object detection is the single shot multi-box detector framework. This framework is capable of accurate object detection on just a single frame of the given environment [71]. During the training process, the neural model must be provided sample images of the environment with manually labelled ground truth bounding boxes around the objects of interest. An example of this can be seen in Fig 3.27a.



(a) Input training image with ground truth bounding boxes

(b) Localization of cat in an 8 x 8 feature map

(c) Location of dog in a 4 x 4 feature map

Figure 3.27: SSD workflow [71]

In a basic sense, the technique behind this framework places a variety of default bounding boxes at each location on the image and on extracted feature maps of the original input image. These bounding boxes vary in size and aspect ratio [71]. Small convolutional filters are then applied within each bounding box to classify the contents of the bounding box [71]. Fig 3.27b shows an 8×8 feature map of the original image. The proposed bounding boxes do not provide a match for the dog, however the bounding boxes on the left side are able to detect the cat. As the feature map is further reduced to a lower resolution in Fig 3.27c, the bounding boxes are now able to detect the larger dog. This demonstrates that higher resolution feature maps are used for detecting smaller objects and lower resolution maps are used for detecting larger scale objects in the image [71]. During the training process, the model also learns to associate certain sizes of boxes or certain aspect ratio of boxes with certain classes of objects that require detection.

At each default bounding box, the model provides confidence values to indicate the probability that a certain class of object is located in the respective bounding box. The class of object with the highest confidence value above a predefined threshold is then labelled with the respective default bounding box [71].

The architecture of the original single shot multi-box detector model is shown in Fig 3.28.

The architecture consists of an initial pretrained feature extraction model which extracts feature maps from the input image. In the case of the original single shot multi-box detector model this feature extractor is the VGG-16 convolutional model [71]. In the case of this thesis, the MobileNetV2 architecture is used as the feature extractor instead. The architecture of the single shot multi-box detector is then

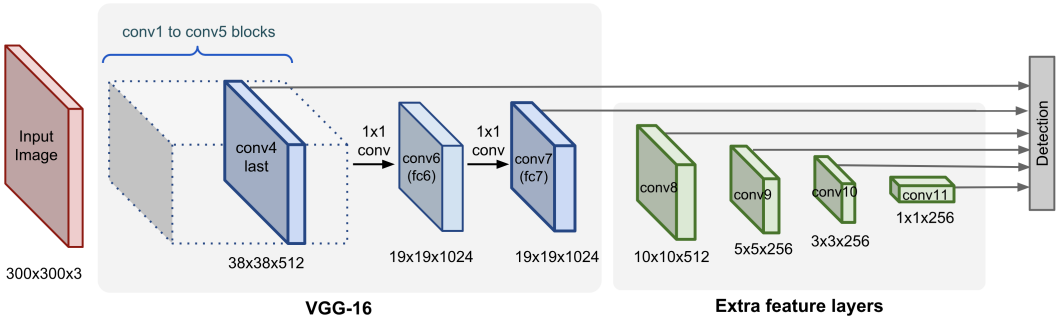


Figure 3.28: Architecture of the original single shot multi-box detector model [72]

followed by a series of extra convolutional layers of varying depth for object classification and localization using bounding boxes [71].

The model used in this thesis is the single shot multi-box detector MobileNetV2 COCO, meaning that it is a combination of the single shot multi-box detector and MobileNetV2 frameworks and it has been trained on the COCO object detection data set already. COCO is an object detection data set that consists of over 200,000 labelled images of over 80 object classes [73].

This pretrained model is robust and comes with state of the art feature extraction and object detection capabilities due to the large COCO data set that it was trained on. For this reason, a technique called transfer learning is used to take advantage of the existing feature extraction and object detection capabilities of the pretrained single shot multi-box detector MobileNetV2 COCO.

This is necessary as the pretrained model is generalized and is not capable of detecting specific objects such as the Alphabot2 agents. With transfer learning, the pretrained model is imported along with all the weight values of its convolutional layers. The training configuration of the model is then adapted and the model is trained further on labelled images of the Alphabot2. This effectively builds on the capabilities of the single shot multi-box detector MobileNetV2 model and allows it to detect the Alphabot2 robot with high accuracy.

3.6.1.7 Transfer Learning for Custom Object Detection

Google Colaboratory: Google colaboratory was used for the transfer learning process. Training object detection models can be computationally intensive and therefore time consuming. However, the time required for training can be significantly reduced by making use of a graphics processing unit to carry out the computations required. The graphics processing unit takes advantage of a paral-

lel computing setup to complete the job more quickly. Google colaboratory offers free graphics processing unit usage and was therefore used as the programming environment for the transfer learning process.

Data set: The task of object detection falls in the realm of supervised learning. In supervised learning, the neural network is trained on a data set that is labelled. It is based on these labels that the neural network is able to learn the task at hand. In the case of object detection for this thesis, the neural network is trained on a data set containing 83 images of the Alphabot2 in the lab environment from the point of view of the overhead camera. The data set can be found at the following link: https://github.com/Sarmyt/masters_object_detection/tree/master/images.

All images contain at least one instance of an Alphabot2, while other images contain two or more instances. The images in this data set were manually labelled using the LabelImg tool [74]. In this case, labelling simply refers to the addition of ground truth bounding boxes around all Alphabot2 robots within a given image. Examples of some images from the data set can be observed in Fig 3.29. All 83 images in the data set are accompanied by a respective XML file which indicates the location of bounding boxes around all Alphabot2 agents in a given image. These XML files can be found at the following link: https://github.com/Sarmyt/masters_object_detection/tree/master/annotations/xmls.

Figure 3.29: Samples from Alphabot2 detection data set

Optimization and Loss: The process of supervised learning makes use of optimization algorithms which enable neural networks to "learn". The training process for a supervised learning task works by providing inputs to the neural model

that is being trained. These inputs then interact with the parameters of the neural model and come out as outputs or predictions. These outputs are then compared with the ground truth labels of the data set. Any deviation between the outputs of the model and the ground truth labels is then modelled as a loss function. Since the output of the model is a direct function of the parameters within that model, the loss function is also a function of the parameters within the neural model [65, chapter 3].

Once this loss function is obtained, an optimization algorithm is used to minimize the loss function with respect to the inherent neural model parameters. The goal being that the next time the model receives similar input, it will provide output that is more accurate to the ground truth.

In the case of object detection with the single shot multi-box detector MobileNetV2 COCO, the ground truth is represented by the bounding boxes around the Alphabot2 agents in the training images. The object detection model attempts to match a variety of default bounding boxes to these ground truth boxes. With respect to calculating loss, a default bounding box is classified as matching a ground truth box if it has a higher jaccard overlap with the ground truth box than all other default bounding boxes. A default bounding box is also considered to match a ground truth box if it has a jaccard overlap of greater than 0.5 [71].

Using the following notation, where if the $i-th$ default box matches the $j-th$ ground truth box for a class of object p then $x_{ij}^{p} = 1$ and if the $i-th$ default box does not match the $j-th$ ground truth box for a class of object p then $x_{ij}^{p} = 0$. The overall loss function for the single shot multi-box detector MobileNetV2 framework is defined as [71]:

$$L(x,c,l,g) = \frac{1}{N}\left(L_{conf}(x,c) + \alpha L_{loc}(x,l,g)\right) \tag{3.12}$$

where N represents the total number of matched default boxes. The localization loss, L_{loc}, is the loss between the parameters of a predicted box (l) and the parameters of a ground truth box (g). This loss is defined as [71]:

$$L_{loc}(x,l,g) = \sum_{\substack{i\in Pos \\ m\in\{c_x,c_y,w,h\}}} x_{ij}^{p}\text{smooth}_{L1}\left(l_i^m - \hat{g}_j^m\right) \tag{3.13}$$

where c_x and c_y represent the x and y coordinates of the center of the box respectively, and w and h represent the width and height of the box respectively. We also

define the following quantities

$$\begin{aligned}
\hat{g}_j^{c_x} &= \frac{g_j^{c_x} - d_i^{c_x}}{d_i^w} \\
\hat{g}_j^{c_y} &= \frac{g_j^{c_y} - d_i^{c_y}}{d_i^h} \\
\hat{g}_j^w &= \log\left(\frac{g_j^w}{d_i^w}\right) \\
\hat{g}_j^h &= \log\left(\frac{g_j^h}{d_i^h}\right)
\end{aligned} \tag{3.14}$$

Where d_i is the $i-th$ default box that is matched to the $j-th$ ground truth box g_j. The smooth_{L1} function in Equation (3.13) is defined as [75]:

$$\text{smooth}_{L1}(x) = \begin{cases} 0.5x^2 & if\ |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases} \tag{3.15}$$

The confidence loss, L_{conf}, is a typical softmax classification loss for confidence scores c for multiple classes. This loss is defined as [71]:

$$L_{conf}(x, c) = -\sum_{i \in Pos}^{N} x_{ij}^p \log\left(\hat{c}_i^p\right) - \sum_{i \in Neg} \log\left(\hat{c}_i^0\right) \tag{3.16}$$

where

$$\hat{c}_i^p = \frac{\exp\left(c_i^p\right)}{\sum_p \exp\left(c_i^p\right)} \tag{3.17}$$

In this expression, $i \in Pos$ refers to all positively matched default boxes. Similarly, $i \in Neg$ refers to all negative default boxes that did not match a ground truth box.

The α value in Equation 3.12 is a balancing term used for changing the weight ratio of the two loss terms. In the case of this implementation, this term is set to 1 [71].

Optimizer: Having defined the loss function, the next step is to define the optimization process for minimizing the proposed loss function. The optimizer algorithm used for the training process for the single shot multi-box detector Mo-

bileNetV2 is called RMSProp. The RMSProp is an adaptive learning rate optimization method informally proposed by Geoff Hinton. Formally, this optimizer is defined as [76]:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E[(\frac{\partial L}{\partial w})^2]_t}} \frac{\partial L_t}{\partial w_t} \tag{3.18}$$

where w_{t+1} represents the updated weight parameter value and w_t represents the current weight parameter value, and $t \in \mathbb{N}_0$. The learning rate, η, is set to 0.004 following the same training configuration for the SSD MobileNetV2 COCO as Huang et al. [63].

The term $\frac{\partial L_t}{\partial w_t}$ is the gradient of the loss function with respect to the weight w at iteration t, and $\sqrt{E[(\frac{\partial L}{\partial w})^2]_t}$ models the square root of the moving average of squared gradient values over all previous iterations until iteration t. This term is defined as [76]:

$$E[(\frac{\partial L}{\partial w})^2]_t = \beta E[(\frac{\partial L}{\partial w})^2]_{t-1} + (1-\beta)\left(\frac{\partial L_t}{\partial w_t}\right)^2 \tag{3.19}$$

where $t \in \mathbb{N}_0$ and $E[(\frac{\partial L}{\partial w})^2]_0 = (1-\beta)\left(\frac{\partial L_0}{\partial w_0}\right)^2$. The decay factor, β, is set to 0.9 as suggested by Hinton et al. for typical application of this optimizer [76]. These are the major training parameters for transfer learning, however additional training parameters can be found in the source code file titled "ssd_mobilenet_v2_coco.config" at https://github.com/Sarmyt/masters_object_detection [63].

Training and Loss: The training process is iterative and moves on a basis of steps. During one step, the model trains and optimizes the loss function and network parameters on a batch size of 24 training images. The model was trained for a total of 1207 steps and this process took approximately 600 seconds or 10 minutes to complete on a Tesla T4 GPU. The loss stabilized at approximately 200 steps into the training process, however training was continued to ensure convergence. The plot of loss versus steps is seen in Fig 3.30. The configuration of the single shot multi-box detector MobileNetV2 was modified to only detect Alphabot2 agents and no other onject classes, as this was all that was required for localization of the Alphabot2 robots in the environment. This was done to improve the accuracy and reliability of the object detection framework.

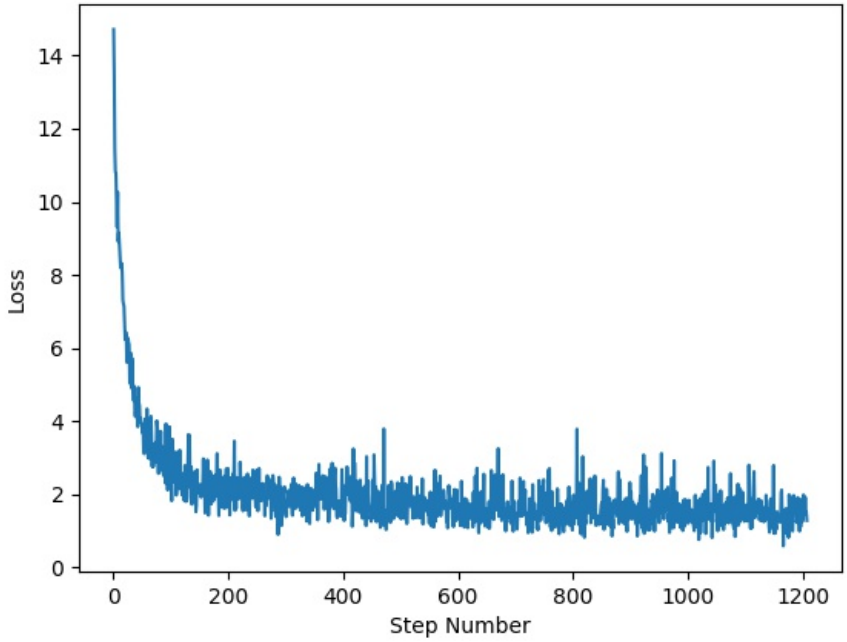


Figure 3.30: Plot of loss versus training steps for custom object detection training with single shot multi-box detector MobileNetV2

3.6.1.8 Inference and Results

The trained model was exported and tested in real time. It was able to consistently detect all five Alphabot2 agents at a given time in the environment with $\approx$ 99% confidence. The output of the inference from the trained model is seen in Fig 3.31 on images containing multiple agents.

The lightweight structure of the MobileNetV2 feature extractor with the single shot multi-box detector framework allows for close to real time inference. The overhead webcam has a real time frame rate of $\sim$ 6 frames per second at a resolution of 640 pixels by 480 pixels as mentioned in Section 3.3.1. The object detection model was able to run inference on these images in close to real time at $\sim$ 5 frames per second.

With this object detection localization technique, the localization accuracy of the Alphabot2 agents within the environment was found to be within the sufficient range of $\sim$1cm. Visualization of the inference results showed that when calculating the position of the centroids of the positive bounding boxes predicted by the SSD MobileNetV2, their position would coincide with the Alphabot2 robot in the frame 100% of the time. This indicates high accuracy indoor localization ($\sim$1cm).

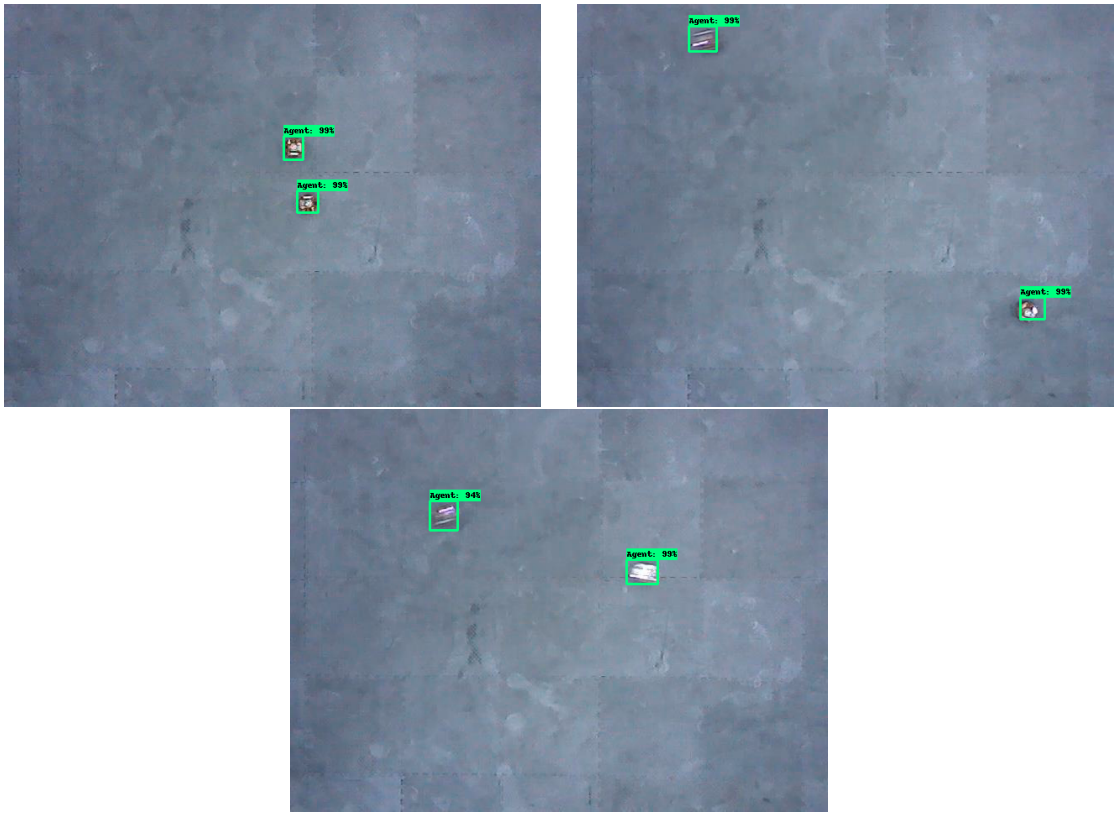


Figure 3.31: Real time inference samples for localization of Alphabot2 agents in the environment using object detection techniques

3.6.1.9 Object Tracking

For complete localization, it is not enough to detect the object of interest, but the movement of the object must also be tracked. A frame by frame centroid tracking algorithm is implemented for this purpose. This algorithm is repetitive and is repeated for every new frame collected from the video stream. The structure of this algorithm is as follows [77]:

- **Initialize:** The first frame of the video stream is collected and an instance of object detection is run on this frame. The bounding boxes of all detected objects in this initial frame are collected, the centroid of each bounding box is computed and each object is given a unique ID. Bounding boxes are of rectangular shape and their position is defined by two corner points. The upper left corner $R_1 = (x_1, y_1)$ and the lower right corner $R_2 = (x_2, y_2)$. Given these points, the centroid of a bounding box is computed as:

$$\text{Centroid} = (c_x, c_y) \quad \text{where} \quad c_x = \frac{x_1 + x_2}{2} \quad \text{and} \quad c_y = \frac{y_1 + y_2}{2} \tag{3.20}$$

- **Step 1:** Collect the next frame from the video stream and run object detection on this frame.
- **Step 2:** Compute the centroids of all the detected bounding boxes in the new frame.
- **Step 3:** Calculate the euclidean distance between all the centroids in the previous frame and all the centroids in the new frame.
- **Step 4:** Update the centroid position of all objects found in the previous frame to a new centroid position corresponding to the position of the centroid in the new frame that is the smallest euclidean distance away from the previous centroid position of each object.
- **Step 5:** If a new object of interest enters the field of view, then the new frame will have more objects in it than the previous frame did. In this case, any additional object is provided a unique ID and added to the tracking system.
- **Step 6:** If an object of interest leaves the field of view, then the new frame will have less objects in it than the old frame did. In this case, any centroid from the previous frame which is not matched to a centroid in the new frame is considered to have left the field of view. If this object remains out of the field of view for 50 consecutive frames, then it stops being tracked entirely.
- **Step 7:** Repeat Steps 1 to 6.

A common problem that can occur during object tracking is the problem of crossing. Crossing refers to instances where the trajectories of multiple objects cross making it difficult to resolve the new position of those object. Step 4 of the centroid tracking algorithm helps in dealing with the problem of crossing.

Step 4 of the centroid tracking algorithm is based on the assumption that the movement of an object between one frame to the next frame will be minimal. For this reason the updated location of each object in the new frame is assumed to be the location that is closest to its previous location [77]. Given the high frame rate of cameras, this assumption holds true in most cases. Due to the high frame rate of cameras, objects typically do not get enough travel time between frames to cross trajectories of other objects. Additionally, due to object avoidance algorithms on board the Alphabot2 robots, collisions between robots are rare and significantly reduce chances of crossing occurring.

However, an important note is that this assumption has its drawbacks in three dimensional space, as from the point of view of a camera, objects in three dimensions can overlap. However, since the hardware test platform consists of terrestrial locomotion robots on a planar environment, overlapping of objects is not possible.

The overall performance of this centroid tracking algorithm was found to be satisfactory as assessed by visual observation.

3.6.1.10 Determining Orientation

The object detection model and the centroid tracking algorithm are used for the purpose of obtaining and tracking the x and y coordinate information of the Alphabot2 agents in the environment. In order to obtain the full pose of these agents, their orientation must also be known. The full pose of each robotic agent is necessary for determining the appropriate control actions required to navigate the agent to its desired position.

This orientation information is obtained by recognizing the fact that the control space of the Alphabot2 agents is limited to rotation in one spot, or straight line motion with constant heading. The instances of straight line motion by the Alphabot2 agents can be used for determining orientation.

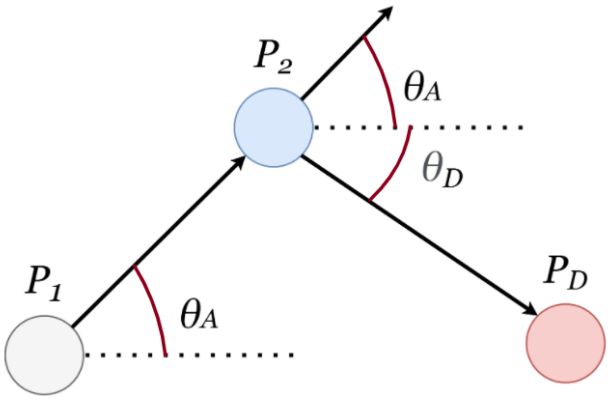


Figure 3.32: Determining orientation and control action for the Alphabot2 robots

In Fig 3.32, this situation is analyzed. The point $P_1(x_1, y_1)$ represents the previous position of the agent. The agent then moves in straight line motion with constant heading and arrives at $P_2(x_2, y_2)$. The point $P_D(x_d, y_d)$ is the desired position of the agent. In this situation θ_A refers to the heading of the agent with respect to the positive x-axis. The angle θ_D refers to the angle with respect to the positive x-axis of the vector which connects the agent to the desired position.

If the coordinates of all three points are known, then θ_A can be found by:

$$\theta_A = \tan^{-1}\left(\frac{x_2 - x_1}{y_2 - y_1}\right) \tag{3.21}$$

and similarly, θ_D can be found using:

$$\theta_D = \tan^{-1}\left(\frac{x_d - x_2}{y_d - y_2}\right) \tag{3.22}$$

The required control action to then get the agent to its desired position is as follows. The agent must first rotate to align its heading with its desired position. This desired rotation is θ_D - θ_A. After aligning its heading to the desired position, the agent simply travels in a straight line towards the desired position.

3.6.2 Wireless Communication Network Using XBee

Section 3.5.1 discusses the process of receiving XBee data packets on the side of the Alphabot2 agents. On the other hand this section discusses the process of sending data from the controller XBee module to all the other receiver XBee devices that are attached to their respective Alphabot2 agents using Python.

The open source XBee Python library is used for this implementation [54]. The implementation begins by initializing the XBee controller module which is attached to the central processing unit device that is running the Python code. This is achieved by connecting to a local port on the central processing unit that the XBee controller is connected to. All the receiving XBee devices are then initialized using their unique 64 bit addresses. Keeping in mind that all configuration steps mentioned in Section 3.4 are performed first.

Once this connection is initialized, data can be sent between the controller XBee to any specific receiving XBee device in a synchronous or asynchronous manner. The process of sending synchronous data requires all XBee modules to be operating in API mode. When data is sent synchronously to a receiver module, the Python code execution is blocked until a status update is received back to the controller module. This status update indicates whether the message was successfully received by the receiver module or not. This is a slower, but more reliable form of sending data packets as the status of data packets is provided [54].

On the other hand, data can be sent asynchronously. In this scenario, data is sent to the receiver module and the execution of the Python code is unaffected. No

status update message is sent back to the controller, so the successful delivery of the data packet is never confirmed. However, the execution of this form of sending data is more quick and simple [54].

In the case of the experiments conducted in this thesis, data from the controller module is sent asynchronously as data packet status updates are not required. Furthermore, in the lab setting, these XBee modules are relatively close in proximity and usually have a line of sight connection with each other, making data packet drops rare to begin with.

3.6.3 Workflow of Python Implementation

The general workflow of the Python implementation of this thesis uses a three phase approach.

3.6.3.1 Pre-initialization

The pre-initialization phase deals with the initial placement of robotic agents within the environment. This step is implemented manually and allows the user of the test platform to decide the initial positions of the robotic agents in the particular algorithm scenario being tested.

3.6.3.2 Initialization

The localization framework of the algorithm testing platform (object detection model using overhead camera) is not linked to the wireless communication protocol of the algorithm testing platform (XBee mesh network). This is an issue because the localization framework localizes the agents in the environment and determines the appropriate control action required to navigate these agents to their respective desired location. However, after determining the appropriate control action for each agent at a given time step, the localization framework has no way of recognizing which unique XBee address each control action should be sent to. The initialization phase works to link these two systems together to ensure that the correct control action is sent to the correct XBee device.

This process of initialization is best explained through an example. The process begins with the placement of five Alphabot2 agents in the work space. The object detection and localization framework is then used to localize the agents in the work space. The localization framework, in no particular order, assigns arbi-

trary IDs to the Alphabot2 agents that it detects in order to continue tracking their movements. In a group of five robots, each robot is assigned an ID of either 1, 2, 3, 4 or 5.

Each Alphabot2 has a corresponding XBee device attached to it as well for receiving control actions in the form of wireless data packets. These XBee devices each have a unique 64 bit address, however for purposes of the example, assume that the XBee devices are identified as XBee 1, XBee 2, XBee 3, XBee 4 and XBee 5.

For initialization, a control action to move forward is sent from the central processing unit to the first XBee device, XBee 1. The localization framework then identifies which Alphabot2 moves forward. For sake of example, if the Alphabot2 with ID 4 moves forward, then a link can now be created between XBee 1 and the Alphabot2 robot with ID 4. This means that the first XBee device, XBee 1, is attached to the Alphabot2 robot that was labelled with ID 4. Next a control action to move forward is sent to XBee 2, and the corresponding Alphabot2 agent which moves forward is identified. This process continues until a link is created between all Alphabot2 agents identified by the object detection algorithm and all XBee devices. This process ensures that the right message is sent to the right device during the algorithm testing phase.

3.6.3.3 Multi-Agent System Control Algorithm Testing

The algorithm testing phase is relatively general. After a link has been created between the robots being detected and the ID of their corresponding XBee receivers, the test platform is able to send specific control actions to specific Alphabot2 agents. As such, in this phase a variety of algorithms can be implemented and tested. The next section will discuss the specific implementation and evaluation of multi-agent area coverage control and optimization algorithms using this test platform and the Python programming language.

3.6.4 Area Coverage Control and Optimization in Python

This design of a multi-agent hardware test platform is presented for general purpose multi-agent algorithm testing. For the purpose of validation of the proposed test platform design, the test platform is validated on multi-agent area coverage control and optimization algorithms. Within this domain, simulation data is generated for three scenarios of varying complexity. These scenarios are then tested on the hardware test platform in real time. The experimental results are then com-

pared with the results obtained from the simulation to evaluate the plausibility of the hardware test platform for lab testing this class of systems, and to gain insights into intrinsic limitations posed by the hardware.

The following three scenarios are implemented and evaluated using the hardware test platform:

1. Area coverage control and optimization of a rectangular convex environment, Ω, with uniform risk density.

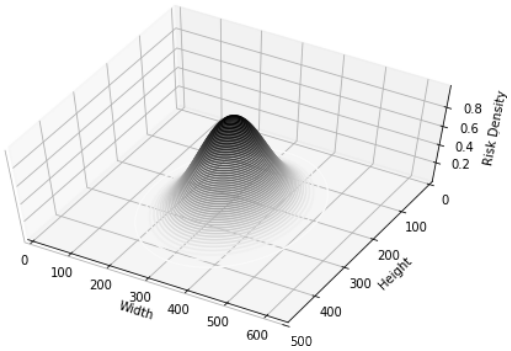
2. Area coverage control and optimization of a rectangular convex environment, Ω, with a non-uniform risk density distribution described by:

$$\phi(q) = \alpha e^{-\beta||d-q||^2} \tag{3.23}$$

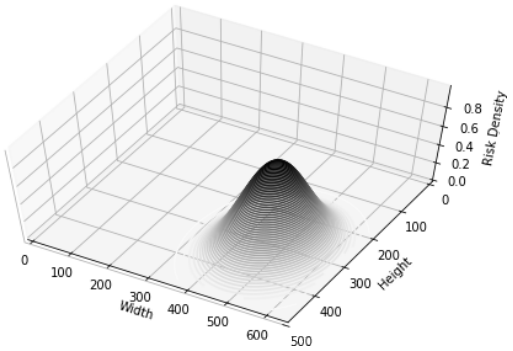
   where $q \in \Omega$ and d is the center of the risk density distribution, corresponding to the peak of the bell shaped distribution. For scenario 2, the point d is selected to be located at the centroid of the rectangular environment. The parameters α and β are chosen as 1 and 0.0001 respectively.

3. The setup of scenario 3 is similar to scenario 2. However, in this case the center of the risk density distribution, d, is selected to be off center of the rectangular environment. This scenario is implemented to introduce asymmetry to the environment, and to evaluate if the behaviour of the hardware test platform is affected by this lack of symmetry.

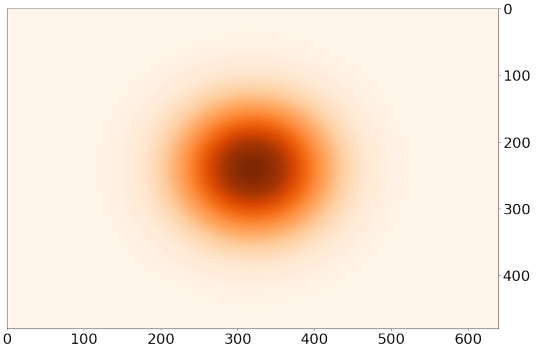
In the context of the Python code, the convex rectangular environment is covered by the webcam mentioned in Section 3.3.1. This webcam has a pixel resolution of 640 pixel width and 480 pixel height. It is these image representations of the environment which are processed and used for area coverage control and optimization calculations. As such, the pixel size (640 x 480) of the images collected by the webcam is used to represent the overall size of the environment. This means that for scenario 2 mentioned above, the location of the center of the risk density distribution is located at the pixel center of the images collected, which is at the pixel coordinate of 320 and 240. For scenario 3, the center of the risk density distribution is placed at an off center location corresponding to the pixel coordinate of 485 and 320. Fig 3.33 shows a three dimensional plot of the risk density distribution in both scenarios 2 and 3.



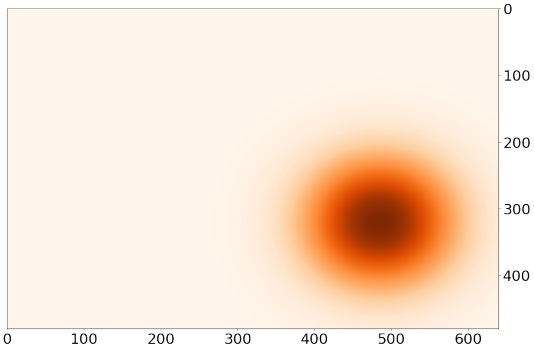
(a) Scenario 2 risk density distribution centered at the point (320, 240)



(b) Scenario 3 risk density distribution centered at the point (485, 320)



(c) Top view of scenario 2 risk density distribution centered at the point (320, 240)



(d) Top view of scenario 3 risk density distribution centered at the point (485, 320)

Figure 3.33: Visualization of the risk density distribution $\phi(q)$ in scenarios 2 and 3

3.6.4.1 Discretization of Environment

In order to perform real time calculations related to area coverage control and optimization, the environment is discretized into a coarse mesh grid (Fig 3.34) consisting of 10 by 10 pixel sized cells with coordinates at the center of each cell represented by red dots. The coordinates of this mesh grid are then sampled one by one to obtain a coarse representation of the Voronoi diagram of a particular arrangement of agents in the environment. A finer mesh grid can be used to obtain more accurate Voronoi diagram construction, however Voronoi diagram construction on a finer mesh grid demands more computing power and results in delays. Discretization of the environment into 10 by 10 pixel cells was found to result in a good practical balance of accuracy and speed of execution.

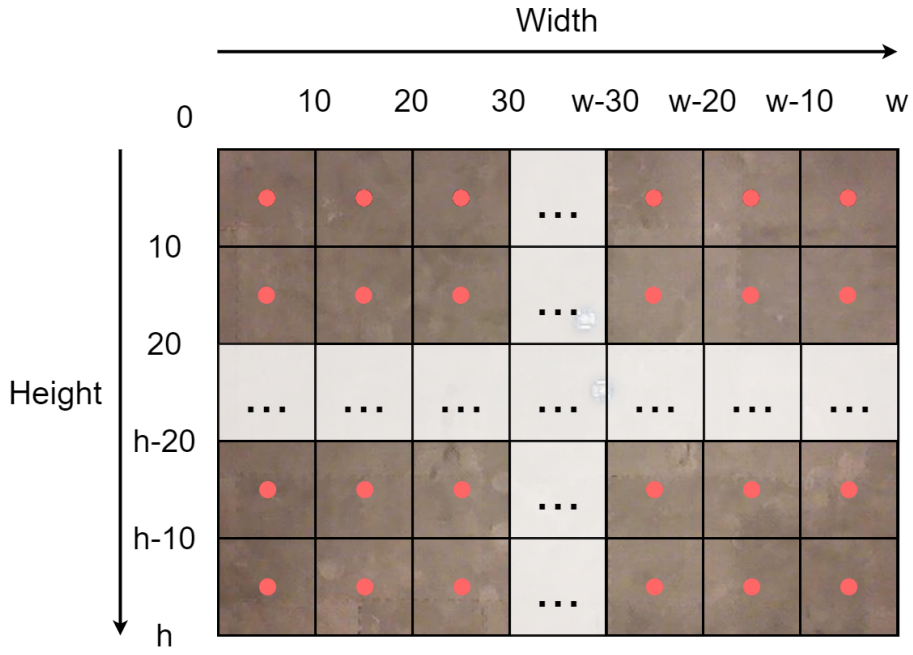


Figure 3.34: Discretization of multi-agent environment

3.6.4.2 Voronoi Diagrams

The process of Voronoi diagram construction begins with the object detection framework that is used for localization. Once all robotic agents in the environment are localized, the positions of the robotic agents act as generators for the Voronoi diagram construction.

The actual diagram construction process cycles through each coordinate of the mesh grid described above in the discretization process. For each coordinate of the mesh grid, the euclidean distance between that coordinate and all existing agents is calculated. The coordinate being evaluated and the area of its respective cell is then assigned to the Voronoi partition of the agent that is located the closest to that coordinate (shortest euclidean distance away). An example of the output of this process can be seen in Fig 3.35, where the black dots represent agent positions in the environment. Each agent has a collection of sampled coordinate points that were found to lie closer to it than all other agents in the environment.

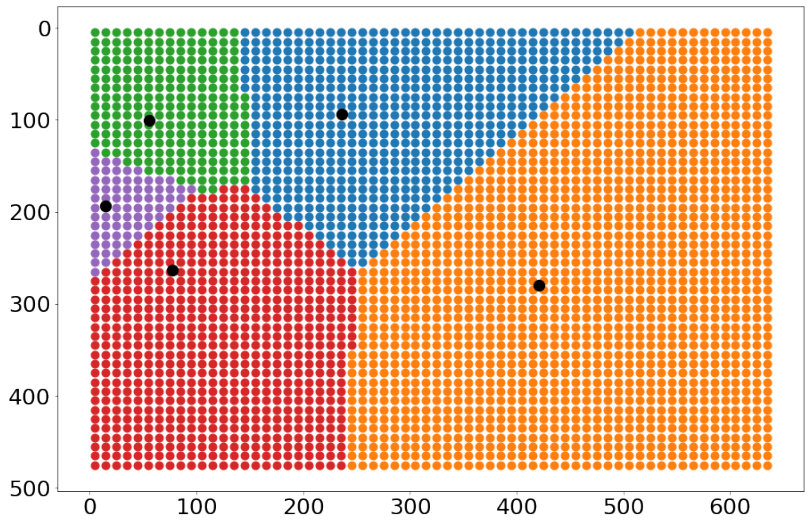


Figure 3.35: Voronoi diagram construction in a discretized environment containing five robotic agents represented by black dots

3.6.4.3 Lloyd's Algorithm and Centroidal Voronoi Diagrams

To refresh the background theory introduced in Section 2.2.3, centroidal Voronoi diagrams represent optimal coverage placements in the context of multi-agent area coverage. For this reason they are a useful tool for multi-robot networks. Lloyd's algorithm is an iterative technique that can be used to find centroidal Voronoi diagram arrangements.

An important component of Lloyd's Algorithm is the calculation of the generalized centroid of each Voronoi partition at each iteration. This section focuses on achieving this task in a discretized environment. Equations 2.13 and 2.14 can both be modified and applied to a discrete environment as follows:

- Generalized mass of Voronoi cell $\mathcal{V}$:

$$\tilde{M}_{\mathcal{V}_i} = \sum_{Q_{\mathcal{V}_i}} \tilde{\phi}(p_i, q) \times q_{area} \tag{3.24}$$

- Centroid (center of mass) of Voronoi cell $\mathcal{V}$:

$$\tilde{C}_{\mathcal{V}_i} = \frac{1}{\tilde{M}_{\mathcal{V}_i}} \sum_{Q_{\mathcal{V}_i}} q\tilde{\phi}(p_i, q) \times q_{area} \tag{3.25}$$

Here, $Q_{\mathcal{V}_i}$ represents the collection of coordinates in the discrete space that belong to the Voronoi cell $\mathcal{V}_i$ and $\tilde{\phi}(p_i, q)$ represents the generalized risk density dis-

tribution over the Voronoi cell $\mathcal{V}_i$ as defined in Equation 2.12. The term q_{area} refers to the area of the cell of the coordinate point q. As mentioned in Section 3.6.4.1, the environment is discretized into cells with a dimension of 10 by 10 pixels. This means that q_{area} is set to 100. This effectively approximates the integral terms in Equations 2.13 and 2.14 in the context of a discretized environment.

To continue this discussion, the function f will be discussed. This function describes the sensing capabilities, and consistently with the literature on area coverage, it is defined to be continuously differentiable and strictly decreasing with the argument. A commonly adopted form is

$$f(r_i) = \exp\left(-\frac{r_i^2}{\sigma^2}\right) \tag{3.26}$$

which is a bell shaped function centered at the agent's position p_i. The value $\sigma = 150$ is adopted. This parameter is related to how the radius of influence of an agent decays. Given that only five agents populate the environment, a relatively large σ value was chosen to ensure that there is sufficient overlap between the radius of influence of the five agents. A value of $\sigma = 150$ provides the agent with a reasonable radius of influence of 150 pixels. The total area of such a circle is $\pi r^2 = \pi(150)^2 = 70686$. The total area of the environment is $W \times H = 640 \times 480 = 307200$. This means that each robot has the capacity to reasonably sense $\approx 20 - 25\%$ of its environment at a given time. Given that the environment is being covered with five agents, this is a sensible value that ensures the environment will be covered effectively by the five agents.

For future work on this test platform, a parametric approach with respect to σ is recommended. Running multiple experiments with a variety of values for σ will provide more insights into the behaviour of the hardware test platform and will allow for a more complete evaluation of the test platform with respect to agent sensing performance in area coverage control and optimization algorithms.

Adopting this form of f, and substituting into Equation 2.12, we obtain:

$$\tilde{\phi}(p_i, q) = \frac{1}{11250}\phi(q)\exp\left(-\frac{r_i^2}{\sigma^2}\right) \tag{3.27}$$

substituting Equation 3.27 into Equations 3.24 and 3.25 and setting $q_{area} = 100$ as discussed above, we obtain:

$$\tilde{M}_{\mathcal{V}_i} = \frac{1}{112.50} \sum_{Q_{\mathcal{V}_i}} \phi(q) \exp\left(-\frac{r_i^2}{\sigma^2}\right) \tag{3.28}$$

$$\tilde{C}_{\mathcal{V}_i} = \frac{1}{112.50\tilde{M}_{\mathcal{V}_i}} \sum_{Q_{\mathcal{V}_i}} q\phi(q) \exp\left(-\frac{r_i^2}{\sigma^2}\right) \tag{3.29}$$

Figure 3.36 shows three plots, in which the positions of the five agents within the environment is constant. Figure 3.36a shows the calculation of the generalized centroid of each Voronoi partition (navy blue dots) in scenario 1, where the risk density distribution is constant. Similarly, Figures 3.36b and 3.36c show this same calculation for scenario 2 and scenario 3 respectively. In each of these cases, the center of the risk density distribution is indicated by the single large red dot on the plot. It can be seen how the risk density distribution affects the generalized centroid calculation process.

3.6.4.4 Lloyd's Algorithm Using Alphabot2 Agents

Once generalized centroids are calculated, the final step of a single iteration of Lloyd's algorithm involves moving each agent to the position of the generalized centroid of its respective Voronoi cell. If the pose of each agent is known and the position of the generalized centroids is known, then this task becomes trivial. A control action can simply be sent to each agent using the XBee protocol, indicating the change in orientation required for the agent to be facing its desired position. The robot is then actuated forwards towards the desired position.

The complete Python workflow for the implementation of Lloyd's algorithm within the framework of the proposed hardware test platform design is summarized below:

- Step 1: Agent positions are detected and a unique ID is assigned to each robot for tracking
- Step 2: A Voronoi diagram is constructed using the agent positions as generators
- Step 3: The generalized centroids of all Voronoi cells are calculated
- Step 4: Each Alphabot2 agent is provided the appropriate control actions required for it to move from its current position to the position of the generalized centroid of its corresponding Voronoi cell

- Step 5: Once all Alphabot2 agents have arrived at their corresponding generalized centroid locations, the program checks if the new arrangement results in a centroidal Voronoi diagram. If yes, then the algorithm has converged. If no, then steps 2 to 5 are repeated until a centroidal Voronoi diagram is constructed.

3.6.4.5 Coverage Metric

Throughout the process of convergence of Lloyd's algorithm, the program collects information related to the coverage metric of the environment. This helps quantify how the area coverage optimization process is evolving over time. The definition of the coverage metric as introduced by Equation 2.9 can be modified for a discretized space:

$$\mathcal{H}(P) = \sum_{i=1}^{n} \sum_{Q_{\mathcal{V}_i}} f(r_i)\phi(q) \times q_{area} \tag{3.30}$$

where $P = \{p_1, ..., p_n\}$ is the collection of the n agents' states, the position of the $i-th$ agent in the environment is given by p_i, $Q_{\mathcal{V}_i}$ represents the collection of coordinates in the discrete space that belong to the Voronoi cell $\mathcal{V}_i$ of the $i-th$ agent. As before, $\phi(q)$ represents the risk density distribution over the discretized space, and r_i represents the euclidean distance between the position of the $i-th$ agent, p_i, and the coordinate q in the work space. The term q_{area} refers to the area of the cell of the coordinate point q. As mentioned in Section 3.6.4.1, the environment is discretized into cells with a dimension of 10 by 10 pixels. This means that q_{area} is set to 100. This effectively approximates the integral term in Equation 2.9 in the context of a discretized environment. The function f describes the sensing capabilities, and consistently with the literature on area coverage, it is defined to be continuously differentiable and strictly decreasing with the argument. As mentioned in Section 3.6.4.3, the form adopted for this function is:

$$f(r_i) = \exp\left(-\frac{r_i^2}{\sigma^2}\right) \tag{3.31}$$

where $\sigma = 150$.

3.7 Simulation of Multi-agent Coverage Control and Optimization

Simulations of each scenario were also conducted in order to verify the correspondence between the the hardware test platform set up for area coverage coverage control, and well established theoretical predictions. Simulations were conducted in a visualization friendly application known as Processing, using Python code. The code for the simulation can be found in Appendix C.

The simulation uses all the same principles mentioned in the previous section, such as environment discretization and the same algorithms for calculating Voronoi diagrams, generalized centroids of Voronoi cells and the coverage metric. An environment with the same dimensions is used (640 by 480) and the environment is discretized into 10 by 10 cells to ensure consistency between the theoretical simulation and the hardware test platform simulation. The same workflow is used for Lloyd's algorithm as mentioned in Section 3.6.4.4.

While typically we validate theory with experimentation, in this thesis we want to compare the experimental results to simulations of well researched area coverage control algorithms. As this thesis focuses on the design of a hardware test platform for multi-agent algorithms, the objective here is to evaluate if the proposed hardware test platform design can plausibly reproduce behaviour that is consistent with theoretical predictions of area coverage control algorithms. Theoretical research on algorithms in the domain of area coverage control with time invariant risk distribution is quite mature. Therefore if the hardware test platform is able to show plausibility of consistent behaviour with this class of algorithms, it will indicate that the proposed hardware test platform design has usability in this domain.

The key differences between the simulation and the experimental implementation stem from the fact that the simulation makes use of point agents whereas the experimental implementation makes use of Alphabot2 mechatronic agents. As such, the simulation has no localization framework, no wireless communication protocol and no complex process for receiving and implementing control actions at the level of the agents. Furthermore, the kinematics of the mechatronic robotic agents in the experimental implementation introduce inherent complexity and limitations. Due to these differences, the scope of the comparison between the experimental results and the simulation results is limited, and is only meant to evaluate if the proposed hardware test platform design can plausibly reproduce the theo-

retically predicted behaviour for this class of algorithms.

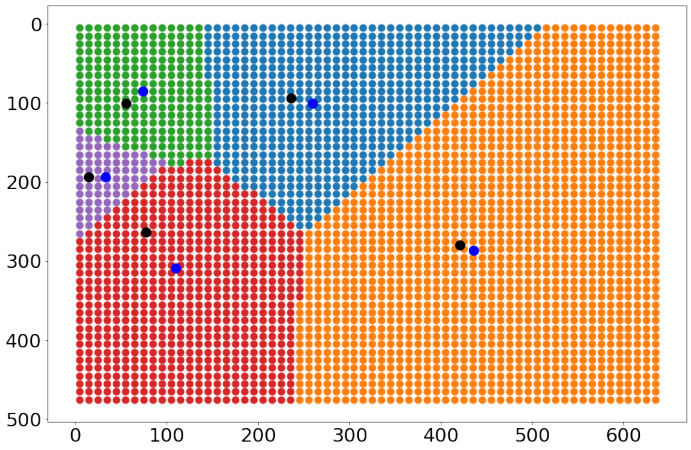
For all three scenario that were tested, the initial positions of the agents used in the experimental implementation were the same as the initial positions used in the simulation as well for consistency. Finally, the hardware test platform has a time factor of approximately five seconds required for it to go through the process of executing a control action. This means the process of sending data to a receiver, receiving that data at the receiver side and then implementing the control action contained in that message has five seconds allocated to it. In basic terms, a control action is sent every five seconds, providing the robotic agents with a delay during which the control action can be executed. This process is emulated in the simulation as well to maintain consistency with respect to time of execution.

3.7.1 Higher Fidelity Simulations

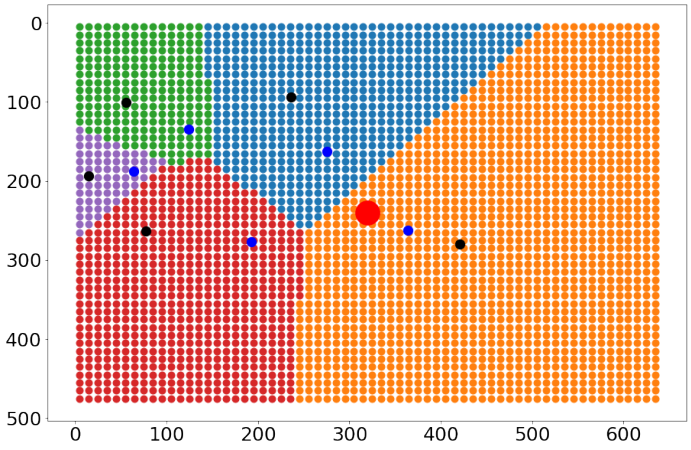
On the topic of simulations, it is worth noting that higher fidelity simulation software such as Gazebo and Webots can also be used for multi-agent algorithm testing. Using higher fidelity simulation software for algorithm testing has benefits. In simulations, a variety of robots can be used to model agents within the work space. Simulated hardware components can be swapped in or out with relative ease. Multiple trials can be run with varying algorithmic parameters and environmental conditions. Finally, in simulations, unpredictable behaviour from hardware components is not an issue.

However, this thesis proposed a design for a hardware based test platform with the understanding that their exists a reality gap between simulation and experiment. High fidelity simulations can still not fully capture the lessons that experimental implementation can provide, such as hardware based limitations or the effects of complex environmental factors that are difficult to model in simulation.

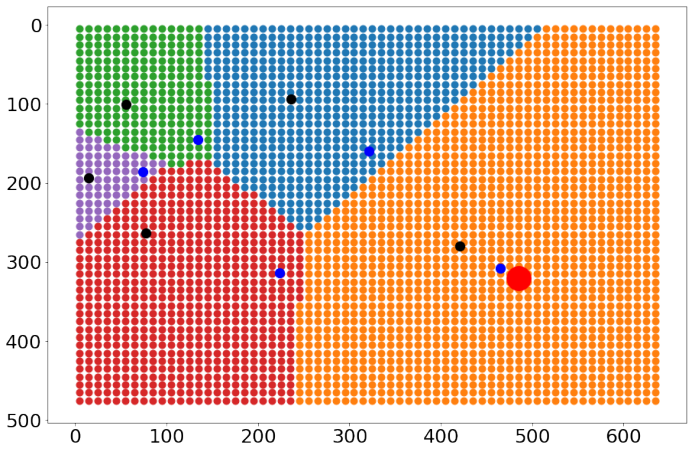
As such, the aim is to use this hardware based testing platform to learn which experimental limitations exist in hardware implementation. Once these lessons have been learned, they can eventually be migrated over to a simulated environment. This will help improve the accuracy and realism of simulated environment. The eventual goal being to leverage both experimental implementation and high fidelity simulation to improve multi-agent algorithm validation in the lab setting.



(a) Scenario 1



(b) Scenario 2



(c) Scenario 3

Figure 3.36: Calculation of generalized centroids for Voronoi cells in various discretized environments (agents are represented by black dots, generalized Voronoi cell centroids are represented by navy blue dots)

Chapter 4

Presentation and Discussion of Results

This chapter presents and compares experimental results and simulation results of area coverage control and optimization. The chapter also presents a discussion of the obtained results and evaluates if the proposed hardware test platform design can plausibly reproduce behaviour that is consistent with theoretical predictions of area coverage control algorithms. The discussion also identifies shortcomings that can inform improvements to be addressed in future work.

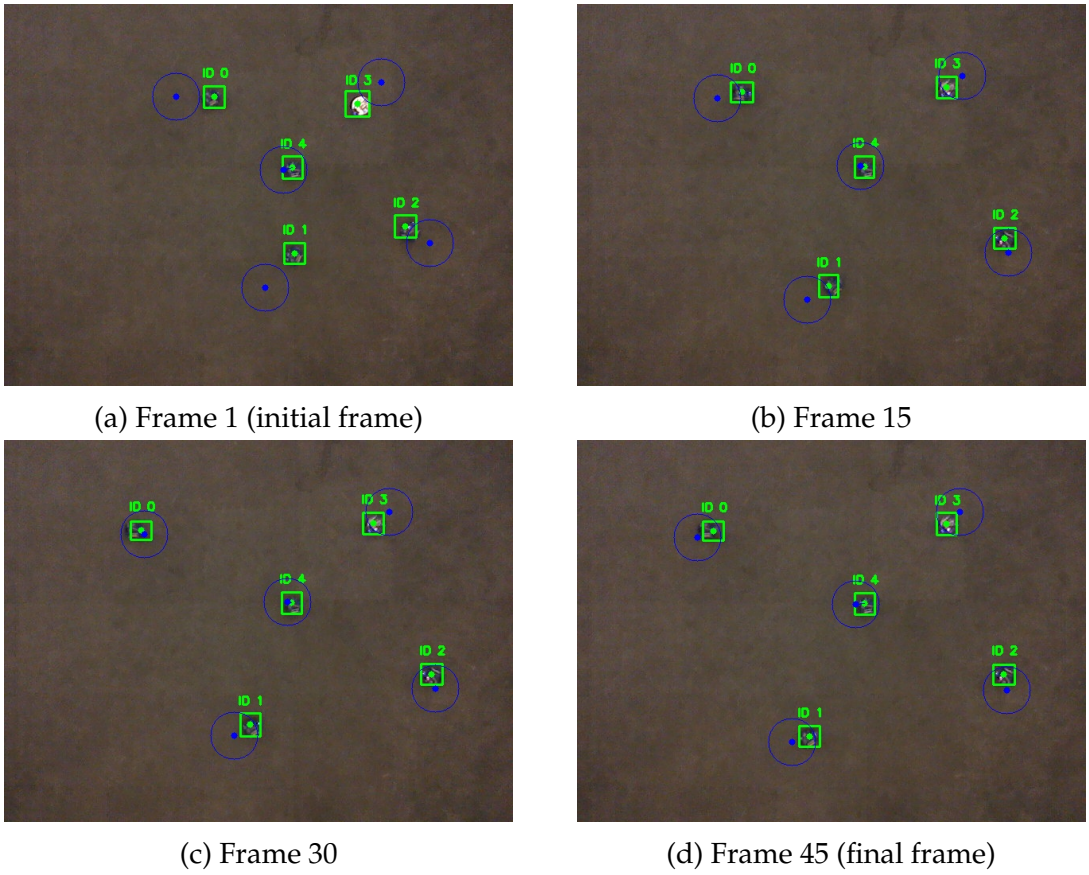
4.1 Results

This results section is broken down into three subsections, one for each area coverage optimization and control scenario that was tested.

4.1.1 Scenario 1: Constant Risk Density Distribution

In this scenario, area coverage control and optimization within a rectangular (640 by 480 pixel), convex environment, Ω, is implemented with five agents. This environment has a uniform risk density distribution represented by $\phi(q) = 1$ where $q \in \Omega$. The randomized initial positions of the five agents were (264, 116), (365, 313), (504, 279), (444, 125) and (362, 205) respectively. These same initial positions were used in both the experimental implementation and the simulation for consistency.

Fig 4.1 shows sample frames from the experimental results of area coverage control and optimization using the proposed hardware test platform design for



(a) Frame 1 (initial frame)

(b) Frame 15

(c) Frame 30

(d) Frame 45 (final frame)

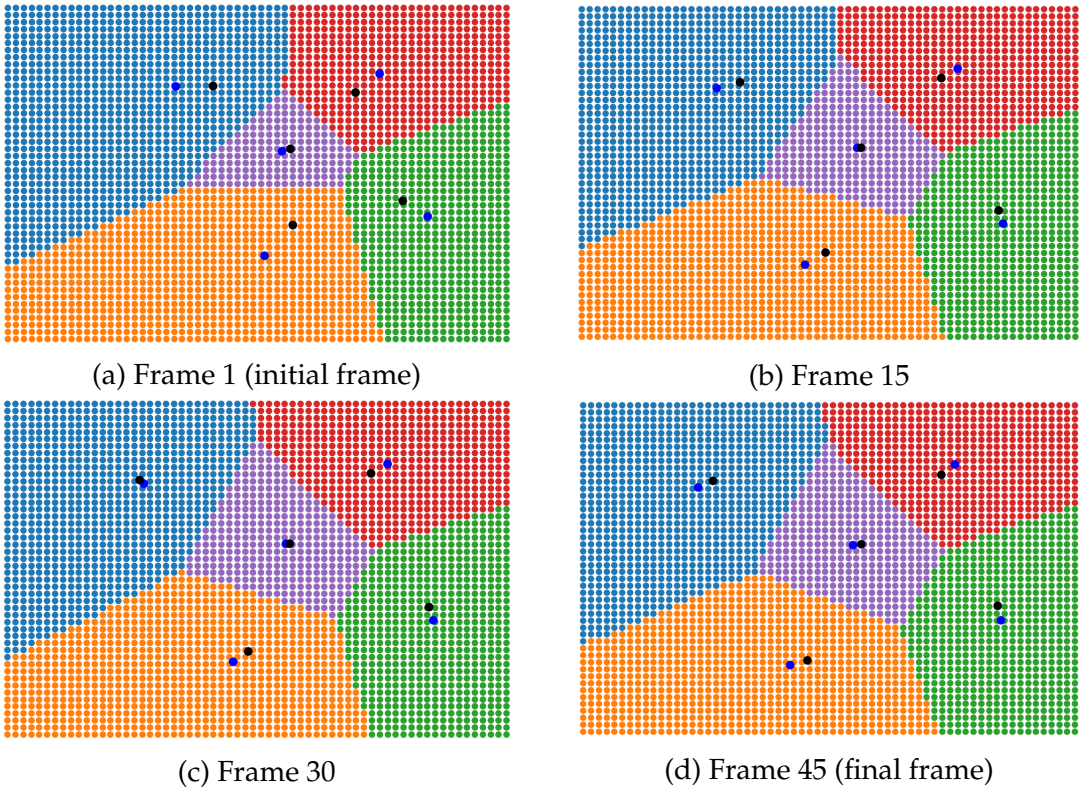
Figure 4.1: Sample frames from the experimental results of the implementation of scenario 1

scenario 1.

Fig 4.2 provides additional details about the sample frames seen in Fig 4.1. In Fig 4.2 we see the underlying Voronoi diagram of each frame along with parameters for Lloyd's algorithm. In this figure agents are represented by black dots, and their desired centroidal locations are represented by navy blue dots. The Voronoi partitions are represented by a collection of adjacent points sharing the same colour.

Fig 4.3 shows sample frames from the simulation of scenario 1. Additional details can be seen that highlight the underlying Voronoi diagram of each frame along with parameters for Lloyd's algorithm. In this figure agents are represented by black dots, and their desired centroidal locations are represented by navy blue dots. The Voronoi partitions are represented by a collection of adjacent points sharing the same colour.

Fig 4.4 shows the relative paths travelled within the environment by each agent during the experimental implementation (blue) and the simulation (purple) for



(a) Frame 1 (initial frame)

(b) Frame 15

(c) Frame 30

(d) Frame 45 (final frame)

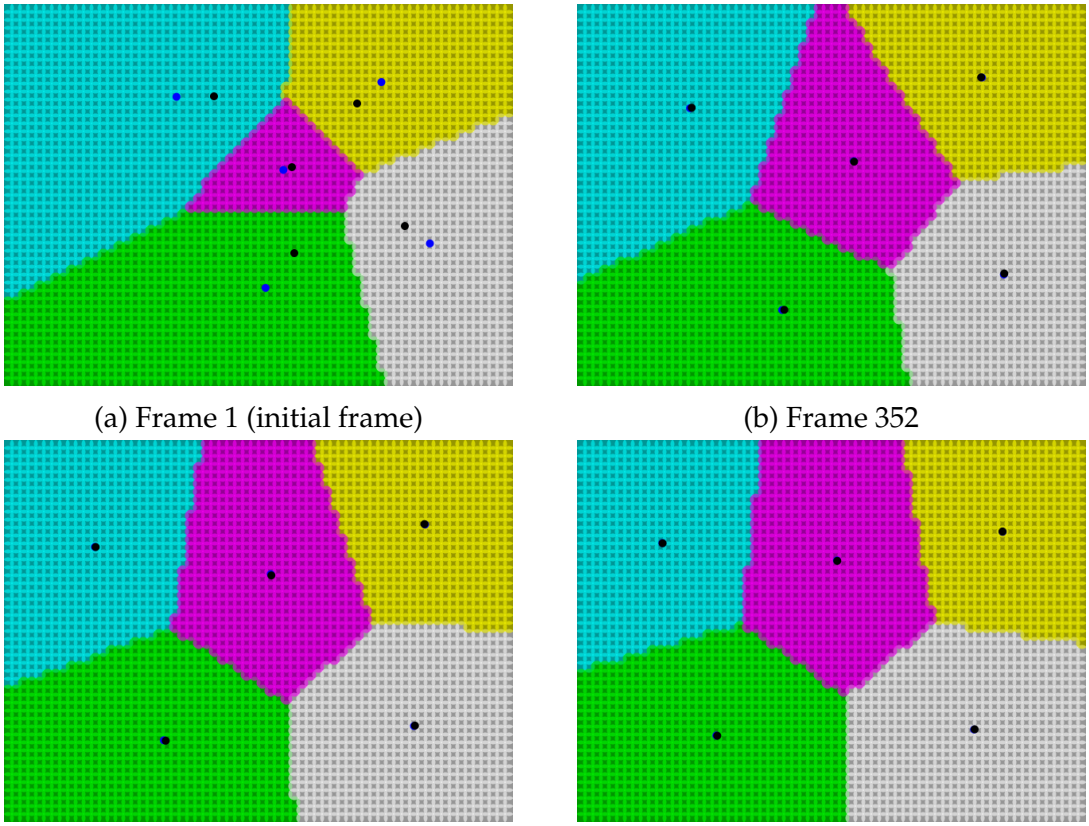
Figure 4.2: Underlying Voronoi diagram and Lloyd's algorithm parameters for sample frames from the experimental implementation of scenario 1

scenario 1. In this scenario the underlying risk density is represented by $\phi(q) = 1$, represented by the uniform distribution of orange in Fig 4.4.

To build on Fig 4.4, the plot in Fig 4.5 shows the exact pixel length of the path taken by each agent in both the experimental implementation and the simulation of scenario 1.

The results in Fig 4.6 show some consistency between the evolution of the coverage metric during the experimental implementation and the simulation. In the first 20 seconds, before the experimental implementation converges, the behaviour of both curves is similar and in both cases the coverage metric increases sharply at first and then levels out. However, the coverage metric in the simulated case evolves more smoothly. Furthermore, in the simulated case the coverage metric maximizes to a higher value with a longer run time to convergence. Some reasons are provided in the discussion to explain the discrepancies seen here between the experimental results and the simulation.

As both simulations start with the same initial conditions, the initial coverage metric is 154109 in both cases. The experimental implementation converges to a



(a) Frame 1 (initial frame)

(b) Frame 352

(c) Frame 705

(d) Frame 1058 (final frame)

Figure 4.3: Underlying Voronoi diagram and Lloyd's algorithm parameters for sample frames from the simulation of scenario 1

final value of 183499. The simulation converges to a final value of 199885. Based on these values the change in metric coverage for each scenario is:

$$\Delta_{exp} = 183499 - 154109 = 29390$$
$$\Delta_{theo} = 199885 - 154109 = 45776$$

Given these values the percent difference between these values is found to be 43.60%:

$$\% \text{ difference} = \frac{|\Delta_{theo} - \Delta_{exp}|}{\frac{\Delta_{theo} + \Delta_{exp}}{2}} * 100 = \frac{|45776 - 29390|}{\frac{45776 + 29390}{2}} * 100 = \frac{16386}{37583} * 100 = 43.60\%$$

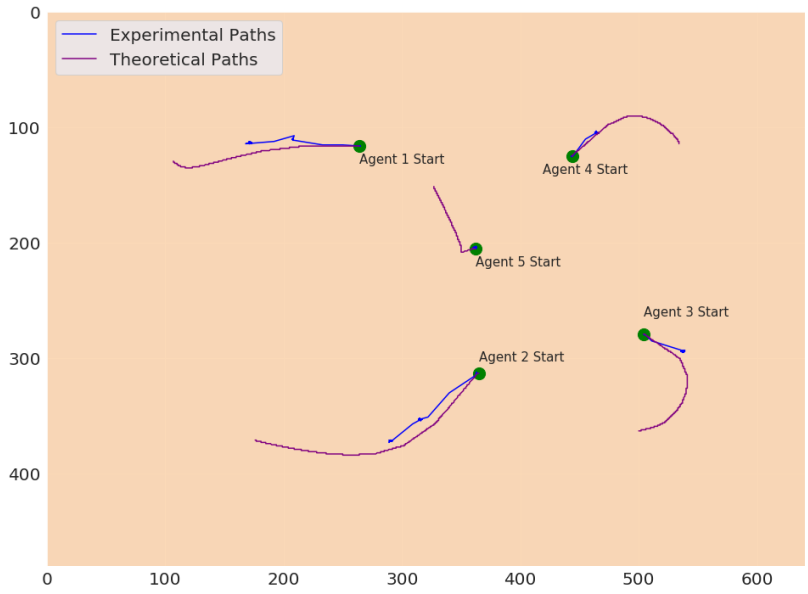


Figure 4.4: Paths travelled by each agent in the experimental implementation and the simulation of scenario 1

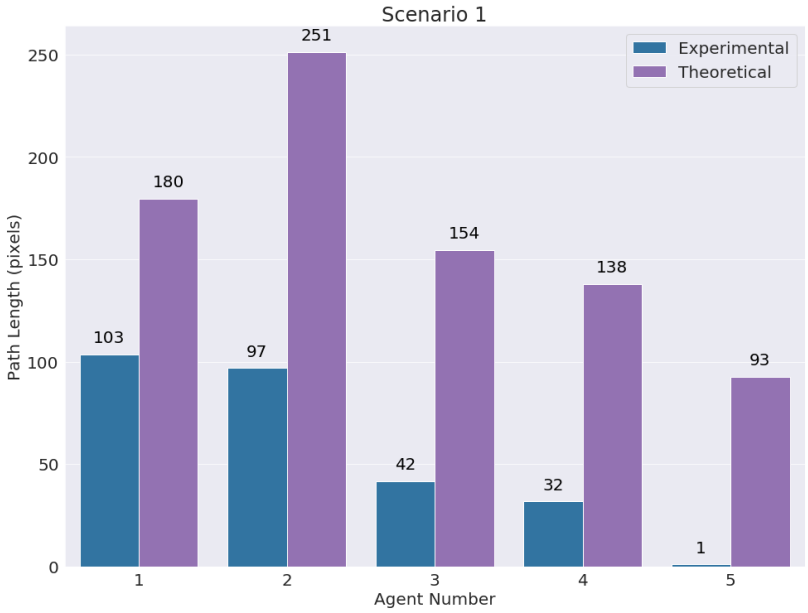


Figure 4.5: Lengths of paths travelled by each agent in the experimental implementation and the simulation for scenario 1

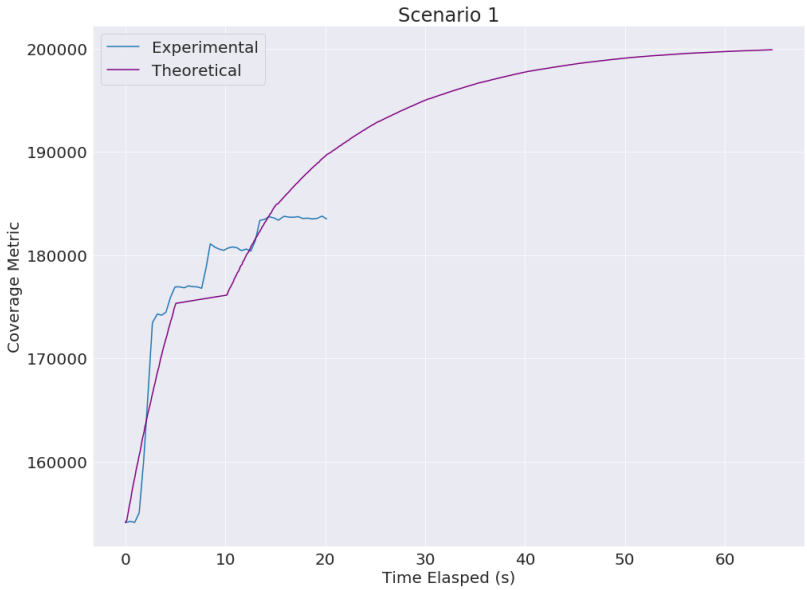
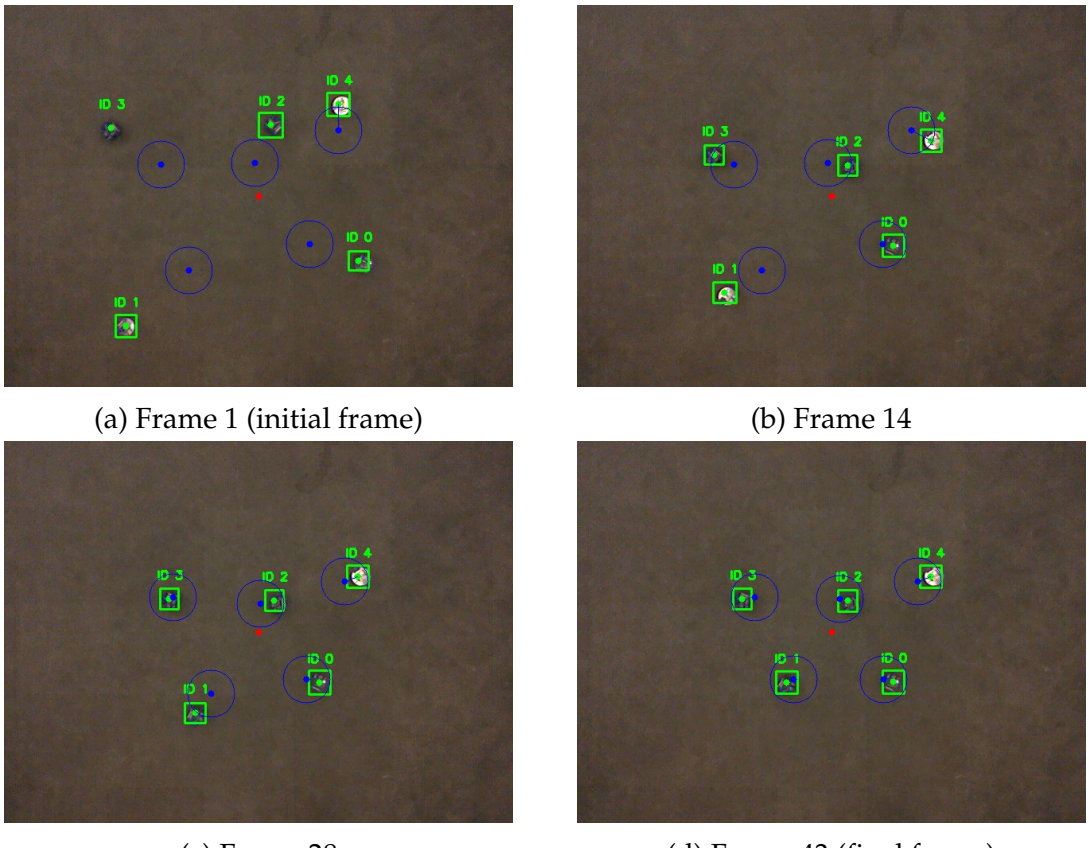


Figure 4.6: Evolution of the coverage metric during the experimental implementation and the simulation for scenario 1

4.1.2 Scenario 2: Time Invariant Nonuniform Risk Density Distribution Located at the Center of the Environment

In this scenario, area coverage control and optimization within a rectangular (640 by 480 pixel), convex environment, Ω, is implemented with five agents. This environment has a nonuniform time invariant risk density distribution represented by $\phi(q) = \alpha e^{-\beta||d-q||^2}$ where $q \in \Omega$ and d is the coordinate (320, 240). The parameters α and β are chosen as 1 and 0.0001 respectively. The randomized initial positions of the five agents were (445, 321), (153, 403), (335, 150), (134, 154) and (420, 124) respectively. These same initial positions were used in both the experimental implementation and the simulation for consistency.



(a) Frame 1 (initial frame)

(b) Frame 14

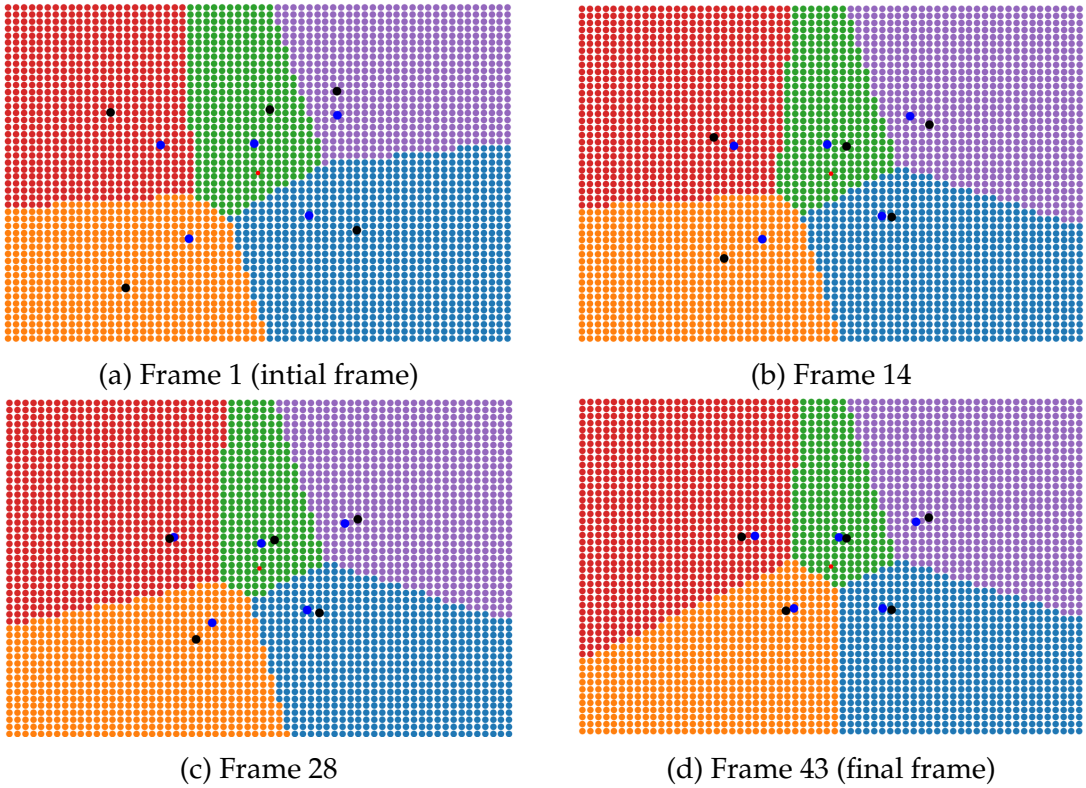
(c) Frame 28

(d) Frame 43 (final frame)

Figure 4.7: Sample frames from the experimental results of the implementation of scenario 2

Fig 4.7 shows sample frames from the experimental results of area coverage control and optimization using the proposed hardware test platform design for scenario 2.

Fig 4.8 provides additional details about the sample frames seen in Fig 4.7. In



(a) Frame 1 (intial frame)

(b) Frame 14

(c) Frame 28

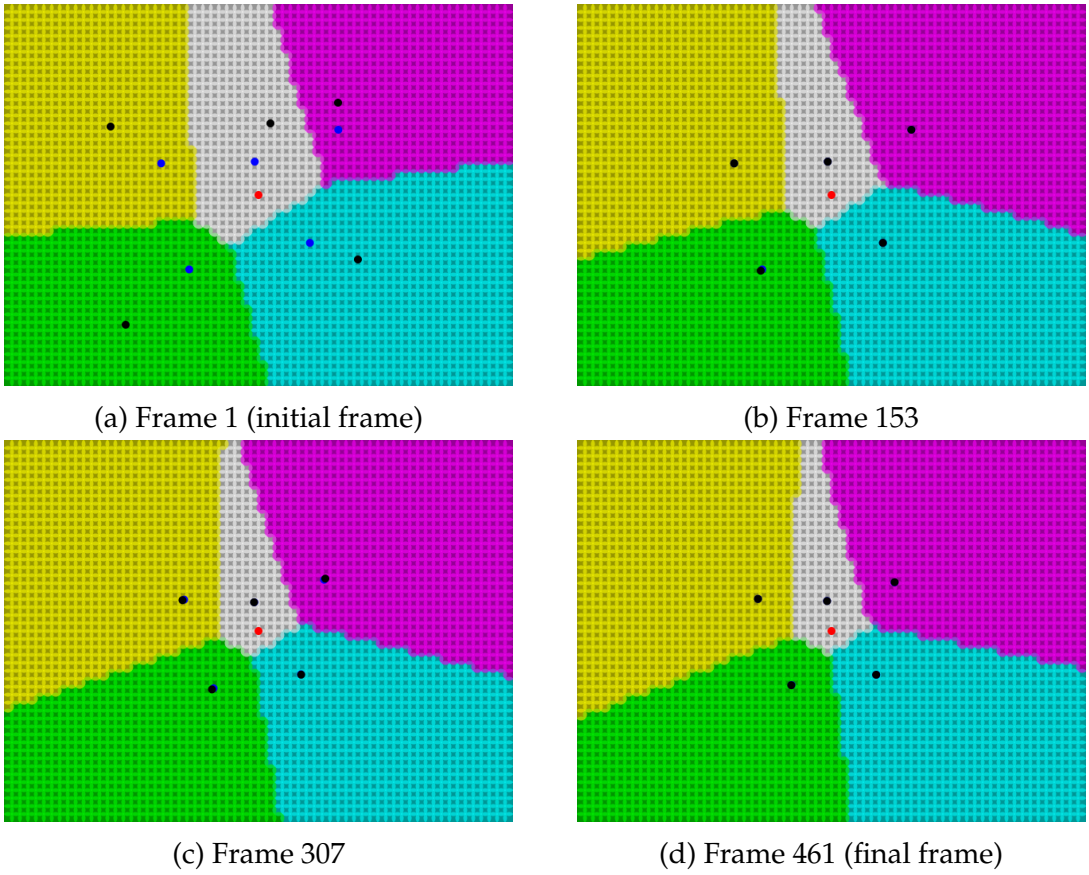
(d) Frame 43 (final frame)

Figure 4.8: Underlying Voronoi diagram and Lloyd's algorithm parameters for sample frames from the experimental implementation of scenario 2

Fig 4.8 we see the underlying Voronoi diagram of each frame along with parameters for Lloyd's algorithm. In this figure agents are represented by black dots, and their desired centroidal locations are represented by navy blue dots. The Voronoi partitions are represented by a collection of adjacent points sharing the same colour and the center of the risk density distribution (320, 240) is visualized as a bright red dot.

Fig 4.9 shows sample frames from the simulation of scenario 2. Additional details can be seen that highlight the underlying Voronoi diagram of each frame along with parameters for Lloyd's algorithm. In this figure agents are represented by black dots, and their desired centroidal locations are represented by navy blue dots. The Voronoi partitions are represented by a collection of adjacent points sharing the same colour and the center of the risk density distribution (320, 240) is visualized as a bright red dot.

Fig 4.10 shows the relative paths travelled within the environment by each agent during the experimental implementation (blue) and the simulation (purple) for scenario 2. In this scenario the underlying risk density is represented by



(a) Frame 1 (initial frame)

(b) Frame 153

(c) Frame 307

(d) Frame 461 (final frame)

Figure 4.9: Underlying Voronoi diagram and Lloyd's algorithm parameters for sample frames from the simulation of scenario 2

$\phi(q) = \alpha e^{-\beta||d-q||^2}$ where $q \in \Omega$ and d is the coordinate (320, 240). The risk distribution is modelled by the magnitude of intensity of the orange color in Fig 4.10.

To build on Fig 4.10, the plot in Fig 4.11 shows the exact pixel length of the path taken by each agent in both the experimental implementation and the simulation of scenario 2.

The results in Fig 4.12 show consistency between the evolution of the coverage metric during the experimental implementation and the simulation. In both cases the coverage metric increases sharply at first and then levels out. Both curves also evolve over a similar range of time. The coverage metric in the simulated case evolves slightly more smoothly. Furthermore, in the simulated case the coverage metric maximizes to a slightly higher value with a longer run time. However, the important factor here is that the general behaviour of both coverage metric evolution curves is similar, and they both converge to maximum values that are relatively close to each other.

As both simulations start with the same initial conditions, the initial coverage

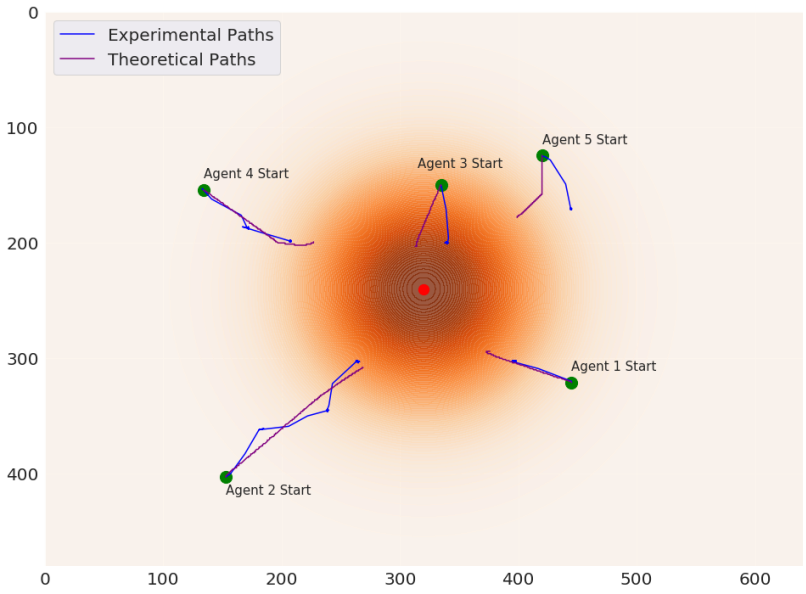


Figure 4.10: Paths travelled by each agent in the experimental implementation and the simulation of scenario 2

metric is 21874 in both cases. The experimental implementation converges to a final value of 27358. The simulation converges to a final value of 27681. Based on these values the change in metric coverage for each scenario is:

$$\Delta_{exp} = 27358 - 21874 = 5484$$
$$\Delta_{theo} = 27681 - 21874 = 5807$$

Given these values the percent difference between these values is found to be 5.72%:

$$\%\ \text{difference} = \frac{|\Delta_{theo} - \Delta_{exp}|}{\frac{\Delta_{theo} + \Delta_{exp}}{2}} * 100 = \frac{|5807 - 5484|}{\frac{5807 + 5484}{2}} * 100 = \frac{323}{5645.5} * 100 = 5.72\%$$

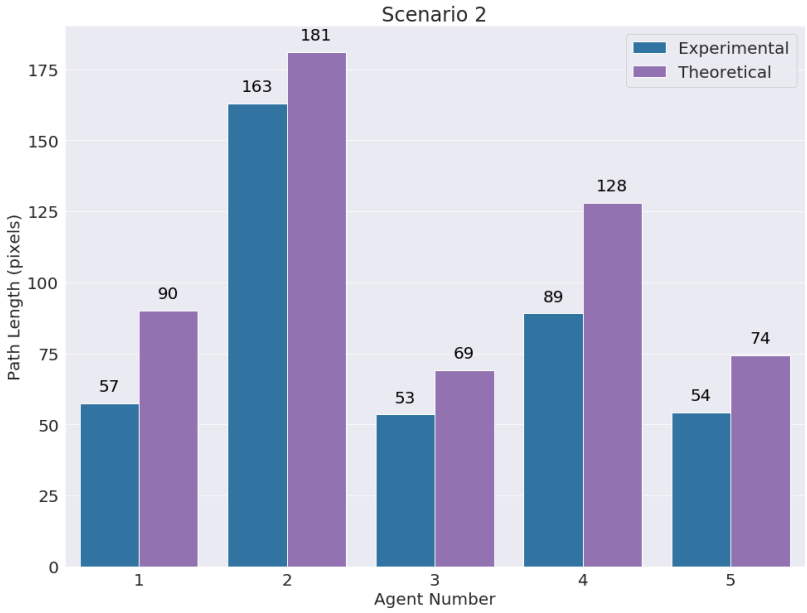


Figure 4.11: Lengths of paths travelled by each agent in the experimental implementation and the simulation for scenario 2

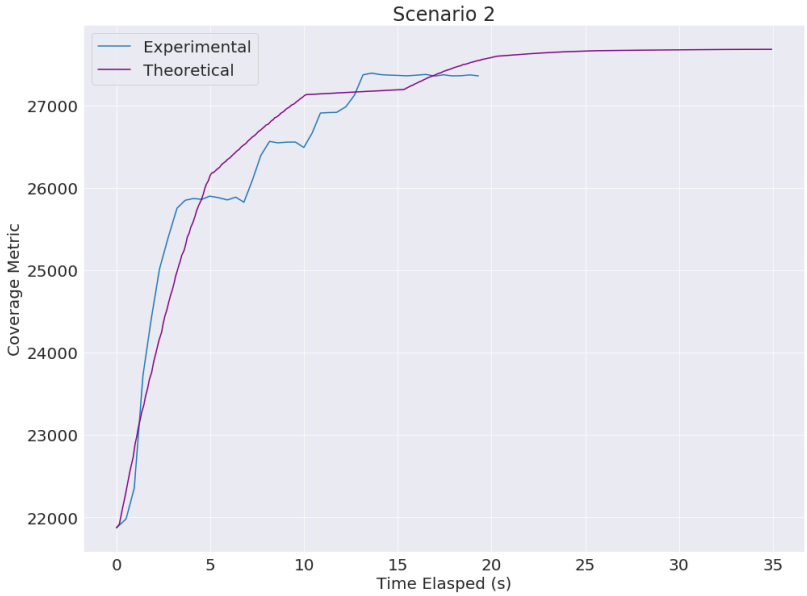
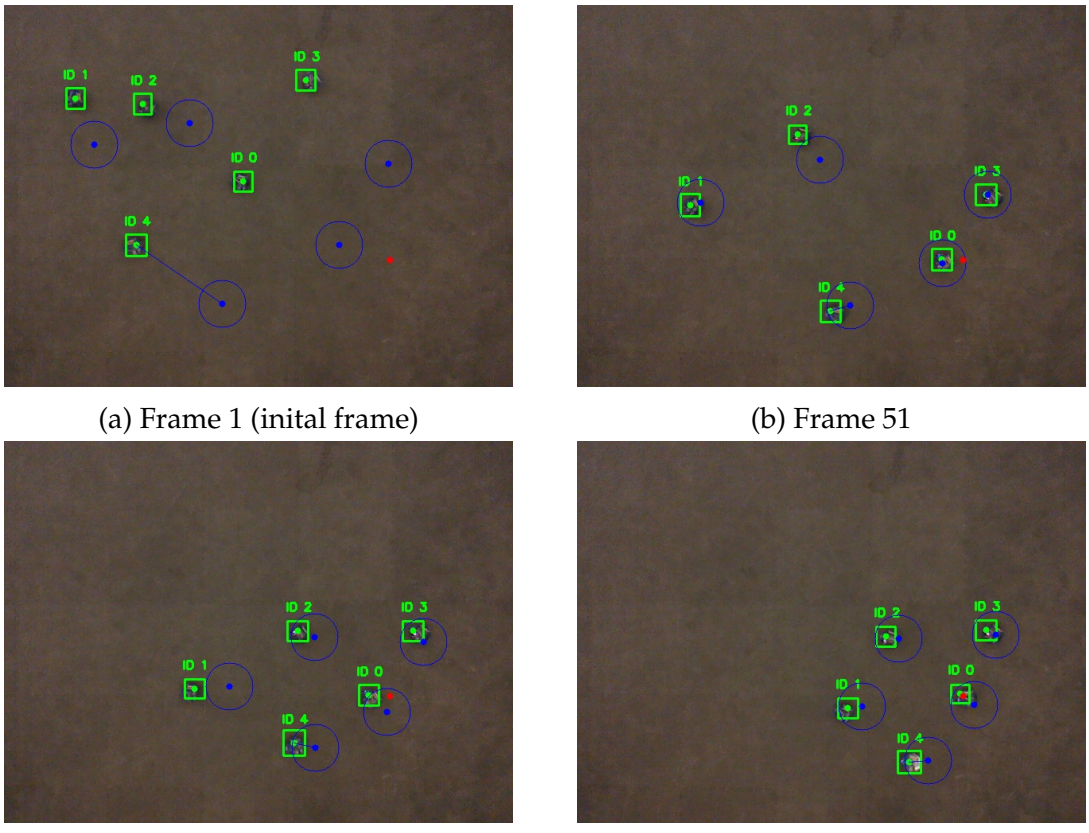


Figure 4.12: Evolution of the coverage metric during the experimental implementation and the simulation for scenario 2

4.1.3 Scenario 3: Time Invariant Nonuniform Risk Density Distribution Located Off-center of the Environment

In this scenario, area coverage control and optimization within a rectangular (640 by 480 pixel), convex environment, Ω, is implemented with five agents. This environment has a nonuniform time invariant risk density distribution represented by $\phi(q) = \alpha e^{-\beta||d-q||^2}$ where $q \in \Omega$ and d is the coordinate (485, 320). The parameters α and β are chosen as 1 and 0.0001 respectively. The randomized initial positions of the five agents were (300, 221), (89, 117), (174, 124), (379, 94) and (166, 301) respectively. These same initial positions were used in both the experimental implementation and the simulation for consistency.



(a) Frame 1 (inital frame)

(b) Frame 51

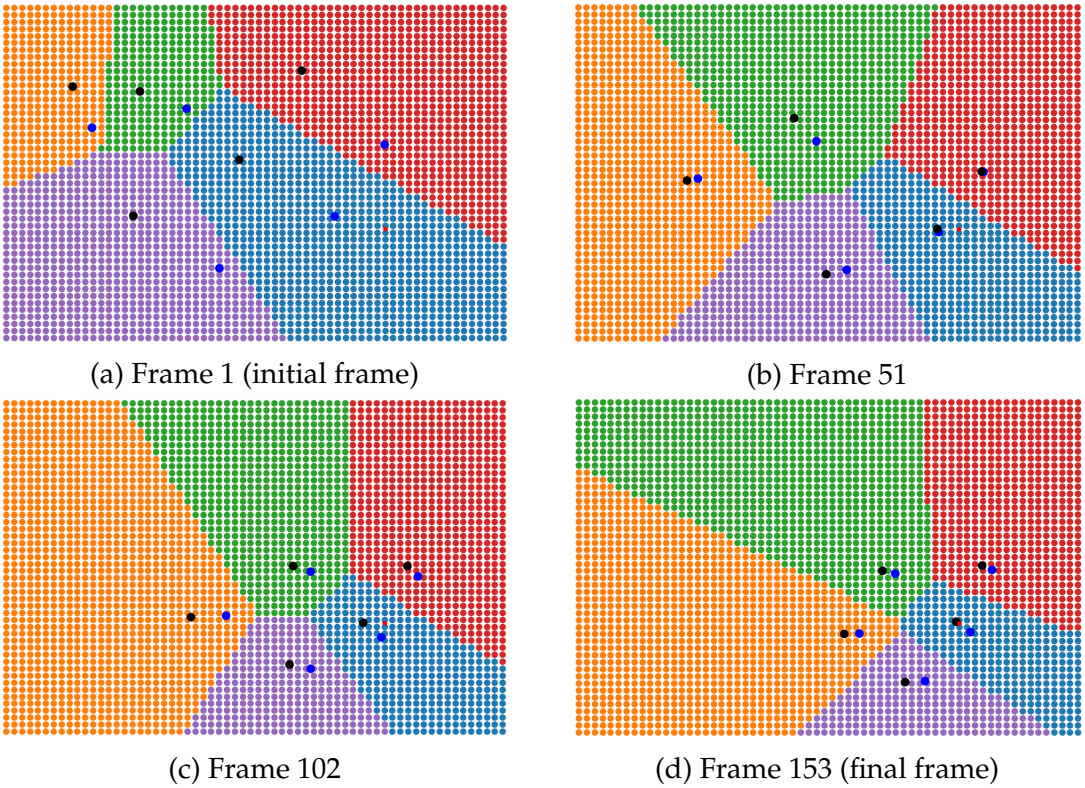
(c) Frame 102

(d) Frame 153 (final frame)

Figure 4.13: Sample frames from the experimental results of the implementation of scenario 3

Fig 4.13 shows sample frames from the experimental results of area coverage control and optimization using the proposed hardware test platform design for scenario 3.

Fig 4.14 provides additional details about the sample frames seen in Fig 4.13. In



(a) Frame 1 (initial frame)

(b) Frame 51

(c) Frame 102

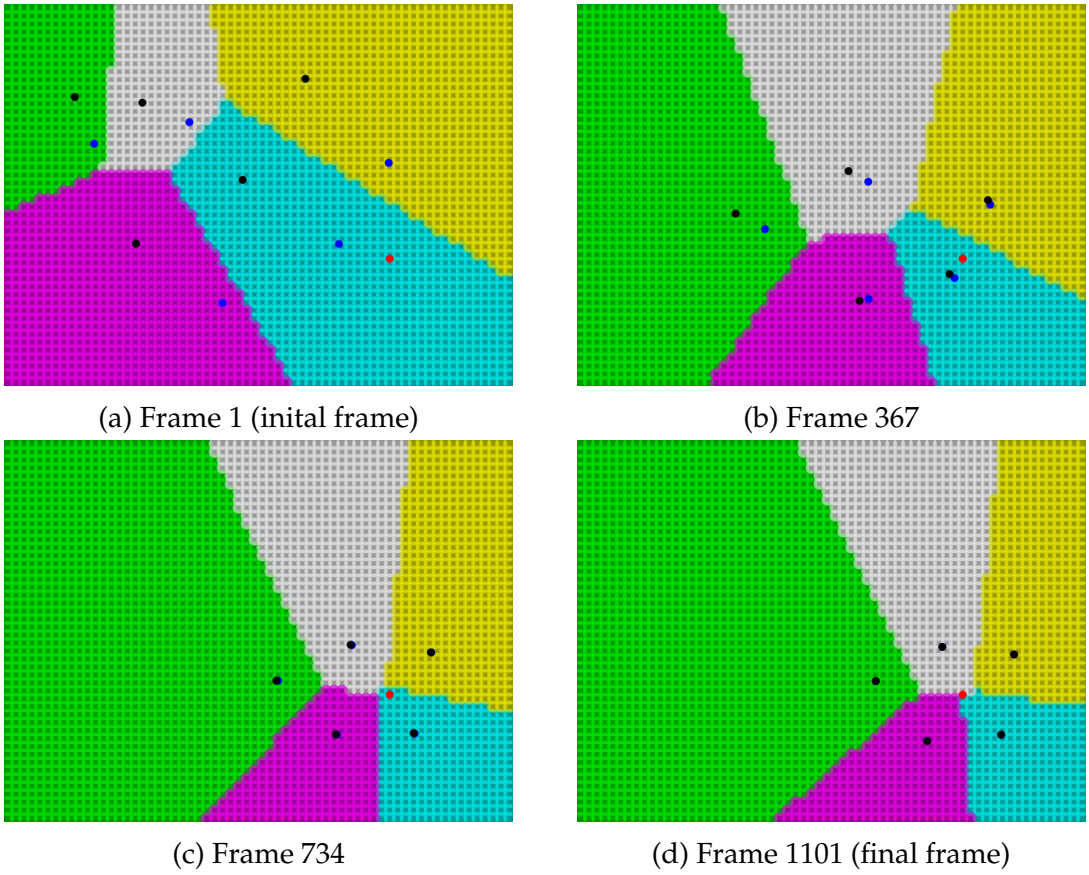
(d) Frame 153 (final frame)

Figure 4.14: Underlying Voronoi diagram and Lloyd's algorithm parameters for sample frames from the experimental implementation of scenario 3

Fig 4.14 we see the underlying Voronoi diagram of each frame along with parameters for Lloyd's algorithm. In this figure agents are represented by black dots, and their desired centroidal locations are represented by navy blue dots. The Voronoi partitions are represented by a collection of adjacent points sharing the same colour and the center of the risk density distribution (485, 320) is visualized as a bright red dot.

Fig 4.15 shows sample frames from the simulation of scenario 3. Additional details can be seen that highlight the underlying Voronoi diagram of each frame along with parameters for Lloyd's algorithm. In this figure agents are represented by black dots, and their desired centroidal locations are represented by navy blue dots. The Voronoi partitions are represented by a collection of adjacent points sharing the same colour and the center of the risk density distribution (485, 320) is visualized as a bright red dot.

Fig 4.16 shows the relative paths travelled within the environment by each agent during the experimental implementation (blue) and the simulation (purple) for scenario 3. In this scenario the underlying risk density is represented by



(a) Frame 1 (inital frame)

(b) Frame 367

(c) Frame 734

(d) Frame 1101 (final frame)

Figure 4.15: Underlying Voronoi diagram and Lloyd's algorithm parameters for sample frames from the simulation of scenario 3

$\phi(q) = \alpha e^{-\beta||d-q||^2}$ where $q \in \Omega$ and d is the coordinate (485, 320). The risk distribution is modelled by the magnitude of intensity of the orange color in Fig 4.16.

To build on Fig 4.16, the plot in Fig 4.17 shows the exact pixel length of the path taken by each agent in both the experimental implementation and the simulation of scenario 3.

The results in Fig 4.18 show consistency between the evolution of the coverage metric during the experimental implementation and the simulation. In both cases the coverage metric increases sharply at first and then levels out. Both curves also evolve over a similar range of time. The coverage metric in the simulated case evolves slightly more smoothly. Furthermore, in the simulated case the coverage metric maximizes to a slightly higher value with a longer run time. However, the important factor here is that the general behaviour of both coverage metric evolution curves is similar, and they both converge to maximum values that are relatively close to each other.

As both simulations start with the same initial conditions, the initial coverage

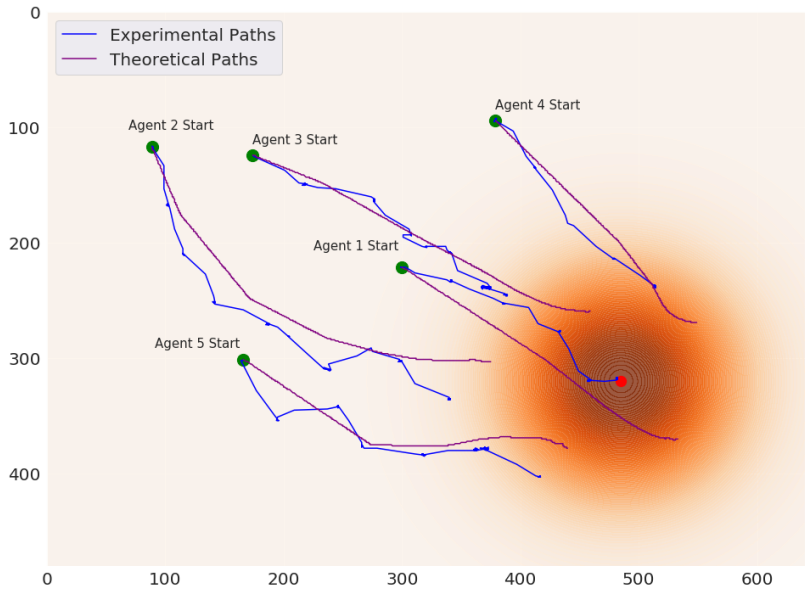


Figure 4.16: Paths travelled by each agent in the experimental implementation and the simulation of scenario 3

metric is 6054 in both cases. The experimental implementation converges to a final value of 25949. The simulation converges to a final value of 27283. Based on these values the change in metric coverage for each scenario is:

$$\Delta_{exp} = 25949 - 6054 = 19895$$
$$\Delta_{theo} = 27283 - 6054 = 21229$$

Given these values the percent difference between these values is found to be 6.49%:

$$\%\ \text{difference} = \frac{|\Delta_{theo} - \Delta_{exp}|}{\frac{\Delta_{theo} + \Delta_{exp}}{2}} * 100 = \frac{|21229 - 19895|}{\frac{21229 + 19895}{2}} * 100 = \frac{1334}{20562} * 100 = 6.49\%$$

Figure 4.17: Lengths of paths travelled by each agent in the experimental implementation and the simulation for scenario 3

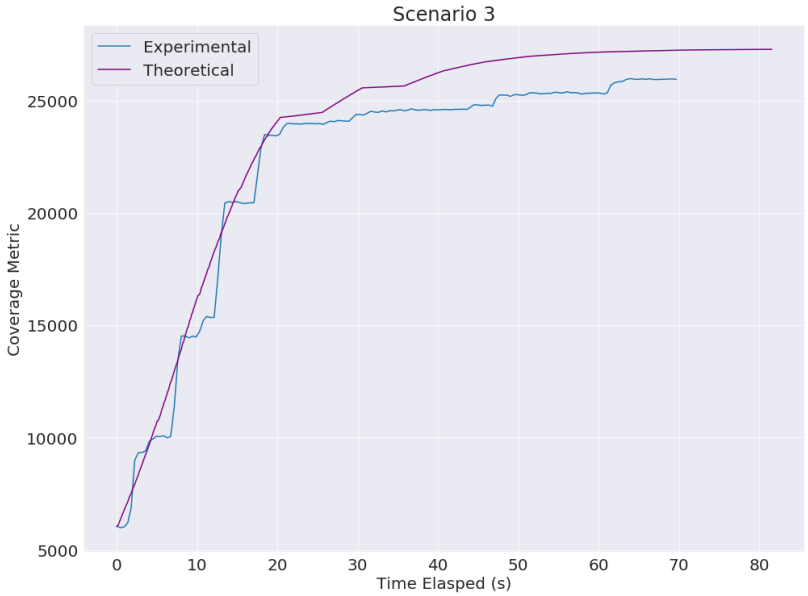


Figure 4.18: Evolution of the coverage metric during the experimental implementation and the simulation for scenario 3

4.2 Discussion

While typically we validate theory with experimentation, in this analysis we wanted to compare the experimental results to simulations of well researched area coverage control algorithms. As this thesis focused on the design of a hardware test platform for multi-agent algorithms, the objective here was to evaluate if the proposed hardware test platform design can plausibly reproduce behaviour that is consistent with theoretical predictions of area coverage control algorithms.

The simulation makes use of point agents and therefore simplifies the kinematics of robotic agents in the environment. Whereas the experimental implementation makes use of mechatronic robotic agents which introduces differential drive kinematics, hardware latency and additional hardware based uncertainties. Despite this, the behaviour of the point agents in the simulation is a good indicator of the theoretical expected behaviour of a system that is modelled by these area coverage control algorithms.

For these reasons, the simulations of area coverage control with time invariant risk distribution were used as benchmarks to evaluate the consistency of the experimental results.

4.2.1 Evolution of Coverage Metric

In the case of area coverage control and optimization, the coverage metric is a useful way to quantify how well an environment is being covered. In the three scenarios that were simulated, the results obtained from the simulation were used as benchmark values to evaluate if the experimental test platform could plausibly reproduce behaviour that was consistent with theoretical predictions of area coverage control algorithms.

In the three scenarios, the percent difference values between the final coverage metric between the experimental results and the simulation were 43.60%, 5.72% and 6.49% respectively. The percentage difference in the final coverage metric value for scenario 1 was relatively large compared to the low percent difference values in scenario 2 and 3. Given this analysis, the relatively low percent difference values in scenario 2 and 3 support the idea that the experimental test platform can plausibly reproduce behaviour that is consistent with theoretical predictions of area coverage control algorithms. With respect to scenario 1 in this analysis, the percent difference value of 43.60% is unsatisfactory. The high percent difference value appears to stem from the early convergence of the experimental implemen-

tation of scenario 1 when compared to the simulation of scenario 1. Some reasons are provided below relating to the precision of Alphabot2 locomotion during the experimental implementation to address the relatively early convergence of the experimental implementation.

Beyond just this analysis, the overall coverage metric evolution curves obtained from experimental results and from simulation (Fig 4.6, Fig 4.12 and Fig 4.18) showed similar behaviour in all three scenarios. In all three scenarios the curves would increase sharply and then level out to a maximum value in both the experimental results and the simulation. Furthermore, both experimental curves and simulated curves would evolve over a similar time range indicating that the proposed hardware test platform design can plausibly reproduce behaviour that is consistent with theoretical predictions of area coverage control algorithms.

4.2.2 Inconsistencies Between Simulation and Experiment

Some inconsistencies did exist between the experimental results and the simulation. When comparing the coverage metric curves, the simulated curves would typically be more smooth and would converge to higher values, indicating more optimal coverage. Considering that the simulations make use of point agents and have intrinsic assumptions about the kinematics of the robotic agents, it is expected that such discrepancies exist between experimental results and simulation.

With respect to the localization, inter-agent communication and locomotion systems embedded in the hardware test platform design, the following observations were made.

Throughout all three experimental implementations the object detection based localization framework performed well and was consistently able to detect and track all five agents in the environment. Similarly, no performance issues were found with the wireless communication protocol of the test platform. The appropriate controls actions were determined using area coverage control and optimization algorithms and packets were sent to the proper agents, with no packet drops.

After a robotic agent receives a packet containing a control action, the control action must then be executed. It is in this phase that issues were observed. As mentioned in Sections 3.5.3 and 3.5.4, an MPU6050 inertial measurement unit was used in conjunction with PID controllers to achieve guided locomotion. While this process was effective, it was not perfect. Certain factors such as MPU6050 measurement errors, variance in surface conditions within the environment, variance

in battery charge on the robotic agents and minor differences in electric flow on board the robotic agents still proved to be significant obstacles for precise locomotion.

With respect to precise rotation, the robots would rotate within a range of ± 5 degrees of the target orientation. While this error does not seem significant, it can be enough to cause the robots to take less than optimal paths towards their desired positions.

With respect to straight line motion, due to the factors mentioned above, it was difficult to ensure that the straight line motion of the robot was truly straight. While the motion was generally straight, any minor errors had an impact on the trajectory of the robot and effectively stopped the robot from following optimal paths.

Furthermore, due to the same factors mentioned above, it proved to be difficult to calculate exactly how far straight a robot would travel within a given time constraint. In a basic sense, it was difficult to get the robot to move a specified number of units forward in real time as the robot localization framework was so decoupled from the on board locomotion system. This meant that often times when a robot approached its desired position, it would overshoot and end up going past its desired position.

4.2.3 Relaxed Criteria of Arrival

Due to all the previously mentioned issues with precise locomotion, a less precise criteria had to be established for measuring when a robot had arrived at its desired position. If a robot arrived within a 30 pixel radius of its desired position, then it was considered to have arrived. This adjustment proved essential for the experimental implementation to run smoothly without many redundant movements by the robots.

This relaxed arrival criteria is hypothesized to be a major contributing factor to the discrepancies between the coverage metric evolution curves seen in experimental implementation and simulation, which are evident in all three scenarios but are more pronounced in scenario 1. Since the robotic agents had a difficult time travelling to the precise desired positions, they ended up taking less optimal paths towards optimal coverage and were not able to arrive at the required positions to further increase the coverage metric.

This can be visualized in the results as well. In the final frame of each experi-

mental implementation (Figures 4.1d, 4.7d and 4.13d), it can be seen that the robots have final positions that are within 30 pixel radius of their desired position, but not precisely at their desired positions. This means the final configuration of these robots is close to optimal, but it is not exactly optimal. Essentially the test platform was able to follow Lloyd's algorithm quite well, but not optimally, due to relaxed criteria of arrival and convergence. This means that additional improvements can be made to the test platform in the future. This was more evident in the experimental implementation of scenario 1, where the agents arrived within 30 pixels of their respective generalized centroid after limited movement.

4.2.4 Agent Trajectories

Figures 4.4, 4.10 and 4.16 provide a visual comparison of the trajectory of the agents in the experimental implementation and the simulation. In general across the three scenarios, it can be seen that the agents in the simulation have relatively smooth and directed trajectories, whereas the experimental trajectories are more jagged and less directed. This behaviour is expected as the experimental implementation is subject to more realistic environmental conditions, agent kinematics and hardware limitations. Even though the experimental implementation of scenario 1 converged relatively quickly when compared to its simulated counterpart, the movement of the agents in this implementation was consistent with simulation until convergence of the experimental implementation. The general similarity between the paths taken by agents in the experimental implementation and the simulations is a good indicator that the hardware test platform can plausibly reproduce behaviour that is consistent with theoretical predictions of area coverage control algorithms. This supports the overall usability of the proposed hardware test platform design for general purpose multi-agent algorithm testing.

When comparing the specific path length in pixels of each agent throughout all 3 scenarios (Figures 4.5, 4.11 and 4.17), it can be seen that the length of the paths taken by agents in the experimental implementation are generally equal to or less than the length of their respective counterparts in the simulation. This makes sense as the simulations ran longer and the agents in these simulations travelled further in order to converge to more optimal arrangements. Situations where the agents in the experimental implementation had path lengths equal to or greater than their respective simulated counterparts highlight the minor inefficiencies and precise locomotion obstacles present in the experimental setup.

4.2.5 General Findings

In general the results indicate that the proposed design for the hardware test platform can plausibly implement this class of multi-agent motion control algorithms consistently with an established theoretical framework used to predict the response of this class of systems. In all three scenarios the coverage metric was maximized consistently with simulations. The paths taken by agents in the experimental implementation were similar to paths taken by agents in the simulation with respect to overall direction and intent to maximize coverage metric. The test platform was able to compute the appropriate area coverage control and optimization algorithms and the robots were able to follow the control actions they received. Features such as localization of the agents and wireless communication within the test platform worked as intended.

Theoretical research on algorithms in the domain of area coverage control with time invariant risk distribution is quite mature. The hardware test platform was able to show that it can plausibly reproduce behaviour that is consistent with theoretical predictions of this class of algorithms. This indicates that the proposed hardware test platform design has usability in this domain.

Some sources of error were identified with respect to the precision of the locomotion of the robotic agents. This is an area of the test platform that could use additional work in the future. The experimental implementation could also be sped up by decreasing computational time. Beyond that, the test platform could be further validated with more complex area coverage control and optimization scenarios such as scenarios including non-convex environments, time variant density distributions or stochastic intermittent communication between agents [9, 10, 35].

Chapter 5

Summary and Conclusions

Wireless sensor networks in the form of multi-robot networks have applications in a variety of military and civilian tasks such as harbour protection, perimeter surveillance, search & rescue missions and cooperative estimation, among others. In order to develop functional multi-robot networks to achieve these tasks, a combination of theoretical and experimental work is required. Theoretical research aims to model the open and closed loop dynamics of multi-robot networks and to develop corresponding control algorithms for tackling the previously mentioned tasks. Experimental work focuses on the hardware or simulated implementations to test and evaluate the algorithms' performance.

As research in the field of multi-robot networks gains more momentum and a variety of theoretical work is being developed, a need for experimental validation of this theoretical work becomes apparent. As such, this thesis contributed a design for a hardware test platform for evaluating and implementing algorithms in the realm of multi-robot networks.

The design consists of a planar rectangular work space (3.2m by 2.4m) that is populated by mechatronic agents in the form of Alphabot2 robots that have a diameter of 0.11m. The proposed design leverages machine learning based object detection algorithms to provide a method of localization for the mechatronic agents. A ZigBee based protocol referred to as XBee is used to create a meshed wireless communication architecture between the robotic agents on the test platform. The various benefits of the XBee protocol are that it requires low power consumption, has high range and is relatively cost effective. An MPU6050 inertial measurement unit is embedded onto each Alphabot2 agent to allow for collection of yaw information for each agent. This information is used in conjunction with PID controllers to allow for more precise locomotion for the robotic agents. On board sensors on

the Alphabot2 agents include an ultrasonic sensor and two infrared sensors for proximity measurement and they come with a line tracking module as well.

The proposed design for the test platform introduces an intentionally cooperative type of multi-robot network that is capable of strong cooperation. In terms of the architecture of the test platform, the proposed design presents a hybrid approach that has a central unit along with minor decentralized capabilities. This means that some control actions are computed locally by the agents themselves (obstacle avoidance), while other actions are provided by the central unit. Ongoing research is being dedicated to implement decentralized architectures by making use of inter-agent communication capabilities provided by the XBee protocol.

In order to evaluate if the hardware test platform design can plausibly reproduce behaviour that is consistent with theoretical predictions of area coverage control algorithms, it was tested on a class of area coverage control and optimization algorithms. More specifically, Lloyd's algorithm was implemented for area coverage control and optimization in an environment with time invariant risk distribution. Cases involving uniform risk distribution and nonuniform risk distribution were both tested. A simulation of these algorithms was run, and similarly an experimental implementation of these algorithms was run using the proposed design for the test platform. Three scenarios were tested in this class of algorithms and the behaviour of the agents in the test platform was consistent with the behaviour of the agents in the simulation.

In all three experimental scenarios, the test platform was able to compute the appropriate area coverage control and optimization algorithms and the robots were able to follow the control actions they received. The localization framework and wireless communication protocol both worked effectively with no issues.

In scenarios 2 and 3 the coverage metric in the experimental implementation was maximized consistently with simulations, over similar ranges of time. In these two scenarios, the percent difference values between the final coverage metric between the experimental results and the simulation were 5.72% and 6.49% respectively. The paths taken by agents in the experimental implementation for these two scenarios were similar to paths taken by agents in the simulation with respect to overall direction and intent to maximize the coverage metric. In scenario 1, the behaviour of the test platform was similar to simulation, however the percent difference value between the final coverage metric between the experimental results and the simulation was 43.60%. This was relatively larger than the percent difference value between the same metric in scenarios 2 and 3. Issues related to precise

locomotion of the Alphabot2 agents were hypothesized to be a contributing factor for this larger percent difference value between the final coverage metric values for the experimental implementation and the simulation for scenario 1.

These precise locomotion issues were caused by certain factors such as MPU6050 measurement errors, variance in surface conditions within the environment, variance in battery charge on the robotic agents and minor differences in electric flow on board the robotic agents. Precise locomotion is a component of the test platform that could use additional work in the future to improve the precision of the experimental test platform.

In general the results indicated that the proposed design for the hardware test platform could implement this class of multi-agent motion control algorithms consistently with an established theoretical framework used to predict the response of this class of systems. Overall, the proposed hardware test platform design is a first step for my research team towards evaluating and implementing complex multi-robot network algorithms in a real time hardware based setting. Future work includes:

- Decreasing computational time required to speed up convergence of test platform.
- Exploring additional scenarios such as non-convex environments, time variant density distributions or stochastic intermittent communication between agents.
- Improving precise locomotion capabilities of Alphabot2 agents.
- Exploring the implementation of a decentralized architecture for the hardware test platform by making use of inter-agent communication capabilities, migrating some computation to the local level of the robots and by improving environmental sensing capabilities of the robots at the local level.
- Testing on different multi-agent control and estimation algorithms and evaluate versatility and robustness of the test platform.

Appendix A

Robot Localization and Control Code in Python

The main code in Python is split into two parts. The first file is the main "Run.py" file. The second file is the "Lib.py" which contains all the utility classes and functions used inside the "Run.py" file.

A.1 Main Run.py File

The source code can be found at the following link: https://github.com/Sarmyt/masters_python_code/blob/master/Run.py.

Below is a snapshot of the code to support the thesis submission:

A.1.1 Pre-initialization

```
# Dependent libraries

import Lib

# Importing utility libraries

import numpy as np
import tensorflow as tf
import cv2
import math
import time
```

```
import random
import msvcrt
import argparse
import os
from matplotlib import pyplot as plt
import shutil
from utils import label_map_util

# Importing XBee libraries for wireless communication with
    Alphabot2 robots

from digi.xbee.devices import XBeeDevice
from digi.xbee.devices import RemoteXBeeDevice
from digi.xbee.devices import XBee64BitAddress

# Defining test configuration, if true then code is run on sample
    video for testing purposes

test = False

# Defining risk density option and parameters

option = 3 # Define which area coverage scenario is being tested
    (possible values include 1, 2 or 3)
alpha = 1
beta = 0.0001

assert option == 1 or option == 2 or option == 3

# Ensure that data collection directory exists, reset data
    collection for new run

if not os.path.exists("data"):
    os.mkdir("data")

if not os.path.exists("data/option{}".format(option)):
    os.mkdir("data/option{}".format(option))
```

```
if os.path.exists("data/option{}/frames".format(option)):
    shutil.rmtree('data/option{}/frames'.format(option))

if os.path.exists("data/option{}/plots".format(option)):
    shutil.rmtree('data/option{}/plots'.format(option))

os.mkdir('data/option{}/frames'.format(option))
os.mkdir('data/option{}/plots'.format(option))

# Defining paths to label map and frozen inference graph for
   object detection

PATH_TO_FROZEN_GRAPH = 'frozen_inference_graph.pb'
PATH_TO_LABELS = 'label_map.pbtxt'

# Variable assignment for pre-initialization phase

iterator = 0

if not test:

    # Define Controller Device

    device = XBeeDevice("COM4", 9600) # Pick your specific port

    # Open Controller Device

    device.open()

    # Define Remote Devices

    remote_devices = []

    remote_devices.append(RemoteXBeeDevice(device,
        XBee64BitAddress.from_hex_string("0013A20040D835D1")))
    remote_devices.append(RemoteXBeeDevice(device,
        XBee64BitAddress.from_hex_string("0013A200418FE7A3")))
    remote_devices.append(RemoteXBeeDevice(device,
```

```
            XBee64BitAddress.from_hex_string("0013A200415E8441")))
        remote_devices.append(RemoteXBeeDevice(device,
            XBee64BitAddress.from_hex_string("0013A2004190B158")))
        remote_devices.append(RemoteXBeeDevice(device,
            XBee64BitAddress.from_hex_string("0013A200417CA46A")))

else:
    remote_devices = [[0], [1]]

# Variable assignment for initialization phase

positionInit = [] # Initialization position variable
changePosition = []
distanceInit = [] # Initialization distance variable
idOrder = [] # For matching XBee id to object detection id
orientationInit = [] # Initialization orientation variable

# Variable assignment for main run loop phase

iterations = 0
frame_count = 0

edgeTimer = []
timer = []

pass_var = [] # Indicates if this agent should be passed this
    iteration
isSuccess = [] # Indicates if desired position has been achieved
isEdge = [] # Indicates if agent is near an edge
reset = []
distanceDes = [] # Real time distance between agent and desired
    position

currentPosition = [] # Current position of agent
prevPosition = [] # Previous position of agent
timestepDistance = [] # Distance between current position and
    previous position of agent
```

```
orientation = [] # Current orientation of each agent
orientationDes = [] # Current orientation of each agent
angle = [] # Angle from current orientation, to optimal
    orientation
messageArray = [] # The message to be sent to each agent

for i in range(len(remote_devices)):

    # Initialization variable assignment

    positionInit.append(None) # Initialization position variable
    changePosition.append(None)
    distanceInit.append(0.0) # Initialization distance variable
    idOrder.append(None) # For matching XBee id to object
        detection id
    orientationInit.append(None) # Initialization orientation
        variable

    # Run loop variable assignment

    edgeTimer.append(time.time())
    timer.append(time.time())

    pass_var.append(False) # Indicates if this agent should be
        passed this iteration
    isSuccess.append(False) # Indicates if desired position has
        been achieved
    isEdge.append(False) # Indicates if agent is near an edge
    reset.append(True)
    distanceDes.append(None) # Real time distance between agent
        and desired position

    currentPosition.append(None) # Current position of agent
    prevPosition.append(None) # Previous position of agent
    timestepDistance.append(None) # Distance between current
        position and previous position of agent

    orientation.append(None) # Current orientation of each agent
```

```
        orientationDes.append(None) # Current orientation of each
            agent
        angle.append(None) # Angle from current orientation, to
            optimal orientation
        messageArray.append(None) # The message to be sent to each
            agent

# Define centroid tracker object

tracker = Lib.Tracker()

# Define object for Voronoi diagram and Lloyds algorithm
    calculations

voronoi = Lib.Voronoi()

# Define object for utility functions

function = Lib.Function()

# Define frozen inference graph from path

detection_graph = tf.Graph()
with detection_graph.as_default():
  od_graph_def = tf.compat.v1.GraphDef()
  with tf.io.gfile.GFile(PATH_TO_FROZEN_GRAPH, 'rb') as fid:
    serialized_graph = fid.read()
    od_graph_def.ParseFromString(serialized_graph)
    tf.import_graph_def(od_graph_def, name='')

# Define label map from path

category_index = label_map_util.
    create_category_index_from_labelmap(PATH_TO_LABELS,
    use_display_name=True)

# Start video capture from webcam, or from "Test.mp4" video is
    testing code
```

```
if test:
    vidcap = cv2.VideoCapture("Test.mp4")
else:
    vidcap = cv2.VideoCapture(0)

success, frame = vidcap.read()

# Define the discretized space

width_sections = 64
height_sections = 48

xdim = np.linspace(5, frame.shape[1] - 5, width_sections)
ydim = np.linspace(5, frame.shape[0] - 5, height_sections)
X, Y = np.meshgrid(xdim,ydim)

# Define map of risk density based on selected option

if option == 1:
    risk = np.ones((height_sections, width_sections))

elif option == 2:
    target = [frame.shape[1]/2, frame.shape[0]/2]
    risk = voronoi.risk_density(xdim, ydim, target, alpha=alpha,
       beta=beta)
    print(target)

elif option == 3:
    target = [485, 320]
    risk = voronoi.risk_density(xdim, ydim, target, alpha=alpha,
       beta=beta)
    print(target)

if not test:

    # Start pre-initialization loop
```

```
while True:
    success, frame = vidcap.read()

    if (success):
        cv2.imshow("Frame", frame)
        cv2.waitKey(1)

    if msvcrt.kbhit(): # Record user input from keyboard

        key = msvcrt.getch()
        print(key)

        if key == b'w':
            message = "P,F>" # If you press "w" the robot
                will go forward

        if key == b'a':
            message = "P,90>" # If you press "a" the robot
                will rotate 90 degrees counter-clockwise

        if key == b's':
            message = "P,B>"  # If you press "s" the robot
                will go backwards

        if key == b'd':
            message = "P,270>" # If you press "d" the robot
                will rotate 90 degrees clockwise

        if key == b'z':
            message = "P,S>" # If you press "z" the robot
                will stand still

        if key == b'q':
            iterator = iterator - 1 # If you press "q", you
                will take control of the previous robot
            message = "P,S>"

            if (iterator < 0):
```

```
                iterator = 0

            if key == b'e':
                iterator = iterator + 1 # If you press "e", you
                    will take control of the next robot
                message = "P,S>"

            if key == b'x': # If you press "x" the pre-
                initialization loop will end
                break

            if (iterator > (len(remote_devices) - 1)):
                break

            device.send_data_async(remote_devices[iterator],
                message)
            print(message)
```

A.1.2 Initialization

```
# Start object detection

with detection_graph.as_default() as graph:
    with tf.compat.v1.Session() as sess:

            if not test:

    # Start initialization loop

                while True:

                    for i in range(0, len(remote_devices)):

                        success, frame = vidcap.read()

                        frame_expanded = np.expand_dims(frame,
                            axis=0)

                        output_dict = function.
```

```
            run_inference_for_single_image(
            frame_expanded, graph, sess) #
            Detect Alphabot2 in the frame

        rects = []

        # Draw rectangles around all Alphabot2
            robots detected

        for j in range(0, len(remote_devices)):
          if output_dict['detection_scores'][j]
              > 0.85:

                (startY, startX, endY, endX) =
                    output_dict['detection_boxes
                    '][j]

                rects.append([int(startX*frame.
                    shape[1]), int(startY*frame.
                    shape[0]), int(endX*frame.
                    shape[1]), int(endY*frame.
                    shape[0])])

                cv2.rectangle(frame, (int(
                    startX*frame.shape[1]), int(
                    startY*frame.shape[0])), (
                    int(endX*frame.shape[1]),
                    int(endY*frame.shape[0])),
                    (0, 255, 0), 1)

        objects = tracker.update(rects) # Begin
            tracking all Alphabot2 robots that
            are detected

        for (objectID, centroid) in objects.
            items():
             text = "ID␣{}".format(objectID)
             cv2.putText(frame, text, (centroid
```

```
                [0] - 15, centroid[1] - 25),
                 cv2.FONT_HERSHEY_SIMPLEX, 0.5,
                    (0, 255, 0), 2)
            cv2.circle(frame, (centroid[0],
                centroid[1]), 4, (0, 255, 0),
                -1)
            positionInit[objectID] = centroid

    cv2.imshow("Frame", frame)
    cv2.waitKey(1)

    device.send_data_async(remote_devices[i
        ], "P,F>") # Send command for robot
        to move forward

    for j in range(0, 35):
        success, frame = vidcap.read()

        frame_expanded = np.expand_dims(
            frame, axis=0)

        output_dict = function.
            run_inference_for_single_image(
            frame_expanded, graph, sess)

        rects = []

        for k in range(0, len(
            remote_devices)):
          if output_dict['detection_scores'
              ][k] > 0.85:

                (startY, startX, endY, endX
                    ) = output_dict['
                    detection_boxes'][k]

                rects.append([int(startX*
                    frame.shape[1]), int(
```

```
                    startY*frame.shape[0]),
                    int(endX*frame.shape[1])
                    , int(endY*frame.shape
                    [0])])

                cv2.rectangle(frame, (int(
                    startX*frame.shape[1]),
                    int(startY*frame.shape
                    [0])), (int(endX*frame.
                    shape[1]), int(endY*
                    frame.shape[0])),
                    (0, 255, 0), 1)

            objects = tracker.update(rects)

            for (objectID, centroid) in objects
                .items():
                text = "ID {}".format(objectID)
                cv2.putText(frame, text, (
                    centroid[0] - 15, centroid
                    [1] - 25),
                     cv2.FONT_HERSHEY_SIMPLEX,
                        0.5, (0, 255, 0), 2)
                cv2.circle(frame, (centroid[0],
                     centroid[1]), 4, (0, 255,
                    0), -1)
                changePosition[objectID] =
                    centroid
                distanceInit[objectID] =
                    function.get_distance(point1
                    =positionInit[objectID],
                    point2=changePosition[
                    objectID])

            cv2.imshow("Frame", frame)
            cv2.waitKey(1)

        idOrder[i] = np.argmax(distanceInit) #
```

```
            Record ID of robot that moved
            forward

        device.send_data_async(remote_devices[i
            ], "P,B>")

        for j in range(0, 35):
            success, frame = vidcap.read()

            frame_expanded = np.expand_dims(
                frame, axis=0)

            output_dict = function.
                run_inference_for_single_image(
                frame_expanded, graph, sess)

            rects = []

            for k in range(0, len(
                remote_devices)):
              if output_dict['detection_scores'
                  ][k] > 0.85:

                    (startY, startX, endY, endX
                        ) = output_dict['
                        detection_boxes'][k]

                    rects.append([int(startX*
                        frame.shape[1]), int(
                        startY*frame.shape[0]),
                        int(endX*frame.shape[1])
                        , int(endY*frame.shape
                        [0])])

                    cv2.rectangle(frame, (int(
                        startX*frame.shape[1]),
                        int(startY*frame.shape
                        [0])), (int(endX*frame.
```

```
                                shape[1]), int(endY*
                                frame.shape[0])),
                                (0, 255, 0), 1)

                        objects = tracker.update(rects)

                        for (objectID, centroid) in objects
                            .items():
                            text = "ID␣{}".format(objectID)
                            cv2.putText(frame, text, (
                                centroid[0] - 15, centroid
                                [1] - 25),
                                 cv2.FONT_HERSHEY_SIMPLEX,
                                    0.5, (0, 255, 0), 2)
                            cv2.circle(frame, (centroid[0],
                                 centroid[1]), 4, (0, 255,
                                0), -1)
                            positionInit[objectID] =
                                centroid

                        cv2.imshow("Frame", frame)
                        cv2.waitKey(1)

                    orientationInit[idOrder[i]] = function.
                        get_orientation(point1=positionInit[
                        idOrder[i]], point2=changePosition[
                        idOrder[i]]) # Record initial
                        orientation of robot

                    device.send_data_async(remote_devices[i
                        ], "I," + str(orientationInit[
                        idOrder[i]]) + ">")


    # Check user input

                    print(idOrder)
```

```
test1 = function.test_1(idArray=idOrder,
    numRobots=len(remote_devices)) # Test 1
    that all robots were detected and
    initialized correctly

test2 = function.test_2(idArray=idOrder,
    numRobots=len(remote_devices)) # Test 2
    that all robots were detected and
    initialized correctly

if test1:
    print("Test 1 passed.")

if not test1:
    print("Test 1 failed.")

if test2:
    print("Test 2 passed.")

if  not test2:
    print("Test 2 failed.")

success, frame = vidcap.read()

frame_expanded = np.expand_dims(frame, axis
    =0)

output_dict = function.
    run_inference_for_single_image(
    frame_expanded, graph, sess)

rects = []

for i in range(0, len(remote_devices)):
    if output_dict['detection_scores'][i] >
        0.85:

        (startY, startX, endY, endX) =
```

```
                output_dict['detection_boxes'][i]

            rects.append([int(startX*frame.shape
                [1]), int(startY*frame.shape[0]),
                int(endX*frame.shape[1]), int(endY
                *frame.shape[0])])

            cv2.rectangle(frame, (int(startX*
                frame.shape[1]), int(startY*frame.
                shape[0])), (int(endX*frame.shape
                [1]), int(endY*frame.shape[0])),
                        (0, 255, 0), 1)

    objects = tracker.update(rects)

    for (objectID, centroid) in objects.items():
        text = "ID {}".format(objectID)
        cv2.putText(frame, text, (centroid[0]
            - 15, centroid[1] - 25),
                cv2.FONT_HERSHEY_SIMPLEX,
                    0.5, (0, 255, 0), 2)
        cv2.putText(frame, str(
            orientationInit[objectID]), (
            centroid[0] - 15, centroid[1] -
            50),
                cv2.FONT_HERSHEY_SIMPLEX,
                    0.5, (0, 255, 0), 2)
        cv2.circle(frame, (centroid[0],
            centroid[1]), 4, (0, 255, 0), -1)
        cv2.arrowedLine(frame, (centroid[0],
            centroid[1]), (changePosition[
            objectID][0], changePosition[
            objectID][1]), (255, 0, 0), 1)
        cv2.line(frame, (centroid[0],
            centroid[1]), (centroid[0] + 30,
            centroid[1]), (255, 0, 0), 1)

    cv2.imshow("Frame", frame)
```

```
            cv2.waitKey(1)

            inputVar = input("Are you satisfied with the
               initialization? (y/n)") # Ask for user
               input that initialization results make
               sense

            if (inputVar == 'y'):
                break

            if cv2.waitKey(1) & 0xFF == ord("x"):
                break
```

A.1.3 Run Loop

```
# Start run loop

        idOrder_array = np.array(idOrder)

        data_txt = open("data/option{}/data.txt".format(
           option), "w") # Begin saving data to text file

        while True:

          success, frame = vidcap.read()

          frame_expanded = np.expand_dims(frame, axis=0)

          output_dict = function.
             run_inference_for_single_image(frame_expanded,
             graph, sess)

          rects = []

          for i in range(0, len(remote_devices)):
            if output_dict['detection_scores'][i] > 0.85:

                (startY, startX, endY, endX) = output_dict[
                   'detection_boxes'][i]
```

```
                    rects.append([int(startX*frame.shape[1]),
                        int(startY*frame.shape[0]), int(endX*
                        frame.shape[1]), int(endY*frame.shape
                        [0])])

                    cv2.rectangle(frame, (int(startX*frame.
                        shape[1]), int(startY*frame.shape[0])),
                        (int(endX*frame.shape[1]), int(endY*
                        frame.shape[0])),
                        (0, 255, 0), 2)

                objects = tracker.update(rects)

                agentPosition = []

                for (objectID, centroid) in objects.items():
                    text = "ID {}".format(objectID)
                    cv2.putText(frame, text, (centroid[0] - 15,
                        centroid[1] - 25),
                         cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0),
                             2)
                    cv2.circle(frame, (centroid[0], centroid[1]),
                        4, (0, 255, 0), -1)
                    agentPosition.append([centroid[0], centroid
                        [1]])

# Get desired location of each robot

                distance_update, voronoi_update, index_update =
                    voronoi.voronoi(agentPosition=agentPosition,
                    numAgents=len(remote_devices), xdim=xdim, ydim=
                    ydim) # Calculate Voronoi diagram given agent
                    positions

                if np.sum(isSuccess) == len(remote_devices) or np.
                    sum(reset) == len(remote_devices): # If all
                    robots have arrived at desired location, or this
```

```
            is the first run through of the loop

            iterations = iterations + 1

            desPosition = []

            for n in range(len(remote_devices)):
                desPosition.append(voronoi.centroid(
                    voronoi_update[n], index_update[n], risk
                    , agentPosition[n])) # Find centroid of
                    each Voronoi partition
                isSuccess[n] = False

        coverage = 0

        for i in range(len(remote_devices)):
            cv2.circle(frame, (int(desPosition[i][0]), int(
                desPosition[i][1])), 4, (255, 0, 0), -1)
            cv2.circle(frame, (int(desPosition[i][0]), int(
                desPosition[i][1])), 30, (255, 0, 0), 1)
            coverage += voronoi.coverage_metric(
                agentPosition[i], voronoi_update[i],
                index_update[i], risk) # Calculate coverage
                metric at current time

        # Write data to text file

        if np.sum(reset) == len(remote_devices):
            start_time = time.time()
            data_txt.write("Time: {}, Coverage Metric: {},
                Iteration: {}, Agent Positions: {}, Desired
                Positions: {}".format(0, coverage,
                iterations, agentPosition, desPosition))
        else:
            data_txt.write("Time: {}, Coverage Metric: {},
                Iteration: {}, Agent Positions: {}, Desired
                Positions: {}".format(time.time() -
                start_time, coverage, iterations,
```

```
                agentPosition, desPosition))
            data_txt.write("\n")

            for agent in range(len(remote_devices)):

    # Check distance from desired location

                distanceDes[agent] = function.get_distance(
                    point1=desPosition[agent], point2=
                    agentPosition[agent]) # Check distance from
                    an agent to its desired position

                if (distanceDes[agent] > 30.0): # If distance
                    is more than 30 pixels, then the agent has
                    not arrived
                        isSuccess[agent] = False

                if (distanceDes[agent] <= 30.0 and isSuccess[
                    agent] == False): # If distance is less than
                     30 pixels, then the agent has arrived
                        if (reset[agent] == True) and not
                            test:
                             index = np.where(idOrder_array==
                                agent)
                             device.send_data_async(
                                remote_devices[index[0][0]], "
                                A," + str(orientationInit[
                                agent]) + ">")
                        if (reset[agent] == False) and not
                            test:
                             index = np.where(idOrder_array==
                                agent)
                             device.send_data_async(
                                remote_devices[index[0][0]], "
                                A," + str(orientation[agent])
                                + ">")

                        isSuccess[agent] = True
```

```
                    pass_var[agent] = True


    # Check edge conditions, if robot is at the edge of the
       boundary then a control action is sent to make it turn
       around

                if (((time.time() - edgeTimer[agent]) >= 5) and
                    ((agentPosition[agent][0] <= frame.shape
                   [1]*0.01) or (agentPosition[agent][0] >=
                   frame.shape[1]*0.99) or (agentPosition[agent
                   ][1] <= frame.shape[0]*0.01) or (
                   agentPosition[agent][1] >= frame.shape
                   [0]*0.99))):
                    isEdge[agent] = True
                    timer[agent] = time.time() + 5
                    edgeTimer[agent] = time.time()

                    currentPosition[agent] = agentPosition[
                       agent]
                    orientationDes[agent] = function.
                       get_orientation(currentPosition[agent],
                       desPosition[agent])

                    if (agentPosition[agent][0] <= frame.shape
                       [1]*0.01) and not test:
                        index = np.where(idOrder_array==agent)
                        device.send_data_async(remote_devices[
                           remote_devices[index[0][0]]], "E,L,"
                            + str(orientationDes[agent]) + ">")

                    if (agentPosition[agent][0] >= frame.shape
                       [1]*0.99) and not test:
                        index = np.where(idOrder_array==agent)
                        device.send_data_async(remote_devices[
                           remote_devices[index[0][0]]], "E,R,"
                            + str(orientationDes[agent]) + ">")
```

```
                    if (agentPosition[agent][1] <= frame.shape
                        [0]*0.01) and not test and not (
                        agentPosition[agent][0] <= frame.shape
                        [1]*0.01 or agentPosition[agent][0] >=
                        frame.shape[1]*0.99):
                         index = np.where(idOrder_array==agent)
                         device.send_data_async(remote_devices[
                             remote_devices[index[0][0]]], "E,U,"
                              + str(orientationDes[agent]) + ">")

                    if (agentPosition[agent][1] >= frame.shape
                        [0]*0.99) and not test and not (
                        agentPosition[agent][0] <= frame.shape
                        [1]*0.01 or agentPosition[agent][0] >=
                        frame.shape[1]*0.99):
                         index = np.where(idOrder_array==agent)
                         device.send_data_async(remote_devices[
                             remote_devices[index[0][0]]], "E,D,"
                              + str(orientationDes[agent]) + ">")

# Check time conditions

                if ((((time.time() - timer[agent]) >= 5) or (
                    reset[agent])) and not isSuccess[agent] and
                    not isEdge[agent] and not pass_var[agent]):

# Determine orientation and position of each robot

                      timer[agent] = time.time()

                      if (reset[agent] == False): # If not
                          first run through of loop for this
                          agent

                              messageArray[agent] = "F"

                              timestepDistance[agent] =
```

```
            function.get_distance(
            currentPosition[agent],
            agentPosition[agent])
        if (timestepDistance[agent] > 5):
            prevPosition[agent] =
                currentPosition[agent]
            currentPosition[agent] =
                agentPosition[agent]
            orientation[agent] = function
                .get_orientation(
                prevPosition[agent],
                currentPosition[agent])
            orientationDes[agent] =
                function.get_orientation(
                currentPosition[agent],
                desPosition[agent])
            angle[agent] = function.
                get_angle(orientation[
                agent], orientationDes[
                agent])
            messageArray[agent] =
                function.angle_to_message(
                angle[agent])

        if not test:
            index = np.where(
                idOrder_array==agent)
            device.send_data_async(
                remote_devices[index
                [0][0]], "R," +
                messageArray[agent] + ">")
        print("Object␣ID:␣" + str(agent)
            + "␣Orientation:␣" + str(
            orientation[agent])[:4] + "␣
            Desired␣Orientation:␣" + str(
            orientationDes[agent])[:4] + "
            ␣Message␣Sent:␣"+ messageArray
            [agent][:4])
```

```
                if (reset[agent] == True): # If first run
                    through of loop for this agent
                    reset[agent] = False
                    currentPosition[agent] =
                        agentPosition[agent]
                    if not test:
                        orientation[agent] =
                            orientationInit[agent]
                    else:
                        orientation[agent] = 0
                    orientationDes[agent] = function.
                        get_orientation(currentPosition[
                        agent], desPosition[agent])
                    angle[agent] = function.get_angle(
                        orientation[agent], orientationDes
                        [agent])
                    messageArray[agent] = function.
                        angle_to_message(angle[agent])
                    if not test:
                        index = np.where(idOrder_array==
                            agent)
                        device.send_data_async(
                            remote_devices[index[0][0]], "
                            R," + messageArray[agent] + ">
                            ")
                    print("Object ID: " + str(agent) + "
                        Orientation: " + str(orientation[
                        agent])[:4] + " Desired
                        Orientation: " + str(
                        orientationDes[agent])[:4] + "
                        Message Sent: "+ messageArray[
                        agent][:4])

            isEdge[agent] = False
            pass_var[agent] = False

# Annotate image
```

```
                cv2.line(frame, (agentPosition[agent][0],
                    agentPosition[agent][1]), (int(desPosition[agent
                    ][0]), int(desPosition[agent][1])), (255, 0, 0),
                     1)

# Display image, and store plots for this frame

                fig, axs = plt.subplots(1, 1, figsize=(15, 10))

                for i in range(len(remote_devices)):
                    x, y = zip(*voronoi_update[i])
                    plt.scatter(x, y, s=80)

                x, y = zip(*desPosition)
                plt.scatter(x, y, color='b', s = 150)

                x, y = zip(*agentPosition)
                plt.scatter(x, y, color='k', s = 150)

                if option == 2 or option == 3:
                    plt.scatter(target[0], target[1], color='r')

                plt.axis('off')
                plt.gca().invert_yaxis()
                plt.savefig("Data/option{}/plots/plot{}".format(
                    option, frame_count))
                plt.close('all')

                if option == 2 or option == 3:
                    cv2.circle(frame, (int(target[0]), int(target
                        [1])), 4, (0, 0, 255), -1)
                cv2.imwrite("Data/option{}/frames/frame{}.jpg".
                    format(option, frame_count), frame)
                cv2.imshow("Frame", frame)
                cv2.waitKey(1)
                frame_count = frame_count + 1
```

```
# End loop if "x" key is pressed

            if cv2.waitKey(1) & 0xFF == ord("x"):
                for i in range(0, len(remote_devices)):
                    if not test:
                        device.send_data_async(remote_devices[i
                            ], "P,S>")
                break

# Close video capture and windows

cv2.destroyAllWindows()
vidcap.release()
data_txt.close()
if not test:
    device.close()
```

A.2 Supporting Lib.py File

The source code can be found at the following link: https://github.com/Sarmyt/masters_python_code/blob/master/Lib.py.

Below is a snapshot of the code to support the thesis submission:

A.2.1 Voronoi Diagram Functions

```
from scipy.spatial import distance as dist
from collections import OrderedDict

# Importing utility libraries

import numpy as np
import tensorflow as tf
import cv2
import math
import time
import random

class Voronoi:
```

```
def get_distance(self, point1, point2): # Calculate distance
    between two points
      distance = math.sqrt((point1[0] - point2[0])**2 + (
         point1[1] - point2[1])**2)
      return distance

def voronoi(self, agentPosition, numAgents, xdim, ydim): #
    Calculate the Voronoi diagram given agent positions,
    number of Voronoi generators and xy coordinates of a
    discretized space
     distanceArray = []
     distanceCollection  = []
     indexCollection = []
     voronoiCollection = []
     counter = 0

     for n in range(numAgents):
         distanceArray.insert(n,[])
         indexCollection.insert(n,[])
         voronoiCollection.insert(n,[])
         for i in range(len(ydim)):
             for j in range(len(xdim)):
                 dist = self.get_distance(agentPosition[n], [
                    xdim[j], ydim[i]]) # Check distance
                    between agent "n" and coordinate [j, i]
                 distanceArray[n].append(dist)
         distanceCollection.insert(n, np.array(distanceArray[n
            ]).reshape( np.size(ydim),np.size(xdim) ))

     for n in range(numAgents):
         for ii in range(len(ydim)):
             for jj in range(len(xdim)):

                 for m in range(numAgents):
                     if distanceCollection[n][ii,jj] <=
                        distanceCollection[m][ii,jj]: # Check
                        that the point [ii, jj] is closer to
```

```
                        agent "n" than all other agents
                        counter = counter + 1

                    if counter == numAgents:
                        voronoiCollection[n].append([xdim[jj
                            ],ydim[ii]])
                        indexCollection[n].append([jj,ii])

                counter = 0

        return distanceCollection, voronoiCollection,
            indexCollection

    def risk_density(self, x_dim, y_dim, target, alpha, beta):
        phi = np.zeros((len(y_dim), len(x_dim)))

        # Assign risk density to each coordinate on the mesh grid
            given a distribution function

        for i in range(len(y_dim)):
            for j in range(len(x_dim)):
                dis = self.get_distance([x_dim[j], y_dim[i]],
                    target)
                dis = dis**2
                phi[i][j] = alpha*(np.exp(-beta*dis))

        return phi

    def centroid(self, vor, index, risk_den, agent_pos): #
        Function for finding the centroid of a Voronoi cell given
        the risk distribution in that cell
        risk = 0
        risk_x = 0
        risk_y = 0

        for i in range(len(vor)):
            dis = self.get_distance(agent_pos, vor[i])
```

```
                risk += risk_den[index[i][1]][index[i][0]]*100.0*self
                    .sensing(dis)
                risk_x += vor[i][0]*risk_den[index[i][1]][index[i
                    ][0]]*100.0*self.sensing(dis)
                risk_y += vor[i][1]*risk_den[index[i][1]][index[i
                    ][0]]*100.0*self.sensing(dis)

            C_x = (risk_x/risk)
            C_y = (risk_y/risk)

            return [C_x, C_y]

        def sensing(self, dis): # Sensing capabilities of an agent
            with respect to calculation of the coverage metric
            sense = np.exp(-(dis**2)/(150**2))
            return sense

        def coverage_metric(self, agent_pos, vor, index, risk_den): #
            Calculating the coverage metric over a single Voronoi
            cell
            coverage = 0

            for i in range(len(vor)):
                dis = self.get_distance(agent_pos, vor[i])
                coverage += risk_den[index[i][1]][index[i][0]]*self.
                    sensing(dis)*100.0

            return coverage
```

A.2.2 Centroid Tracker Functions

```
class Tracker(): # Credit for this code goes to Adrian Rosebrock
    at https://www.pyimagesearch.com/2018/07/23/simple-object-
    tracking-with-opencv/

        def __init__(self, maxDisappeared=50):
                # initialize the next unique object ID along with
                    two ordered
                # dictionaries used to keep track of mapping a
```

```
            given object
        # ID to its centroid and number of consecutive
            frames it has
        # been marked as "disappeared", respectively
        self.nextObjectID = 0
        self.objects = OrderedDict()
        self.disappeared = OrderedDict()

        # store the number of maximum consecutive frames
            a given
        # object is allowed to be marked as "disappeared"
             until we
        # need to deregister the object from tracking
        self.maxDisappeared = maxDisappeared

    def register(self, centroid):
        # when registering an object we use the next
            available object
        # ID to store the centroid
        self.objects[self.nextObjectID] = centroid
        self.disappeared[self.nextObjectID] = 0
        self.nextObjectID += 1

    def deregister(self, objectID):
        # to deregister an object ID we delete the object
             ID from
        # both of our respective dictionaries
        del self.objects[objectID]
        del self.disappeared[objectID]

    def update(self, rects):
        # check to see if the list of input bounding box
            rectangles
        # is empty
        if len(rects) == 0:
            # loop over any existing tracked objects
                and mark them
            # as disappeared
```

```
            for objectID in list(self.disappeared.
                keys()):
                    self.disappeared[objectID] += 1

                    # if we have reached a maximum
                        number of consecutive
                    # frames where a given object has
                        been marked as
                    # missing, deregister it
                    if self.disappeared[objectID] >
                        self.maxDisappeared:
                            self.deregister(objectID)

            # return early as there are no centroids
                or tracking info
            # to update
            return self.objects

        # initialize an array of input centroids for the
            current frame
        inputCentroids = np.zeros((len(rects), 2), dtype=
            "int")

        # loop over the bounding box rectangles
        for (i, (startX, startY, endX, endY)) in
            enumerate(rects):
                # use the bounding box coordinates to
                    derive the centroid
                cX = int((startX + endX) / 2.0)
                cY = int((startY + endY) / 2.0)
                inputCentroids[i] = (cX, cY)

        # if we are currently not tracking any objects
            take the input
        # centroids and register each of them
        if len(self.objects) == 0:
                for i in range(0, len(inputCentroids)):
                    self.register(inputCentroids[i])
```

```
# otherwise, we are currently tracking objects so
    we need to
# try to match the input centroids to existing
    object
# centroids
else:
        # grab the set of object IDs and
            corresponding centroids
        objectIDs = list(self.objects.keys())
        objectCentroids = list(self.objects.
            values())

        # compute the distance between each pair
            of object
        # centroids and input centroids,
            respectively -- our
        # goal will be to match an input centroid
            to an existing
        # object centroid
        D = dist.cdist(np.array(objectCentroids),
            inputCentroids)

        # in order to perform this matching we
            must (1) find the
        # smallest value in each row and then (2)
            sort the row
        # indexes based on their minimum values
            so that the row
        # with the smallest value as at the *
            front* of the index
        # list
        rows = D.min(axis=1).argsort()

        # next, we perform a similar process on
            the columns by
        # finding the smallest value in each
            column and then
```

```
            # sorting using the previously computed
                row index list
            cols = D.argmin(axis=1)[rows]

            # in order to determine if we need to
                update, register,
            # or deregister an object we need to keep
                 track of which
            # of the rows and column indexes we have
                already examined
            usedRows = set()
            usedCols = set()

            # loop over the combination of the (row,
                column) index
            # tuples
            for (row, col) in zip(rows, cols):
                    # if we have already examined
                        either the row or
                    # column value before, ignore it
                    # val
                    if row in usedRows or col in
                        usedCols:
                            continue

                    # otherwise, grab the object ID
                        for the current row,
                    # set its new centroid, and reset
                         the disappeared
                    # counter
                    objectID = objectIDs[row]
                    self.objects[objectID] =
                        inputCentroids[col]
                    self.disappeared[objectID] = 0

                    # indicate that we have examined
                        each of the row and
                    # column indexes, respectively
```

```
                usedRows.add(row)
                usedCols.add(col)

        # compute both the row and column index
            we have NOT yet
        # examined
        unusedRows = set(range(0, D.shape[0])).
            difference(usedRows)
        unusedCols = set(range(0, D.shape[1])).
            difference(usedCols)

        # in the event that the number of object
            centroids is
        # equal or greater than the number of
            input centroids
        # we need to check and see if some of
            these objects have
        # potentially disappeared
        if D.shape[0] >= D.shape[1]:
                # loop over the unused row
                    indexes
                for row in unusedRows:
                        # grab the object ID for
                            the corresponding row
                        # index and increment the
                             disappeared counter
                        objectID = objectIDs[row]
                        self.disappeared[objectID
                            ] += 1

                        # check to see if the
                            number of consecutive
                        # frames the object has
                            been marked "
                            disappeared"
                        # for warrants
                            deregistering the
                            object
```

```
                                        if self.disappeared[
                                            objectID] > self.
                                            maxDisappeared:
                                                self.deregister(
                                                    objectID)

                            # otherwise, if the number of input
                                centroids is greater
                            # than the number of existing object
                                centroids we need to
                            # register each new input centroid as a
                                trackable object
                            else:
                                    for col in unusedCols:
                                            self.register(
                                                inputCentroids[col])

                    # return the set of trackable objects
                    return self.objects
```

A.2.3 Utility Functions

```
class Function():

        def angle_to_message(self, angle): # Convert angle to
            string in order to send to Alphabot2 robot
             message = str(angle)
             return message

        def get_angle(self, angle1, angle2): # Get angle
            difference between the orientation of a robot and the
            desired orientation of a robot
             angle = angle2 - angle1
             if angle > 360:
                 angle = angle - 360
             if angle < 0:
                 angle = angle + 360
             return angle
```

```
def get_distance(self, point1, point2): # Get distance
    between two points
      distance = math.sqrt((point1[0] - point2[0])**2 + (
         point1[1] - point2[1])**2)
      return distance

def get_orientation(self, point1, point2): # Get
    orientation of a robot given its previous position and
     its current position
      orientation = math.degrees(math.atan2(point1[1] -
         point2[1], point2[0] - point1[0]))
      if orientation < 0:
          orientation = 360 + orientation
      return orientation

def test_1(self, idArray, numRobots): # Test 1 to ensure
    proper initialization, check that all IDs are present
    and in order
      test1 = False
      testSum = ((numRobots - 1)*(numRobots))/2
      sum = 0
      for i in range(0, numRobots):
          sum += idArray[i]

      if (sum == testSum):
          test1 = True

      return test1

def test_2(self, idArray, numRobots): # Test 2 to ensure
    proper initialization, check that all IDs are unique
      test2 = False
      testArray = []
      for i in range(0, numRobots):
          testArray.append(idArray[i])

      if (len(testArray) == len(set(testArray))):
          test2 = True
```

```
        return test2

    def run_inference_for_single_image(self, image, graph,
        sess): # Given an image, use TensorFlow to run object
        detection on the image to localize Alphabot2 robots
            # Get handles to input and output tensors
            ops = tf.compat.v1.get_default_graph().
                get_operations()
            all_tensor_names = {output.name for op in ops for
                output in op.outputs}
            tensor_dict = {}
            for key in [
                'num_detections', 'detection_boxes', '
                    detection_scores',
                'detection_classes', 'detection_masks'
            ]:
              tensor_name = key + ':0'
              if tensor_name in all_tensor_names:
                tensor_dict[key] = tf.compat.v1.
                    get_default_graph().get_tensor_by_name(
                     tensor_name)
            if 'detection_masks' in tensor_dict:
              # The following processing is only for single
                  image
              detection_boxes = tf.squeeze(tensor_dict['
                  detection_boxes'], [0])
              detection_masks = tf.squeeze(tensor_dict['
                  detection_masks'], [0])
              # Reframe is required to translate mask from box
                  coordinates to image coordinates and fit the
                  image size.
              real_num_detection = tf.cast(tensor_dict['
                  num_detections'][0], tf.int32)
              detection_boxes = tf.slice(detection_boxes, [0,
                  0], [real_num_detection, -1])
              detection_masks = tf.slice(detection_masks, [0,
                  0, 0], [real_num_detection, -1, -1])
```

```
    detection_masks_reframed = utils_ops.
        reframe_box_masks_to_image_masks(
          detection_masks, detection_boxes, image.shape
              [1], image.shape[2])
    detection_masks_reframed = tf.cast(
          tf.greater(detection_masks_reframed, 0.5), tf
              .uint8)
    # Follow the convention by adding back the batch
        dimension
    tensor_dict['detection_masks'] = tf.expand_dims(
          detection_masks_reframed, 0)
image_tensor = tf.compat.v1.get_default_graph().
    get_tensor_by_name('image_tensor:0')

# Run inference
output_dict = sess.run(tensor_dict,
                       feed_dict={image_tensor:
                           image})

# all outputs are float32 numpy arrays, so convert
    types as appropriate
output_dict['num_detections'] = int(output_dict['
    num_detections'][0])
output_dict['detection_classes'] = output_dict[
     'detection_classes'][0].astype(np.int64)
output_dict['detection_boxes'] = output_dict['
    detection_boxes'][0]
output_dict['detection_scores'] = output_dict['
    detection_scores'][0]
if 'detection_masks' in output_dict:
  output_dict['detection_masks'] = output_dict['
      detection_masks'][0]
return output_dict
```

Appendix B

Arduino Code on Alphabot2 Robots

The source code can be found at the following link: https://github.com/Sarmyt/masters_arduino_code/blob/master/Final_Arduino/Final_Arduino.ino.

Below is a snapshot of the code to support the thesis submission:

B.1 Setup Code

```
#include <Wire.h>

#define PWMA   6 // Left wheel power pin
#define AIN2   A0 // Left wheel forward power pin
#define AIN1   A1 // Left wheel backwards power pin
#define PWMB   5 // Right wheel power pin
#define BIN1   A2 // Right wheel backwards power pin
#define BIN2   A3 // Right wheel forwards power pin
#define ECHO   2 // Echo pin for ultrasonic sensor
#define TRIG   3 // Trig pin for ultrasonic sensor

const int MPU = 0x68; // MPU6050 register
float AccX, AccY, AccZ; // Variables for storing accelerometer
    information
float GyroX, GyroY, GyroZ; // Variables for storing gyroscope
    information
float accAngleX, accAngleY, gyroAngleX, gyroAngleY, gyroAngleZ;
    // Variables for storing accelerometer and gyroscopic angles
float roll, pitch, yaw; // Variables for storing roll, pitch and
```

```
   yaw information
float AccErrorX, AccErrorY, GyroErrorX, GyroErrorY, GyroErrorZ;
   // Variables for storing accelerometer and gyroscopic reading
   error
float elapsedTime, currentTime, previousTime; // Variables for
   storing time measurements
int c = 0;

void processIncomingByte (const byte); // Function to process
   incoming byte data
void process_data (const char); // Function to process collection
    of bytes
const unsigned int MAX_INPUT = 50; // Max message character
   length

void calculateImuError(); // Function to calculate accelerometer
   and gyroscopic reading errors

int distance; // Storing distance reading from ultrasonic sensor
int orientation; // Storing orientation of robot
long integral = 0; // Variable used for PID calculation (integral
    term)
unsigned int last_proportional = 0; // Variable used for storing
   previous proportional error
int target = 0; // Variable for storing target orientation of
   robot
double timer = 0; // Timer variable for timing behaviour of robot

int distanceTest(); // Function for testing distance from robot
void robotForward(int, bool); // Function to make robot move
   forward
void robotBackward(int); // Function to make robot move backwards
void robotTurn(int, bool); // Function to make robot turn
void robotStop(); // Function to make robot stop

void setup() {
  Serial.begin(9600);
  Wire.begin();
```

```
  Wire.beginTransmission(MPU);       // Begin communication with
      MPU6050
  Wire.write(0x6B);
  Wire.write(0x00);
  Wire.endTransmission(true);

  calculateImuError(); // Get accelerometer and gyroscopic erros

  delay(20);

  // Define all pins

  pinMode(ECHO, INPUT);
  pinMode(TRIG, OUTPUT);

  pinMode(PWMA,OUTPUT);
  pinMode(AIN2,OUTPUT);
  pinMode(AIN1,OUTPUT);

  pinMode(PWMB,OUTPUT);
  pinMode(BIN1,OUTPUT);
  pinMode(BIN2,OUTPUT);

  robotStop();
}
```

B.2 Loop Code

```
void loop() {
  while (Serial.available() > 0) { // Look for availability of
      serial data
    processIncomingByte(Serial.read()); // If serial data is
        available then process it
  }
}

void process_data (char * data) { // Decode message that was
    received by robot
```

```
const char * splitData;
splitData = strtok(data, ","); // Split message by "," token

switch (splitData[0]) {
case 'P': // Pre-intialization message type
  splitData = strtok(NULL, ",");
  switch (splitData[0]) {
  case 'F':
    robotForward(1000, false); // Move robot forward for one
       second
    break;

  case 'B':
    robotBackward(1000); // Move robot backwards for one second
    break;

  case 'S':
    robotStop(); // Stop robot
    break;

  default:
    robotTurn(atoi(splitData), false); // Turn robot by a given
        angle
    break;
  }
  break;

case 'I': // Initialization message type
  splitData = strtok(NULL, ",");
  orientation = atoi(splitData);
  break;

case 'E': // Edge message type
  splitData = strtok(NULL, ",");
  switch (splitData[0]) {
  case 'U':
    robotTurn(180, true); // Robot has hit an edge, turn robot
       by 180 degrees
```

```
      break;

    case 'D':
      robotTurn(180, true); // Robot has hit an edge, turn robot
         by 180 degrees
      break;

    case 'L':
      robotTurn(180, true); // Robot has hit an edge, turn robot
         by 180 degrees
      break;

    case 'R':
      robotTurn(180, true); // Robot has hit an edge, turn robot
         by 180 degrees
      break;

    default:
      break;
    }
    break;

  case 'A': // Arrival message type
    robotStop(); // Robot has arrived where it needs to be, stop
       it
    splitData = strtok(NULL, ",");
    orientation = atoi(splitData);
    break;

  case 'R': // Run message type
    splitData = strtok(NULL, ",");
    robotTurn(atoi(splitData), true); // Turn robot by desired
       amount indicated in message
    break;

  default:
    break;
  }
```

```
}

void processIncomingByte (const byte inByte) {
  static char input_line[MAX_INPUT];
  static unsigned int input_pos = 0;

  switch (inByte) {
    case '>': // This character indicates end of message
      input_line [input_pos] = 0;

      process_data (input_line);

      input_pos = 0;
      break;

    default: // Message has not ended so move onto next byte
      if (input_pos < (MAX_INPUT - 1))
        input_line [input_pos++] = inByte;
      break;
  }
}

void getOrientation() { // This code is derived from Dejan at
    https://howtomechatronics.com/tutorials/arduino/arduino-and-
    mpu6050-accelerometer-and-gyroscope-tutorial/

  previousTime = currentTime;
  currentTime = millis();
  elapsedTime = (currentTime - previousTime) / 1000; // Time
     elasped from previous reading
  Wire.beginTransmission(MPU);
  Wire.write(0x43); // Access first gyroscope register
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 6, true); // Access data from all 6
     gyroscopic registers
  GyroX = (Wire.read() << 8 | Wire.read()) / 131.0; // Divide
     data collected by sensitivity scale factor
  GyroY = (Wire.read() << 8 | Wire.read()) / 131.0; // Divide
```

```
    data collected by sensitivity scale factor
  GyroZ = (Wire.read() << 8 | Wire.read()) / 131.0; // Divide
    data collected by sensitivity scale factor

  GyroX = GyroX - GyroErrorX; // Adjust for gyroscopic error in x
    -axis
  GyroY = GyroY - GyroErrorY; // Adjust for gyroscopic error in y
    -axis
  GyroZ = GyroZ - GyroErrorZ; // Adjust for gyroscopic error in z
    -axis

  gyroAngleX = gyroAngleX + GyroX * elapsedTime; // Integrate
    gyroscope acceleration over time to get change in angle
  gyroAngleY = gyroAngleY + GyroY * elapsedTime; // Integrate
    gyroscope acceleration over time to get change in angle
  gyroAngleZ = gyroAngleZ + GyroZ * elapsedTime; // Integrate
    gyroscope acceleration over time to get change in angle

  yaw = gyroAngleZ;
  roll = gyroAngleX;
  pitch = gyroAngleY;
}

void calculateImuError() { // This code is derived from Dejan at
    https://howtomechatronics.com/tutorials/arduino/arduino-and-
    mpu6050-accelerometer-and-gyroscope-tutorial/

  while (c < 200) { // Collect gyroscope data 200 times
    Wire.beginTransmission(MPU);
    Wire.write(0x43);
    Wire.endTransmission(false);
    Wire.requestFrom(MPU, 6, true);
    GyroX = Wire.read() << 8 | Wire.read();
    GyroY = Wire.read() << 8 | Wire.read();
    GyroZ = Wire.read() << 8 | Wire.read();

    GyroErrorX = GyroErrorX + (GyroX / 131.0);
    GyroErrorY = GyroErrorY + (GyroY / 131.0);
```

```
    GyroErrorZ = GyroErrorZ + (GyroZ / 131.0);
    c++;
  }

  GyroErrorX = GyroErrorX / 200; // Average out the 200 readings
     to obtain average error reading
  GyroErrorY = GyroErrorY / 200; // Average out the 200 readings
     to obtain average error reading
  GyroErrorZ = GyroErrorZ / 200; // Average out the 200 readings
     to obtain average error reading
}

int distanceTest()
{
  digitalWrite(TRIG, LOW); // Set trig pin off
  delayMicroseconds(2); // Wait two milliseconds
  digitalWrite(TRIG, HIGH); // Set trig pin on
  delayMicroseconds(10); // Wait 10 milliseconds
  digitalWrite(TRIG, LOW); // Set trig pin off
  float Fdistance = pulseIn(ECHO, HIGH); // Turn on echo pin to
     listen for reflection of emitted signal
  Fdistance= Fdistance/59; // Calculate distance to obstacle
  return (int)Fdistance;
}

void robotForward(int duration, bool isRun) // Moves robot
   forward for a given duration
{
  getOrientation(); // Get orientation of robot
  target = roll; // Set heading of robot, this must be maintained
      constant for straightline motion

  unsigned long destination = millis() + duration; // Determine
     length of time to move forward

  analogWrite(PWMA, 0); // Left wheel power to zero
  analogWrite(PWMB, 0); // Right wheel power to zero
```

```
digitalWrite(AIN1,LOW);
digitalWrite(AIN2,HIGH); // Left wheel to move forward
digitalWrite(BIN1,LOW);
digitalWrite(BIN2,HIGH); // Right wheel to move foward

while (millis() < destination) {
getOrientation();

int proportional = roll - target; // Calculate proportional
   error for PID
int derivative = proportional - last_proportional; // Calculate
    derivative error for PID
integral += proportional; // Calculate integral error for PID
last_proportional = proportional; // Store proportional error
   for next time

int power_difference = proportional/20 + derivative*10; // Use
   PD controller to determine appropriate control action

const int maximum = 60;

if (power_difference > maximum)
  power_difference = maximum;
if (power_difference < - maximum)
  power_difference = - maximum;

if (power_difference < 0)
{
   analogWrite(PWMA,maximum + power_difference); // Robot
      orientation is moving right, spin right wheel faster than
       left wheel
   analogWrite(PWMB,maximum);
 }
 else
 {
    analogWrite(PWMA,maximum);
    analogWrite(PWMB,maximum - power_difference); // Robot
       orientation is moving left, spin left wheel faster than
```

```
        right wheel
    }

    distance = distanceTest();

    if (distance <= 3) {
      robotStop(); // If obstacle in the way then stop moving
    }

    if (isRun) { // If robot receives another message then stop
       the forward motion
        if (Serial.available() > 0) {
            destination = millis() - 1;
        }
    }

  }
  integral = 0; // Reset variable to 0
  last_proportional = 0; // Reset variable to 0
  robotStop(); // Stop the robot
}

void robotBackward(int duration) // Look at "robotForwards" for
   reference, this function just makes the robot move backwards
{
  getOrientation();
  target = roll;

  unsigned long destination = millis() + duration;

  analogWrite(PWMA, 0);
  analogWrite(PWMB, 0);

  digitalWrite(AIN1,HIGH);
  digitalWrite(AIN2,LOW);
  digitalWrite(BIN1,HIGH);
  digitalWrite(BIN2,LOW);
```

```
  while (millis() < destination) {
  getOrientation();

  int proportional = roll - target;
  int derivative = proportional - last_proportional;
  integral += proportional;
  last_proportional = proportional;

  int power_difference = proportional/20 + derivative*10;

  const int maximum = 60;

  if (power_difference > maximum)
    power_difference = maximum;
  if (power_difference < -maximum)
    power_difference = -maximum;

  if (power_difference < 0)
  {
     analogWrite(PWMB,maximum + power_difference);
     analogWrite(PWMA,maximum);
   }
   else
   {
      analogWrite(PWMB,maximum);
      analogWrite(PWMA,maximum - power_difference);
   }
  }
  integral = 0;
  last_proportional = 0;
  robotStop();
}

void robotTurn(int angle, bool isRun) // Spins robot by a given
   angle amount
{
  unsigned long timer = millis();
  analogWrite(PWMA, 0); // Left wheel power to zero
```

```
analogWrite(PWMB, 0); // Right wheel power to zero

if (angle > 180) { // Normalize angle between -180 to 180
    degrees
  angle = angle - 360;
}

getOrientation(); // Find current orientation of robot
target = roll + angle; // Determine desired orientation of
    robot
//Serial.println(roll);

while (abs(target - roll) > 2) { // Until the robot is within 2
     degrees of desired angle, keep turning robot
if ((millis() - timer) > 8000) { // If robot has been turning
    for over 8 seconds, then stop turning
  break;
}

getOrientation();

int proportional = roll - target; // Calculate proportional
    error for PID
int derivative = proportional - last_proportional; // Calculate
     derivative error for PID
integral += proportional; // Calculate integral error for PID
last_proportional = proportional; // Store proportional error
    for next time

int power_difference = proportional/40 + integral/500 +
    derivative*10; // Use PID controller to determine
    appropriate control action
power_difference = abs(power_difference);
const int maximum = 60;

if (power_difference > maximum)
  power_difference = maximum;
if (power_difference < 0)
```

```
    power_difference = 0;

  if (target < roll) { // If clockwise rotation is required
      digitalWrite(AIN1,LOW);
      digitalWrite(AIN2,HIGH);
      digitalWrite(BIN1,HIGH);
      digitalWrite(BIN2,LOW);
   }
   else { // If counter-clockwise rotation is required
       digitalWrite(AIN1,HIGH);
       digitalWrite(AIN2,LOW);
       digitalWrite(BIN1,LOW);
       digitalWrite(BIN2,HIGH);
   }

   analogWrite(PWMB, power_difference); // Use PID controller
      output to determine appropriate control action
   analogWrite(PWMA, power_difference); // Use PID controller
      output to determine appropriate control action
  }
  integral = 0; // Reset variable to 0
  last_proportional = 0; // Reset variable to 0
  robotStop(); // Stop the robot

  if (isRun) {
      robotForward(1000, isRun); // When done turning, move robot
          forwards for one second
  }
}

void robotStop()
{ // Stop the robot by stopping all motors
  analogWrite(PWMA,0);
  analogWrite(PWMB,0);
  digitalWrite(AIN1,LOW);
  digitalWrite(AIN2,LOW);
  digitalWrite(BIN1,LOW);
  digitalWrite(BIN2,LOW);
```

```
}
```

Appendix C

Simulation Code Using Processing

The source code can be found at the following link: https://github.com/Sarmyt/masters_processing_simulation/blob/master/Masters_Simulation/Masters_Simulation.pyde.

Below is a snapshot of the code to support the thesis submission:

C.1 Setup Code

```
# Import required Python libraries

import time
import random
from functools import reduce
from operator import mul
import math
import atexit

# Define a linspace function, this function has the same
    behaviour as the numpy.linspace() function

def linspace(start, stop, num=50, endpoint=True):
    num = int(num)
    start = start * 1.
    stop = stop * 1.

    if num == 1:
```

```
        yield stop
        return
    if endpoint:
        step = (stop - start) / (num - 1)
    else:
        step = (stop - start) / num

    for i in range(num):
        yield start + step * i

# Define a meshgrid function, this function has the same
   behaviour as the numpy.meshgrid() function

def meshgrid(xdim, ydim):
    X = []
    Y = []
    for i in range(len(ydim)):
        X.append(xdim)

    for i in range(len(ydim)):
        temp_Y = []
        for j in range(len(xdim)):
            temp_Y.append(ydim[i])
        Y.append(temp_Y)

    return X, Y

# Define an exit handling function, saves all data to a text file
    when the program exits

def exit_handler():
    global string_list
    saveStrings("data/option{}_data_theo.txt".format(option),
       string_list)

atexit.register(exit_handler)

# This reshape function reshapes a Python list into a specified
```

```
    shape

def reshape(lst, shap):
    if len(shap) == 1:
        return lst
    n = reduce(mul, shap[1:])
    return [reshape(lst[i*n:(i+1)*n], shap[1:]) for i in range(
        len(lst)//n)]

# Width and Height of the environment in pixels

w = 640
h = 480

# Discretization of the environment into 64 width sections and 48
    height sections

width_sections = 64
height_sections = 48

# Creation of mesh grid to discretize environment

xdim = list(linspace(5, w - 5, width_sections))
ydim = list(linspace(5, h - 5, height_sections))
X, Y = meshgrid(xdim,ydim)

step_size = 50 # Step size taken by a point agent towards its
    desired position
num_agents = 5 # Number of agents in the environment

# Variables for defining the risk distribution scenario

option = 3 # Variable for selecting between scenario 1, scenario
    2 and scenario 3
a = 1
b = 0.0001

isSuccess = [False, False, False, False, False] # Array
```

```
    indicating which agents have arrived at desired position
reset = True # Variable indicating first run through of the loop
moving = [False, False, False, False, False] # Array to indicate
    point agent motion for 5 seconds
abso_start = millis() # Recording start time of program
buffer = 0 # Time correction to offset time required for
    simulation setup

start = [millis(), millis(), millis(), millis(), millis()] #
    Initiating movement timer

# Variables used for changing position of point agents

changex = [0, 0, 0, 0, 0]
changey = [0, 0, 0, 0, 0]
temp_agent_x = [0, 0, 0, 0, 0]
temp_agent_y = [0, 0, 0, 0, 0]

# Variables used for storing data output information

iteration = 0
string_list = [] # Storing data to be written t otext

# Color of different Voronoi regions

trans = 150
vor_colors = [[0, 255, 255, trans], [0, 255, 0, trans], [255,
    255, 255, trans], [255, 255, 0, trans], [255, 0, 255, trans]]
    # Color palette for Voronoi regions

# Array for containing centroids of each Voronoi region

cent = [[0, 0], [0, 0], [0, 0], [0, 0], [0, 0]]

# Defining initial agent positions and risk distribution based on
    selected option

risk = []
```

```
if option == 1:
    agent_pos = [[264, 116], [365, 313], [504, 279], [444, 125],
       [362, 205]]

elif option == 2:
    target = [320, 240]
    agent_pos = [[445, 321], [153, 403], [335, 150], [134, 154],
       [420, 124]]

elif option == 3:
    target = [485, 320]
    agent_pos = [[300, 221], [89, 117], [174, 124], [379, 94],
       [166, 301]]

for i in range(len(ydim)):
    temp_risk = []
    for j in range(len(xdim)):
        if option == 1:
            temp_risk.extend([1.0])

        else:
            dis = dist(xdim[j], ydim[i], target[0], target[1])
            dis = dis**2
            temp_risk.append(a*(math.exp(-b*dis)))
    risk.append(temp_risk)

# Setup function to create environment with defined Width and
   Height in pixels

def setup():
    size(w, h)
    noStroke()
    global max_distance
    max_distance = dist(0, 0, width, height)

# Loop function which runs continously until program is closed
```

C.2 Loop Code

```
def draw():

# Erasing background with each new loop

    background(0)

# Bringing all global variables defined earlier into the "draw"
   function

    global agent_pos
    global agent
    global reset
    global cent
    global start
    global step_size
    global option
    global moving
    global changex
    global changey
    global temp_agent_x
    global temp_agent_y
    global abso_start
    global buffer
    global iteration
    global string_list
    global xdim
    global ydim

    # Voronoi diagram calculation

    dis_array = []
    distance = []
    index = []
    vors = []

    for n in range(num_agents):
```

```
        dis_array.insert(n,[])
        index.insert(n,[])
        vors.insert(n,[])

        for i in range(len(ydim)):
            for j in range(len(xdim)):
                dis = dist(agent_pos[n][0], agent_pos[n][1], xdim
                    [j], ydim[i]) # obtain distance between agent
                    "n" and point [i, j]
                dis_array[n].append(dis)

        distance.insert(n, reshape(dis_array[n], [height_sections
            , width_sections]))

    for n in range(num_agents):
        for i in range(len(ydim)):
            for j in range(len(xdim)):
                # Ensure that distance between agent "n" and
                    point [i, j] is less than distance between all
                     other agents and point [i, j]
                if distance[n][i][j] \
                <= min(distance[0][i][j], \
                        distance[1][i][j], \
                        distance[2][i][j], \
                        distance[3][i][j], \
                        distance[4][i][j]):

                    vors[n].append([xdim[j],ydim[i]])
                    index[n].append([j,i])

    for i in range(num_agents): # Draw the Voronoi diagram
        fill(vor_colors[i][0], vor_colors[i][1], vor_colors[i
            ][2], vor_colors[i][3])
        for j in range(len(vors[i])):
            ellipse(vors[i][j][0], vors[i][j][1], 15, 15)

    # Centroid calculation for Voronoi cells
```

```
if sum(isSuccess) == 5 or reset: # If this is first loop of
   program, or all agents have arrived at desired location

    iteration = iteration + 1

    for i in range(num_agents): # Calculate the centroid of
       each Voronoi cell given the risk distribution
        isSuccess[i] = False
        risk_t = 0
        risk_x = 0
        risk_y = 0
        for j in range(len(vors[i])):
            dis = dist(agent_pos[i][0], agent_pos[i][1], vors
               [i][j][0], vors[i][j][1])
            sense = math.exp(-(dis**2)/(150**2))

            risk_t += risk[index[i][j][1]][index[i][j
               ][0]]*100.0*sense
            risk_x += vors[i][j][0]*risk[index[i][j][1]][
               index[i][j][0]]*100.0*sense
            risk_y += vors[i][j][1]*risk[index[i][j][1]][
               index[i][j][0]]*100.0*sense

        C_x = (risk_x/risk_t)
        C_y = (risk_y/risk_t)
        cent[i][0] = C_x
        cent[i][1] = C_y

fill(255, 0, 0)
if not option == 1:
    ellipse(target[0], target[1], 10, 10) # Draw the target (
       center of risk distribution)

fill(0, 0, 255)
for i in range(num_agents):
    ellipse(cent[i][0], cent[i][1], 10, 10) # Draw the
       centroid of each Voronoi region
```

```
fill(0, 0, 0)
for i in range(num_agents):
    ellipse(agent_pos[i][0], agent_pos[i][1], 10, 10) # Draw
        the position of all agents

for agent in range(num_agents):
    if moving[agent] == False: # If the agent is currently
        not moving then initiate movement

        angle = math.atan2(-(cent[agent][1] - agent_pos[agent
            ][1]), cent[agent][0] - agent_pos[agent][0]) #
            Calculate angle of movement to get robot to its
            desired position
        temp_distance = dist(agent_pos[agent][0], agent_pos[
            agent][1], cent[agent][0], cent[agent][1]) #
            Calculate distance between agent and its desired
            position

        # Calculate the required movement of agent in the X
            direction

        if (cent[agent][0] - agent_pos[agent][0]) < 0:
            changex[agent] = max(-math.cos(angle)*(-
                temp_distance), -math.cos(angle)*(-step_size))
        elif (cent[agent][0] - agent_pos[agent][0]) > 0:
            changex[agent] = min(math.cos(angle)*(
                temp_distance), math.cos(angle)*step_size)
        else:
            changex[agent] = 0

        # Calculate the required movement of agent in the Y
            direction

        if (cent[agent][1] - agent_pos[agent][1]) < 0:
            changey[agent] = max(math.sin(angle)*(-
                temp_distance), math.sin(angle)*(-step_size))
        elif (cent[agent][1] - agent_pos[agent][1]) > 0:
```

```
            changey[agent] = min(-math.sin(angle)*(
                temp_distance), -math.sin(angle)*step_size)
        else:
            changey[agent] = 0

        temp_agent_x[agent] = agent_pos[agent][0]
        temp_agent_y[agent] = agent_pos[agent][1]
        moving[agent] = True # Record that the robot is now
            moving, and will move for 5 seconds
        start[agent] = millis() # Start 5 second timer for
            robot movement

# Coverage metric calculation

global_coverage = 0

for i in range(num_agents): # Calculate coverage metric of
    the environment in this particular frame of the simulation
    coverage = 0
    for j in range(len(vors[i])):
        dis = dist(agent_pos[i][0], agent_pos[i][1], vors[i][
            j][0], vors[i][j][1])
        sense = math.exp(-(dis**2)/(150**2))
        coverage += risk[index[i][j][1]][index[i][j][0]]*
            sense*100.0

    global_coverage += coverage

fill(255, 255, 255)

saveFrame("data/Option_{}/frame-######.png".format(option)) #
    Store current frame of simulation for data collection

# Record data metrics of the simulation

if reset:
    reset = False
    buffer = millis() - abso_start
```

```
        string_list.append("Time: {}, Coverage Metric: {},
            Iteration: {}, Agent Positions: {}, Desired Positions:
            {}".format(0, global_coverage, iteration, agent_pos,
            cent))
    else:
        string_list.append("Time: {}, Coverage Metric: {},
            Iteration: {}, Agent Positions: {}, Desired Positions:
            {}".format((millis() - abso_start - buffer)/1000.0,
            global_coverage, iteration, agent_pos, cent))

    for agent in range(num_agents):

        # Record the amount of time an agent has been moving

        diff = millis() - start[agent]

        # If point agent has been moving for less than 5 seconds
            then continue moving agent to desired position

        if (diff <= 5000):

            agent_pos[agent][0] = temp_agent_x[agent] + changex[
                agent]*((diff)/5000.0)
            agent_pos[agent][1] = temp_agent_y[agent] + changey[
                agent]*((diff)/5000.0)

        # If point agent has been moving for more than 5 seconds
            then stop moving agent

        else:

            agent_pos[agent][0] = temp_agent_x[agent] + changex[
                agent]
            agent_pos[agent][1] = temp_agent_y[agent] + changey[
                agent]

            moving[agent] = False
```

```
            if int(cent[agent][0] - agent_pos[agent][0]) == 0 and
                int(cent[agent][1] - agent_pos[agent][1]) == 0: #
                If distance between agent and its desired
                position is zero, then agent has arrived at
                desired position
                isSuccess[agent] = True
```

Appendix D

Code for Training SSD MobilenetV2 COCO to Detect Alphabot2 Robot in Google Colab

The source code can be found at the following link: https://colab.research.google.com/drive/1_qgZ8TM8YvMEtIhbZXg-49j5NzODt6hU?authuser=2#scrollTo=zFefQLaSqo_V.

While all supporting data sets, training configuration files and utility folders can be found at: https://github.com/Sarmyt/masters_object_detection.

Below is a snapshot of the code to support the thesis submission:

Ensure that Python version is between 3.5.x and 3.7.x for proper execution.

[ ]:
```
!python --version
```

Ensure that TensorFlow 1.15.x is installed for proper execution.

[ ]:
```
%tensorflow_version 1.x
```

[ ]:
```
import tensorflow as tf
print(tf.__version__)
```

Navigate to root directory.

[ ]:
```
%cd
```

Clone the object detection repository from GitHub.

[ ]:
```
!git clone --quiet https://github.com/Sarmyt/masters_object_detection.git
```

Install required Python libraries for proper execution.

[ ]:
```
!apt-get install -qq protobuf-compiler python-tk
```

[ ]:
```
!pip install -q Cython contextlib2 pillow lxml matplotlib PyDrive
```

[ ]:
```
!pip install -q pycocotools
```

Run compiler for protocol buffer definition files.

[ ]:
```
%cd ~/masters_object_detection/research
!protoc object_detection/protos/*.proto --python_out=.
```

Build and install the setup file to allow for training of the SSD Mobilenet V2 COCO object detection model.

[ ]:
```
!python setup.py build
!python setup.py install
```

Install the TensorFlow Slim package to use the Mobilenet V2 model.

[ ]:
```
%cd slim
!pip install -e .
```

Run test to check that object detection models are built properly.

[ ]:
```
%cd ..
!python object_detection/builders/model_builder_test.py
```

Navigate to main GitHub directory.

[ ]:
```
%cd ..
```

Initiate training process to to iterate through the custom dataset for the Alphabot2 and optimize object detection.

[ ]:
```
!python ~/masters_object_detection/research/object_detection/legacy/train.py \
    --pipeline_config_path=ssd_mobilenet_v2_coco.config \
    --train_dir=train \
    --logtostderr \
```

Navigate through directories to evaluate the training results.

[ ]:
```
%cd train
```

[ ]:
```
!ls
```

[ ]:
```
%cd ..
```

[ ]:
```
# Run All Cells Above
```

Use a TensorFlow provided script to export the trained object detection model as an inference graph for use in the lab. Make sure to change the checkpoint number to the specific checkpoint for your training process.

[ ]:
```
!mkdir fine_tuned_model

!python ~/masters_object_detection/research/object_detection/
 ↪export_inference_graph.py --input_type image_tensor --pipeline_config_path␣
 ↪ssd_mobilenet_v2_coco.config --trained_checkpoint_prefix  train/model.
 ↪ckpt-1207 --output_directory fine_tuned_model
```

[ ]:
```
%cd fine_tuned_model
```

[ ]:
```
!ls
```

Download the inference graph for use with local code.

[ ]:
```
from google.colab import files
files.download('frozen_inference_graph.pb')
```

Appendix E

Cost of Test Platform

The cost of the major components of the hardware test platform design are provided. The cost for a laptop/desktop as a central processing unit is not included, as this cost may vary depending the specifications of the central processing unit. The total cost of the proposed hardware test design comes to 1100 CAD. The proposed hardware design has the added benefit of being relatively low cost and modular, making it easy to scale.

Item	Cost per Unit (CAD)	Number of Units	Total (CAD)
Alphabot2	80	5	400
14500 Batteries	4	10	40
Battery Charger	30	1	30
Arduino UNO	30	5	150
MPU6050 IMU	10	5	50
XBee Series 2X Pro	65	5	325
XBee Shield	15	5	75
XBee Explorer	30	1	30

Table E.1: Cost of Components for the Hardware Test Platform

TOTAL: 1100 CAD

Appendix F

Raw Data for Experiment and Simulation

Raw data for this thesis can be found at the following link: https://drive.google.com/drive/folders/17o7wstouWP3vnLcLDFxX6qCyarM7WFql?usp=sharing

The data related to the main results of the thesis as presented in Chapter 4 can be found in the folder labelled "results".

Supplementary data, such as data for the object detection model training (Section 3.6.1.7) and data for the sample responses from the Alphabot2 system with PD/PID controllers (Sections 3.5.3 and 3.5.4) can be found in the folder labelled "supplementary".

Bibliography

[1] Seong-eun Yoo et al. "Technological Advances in Wireless Sensor Networks Enabling Diverse Internet of Things Applications". In: *International Journal of Distributed Sensor Networks* 14 (Mar. 1, 2018), p. 155014771876322. DOI: 10.1177/1550147718763220.

[2] Silvia Liberata Ullo and G. R. Sinha. "Advances in Smart Environment Monitoring Systems Using IoT and Sensors". In: *Sensors (Basel, Switzerland)* 20.11 (May 31, 2020). ISSN: 1424-8220. DOI: 10.3390/s20113113. pmid: 32486411. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7309034/.

[3] Suruz Miah et al. "Nonuniform Deployment of Autonomous Agents in Harbor-Like Environments". In: *Unmanned Systems* 02.04 (Oct. 1, 2014), pp. 377–389. ISSN: 2301-3850. DOI: 10.1142/S2301385014400111. URL: https://www.worldscientific.com/doi/abs/10.1142/S2301385014400111.

[4] E. Simetti et al. "Towards the Use of a Team of USVs for Civilian Harbour Protection: USV Interception of Detected Menaces". In: *IFAC Proceedings Volumes*. 7th IFAC Symposium on Intelligent Autonomous Vehicles 43.16 (Jan. 1, 2010), pp. 145–150. ISSN: 1474-6670. DOI: 10.3182/20100906-3-IT-2019.00027. URL: https://www.sciencedirect.com/science/article/pii/S1474667016350479.

[5] Luciano C. A. Pimenta et al. "Decentralized Controllers for Perimeter Surveillance with Teams of Aerial Robots". In: *Advanced Robotics* 27.9 (June 1, 2013), pp. 697–709. ISSN: 0169-1864. DOI: 10.1080/01691864.2013.778942. URL: https://doi.org/10.1080/01691864.2013.778942.

[6] Guoxian Zhang, Gregory K. Fricke, and Devendra P. Garg. "Spill Detection and Perimeter Surveillance via Distributed Swarming Agents". In: *IEEE/ASME Transactions on Mechatronics* 18.1 (Feb. 2013), pp. 121–129. ISSN: 1941-014X. DOI: 10.1109/TMECH.2011.2164578.

[7] Jinwen Hu et al. "Multiagent Information Fusion and Cooperative Control in Target Search". In: *IEEE Transactions on Control Systems Technology* 21.4 (July 2013), pp. 1223–1235. ISSN: 1558-0865. DOI: 10.1109/TCST.2012.2198650.

[8] Davide Spinello and Daniel J. Stilwell. "Distributed Full-State Observers With Limited Communication and Application to Cooperative Target Localization". In: *Journal of Dynamic Systems, Measurement, and Control* 136.031022 (Mar. 4, 2014). ISSN: 0022-0434. DOI: 10.1115/1.4026173. URL: https://doi.org/10.1115/1.4026173.

[9] Suruz Miah et al. "Nonuniform Coverage Control With Stochastic Intermittent Communication". In: *IEEE Transactions on Automatic Control* 60.7 (July 2015), pp. 1981–1986. ISSN: 1558-2523. DOI: 10.1109/TAC.2014.2368233.

[10] Suruz Miah, Mostafa M. H. Fallah, and Davide Spinello. "Non-Autonomous Coverage Control With Diffusive Evolving Density". In: *IEEE Transactions on Automatic Control* 62.10 (Oct. 2017), pp. 5262–5268. ISSN: 1558-2523. DOI: 10.1109/TAC.2016.2633789.

[11] Suruz Miah et al. "Generalized Non-Autonomous Metric Optimization for Area Coverage Problems with Mobile Autonomous Agents". In: *Automatica (Journal of IFAC)* 80.C (June 1, 2017), pp. 295–299. ISSN: 0005-1098. DOI: 10.1016/j.automatica.2017.02.044. URL: https://doi.org/10.1016/j.automatica.2017.02.044.

[12] J. Cortes et al. "Coverage Control for Mobile Sensing Networks". In: *IEEE Transactions on Robotics and Automation* 20.2 (Apr. 2004), pp. 243–255. ISSN: 2374-958X. DOI: 10.1109/TRA.2004.824698.

[13] M. Emelianenko, Lili Ju, and Alexander Rand. "Nondegeneracy and Weak Global Convergence of the Lloyd Algorithm in Rd". In: *SIAM Journal on Numerical Analysis* 46 (Jan. 1, 2008). DOI: 10.1137/070691334.

[14] Imad Jawhar et al. "Networking of Multi-Robot Systems: Architectures and Requirements". In: *Journal of Sensor and Actuator Networks* 7 (Nov. 30, 2018), p. 52. DOI: 10.3390/jsan7040052.

[15] Andrei Stancovici, Mihai V. Micea, and Vladimir Cretu. "Cooperative Positioning System for Indoor Surveillance Applications". In: *2016 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*. 2016 International Conference on Indoor Positioning and Indoor Navigation (IPIN). Oct. 2016, pp. 1–7. DOI: 10.1109/IPIN.2016.7743584.

[16] Ricardo Mendonça et al. "A Cooperative Multi-Robot Team for the Surveillance of Shipwreck Survivors at Sea". In: *OCEANS 2016 MTS/IEEE Monterey*. OCEANS 2016 MTS/IEEE Monterey. Sept. 2016, pp. 1–6. DOI: 10.1109/OCEANS.2016.7761074.

[17] Nikhil Nigam et al. "Control of Multiple UAVs for Persistent Surveillance: Algorithm and Flight Test Results". In: *IEEE Transactions on Control Systems Technology* 20.5 (Sept. 2012), pp. 1236–1251. ISSN: 1558-0865. DOI: 10.1109/TCST.2011.2167331.

[18] Jason Gregory et al. "Application of Multi-Robot Systems to Disaster-Relief Scenarios with Limited Communication". In: *Field and Service Robotics: Results of the 10th International Conference*. Ed. by David S. Wettergreen and Timothy D. Barfoot. Springer Tracts in Advanced Robotics. Cham: Springer International Publishing, 2016, pp. 639–653. ISBN: 978-3-319-27702-8. DOI: 10.1007/978-3-319-27702-8_42. URL: https://doi.org/10.1007/978-3-319-27702-8_42.

[19] Marco A. Gutiérrez et al. "Multi-Robot Collaborative Platforms for Humanitarian Relief Actions". In: *2015 IEEE Region 10 Humanitarian Technology Conference (R10-HTC)*. 2015 IEEE Region 10 Humanitarian Technology Conference (R10-HTC). Dec. 2015, pp. 1–6. DOI: 10.1109/R10-HTC.2015.7391867.

[20] L.E. Parker. "ALLIANCE: An Architecture for Fault Tolerant Multirobot Cooperation". In: *IEEE Transactions on Robotics and Automation* 14.2 (Apr. 1998), pp. 220–240. ISSN: 2374-958X. DOI: 10.1109/70.681242.

[21] M. Schneider-Fontan and M.J. Mataric. "Territorial Multi-Robot Task Division". In: *IEEE Transactions on Robotics and Automation* 14.5 (Oct. 1998), pp. 815–822. ISSN: 2374-958X. DOI: 10.1109/70.720357.

[22] Adam Lein and Richard T. Vaughan. "Adapting to Non-Uniform Resource Distributions in Robotic Swarm Foraging through Work-Site Relocation". In: *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems. Oct. 2009, pp. 601–606. DOI: 10.1109/IROS.2009.5354693.

[23] Xiaoli Wang, Wei Ni, and Xinsheng Wang. "Leader-Following Formation of Switching Multirobot Systems via Internal Model". In: *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics: a publication of the IEEE Sys-*

tems, Man, and Cybernetics Society 42.3 (June 2012), pp. 817–826. ISSN: 1941-0492. DOI: 10.1109/TSMCB.2011.2178022. pmid: 22262683.

[24] Yunpeng Wang et al. "Optimal Formation of Multirobot Systems Based on a Recurrent Neural Network". In: *IEEE Transactions on Neural Networks and Learning Systems* 27.2 (Feb. 2016), pp. 322–333. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2015.2464314.

[25] Rattanapol Phanichnitinon et al. "Multi Modular Robots Maneuver for Geometry Formation Control". In: *2014 IEEE 7th International Workshop on Computational Intelligence and Applications (IWCIA)*. 2014 IEEE 7th International Workshop on Computational Intelligence and Applications (IWCIA). Nov. 2014, pp. 195–200. DOI: 10.1109/IWCIA.2014.6988105.

[26] Jacopo Banfi et al. "Asynchronous Multirobot Exploration under Recurrent Connectivity Constraints". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016 IEEE International Conference on Robotics and Automation (ICRA). May 2016, pp. 5491–5498. DOI: 10.1109/ICRA.2016.7487763.

[27] Kyle Cesare et al. "Multi-UAV Exploration with Limited Communication and Battery". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015 IEEE International Conference on Robotics and Automation (ICRA). May 2015, pp. 2230–2235. DOI: 10.1109/ICRA.2015.7139494.

[28] Flavio Cabrera-Mora and Jizhong Xiao. "A Flooding Algorithm for Multirobot Exploration". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 42.3 (June 2012), pp. 850–863. ISSN: 1941-0492. DOI: 10.1109/TSMCB.2011.2179799.

[29] Arnab Ghosh et al. "Multi-Robot Cooperative Box-Pushing Problem Using Multi-Objective Particle Swarm Optimization Technique". In: *2012 World Congress on Information and Communication Technologies*. 2012 World Congress on Information and Communication Technologies. Oct. 2012, pp. 272–277. DOI: 10.1109/WICT.2012.6409087.

[30] Dominik Sieber, Frederik Deroo, and Sandra Hirche. "Formation-Based Approach for Multi-Robot Cooperative Manipulation Based on Optimal Control Design". In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems. Nov. 2013, pp. 5227–5233. DOI: 10.1109/IROS.2013.6697112.

[31] Andrea Zambelli Bais et al. "Dynamic Load Distribution in Cooperative Manipulation Tasks". In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). Sept. 2015, pp. 2380–2385. DOI: 10.1109/IROS.2015.7353699.

[32] Zekai Sen. *Spatial Modeling Principles in Earth Sciences*. Springer, Oct. 4, 2016. 424 pp. ISBN: 978-3-319-41758-5. Google Books: 6N0yDQAAQBAJ.

[33] Adam Dobrin. *A REVIEW OF PROPERTIES AND VARIATIONS OF VORONOI DIAGRAMS*. URL: https://www.whitman.edu/Documents/Academics/Mathematics/dobrinat.pdf.

[34] Qiang Du and Desheng Wang. "The Optimal Centroidal Voronoi Tessellations and the Gersho's Conjecture in the Three-Dimensional Space". In: *Computers & Mathematics with Applications* 49.9 (May 1, 2005), pp. 1355–1373. ISSN: 0898-1221. DOI: 10.1016/j.camwa.2004.12.008. URL: https://www.sciencedirect.com/science/article/pii/S0898122105001550.

[35] Andreas Breitenmoser et al. "Voronoi Coverage of Non-Convex Environments with a Group of Networked Robots". In: *2010 IEEE International Conference on Robotics and Automation*. 2010 IEEE International Conference on Robotics and Automation. May 2010, pp. 4982–4989. DOI: 10.1109/ROBOT.2010.5509696.

[36] Luciano Pimenta et al. "Simultaneous Coverage and Tracking (SCAT) of Moving Targets with Robot Networks". In: Springer Tracts in Advanced Robotics. Vol. 57. Jan. 1, 2008, pp. 85–99. ISBN: 978-3-642-00311-0. DOI: 10.1007/978-3-642-00312-7_6.

[37] Patrick Mannion. *Comparing Low-Power Wireless Technologies (Part 1)*. URL: https://www.digikey.ca/en/articles/comparing-low-power-wireless-technologies.

[38] Ali Nikoukar et al. "Low-Power Wireless for the Internet of Things: Standards and Applications". In: *IEEE Access* 6 (2018), pp. 67893–67926. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2879189.

[39] Alyssa Apsel. "A Simple Guide to Low-Power Wireless Technologies: Balancing the Tradeoffs for the Internet of Things and Medical Applications". In: *IEEE Solid-State Circuits Magazine* 10.4 (Aut. 2018), pp. 16–23. ISSN: 1943-0590. DOI: 10.1109/MSSC.2018.2867404.

[40] Patrick Mannion. *Comparing Low Power Wireless Technologies (Part 2)*. URL: https://www.digikey.ca/en/articles/comparing-low-power-wireless-technologies-part-2.

[41] Faheem Zafari, Athanasios Gkelias, and Kin K. Leung. "A Survey of Indoor Localization Systems and Technologies". In: *IEEE Communications Surveys Tutorials* 21.3 (2019), pp. 2568–2599. ISSN: 1553-877X. DOI: 10.1109/COMST.2019.2911558.

[42] Ayuba Peter, Yohanna Tella, and Gabriel Dams. *An Overview of Indoor Localization Technologies and Applications*. June 4, 2015. URL: https://www.researchgate.net/publication/277698379_An_overview_of_Indoor_Localization_Technologies_and_applications.

[43] D. Stojanovic and N. Stojanovic. *INDOOR LOCALIZATION AND TRACKING: METHODS, TECHNOLOGIES AND RESEARCH CHALLENGES*. 2014. URL: http://casopisi.junis.ni.ac.rs/index.php/FUAutContRob/article/view/208/88.

[44] Michael Rubenstein, Christian Tjornelund Ahler, and Radhika Nagpal. "Kilobot: A Low Cost Scalable Robot System for Collective Behaviors". In: *Proceedings of 2012 IEEE International Conference on Robotics and Automation* (2012). ISSN: 1050-4729. DOI: 10.1109/ICRA.2012.6224638. URL: https://dash.harvard.edu/handle/1/9367001.

[45] P. Vartholomeos and E. Papadopoulos. "Analysis, Design and Control of a Planar Micro-Robot Driven by Two Centripetal-Force Actuators". In: *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.* Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006. 2006, pp. 649–654. DOI: 10.1109/ROBOT.2006.1641784.

[46] Waveshare. *AlphaBot - Waveshare Wiki*. URL: https://www.waveshare.com/wiki/AlphaBot.

[47] Waveshare. *AlphaBot2-Ar - Waveshare Wiki*. URL: https://www.waveshare.com/wiki/AlphaBot2-Ar.

[48] UCTRONICS. *UCTRONICS Smart Bluetooth Robot Car Kit*. URL: https://www.uctronics.com/wiki/(SKU:_K0072)UCTRONICS_Smart_Bluetooth_Robot_Car_Kit.

[49] SparkFun. *SparkFun RedBot*. URL: https://www.sparkfun.com/products/12649.

[50] Eka Maulana, M. Aziz Muslim, and Akhmad Zainuri. "Inverse Kinematics of a Two-Wheeled Differential Drive an Autonomous Mobile Robot". In: *2014 Electrical Power, Electronics, Communicatons, Control and Informatics Seminar (EECCIS)*. 2014 Electrical Power, Electronics, Communicatons, Control and Informatics Seminar (EECCIS). Aug. 2014, pp. 93–98. DOI: 10.1109/EECCIS.2014.7003726.

[51] Frederick Crabbe. *Mobile Robot Kinematics*. URL: https://www.usna.edu/Users/cs/crabbe/SI475/current/mob-kin/mobkin.pdf.

[52] Invensense. *MPU6050 Datasheet*. 2013. URL: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf.

[53] Digi. *XCTU*. URL: https://www.digi.com/products/embedded-systems/digi-xbee/digi-xbee-tools/xctu.

[54] Digi. *XBee Python Library — Digi XBee Python Library*. URL: https://xbplib.readthedocs.io/en/latest/.

[55] Sparkfun. *XBee Guide*. URL: https://www.sparkfun.com/pages/xbee_guide.

[56] Digi. *XBee-Pro S2C Zigbee*. 2020. URL: https://www.digi.com/resources/documentation/digidocs/pdfs/90002002.pdf.

[57] Robotshop. *XBee-Pro Transceiver w/ Wire Antenna*. URL: https://www.robotshop.com/ca/en/xbee-pro-transceiver-wire-antenna.html.

[58] Digi. *XBee-Pro Modules Guide*. 2018. URL: https://www.digi.com/resources/documentation/digidocs/pdfs/90000976.pdf.

[59] Jeff Watson. *Mems Gyroscope Provides Precision Inertial Sensing in Harsh, High Temperature Environments*. URL: https://www.analog.com/media/en/technical-documentation/tech-articles/MEMS-Gyroscope-Provides-Precision-Inertial-Sensing-in-Harsh-High-Temps.pdf.

[60] Invensense. *MPU6050 Register Map*. 2013. URL: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf.

[61] Katsuhiko Ogata. *Modern Control Engineering 5th Edition*. URL: https://www.pearson.com/us/higher-education/program/Ogata-Modern-Control-Engineering-5th-Edition/PGM100186.html.

[62] Sergio Guadarrama and Nathan Silberman. *TensorFlow-Slim Image Classification Model Library*. URL: https://github.com/tensorflow/models/tree/master/research/slim.

[63] Jonathan Huang et al. *Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors*. Apr. 24, 2017. arXiv: 1611.10012 [cs]. URL: http://arxiv.org/abs/1611.10012.

[64] Martin Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[65] Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer, Aug. 25, 2018. 512 pp. ISBN: 978-3-319-94463-0. Google Books: achqDwAAQBAJ.

[66] Justin Beile. *Introduction to Perceptron: Neural Network*. 2017. URL: https://blog.knoldus.com/introduction-to-perceptron-neural-network/.

[67] Anastasia Kyrykovych. *Deep Neural Networks*. URL: https://www.kdnuggets.com/2020/02/deep-neural-networks.html.

[68] Daphne Cornelisse. *An Intuitive Guide to Convolutional Neural Networks*. URL: https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/.

[69] Mark Sandler et al. *MobileNetV2: Inverted Residuals and Linear Bottlenecks*. Mar. 21, 2019. arXiv: 1801.04381 [cs]. URL: http://arxiv.org/abs/1801.04381.

[70] Lars Hulstaert. *A Beginner's Guide to Object Detection*. 2018. URL: https://www.datacamp.com/community/tutorials/object-detection-guide.

[71] Wei Liu et al. *SSD: Single Shot MultiBox Detector*. 2016. DOI: 10.1007/978-3-319-46448-0_2. arXiv: 1512.02325 [cs]. URL: http://arxiv.org/abs/1512.02325.

[72] Lilian Weng. *Object Detection Part 4: Fast Detection Models*. 2018. URL: https://lilianweng.github.io/lil-log/2018/12/27/object-detection-part-4.html.

[73] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. Feb. 20, 2015. arXiv: 1405.0312 [cs]. URL: http://arxiv.org/abs/1405.0312.

[74] Tzutalin. *LabelImg*. URL: https://github.com/tzutalin/labelImg.

[75] Ross Girshick. *Fast R-CNN*. Sept. 27, 2015. arXiv: 1504.08083 [cs]. URL: http://arxiv.org/abs/1504.08083.

[76] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. *Neural Networks for Machine Learning - Lecture 6*. URL: https://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf.

[77] Adrian Rosebrock. *Simple Object Tracking with OpenCV*. PyImageSearch. July 23, 2018. URL: https://www.pyimagesearch.com/2018/07/23/simple-object-tracking-with-opencv/.