Robotic Swarm Control using Deep Reinforcement Learning Strategies based on Mean-Field Models

by

Zahi Kakish

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved February 2021 by the
Graduate Supervisory Committee:

Spring Berman, Chair
Sze Zheng Yong
Hamid Marvi
Theodore Pavlic
Stephen Pratt
Hani Ben Amor

ARIZONA STATE UNIVERSITY

May 2021

ABSTRACT

As technological advancements in silicon, sensors, and actuation continue, the development of robotic swarms is shifting from the domain of science fiction to reality. Many swarm applications, such as environmental monitoring, precision agriculture, disaster response, and lunar prospecting, will require controlling numerous robots with limited capabilities and information to redistribute among multiple states, such as spatial locations or tasks. A scalable control approach is to program the robots with stochastic control policies such that the robot population in each state evolves according to a mean-field model, which is independent of the number and identities of the robots. Using this model, the control policies can be designed to stabilize the swarm to the target distribution. To avoid the need to reprogram the robots for different target distributions, the robot control policies can be defined to depend only on the presence of a "leader" agent, whose control policy is designed to guide the swarm to a particular distribution.

This dissertation presents a novel deep reinforcement learning (deep RL) approach to designing control policies that redistribute a swarm as quickly as possible over a strongly connected graph, according to a mean-field model in the form of the discrete-time Kolmogorov forward equation. In the leader-based strategies, the leader determines its next action based on its observations of robot populations and shepherds the swarm over the graph by probabilistically repelling nearby robots. The scalability of this approach with the swarm size is demonstrated with leader control policies that are designed using two tabular Temporal-Difference learning algorithms, trained on a discretization of the swarm distribution. To improve the scalability of the approach with robot population and graph size, control policies for both leader-based and leaderless strategies are designed using an actor-critic deep RL method that is trained on the swarm distribution predicted by the mean-field model. In the leader-

less strategy, the robots' control policies depend only on their local measurements of nearby robot populations. The control approaches are validated for different graph and swarm sizes in numerical simulations, 3D robot simulations, and experiments on a multi-robot testbed.

DEDICATION

To my wife Amanda and our daughter, my mom and dad, my supportive family, and my friends.

ACKNOWLEDGEMENTS

Firstly, I would like to acknowledge all my family and friends for their never-ending love, support, and patience. It was a difficult journey, but they prove time and time again that they are the rock from which I grow. To my lovely wife, this work is for you and our lovely daughter. To my supportive father and mother, thank you for all you have given me and I hope I've made you proud. To my sisters, thank you for always having my back when things got tough.

Secondly, I would like to thank my advisor Dr. Spring Berman whose help and mentorship has guided me throughout my PhD career. Her flexibility, enthusiasm, and support truly allowed me to explore the field in ways I didn't think possible. To Dr. Ted Pavlic, Dr. Stephen Pratt, Dr. Sze Zheng Yong, Dr. Heni Ben Amor, and Dr. Hamid Marvi, thank you all for being such an inspiration for my work in all your respective fields. I truly learned to be a better researcher, mentor, and person from my work with all of you.

I would like to especially thank Dr. Karthik Elamvazhuthi and Dr. Shiba Biswal for their amazing support, friendship, encouragement, and guidance throughout my PhD career. They both will flourish in academia and in their research work. Not only are they some of the smartest people I know, but they are also the kindest, most down-to-earth friends I have ever met.

To my friends Dr. Justin Hunt, Dr. Varun Nalam, and soon-to-be Dr. Andrew Barkan, thank you so much for your help and friendship throughout our time at ASU. I thoroughly enjoyed our coffee runs, our continued adventures building a robotics startup, and those fun rock climbing sessions. You guys alone have made my experience at ASU worth every moment.

I dedicate this work to all of you.

This research was supported by the Arizona State University Global Security Initiative.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Chapter 1

INTRODUCTION

As technology for robotic sensing, actuation, and fabrication advances and the price of low-power computing devices continues to decrease, the robotics community recognizes the increasing viability of robotic swarms for a multitude of applications. The emphasis on scalability and hardware redundancy in robotic swarms, however, makes it challenging to develop robust and stable management and control tools for these systems. In this dissertation, we develop approaches to controlling a robotic swarm through the influence of an entity designated as a *leader*, without requiring explicit communication with the robots or direct control of individual robots. This leader-follower control approach can be used to redistribute a swarm of low-cost robots with limited capabilities and information using a single robot with sophisticated sensing, localization, computation, and planning capabilities, in scenarios where the leader lacks a model of the swarm dynamics. Such a control strategy is useful for many applications in swarm robotics, including exploration, environmental monitoring, inspection tasks, disaster response, and targeted drug delivery at the micro-nanoscale.

Specifically, we look at Deep Reinforcement Learning (deep RL) algorithms trained on a mean-field model approach for finding an optimal leader control policy. This methodology is then expanded to a leaderless approach where robots utilize a deep RL control policy pre-trained on a modification of the mean-field model. We address this problem in multiple scenarios, each of which entails a different mechanism of control depending on the type of robots deployed, the type of environment, and the nature of the interaction between the leader and the swarm. Here, we summarize

these control approaches and discuss their potential applications and related work from the literature.

1.1 Contributions

The contributions of this dissertation are as follows:

1. A methodology for designing leader-based or leaderless control strategies that quickly redistribute a swarm of robots among a set of states, e.g. tasks or spatial locations, using deep reinforcement learning algorithms that are trained on mean-field models of the swarm population dynamics, rather than on models or observations of the behaviors of individual robots.

2. Insight into the effects of the neural network configuration, optimal hidden network depth, and number of network parameters in these deep reinforcement learning algorithms on the performance of the resulting leader-based control strategies for swarm redistribution.

3. Validation of the control approaches for different numbers of robots and states using 2D point-robot simulations, 3D robot simulations, and physical robot experiments.

1.2 Literature Review

There has been a considerable amount of work on leader-follower multi-agent control schemes in which the leader has an attractive effect on the followers (Ji *et al.*, 2008; Mesbahi and Egerstedt, 2010). Several recent works have presented models for herding robotic swarms using leaders that have a repulsive effect on the swarm (Pierson and Schwager, 2017; Elamvazhuthi *et al.*, 2016; Paranjape *et al.*, 2018). Using such models, analytical controllers for herding a swarm have been constructed for

the case when there is a single leader (Elamvazhuthi *et al.*, 2016; Paranjape *et al.*, 2018) and multiple leaders (Pierson and Schwager, 2017). The controllers designed in these works are not necessarily optimal for a given performance metric. To design optimal control policies for a herding model, the authors in (Go *et al.*, 2016) consider a reinforcement learning (RL) approach. While existing herding models are suitable for the objective of confining a swarm to a small region in space, many applications require a swarm to cover an area according to some target probability density. If the robots do not have spatial localization capabilities, then the controllers developed in (Ji *et al.*, 2008; Mesbahi and Egerstedt, 2010; Pierson and Schwager, 2017; Elamvazhuthi *et al.*, 2016; Paranjape *et al.*, 2018; Go *et al.*, 2016) cannot be applied for such coverage problems. Moreover, these models are not suitable for herding large swarms using RL-based control approaches, since such approaches would not scale well with the number of robots. This loss of scalability is due to the fact that the models describe individual agents, which may not be necessary since robot identities are not important for many swarm applications.

We consider a *mean-field* or *macroscopic* model that describes the swarm of follower agents as a probability distribution over a graph, which represents the configuration space of each agent. Previous work has utilized similar mean-field models to design a set of control policies that is implemented on each robot in a swarm in order to drive the entire swarm to a target distribution, e.g. for problems in spatial coverage and task allocation (Elamvazhuthi and Berman, 2019). In this prior work, all the robots must be reprogrammed with a new set of control policies if the target distribution is changed. In contrast, our approach can achieve new target swarm distributions via redesign of the control policy of a single leader agent, while the control policies of the swarm agents remain fixed. The follower agents switch stochastically out of their current location on the graph whenever the leader is at their location; in

this way, the leader has a "repulsive" effect on the followers. The transition rates out of each location are common to all the followers, and are therefore independent of the agents' identities. Using the mean-field model, herding objectives for the swarm are framed in terms of the distribution of the followers over the graph. The objective is to compute leader control policies that are functions of the agent distribution, rather than the individual agents' states, which makes the control policies scalable with the number of agents.

We apply RL-based approaches to the mean-field model to construct leader control policies that minimize the time required for the swarm of follower agents to converge to a user-defined target distribution. The RL-based control policies are not hindered by curse-of-dimensionality issues that arise in classical optimal control approaches. Additionally, RL-based approaches can more easily accommodate the stochastic nature of the follower agent transitions on the graph. There is prior work on RL-based control approaches for mean-field models of swarms in which each agent can localize itself in space and a state-dependent control policy can be assigned to each agent directly (Šošić *et al.*, 2018; Hüttenrauch *et al.*, 2019; Yang *et al.*, 2018). Other work by Nguyen *et al.* apply hierarchical deep RL to simplify the shepherding task by decomposing interactions between an aerial UAV and a small group of ground robot in order to simplify learning (Nguyen *et al.*, 2019). Later work by the group Nguyen *et al.* (2020) extended this approach to maintaining swarm formations between the UAV and ground robots while minimizing UAV traversal of the space. Similarly, Zhi *et al.* enabled a leader to herd three agents through obstacles to reach a goal point using a deep RL approach (Zhi and Lien, 2020). However, to our knowledge, there are no applications of deep RL to mean-field models for herding a swarm using a leader agent, aside from our work **Zahi M Kakish** *et al.* (2020).

Chapter 2

LEADER-BASED CONTROL OF A ROBOTIC SWARM USING MEAN-FIELD MODELS: DESCRIPTION AND IMPLEMENTATION

2.1 Summary

In this chapter, we introduce a model and a control approach for herding a swarm of "follower" agents to a target distribution among a set of states using a single "leader" agent. The follower agents evolve on a finite state space that is represented by a graph and transition between states according to a continuous-time Markov chain, whose transition rates are determined by the location of the leader agent. The control problem is to define a sequence of states for the leader agent that steers the probability density of the forward equation of the Markov chain. We demonstrate this control approach through numerical simulations with varied numbers of follower agents that evolve on graphs of different sizes, through a 3D multi-robot simulation in which a quadrotor is used to control the spatial distribution of eight ground robots over four regions, and through a physical experiment in which a swarm of ten robots is herded by a virtual leader over four regions.

2.2 Leader-Based Control Strategy Overview and Implementation

This section contains results from Elamvazhuthi *et al.* (2020).

Overview In this formulation, we use a single "leader" agent to herd a swarm of "follower" agents to a target distribution among a set of states. This control approach aims to drive the swarm of follower agents to the target distribution based on a

repulsive effect of the leader on the followers. Two leader controllers are developed: an open-loop controller, $u_e(t)$, and a closed-loop controller defined as $u_e(\mathbf{x}(t))$, where $\mathbf{x}(t)$ is the vector of probability distributions of the random variable $X_k(t)$ at time t. When a leader repulses the agents, the location of each follower agent evolves on the state space $\mathcal{V}$ according to the conditional probabilities

$$\mathbb{P}(X_i(t+h) = T(e)|X_i(t) = S(e)) = u_e(t)h + o(h), \tag{2.1}$$

which is a continuous-time version of (3.1). The author worked on two experimental setups to validate the open-loop and closed-loop controllers, which are elaborated on in Elamvazhuthi *et al.* (2020). The first setup is an experimental implementation of the open-loop controller on the *Robotarium* (Wilson *et al.*, 2020) with 10 robots and a virtual leader whose motion and state are projected on the testbed surface. The second setup is a *Gazebo* (Koenig and Howard, 2004) simulation with eight Pheeno robots (Wilson *et al.*, 2016) and a generic quadrotor in a 2.4 m $\times$ 2.4 m testbed and was used to simulate the closed-loop controller.

Physical Robot Experiments A multi-robot experiment was implemented using the *Robotarium* (Wilson *et al.*, 2020) to validate the closed-loop controller. The Robotarium is a swarm robotics testbed that users can remotely access to validate their controllers and algorithms on physical hardware. The experiment was conducted in a centralized manner, in that the robot population in each state was measured from images taken from multiple VICON motion capture cameras, and the robots initiated and completed their transitions between states when commanded by a central computer.

The environment was modeled as an undirected 4-vertex grid graph $\mathcal{G}$, and $N = 10$ robots were used as follower agents. The robots move on the testbed surface shown in Figure 2.1, which is divided into four regions of equal size, each of which corresponds to

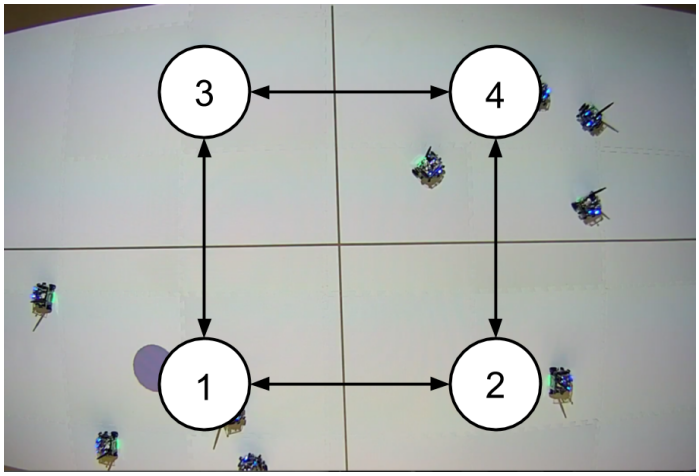


Figure 2.1: Initial setup of a physical experiment on the Robotarium swarm robotics testbed, with the graph $\mathcal{G}$ superimposed

a vertex of the graph $\mathcal{G}$ (superimposed on the testbed). A virtual leader agent, shown as the blue circle in Figure 2.1, and the boundaries of the four regions were projected onto the testbed using an overhead projector. The initial and target follower agent distributions were defined as $N\mathbf{x}^0 = [4\ 1\ 1\ 4]^T$ and $N\mathbf{x}^{eq} = [1\ 4\ 4\ 1]^T$, respectively. The leader moves along the path $\mathcal{W}^\infty = ((1,2),(2,4),(4,3),(3,1),(1,2),...)$. The leader remains stationary in its current state, repelling followers in that state, until the follower population in the leader's state is less than or equal to the target population; then, the leader transitions to the next state in its path.

During the experiment with the closed-loop controller, the leader is red if it is stationary at its current state, and blue if it is moving to the next state in its path. The current time step k and leader action (either *Stay* if it is stationary, or the direction of its motion) are displayed at the top of the testbed. The leader was able to herd the robots into the target distribution in 20 iterations, as shown in Figure 2.2, which plots the distribution of follower robots in each state over time.

Gazebo Simulation Results We also validated the closed-loop controller in a 3D physics simulation with realistic leader and follower robot dynamics. We used the

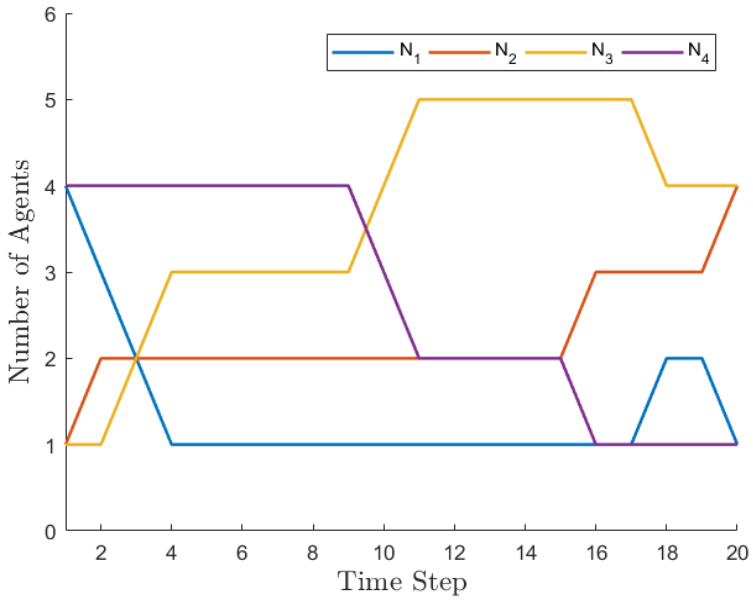


Figure 2.2: Follower distribution over time in the Robotarium experiment with the closed-loop controller.

Robot Operating System (ROS) to program the low-level and high-level control of the simulated robots in a completely decentralized manner, meaning that all robots take sensor measurements and decide on their next action autonomously, without the input of a supervisory agent or global observer. Additionally, each robot performs its computation and control independently of one another, with no inter-robot communication. We used Gazebo (Koenig and Howard, 2004) for 3D simulation and rendering. The graph $\mathcal{G}$ and leader path $\mathcal{W}^{\infty}$ were the same as in the numerical simulations.

In the simulation, a quadrotor acts as the leader, and $N = 8$ Pheeno robots (Wilson *et al.*, 2016) act as the followers. Pheeno is a customizable, low-cost mobile robot developed by our laboratory. Each simulated Pheeno is equipped with an upward-facing sonar sensor and a ground-facing IR sensor. These sensors model the functionality of the HC-SR04 Ultrasonic Sensor and the QRE1113 Digital IR Sensor, respectively.

The robots move within a 2.4 m $\times$ 2.4 m bounded arena, shown in Figure 2.3,

that is divided into four black or white regions of equal size. Each region corresponds to a vertex of the graph $\mathcal{G}$. The regions contain different colored blocks that are used by the quadrotor to assist with its localization and to identify the region (state) over which it is flying: state 1 is green, state 2 is blue, state 3 is red, and state 4 is orange. The visual servo approach in Chaumette and Hutchinson (2006) is used to localize the quadrotor with respect to the colored blocks in this way. The quadrotor determines the number of followers in its current state by counting the yellow circles on top of the Pheenos below. The eight pillars surrounding the arena are labeled with *ArUco* fiducial markers Romero-Ramirez *et al.* (2018), which the Pheenos use to determine the heading to the next state in their transition. The two pillars that are adjacent to each region (state) have the same marker. Each Pheeno uses its ground-facing IR sensor to recognize when it has crossed into a new region by sensing the change in reflection that occurs when it travels from a white surface to a black surface, or vice versa. Each Pheeno also uses its upward-facing sonar sensor to detect the quadrotor when it is hovering at a low altitude above the Pheeno.

The closed-loop controller is implemented in the simulation as follows. The quadrotor moves according to a series of equations in Elamvazhuthi *et al.* (2020) that are a function of the current density of robots in the leader's current state. If this number is less than or equal to the target number, then the quadrotor moves to the next state in the path. If the number exceeds the target number, then the quadrotor descends, which triggers the sonar sensors on the Pheenos in that state. These Pheenos then transition to adjacent states, following (2.1). The quadrotor repeats these actions until the number of agents in each state equals the target number.

We performed two simulations of the closed-loop controller, each with different initial and target distributions. In *Scenario 1*, $N\mathbf{x}^0 = [4\ 4\ 0\ 0]^T$ and $N\mathbf{x}^{eq} = [2\ 2\ 2\ 2]^T$. In *Scenario 2*, $N\mathbf{x}^0 = [2\ 2\ 2\ 2]^T$ and $N\mathbf{x}^{eq} = [1\ 1\ 3\ 3]^T$. Figures 2.4a and 2.4b plot

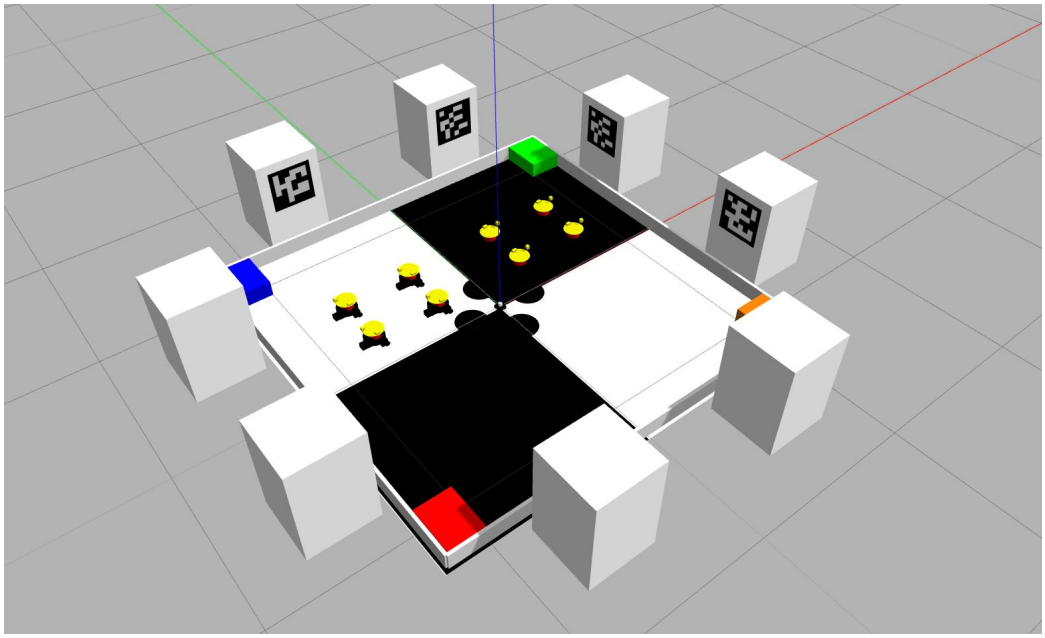


Figure 2.3: The *Gazebo* simulation testbed. Each pair of pylons located near a corner color block contains a fiducial marker with the state ID. These serve as guideposts for the Pheeno to locate their next transition state. The checker pattern on the testbed floor is used by a downward facing IR sensor located on each Pheeno detects the change from one state to another. The quadrotor uses the four colored blocks to localize itself within a state.

the distribution of followers over time for both scenarios. The figures show that the closed-loop controller successfully drove the followers to the target distribution in each scenario.

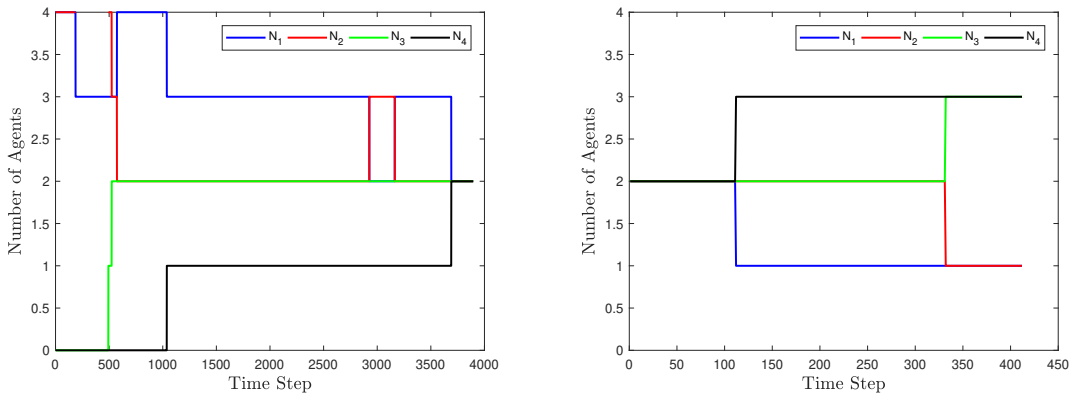


Figure 2.4: Follower distribution over time in the 3D simulation with the closed-loop controller. (a) Scenario 1; (b) Scenario 2.

Chapter 3

LEADER-BASED STRATEGIES FOR SWARM CONTROL DESIGNED USING TABULAR REINFORCEMENT LEARNING (RL) ALGORITHMS

3.1 Summary

In this chapter, we present a reinforcement learning approach to designing a control policy for a "leader" agent that herds a swarm of "follower" agents, via repulsive interactions, as quickly as possible to a target probability distribution over a strongly connected graph. The leader control policy is a function of the swarm distribution, which evolves over time according to a mean-field model in the form of an ordinary difference equation. The dependence of the policy on agent populations at each graph vertex, rather than on individual agent activity, simplifies the observations required by the leader and enables the control strategy to scale with the number of agents. Two Temporal-Difference learning algorithms, SARSA and Q-Learning, are used to generate the leader control policy based on the follower agent distribution and the leader's location on the graph. Both learning algorithms use a tabular approach to storing the trained state-action values Q. Therefore, agent population fractions are transformed to discretized representations of the robot populations for indexing within a multi-dimensional matrix. A simulation environment corresponding to a grid graph with 4 vertices was used to train and validate the control policies for follower agent populations ranging from 10 to 1000. Finally, the control policies trained on 1000 and 100 simulated agents were used to successfully redistribute a physical swarm of 10 small robots to a target distribution among 4 spatial regions. This provides evidence of the existence of a "mean-field" effect, whereby the time evolution of populations of

agents whose states evolve according to a discrete-time Markov chain becomes more deterministic as the number of agents in the swarm increases.

3.2 Problem Statement

We first define some notation from graph theory and matrix analysis that we use to formally state our problem. We denote by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ a directed graph with a set of M vertices, $\mathcal{V} = \{1, ..., M\}$, and a set of $N_{\mathcal{E}}$ edges, $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$, where $e = (i, j) \in \mathcal{E}$ if there is an edge from vertex $i \in \mathcal{V}$ to vertex $j \in \mathcal{V}$. We define a *source map* $\sigma : \mathcal{E} \to \mathcal{V}$ and a *target map* $\tau : \mathcal{E} \to \mathcal{V}$ for which $\sigma(e) = i$ and $\tau(e) = j$ whenever $e = (i, j) \in \mathcal{E}$. Given a vector $X \in \mathbb{R}^M$, X_i refers to the i^{th} coordinate value of X. For a matrix $A \in \mathbb{R}^{M \times N}$, A^{ij} refers to the element in the i^{th} row and j^{th} column of A.

We consider a finite swarm of N follower agents and a single leader agent. The locations of the leader and followers evolve on a graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{1, ..., M\}$ is a finite set of vertices and $\mathcal{E} = \{(i, j) \mid i, j \in \mathcal{V}\}$ is a set of edges that define the pairs of vertices between which agents can transition. The vertices in $\mathcal{V}$ represent a set of spatial locations obtained by partitioning the agents' environment. We will assume that the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is strongly connected and that there is a self-edge $(i, i) \in \mathcal{E}$ at every vertex $i \in \mathcal{V}$. We assume that the leader agent can count the number of follower agents at each vertex in the graph. The follower agents at a location v only decide to move to an adjacent location if the leader agent is currently at location v and is in a particular behavioral state. In other words, the presence of the leader *repels* the followers at the leader's location. The leader agent does not have a model of the follower agents' behavior.

The leader agent performs a sequence of transitions from one location (vertex) to another. The leader's location at time $k \in \mathbb{Z}_+$ is denoted by $\ell_1(k) \in \mathcal{V}$. In addition to the spatial state $\ell_1(k)$, the leader has a behavioral state at each time k,

defined as $\ell_2(k) \in \{0, 1\}$. The location of each follower agent $i \in \{1, ..., N\}$ is defined by a discrete-time Markov chain (DTMC) $X_i(k)$ that evolves on the state space $\mathcal{V}$ according to the conditional probabilities

$$\mathbb{P}(X_i(k+1) = \tau(e) \mid X_i(k) = \sigma(e)) = u_e(k) \tag{3.1}$$

For each $v \in \mathcal{V}$ and each $e \in \mathcal{E}$ such that $\sigma(e) = v \neq \tau(e)$, $u_e(k)$ is given by

$$u_e(k) = \begin{cases} \beta_e & \text{if } \ell_1(k) = \sigma(e) \text{ and } \ell_2(k) = 1, \\ 0 & \text{if } \ell_1(k) = \sigma(e) \text{ and } \ell_2(k) = 0, \\ 0 & \text{if } \ell_1(k) \neq \sigma(e), \end{cases} \tag{3.2}$$

where β_e are positive parameters such that $\sum_{\substack{e \in \mathcal{E} \\ v=\sigma(e) \neq \tau(e)}} \beta_e < 1$. Additionally, for each $v \in \mathcal{V}$, $u_{(v,v)}(k)$ is given by

$$u_{(v,v)}(k) = 1 \; - \sum_{\substack{e \in \mathcal{E} \\ v=\sigma(e) \neq \tau(e)}} u_e(k) \tag{3.3}$$

For each vertex $v \in \mathcal{V}$, we define a set of possible actions A_v taken by the leader when it is located at v:

$$A_v = \bigcup_{\substack{e \in \mathcal{E} \\ v=\sigma(e)}} \{e\} \times \{0, 1\} \tag{3.4}$$

The leader transitions between states in $\mathcal{V} \times \{0, 1\}$ according to the conditional probabilities

$$\mathbb{P}(\ell_1(k+1) = \tau(e), \ell_2(k+1) = d \mid \ell_1(k) = \sigma(e)) = 1 \tag{3.5}$$

if $p(k)$, the action taken by the leader at time k when it is at vertex v, is given by $p(k) = (e, d) \in A_v$.

The fraction, or *empirical distribution*, of follower agents that are at location $v \in \mathcal{V}$ at time k is given by $\frac{1}{N} \sum_{i=1}^{N} \chi_v(X_i(k))$, where $\chi_v(w) = 1$ if $w = v$ and 0 otherwise. Our goal is to learn a policy that navigates the leader between vertices

using the actions $p(k)$ such that the follower agents are redistributed ("herded") from their initial empirical distribution $\frac{1}{N}\sum_{i=1}^{N}\chi_v(X_i(0))$ among the vertices to a desired empirical distribution $\frac{1}{N}\sum_{i=1}^{N}\chi_v(X_i(T))$ at some final time T, where T is as small as possible. Since the identities of the follower agents are not important, we aim to construct a control policy for the leader that is a function of the current empirical distribution $\frac{1}{N}\sum_{i=1}^{N}\chi_v(X_i(k))$, rather than the individual agent states $X_i(k)$. However, $\frac{1}{N}\sum_{i=1}^{N}\chi_v(X_i(k))$ is not a state variable of the DTMC. In order to treat $\frac{1}{N}\sum_{i=1}^{N}\chi_v(X_i(k))$ as the state, we consider the *mean-field limit* of this quantity as $N\to\infty$. Let $\mathcal{P}(\mathcal{V}) = \{Y \in \mathbb{R}^M_{\geq 0};\ \sum_{v=1}^{M} Y_v = 1\}$ be the simplex of probability densities on $\mathcal{V}$. When $N\to\infty$, the empirical distribution $\frac{1}{N}\sum_{i=1}^{N}\chi_v(X_i(k))$ converges to a deterministic quantity $\hat{S}(k)\in\mathcal{P}(\mathcal{V})$, which evolves according to the following *mean-field model*, a system of difference equations that define the discrete-time Kolmogorov Forward Equation:

$$\hat{S}(k+1) = \sum_{e\in\mathcal{E}} u_e(k)B_e\hat{S}(k), \quad \hat{S}(0) = \hat{S}^0 \in \mathcal{P}(\mathcal{V}), \tag{3.6}$$

where B_e are matrices whose entries are given by

$$B_e^{ij} = \begin{cases} 1 & \text{if } i=\tau(e),\ j=\sigma(e), \\ 0 & \text{otherwise.} \end{cases}$$

The random variable $X_i(k)$ is related to the solution of the difference equation (3.6) by the relation $\mathbb{P}(X_i(k)=v) = \hat{S}_v(k)$.

We formulate an optimization problem that minimizes the number of time steps k required for the follower agents to converge to $\hat{S}_{target}$, the target distribution. In this optimization problem, the reward function is defined as

$$R(k) = -1\cdot\mathbb{E}||\hat{S}(k) - \hat{S}_{target}||^2. \tag{3.7}$$

Problem 3.2.1. *Given a target follower agent distribution $\hat{S}_{target}$, devise a leader control policy $\pi : \mathcal{P}(\mathcal{V})\times\mathcal{V}\to A$ that drives the follower agent distribution to $\hat{S}(T) =$*

$\hat{S}_{target}$, *where the final time* T *is as small as possible, by minimizing the total reward* $\sum_{k=1}^{T} R(k)$. *The leader action at time* k *when it is at vertex* $v \in \mathcal{V}$ *is defined as* $p(k) = \pi(\hat{S}(k), \ell_1(k)) \in A_v$ *for all* $k \in \{1, ..., T\}$, *where* $A = \cup_{v \in \mathcal{V}} A_v$.

3.2.1 Design of Leader Control Policies using Temporal-Difference Methods

Two Temporal-Difference (TD) learning methods (Sutton and Barto, 2018), *SARSA* and *Q-Learning*, were adapted to generate an optimal leader control policy. These methods use of bootstrapping provides the flexibility needed to accommodate the stochastic nature of the follower agents' transitions between vertices. Additionally, TD methods are model-free approaches, which are suitable for our control objective since the leader does not have a model of the followers' behavior. We compare the two methods to identify their advantages and disadvantages when applied to our swarm herding problem. Our approach is based on the mean-field model (3.6) in the sense that the leader learns a control policy using its observations of the population fractions of followers at all vertices in the graph.

Sutton and Barto (2018) provide a formulation of the two TD algorithms that we utilize. Let S denote the state of the environment, defined later in this section; A denote the action set of the leader, defined as the set A_v in Equation (3.4); and $Q(S, A)$ denote the state-action value function. We define $\alpha \in [0, 1]$ and $\gamma \in [0, 1]$ as the learning rate and the discount factor, respectively. The policy used by the leader is determined by a state-action pair (S, A). R denotes the reward for the implemented policy's transition from the current to the next state-action pair and is defined in Equation (3.7). In the SARSA algorithm, an on-policy method, the state-action value function is defined as:

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)] \tag{3.8}$$

where the update is dependent on the current state-action pair (S, A) and the next state-action pair (S', A') determined by enacting the policy. In the Q-Learning algorithm, an off-policy method, the state-action value function is:

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)] \tag{3.9}$$

Whereas the SARSA algorithm update (3.8) requires knowing the next action A' taken by the policy, the Q-learning update (3.9) does not require this information.

Both algorithms use a discretization of the observed state S and represent the state-action value function Q in tabular form as a multi-dimensional matrix, indexed by the leader actions and states. The state S is defined as a vector that contains a discretized form of the population fraction of follower agents at each location $v \in \mathcal{V}$ and the location $\ell_1(k) \in \mathcal{V}$ of the leader agent. The leader's spatial state $\ell_1(k)$ must be taken into account because the leader's possible actions depend on its current location on the graph. Since the population fractions of follower agents are continuous values, we convert them into discrete integer quantities serving as a discrete function approximation of the continuous fraction populations. Instead of defining F_v as the integer count of followers at location v, which could be very large, we reduce the dimensionality of the state space by discretizing the follower population fractions into D intervals and scaling them up to integers between 1 and D:

$$\begin{aligned} &F_v = \text{round}\left(\tfrac{D}{N} \textstyle\sum_{i=1}^{N} \chi_v(X_i(0))\right), \\ &\text{where} \quad F_v \in [1, \ldots, D], \quad v \in \mathcal{V}. \end{aligned} \tag{3.10}$$

For example, suppose $D = 10$. Then a follower population fraction of 0.24 at location v would have a corresponding state value $S_v = 2$. Using a larger value of D provides a finer classification of agent populations, but at the cost of increasing the size of the state S. Given these definitions, the state vector S is defined as:

$$S_{env} = [F_1, \ldots, F_M, \ell_1] \tag{3.11}$$

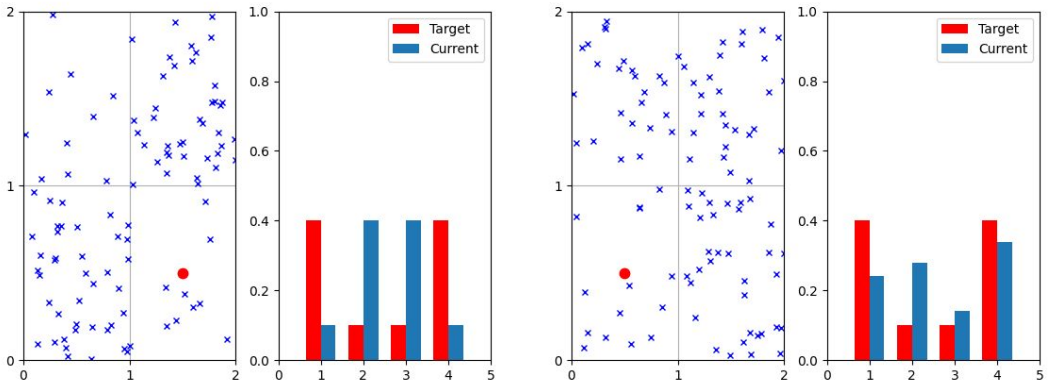
The state vector S_{env} contains many states that are inapplicable to the learning process. For example, the state vector for a 2×2 grid graph with $D = 10$ has $10 \times 10 \times 10 \times 10 \times 4$ possible variations, but only $10 \times 10 \times 10 \times 4$ are applicable since they satisfy the constraint that the follower population fractions at all vertices must sum up to 1 (note that the sum $\sum_v F_v$ may differ slightly from 1 due to the rounding used in Equation (3.10).) The new state S_{env} is used as the state S in the state-action value functions (3.8) and (3.9).

The leader's control policy for both functions (3.8) and (3.9) is the following ϵ-greedy policy, where $X \in [0, 1]$ is a uniform random variable and ϵ is a threshold parameter that determines the degree of state exploration during training:

$$\pi(S_{env}) = \arg\max_{A} Q(S_{env}, A) \quad \text{if } X > \epsilon \tag{3.12}$$

3.3 Simulation Results

An *OpenAI Gym* environment (Brockman *et al.*, 2016) was created in order to design, simulate, and visualize our leader-based herding control policies (**Zahi M Kakish**, 2019). This open source virtual environment can be easily modified to simulate swarm controllers for different numbers of agents and graph vertices, making it a suitable environment for training leader control policies using our model-free approaches. The simulated controllers can then be implemented in physical robot experiments. Figure 3.1 shows the simulated environment for a scenario with 100 follower agents, represented by the blue $\times$ symbols, that are herded by a leader, shown as a red circle, over a 2×2 grid. The *OpenAI* environment does not store the individual positions of each follower agent within a grid cell; instead, each cell is associated with an agent count. The renderer disperses agents randomly within a cell based on the cell's current agent count. The agent count for a grid cell is updated whenever an agent enters or leaves the cell according to the DTMC (3.1), and the



***k* = 0** ***k* = 50**

Figure 3.1: Visual rendering of a simulated scenario in our *OpenAI* environment for iterations $k = 0$ and 50. The environment simulates a strongly connected 2×2 grid graph such as the one shown in Figure 3.2. The leader (red circle) moves between grid cells in a horizontal or vertical direction. It may not move diagonally. Follower agents (blue $\times$ symbols) are randomly distributed in each cell. The borders of each cell are represented by the grid lines. The histogram to the right of each grid shows both the target (red) and current (blue) agent population fractions in each vertex at iteration k.

environment is re-rendered. Recording the agent counts in each cell rather than their individual positions significantly reduces memory allocation and computational time when training the leader control policy on scenarios with large numbers of agents.

The graph $\mathcal{G}$ that models the environment in Figure 3.1, with each vertex of $\mathcal{G}$ corresponding to a grid cell, was defined as the 2×2 graph in Figure 3.2. In the graph, agents transition along edges in either a horizontal or vertical direction, or they can stay at the current vertex. The action set is thus defined as:

$$A = [\ \textit{Left},\ \textit{Right},\ \textit{Up},\ \textit{Down},\ \textit{Stay}\ ] \tag{3.13}$$

Using the graph in Figure 3.2, we trained and tested a leader control policy for follower agent populations of $N = 10$–100 at 10-agent increments. Both the SARSA and Q-Learning paradigms were applied and trained on 5000 episodes with 5000 iterations each. In every episode, the initial distribution $\hat{S}_{initial}$ and target distribution

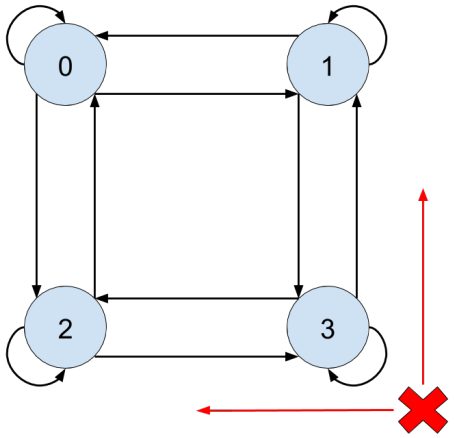


Figure 3.2: The bidirected grid graph $\mathcal{G}$ used in our simulated scenario. The leader agent (red $\times$ symbol) is at vertex 3. The movement options for the leader are *Left* to vertex 2 or *Up* to vertex 1. The leader can also *Stay* at vertex 3, where its presence triggers follower agents at the vertex to probabilistically transition to vertex 1 or vertex 2.

$\hat{S}_{target}$ of the follower agents were defined as:

$$\hat{S}_{initial} = \begin{bmatrix} 0.4 & 0.1 & 0.1 & 0.4 \end{bmatrix}^T \tag{3.14}$$

$$\hat{S}_{target} = \begin{bmatrix} 0.1 & 0.4 & 0.4 & 0.1 \end{bmatrix}^T \tag{3.15}$$

The initial leader location, ℓ_1, was randomized to allow many possible permutations of states S_{env} for training. During training, an episode completes once the distribution of N follower agents reaches a specified terminal state. Instead of defining the terminal state as the exact target distribution $\hat{S}_{target}$, which becomes more difficult to reach as N increases due to the stochastic nature of the followers' transitions, we define this state as a distribution that is sufficiently close to $\hat{S}_{target}$. The learning rate and discount factor were set to $\alpha = 0.3$ and $\gamma = 0.9$, respectively. The follower agent transition rate β_e was defined as the same value β for all edges e in the graph and was set to $\beta = 0.025$, 0.05, or 0.1. We use the mean squared error (MSE) to measure the difference between the current follower distribution and $\hat{S}_{target}$. The terminal

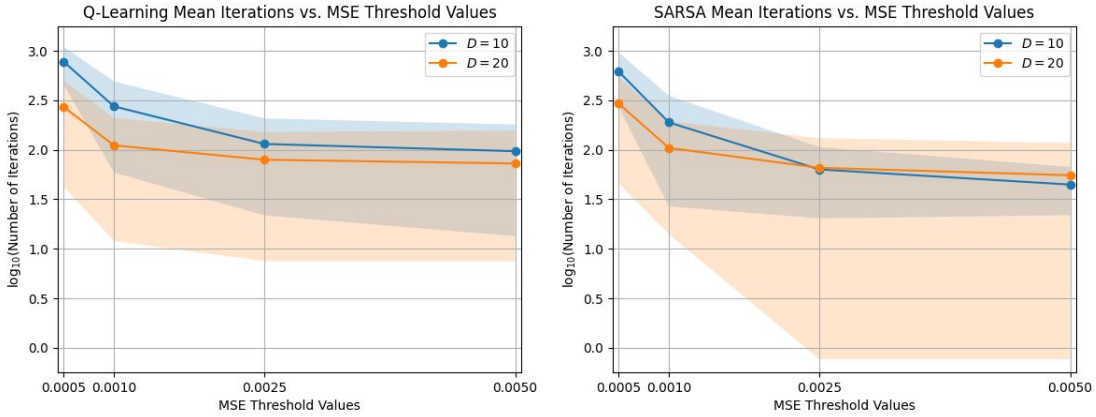


Figure 3.3: Number of iterations until convergence to $\hat{S}_{target}$ (plotted on a log scale) versus the MSE threshold value μ for leader control policies that were learned using Q-Learning (*left*) and SARSA (*right*) with $\beta = 0.05$ and $N = 100$ follower agents. Each circle on the plots marks the mean number of iterations until convergence over 1000 test runs of a leader policy in the simulated grid graph environment in Figure 3.2. The shaded regions indicate the range of ± 1 standard deviation about the mean numbers of iterations (blue for $D = 10$; orange for $D = 20$.)

state is reached when the MSE decreases below a threshold value μ. We trained our algorithms on threshold values of $\mu = 0.0005$, 0.001, 0.0025, and 0.005.

After training the leader control policies on each follower agent population size N, the policies were tested on scenarios with the same environment and value of N. The policy for each scenario was run 1000 times to evaluate its performance. The policies were compared for terminal states that corresponded to the four different MSE threshold values μ, and were given 1000 iterations to converge within the prescribed MSE threshold of the target distribution (3.15) from the initial distribution (3.14).

Figure 3.3 compares the performance of leader control policies that were designed using each TD algorithm as a function of the tested values of μ. The leader control policies were trained on $N = 100$ follower agents, using the parameters $\beta = 0.05$ and $D = 10$ or 20, and tested in simulations with $N = 100$. The plots show that for both policies, the mean number of iterations required to converge to $\hat{S}_{target}$ decreases as the threshold μ increases for constant D, and at low values of μ, the mean number

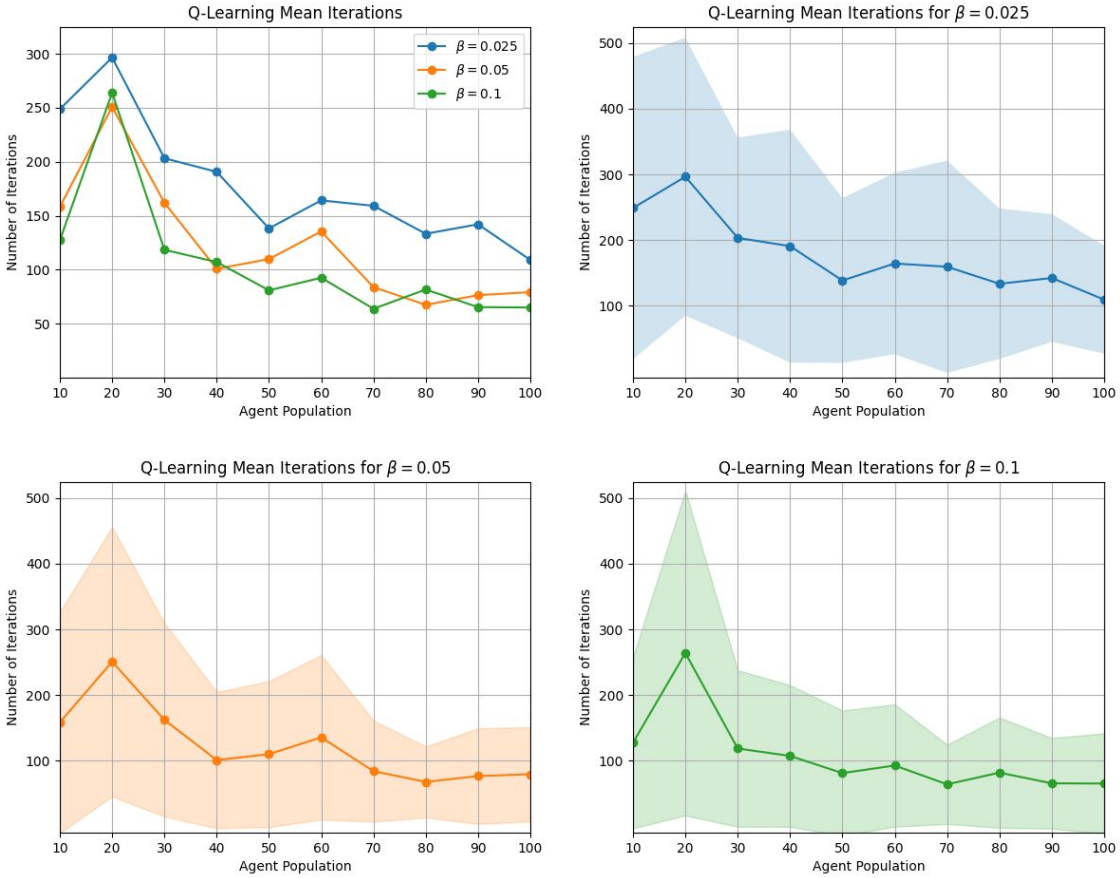


Figure 3.4: Number of iterations until convergence to $\hat{S}_{target}$ versus number of follower agents N for leader control policies that were learned using Q-Learning with $D = 20$ and $\mu = 0.0025$. Each circle on the plots marks the mean number of iterations until convergence over 1000 test runs of a leader policy in the simulated grid graph environment in Figure 3.2 with the same value of N that the policy was trained on. The plot for each β value in the top two figures are reproduced individually in the three figures below them, along with shaded regions that indicate the range of ± 1 standard deviation about the mean numbers of iterations.

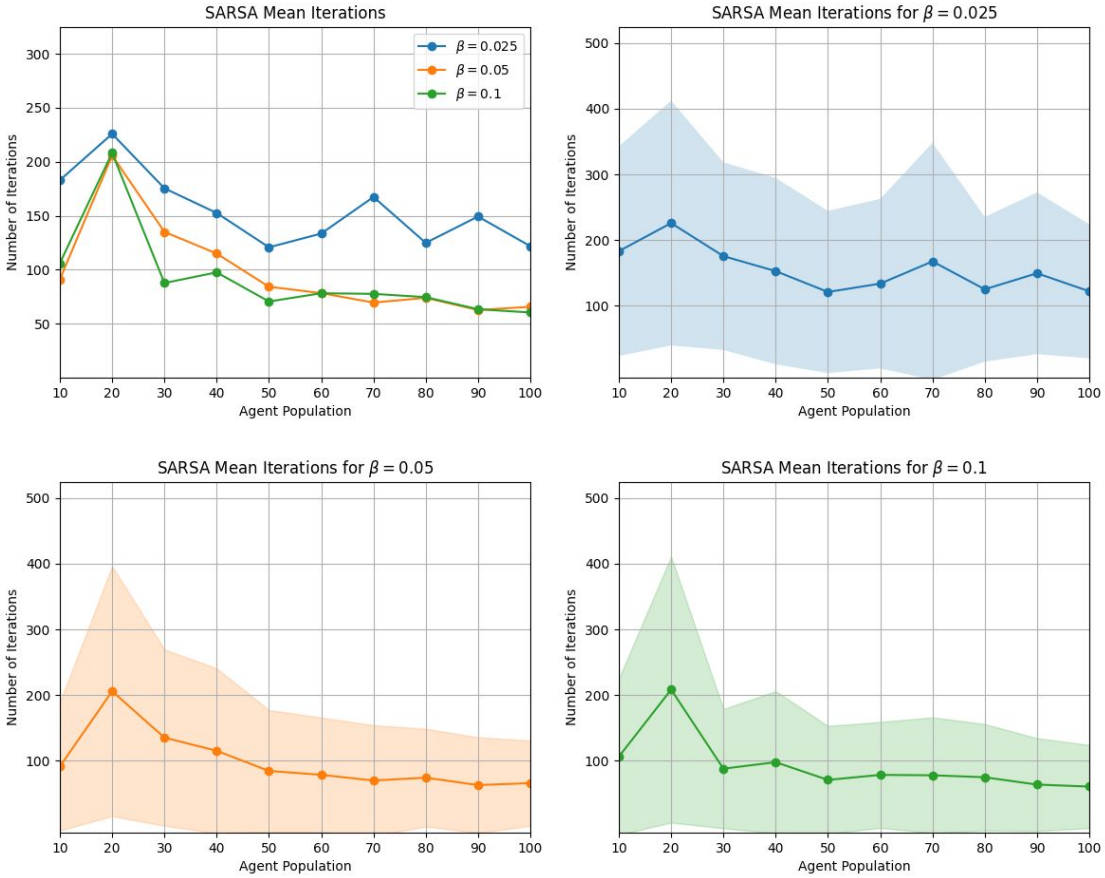


Figure 3.5: Number of iterations until convergence to $\hat{S}_{target}$ versus number of follower agents N for leader control policies that were learned using SARSA with $D = 20$ and $\mu = 0.0025$. Each circle on the plots marks the mean number of iterations until convergence over 1000 test runs of a leader policy in the simulated grid graph environment in Figure 3.2 with the same value of N that the policy was trained on. The plot for each β value in the top two figures are reproduced individually in the three figures below them, along with shaded regions that indicate the range of ± 1 standard deviation about the mean numbers of iterations.

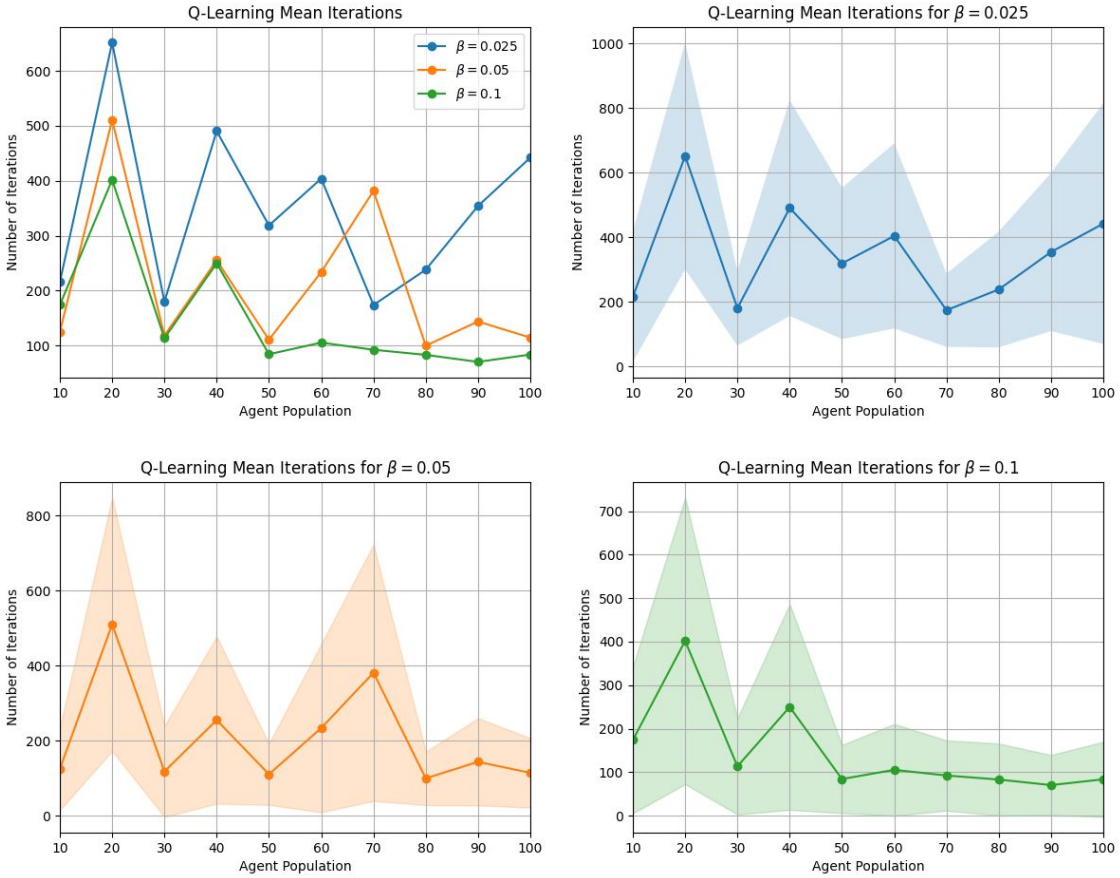


Figure 3.6: Number of iterations until convergence to $\hat{S}_{target}$ versus number of follower agents N for leader control policies that were learned using Q-Learning with $D = 10$ and $\mu = 0.0025$. Each circle on the plots marks the mean number of iterations until convergence over 1000 test runs of a leader policy in the simulated grid graph environment in Figure 3.2 with the same value of N that the policy was trained on. The plot for each β value in the top two figures are reproduced individually in the three figures below them, along with shaded regions that indicate the range of ± 1 standard deviation about the mean numbers of iterations.

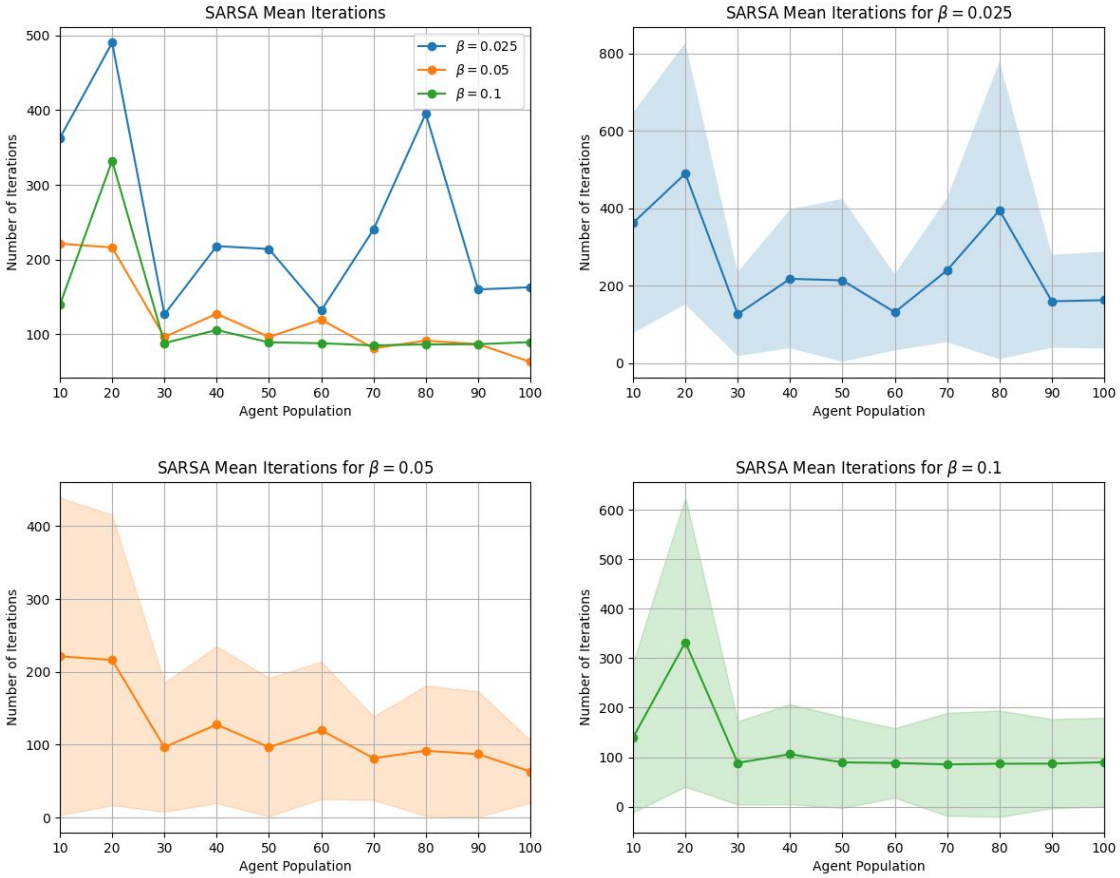


Figure 3.7: Number of iterations until convergence to $\hat{S}_{target}$ versus number of follower agents N for leader control policies that were learned using SARSA with $D = 10$ and $\mu = 0.0025$. Each circle on the plots marks the mean number of iterations until convergence over 1000 test runs of a leader policy in the simulated grid graph environment in Figure 3.2 with the same value of N that the policy was trained on. The plot for each β value in the top two figures are reproduced individually in the three figures below them, along with shaded regions that indicate the range of ± 1 standard deviation about the mean numbers of iterations.

of iterations decreases when D is increased. In addition, as μ increases, the variance in the number of iterations decreases (note the log scale of the y-axis in the plots) or remains approximately constant, except for the $D = 20$ case of SARSA.

Figures 3.4 and 3.5 compare the performance of leader control policies that were designed using each algorithm as a function of N, where the leader policies were tested in simulations with the same value of N that they were trained on. The other parameters used for training were $\mu = 0.0025$, $D = 20$, and $\beta = 0.025$, 0.05, or 0.1. The figures show that raising β from 0.05 to 0.1 does not significantly affect the mean number of iterations until convergence, while decreasing β from 0.05 to 0.025 results in a higher mean number of iterations. This effect is evident for both Q-Learning and SARSA trained leader control policies for $N > 50$. Both leader control policies result in similar numbers of iterations for convergence at each agent population size. Therefore, both the Q-Learning and SARSA training algorithms yield comparable performance for these scenarios.

The results in Figures 3.4 and 3.5 shows that as N increases above 50 agents, the mean number of iterations until convergence decreases slightly or remains approximately constant for all β values and for $\mu = 0.0025$. Moreover, from Figure 3.3, we see that MSE threshold values $\mu < 0.0025$ for $N = 100$ result in a higher number of iterations than the $N = 100$ cases in Figures 3.4 and 3.5.This trend may be due to differences in the magnitude of the smallest possible change in MSE over an iteration k relative to the MSE threshold μ for different values of N. For example, for $N = 10$, a similarity in iteration counts for all four MSE thresholds μ can be attributed to the fact that the change in the MSE due to a transition of one agent, corresponding to a change in population fraction of $1/N = 1/10$, is much higher than the four MSE thresholds (i.e., $(1/10)^2 > 0.005$, 0.0025, 0.001, and 0.0005). Compare this to the iteration count for $N = 50$, which would have a corresponding change in

MSE of $(1/50)^2$; this quantity is much smaller than 0.005 and 0.0025, but not much smaller than 0.001 and 0.0005. The iteration counts for $N = 100$ are much lower, since $(1/100)^2$ is much smaller than all four MSE thresholds.

Finally, Figure 3.8 compares the performance of leader control policies that were designed using each algorithm as a function of N, where the leader policies were trained with $N = 10$, 100, or 1000 follower agents and tested in simulations with $N = 10$–100 (at 10-agent increments) and $N = 1000$ agents. This was done to evaluate the robustness of the policies trained on the three agents populations to changes in N. The other parameters used for training were $\mu = 0.0025$, $D = 20$, and $\beta = 0.025$, 0.05, or 0.1. As the plots in Figure 3.8 show, policies trained on the smallest population, $N = 10$, yield an increased mean number of iterations until convergence when applied to populations $N > 10$. The reverse effect is observed, in general, for policies that are trained on higher values of N than they are tested on. An exception is the case where the policies are trained on $N = 100$ and 1000 and tested on $N = 10$, which produce much higher numbers of iterations than the policies that are both trained and tested on $N = 10$. This is likely a result of the large variance, and hence greater uncertainty, in the time evolution of such a small agent population. The lower amount of uncertainty in the time evolution of large swarms may make it easier for leader policies that are trained on large values of N to control the distribution of a given follower agent population than policies that are trained on smaller values of N. We thus hypothesize that training a leader agent with the mean-field model (3.6) instead of the DTMC model would lead to improved performance in terms of a lower training time, since the policy would only need to be trained on one value of N, and fewer iterations until convergence to the target distribution.

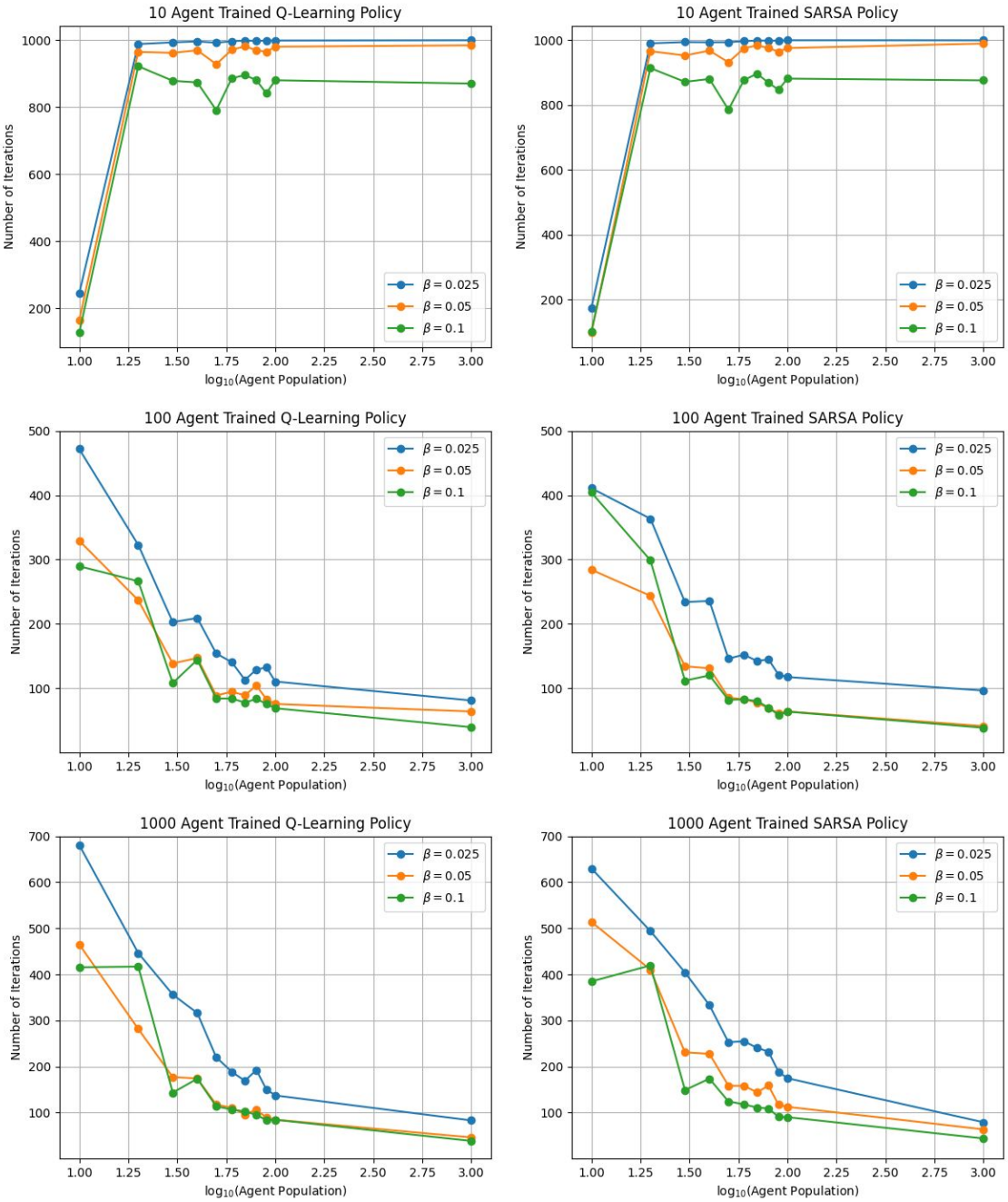


Figure 3.8: Number of iterations until convergence to $\hat{S}_{target}$ versus number of follower agents N (plotted on a log scale) for leader control polices that were trained using Q-Learning (*left column*) and SARSA (*right column*) with $D = 20$; $\mu = 0.0025$; and $\beta = 0.025$, 0.05, or 0.1; and $N = 10$, 100, or 1000 agents. Each circle on the plots marks the mean number of iterations until convergence over 1000 test runs of a leader policy in the simulated grid graph environment in Figure 3.2.

Figure 3.9: Number of iterations until convergence to $\hat{S}_{target}$ versus number of follower agents N (plotted on a log scale) for leader control polices that were trained using Q-Learning (*left column*) and SARSA (*right column*) with $D = 10$; $\mu = 0.0025$; and $\beta = 0.025$, 0.05, or 0.1; and $N = 10$, 100, or 1000 agents. Each circle on the plots marks the mean number of iterations until convergence over 1000 test runs of a leader policy in the simulated grid graph environment in Figure 3.2.

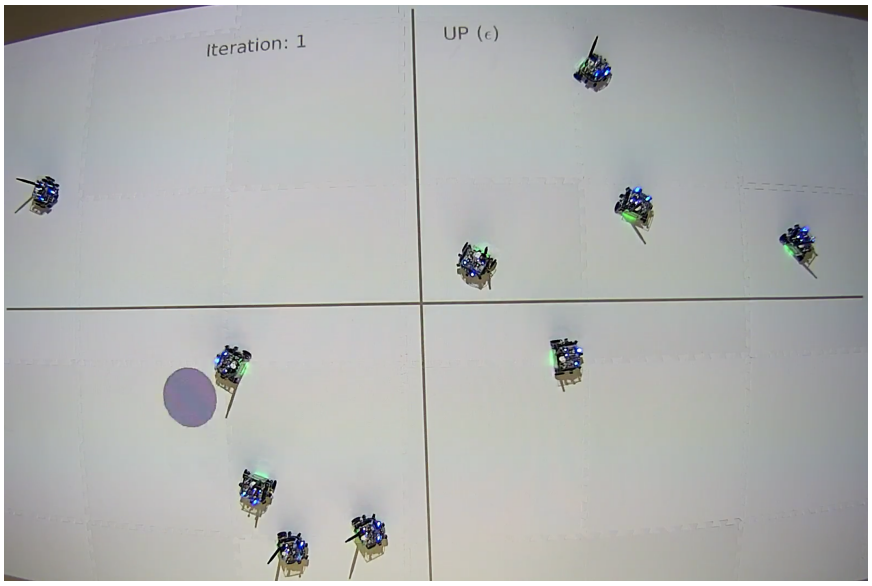


Figure 3.10: Initial setup of a physical experiment as seen from the fish-eye camera of the Robotarium testbed (Wilson *et al.*, 2020).

3.4 Experimental Results

We also conducted experiments to verify that our herding approach is effective in a real-world environment with physical constraints on robot dynamics and inter-robot spacing. Two of the leader control policies that were generated in the simulated environment were tested on a swarm of small differential-drive robots in the *Robotarium* (Wilson *et al.*, 2020), a remotely accessible swarm robotics testbed that provides an experimental platform for users to validate swarm algorithms and controllers. Experiments are set up in the Robotarium with MATLAB or Python scripts. The robots move to target locations on the testbed surface using a position controller and avoid collisions with one another through the use of barrier certificates (Wang *et al.*, 2017), a modification to the robots' controllers that satisfy particular safety constraints. To implement this collision-avoidance strategy, the robots' positions and orientations in a global coordinate frame are measured from images taken from multiple *VICON* motion capture cameras.

A video recording of our experiments is shown in **Zahi M Kakish** *et al.* (2020). The environment was represented as a 2×2 grid, as in the simulations, and $N = 10$ robots were used as follower agents. The leader agent, shown as the blue circle, and the boundaries of the four grid cells were projected onto the surface of the testbed using an overhead projector. As in the simulations, at each iteration k, the leader moves from one grid cell to another depending on the action prescribed by its control policy. Both the SARSA and Q-Learning leader control policies trained with $N = 100$ follower agents, $D = 10$, $\mu = 0.0025$, and a $\beta = 0.1$ were implemented, and **Zahi M Kakish** *et al.* (2020) shows the performance of both control policies. In the video, the leader is red if it is executing the *Stay* action and blue if it is executing any of the other actions in the set A (i.e., a movement action). The current iteration k and leader action are displayed at the top of the video frames. Actions that display ϵ next to them signify a random action as specified in (3.12). Each control policy was able to achieve the exact target distribution (3.15). The SARSA method took 59 iterations to reach this distribution, while the Q-Learning method took 23 iterations.

Chapter 4

LEADER-BASED STRATEGIES FOR SWARM CONTROL FROM DEEP RL ALGORITHMS TRAINED ON MEAN-FIELD MODELS

4.1 Summary

In this chapter, we present a novel deep reinforcement learning (RL) approach to designing a control policy for a leader agent that herds a swarm of "follower" agents, via repulsive interactions, as quickly as possible to a target swarm distribution over a strongly connected graph. The leader control policy is a function of the swarm distribution, which evolves over time according to a mean-field model in the form of an ordinary difference equation. The dependence of the policy on agent populations at each graph vertex, rather than on individual agent activity, simplifies the observations required by the leader and enables the control strategy to scale with the number of agents. The leader control policy is devised using an actor-critic RL model and a neural network function approximator trained on outputs of a modified Kolmogorov Forward Equation, which removes the requirement to specify numbers of agents by instead using population fractions. A 3D simulation environment corresponding to a grid graph with 4 vertices and 20 robots is used to validate the approach.

4.2 Problem Statement

We first define some notation from graph theory and matrix analysis that we use to formally state our problem. We denote by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ a directed graph with a set of M vertices, $\mathcal{V} = \{1, ..., M\}$, and a set of $N_{\mathcal{E}}$ edges, $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$, where $e = (i, j) \in \mathcal{E}$ if there is an edge from vertex $i \in \mathcal{V}$ to vertex $j \in \mathcal{V}$. We define a *source map* $\sigma : \mathcal{E} \to \mathcal{V}$

and a *target map* $\tau : \mathcal{E} \to \mathcal{V}$ for which $\sigma(e) = i$ and $\tau(e) = j$ whenever $e = (i, j) \in \mathcal{E}$. Given a vector $X \in \mathbb{R}^M$, X_i refers to the i^{th} coordinate value of X. For a matrix $A \in \mathbb{R}^{M \times N}$, A^{ij} refers to the element in the i^{th} row and j^{th} column of A.

We consider a finite swarm of N follower agents and a single leader agent. The locations of the leader and followers evolve on a graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{1, ..., M\}$ is a finite set of vertices and $\mathcal{E} = \{(i, j) \mid i, j \in \mathcal{V}\}$ is a set of edges that define the pairs of vertices between which agents can transition. The vertices in $\mathcal{V}$ represent a set of spatial locations obtained by partitioning the agents' environment. We will assume that the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is strongly connected and that there is a self-edge $(i, i) \in \mathcal{E}$ at every vertex $i \in \mathcal{V}$. We assume that the leader agent can count the number of follower agents at each vertex in the graph. The follower agents at a location v move to an adjacent location with a predefined probability if the leader agent is currently at location v and is in a particular behavioral state; otherwise, the followers remain at v. In other words, the presence of the leader *repels* the followers at the leader's location. The leader agent does not have a model of the follower agents' behavior.

The leader agent performs a sequence of transitions from one location (vertex) to another. The leader's location at time $k \in \mathbb{Z}_+$ is denoted by $\ell_1(k) \in \mathcal{V}$. In addition to the spatial state $\ell_1(k)$, the leader has a behavioral state at each time k, defined as $\ell_2(k) \in \{0, 1\}$. The location of each follower agent $i \in \{1, ..., N\}$ is defined by a discrete-time Markov chain (DTMC) $X_i(k)$ that evolves on the state space $\mathcal{V}$ according to the conditional probabilities

$$\mathbb{P}(X_i(k+1) = \tau(e) \mid X_i(k) = \sigma(e)) = u_e(k) \tag{4.1}$$

For each $v \in \mathcal{V}$ and each $e \in \mathcal{E}$ such that $\sigma(e) = v \neq \tau(e)$, $u_e(k)$ is given by

$$u_e(k) = \begin{cases} \beta_e & \text{if } \ell_1(k) = \sigma(e) \text{ and } \ell_2(k) = 1, \\ 0 & \text{if } \ell_1(k) = \sigma(e) \text{ and } \ell_2(k) = 0, \\ 0 & \text{if } \ell_1(k) \neq \sigma(e), \end{cases} \tag{4.2}$$

where β_e are positive parameters such that $\sum_{\substack{e \in \mathcal{E} \\ v=\sigma(e)\neq\tau(e)}} \beta_e < 1$. Additionally, for each $v \in \mathcal{V}$, $u_{(v,v)}(k)$ is given by

$$u_{(v,v)}(k) = 1 \; - \sum_{\substack{e \in \mathcal{E} \\ v=\sigma(e)\neq\tau(e)}} u_e(k) \tag{4.3}$$

For each vertex $v \in \mathcal{V}$, we define a set of possible actions A_v taken by the leader when it is located at v:

$$A_v = \bigcup_{\substack{e \in \mathcal{E} \\ v=\sigma(e)}} \{e\} \times \{0, 1\} \tag{4.4}$$

The leader transitions between states in $\mathcal{V} \times \{0, 1\}$ according to the conditional probabilities

$$\mathbb{P}(\ell_1(k+1) = \tau(e), \ell_2(k+1) = d \mid \ell_1(k) = \sigma(e)) = 1 \tag{4.5}$$

if $p(k)$, the action taken by the leader at time k when it is at vertex v, is given by $p(k) = (e, d) \in A_v$, where $d \in [0, 1]$ is the leader's behavioral state.

The fraction, or *empirical distribution*, of follower agents that are at location $v \in \mathcal{V}$ at time k is given by $\frac{1}{N}\sum_{i=1}^{N} \chi_v(X_i(k))$, where $\chi_v(w) = 1$ if $w = v$ and 0 otherwise. Our goal is to learn a policy that navigates the leader between vertices using the actions $p(k)$ such that the follower agents are redistributed ("herded") from their initial empirical distribution $\frac{1}{N}\sum_{i=1}^{N} \chi_v(X_i(0))$ among the vertices to a desired empirical distribution $\frac{1}{N}\sum_{i=1}^{N} \chi_v(X_i(T))$ at some final time T, where T is as small as possible. Since the identities of the follower agents are not important, we aim

to construct a control policy for the leader that is a function of the current empirical distribution $\frac{1}{N}\sum_{i=1}^{N}\chi_v(X_i(k))$, rather than the individual agent states $X_i(k)$. However, $\frac{1}{N}\sum_{i=1}^{N}\chi_v(X_i(k))$ is not a state variable of the DTMC. In order to treat $\frac{1}{N}\sum_{i=1}^{N}\chi_v(X_i(k))$ as the state, we consider the *mean-field limit* of this quantity as $N\to\infty$. Let $\mathcal{P}(\mathcal{V})=\{Y\in\mathbb{R}^M_{\geq 0};\ \sum_{v=1}^{M}Y_v=1\}$ be the simplex of probability densities on $\mathcal{V}$. When $N\to\infty$, the empirical distribution $\frac{1}{N}\sum_{i=1}^{N}\chi_v(X_i(k))$ converges to a deterministic quantity $\hat{S}(k)\in\mathcal{P}(\mathcal{V})$, which evolves according to the following *mean-field model*, a system of difference equations that define the discrete-time Kolmogorov forward equation:

$$\hat{S}(k+1)=\sum_{e\in\mathcal{E}}u_e(k)B_e\hat{S}(k),\quad \hat{S}(0)=\hat{S}^0\in\mathcal{P}(\mathcal{V}), \tag{4.6}$$

where B_e are matrices whose entries are given by

$$B_e^{ij}=\begin{cases}1 & \text{if } i=\tau(e),\ j=\sigma(e),\\ 0 & \text{otherwise.}\end{cases}$$

The random variable $X_i(k)$ is related to the solution of the difference equation (4.6) by the relation $\mathbb{P}(X_i(k)=v)=\hat{S}_v(k)$. We define $\mathcal{Z}_i(k)$ as the set of edges $e=(i,j)\in\mathcal{E}$ for which $i=\ell_1(k)$, the leader's location at time k. For the example 2×2 grid graph in Fig. 4.1, in which the leader is currently at vertex $\ell_1(k)=i=3$, this edge set is $\mathcal{Z}_i(k)=\{(3,3),(3,2),(3,1)\}$ (note the inclusion of a self-edge). We introduce a matrix $H_i(k)$, defined as

$$H_i(k)=\sum_{z\in\mathcal{Z}_i(k)}B_z, \tag{4.7}$$

and a vector $u_i(k)$, defined as

$$u_i(k)=\sum_{z\in\mathcal{Z}_i(k)}u_z(k), \tag{4.8}$$

where $u_z(k)$ is given by (4.2) for each $z\in\mathcal{Z}_i(k)$.

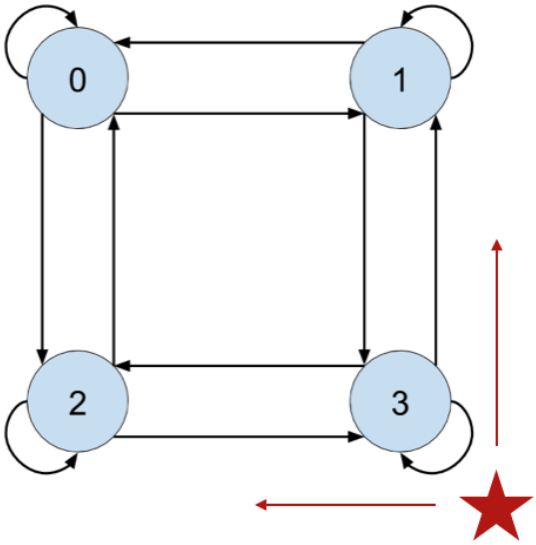


Figure 4.1: A strongly connected grid graph $\mathcal{G}$. The leader, shown as a red star, is located at vertex $\ell_1(k) = 3$. It can either stay at this vertex, where its presence triggers follower agents at the vertex to probabilistically transition to vertex 1 or vertex 2, or move in the direction of a red arrow to one of these vertices.

The modified discrete-time Kolmogorov forward equation is defined as

$$\hat{S}(k+1) = \sum_{z \in \mathcal{Z}_i(k)} u_i(k) H_i \hat{S}(k), \quad \hat{S}(0) = \hat{S}^0 \in \mathcal{P}(\mathcal{V}), \tag{4.9}$$

where i is the leader's current location $\ell_1(k)$. This model now incorporates the effect of the leader's presence at a given sequence of vertices on the time evolution of the follower agent distribution over the graph.

We formulate an optimization problem that minimizes the number of time steps k required for the follower agents to converge to $\hat{S}_{target}$, the target distribution:

Problem 4.2.1. *Given a target follower agent distribution $\hat{S}_{target}$, and defining $A = \cup_{v \in \mathcal{V}} A_v$, devise a leader control policy $\pi : \mathcal{P}(\mathcal{V}) \times \mathcal{V} \to A$ that drives the follower agent distribution to $\hat{S}(T) = \hat{S}_{target}$, where the final time T is as small as possible, by maximizing the total reward $\sum_{k=1}^{T} R(k)$. The leader action at time $k \in \{1, ..., T\}$ when it is at vertex $v \in \mathcal{V}$ is defined as $p(k) = \pi(\hat{S}(k), \ell_1(k)) \in A_v$.*

4.3 Design of Leader Control Policy using an Actor-Critic Method

Our previous work in Chapter 3 tested two tabular Temporal-Difference (TD) algorithms, Q-Learning and SARSA Sutton and Barto (2018), that learned the leader control policy from the follower agent state dynamics generated by the DTMC defined in (4.1). This work demonstrated that RL is a viable optimization technique for leader-based swarm herding. In contrast to this model-free learning approach, here we apply a model-based deep RL approach that learns the leader control policy from the solution to the mean-field model (4.9), which consists of the follower agent populations at each location (vertex) over time. We modify the mean-field model to incorporate the effect of the leader's presence on the follower agents' movement over the vertices of the graph.

Our model-based approach uses an Actor-Critic policy gradient algorithm Sutton and Barto (2018), which has certain advantages over analogous value-based deep RL variants such as Deep Q-Learning Mnih *et al.* (2013). In tabular value-based RL approaches, the state-action value function $Q(s, a)$ can be artificially modified to implement out-of-bound action conditions. However, this is not possible in Deep Q-Learning algorithms, since they rely on a function approximator instead of a multi-dimensional matrix for estimating state-action Q values. Since here we use a model-based approach with a bilinear system model, we utilize Actor-Critic policy optimization, which can be easily applied to models in this form Kamalapurkar *et al.* (2018).

We begin by reviewing the basics of a policy-based reinforcement learning algorithm before defining an Actor-Critic algorithm. Consider a Markov decision process (MDP) $\mathcal{M}(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ with a set of states $\mathcal{S}$; a set of actions $\mathcal{A}$; a state transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, which may be deterministic or stochastic; and a reward

function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. In our scenario, a leader agent in a particular state $s_t \in \mathcal{S}$ at time t undergoes an action $a_t \in \mathcal{A}$ dictated by a policy $\pi : \mathcal{S} \to \mathcal{A}$, which is a mapping of the state space to action probabilities, *i.e.* the probability of doing action a given the state, typically denoted as $\pi(a|s)$. The leader then transitions to state s_{t+1} at time $t+1$ according to the transition function $\mathcal{P}(s_{t+1}|s_t, a_t)$. Upon transitioning to the new state s_{t+1}, the leader acquires a reward $r_t = \mathcal{R}(s_t, a_t)$ and records its *experience*, defined as the tuple (s_t, a_t, r_t). The leader repeats this process at each time step until a final time T. The sequence of tuples from time $t = 0$ to $t = T$ is defined as the trajectory τ. For a given trajectory, the sum of the rewards is defined as the *return* $G(\tau) = \sum_{t=0}^{T} r_t$. We can choose to weight the rewards at times $t = 1, ..., T$ as follows, where $\gamma \in [0, 1]$ is a discount factor, making our return a discounted sum:

$$G(\tau) = r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots + \gamma^T r_T = \sum_{t=0}^{T} \gamma^t r_t$$

Values of γ that are close to 0 place greater weight on older rewards than newer ones. A return with $\gamma = 1$ is considered a Monte Carlo return, since it weights all rewards of the trajectory equally. Given a policy parameterized by the learnable vector of parameters $\theta \in \mathbb{R}^2$, a policy-based RL algorithm computes θ as the vector that maximizes the following cost function $J(\pi_\theta)$,

$$J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[G(\tau)].$$

Thus, our algorithm computes the vector θ that produces a policy π_θ which maximizes the expected return for a trajectory τ. This optimization problem is solved using stochastic gradient descent with a learning rate $\alpha \in \mathbb{R}$. At each step of the optimization procedure, θ is computed as

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta). \tag{4.10}$$

The policy gradient used in Sutton and Barto (2018) is defined as follows:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} G_t(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)\right], \tag{4.11}$$

where $G_t(\tau)$ is the discounted return from time t to T for the trajectory τ, defined as

$$G_t(\tau) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^T r_T = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}. \tag{4.12}$$

Observe that the discount factor γ has a power raised to begin at time t. Equation (4.11) is the basis for the REINFORCE policy gradient algorithm Sutton and Barto (2018); Williams (1992). However, defining the discounted return $G_t(\tau)$ as the weighted sum of accumulated rewards per trajectory τ can produce significant variance in the policy gradient $\nabla_\theta J(\pi_\theta)$ Graesser and Keng (2019). We can reduce the variance in $\nabla_\theta J(\pi_\theta)$ by subtracting an action-independent baseline value or function from $G_t(\tau)$. A commonly used baseline is a policy-dependent state value function $V^\pi : \mathcal{S} \to \mathbb{R}$. Thus, the policy gradient becomes

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} (G_t(\tau) - V^\pi(s_t))\nabla_\theta \log \pi_\theta(a_t|s_t)\right].$$

The selection of a baseline guides the definition of the Actor-Critic algorithm Sutton and Barto (2018). The discounted return for the leader agent's trajectory is considered an estimate of a state-action value function, $Q^\pi(s_t, a_t) = \mathbb{E}[G_t|s_t, a_t]$, for which each reward received during the trajectory is action-dependent. The difference between the state-action value function Q^π (the *actor*) estimated from the trajectory resulting from a given policy π_θ and the state value function V^π (the *critic*) of the policy π is defined as the *advantage function*,

$$A^\pi(s_t,\ a_t) \ = \ Q^\pi(s_t,\ a_t) - V^\pi(s_t). \tag{4.13}$$

Thus, our policy gradient algorithm becomes an Advantage Actor-Critic (A2C) algorithm Mnih *et al.* (2016); Graesser and Keng (2019), in which

$$\nabla_\theta J(\pi_\theta) \ = \ \mathbb{E}_t\big[A_t^\pi(s_t,\ a_t)\nabla_\theta \log \pi_\theta(a_t|s_t)\big]. \tag{4.14}$$

To calculate the return of a trajectory, we use a *full-rollout* Monte Carlo estimate instead of an n-step Temporal Difference (TD) estimate of Q^{π}, as we did in our prior work **Zahi M Kakish** *et al.* (2020). In Amiranashvili *et al.* (2018), it was shown that for certain reward attributes such as noisy rewards, sparse rewards, or delayed rewards, RL-trained policies with returns calculated using Monte Carlo estimates outperform policies with returns calculated using TD estimates. As shall be discussed in Section 4.3.1, to train the leader control policy with our particular action set, we use a reward function with characteristics similar to those of sparse and delayed rewards.

Suppose that at time k, the leader agent can measure the population fraction $F_v(k)$ of follower agents at P locations v in some set $\mathcal{V}_f \subseteq \mathcal{V}$, where

$$F_v(k) = \frac{1}{N} \sum_{i=1}^{N} \chi_v(X_i(k)). \tag{4.15}$$

The system state $S_{env}(k)$ at time k is defined as a vector that contains the location $\ell_1(k) \in \mathcal{V}$ of the leader and the population fractions $F_v(k)$ of follower agents at the P observed locations $v \in \mathcal{V}_f \equiv \{v_1, ..., v_P\}$:

$$S_{env}(k) = [F_{v_1}(k), \ \ldots \ , \ F_{v_P}(k), \ \ell_1(k)]. \tag{4.16}$$

The leader's spatial state $\ell_1(k)$ must be included because the leader's possible actions over the next time step depend on its current location on the graph. In the case where $\mathcal{V}_f = \mathcal{V}$, the system state is considered *fully observable* since it includes the follower agent population fractions at every vertex and the leader's position:

$$S_{env}(k) = [F_1(k), \ \ldots \ , \ F_M(k), \ \ell_1(k)]. \tag{4.17}$$

In the case where the population fraction of follower agents can only be measured at the leader's current spatial state $\ell_1(k)$, we refer to the system state, defined below, as *locally observable*:

$$S_{env}(k) = [F_{\ell_1(k)}(k), \ \ \ell_1(k)]. \tag{4.18}$$

The leader can perform an action from the following action set:

$$A = [\ \textit{Left},\ \textit{Right},\ \textit{Up},\ \textit{Down},\ \textit{Stay}\ ]. \tag{4.19}$$

The first four actions indicate that the leader moves to an adjacent vertex on the graph in the specified direction (see Fig. 4.1 for an example), and the *Stay* action indicates that the leader remains at its current vertex and changes its behavioral state to $\ell_2 = 1$. If the next action following a *Stay* action is a motion action, the leader changes its behavioral state back to $\ell_2 = 0$ and proceeds to move to the next state.

4.3.1 Reward Function Definition

Our objective is to solve the optimization problem in Problem 1, which minimizes the number of time steps k required for the follower agents to converge to the target distribution $\hat{S}_{target}$. We define the reward function in the optimization problem such that it outputs a negative number. In our previous work **Zahi M Kakish** *et al.* (2020), we defined the reward function as the Mean-Squared Error (MSE),

$$R(k) = -1 \cdot \mathbb{E}||\hat{S}(k) - \hat{S}_{target}||^2. \tag{4.20}$$

The MSE reward function outputs increasing rewards as the distribution at k converges towards $\hat{S}_{target}$.

In this paper, the distribution of follower agents over the graph at time k is characterized as the follower population fraction $F_v(k)$ at each vertex $v \in \mathcal{V}$. This set of population fractions can alternatively be viewed as the probability density of a single follower agent's location on the graph at time k, which allows us to define the reward function using relative entropy statistical measures. In particular, we use the negative Kullback–Leibler (KL) divergence Kullback and Leibler (1951) as the reward function. The choice of logarithm used in the KL divergence modifies the scale of

the calculated reward. Therefore, we investigate two versions of this measure, which differ in the base of the logarithm. We will refer to the measure with the natural logarithm as the *KL Divergence*,

$$R(k) = -1 \cdot D_{KL}(\hat{S}(k)||\hat{S}_{target}) = -1 \cdot \sum_{v \in \mathcal{V}} \hat{S}_v(k) \cdot \log\left(\frac{\hat{S}_v(k)}{\hat{S}_{v,target}}\right), \tag{4.21}$$

and the measure with the base-10 logarithm as the *log_{10} KL Divergence*,

$$R(k) = -1 \cdot D_{KL}(\hat{S}(k)||\hat{S}_{target}) = -1 \cdot \sum_{v \in \mathcal{V}} \hat{S}_v(k) \cdot \log_{10}\left(\frac{\hat{S}_v(k)}{\hat{S}_{v,target}}\right). \tag{4.22}$$

Figure 4.2 plots the values of each of the three reward functions above over the same leader trajectory spanning 100 iterations within a single trajectory. Both the MSE and the $\log_{10}$ KL Divergence return nearly equivalent rewards. The KL Divergence has the largest difference in returned rewards, but it trends toward similar rewards as the other two reward functions as $\hat{S}_v$ converges to $\hat{S}_{v,target}$. The MSE and KL Divergence rewards are approximately 2.2 times lower than the $\log_{10}$ KL Divergence rewards. Consequently, we hypothesize that the three reward functions will result in similar control policy performance, holding all other parameters of the training procedure constant.

4.3.2 Stopping Criteria

In addition to testing the MSE and the two versions of the KL Divergence as reward functions, we will test their effectiveness as metrics for defining stopping criteria for ending an episode. Since the reward functions are measures of the distance between the current and target follower agent distributions, their absolute value can be used to define stopping criteria that determine when the agent distribution is sufficiently close to the target distribution. We specify that the stopping criterion is met if $|R(k)| \leq \mu$, where μ is a positive constant and $R(k)$ is defined by Eq. (4.20), (4.21), or (4.22).

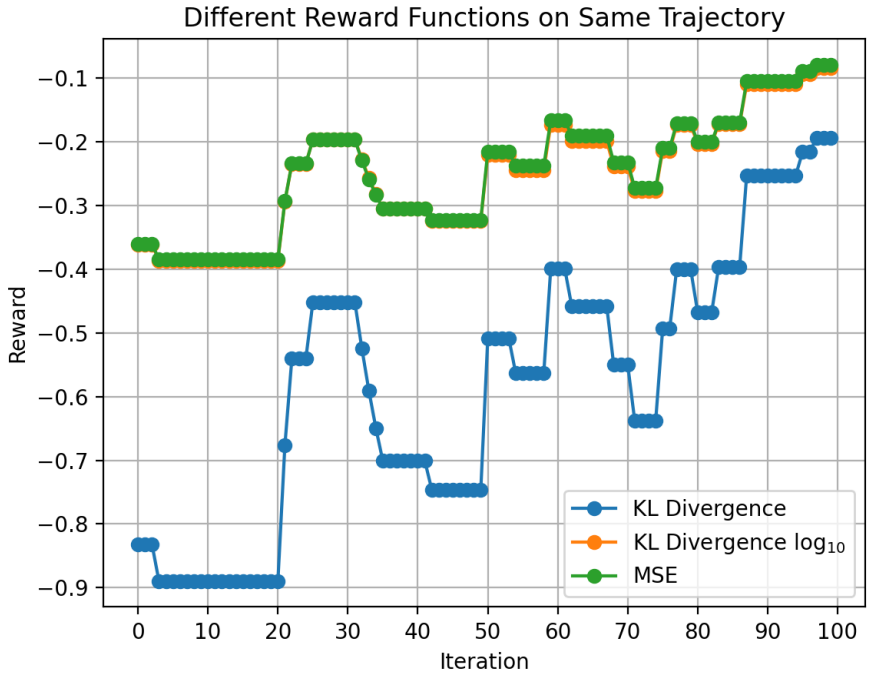
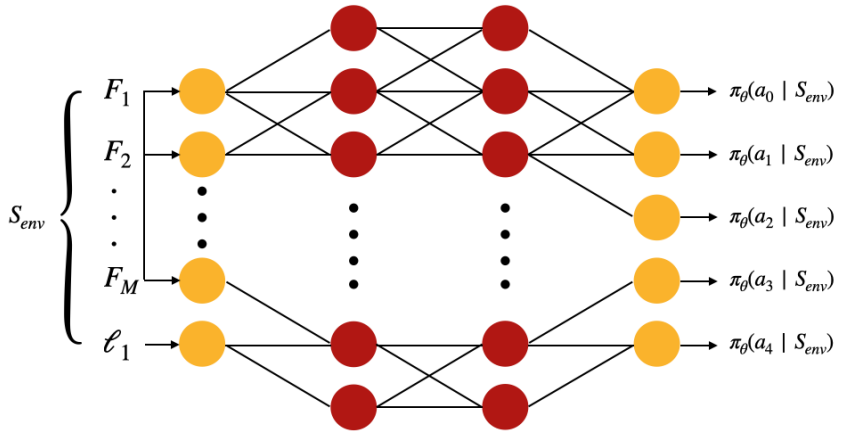


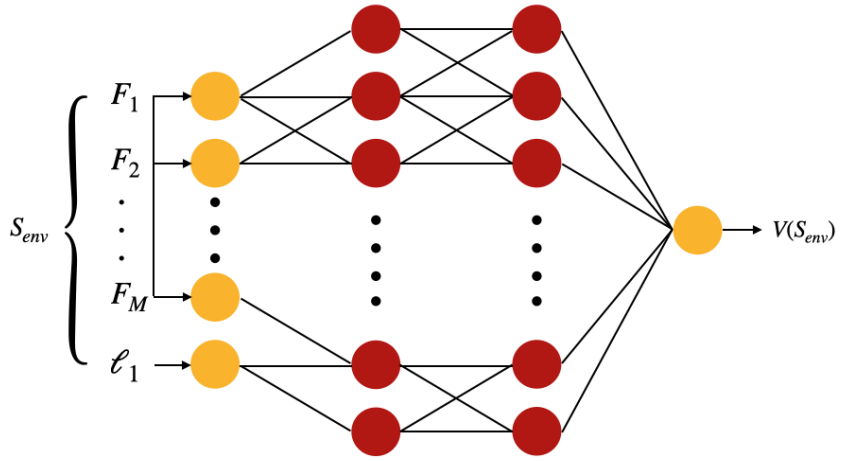
Figure 4.2: Reward dynamics of different reward functions, given the same leader trajectory and the action set (4.19).

4.4 Neural Network Function Approximation for Control Policy

Our Actor-Critic algorithm uses a dual multi-layer perceptron (MLP) neural network to approximate the value function V^{π} and the policy π_{θ} defined in Section 4.3. Figures 4.3 and 4.4 illustrate the neural networks and their respective input and output variables. In the fully observable policy network, both the Actor and Critic networks have the input S_{env} defined in (4.17). The output of the Actor network is the policy's probability of doing action $a \in A$ given the current system state S_{env}, and the output of the Critic network is the inferred state value for S_{env}. In the locally observable policy network, the Actor and Critic networks both have the input S_{env} defined in (4.18), and the network outputs are the same as for the fully observable policy networks.

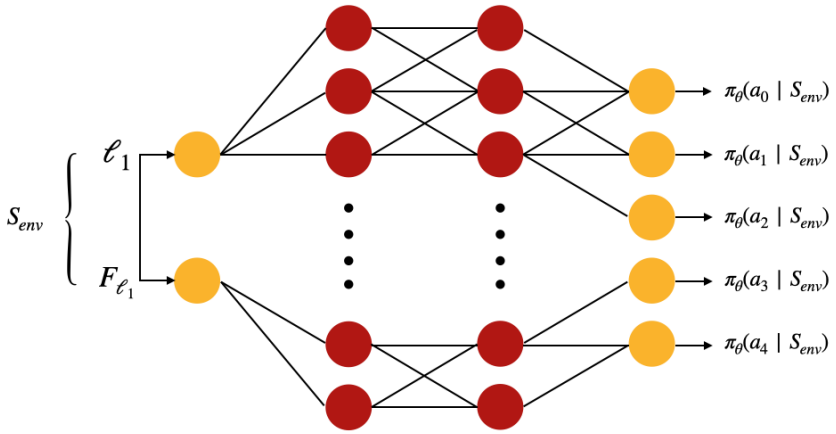


(a) Fully Observable Actor Network

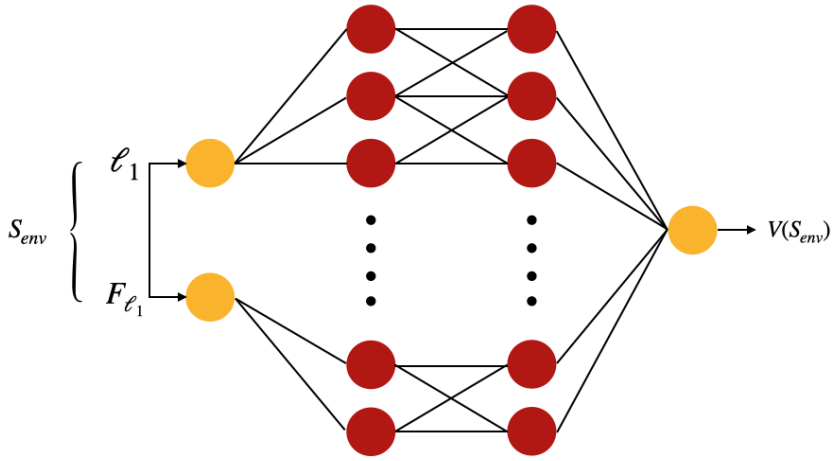


(b) Fully Observable Critic Network

Figure 4.3: Function approximators for the Actor-Critic fully observable control policy, consisting of the Actor neural network model at left and the Critic neural network model at right.



(a) Locally Observable Actor Network



(b) Locally Observable Critic Network

Figure 4.4: Function approximators for the Actor-Critic locally observable control policy, consisting of the Actor neural network model at left and the Critic neural network model at right.

4.4.1 Training

We employ an asynchronous Actor-Critic approach to train the fully observable and locally observable control policies. An asynchronous approach benefits from higher throughput data for updating the gradient by utilizing multiple agents to generate trajectories thus leading to increased traversal of the policy space. Another benefit is reduced training time compared to a single actor, since multiple actors can cover a single policy space more efficiently than one actor. The Actor-Critic control policies are trained asynchronously using a deep RL training framework that we developed, called *multi_robot_trainer* **Zahi M Kakish** (2020a). This flexible framework can parallelize the training of user-defined controllers based on state-space models or learning model-free policies for agents in *OpenAIGym* environments Brockman *et al.* (2016) using a variety of deep RL algorithms such as Deep Q-Learning, Double Deep Q-learning, REINFORCE, and shared and dual neural network Actor-Critic variants. We built the framework on top of TensorFlow Abadi *et al.* (2015) and ROS 2 Thomas *et al.* (2014), using the built-in Data Distribution Service (DDS) communication middleware of ROS 2 to set up a Server-Worker parallelization structure resembling similar asynchronous and synchronous deep RL frameworks defined in Mnih *et al.* (2016). Our platform extends the capabilities of current widely-used asynchronous deep RL training implementations Babaeizadeh *et al.* (2017) by allowing for hybrid GPU/CPU device configurations and multi-device support, although it lacks dynamic scheduling and advanced queuing.

Figure 4.5 provides an overview of the framework's parallelization procedure using a shared global neural network, with a learnable vector of weights θ_G, and local gradient calculations. For an asynchronous setup, a server node distributes the initial neural network weights θ_G to each worker. The workers then generate a trajectory

using the controller or RL environment with the network weights θ_G and compute the gradients (4.14), which are then sent back to the server to compute θ_G at the next step of the optimization procedure. Finally, any worker that computed the gradient gets the updated network weights θ_G from the server for the next iteration. This process differs slightly for synchronous parallelization. The workers still receive the weights from the server, generate trajectories, and calculate gradients to send back to the server; however, the server only sends the updated network weights θ_G to a worker upon receiving the gradients from all of the workers. Once all the gradients have been used to update the network weights, a synchronizer sends the weights to the workers and a new episode begins.

The training episodes are executed with three worker nodes and an asynchronous update server containing the Actor and Critic global network parameters θ. Each worker generates a trajectory over a 2×2 grid graph $\mathcal{G}$ using the mean-field model (4.9). The initial distribution of follower agents over the graph was defined as

$$\hat{S}_{initial} = \begin{bmatrix} 0.1 & 0.4 & 0.4 & 0.1 \end{bmatrix}^T, \tag{4.23}$$

and the target distribution was set to

$$\hat{S}_{target} = \begin{bmatrix} 0.4 & 0.1 & 0.1 & 0.4 \end{bmatrix}^T. \tag{4.24}$$

An episode is considered complete when either the stopping criterion defined in Section 4.3.2 is met, or the time step k of the trajectory exceeds the final time $T = 400$ time steps. Training is complete once the server node has tracked 5000 episodes across the three worker nodes.

The Critic neural network loss function for both the fully observable and locally observable control policies is defined as the Huber Loss Huber (1964), in order to reduce the chance that outlier Critic network inference values will derail the gradient

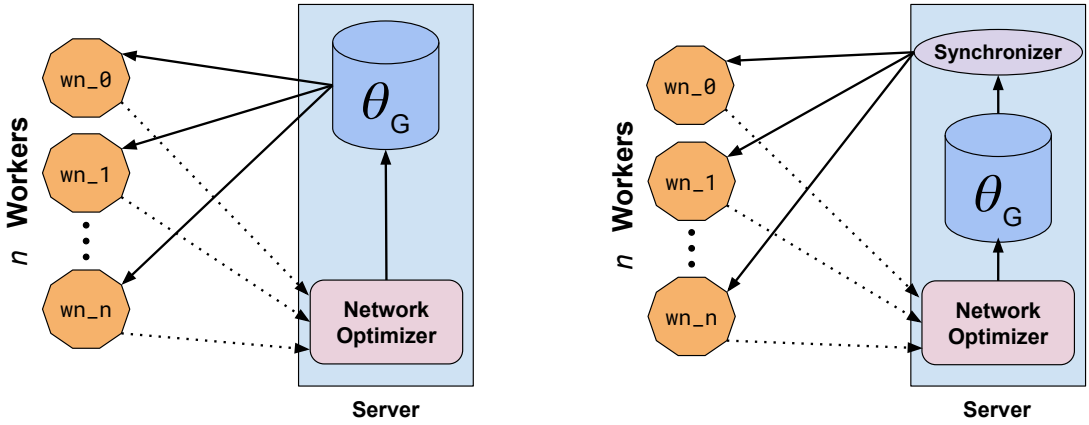


Figure 4.5: Asynchronous (*left*) and synchronous (*right*) gradient worker implementations for deep RL training using the *multi_robot_trainer* **Zahi M Kakish** (2020a). The solid lines indicate the transfer of the weights θ_G of the global network to individual worker nodes. Upon completing the simulation of a trajectory and calculating the gradient with the network weights received from the server, each worker sends the gradient back to the server, as indicated by the dashed lines, for optimization of the global network weights θ_G.

descent towards an optimal policy. This loss function is tuned by a hyperparameter δ, which determines the point at which loss is considered quadratic. We set the jump rate parameter to $\beta = 0.1$, the threshold value to $\mu = 0.0025$, the discount factor to $\gamma = 0.99$, and the loss function hyperparameter to $\delta = 1.0$ in all the training runs.

Both the fully observable and locally observable control policies were trained on a commercial-grade desktop computer with a 4th-generation Intel i7 8-core CPU with 32GB of RAM running Ubuntu 20.04. Each worker node and the server node were run in individual *Docker* containers with their networks configured to allow the transfer of gradients and network weights.

4.5 Simulation Results

In this section, we investigate the effect of network size (Section 4.5.1), learning rate (Section 4.5.2), and reward function (Section 4.5.3) on the performance of fully and locally observable control policies that were designed using the asynchronous

Actor-Critic approach. We also compare these control policies to those designed in our prior work **Zahi M Kakish** *et al.* (2020) using Temporal-Difference learning approaches (Section 4.5.4). The control policies evaluated in Sections 4.5.1, 4.5.2, and 4.5.4 were trained using the MSE (4.20) as the reward function $R(k)$ in both the objective function $\sum_{k=1}^{T} R(k)$ and the stopping criterion $|R(k)| \leq \mu$. In Sections 4.5.1–4.5.4, we validated the control policies in 2D agent-based simulations run in *gym-herding* **Zahi M Kakish** (2019), an open-source *OpenAI Gym* environment that we developed and used in our previous work **Zahi M Kakish** *et al.* (2020) to train, simulate, test, and visualize leader-based control policies for herding swarms of follower agents whose locations evolve on a graph according to the DTMC (4.1). Section 4.6 describes our validation of the control policies in 3D simulations of a real-world testbed environment with physical constraints on robot movement. For the control policies in all sections, the graph $\mathcal{G}$ was defined as the grid graph in Fig. 4.1.

4.5.1 Effect of Network Size

We investigated the effect of the number of hidden layers and layer units in the MLP neural network configuration on its robustness and efficacy. We tested networks with depths of 1 to 4 hidden layers, each with a width of 16, 32, or 64 units. All layers in each tested network had the same width, and each hidden layer used a Rectified Linear Unit (ReLU) activation function Nair and Hinton (2010). We tested the trained policies in agent-based simulations with the graph $\mathcal{G}$ defined as the 2×2 grid graph in Fig. 4.1 and the initial and target follower agent distributions defined in (4.23) and (4.24), respectively. The same training procedure was repeated for each network width and depth.

Figures 4.6 and 4.7 plot the total reward $\sum R(k)$ received by the leader given its trajectory per training episode for the fully observable and locally observable

control policies, respectively. Each subfigure plots the total reward from each of the three asynchronous worker nodes associated with the leader during training. The total reward initially varies considerably as the policy space is explored; as training progresses, the reward tends to increase in most of the trials to an approximately steady-state value with relatively small variation. This variability tends to decrease with increasing width or depth of the hidden layers of the neural network. Certain network sizes result in large erratic spikes in total rewards during training of the fully observable control policy, likely reflecting performance collapse due to unstable Q value estimation in the Actor loss calculations Lipton *et al.* (2018); Anschel *et al.* (2017); Graesser and Keng (2019). As Fig. 4.6 shows, the network of depth 4 and width 64 displayed the most severe performance collapse. Smaller fluctuations in total reward occur during training of the locally observable control policy, as shown in Fig. 4.7, and thus are less likely to derail the policy optimization. Compared to other network depths, a depth of 3 yields a sufficient increase in total reward for control policy optimization without producing erratic variations in reward that lead to performance collapse.

To evaluate their performance, the fully observable and locally observable leader control policies that were trained using each network were implemented on simulations of the mean-field model (4.9). Each control policy was tested on 100 simulations of this model, and its performance was measured in terms of the mean reward accumulated by the leader over all simulations and the mean number of time steps (iterations) during all simulations until the follower agents were herded to the target distribution. For the fully observable controller, the mean reward and mean number of iterations are plotted in Fig. 4.8 for each neural network depth and width. The lower the mean number of iterations, the closer the total reward metric is to 0. The leader control policies that were optimized using networks of depth 4 and widths 16 and

64 performed poorly compared to the control policies optimized using networks of lower depth. These deeper networks results in a more negative total reward and an increased number of iterations In addition, the performance degradation and collapse observable with the deeper networks are reflected in their respective training results shown in Fig. 4.8, where the breakdowns in training are prominent. The network with depth 4 and width 32 resulted in good performance likely due to the training quickly correcting itself upon subsequent episodes, as evident by earlier spikes and corrections. Judging by the breakdowns and corrections in episodes prior to training ended, the performance may have degraded again if the number of training episodes were increased. The network with depth 3 and width 32 produced acceptable mean rewards and number of iterations. During training, this network trended more stable and resulted in less overall variability in our metrics than networks of other sizes. In essence, deeper networks tended to fail after three hidden layers while larger network widths affected performance negatively at shorter network depths.

The locally observable controller tests on the mean-field model show a more consistent trend in average reward and iterations when utilizing different network depths and widths. As can be seen in Fig. 4.9, every network size performed better than its fully observable counterpart in both metrics. The performance improved with increased network depth and width. The severe performance degradation and collapse present in the fully observable policy that was optimizing using a network of depth 4 is greatly diminished in comparison to the same network depth on the locally observable policy. A network of depth 4 and width 32 produces the best results in terms of average reward and iterations; however, the total reward for this network varies erratically (see Fig. 4.7). The control policy for this network configuration may be susceptible to performance degradation if trained for additional episodes. While the network with depth 3 and width 64 yields adequate performance, it too suffers from

erratic behavior during training. Given its mean reward and number of iterations, as well as its low variability in total reward during training, we expect that a network of depth 3 and width 32 would likely optimize the locally observable control policy.

4.5.2 Effect of Learning Rate

In addition to network size, we investigated the effect of the learning rate α on the performance of the optimized control policies. We implemented the optimization procedure with $\alpha \in \{0.0001, 0.00025, 0.000375, 0.0005, 0.000625, 0.00075, 0.001\}$ and tested the performance of the resulting leader control policies on 100 simulations of the mean-field model for each value of α, and on 100 runs of agent-based simulations with each value of α and with $N = 10, 100$, or 1000 follower agents.

Figure 4.10 plots the total reward from each worker during training of the fully observable control policy with each tested value of α. Training with $\alpha = 0.0005, 0.000625$, and 0.00075 yielded the lowest variability in total reward during the later training episodes. Training with $\alpha = 0.001$ produced significant spikes in total rewards, possibly due to overly large changes in θ during the optimization procedure (Eq. (4.10)).

Figure 4.11 plots the total reward during training of the locally observable control policy with each value of α. These plots display similar trends to the plots in Fig. 4.10, except for $\alpha = 0.001$, which has smaller fluctuations in reward than the corresponding plots for the fully observable control policy. The reward plot for $\alpha = 0.000625$ shows the lowest variability during the later training episodes.

Figures 4.12 and 4.13 plot the mean number of iterations until convergence to the target distribution that resulted from both control policies when implemented on the mean-field model and on agent-based simulations, for the tested values of α and for $N = 10, 100$, and 1000 agents. For the fully observable control policy, $\alpha = 0.000625$ yielded the minimum number of iterations (13.9) in the mean-field model, but not

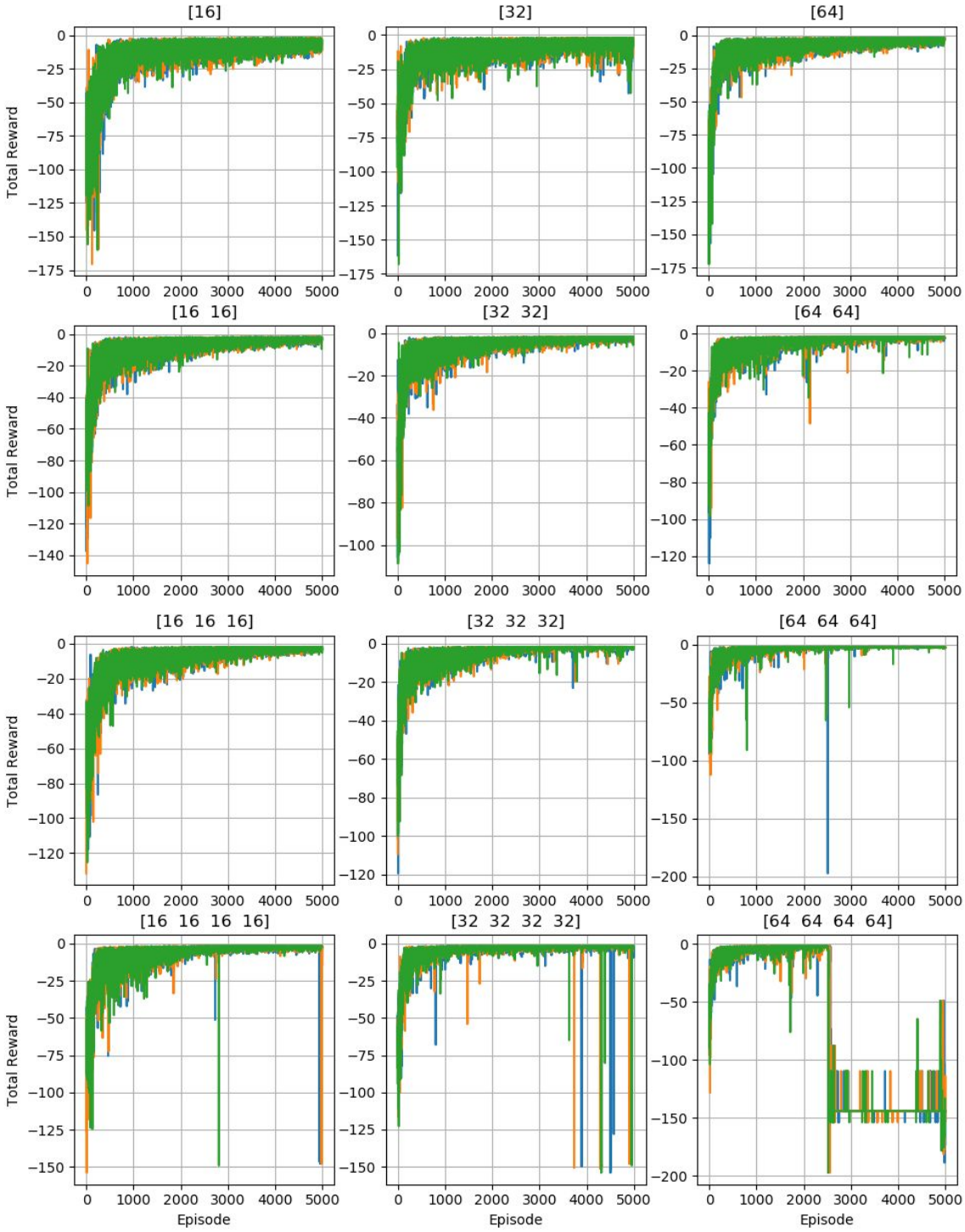


Figure 4.6: Total rewards from the three worker nodes, plotted in green, blue, and orange, during 5000 training episodes of the fully observable control policy for different network sizes. The subfigures in rows 1, 2, 3, and 4 correspond to networks of depth 1, 2, 3, and 4, respectively. The subfigures in columns 1, 2, and 3 correspond to networks of width 16, 32, and 64, respectively.

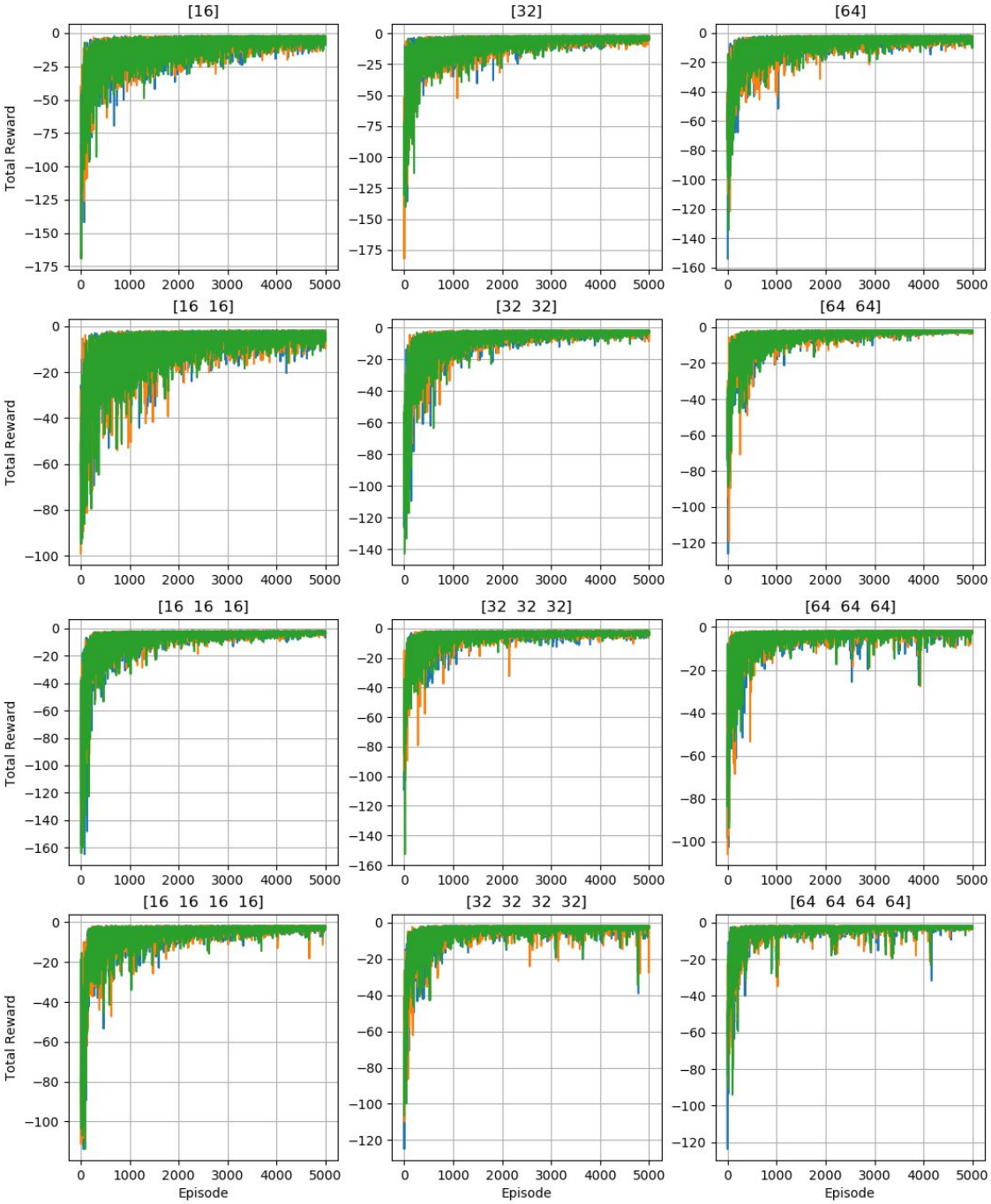


Figure 4.7: Total rewards from the three worker nodes, plotted in green, blue, and orange, during 5000 training episodes of the locally observable control policy for different network sizes. The subfigures in rows 1, 2, 3, and 4 correspond to networks of depth 1, 2, 3, and 4, respectively. The subfigures in columns 1, 2, and 3 correspond to networks of width 16, 32, and 64, respectively.

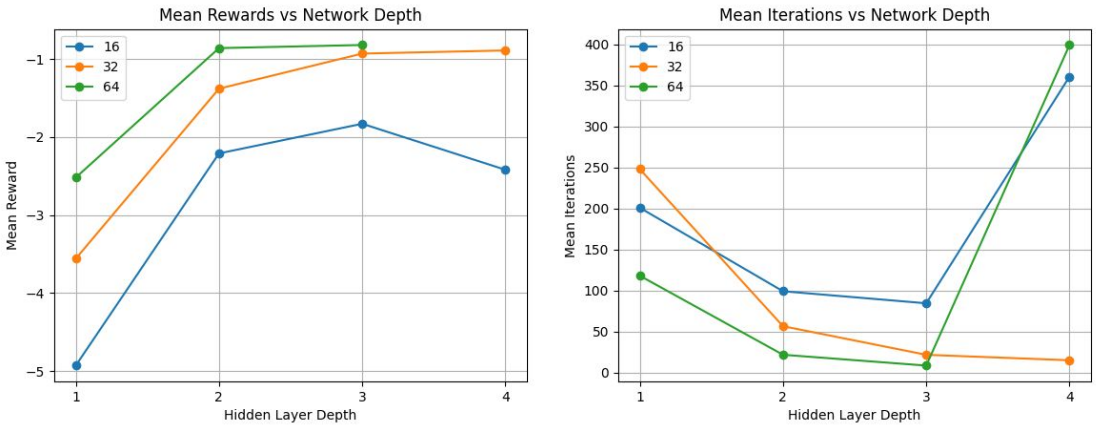


Figure 4.8: Mean reward and mean number of iterations resulting from the fully observable leader control policy trained with networks of different sizes. The mean reward (-91.65) for a network of depth 4 and width 64 is not displayed, since the control policy trained using this network failed to achieve the herding objective during training.

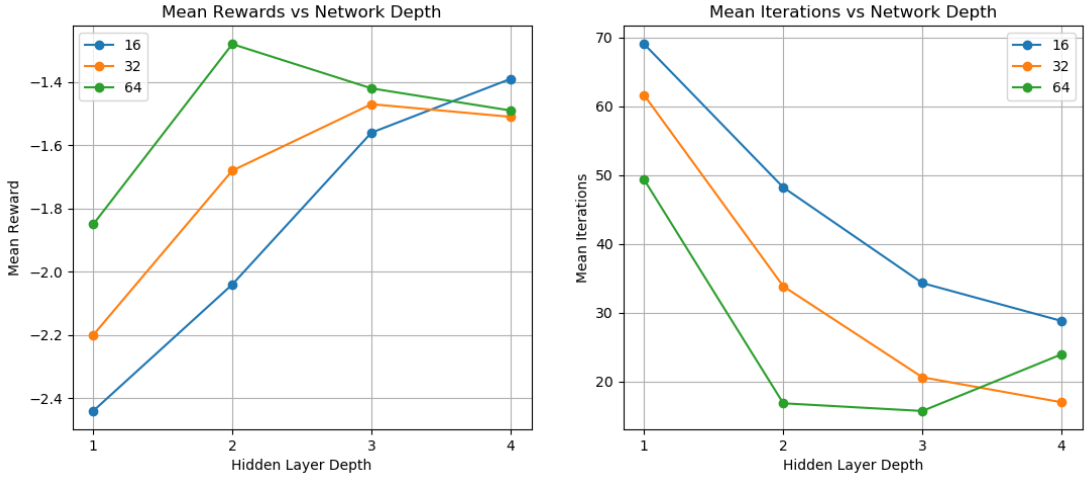


Figure 4.9: Mean reward and mean number of iterations resulting from the locally observable leader control policy trained with networks of different sizes.

in the agent-based simulations with $N = 10$ or 100. Training this control policy with $\alpha = 0.000375$ resulted in the minimum number of iterations in the agent-based simulations with all values of N, with a slightly higher number of iterations (16.5) in the mean-field model than for $\alpha = 0.000625$. For the locally observable control policy, the number of iterations in the mean-field model decreased with increasing α until $\alpha = 0.00075$, with a slight increase at $\alpha = 0.001$. Although $\alpha = 0.000625$ yielded a higher number of iterations in the mean-field model than $\alpha = 0.00075$ and $\alpha = 0.001$, this value of α produced the minimum, or close to the minimum, number of iterations in the agent-based simulations. Therefore, we used $\alpha = 0.000625$ as the learning rate to train control policies for comparison to the control policies that we designed in our previous work **Zahi M Kakish** *et al.* (2020).

4.5.3 Effect of Reward Function

We explored the effect of different reward functions $R(k)$, defined by the MSE (4.20), KL Divergence (4.21), or $\log_{10}$ KL Divergence (4.22), on the performance of the leader control policies. We tested different combinations of these definitions of $R(k)$ for both the objective function $\sum_{k=1}^{T} R(k)$ and the stopping criterion $|R(k)| \leq \mu$. We set $\mu = 0.0025$ for $R(k)$ defined by (4.20) in the stopping criterion. To maintain equivalent stopping criteria for the other two definitions of $R(k)$, we set $\mu = 0.0075$ for $R(k)$ defined by (4.21) and $\mu = 0.0033$ for $R(k)$ defined by (4.22). The control policy training procedure for all three value of μ was the same as for the control policies in the previous two subsections. In all tests, we defined the learning rate as $\alpha = 0.000375$ for the fully observable control policy and $\alpha = 0.000625$ for the locally observable control policy, which we found to produce the best performance (see Section 4.5.2.)

Figure 4.14 plots the mean number of iterations until convergence to the target

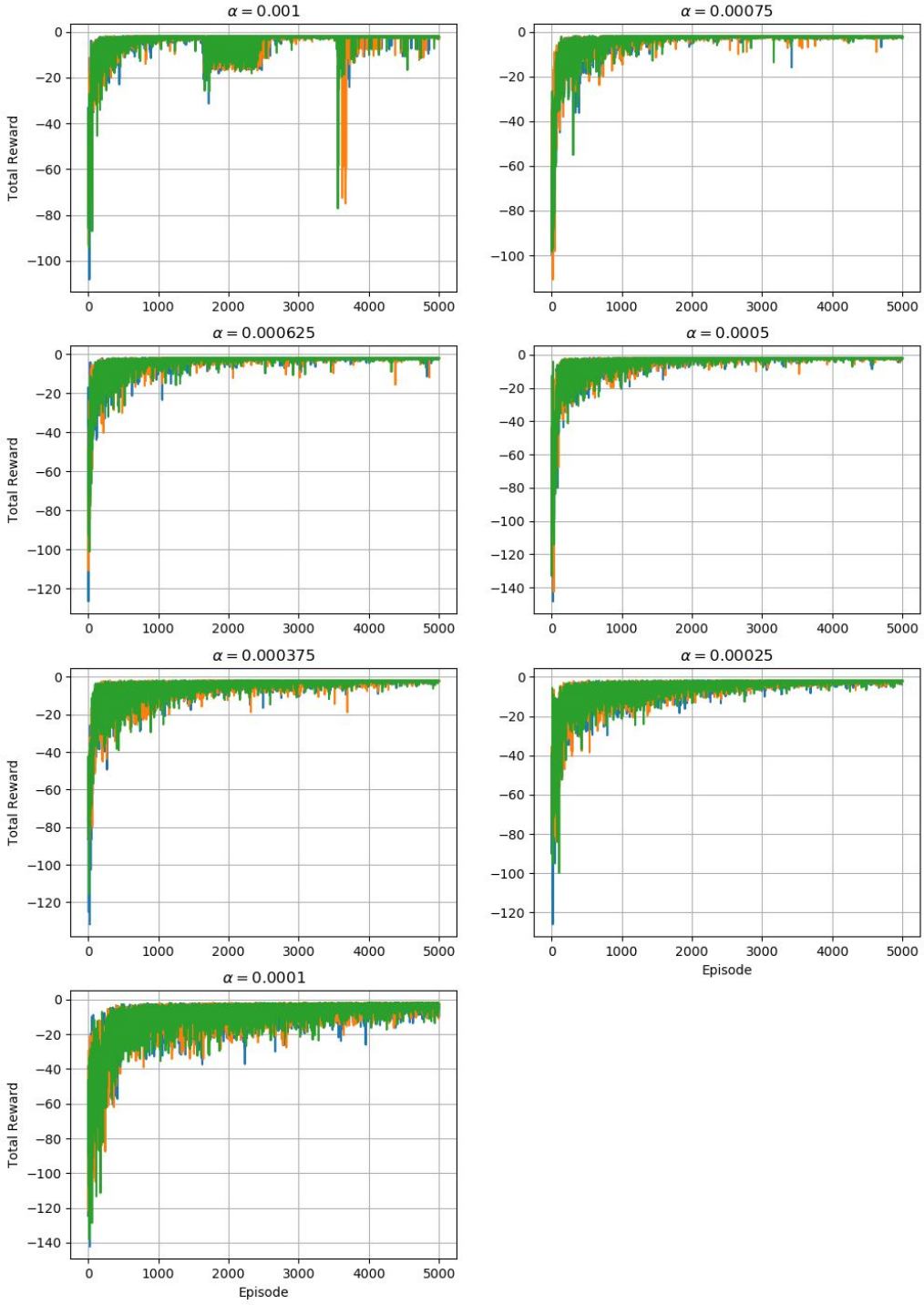


Figure 4.10: Total reward from the three worker nodes, plotted in green, blue, and orange, during 5000 training episodes of the fully observable control policy for different values of the learning rate α.

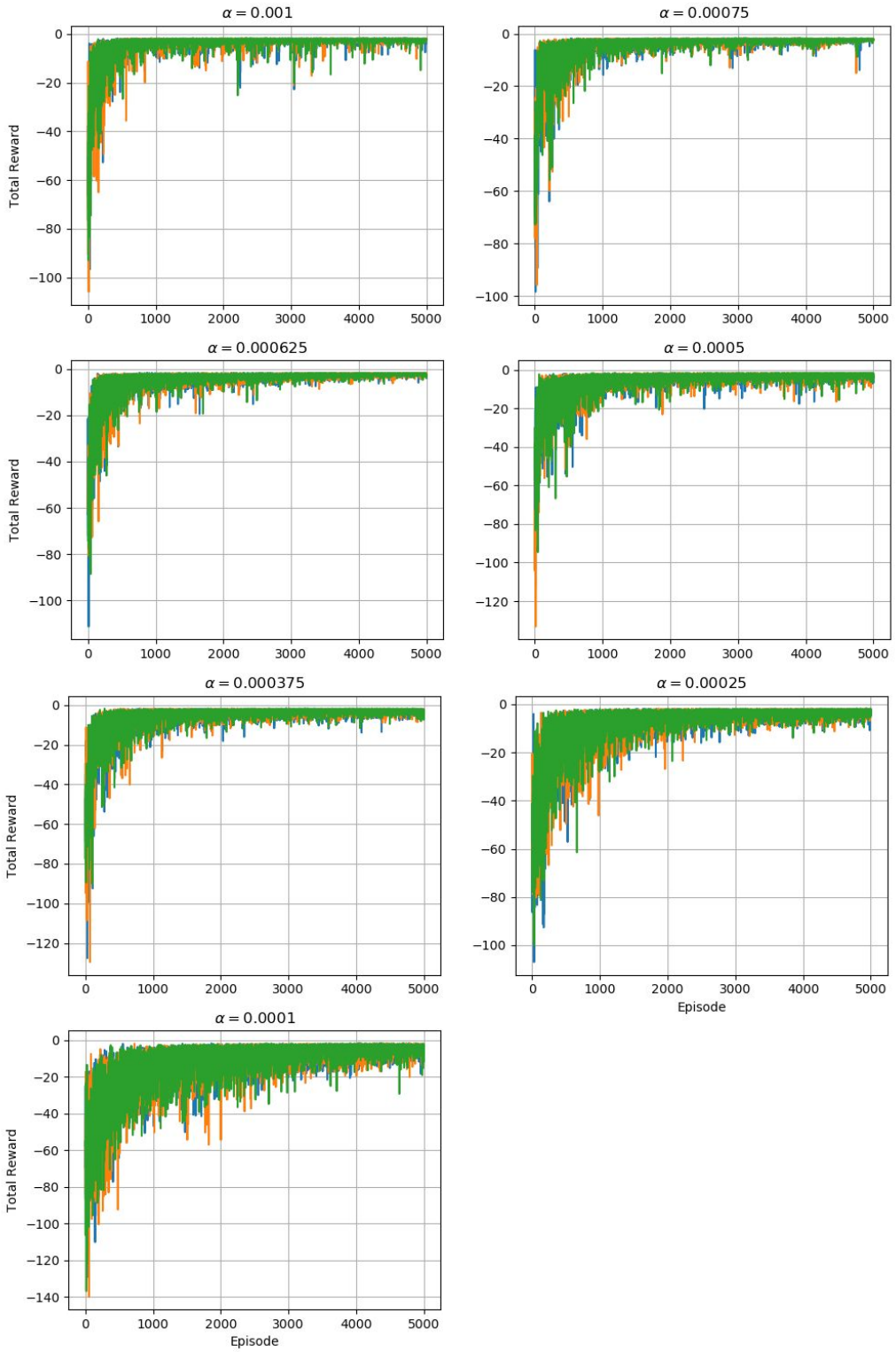


Figure 4.11: Total reward from the three worker nodes, plotted in green, blue, and orange, during 5000 training episodes of the locally observable control policy for different values of the learning rate α.

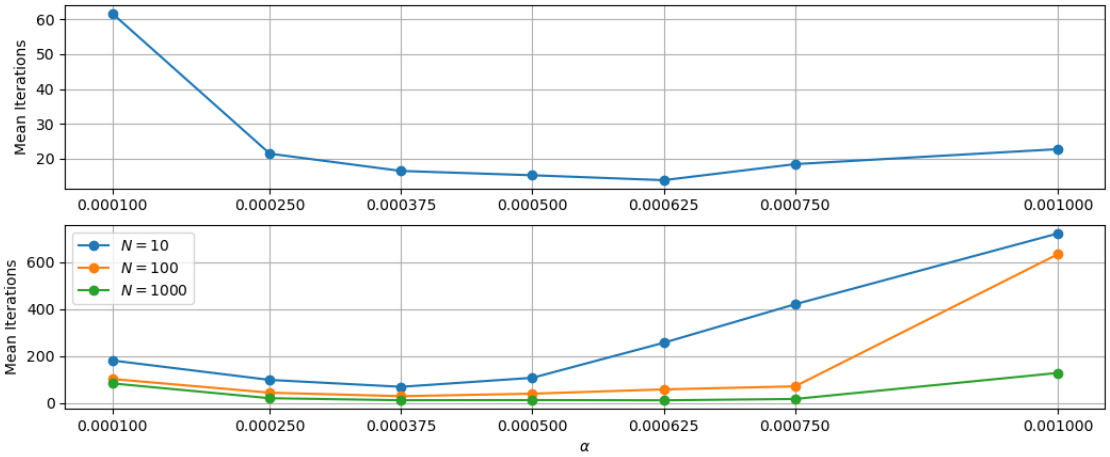


Figure 4.12: Mean number of iterations resulting from the fully observable leader control policy, trained using 7 different values for the learning rate α, when run on the mean-field model (*top*) and on agent-based simulations with $N = 10, 100$, and 1000 follower agents (*bottom*).

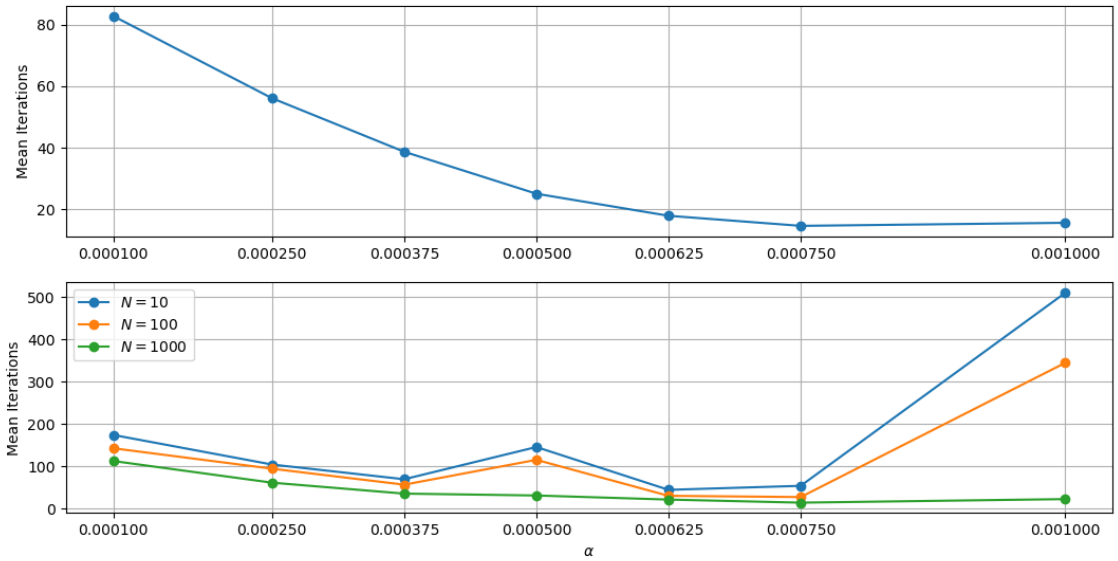


Figure 4.13: Mean number of iterations for the locally observable leader control policy, trained using 7 different values for the learning rate α, when run on the mean-field model (*top*) and on agent-based simulations with $N = 10, 100$, and 1000 follower agents (*bottom*).

distribution for the fully observable control policy with different definitions of $R(k)$ in the objective function and stopping criterion, for 100 runs of agent-based simulations with $N = 10, 100$, or 1000 follower agents. The plots show that the convergence time decreases monotonically with population size N for all cases, and in general, the choice of $R(k)$ in the objective function and stopping criterion produces little to no effect on the convergence time, regardless of N. When $R(k)$ was the same in both the objective function and stopping criterion, defining it as the KL Divergence produced slower convergence for all values of N than defining it as the MSE or $\log_{10}$ KL Divergence. This variability in performance can be attributed to the differences in reward dynamics of the three definitions of $R(k)$, illustrated in Fig. 4.2, with the KL Divergence yielding the lowest rewards.

Figure 4.15 plots the same results as Fig. 4.14 for 100 runs of agent-based simulations of the locally observable control policy with $N = 10, 100$, or 1000, with the same $R(k)$ definitions. Again, convergence time generally decreases monotonically with N. For each stopping criterion, the definition of $R(k)$ in the objective function significantly affected convergence for lower agent populations ($N = 10, 100$) but not for higher populations ($N = 1000$), which all had similar convergence times regardless of stopping criterion. In contrast to the fully observable policy, when $R(k)$ was the same in both the objective function and stopping criterion, defining it as the KL Divergence produced faster convergence for all values of N than defining it as the MSE or $\log_{10}$ KL Divergence. We hypothesize that the larger magnitude of $R(k)$ when defined as the KL divergence rewards given by the KL divergence provided better information on the entropy of the follower agent population distribution. This more accurate estimate of the system's state resulted in considerable performance gains by the locally observable control policy.

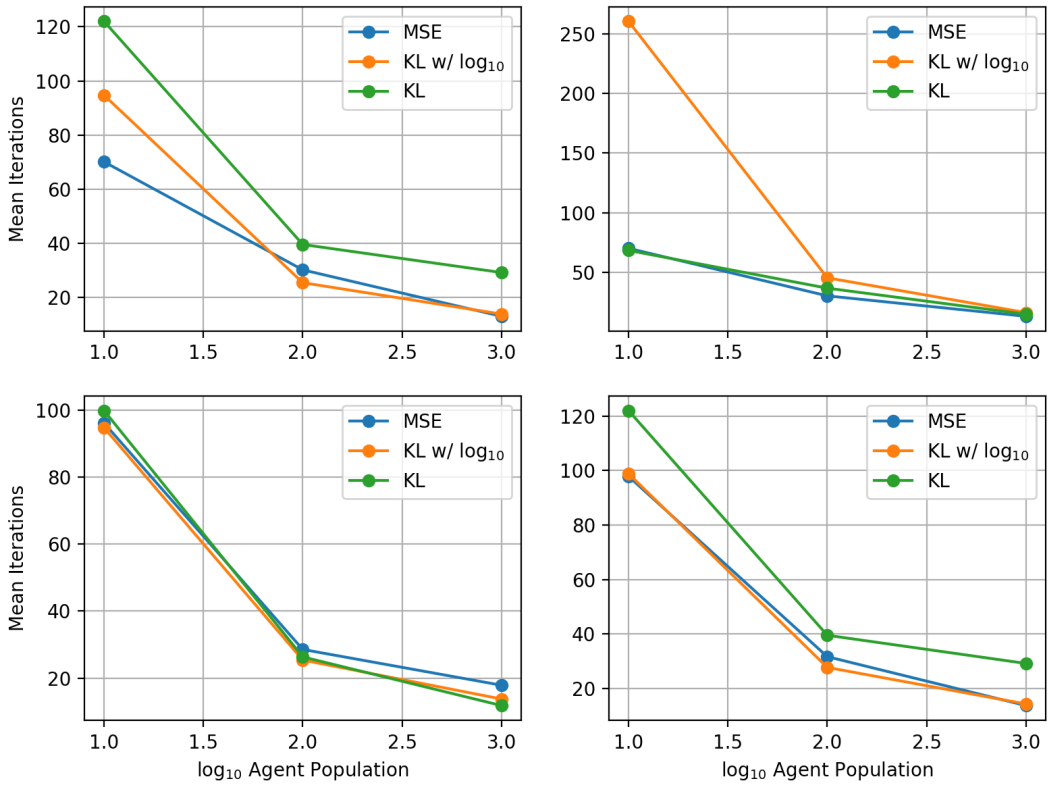


Figure 4.14: Mean number of iterations until convergence versus number of follower agents N when run on agent-based simulations of the fully observable control policy, with $R(k)$ in the objective function defined as the MSE (4.20), KL Divergence (4.21), or $\log_{10}$ KL Divergence (4.22) (indicated by the legend). In the stopping criterion, $R(k)$ is defined as (*top left*) the same function as the objective function; (*top right*) the MSE (4.20); (*bottom left*) the $\log_{10}$ KL Divergence (4.22); or (*bottom right*) the KL Divergence (4.21).

4.5.4 Comparison with Policies Designed using Temporal-Difference Methods

After choosing a suitable network size and learning rate α, we compared the performance of fully and locally observable control policies designed using the Actor-Critic algorithm presented here to the performance of control policies designed in our previous work **Zahi M Kakish** *et al.* (2020) using two tabular Temporal-Difference learning approaches, SARSA and Q-Learning. Both learning algorithms in **Zahi M Kakish** *et al.* (2020) used a discretization of the system state: the population

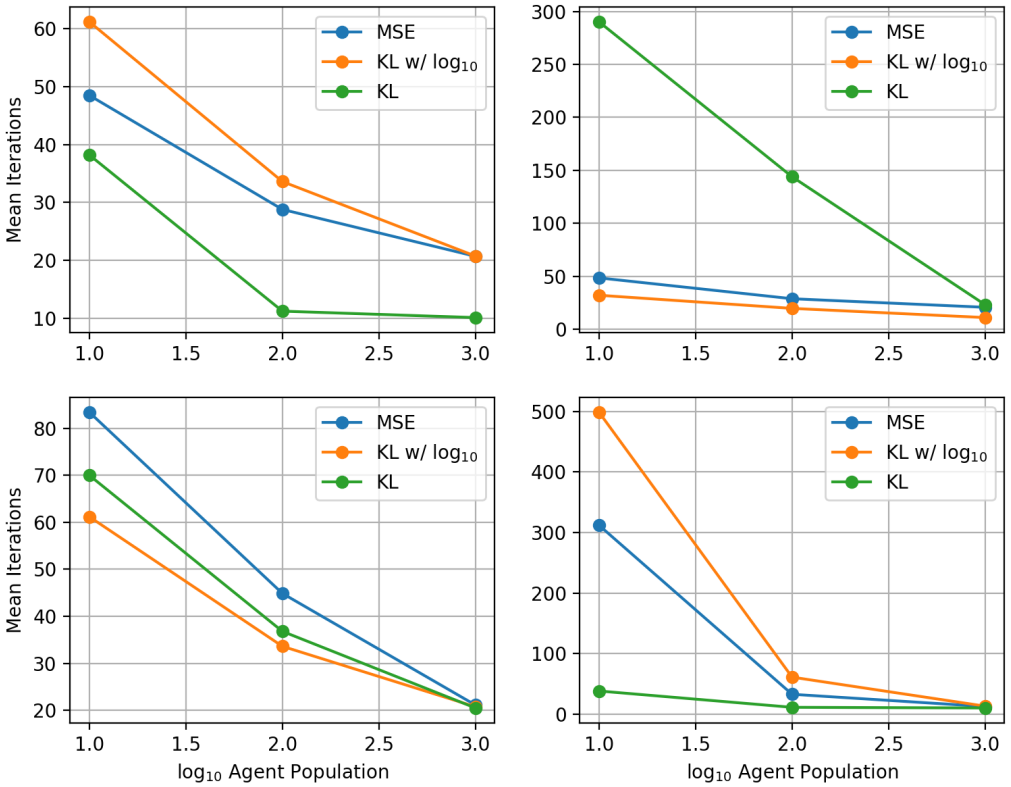


Figure 4.15: Mean number of iterations until convergence versus number of follower agents N when run on agent-based simulations of the locally observable control policy, with $R(k)$ in the objective function defined as the MSE (4.20), KL Divergence (4.21), or $\log_{10}$ KL Divergence (4.22) (indicated by the legend). In the stopping criterion, $R(k)$ is defined as (*top left*) the same function as the objective function; (*top right*) the MSE (4.20); (*bottom left*) the $\log_{10}$ KL Divergence (4.22); or (*bottom right*) the KL Divergence (4.21).

fractions of follower agents at the vertices of $\mathcal{G}$ were divided into D intervals and scaled up to integers between 1 and D. For example, for $D = 10$, a population fraction of 0.33 would be represented as 3. Using a higher value of D yields a finer discretization of agent populations, but increases the size of the system state.

We test the Q-Learning and SARSA control policies **Zahi M Kakish** *et al.* (2020) against our Actor-Critic trained deep RL policy in the agent-based simulation. For each test, 100 runs each were performed with $N = 10$–100 agents, at 10-agent increments, and with $N = 1000$ agents. As N increases, the fluctuations around the

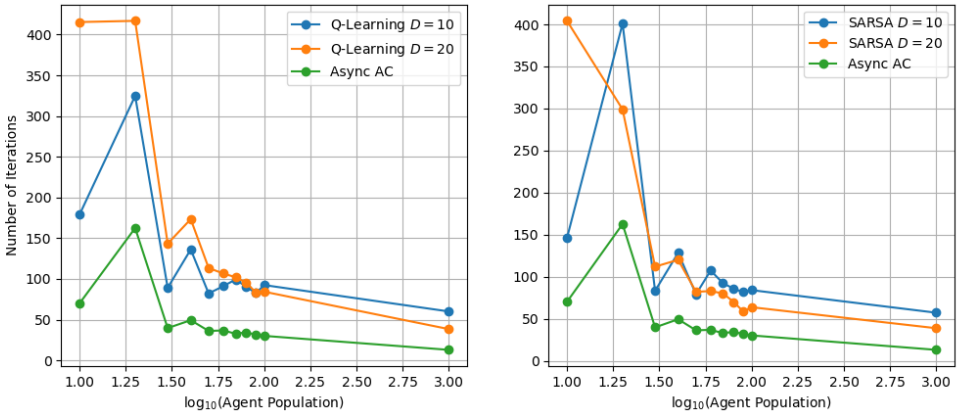


Figure 4.16: Comparison of our Q-Learning (*left*) and SARSA (*right*) algorithms trained on 1000 agents following a DTMC from **Zahi M Kakish** *et al.* (2020) at both population fraction interval values $D = 10$ and 20 to our asynchronous actor-critic trained fully-observable control policy. Each test is run on a $N = 10 - 100$ and 1000 agent populations.

mean-field model solution decrease, so the time evolution of populations in each state can be described more accurately by this model. Therefore, the controller tests for our Actor-Critic approaches are only analogous to the case of $N \geq 1000$ utilizing our earlier Q-learning or SARSA leader controllers since they exhibit a *mean-field effect* for larger agent populations on a DTMC. For smaller N, the Actor-Critic controller was tested on agent-based simulations to adequately compare its performance to that of our earlier controllers.

Figures 4.16 and 4.17 compare the mean number of iterations until convergence to the target distribution versus agent population N for the fully and locally observable control policies designed in this work and those from **Zahi M Kakish** *et al.* (2020), trained with $D = 10$ and $D = 20$. The fully and locally observable control policies outperform the control policies from **Zahi M Kakish** *et al.* (2020) for all values of N and both values of D. The locally observable control policy outperforms the fully observable policy by a small margin at low populations N.

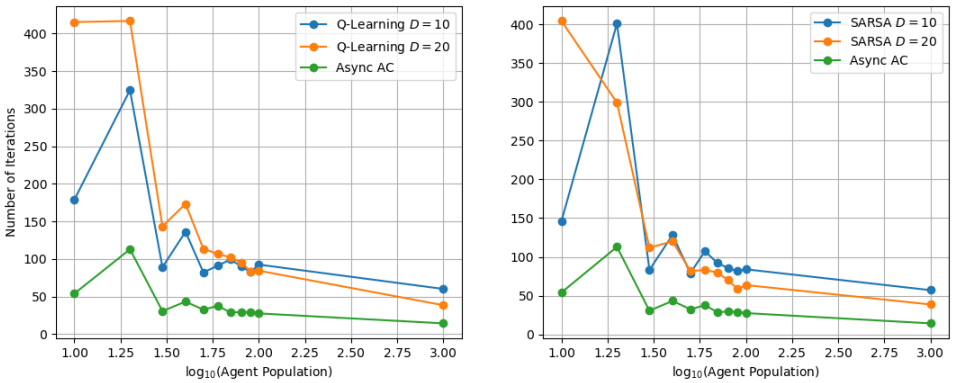


Figure 4.17: Comparison of our Q-Learning (*left*) and SARSA (*right*) algorithms trained on 1000 agents following a DTMC from **Zahi M Kakish** *et al.* (2020) at both population fraction interval values $D = 10$ and 20 to our asynchronous actor-critic trained locally-observable control policy. Each test is run on a $N = 10 - 100$ and 1000 agent populations.

4.6 Validation of Control Policies in a 3D Simulation of a Real-World Testbed

We conducted additional simulations using a program that we developed in Gazebo Aguero *et al.* (2015) and ROS 2 called *ros2_robotarium* **Zahi M Kakish** (2020b), a 3D simulation of the Georgia Tech Robotarium Wilson *et al.* (2020). These simulations were developed to verify that the control policies are effective in a real-world environment with physical constraints on robot dynamics and inter-robot spacing. The simulated testbed was designed to be similar to the setup of the physical Robotarium testbed that we used in our previous work **Zahi M Kakish** *et al.* (2020) (a video recording of the experiments is shown in **Zahi M Kakish** *et al.* (2020)). The leader agent in these simulations was virtual, and the follower agents were modeled after the Robotarium *GritsBotX* robots. The robots avoid obstacles and one another using barrier certificates Wang *et al.* (2017), a modification to the robots' controllers that satisfies particular safety constraints. The environment is represented as a 2×2 grid, in which the grid cells correspond to the vertices of the graph in Fig. 4.1. The robots can move between pairs of grid cells that are connected by an edge

in this graph. Figure 4.18 shows overhead and isometric views of the initial robot configuration in the simulation.

Experiments run on the physical Robotarium have a 6–7 minute time limit, after which the robots must be recharged. This constraint limited the number of robots we could use in our experiments in **Zahi M Kakish** *et al.* (2020), since larger robot populations took longer to reach a target distribution due to the increased time required for robots to avoid collisions with other robots. The simulated testbed did not have this constraint, which allowed us to increase the robot population to 20.

Using the optimal learning rate, network size, and reward function supported in the previous sections, we trained the fully observable and locally observable control policies to herd $N = 20$ simulated robots from $\hat{S}_{initial}$ and $\hat{S}_{target}$, defined in (4.23) and (4.24). In the simulation, we set the initial robot distribution to

$$\hat{S}_{initial} = \begin{bmatrix} 0.25 & 0.25 & 0.25 & 0.25 \end{bmatrix}^T,$$

different from (4.23), in order to evaluate the robustness of control policies to changes in $\hat{S}_{initial}$ from the initial distribution on which they were trained. The leader agent was able to herd the 20 robots to the target distribution in 13 iterations using the fully observable control policy, and in 12 iterations using the locally observable policy. Both of these convergence times are well below the mean number of iterations until convergence for $N = 20$ robots in the 2D agent-based simulations of the control policies (see Figs. 4.16 and 4.17, *Asynch AC* plots).

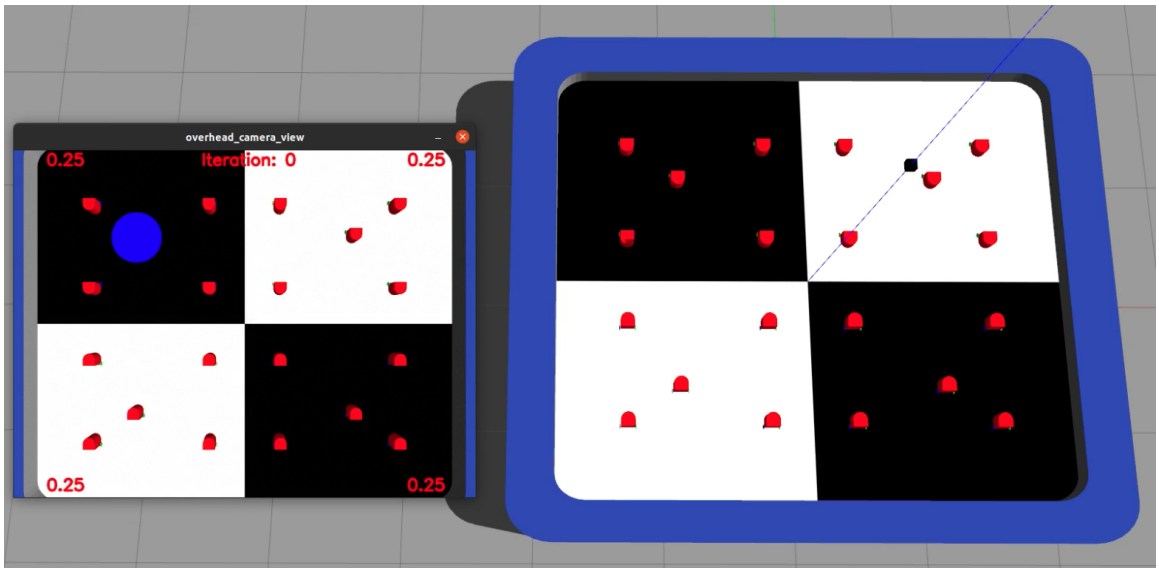


Figure 4.18: A 3D *Gazebo* simulation of a leader-based herding experiment using the *ros2_robotarium* package developed by our laboratory. Spatial states are represented as either black or white squares on the testbed and are labeled similar to the vertex labels in Figure 4.1. Follower agents modeled after the Robotarium *GritsBotX* are in the $\hat{S}_{initial}$ configuration. An overhead camera above the simulated Robotarium testbed publishes video (window on the *left*) of the current testbed along with labels of the current iteration, individual vertex agent population fractions, and a virtual leader.

Chapter 5

IMPROVING THE SCALABILITY OF LEADER-BASED STRATEGIES FROM DEEP RL ALGORITHMS TRAINED ON MEAN-FIELD MODELS

5.1 Summary

In this chapter, we analyze and discuss the effects of scaling the graph size used by the mean-field model to train the leader-based deep RL control policy. Scaling the graph leads to difficulty converging on certain solutions within an adequate number of iterations or under a stopping criteria μ (defined in Section 4.3.2), which reflects the distance the agent population over the graph is from the target population distribution. We investigate the effects that a state and action set have on the fully and locally observable control policies at larger graph sizes and present a new control policy that has improved scaling properties with the graph size.

5.2 Scalability with the Number of Follower Agent States

Our previous work in Chapter 3 represented the state-action value function Q in tabular form as a multi-dimensional matrix. Tabular methods for training limited the scalability of our control policies with the the number of follower agent states, i.e. the order $|\mathcal{V}|$ of the graph $\mathcal{G}$, due to the intractability and impracticality of searching a large matrix or database for an optimal control policy. Additionally, increasing the order of $\mathcal{G}$ required larger values of D, the number of intervals used to discretize the follower agent population fractions, since distributing a swarm of agents among more vertices can reduce the population fractions at each vertex. Using neural network function approximators alleviates these issues by allowing us to scale the number of

parameters of the neural network to fit the number of vertices in the graph.

We tested the robustness of our fully observable and locally observable control policies on 3×3 and 4×4 grid graphs. We trained the control policies using the same method as the previous chapter with changes to the hyperparameters, number of episodes, and final time T. The reward function $R(k)$ in both the objective function and stopping criterion was defined as the MSE (4.20), with the stopping criteria threshold set to $\mu = 0.0025$. Rewriting the vector $\hat{S}$ as a matrix, in which each element is the agent population fraction at the corresponding vertex on the grid graph, the initial and target distributions for the 3×3 and 4×4 graphs were defined as:

$$\hat{S}_{initial} = \begin{bmatrix} 0.92 & 0.01 & 0.01 \\ 0.01 & 0.01 & 0.01 \\ 0.01 & 0.01 & 0.01 \end{bmatrix}, \quad \hat{S}_{initial} = \begin{bmatrix} 0.85 & 0.01 & 0.01 & 0.01 \\ 0.01 & 0.01 & 0.01 & 0.01 \\ 0.01 & 0.01 & 0.01 & 0.01 \\ 0.01 & 0.01 & 0.01 & 0.01 \end{bmatrix},$$

$$\hat{S}_{target} = \begin{bmatrix} 0.15 & 0.05 & 0.15 \\ 0.05 & 0.2 & 0.05 \\ 0.15 & 0.05 & 0.15 \end{bmatrix}, \quad \hat{S}_{target} = \begin{bmatrix} 0.01 & 0.01 & 0.01 & 0.01 \\ 0.01 & 0.22 & 0.22 & 0.01 \\ 0.01 & 0.22 & 0.22 & 0.01 \\ 0.01 & 0.01 & 0.01 & 0.01 \end{bmatrix}.$$

These initial follower agent distributions differ from those for the 2×2 graph in that most of the agents start at a single vertex on the graph. Three worker nodes were used, with 5000 training episodes. In each episode, the leader agent had 4000 iterations to herd the swarm within the prescribed μ threshold. In Chapter 3, we did not test the control policies designed using Temporal-Difference Methods in agent-based simulations. Therefore, we only tested our control policies on the mean-field model.

The initial training of the fully observable control policy, with $\alpha = 0.00037$ and a network depth of 3 and width 32, failed to yield convergence within the stopping criterion threshold $\mu = 0.0025$ for either the 3×3 or 4×4 graph. Increasing the width of the network from 32 to 256 produced convergence within a threshold of about 0.05 for the 3×3, as shown in Fig. 5.1. Training with multiple learning rates values used in previous sections resulted in the same or worse convergence performance. Similar results occurred during the training runs for the 4×4 grid graph, for which the mean μ was approximately 0.075, as shown in Fig. 5.2. Unlike the fully observable control policy, the locally observable control policy did not require increasing the network width to reach a steady-state μ value. The 3×3 grid graph stabilized around $\mu = 0.03$ during training, while the 4×4 grid graph converged to $\mu = 0.08$ before performance collapse toward the end of training. In both control policies, approximately 500 to 1000 of the 4000 iterations per episode resulted in invalid actions, i.e., the leader's attempt to transition to a vertex resulted in an out-of-bounds condition.

These failures can be attributed to the Actor-Critic algorithm's relatively limited exploration of the policy space, whose dimensionality increases significantly with the number of vertices. Moreover, a 2×2 grid graph was small enough for the control policy to quickly learn effective ways to herd the agents; larger grid graphs have more edges, and thus more possible agent transitions between pairs of vertices. As we investigate in the next section, the leader's choice of actions from the action set A determines the scalability of the policy with the number of vertices.

5.3 Redefining the action set on a fully-connected graph

Of the five possible actions for the leader agent, only the *Stay* action results in an increase or decrease in total reward. In Section 4.3.1, we discuss how the reward function generates returns resembling sparse reward characteristics, since only

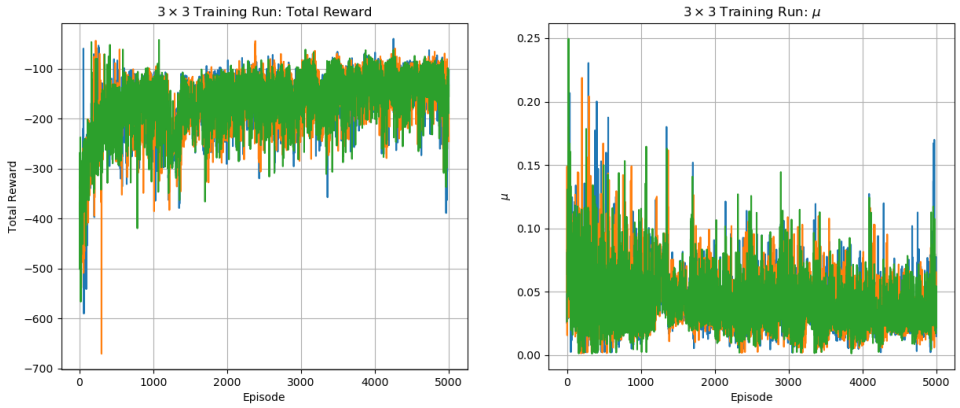


Figure 5.1: Total reward (*left*) and threshold value μ (*right*) per episode during training of the fully observable control policy on a 3×3 grid graph.

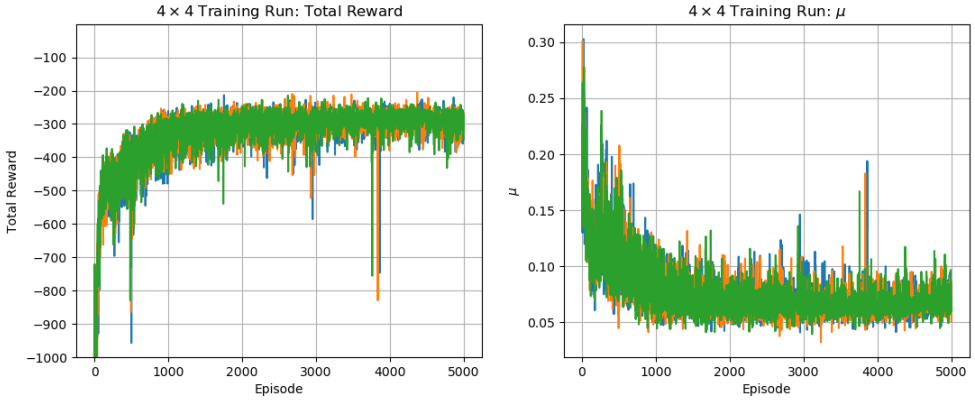


Figure 5.2: Total reward (*left*) and threshold value μ (*right*) per episode during training of the fully observable control policy on a 4×4 grid graph.

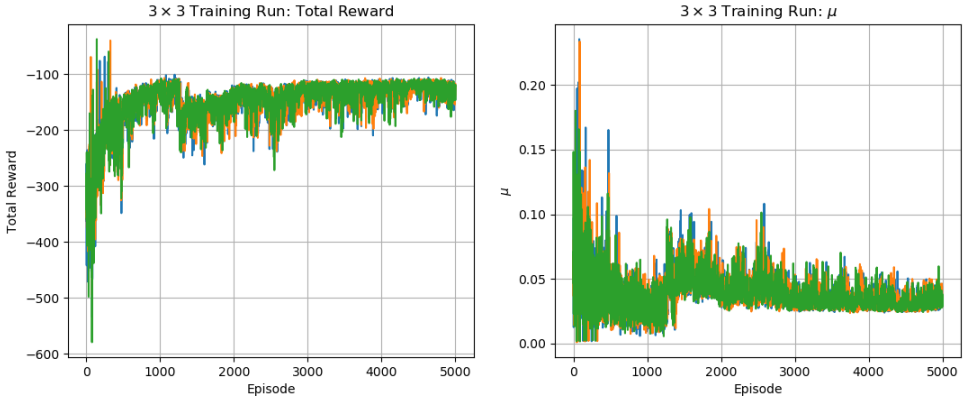


Figure 5.3: Total reward (*left*) and threshold value μ (*right*) per episode during training of the locally observable control policy on a 3×3 grid graph.

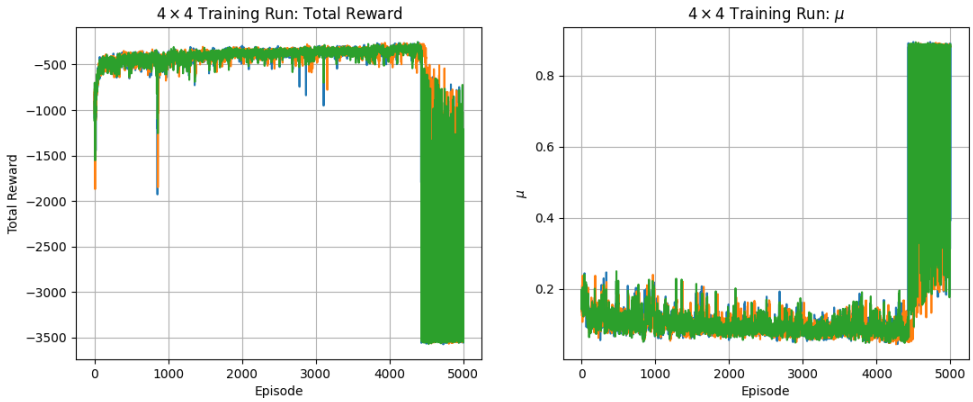


Figure 5.4: Total reward (*left*) and threshold value μ (*right*) per episode during training of the locally observable control policy on a 4×4 grid graph.

the repulsion of follower agents results in any change in reward. Thus, large policy gradient magnitudes computed during the optimization procedure can? frequently be attributed to the *Stay* action. In addition, the action space can result in invalid leader movements, in which the leader attempts to move out-of-bounds on the grid graph. For instance, a leader agent at $\ell_1 = 3$ in Fig. 4.1 can only move *Up*, move *Left*, or *Stay*. If the leader decides to move *Down* or *Right*, it remains at the same vertex since these are invalid actions, yet is given the same reward as a valid action. For example, a DQN algorithm would result in the state-action value function repeatedly learning that the best action a leader can take is to stay at its current vertex and repel follower agents, since that is the only valid action for all vertices. As mentioned in the previous section, many of the actions taken by both the fully and locally observable control polices were invalid.

We modify the action set from the original set (4.19) to include only valid actions. Defining ℓ_1^i as the leader position on the ith vertex, the new action set is:

$$A_{fc} = [\ \ell_1^1,\ \ell_1^2,\ \ell_1^3,\ \ldots\ ,\ \ell_1^M\ ]. \tag{5.1}$$

This requires the leader to be able to transition from any vertex to any other vertex; i.e., the edge set of the graph must be expanded to make the graph fully-connected,

as shown in Fig. 5.5. While the leader follows a fully-connected graph, the individual agents remain evolving according to the strongly connected graph. Motivation for this redefinition of the action set is illustrated by a scenario in our previous work Elamvazhuthi *et al.* (2020), in which we developed a 3D Gazebo simulation of a quadrotor that "herds" a group of *Pheeno* robots Wilson *et al.* (2016) over a 2×2 grid graph whose vertices correspond to spatial regions as in Fig. 4.18. The limited field-of-view of the quadrotor's onboard downward-facing camera required the quadrotor to increase its altitude in order to count the number of robots in each region, which it used to determine its next location on the graph. Although the quadrotor was constrained to move along the edges of the grid graph, in practice it needed to leave its current location towards the center of the graph and increase altitude to obtain the population counts. Hence, implementing our control policies on a real robotic system may require relaxing constraints on the leader's motion so that it can obtain the sensor information necessary to determine its next action. Changing the edge set of the graph that defines the leader's possible movements allows us to remove the leader's current position from the system state $\hat{S}_{env}$ for the fully observable control policy since the policy output is no longer dependent on the leader's position. Thus, the redefined state becomes

$$\hat{S}_{env}(k) = [\ F_1(k),\ F_2(k), \ldots,\ F_M(k)\ ].$$

Our modified function approximators for the Actor-Critic fully observable control policy are shown in Fig. 5.6.

Modifying the locally observable control policy based on the new action set A_{fc} only required changing the neural network to output the same control output as the fully observable control policy. However, in our tests training the locally observable control policy, adjusting the learning rate, network size (depth and width), and reward

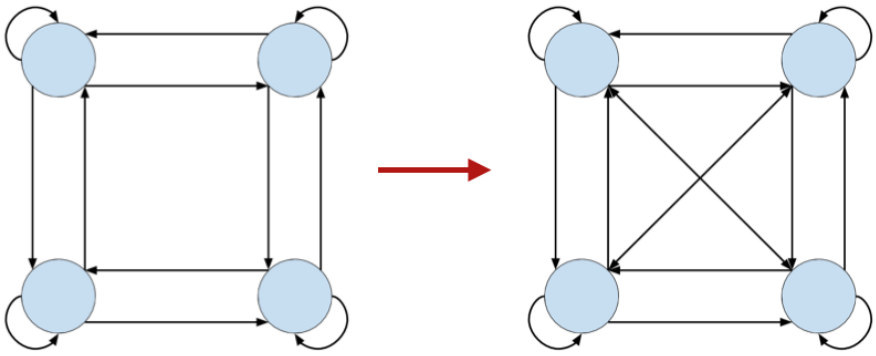
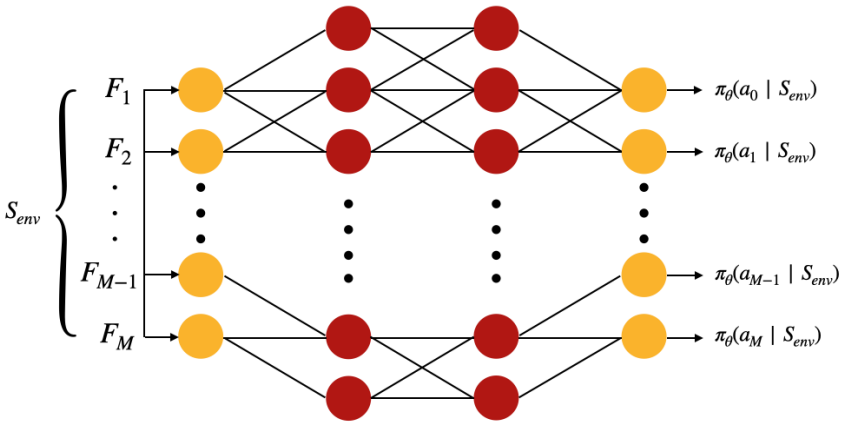


Figure 5.5: Our original approach defined the possible agent transitions according to a strongly-connected grid graph (*left*); however, limitations in the scalability of this approach with the number of vertices prompted us to redefine possible agent transitions according to a fully-connected graph (*right*).

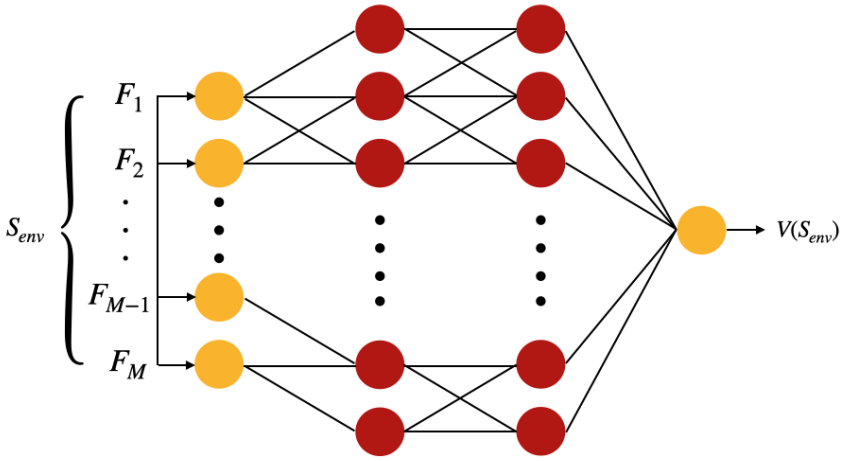
function definition did not produce convergence to the target distribution on the fully-connected 3×3 and 4×4 graphs within the specified threshold. This is likely due to the increased dimensionality of the policy space in the system, since the number of outputs increases with the number of vertices. Therefore, we did not pursue further use of the locally observable control policy with the modified action set.

5.4 Training the fully observable control policy with the new network

Using the new network described in the previous section with $\mathcal{G}$ defined as a 3×3 or 4×4 fully-connected graph, we trained the fully observable control policy with the new action set A_{fc} over 5000 episodes, with 4000 iterations per episode. The network was defined to have depth 3 and width 256. We reduced the learning rate to $\alpha = 0.0001$ and continued using the Huber loss for our gradient descent. The discount factor γ was again set to 0.99. As shown in Figs. 5.7 and 5.8, the resulting fully observable control policy was able to reach the threshold $\mu = 0.0025$, which our previous fully observable and locally observable policies could not attain or sustain for extended periods.



(a) New Fully Observable Actor Network



(b) New Fully Observable Critic Network

Figure 5.6: Function approximators for the modified Actor-Critic fully observable control policy, consisting of the Actor neural network model at the top and the Critic neural network model below, with new state and action spaces for improved scalability with the number of vertices in the graph.

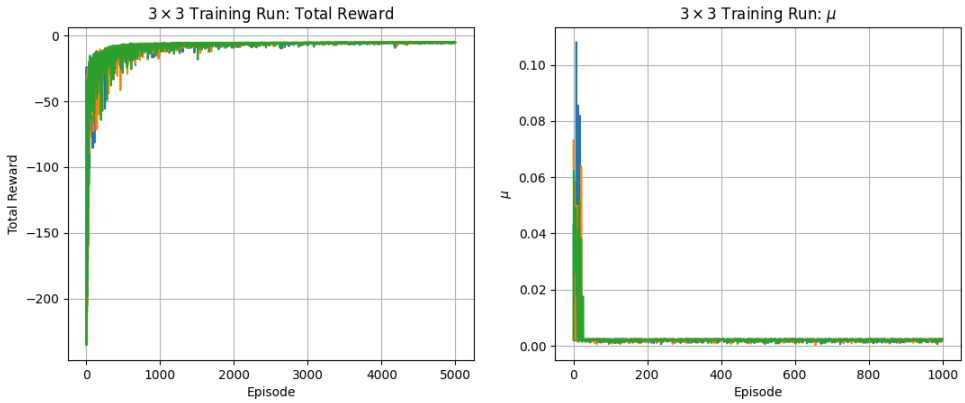


Figure 5.7: Total reward (*left*) and threshold value μ (*right*) per episode during training of the modified fully observable control policy on a 3×3 grid graph. The μ plot is shown over only 1000 episodes, due to the rapid convergence and stability of the new method.

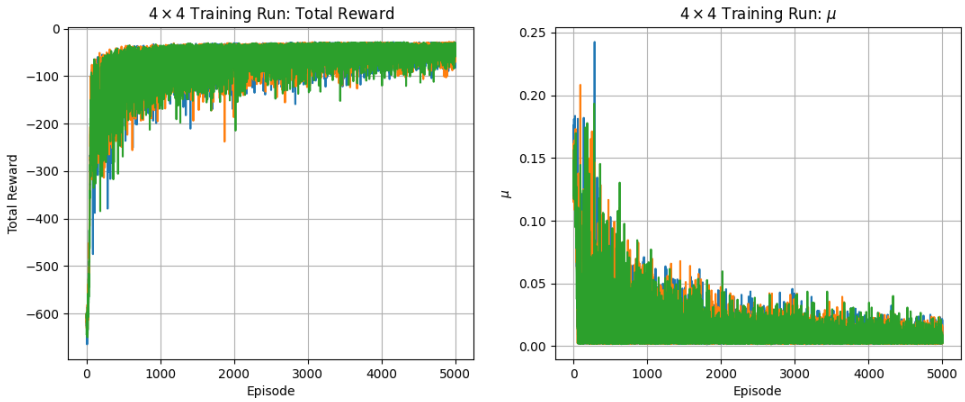


Figure 5.8: Total reward (*left*) and threshold value μ (*right*) per episode during training of the modified fully observable control policy on a 4×4 grid graph.

Chapter 6

LEADERLESS SWARM CONTROL STRATEGIES DESIGNED USING DEEP RL ALGORITHMS TRAINED ON MEAN-FIELD MODELS

6.1 Summary

In this chapter, we extend the work done in (**Zahi M Kakish** *et al.*, 2020, 2021) by exploring a leaderless approach to controller design based on the mean-field model, in which a set of control policies that depend on the target robot distribution is trained and then implemented on all the robots. In particular, we consider the problem of stabilizing a swarm of robots evolving on a Discrete-Time Markov Chain (DTMC) on a graph *without* the use of a leader, as done in our previous work (Deshmukh *et al.*, 2018). We extend the leader-based approach by designing a multi actor, global critic controller architecture for training. We first begin with a review of work done in Deshmukh *et al.* (2018) and then formulate the new problem using our deep RL controller trained on a modification of the mean-field model. We validate the new control policy using a range of simulations with various bidirected graph sizes and agent populations.

6.2 Mean-Field Stabilization of Markov Chain Models for Robotic Swarms

This work presents two computational approaches for synthesizing density feedback laws to stabilize the population densities of a robotic swarm to a strictly positive target equilibrium distribution from an initial distribution. The approach is based on a continuous-time version of the problem statement that is elaborated on in Section 3.2. The evolution of probability distributions is determined by the *Kolmogorov for-*

ward equation, which can be cast in an explicitly control-theoretic form as a bilinear control system,

$$\dot{\mathbf{x}}(t) = \sum_{e \in \mathcal{E}} u_e(t) \mathbf{B}_e \mathbf{x}(t), \quad \mathbf{x}(0) = \mathbf{x}^0 \in \mathcal{P}(\mathcal{V}), \tag{6.1}$$

where $\mathbf{B}_e$, $e \in \mathcal{E}$, are control matrices with entries

$$B_e^{ij} = \begin{cases} -1 & \text{if } i = j = S(e), \\ 1 & \text{if } i = T(e),\ j = S(e), \\ 0 & \text{otherwise.} \end{cases} \tag{6.2}$$

Equation (6.1) is a continuous-time version of (3.6) defined in Section 3.2. With the problem defined, a linear (solved through Linear Matrix Inequality methods) and a nonlinear (solved using a MATLAB Sum-of-Squares Toolbox) controller, $u_e(t)$, were designed to stabilize a positive probability distribution on $\mathcal{V}$ (Deshmukh *et al.*, 2018). The contribution of the author was to implement the computed density feedback laws on a testbed of physical robots. The following sections are an overview of the experimental testbed, procedure, and results.

6.2.1 Experimental Testbed

We evaluated our linear and nonlinear controllers in a scenario in which a group of small differential-drive robots must reallocate themselves among four regions. For these experiments, we used ten Pheeno mobile robots (Wilson *et al.*, 2016), each equipped with a Raspberry Pi 3 computer, a Teensy 3.1 microcontroller board, a Raspberry Pi camera, six IR sensors around its perimeter, and a bottom-facing IR sensor. Pheeno is compatible with ROS, the Robot Operating System, which facilities the implementation of advanced algorithms with our multi-agent system.

For the experiments, we used a 2 m $\times$ 2 m confined arena, shown in Fig. 6.1, that was divided into four regions of equal size. These regions were labeled 1, 2, 3, and 4

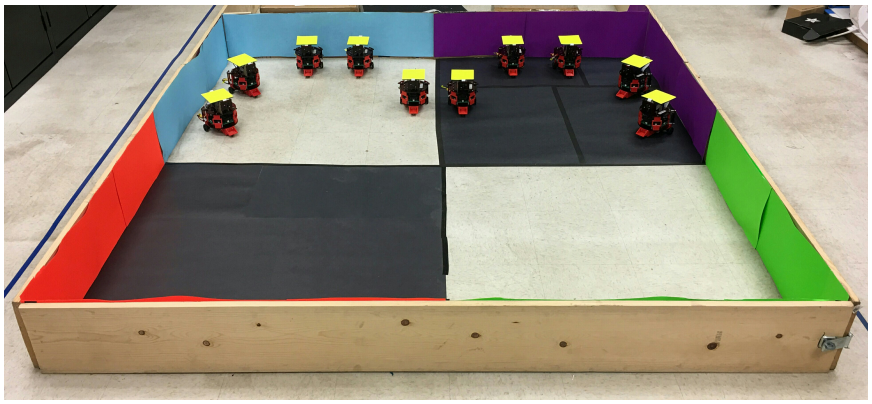


Figure 6.1: Multi-robot experimental testbed.

according to the green numbers in Fig. 6.2. Each region corresponds to a vertex of a bidirected graph that defines the robots' possible transitions between the regions. Robots can move between adjacent regions but not diagonally; i.e., there are no edges between vertices 1 and 4 and between vertices 2 and 3. The walls along the borders of regions 1, 2, 3, and 4 are colored purple, cyan, green, and red, respectively, as shown in Fig. 6.1. In addition, regions 1 and 4 have a black floor, and regions 2 and 3 have a white floor. These color features were included to enable each robot to identify its current state (region) through image processing and IR measurements.

A Microsoft LifeCam camera was mounted over the testbed to obtain overhead images of the experiments. The robots were marked with identical yellow square tags, which were detected using the camera (see the red circles in Fig. 6.2). A central computer processed images obtained by the camera and communicated with all the robots over WiFi. The robots did not have access to information about their positions in a global frame.

6.2.2 Pheeno Robot ROS Projects

A series of ROS packages (**Zahi M Kakish**, 2018) were developed in order to streamline physical experimentation and simulation of multi-agent control strategies.

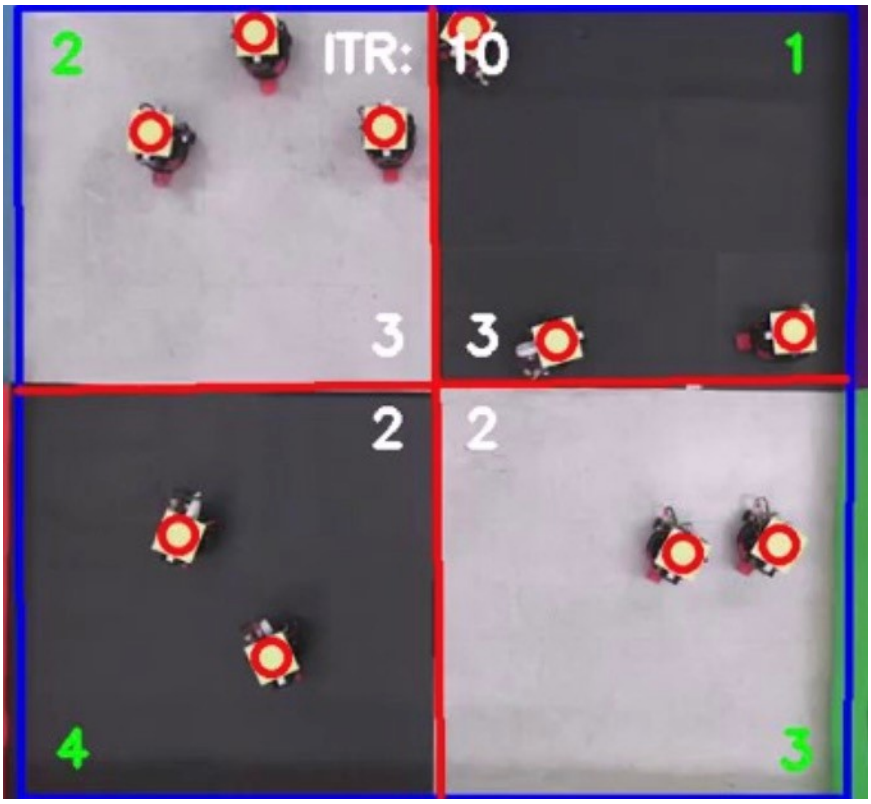


Figure 6.2: Overhead camera view of the testbed during an experimental trial.

These packages allow a user to easily conduct experiments using a single or multiple Pheeno robots (Wilson *et al.*, 2016) in *Gazebo* simulation, *RViz*, and physical environments. Documentation provided for each package makes the Pheeno platform a good beginner's guide for people wishing to learn more about robotics and ROS. In the following table, each package is labeled along with its respective use-case with the Pheeno robot.

6.2.3 ROS Setup

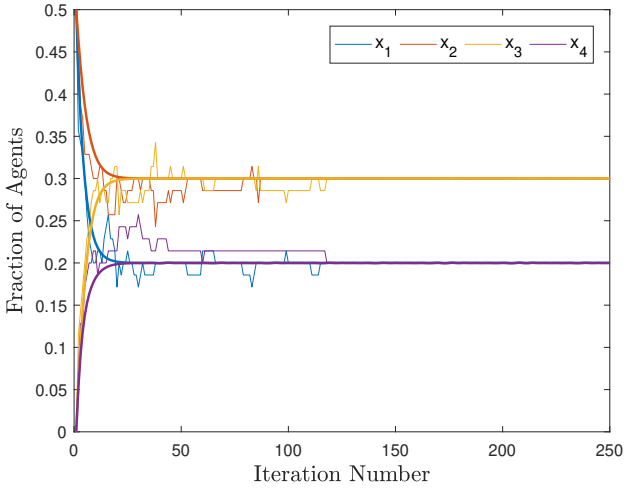
The entire setup utilized ROS middleware. The central computer runs the ROS Master and two ROS nodes: an *overhead camera node* to process images of the testbed from the overhead camera, and a *transition control node* to initiate or end an iteration. The *overhead camera node* calculates the number of robots in each region. This

Table 6.1: Pheeno ROS packages along with their respective use-cases. The third column designates the ROS version for which these packages are available.

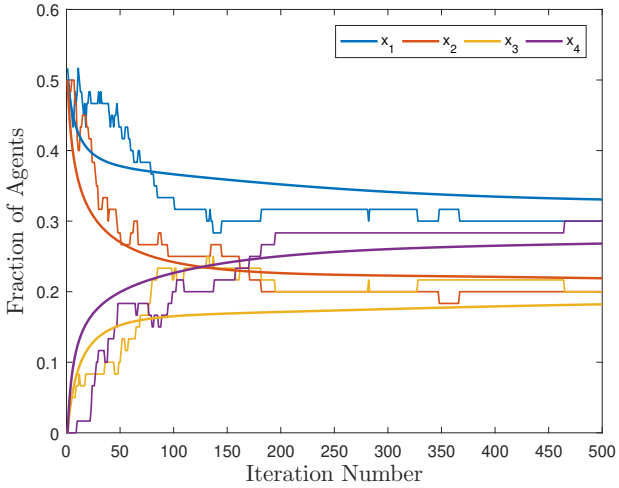
Package	**Description**	**Release**
`pheeno_ros`	Executables for running a Pheeno	Kinetic, Indigo
`pheeno_ros_sim`	Gazebo simulation launch files and simulation files	Kinetic, Indigo
`pheeno_ros_description`	Gazebo and Rviz simulation model files	Kinetic
`pheeno_arduino`	Microcontroller ROS drivers for Pheeno	N/A

calculation does not require the identification of individual robots; instead, the node uses color detection to count the number of yellow identification tags inside each region on the testbed. These numbers are then converted into robot densities in each state and are published on a ROS topic. The *transition control node* monitors the state transition iterations. The node ends an iteration when every robot has reached its desired state.

Each robot runs three ROS nodes: a *sensor node*, a *camera node*, and a *controller node*. The *sensor node* publishes data from all the robot's IR sensors on their corresponding topics and drives the robot's motors by subscribing to movement command topics. The *camera node* publishes raw images from the robot's onboard cameras. Finally, the *controller node* runs the motion control scheme for the robot, described in Section 6.2.4. This node receives the state densities from the central computer's *overhead camera node*. The robot computes the next desired state using the controller



(a) Linear controller with $N = 10$ robots



(b) Nonlinear controller with $N = 10$ robots

Figure 6.3: Trajectories of the mean-field model *(thick lines)* and the robot population fraction in each state, averaged over five experimental trials *(thin lines)*.

input and has to decide whether to stay in its current region or transition to another one.

6.2.4 Robot Motion Controller

Each robot starts in one of the four regions (states), according to the specified initial condition, and is programmed with the desired equilibrium distribution $\mathbf{x}^{eq}$ and the set of transition rates $k_e(\mathbf{x})$ that have been designed by the central computer using one of the procedures described in Section 6.2. The robots know their initial region and update their region after each iteration. At the start of each iteration, the robots receive state feedback $\mathbf{x}$, the current robot densities in each region, from the *overhead camera node*. Using this information, each robot computes its probability $k_e(\mathbf{x})\delta t$, where $\delta t = 0.1$, of transitioning to an adjacent spatial region $T(e)$ within the next iteration. This stochastic decision policy is executed by the robot using a random number generator. If a robot decides to transition to another region, it searches for the color on the wall of that region. As soon as it finds the color, it moves ahead along a straight path. If an encountered object obstructs its path, the robot avoids it and reorients itself toward the target region. Since the robot's onboard camera is unable to detect changes in depth, we assigned each region to have a white or black floor, which can be identified by a bottom-facing IR sensor on each robot. The robot detects that it has entered the target region when it identifies a change in the floor color. The robot then moves forward a small distance, which prevents robots from clustering on the region boundaries, and stops moving. Finally, the robot sends a *True* signal to the *transition control node*. This node initiates the next iteration once it receives a *True* signal from all the robots. The entire process is repeated until the desired distribution is reached by the robots.

6.2.5 Results

The controllers were designed to redistribute a population of $N = 10$ robots on the four-vertex bidirected graph corresponding to the arena. The initial distribution was defined as $\mathbf{x}^0 = [0.5\ 0.5\ 0\ 0]^T$. For the linear controller, the desired distribution was $\mathbf{x}^{eq} = [0.2\ 0.3\ 0.3\ 0.2]^T$, and for the nonlinear controller, it was $\mathbf{x}^{eq} = [0.3\ 0.2\ 0.2\ 0.3]^T$.

Figure 6.3 shows the solution of the mean-field model defined in 6.1 with each of the two controllers and the corresponding robot populations in each state from the experiments, averaged over five trials. For ease of comparison, the total robot populations were normalized to 1. The plots show that the robots successfully redistribute themselves to the target distribution, as predicted by the mean-field model. The nonlinear controller produces a slower convergence rate to equilibrium than the linear controller.

6.3 Centralized Deep Reinforcement Learning Agent Controller

6.3.1 Problem Statement

The notation in Sections 3.2 and 4.2 guide our definitions of this problem statement. However instead of the Kolmogorov forward equation (3.6) and the "leader" modified version (4.9), we define the agent state evolution according to a single time-dependent matrix $\mathbf{K}$ comprised of the sum of our $\mathbf{B}_e$ matrix and our transition rates u_e as follows:

$$\mathbf{K}(t) = \sum_{e \in \mathcal{E}} u_e(t) \mathbf{B}_e \tag{6.3}$$

This leads us to define our bilinear control system as the following discrete-time linear ordinary differential equation (ODE) mean-field model

$$\dot{\mathbf{x}} = -\mathbf{K}(t)\mathbf{x} \tag{6.4}$$

where $\mathbf{K} \in \mathbb{R}^{M \times M}$. Theoretical justification of the linear ODE is given in Gillespie (2007) and Berman *et al.* (2009). The $\mathbf{K}$ matrix has the properties

$$\mathbf{K}^T(t)\mathbf{1} = \mathbf{0} \tag{6.5}$$

$$\mathbf{K}_{ij}(t) \leq 0 \qquad \forall (i,j) \in \mathcal{E} \tag{6.6}$$

which result in the following structure:

$$\mathbf{K}_{ij} = \begin{cases} -k_{ji} & \text{if } i \neq j,\ (j,i) \in \mathcal{E}, \\ 0 & \text{if } i \neq j,\ (j,i) \notin \mathcal{E}, \\ \sum_{(i,l)\in\mathcal{E}} k_{il} & \text{if } i = j. \end{cases}$$

Each element k is a reaction rate that models the agent population inflow and outflow of a vertex. Figure 6.4 illustrates the bidirected graphs with edges associated with the their respective rate k. Instead of using an ij subscript to denote the source vertex i and target vertex j, we label them with a value 0 to $N_{\mathcal{E}}$, where $N_{\mathcal{E}}$ is the number of edges within a graph. The associated source and target, i and j, can be inferred from the graph due to the directional arrow. Equation (6.4) models the state evolution of $\mathbf{x}$ over time t if each reaction rate k is known. Compared to our work in Chapter 4, we do not train a single control policy using data generated from the mean-field model. Instead, we train $N_{\mathcal{E}}$ control policies which output individual k reaction rates that compose the $\mathbf{K}$ matrix. The outputs of the linear ODE model are then used to train and optimize the policies which output the reaction rates k. Therefore, our problem becomes:

Problem 6.3.1. *Given a target agent distribution $\hat{S}_{target}$, and defining $A_e = \mathcal{N}(\mu_e, \sigma_e)$, devise $N_{\mathcal{E}}$ control policies $\pi_e : \mathcal{P}(\mathcal{V}) \times \mathcal{V} \to A$ that drives the agent distribution to $\hat{S}(T) = \hat{S}_{target}$, where the final time T is as small as possible, by minimizing the total reward $\sum_{t=1}^{T} R_e(t)$.*

6.3.2 Leaderless Controller Design

Our devised leaderless controller draws on the advantage actor-critic defined in Section 4.3 but modifies it into a multi actor, global critic framework. Each reaction within the $\mathbf{K}$ matrix defined in (6.4) represents a particular edge e and will have associated with it an actor that takes the population fractions at vertices $\sigma(e) = i$ and $\tau(e) = j$ as inputs. Instead of each actor paired with a critic with similar inputs, we rely on a single global critic that is a function of the agent population fractions at all M vertices. Therefore, gradient estimations of individual actors are taken with respect to the global critic. The overall controller architecture using the multi-actor, global-critic design is illustrated in Figure 6.5.

In Section 4.3, we presented our control policy inferring discrete actions $a \in \mathcal{A}$. Therefore, We represent the transition rates k_e as samples from a normal distribution with mean ϕ_e and standard deviation λ_e,

$$k_e \sim \mathcal{N}(\phi_e, \lambda_e). \tag{6.7}$$

To enable our control policy to learn the transition rates, we modify our actor networks to output the continuous parameters ϕ_e, λ_e for each edge $e \in \mathcal{E}$. Both the mean and standard deviation are clipped between 0.2 to 0.8 and 0 to 0.1, respectively. The reaction rate k_e is thus a sampled random variable from the normal distribution:

$$\mathcal{N}_e(\phi_e, \lambda_e) \sim k_e \tag{6.8}$$

which modifies our actor gradient estimation from our original policy

$$\bigtriangledown_\theta J(\pi_\theta) \;=\; \mathbb{E}_t\left[A_t^\pi \;\bigtriangledown_\theta \log \pi_\theta(a_t|s_t)\right]$$

to a policy that is the log of the normal distribution

$$\bigtriangledown_\theta J(\pi_\theta) \;=\; \mathbb{E}_t\left[A_t^\pi \;\bigtriangledown_\theta \log \pi_\theta(\mathcal{N}(\phi,\lambda))\right] \tag{6.9}$$

However, we cannot use the advantage function when calculating the policy gradient as defined in Section 4.3 since it requires calculating the difference between an estimated state-action value $Q(s,a)$ and the critic value $V(s)$ for the same s used by both actor and critic. In our leaderless approach, the state used by the each actor is $S_p \subset S_{env}$, where p is an edge of the graph, $p = \{1,\ldots,N_\mathcal{E}\}$, defined as

$$S_p = [\; x_i,\; x_j \;] \qquad \text{if} \;\; p = (i,j) \in \mathcal{E}. \tag{6.10}$$

The state subset S_p is composed of only two agent population densities since an edge only connects two vertices within a graph. The global critic uses the full state $S_{env} = [x_0,\ldots,x_M]$ when calculating the state value $V(S_{env})$. Therefore, we employ two separate reward functions to train our actors and global critic. The first is the difference between the desired population fraction at a vertex $x_{d,m}$ and the current population faction at a vertex x_m

$$r(S_p) = x_{d,m} - x_m \qquad \text{if} \;\; \sigma(p) = m \tag{6.11}$$

which is used by the multiple actors. The value m refers to the source vertex within the edge p for the reaction k_e, *i.e.* $\sigma(e) = m$. There are M actor reward functions since multiple edges can have the same source vertex. The discounted return for the actor of edge p with source vertex m is as follows:

$$G_p(t) = r_t(S_{p,t}) + \gamma r_{t+1}(S_{p,t+1}) + \gamma^2 r_{t+2}(S_{p,t+2}) + \ldots + \gamma^T r_{m,T} = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}(S_{p,t'}) \tag{6.12}$$

The global critic maintains the use of the negative mean-squared error as defined previous in Sections 3.2 and 4.3.1 since it provides an adequate measure of the distance the overall agent population S_{env} is to $\hat{S}_{target}$. One major drawback from the actor's reward function is that it does not provide a value pertaining to the full system since the actor only uses the locally observable information of the vertex m. Therefore, we modify our policy gradient estimation to include returns from the actor and the global critic

$$\bigtriangledown_\theta J(\pi_\theta) \;=\; \mathbb{E}_t \left[C_p^\pi(t) \;\; \bigtriangledown_\theta \log \pi_\theta(\mathcal{N}(\phi, \lambda)) \right] \tag{6.13}$$

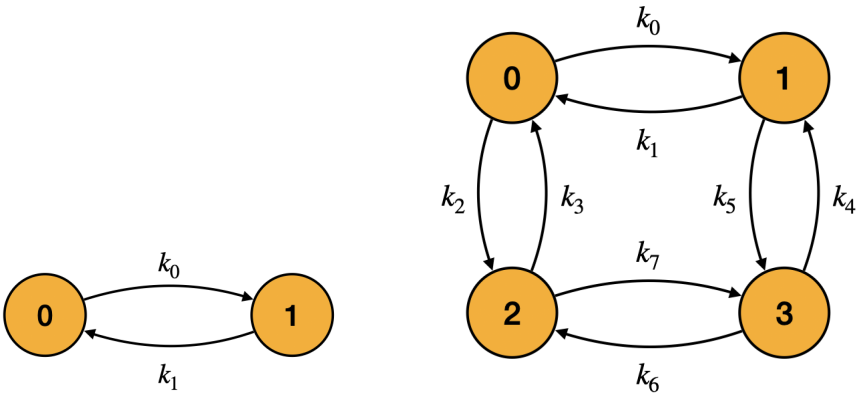
where $C_p^\pi(t)$ is a globally-scaled local return function

$$C_p^\pi(t) = G_p(t) V_t(S_{env,t}) \tag{6.14}$$

By multiplying the actor return $G_p(t)$ by the state value estimate of the fully observable global critic, we ensure that the gradient estimation will scale depending on how close the system is to the desired population density. Having our global critic trained on the MSE of system will predictably make smaller valued outputs if the system is close to the target population density S_{target}. Thus, the actor's return G_p acts as a guide for the individual actor loss in relation to its distance from the target population fraction at the source vertex $x_{d,m}$.

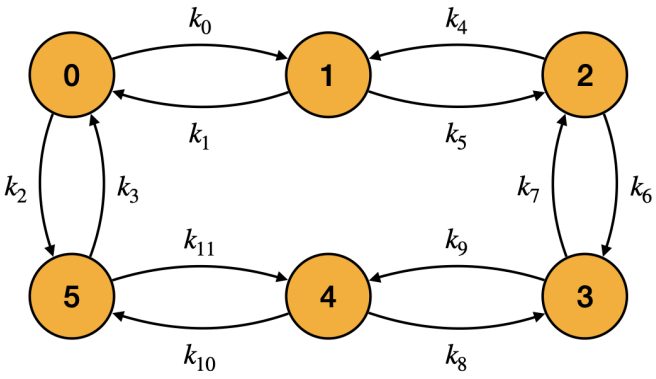
6.3.3 Training

The control policies are trained using TensorFlow (Abadi *et al.*, 2015) on 2, 4, and 6 node bidirected graphs shown in Figure 6.4. Each neural network for the L actors and global critic consist of two hidden layers of 32 units. Each hidden layer uses an $ReLU$ activation function (Nair and Hinton, 2010). The actor's output layer use two different activation functions: a *sigmoid* function for the mean ϕ and a *softplus* function for the standard deviation λ. Each training cycle is comprised of 2500



(a) 2 Node Graph

(b) 4 Node Graph



(c) 6 Node Graph

Figure 6.4: 2, 4, and 6 vertex bidirected graphs.

episodes with 100 iterations of the linear ODE (6.4). At every iteration the actors populate the $\mathbf{K}$ matrix with the reaction rates k_e. Predictably, the larger the graph $\mathcal{G}$ the larger the number of forward passes of the actor's neural network are required to infer the reaction rate per iteration. The initial agent population fraction used by the mean-field model to reach a target population fraction during training are

$$\hat{S}_{initial} = [\ 0.8,\ 0.2\ ] \quad \hat{S}_{target} = [\ 0.2,\ 0.8\ ]$$

for the 2 node bidirected graph,

$$\hat{S}_{initial} = [\ 0.4,\ 0.1,\ 0.1,\ 0.4\ ] \quad \hat{S}_{target} = [\ 0.1,\ 0.4,\ 0.4,\ 0.1\ ]$$

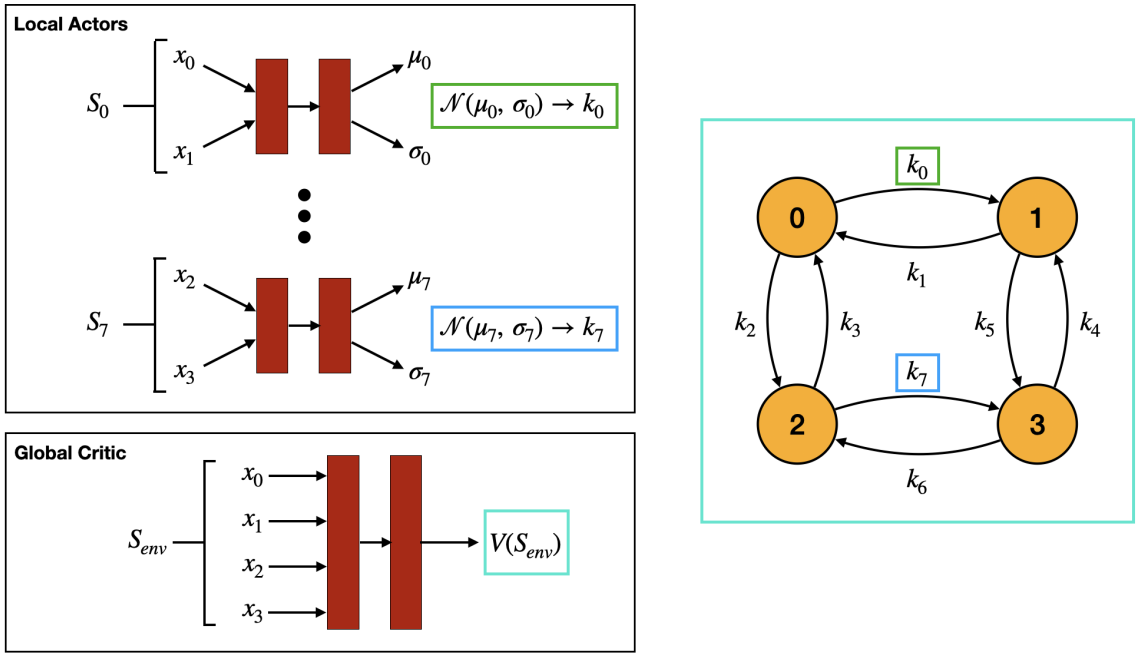


Figure 6.5: Our centralized linear controller composed of M local actor MLPs and a single global critic. The output of each actor are the parameters μ and σ of a Normal distribution. The random variable sampled from the distribution corresponds to the reaction rate k_M on the 4 node bidirected graph. The global critic provides insight into the whole system by inputting the population densities within every vertex.

for the 4 node bidirected graph, and

$$\hat{S}_{initial} = [0.2,\ 0.1,\ 0.2,\ 0.15,\ 0.2,\ 0.15] \quad \hat{S}_{target} = [0.1,\ 0.2,\ 0.05,\ 0.25,\ 0.15,\ 0.25]$$

for the 6 node bidirected graph. We use an Mean-Squared Error stopping criteria $\mu = 0.005$ to provide the policy feedback that population densities in the graph have converged to $\hat{S}_{target}$ similar to the one employed in Chapters 3, 4, and 5. The discount factor for both actor and critic reward functions are $\gamma = 0.99$. The learning rates for the global critic and the edge dependent actors are $\alpha_C = 0.0001$ and $\alpha_A = 0.00001$. The actor learning rate is dropped to 0.000001 when training the control policy on the 6 node bidirected graph. The global critic neural network loss function is the Huber Loss (Huber, 1964) to help minimize the chance of outlier data points impeding the gradient descent towards an optimal policy. We use an Adam optimizer (Kingma and

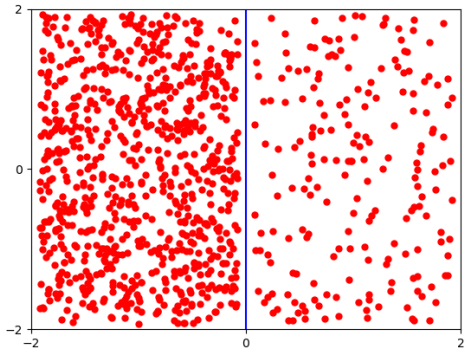
Ba, 2017) for both the global critic and edge dependent actors. The control policies are trained on a desktop with a 4th-generation Intel i7 4-core CPU with 32GB of RAM running Ubuntu 20.04. The code is run in an Docker container to keep the software environment consistent between training runs.

6.3.4 Simulations

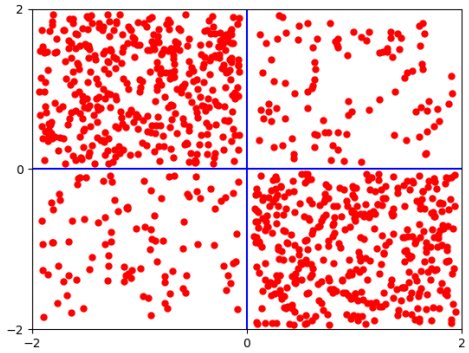
After training, the leaderless policy is tested on a 2, 4, and 6 node bidirected graph with $N = 100$ and $N = 1000$ agents simulated using *matplotlib* (Hunter, 2007). Figure 6.6 presents the simulated bidirected graphs representations with $N = 1000$ robots. Each trained policy is run on their respective trained graph size for $t = 100$ iterations. At every time step, an individual agent will perform a forward pass on the leaderless policies whose source edge is the vertex the agent currently resides $\sigma(e) = i$. The agent will then sample a random variable from a uniform distribution between 0 and 1. If the variable is greater than the reaction rate k_e from the leaderless policy, the agent will transition to the vertex that is the target vertex of $\tau(e)$. The variable is compared to cumulative sum of reaction rates for cases where more than one edge is a source $\sigma(e)$ sharing the agents current vertex position. For example, an agent located in vertex 0 in a 4 node graph computes two reaction rates: k_0 for edge $e = (0, 1)$ and k_2 for edge $e = (0, 2)$. A random variable Y from a uniform distribution is compared to the rates as the following piecewise function:

$$D(t) = \begin{cases} 1 & \text{if } Y < k_0, \\ 2 & \text{if } Y < k_0 + k_1, \\ 0 & \text{otherwise} \end{cases} \tag{6.15}$$

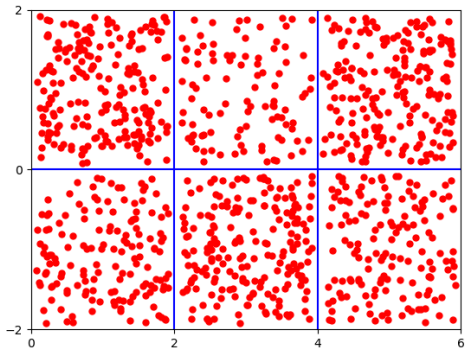
where D is a piecewise function that specifies which state the agent will transition to given the random variable Y. The simulation will move to the next time step after



(a) 2 Node Simulation



(b) 4 Node Simulation



(c) 6 Node Simulation

Figure 6.6: Python simulation displaying the 2, 4, and 6 vertex bidirected graphs at $t = 0$ for agent populations $N = 1000$. At each time step, an agent may transfer to an adjacent vertex or stay within the same vertex depending on the output of the centralized linear controller.

all the agents have either transitioned to a new spatial state or remained within their current one. In addition, the current agent population fractions within the graph are recorded along with the MSE used during training as the stopping criteria μ.

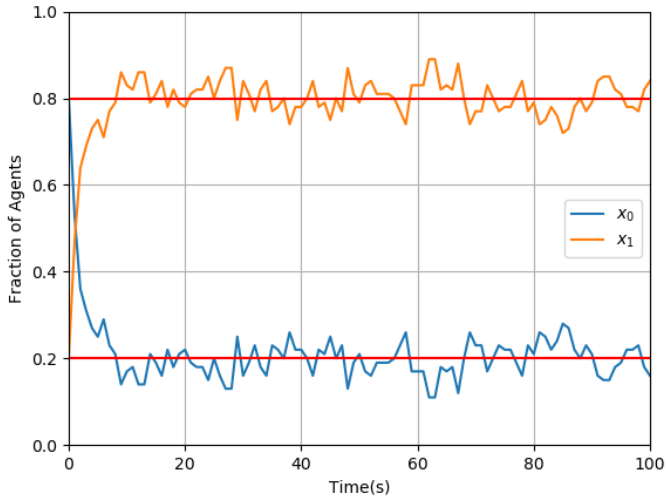
6.3.5 Results

The resultant agent population fraction evolution over time for each simulated test runs are presented in Figures 6.7, 6.9, and 6.11. For the 2 and 4 node graph

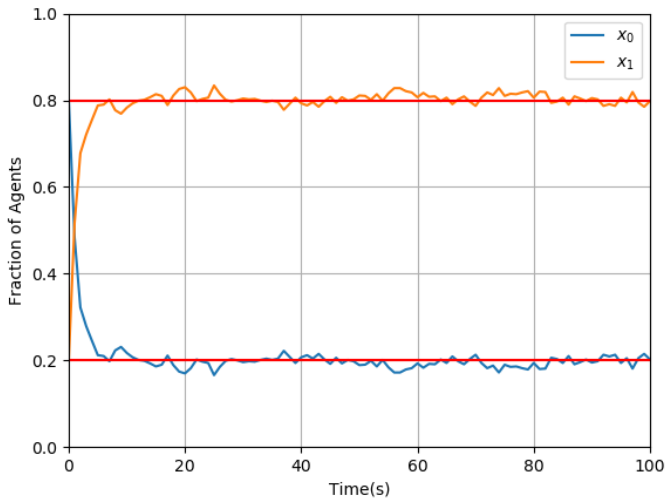
case, we notice that the leaderless deep RL controller provides a quick transient response of approximately $7 - 10$ seconds for both $N = 100$ and $N = 1000$ agent cases. The lower agent number for these cases tended to oscillate above and below the desired population densities x_d at higher frequency in comparison to stability seen when $N = 1000$. The 6 node graph provided degraded performance at the lower agent populations in terms of stability and transient response. For both N agent population cases, we see roughly $10 - 15$ second steady-state response. The stability is worse in terms of maintaining the x_d for each vertex. We believe this is likely due to poor choice of hyperparameters during training. In addition to measuring the population fractions of agents per time step, we plotted the stopping criteria μ as shown in Figures 6.8, 6.10 and 6.12. In previous chapters, μ acted as a measure of convergence towards the desired population densities $\mathbf{x}_d$. All three bidirected graph sizes at both agent populations showed proper convergence to or below our desired $\mu = 0.005$. The 6 node graph at an $N = 100$ showed more erratic behavior once it reached the $\mu = 0.005$; however, it still did maintain an average steady state around our desired μ value.

6.4 Robot Simulation

We test the applicability of our approach on a simulated 4 node bi-directed graph testbed using 10 simulated GRITSBotX robots. The simulated environment is provided by the `ros2_robotarium` package: a ROS 2 Thomas *et al.* (2014) and Gazebo Aguero *et al.* (2015) recreation of the Georgia Tech Robotarium Wilson *et al.* (2020) developed by our laboratory. Figure 6.13 shows the simulated testbed onto which 10 GRITSBotX robots are distributed according to $\hat{S}_{initial}$. The bi-directed graph is represented as a checkered pattern on the surface of the testbed. A robot is considered to have transitioned from one vertex to the next by crossing the barrier between



(a) Deep RL controller with $N = 100$ robots



(b) Deep RL controller with $N = 1000$ robots

Figure 6.7: Agent population fraction simulated results per time step on a 2 node bidirected graph at (a) $N = 100$ and (b) $N = 1000$. Agent populations started at $\hat{S}_{initial} = [0.8, 0.2]$ to an $\hat{S}_{target} = [0.2, 0.8]$. Two red lines are plotted at 0.2 and 0.8 to assist in visualizing the transient response of the agent population fractions to the target distribution over time due to the deep RL leaderless controller.

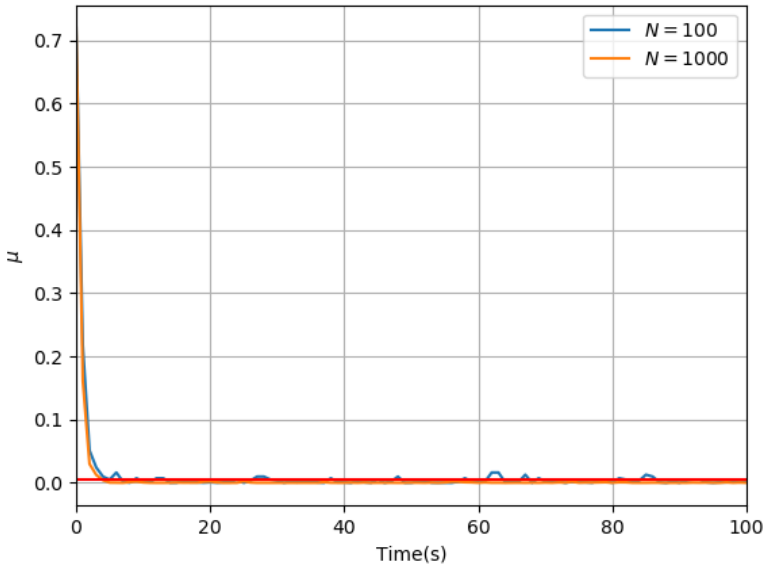
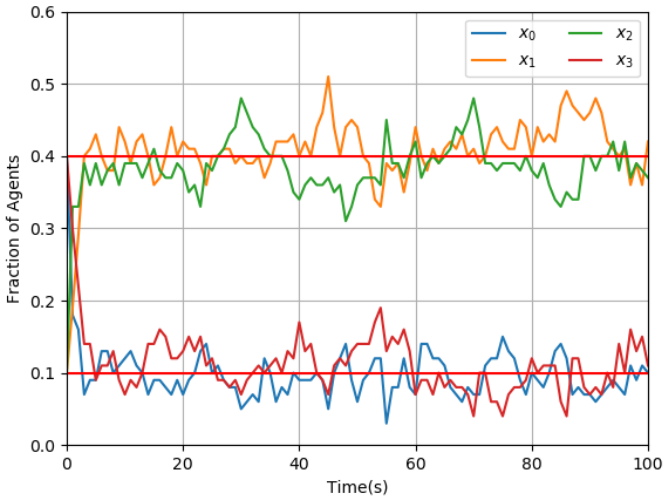


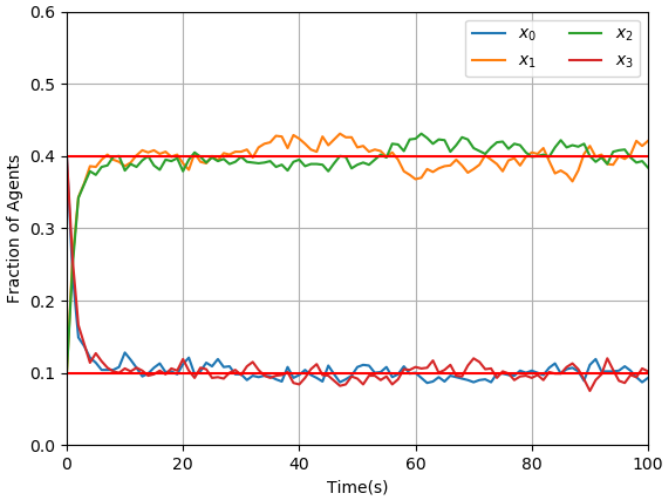
Figure 6.8: System MSE response of $N = 100$ and $N = 1000$ agents on a 2 node bidirected graph to the baseline stopping criteria $\mu = 0.005$ used during training.

the colored grid boxes. A camera object placed above the testbed was added for an overhead view to display pertinent information such as the current iteration and the current population density within each vertex. The simulation is run until the population is distributed on the graph according to $\hat{S}_{target}$.

At each iteration, individual robot receive the population fraction within it's current state and the two adjacent vertices. The robot then select the two proper actor control policies for edges that link the current vertex to the adjacent target vertex. A forward pass from both networks will results in two reaction rates. The robot samples a random number from a uniform distribution $X \in [0, 1)$ which it will then compare to the reaction rates inferred by the two control policies as in (6.15).



(a) Deep RL controller with $N = 100$ robots



(b) Deep RL controller with $N = 1000$ robots

Figure 6.9: Agent population fraction simulated results per time step on a 4 node bidirected graph at (a) $N = 100$ and (b) $N = 1000$. Agent populations started at $\hat{S}_{initial} = [0.4, 0.1, 0.1, 0.4]$ to an $\hat{S}_{target} = [0.1, 0.4, 0.4, 0.1]$. Two red lines are plotted at 0.1 and 0.4 to assist in visualizing the transient response of the agent population fractions to the target distribution over time due to the deep RL leaderless controller.

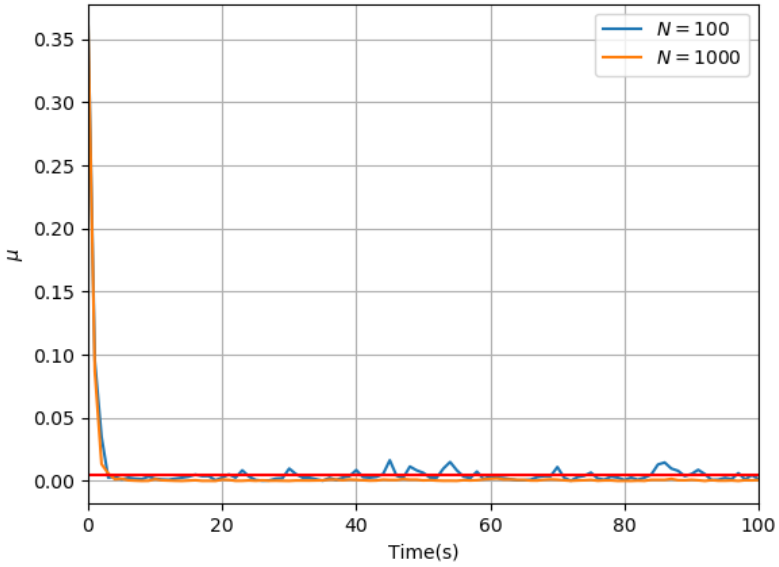
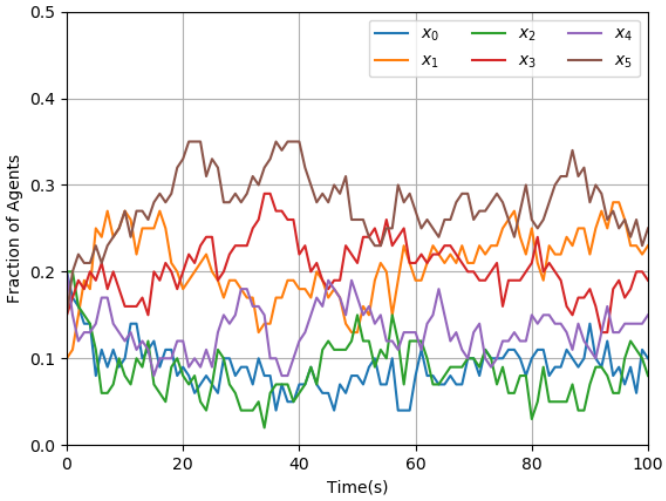
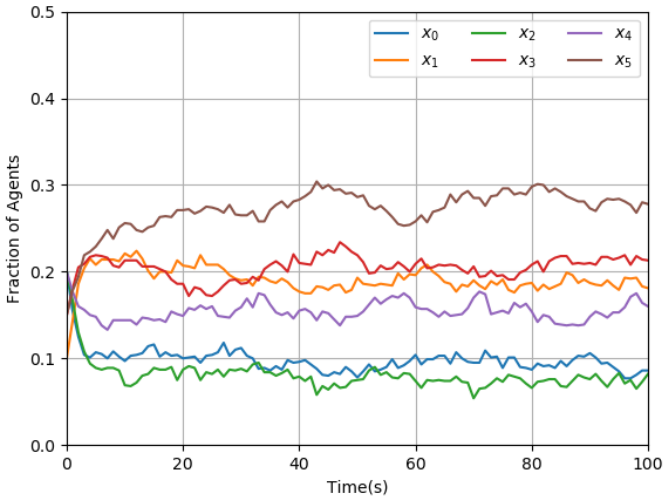


Figure 6.10: System MSE response of $N = 100$ and $N = 1000$ agents on a 4 node bidirected graph to the baseline stopping criteria $\mu = 0.005$ used during training.



(a) Deep RL controller with $N = 100$ robots



(b) Deep RL controller with $N = 1000$ robots

Figure 6.11: Agent population fraction simulated results per time step on a 4 node bidirected graph at (a) $N = 100$ and (b) $N = 1000$. Agent populations started at $\hat{S}_{initial} = [0.2, 0.1, 0.2, 0.15, 0.2, 0.15]$ to an $\hat{S}_{target} = [0.1, 0.2, 0.05, 0.25, 0.15, 0.25]$. Red lines to aid in visualizing the target distribution are not provided to prevent over-complicating figure.

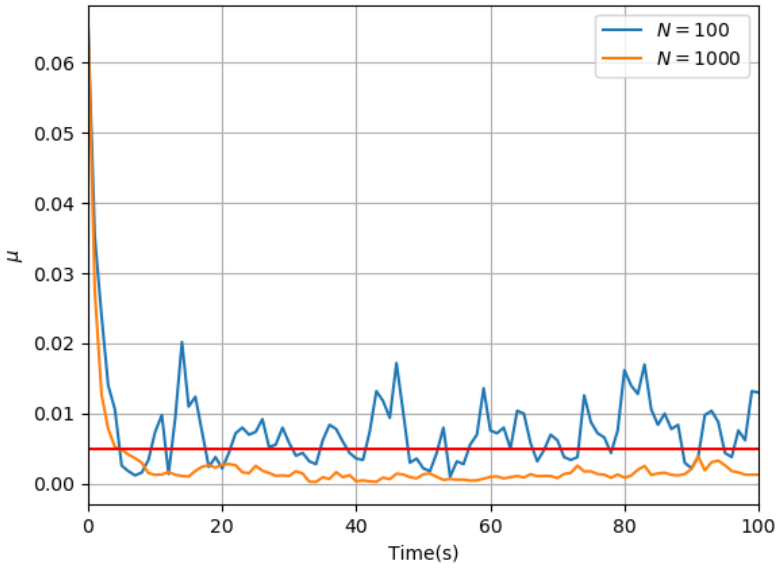


Figure 6.12: System MSE response of $N = 100$ and $N = 1000$ agents on a 6 node bidirected graph to the baseline stopping criteria $\mu = 0.005$ used during training.

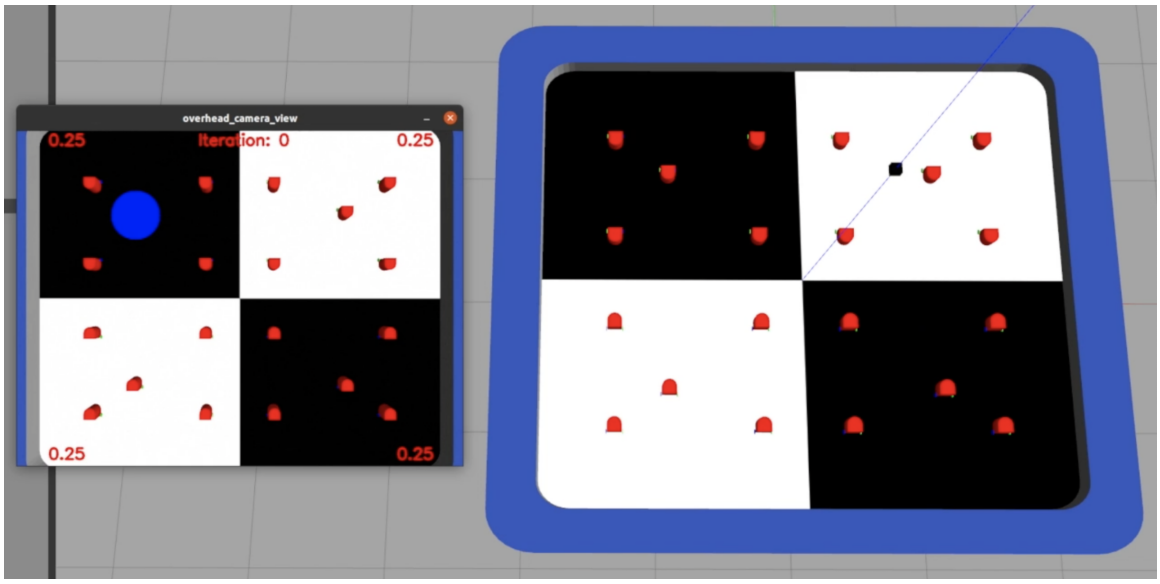


Figure 6.13: Gazebo testbed for a 4 node bidirected graph represented as a checkered pattern. 10 GRITSBotX robots are distributed evenly over the graph. A small window on the left gives an overhead view of the testbed and a superimposed text displaying the iteration and population fraction within each vertex.

Chapter 7

APPLICATIONS FOR RL-BASED SWARM CONTROL STRATEGIES TRAINED ON MEAN-FIELD MODELS

7.1 Summary

In this chapter, we explore two applications of multi-robot systems using different control paradigms that may benefit from using a mean-field model based approach towards controlling multiple robots. Both the applications we shall discuss involve human operators that have an overall control over a number of robots within a particular system, yet these systems become unreasonable for a single operator to complete or allocate tasks when robot population scales too much. The work in previous chapters is capable of alleviating these problems for an operator wishing to delegate tasks or spatial states on-the-fly by providing optimal control policies that can be trained with the mean-field model thereby limiting time to implementation and increasing usability at population scale.

The first application is an open-source AI assistant to coordinate multiple agents on a lunar pole for ice prospecting. In this application, *sites-of-interest* are allocated by a human operators to areas where they believe ice may be found. However, the areas where the robots traverse in search of water contain numerous hazardous environmental hurdles that can ruin a mission. Communication delays, coupled with interference from the harsh lunar topology at the poles, limits the spatial areas that a rover can traverse. We present an AI assistant that can keep multiple science rovers within safe communication distance of one another to reach sites-of-interest.

Second, we investigated human-in-the-loop approaches for managing a swarm of

robots to prevent cognitive overload, which can impair a human operator's ability to continue supervising the swarm (Chen and Barnes, 2014). Various approaches have been attempted to control multi-robot systems by other physical means, including wearable devices using haptic feedback (Music *et al.*, 2017; Ferrer, 2018) and electroencephalogram (EEG) brain-computer interfaces (BCI) (Karavas *et al.*, 2017). However, both these approaches are hindered by the necessity of wearable hardware to function and cumbersome solvent application process to read noisy EEG signals from the cranium, respectively. We present an initial design and validation of an approach to human-swarm interaction based on robot recognition of sequential hand gestures. While not a fully decentralized control approach, it demonstrates the ability to leverage recent improvements in the object classification abilities of Convolutional Neural Networks (CNN) that run on small, low-cost, low-powered computational devices that are typically used on swarm robotic platforms.

7.2 Open-source AI Assistant for Cooperative Multi-agent Systems for Lunar Prospecting Missions

This section contains results from **Zahi M Kakish** *et al.* (2019).

7.2.1 Prospecting Mission Overview

The Resource Prospector (RP) mission aimed to perform "In-Situ resource utilization" (ISRU) using a rover on the Lunar surface. However, the inclusion of ISRU instruments on the rover increases the complexity of classical navigation and exploration tasks. Thus, the process for controlling a rover during a prospecting mission is composed of two Mission Control systems: the Mission Control located on Earth surface (MCoE) and the Mission Control located on the rover vehicle (MCoR). The

former is the one managed by a mission team (human operators) facing a front-end interface and selecting the best driving distances and areas using available information from rover sensors. As a result of this process, they will command the rover to visit a new position or to perform ISRU. The latter is the one deployed on the robot, and allows manual control, teleoperated by a human operator, and semi-autonomous control, where the robot is running with partial supervision. The MCoR would be seen as the control unit for the localization, navigation, and prospecting systems. The robot generates actions and gathers information from its sensors for relaying to the MCoE. Nevertheless, current missions entail several major challenges: 1) Moon-to-Earth communications and interactions with human operators in the loop, 2) harshness and unpredictability of the environment, and 3) the limitations and lack of robustness of single-robot missions.

Given the Earth-Moon geometry, communication between a rover on the Moon and mission control on Earth is not a straightforward process. The round-trip communication latency for a mission on the Moon is significantly shorter (varying between 6 seconds to over 25 seconds for RP) than Mars, which could expect a response time of 28 minutes for a software ping or to perform a rover operation (Gordon, 2012). This scenario discourages real-time command and control. Thus, any action undergone by the rover is prepared and meditated by a team that decomposes each task to fine-grained steps. The idea is to define decision cycles for providing near real-time command and control by means of driving the rover using a waypoint approach Andrews (2015).

The Moon's environment (Heiken *et al.*, 1991) is composed of numerous impediments to simple operations which operators are accustomed to on Earth. It has a solid-surface body made up of a rocky surface covered with regolith. Consequently, the regolith material and spectral properties make traversal difficult using conventional

platforms. Regolith covering the surface varies between 3 meters and 20 meters. The Moon presents (Besancon, 2013) a crust size of 60 to 100 kilometers thick. In addition, the Moon has other properties (Berkelman *et al.*, 1995) such as dust constraints, thermal properties, and vacuum conditions which make operation difficult.

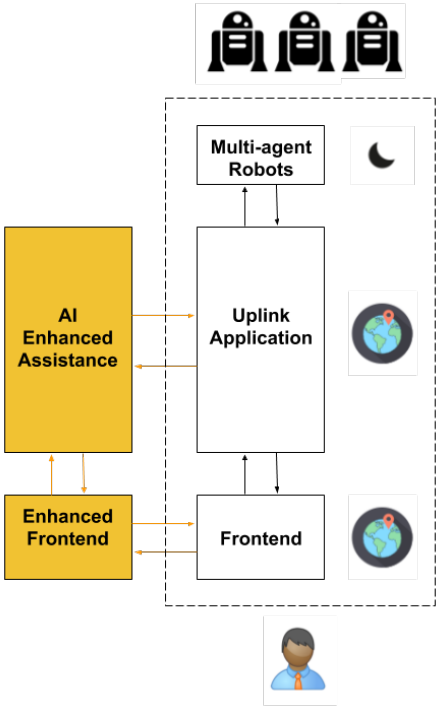


Figure 7.1: Overview of a mission control. Expected goals (supported on AI) vs current scenarios (supported on Earth surface mission control).

In the RP mission, the single rover lacks autonomy, traversal versatility, communication capabilities and characteristics for reaching the goal in an expected amount of time. These challenges can be significantly reduced by multi-robot operations, which can improve mission efficacy and lower risks by autonomously coordinating the agents. Despite continual advances in the field, there are still numerous challenges to reduce the complexities associated with the coordination of multi-robot systems and mission planning strategies, which motivates further research.

We propose a set of tools capable of enhancing the planning capabilities of the

Mission Control Team on Earth during the decision making process of prospecting and exploration missions. In addition to providing operators with important information, the system assists with the control of a multi-agent configuration of rovers. The first iteration of development presents a mission generator that provides multiple possible plans for conducting a prospecting mission on the Moon. Figure 7.1 presents the overview and contributions of our project and the current status of a mission definition.

Before explaining the MARMOT solution, it is necessary to identify the main features associated with MCoE and MCoR. The Moon context clearly bounds the possibilities available for offering a guidance system for Mission Control on Earth. In addition, current data acquisition capabilities of rover sensors and satellites limits the information available to make the best decision. Thus, this project is split into three main elements: 1) Cost Map Generation, 2) Waypoint Generation and Guidance, and 3) Multi-agent Strategy.

7.2.2 Cost Map Generation

To produce a usable cost map, a blank grid-graph containing vertices and edges the size of the operational area is generated. Next, each vertex is associated with it information about the environment. The information is used to generate a cost associated with that vertex for eventual use in path planning strategies. For example, the distance of a vertex to an obstacle is assigned a low value if the vertex is far from obstacle, but a high value is assigned if it is closer. The values given to vertices on the grid-graph are taken from fusing multiple maps in layers of the same area containing different information (Lu *et al.*, 2014; Boumghar *et al.*, 2011). In the case of a Lunar environment, the fusion entailed layering terrain, slope, shadows, and communication maps.

Lunar Terrain and Slope Maps Currently, there are no publicly available high-resolution, finely-detailed maps of the Lunar surface. A terrain map must thus be made utilizing information that *is* publicly available. *Moon Trek* (JPL, 2018), a website hosting publicly available Moon data organized by NASA JPL, provides easy and free access to Lunar surface data. We focused our efforts on the crater *Hermite A* located on the Moon's North Pole since this was the tentative operational area of the Resource Prospector (Mahoney, 2018) mission. Inspired by images from the Lunar **O**rbiter **L**aser **A**ltimeter on board the *Lunar Reconnaissance Orbiter* (LRO) of the area, an artificial Lunar surface was created with the help of a 3D open source creation suite, *Blender* (Community, 2019).

From the LRO mission, Digital Elevation Models (DEM) and other surface information provided approximate values of the Moon's topological surface, which was used for creating mock Lunar area called the *terrain-slope* map. Figure 7.2 is one such example of real data topological surface where each pixel represents terrain inclination. In the image, black pixels correspond to flat surfaces while white pixels are steep slopes. The grayscale variations in-between black and white scale from shallow (dark grey pixels) to steep (light grey pixels) surfaces. Figure 7.3 demonstrates the original image DEM converted and modified to a similar artificial surface using Blender.

Shadow Map The 1.5° tilted position of the Moon causes different shadow periods on surface. Near the poles, some craters have local depressions in continual permanent shadow while, on the other hand, we can find craters with the outer edge remaining sunlit throughout the year. There are identified four different areas on Lunar surface: Sunlit, Short Duration Sunlit, Shadowed Near Sunlit, and Permanently Shadowed Regions (PSR). It is important to highlight that data obtained by Lunar Reconnaissance Orbiter suggest it is possible to find thick ice and other volatiles on the top

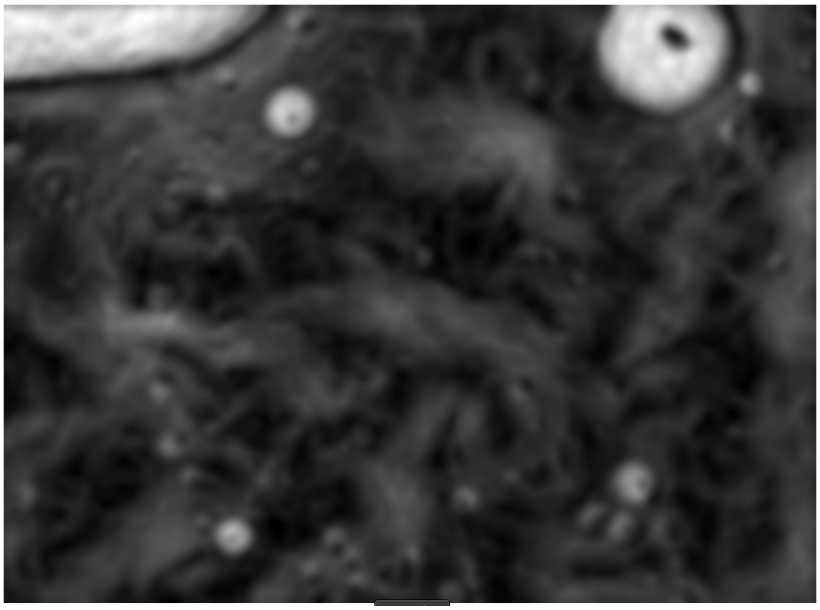


Figure 7.2: Terrain slope representing terrain inclination at each pixel.

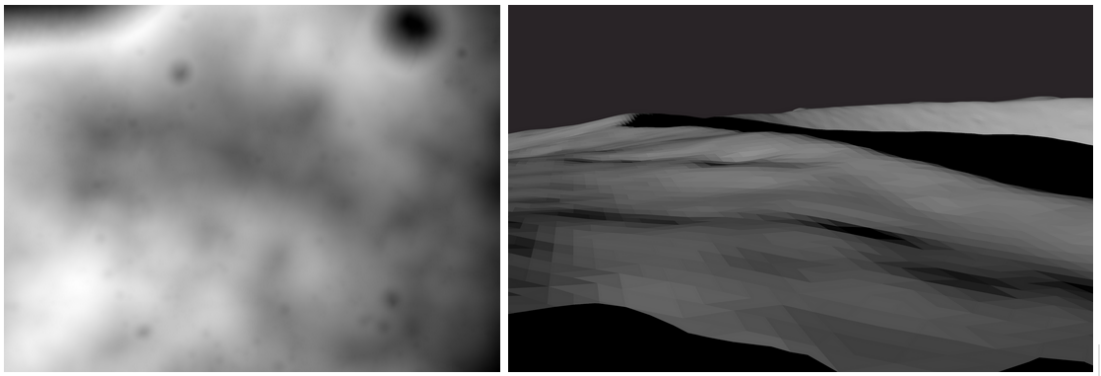


Figure 7.3: Moon surface digital elevation model DEM (left) and a generated view in Blender from this DEM (right).

1-2 meters of regolith on shadowed regions at the lunar poles. Those are thus regions of interest for the prospecting missions, and, for this reason, current missions to the Moon require solar power for energy. A rover caught in a shadowed region can have a detrimental effect on a mission, which include the inability to recharge and having to endure large deviations in temperatures. Data has shown that temperatures in shadowed regions are as low as $-250°C$. Under these circumstances, the *shadow* map layer consists of marking areas that are consistently covered in shadow. However,

as explained earlier these shadows change throughout time. A path obtained taking into account the lunar shadow layout at a certain period of time could be dangerous several hours later.

Using Blender, a simulated Moon cycle can provide the needed shadow information for a potential site as demonstrated and expanded upon in Figure 7.4. The figure provides a great visualization of the variability in surface illumination over time. The map is then used as a basis for generating a cost attributed to shadow coverage. Areas that are more frequently covered in shadow are given a higher costs, while areas with little shadow coverage are given lower costs. This translates to a shadow map that can assist a rover in navigating away from areas that are deemed high-risk due to the prevalence of shadows.

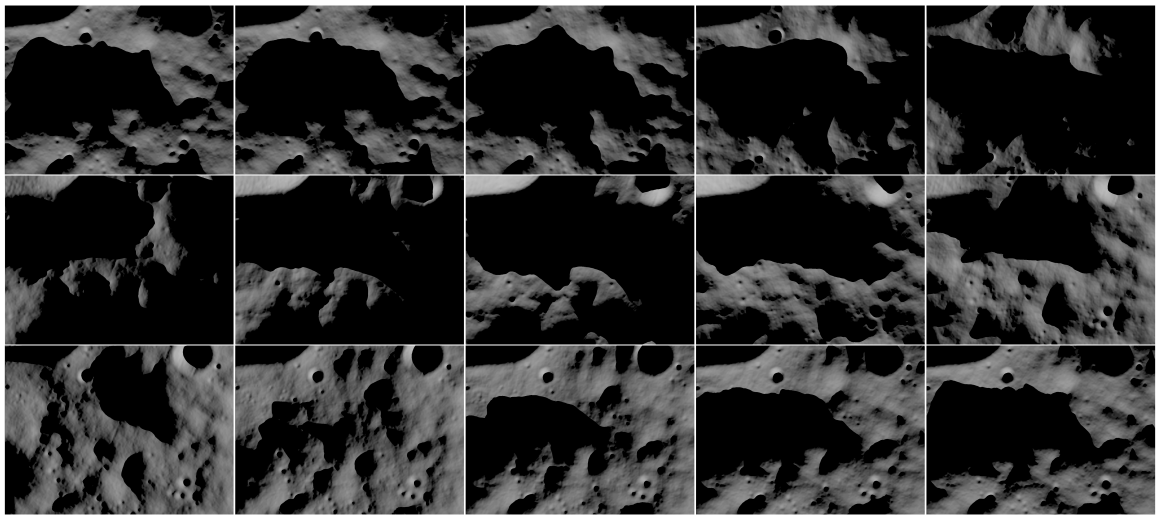


Figure 7.4: Shadow shapes on Moon surface greatly change with respect to moon-sun angle along the moon cycle (28 days). The images show the evolution every 44 hours. Brighter areas are those with exposure to the sun, while the darker areas are occluded by terrain into darkness.

Communications Map During a Lunar mission, direct communication with Earth is required to oversee and control rover operations. However, given certain rover hardware and autonomy limitations, including issues with geographical location, main-

taining direct communication is not consistently achievable. Improving the communication hardware available to a rover is a simple solution, yet this is economically infeasible for many small space companies or research initiatives. Therefore, we explore indirect communication methods using multiple agents as a possible solution; for example, a stationary lander acting as an intermediary between a rover and Earth. Pal and Tiwari (2013) analyzed this type of multi-agent scenario with assumed perfect communication between the lander and rover. This, however, does not accurately represent the communication connection strength between a lander and rover on a real Lunar mission. Multiple factors such as the Moon surface topology, distance-dependent constraints, and antenna intrinsic features (Bapna *et al.*, 1996) significantly complicate communication in this configuration. Additionally, certain operations require the rover to explore areas where communication is completely hindered (such as a crater), and necessitates the assistance of a relay rover to maintain connection.

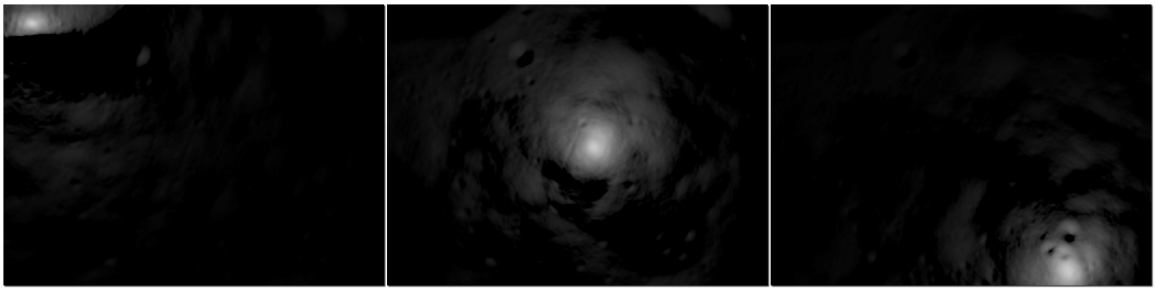


Figure 7.5: Three communications models generated in Blender (ray tracing) showing scientific rover (left), relay rover (middle), and lander (right) communication regions. The scale of worst to best communication coverage and strength.

A *communication* map is thus derived from modeling these constraints and simulating them in a Blender environment. Figure 7.5 shows a Blender generated communication map. The figure shows three images with a representation of the strength and coverage of rover transmission based on its location. The values scale from black (worst) to white (best) for each pixel in the image. The fusion of thse single commu-

nication models produces the complete communication region. Communication cost maps generated from this image contain values analogous to the image's grayscale values. Figure 7.6 is obtained as a result of this process.

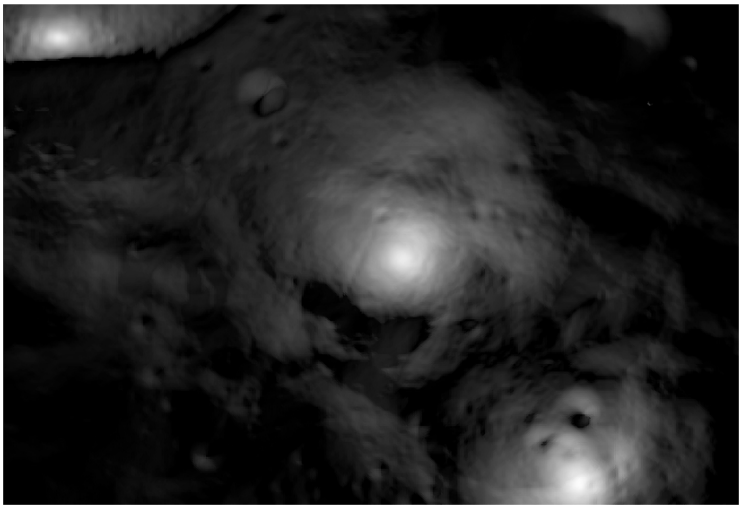


Figure 7.6: Total communication coverage. The grayscale of worst to best communication coverage and strength.

7.2.3 Waypoint Generation and Guidance

It is necessary to locate and select those scientifically interesting targets which could be great sources of information to the missions. Traversal between these targets –points-of-interest– is performed by a navigation system. This navigation system provides the mechanisms to move the rover on the Moon surface between two or more points. The current process includes a team on Earth selecting the region of interest that the rover should visit. This process is supported by several sources of information such as local images, information from the Lunar Reconnaissance Orbiter, and rover sensors.

Points-of-Interest Selection Since our work was modeled on prospection tasks, and more specifically the Resource Prospector mission, points-of-interest (POI) for

a Lunar operation consisted of locating the presence of water and predicting the thermal stability of ice with respect to the depth of the Lunar surface. According to Colaprete (2018), these *Resource Target Regions* (RTRs) are defined as four regions: *dry regions* where temperatures are too warm for ice to be stable, *deep regions* where a stable layer of ice is expected between 50-100 *cm* from the surface, *shallow regions* where a stable layer of ice is expected 50 *cm* from the surface, and *surface regions* where surface ice is expected, which is typically in a PSR location.

Opportunistic Path Planning Current approaches for solving these challenges are based on two main methods (Carsten *et al.*, 2007): *blind drive*, where the Earth operator defines the robot path and the rover follows that route with no identification of hazardous conditions or new points of interest; and *autonomous navigation*, where the rover identifies local threats such as non-traversable rocks or shadowed regions surrounding them on its path to the target location. In the case of Lunar prospecting, opportunistic path planning for multiple agents consists of selecting points-of-interest that will yield the greatest amount of scientific return while maintaining constant communication between the agents. Attempting to solve this requires an extensive state-action space due to discretizing states into number of agents, number of points-of-interest, and path proposals.

One can reduce this space by splitting the problem into two parts. The first selects and manages the relation of the multiple agents and the points-of-interest by using a brute-force traveling salesman solver as specified by Gutin and Punnen (2006). An A^* algorithm is then used to establish an approximated best traversable route between 2 or n points-of-interest. The second part takes the approximated paths for each agent and uses a version of the Distributed Path Consensus (DPC) algorithm (Bhattacharya *et al.*, 2010a) to augment the traversal paths for maintained communication between

the agents.

Multi-Agent Strategy

As briefly mentioned in the Communication Map section, a multi-agent system can significantly expand the capabilities of a Lunar prospecting and exploration mission, yet this paradigm comes with challenges (Colby *et al.*, 2016). Agents in this setting lack full autonomy and are remotely controlled by human operators on Earth. Increasing the number of agents without any additional autonomy concurrently increases human operator burden. The work presented here provides a new multi-agent automation strategy to increase the viability of future multi-agent missions.

For a prospecting mission, we examined cooperative strategies that provided extended, reliable communication on the Lunar surface because agents may enter RTRs that limit communication. The two possible agent configurations are described in the following subsections. The configurations proposed in this research are based on three different agents with different physical characteristics and roles:

- **Lander spacecraft**: This agent serves multiple roles during a Lunar mission. At the beginning of the mission, the Lander acts as the payload module containing the rovers for arrival on the Lunar surface. The lander is strategically placed on higher elevation to allow consistent, unfettered communication with Earth as its second role.

- **Scientific rover**: A robotic rover capable of every possible mission defined scientific experiment. It communicates directly with Earth, or by intermediaries such as a Relay rover or a Lander spacecraft.

- **Relay rover**: Unlike the Scientific rover, this rover is primarily tasked with assisting other rovers maintain communication with Earth. These rovers are

less capable of doing thorough scientific experiments, but may be equipped with some basic experiment support. Some descriptions have the relay rover solely act as a communication relay and have no scientific capabilities.

Single-class Configuration A lander spacecraft on the Lunar surface deploys two Scientific rovers of the same type. Both rovers have the same capabilities, but are more conservative in their task allocation so as to maintain communication to the lander. If a rover must enter an area with little possibility of communication with the lander (such as a crater or a low elevation PSR), the other scientific rover may act as a relay to maintain communication.

Multi-class Configuration A lander spacecraft on the Lunar surface deploys two rovers: a scientific and a relay rover. The lander acts as a relay to communication back to Earth. The scientific rover undergoes the necessary scientific tasks for the mission while the relay rover maintains a communication connection between the scientific rover and the lander. Additionally, the relay rover may provide limited secondary scientific task capabilities.

MARMOT

The Multi-Agent Resource Mission Operations Tool (MARMOT) was developed to help create better automation solutions to multi-agent problems. Specifically, MARMOT gives users the ability to define an optimized set of trajectories to enhance mission performance for exploration and/or prospecting tasks.

Pipeline From a macro-abstract view, MARMOT acts as a simple tool capable of easy integration in existing backend systems. Building a mission pipeline utilizing MARMOT begins with by providing databases (maps and points-of-interests related

to the mission), mission constraints, and operator preferences as detailed in Figure 7.7. The latter two are user-defined and contain required mission information to further tune and modify the generated cost map. For example, mission constraints such as the mission-operation time window (date and length), which will be used to generate the relevant time-dependent version of the shadow maps.

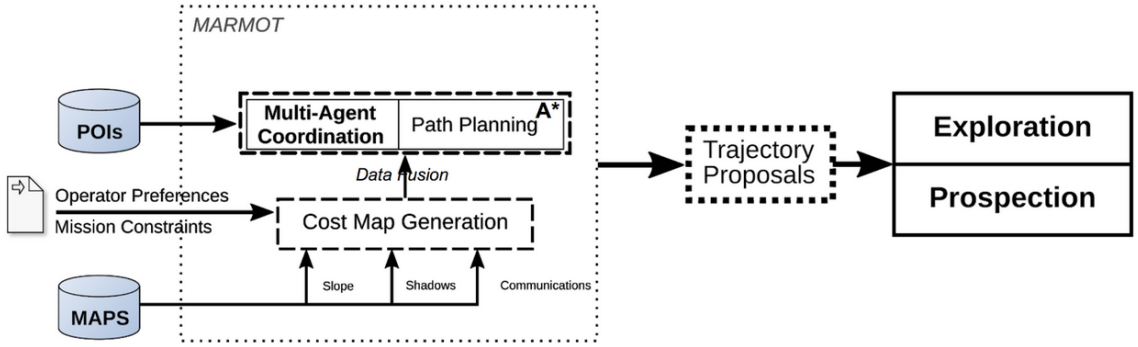


Figure 7.7: Layout of data pipeline and algorithm structure.

Once the data of each map is layered and fused to create the overall terrain cost map, MARMOT selects POIs to integrate into this map. Finally, MARMOT uses the master cost map to generate the multi-agent trajectories by deploying built-in path planning algorithms.

Software Package Description MARMOT is a software deployment that aims to help human operators. It consists of three stand-alone support packages, Graphery, Mapstery, and Utils. The first two packages were built separately from MARMOT to act as dependencies and for utilization in other projects. Graphery performs general graph-theoretic network graphing, while Mapstery generates cost maps. The Utils component of MARMOT is not a separate package, but it comprises a set of generic tools for manipulating general data that companies and researchers may find useful.

Our aim in creating these packages tools was broad generalizability and extensibility with other open-source projects, such as QGIS (Team, 2002), who would like to

employ similar approaches for multi-agent missions. Additionally, MARMOT is built from many open source libraries such as *gdal* from OSGeo (Warmerdam, 2008), *NetworkX* (Hagberg *et al.*, 2008), *OpenCV* (Bradski and Kaehler, 2000), and *NumPy* (Walt *et al.*, 2011). Figure 7.8 provides a basic overview of MARMOT's software structure.

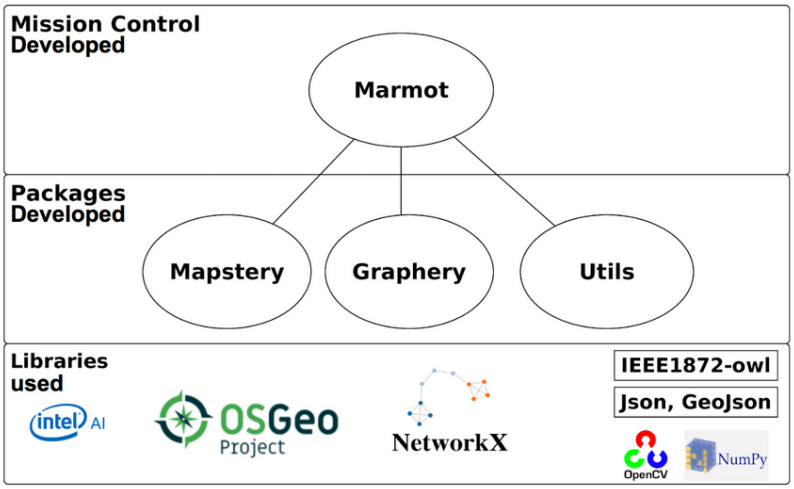


Figure 7.8: MARMOT presents three main components Mapstery: Map fusion and cost map generation, Graphery: Graph generation and path planning, Utils: general purpose tools for data formalization and manipulation

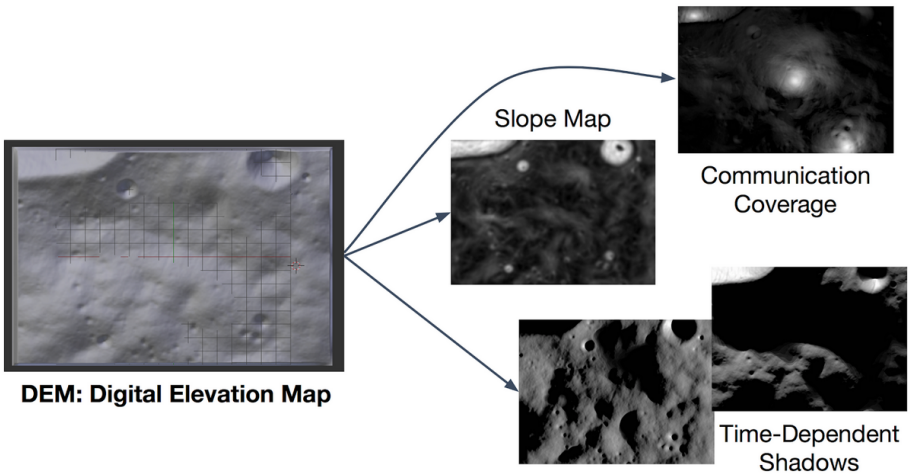


Figure 7.9: Using a Digital Elevation Model (DEM), viewed in Blender on the left, multiple cost maps can be generated.

Mapstery Figure 7.9 presents an illustration of how a DEM is used to make the slope, shadow, and communication maps. Before the layered cost map is generated, mission objectives or preferences may require certain maps to take preference over other maps. The user weighing the information provided by each map can result in different path planning strategies. For instance, if shadowed regions are to be avoided at all cost, a higher weight is associated with the shadow map relative to the communication and slope maps. The layered and modified cost map is thus visualized as a linear combination of individual maps,

$$CostMap = \alpha \cdot M_{shadows} + \beta \cdot M_{slope} + \gamma \cdot M_{comm} \tag{7.1}$$

where α, β, and γ are the user-defined weights, and $M_{shadows}$, M_{slope}, and M_{comm} are the respective map associated to each weight.

Graphery Creation of grid-graphs are done primarily through the Graphery Python package. Given a requisite map size, Graphery will generate the appropriate size blank grid-graph for use with the layered cost map provided by Mapstery. The graph allows users to modify or create certain attributes (or costs) on-the-fly.

Other Utilities Additionally, utility functions were created to ease MARMOT's use in existing codebases. Waypoints, tasks, and other mission primitives are loaded using JSON format files. Visualization tools were developed for users who wish to visualize the changes in non-web frontend formats using matplotlib Hunter (2007).

7.2.4 Coordination and Trajectory Proposals

Point-of-Interest Selection For a given Lunar mission, numerous points-of-interest exist for exploration; however, very limited operation time by rovers make assessing each of these points intractable. With MARMOT, we develop a means of generating

a visit order of waypoints for a subsets of POIs that are explorable given mission constraints. The geographical coordinates of a POIs are input into a list of coordinate-lists, where POIs of equal visit importance reside together. Each coordinate-list constitutes a strata of importance for the mission. The algorithm then selects one coordinate from each coordinate-list to generate a rover's path, up to N user-defined points.

Additionally, the algorithm optimizes the selection based on a distance function. The more POIs the rover can assess in a shorter distance, the better the path is considered. Numerous permutations of the paths are provided to enable the user to make the final selection. Additionally, variations of this algorithm were provided to allow start or end points, or use of different distance functions i.e. Euclidean distance or A^* search using the fused cost-maps mentioned earlier.

Multi-agent Optimal Path Planning and Synchronization MARMOT utilizes standard graph search algorithms such as A^* search to effectively generate path trajectories over the terrain for each agent. However, the system's capabilities were extended to use more modern search algorithms to effectively produce optimal trajectories while still respecting mission constraints. Distributed Path Consensus (DPC) (Bhattacharya *et al.*, 2010a) and its supplementary work, Distributed Path Consensus with Tasks (DPCT) (Bhattacharya *et al.*, 2010b), are modified forms of A^* search that provide the ability to optimize each agent's search path by iteratively applying soft constraints on their individual paths while satisfying a user-defined multi-agent constraint.

In particular, this user-defined constraint is a cost function added to the A^* algorithm's heuristic function. The cost function produces a corrective penalty or reward to each iteration of this modified A^* search. Moreover, an incremental weight is ap-

plied to the cost function to improve its efficacy after each subsequent path search. In the Lunar prospecting scenario, maintaining communication between rovers is crucial to a successful mission. One can define an acceptable distance that rovers must maintain during traversal, and each iteration of the algorithm will result in a path which maintains the distance constraint. Figure 7.10 shows the first and last iteration for two agents using the DPC algorithm.

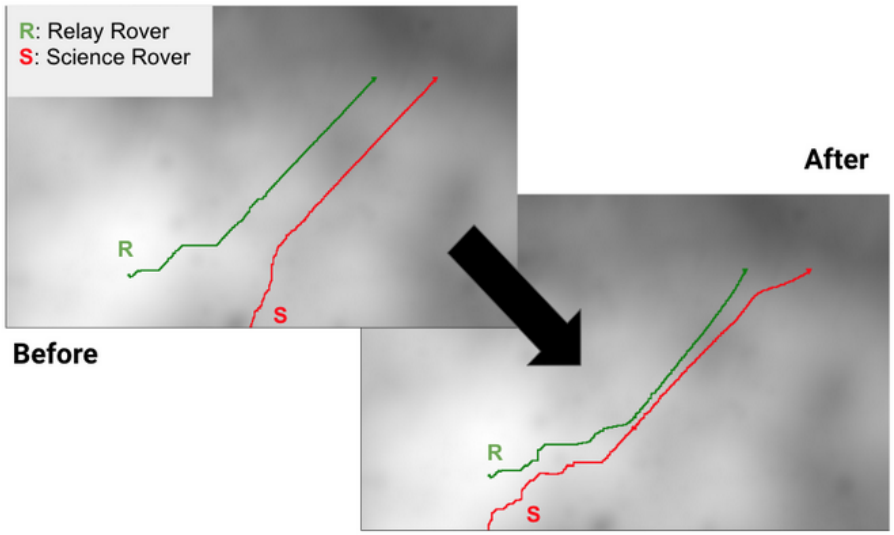


Figure 7.10: DPCT Applied to Static and Dynamic Agents. Applying soft penalty constraints allow path generation/modification that maintain constant communication between a science and relay rover as they proceed to different tasks.

Simulation Study

To illustrate the capabilities of MARMOT, an initial mission scenario was simulated using a three-agent configuration consisting of a lander, a relay rover, and a scientific rover. Operators at mission control direct the science rover to initiate a prospecting task within a PSR. Moreover, this region happens to be a within a crater, making communication to the lander impossible. The relay rover will thus act as the intermediary to establish proper communication link between the lander and the scientific

rover.

Figure 7.11 presents the above example with trajectories generated by MARMOT. In this case, the overall terrain cost map favored avoiding the shadow regions by placing a higher weight on the individual shadow map before MARMOT fused them with the other maps. The path planning generated by MARMOT is an A^* heuristic search algorithm that avoids shadows while maintaining communication.

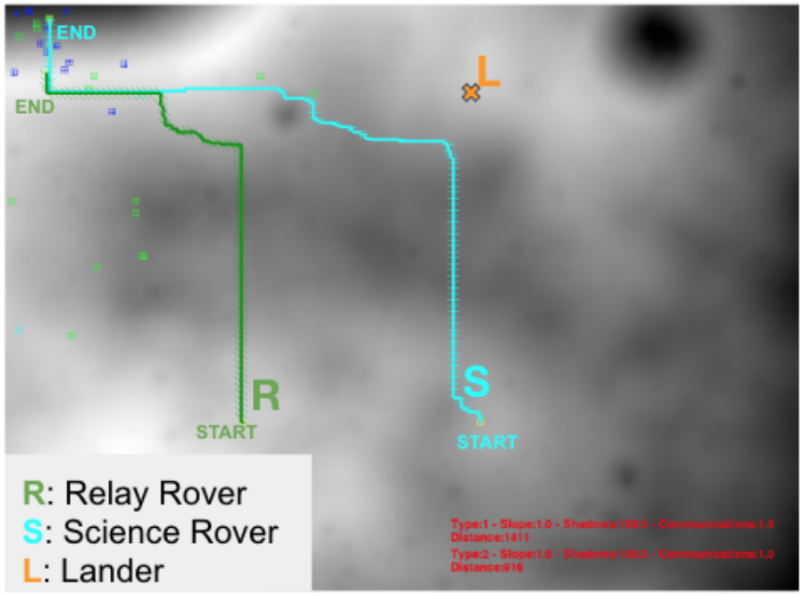


Figure 7.11: Case 1: Relay rover extends lander communication along the crater border while science rover explores a point-of-interest in a crater without communications to the lander. A fused cost map is used for trajectory calculation.

Figure 7.12 presents a slightly modified version of the mission scenario, but less weight is applied to the shadow map. Additionally, the path planning generated by MARMOT is an opportunistic solution. This algorithm adds intermediary points-of-interest for a relay and scientific rover to target as they progress towards their end goal. Once those points-of-interest are chosen, the A^* search algorithm is used to calculate the best path between the waypoints.

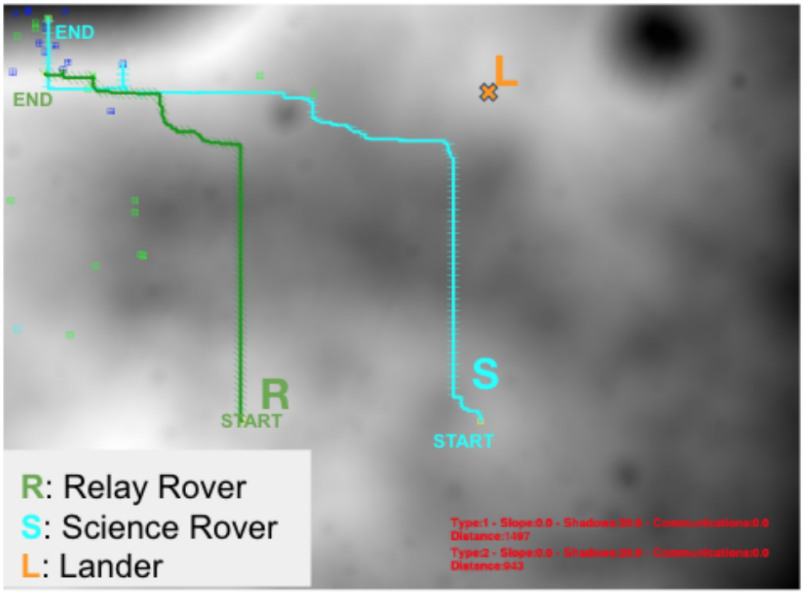


Figure 7.12: Case 2: The rover's goals are the same as in Case 1. A set of traversable points-of-interest are suggested for extending the prospecting mission.

7.2.5 Benefits of a Mean-Field Approach To Lunar Prospecting

One of the drawbacks to the MARMOT is its inability to allow greater relinquishment greater control from a human operator. Communication must be maintained in each of the cases that we explored since the human operator would interact with the regolith to perform ice detection analysis. An interesting avenue of research would be for a system that would allow the human operator to chose certain points on the cost map generated by MARMOT to create a strongly-connected graph, select the number of rovers desired at each site, quickly train a leaderless control policy from a mean-field model, and send the new policies to the rovers. The rovers will then explore the sites-of-interest at the behest of the human operator but will stochastically switch to exploring other sites the human operator selected depending on the reaction rate k defined in Section 6.3.2 and a large time step. Eventually, they will converge to the desired rover distribution at each site desired by the human operator. The individual rovers will still utilize the cost map and the path planning between

sites MARMOT calculates, but now more autonomy is given to agents on when to move from one site to another. This way human operators will only need to observe what the individual agents do from time to time but not require them to constantly control the rover, leading to eventual burnout.

7.3 Towards Decentralized Human-Swarm Interaction by Means of Sequential Hand Gesture Recognition

This section contains results from **Zahi M Kakish** *et al.* (2020).

7.3.1 Overview

In this section, we present an initial design and validation of an approach to human-swarm interaction based on robot recognition of sequential hand gestures. While not a completely decentralized control approach, it demonstrates the ability to leverage recent improvements in the object classification abilities of Convolutional Neural Networks that run on small, low-cost, low-powered computational devices that are typically used on swarm robotic platforms.

7.3.2 Methodology

Our system is composed of two primary components: (1) a deep neural network for classifying various hand gestures, and (2) a simplified multi-robot formation control strategy for a small swarm size. This section will provide a proof-of-concept validation in both simulation and experiment of the utility of sequential hand gestures as a means of controlling a robotic swarm. We do not formulate the approach as a generalized control methodology for a wide range of swarm sizes, which we leave to future work.

Deep Neural Network

Since low-powered, small computationally powerful devices, such as the Raspberry Pi and other *ARM* devices, are common among swarm robotic hardware architecture, most conventional and state-of-the-art neural network models are unusable or difficult to implement on these platforms. To circumvent this issue, a special neural network named SqueezeNet was deployed for hand gesture recognition Iandola *et al.* (2016). The SqueezeNet model is capable of performing as well, or even surpassing, classification rates of many the most popular model types with only a fraction of the parameters, thus reducing the model size to roughly ~ 5 MB. This substantial reduction in model size makes this model ideal for use in hardware typically found in swarm robotics applications.

Dataset and Preprocessing

A training set consisting of six different gestures and $2,956$ images was utilized for training the SqueezeNet model Heintz (2018). The images are silhouettes of the gestures in black and white. A gesture set, G, containing all possible gestures for classification and available in the training data is defined as:

$$G = \{\ C,\ Fist,\ L,\ Ok,\ Peace,\ Palm\ \} \tag{7.2}$$

Figure 7.13 contains examples of the gestures found in the dataset. The dataset was expanded to ensure more robust classification of gestures during real-time operation. The data augmentation began with inverting the images horizontally to resemble the gesture but with the opposite hand. The original image and the inverted image are used to generate three new images, respectively. One rotated by 90° clockwise, another 90° counter-clockwise, and an image flipped upside-down. Therefore, seven

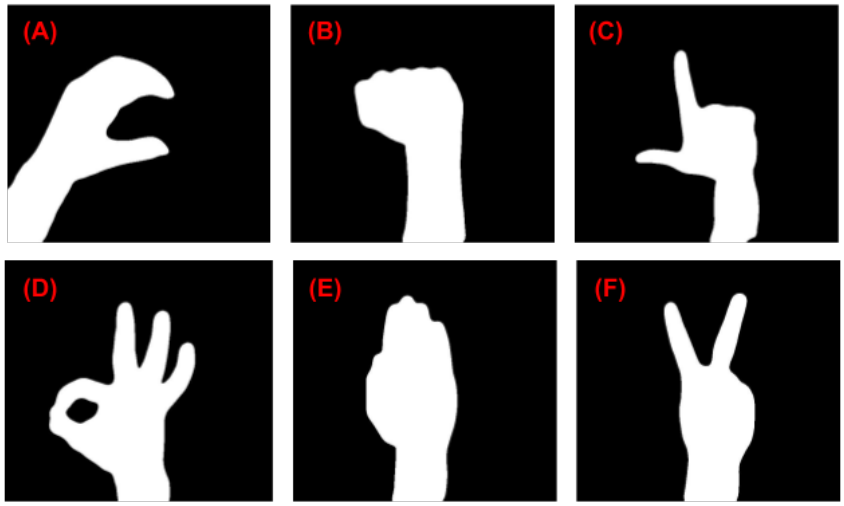


Figure 7.13: Silhouettes of gestures available through the dataset in Heintz (2018). In addition to these gestures, a blank image was used to classify the gesture *None*. A) a *C* shape B) a *Fist* C) an *L* shape D) the *Okay* sign E) a *Palm* F) the *Peace* sign.

new images are generated for each image in the original dataset.

To ensure that the model is capable of real-time performance, a seventh gesture was added to the dataset: 'None'. Since the user's hand is not consistently in the frame, the image is classified as 'None' and awaits a gesture to come into the frame. A dataset containing blank images is generated during preprocessing consisting of black images and black images containing Gaussian noise to add to SqueezeNet model's classification capabilities.

Further preprocessing was performed before training to ensure optimal results. Images in the dataset were cropped from their original size of 640×576 to 570×570. The images were then scaled to 240×240. This final image size is big enough to ensure adequate real-time classification of gestures without increasing computational cost.

Training

The SqueezeNet model was programmed and trained using the *TensorFlow* API version 1.14 Abadi *et al.* (2015) running on a development box containing four *Nvidia* RTX 2080 graphics cards. The model was trained for 10 epochs with a Stochastic Gradient Descent optimizer, which had a learning rate $\alpha = 0.001$, a coefficient of momentum $\eta = 0.9$, and a learning rate decay $\lambda = 0.0002$. The batch size was set to two images to prevent memory issues due to the images' size. The dataset was split into a training and validation set consisting of 90% and 10% of the images, respectively.

Gesture-Driven Controller

We first consider a waypoint system in which a finite number of robots, N, plot a straight motion to a desired goal position $(x_d, y_d)_N$ from an initial starting position $(x_i, y_i)_N$. To help validate the applicability of sequential gesture control, we will only focus on one direction, x, and keep the y values arbitrary. For example, if the initial position of a robot is $(0, 0)$ and the desired position is set to be $(10, 0)$, the desired position is considered achieved if only the robot's x-coordinate matches the desired value. A set of waypoints, $\mathcal{W}$, for each robot n is thus generated from augmenting x_i by x_{offset}, which is calculated from a user-defined number of waypoints, M.

$$x_{\text{offset}} = \frac{|x_i - x_d|}{M} \tag{7.3}$$

$$\mathcal{W}_n = \{(x_i + k \times x_{\text{offset}}, y) : k = 0, \ldots, M\} \tag{7.4}$$

Each robot, n, moves from one waypoint to another in the set $\mathcal{W}$ until it reaches the final desired position $(x_d, y_d)_n$.

With the waypoints generated, we define $\mathcal{C} : y_n \times \beta \to \mathbb{R}$ to be the cohesion of the swarm, that is, a metric measuring the inter-robot distance of a swarm, where y_n is the robot n's y position and β is the cohesion factor. β is a binary variable that

represents an increase in cohesion if $\beta = 0$ or a decrease in cohesion if $\beta = 1$. To increase the cohesion of a swarm is to reduce the distance between each robot, and to decrease the cohesion is to increase the distance between each robot. Additionally, we consider a small swarm size of three agents in a line separated by an offset in the y direction. The cohesion of this small swarm is best represented by a constant addition or subtraction of a small, static offset y_{offset}. An individual robot, n, applies a cohesion change by the following piecewise formula:

$$\mathcal{C}(y_n, \beta) = \begin{cases} y_n - y_{\text{offset}} & \text{if } y_n > 0 \text{ and } \beta = 0 \\ y_n + y_{\text{offset}} & \text{if } y_n > 0 \text{ and } \beta = 1 \\ y_n + y_{\text{offset}} & \text{if } y_n < 0 \text{ and } \beta = 0 \\ y_n - y_{\text{offset}} & \text{if } y_n < 0 \text{ and } \beta = 1 \end{cases} \tag{7.5}$$

Next, we consider a differential drive capable of driving from one waypoint to the next based on the unicycle model Carona *et al.* (2008) as defined by the following state-space formulation, where x and y are the robot's position and ϕ is its angle.

$$\dot{x} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} v_o \cos\phi \\ v_o \sin\phi \\ \omega \end{bmatrix} \tag{7.6}$$

Here, v_o and ω is the robot's linear and angular velocity, respectively. Since we choose to apply a constant linear velocity, the dynamics of the robot using the unicycle model are represented primarily by ω. To control the robot's motion from one point to another reference point, the difference between a desired angle, ϕ_d, and the robot's current angle, ϕ, is calculated and used as the error

$$e = \phi_d - \phi. \tag{7.7}$$

To prevent singularities from arising and restrict the error between 0 and 2π, we update the calculated error using *atan2*:

$$e_{new} = atan2(\sin(e), \cos(e)). \tag{7.8}$$

Finally, our control input into the robot is applied with a proportional gain K_p:

$$\omega = K_p \dot{e}_{new} \tag{7.9}$$

This forms the basis for the control of the swarm using sequential hand gestures.

7.3.3 Simulations

A series of 3D simulations were designed to test the viability of sequential gesture-based control of a swarm. To formalize the experimental simulations, the following assumptions were made. First, the simulations are limited to three robots. Though this is a small number of robots, we believe that the scenario provides a viable minimal benchmark for how our algorithm will actually work. Second, odometry is provided by the simulation environment and not by internal (i.e. encoders) or external (i.e. GPS, cameras, etc.) sensors. Finally, a black background is used when reading images from the gestures to streamline classifications.

Structure

The simulation was developed using *ros2*, the next generation of the Robot Operating System (ROS). Compared to the original ROS, ros2 provides an enhanced middleware programming environment, the removal of a ROS Master to make multi-robot decentralized approaches easier, and expanded platform and architecture support. *Gazebo*, a 3D robot simulation environment, is used to render the experiment. Gazebo is a project developed in tandem with ROS/ros2, and comes with many ROS drivers to

simplify development of robots by providing a simulation environment that functions similarly to a physical one. The experiment is run using the Turtlebot3 Burger robot by ROBOTIS (2019) who provide 3D Gazebo models for use in simulation.

Each robot runs two separate nodes: one that contains drivers to connect to Gazebo to simulate differential drive control, sensor readings, etc., and another that is the primary motion controller as defined in Section 7.3.2. The robots do not intercommunicate and only subscribe to messages from one external node that tells them what gestures were classified. This external node reads information from a generic webcam running the SqueezeNet model. Additionally, images captured by the webcam are modified to reflect those used for training, which were silhouette images of the different hand gestures.

The captured camera images are cropped from the 640×480 resolution to 480×480 resolution and then scaled to 240×240, which is the input size for our SqueezeNet model. The images are then converted to greyscale and a slight Gaussian blur is applied to prevent fine details in the image from causing artifacts when converted to a binary image. Finally, the image is converted using a binary filter and inputted into the SqueezeNet model. The predicted gesture is then published to all the robots. This message is not unique to each robot.

Gestures

For this experiment, the number of gestures used have been reduced to the following five:

$$G_{possible} = \{Palm,\ Peace,\ Fist,\ C,\ L\} \tag{7.10}$$

These gestures are mapped to the subsequent actions the swarm may undergo:

- **Palm**: Stop movement of the swarm

- **Peace**: Resume movement of the swarm
- **Fist**: Read cohesion action
- **C Sign**: Increase
- **L Sign**: Decrease

Additionally, there is a sixth classified gesture is *None*, which, as stated in Section 7.3.2, means that there is no gesture recognized and no swarm action applicable. *Palm* (Stop) and *Peace* (Resume) are the only two gestures capable of controlling the swarm alone. The rest require the user to give a sequence of gestures for the swarm to read. One a gesture pertaining to the swarm behavior (cohesion) the user wishes to modify and the next is a gesture mapped to a modification variable (increase or decrease).

In the simulation, we will rely on two sequences used to modify the cohesion of the swarm to help negotiate an obstacle. The first is increasing the cohesion of the swarm, $\beta = 0$, which means that the swarm will group closer to one another. This is done by giving the following commands in this order:

$$Palm \rightarrow Fist \rightarrow C \rightarrow Peace \tag{7.11}$$

This sequence is explained as *"stop the swarm, read my cohesion command, increase cohesion by one step size, and resume moving."* The second would be to decrease the cohesion of the swarm, $\beta = 1$, resulting in a increase in distance between the robots. This is done using the same hand gestures but with the decrease cohesion command.

$$Palm \rightarrow Fist \rightarrow L \rightarrow Peace \tag{7.12}$$

Just like the increase cohesion sequence, this sequence is similarly explained as *"stop the swarm, read my cohesion command, decrease cohesion by one step size,*

and resume moving." As described earlier, the cohesion of the swarm is set as steps. Specifically, each call to increase or decrease the swarm cohesion decreases or increases the distance between the robots by a calculated y_{offset}, respectively. If an obstacle requires the user to change the swarms cohesion by multiple steps of y_{offset}, the user does not need to repeat the whole sequence twice but can concatenate the swarm command multiple times. For example, if the swarm cohesion is needed to increase by two steps, a user would provide the following command:

$$Palm \rightarrow Fist \rightarrow C \rightarrow Fist \rightarrow C \rightarrow Peace \tag{7.13}$$

Simulated Environment The three robots are placed in a line within three simulated testbeds, shown in Figure 7.14. The first contains a series of two types of openings: one small-sized opening in the middle and two intermediate-sized openings located on both ends of the testbed. The second testbed has only one small-sized opening. Each testbed provides a different validation for the capability of our sequential gesture control scheme. The first demonstrates how individual commands can be given at the onset of an obstacle, and the other shows how for more difficult obstacles a user is capable of stringing together multiple gesture actions into one command. The first two testbeds are $8m \times 4m$ in size. Each robot is placed $1m$ from the end of the testbed and spread to have an inter-robot distance of $1.5m$ between one another. To complete each task, the robots will have to move forward and reach the other end of the testbed. The swarm of robots will need to negotiate the obstacles before them by relying on a user's sequential gesture input.

The third testbed is a recreation of a real testbed used in the physical experiment section of the paper. Compared to the large surface area of the other testbeds, this testbed is significantly smaller at $2.5m \times 2.5m$. Additionally, the initial inter-robot distance is reduced to $0.6m$ and the robots are placed $0.5m$ from the end. The testbed

contains a $1m$ sized opening a meter from the robots starting position. Like the other testbeds, the robots will have to negotiate this obstacle using a single sequence of gestures to increase cohesion. After going through the obstacle, a different sequence is used to decrease the cohesion. This testbed's results act as validation for the success of this experiment using real robots. Figure 7.15 provides a brief overview of the robots increasing and then decreasing their cohesion to negotiate a small opening.

Physical Experiments

To validate the use of sequential hand gesture control of a swarm in a physical setting, an analogous physical testbed to the third simulated testbed was built as shown in Figure 7.16. The experiment was set up in the same manner as the simulated one. The individual robot controller and gesture recognition code was written with the intention of interchangeability between the simulated and physical experiments. The only difference were parameter files which provided environmental constraints and individual robot attributes depending on the environment in which the test is being run.

However, certain aspects of the simulation were not available for use in our physical experiments. Odometry within simulation is accurately calculated by the Gazebo environment, but getting this same information in a physical experiment required use of an overhead camera and *ArUco* fiducial markers (Romero Ramirez *et al.*, 2018; Garrido-Jurado *et al.*, 2015) to calculate each individual robot's pose. A ros2 node was developed to track individual robot position from the overhead camera and calculate poses from a $0.127m \times 0.127m$ marker placed atop each robot that was detected. This odometry information is then published to the robots for use in their controller node's.

During simulation, all the ros2 individual robot nodes and gesture node were run

on the same computer. The physical experiment distributes the computation over a wireless network. Each robot runs its respective controller on its hardware, but does not communicate with other robots. The only communication that the robots receive is from two external sources: the node calculating individual robot odometry from a central overhead camera and the node reading and classifying the sequential gestures from the user. Each of these nodes are also run on separate computers due to convenience rather than the inability of running both on the same one. This decentralization comes with a cost, however, in the form an approximately $0.5s$ delay. Even though the individual robots were capable of running the gesture detection node, we chose to keep the node running on a separate computer to keep the test runs between the simulation and physical experiments similar.

The physical experimental procedure is nearly identical to the simulation except for the software structure changes presented in this section. There are a few small changes in comparison to the simulation; however, we do not believe it reduces the validity of our physical experiment. One additional change to this experiment was the linear velocity of each robot was reduced due to the half-second network lag in the system.

Discussion

We have successfully demonstrated our hypothesis by showcasing a simplified cohesion control model for both simulated and physical testbeds. Each simulated test completed successfully and the controller responded correctly to the properly classified sequential gesture commands given from the SqueezeNet model in real-time. As mentioned in Section 7.3.3, the run corresponding to Testbed 2 demonstrated the ability for the systems to read multiple instances of the same *increase cohesion* gesture sequence in one input. Results from that run show that the provided input

sequence was easily registered and enacted by the robots. Additionally, the physical experiment was able to finish successfully even with the network delay present in the system. We believe that re-creations of bigger testbeds, such as Testbed 1 and 2 in our simulations, would yield successful runs. Although the system is semi-decentralized due to odometry calculations and having the gesture recognition node running separate from the robots, these results of these tests show the feasibility of a human operator interacting with a decentralized robot swarm by showing a robot a sequence of hand gestures.

Although all the experiments were successful, we did run into a minor classification issue during test runs. The SqueezeNet CNN would sporadically misclassify the hand gesture upon the subject's hand leaving the camera's viewing area or when switching between hand gestures. We believe that this issue is likely due to gestures created during the hand's motion unable to be classified. To help reduce this error, publishing of the predicted gesture was limited to once every half second instead of once every tenth of a second, which is the refresh rate of the ros2 node that classified the gestures.

7.3.4 Benefits of a Mean-Field Approach To Decentralized Human-Swarm Interaction

This section provided preliminary work into a system that uses a sequence of hand gestures to control swarm behavior through a combination of a small-sized CNN model capable of recognizing silhouette images of hand gestures in real-time and a decentralized robot development environment. However, scale may still prove this solution intractable for a human-in-the-loop situation where an operator needs to delegate differentiating tasks to a single robot or a group of robots. Current methods such as the work done by Nagi *et al.* (2014) required the human to point to a specific robot in order to issue commands. Instead, a group of robots already equipped with

a finite set of tasks can be issued them through gestures followed by the desired number of agents for each task. That agent will then distribute the information to the remainder of the robots after calculating the policy. The simplicity of training our leaderless control policy on a consumer grade desktop shows that potential of effectively training a more optimal policy on device.

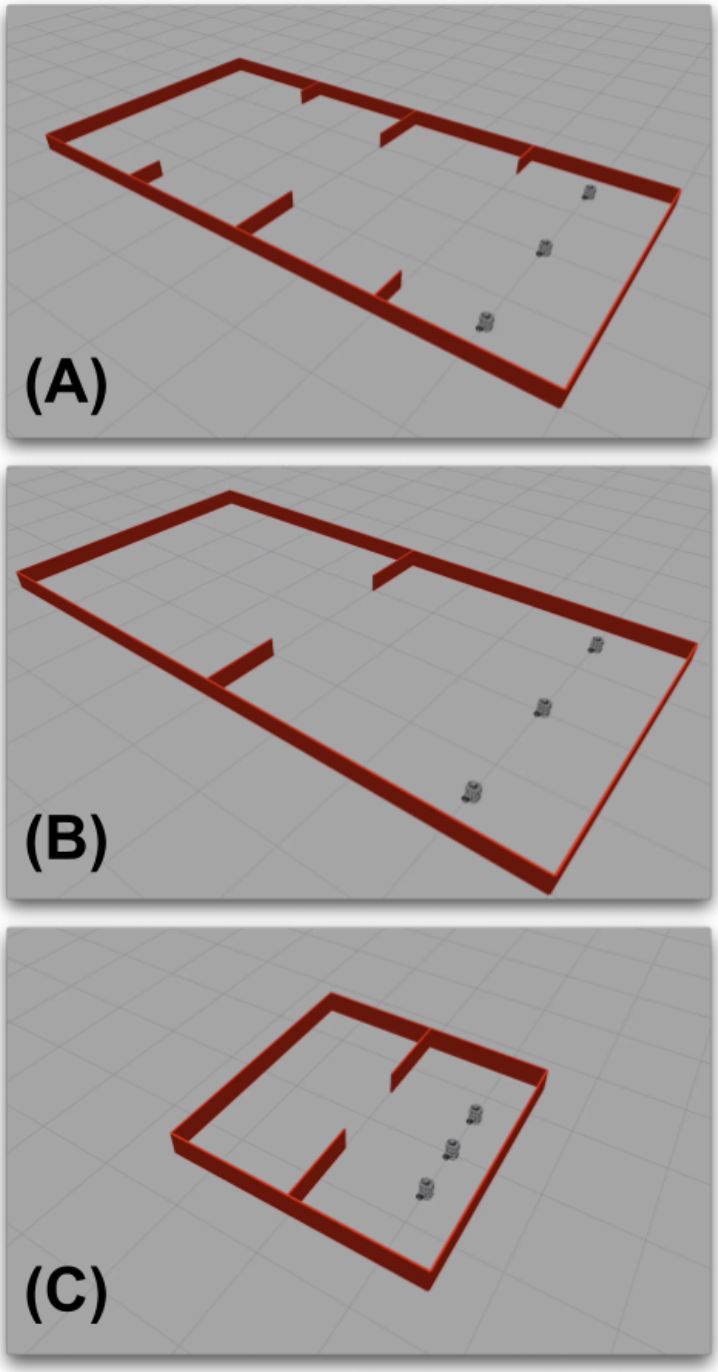


Figure 7.14: The three testbeds created for simulation. A) An $8m \times 4m$ testbed containing multiple openings for the swarm to traverse through. B) An $8m \times 4m$ testbed containing only one small opening for the agents to negotiate. Compared to the previous testbed, this requires the user to string together multiple sequences of the same gesture to complete. C) A small $2.5m \times 2.5m$ testbed with one small $1m$ wide opening. This last testbed is recreated for physical validation in Section 7.3.3

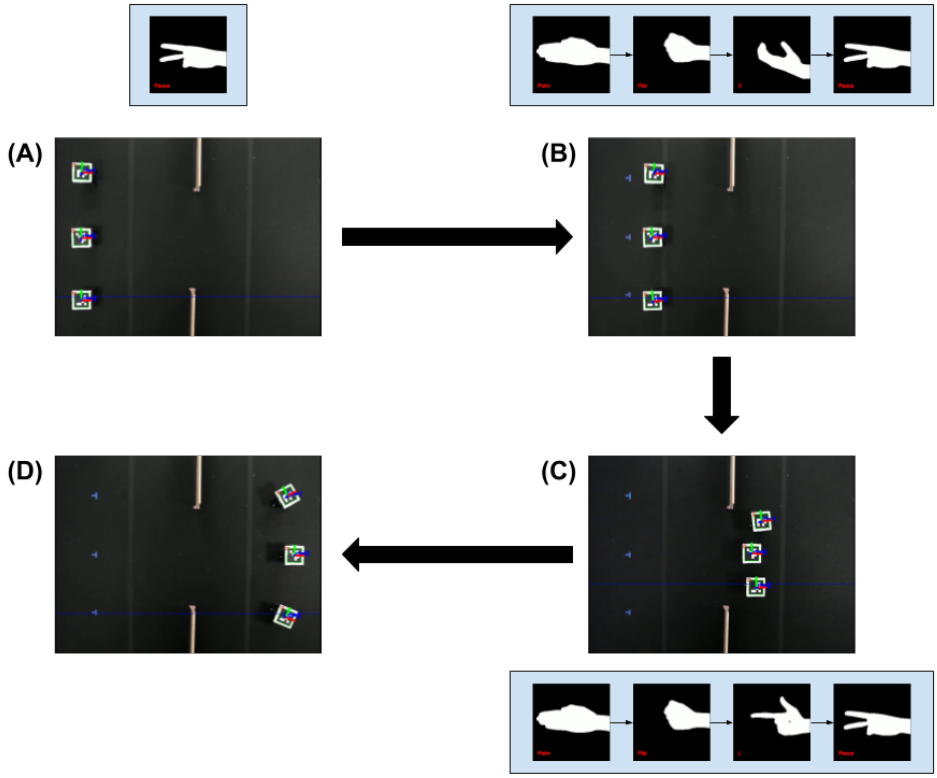


Figure 7.15: An overview of the physical experiment with gesture sequences required for the swarm to get through the small opening in the middle of the arena. A) The robot swarm begins on one end of the testbed. The *Peace* gesture is used as a standalone command to start the experiment. B) Once the robots get closer to the obstacle, the following sequence ($Palm \rightarrow Fist \rightarrow C \rightarrow Peace$) is given to increase the cohesion of the swarm and resume motion. C) After the robots have cleared the opening, another gesture sequence ($Palm \rightarrow Fist \rightarrow L \rightarrow Peace$) is used to return the swarm back to their initial cohesion size. D) Finally, the robots reach the final waypoint and the experiment completes.

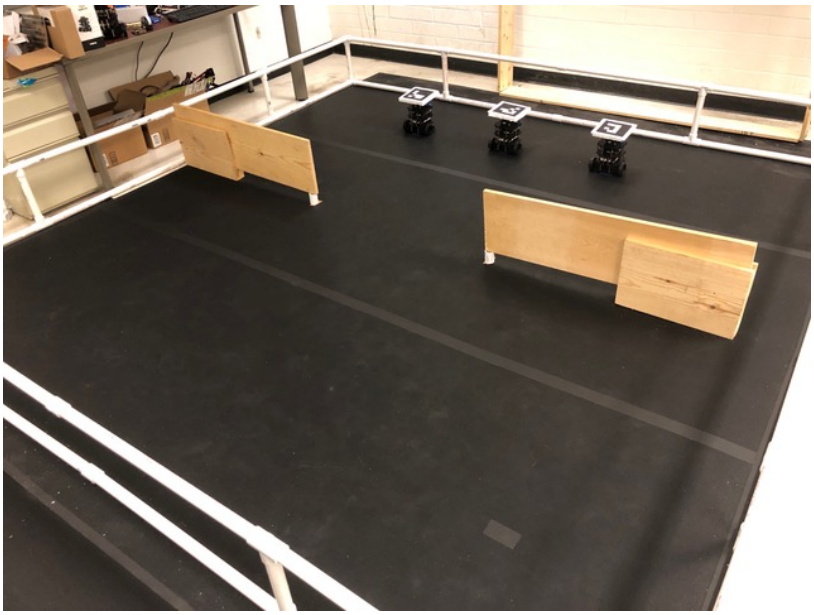


Figure 7.16: The physical testbed.

Chapter 8

CONCLUSION AND FUTURE WORK

8.1 Conclusion

In this dissertation, we explored the design and use of leader-based and leaderless deep RL control policies trained using mean-field models to efficiently and robustly redistribute a swarm of robots among a set of states. We provided empirical evidence to support the use of a macroscopic, model-based approach for training the control policies by experimentally showing the existence of a "mean-field" effect, a phenomenon whereby the populations of agents that redistribute stochastically over a graph evolve in a more deterministic manner as the agent population tends towards infinity, through training a tabular Temporal Difference control policy on different populations of agents that follow a discrete-time Markov chain. Using these results, we develop a method of training a fully and locally observable deep RL leader control policy on a modification of the Kolmogorov forward equation that models the effect of a leader stochastically repelling agents on a macroscopic level. In Chapter 5, we considered the scalability of our deep RL approach in terms of graph size through a change in the state and action sets pertaining to the control policy. Finally, we investigated a leaderless control policy in which the agents' states evolve according to a discrete-time Markov chain, and their populations in each state evolve according to a linear ODE mean-field model. In this approach, the learned parameters are the agents' transition rates between vertices of a bidirected graph, which are parameters of the linear ODE.

8.2 Future Research

The limitations arising from our original action set restrict our ability to train larger graph sizes and places an upper bound on the applicability of an actor-critic approach. However, modifying the problem with the new action set lessened the scalability issue. This is not a complete solution since the new action set requires the leader to transition throughout the grid-graph as a fully-connected graph instead of a strongly-connected graph like the follower agents. One interesting avenue to explore is the applicability of our approach to graph shapes that are not grid-graphs. By using different graph shapes, we can decompose a larger grid-graphs into smaller graphs for training control policies. Advances in Graph Neural Networks (GNNs) (Kipf and Welling, 2017; Zhou *et al.*, 2019; Chen *et al.*, 2020) can yield interesting ways of framing our control problem to irregularly shaped graph sizes to more easily break down larger graphs into smaller optimization problems for use with our deep RL mean-field model approach.

Another avenue is to continue using the new fully connected leader-control policy defined in Chapter 5 while combining it with a path planner like the one in MARMOT shown in Section 7.2. This will assist in finding the shortest possible path between the leader's current state and the next through the use of two separate control policies. This may lead to more optimal and efficient solutions for complex population densities and larger grid-graph sizes while maintaining a strongly-connected leader motion constraint.

Our approach in Section 6.3.1 to designing controllers for swarms described by a linear ODE model can be extended to swarms that are described by multi-affine mean-field models, which represent the population dynamics of chemical reaction networks (CRNs), or systems that behave like them (Matthey *et al.*, 2009; Berman

and Kumar, 2009), that include bimolecular reactions. The linear ODE mean-field model represents the population dynamics of a CRN with only unimolecular reactions, such as

$$\mathrm{X}_0 \xrightarrow{\mathrm{k}_3} \mathrm{X}_1 \quad \mathrm{X}_0 \xrightarrow{\mathrm{k}_2} \mathrm{X}_2, \tag{8.1}$$

which corresponds to robot transitions between vertices connected by the two edges $\{(0,1),(0,2)\} \in \mathcal{E}$ for the graphs defined in previous chapters. A multi-affine model enables us to describe the dynamics of swarms that behave according to CRNs with bimolecular reactions, where robots interact with each other or with objects in the environment to become another "species," or enter another state. In Matthey *et al.* (2009), stochastic robot control policies for a component assembly task were designed using a multi-affine mean-field model. For example, the forward reaction in the following bimolecular reaction,

$$\mathrm{X}_0 + \mathrm{X}_1 \underset{\mathrm{k}_1}{\overset{\mathrm{k}_0}{\leftrightarrow}} \mathrm{X}_2 \tag{8.2}$$

represents a robot X_0 interacting with a component of type X_1. When the robot is near the component, the reaction rate constant k_0 determines whether the robot will pick up the component and transition to a new state, X_2. The backward reaction models a disassembly action in which the robot drops the component with a probability rate determined by the rate constant k_1, thereby reverting back to its original state. Using a deep RL mean-field model approach, it is possible to design optimal reaction rate constants that drive the swarm to specified macroscopic outcomes, using the multi-affine model to generate predictions of the collective effect of the agent-level interactions.

Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems", URL `https://www.tensorflow.org/`, software available from tensorflow.org (2015).

Aguero, C., N. Koenig, I. Chen, H. Boyer, S. Peters, J. Hsu, B. Gerkey, S. Paepcke, J. Rivero, J. Manzo, E. Krotkov and G. Pratt, "Inside the virtual robotics challenge: Simulating real-time robotic disaster response", Automation Science and Engineering, IEEE Transactions on **12**, 2, 494–506 (2015).

Amiranashvili, A., A. Dosovitskiy, V. Koltun and T. Brox, "Td or not td: Analyzing the role of temporal differencing in deep reinforcement learning", (2018).

Andrews, D. R., "Resource prospector (rp)-early prototyping and development", in "AIAA SPACE 2015 Conference and Exposition", p. 4460 (2015).

Anschel, O., N. Baram and N. Shimkin, "Averaged-dqn: Variance reduction and stabilization for deep reinforcement learning", (2017).

Babaeizadeh, M., I. Frosio, S. Tyree, J. Clemons and J. Kautz, "Reinforcement learning through asynchronous advantage actor-critic on a gpu", (2017).

Bapna, D., M. Martin and W. Whittaker, "Earth-moon communication from a moving lunar rover", Proceedings of the 42nd International Instrumentation Symposium pp. 5–9 (1996).

Berkelman, P., M. Chen, J. Easudes, J. Hancock, M. C. Martin, A. Mor, E. Rollins, A. Sharf, J. Silberman, T. Warren and D. Bapna, "Design of a day / night lunar rover", Tech. Rep. CMU-RI-TR-95-24, Carnegie Mellon University, Pittsburgh, PA (1995).

Berman, S., Á. Halász, M. A. Hsieh and V. Kumar, "Optimized stochastic policies for task allocation in swarms of robots", IEEE Transactions on Robotics **25**, 4, 927–937 (2009).

Berman, S. and V. Kumar, "Abstractions and algorithms for assembly tasks with large numbers of robots and parts", in "2009 IEEE International Conference on Automation Science and Engineering", pp. 25–28 (IEEE, 2009).

Besancon, R., *The encyclopedia of physics* (Springer Science & Business Media, 2013).

Bhattacharya, S., M. Likhachev and V. Kumar, "Distributed path consensus algorithm", Tech. Rep. MS-CIS-10-07, University of Pennsylvania (2010a).

Bhattacharya, S., M. Likhachev and V. Kumar, "Multi-agent path planning with multiple tasks and distance constraints", IEEE International Conference on Robotics and Automation (ICRA) pp. 953–959 (2010b).

Boumghar, R., S. Lacroix and O. Lefebvre, "An information-driven navigation strategy for autonomous navigation in unknown environments", Safety Security and Rescue Robotics (2011).

Bradski, G. and A. Kaehler, "Opencv", Dr. Dobb's Journal of Software Tools (2000).

Brockman, G., V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang and W. Zaremba, "OpenAI Gym", arXiv preprint arXiv:1606.01540 (2016).

Carona, R., A. P. Aguiar and J. Gaspar, "Control of unicycle type robots tracking, path following and point stabilization", (2008).

Carsten, J., A. Rankin, D. Ferguson and A. Stentz, "Global path planning on board the mars exploration rovers", in "2007 IEEE Aerospace Conference", pp. 1–11 (IEEE, 2007).

Chaumette, F. and S. Hutchinson, "Visual servo control, Part I: Basic approaches", IEEE Robotics & Automation Magazine **13**, 4, 82–90 (2006).

Chen, J. Y. C. and M. J. Barnes, "Human-Agent Teaming for Multirobot Control: A Review of Human Factors Issues", IEEE Transactions on Human-Machine Systems **44**, 1, 13–29, URL `http://ieeexplore.ieee.org/document/6697830/` (2014).

Chen, Z., F. Chen, L. Zhang, T. Ji, K. Fu, L. Zhao, F. Chen and C.-T. Lu, "Bridging the gap between spatial and spectral domains: A survey on graph neural networks", (2020).

Colaprete, A., "Resource prospector mission goals and landing site constraints", SSERVI: Lunar Science for Landed Missions Workshop (2018).

Colby, M., L. Yliniemi and K. Tumer, "Autonomous multi agent space exploration with high-level human feedback", Journal of Aerospace Information Systems pp. 301–315 (2016).

Community, B. O., *Blender - a 3D modelling and rendering package*, Blender Foundation, Blender Institute, Amsterdam, URL `http://www.blender.org` (2019).

Deshmukh, V., K. Elamvazhuthi, S. Biswal, **Zahi M Kakish** and S. Berman, "Mean-field stabilization of Markov chain models for robotic swarms: Computational approaches and experimental results", IEEE Robotics and Automation Letters **3**, 3, 1985–1992 (2018).

Elamvazhuthi, K. and S. Berman, "Mean-field models in swarm robotics: A survey", Bioinspiration & Biomimetics **15**, 1, 015001 (2019).

Elamvazhuthi, K., **Zahi M Kakish**, A. Shirsat and S. Berman, "Controllability and stabilization for herding a robotic swarm using a leader: A mean-field approach", IEEE Transactions on Robotics 10.1109/TRO.2020.3031237 (2020).

Elamvazhuthi, K., S. Wilson and S. Berman, "Confinement control of double integrators using partially periodic leader trajectories", in "American Control Conference", pp. 5537–5544 (2016).

Ferrer, E. C., "A wearable general-purpose solution for Human-Swarm Interaction", in "Proceedings of the Future Technologies Conference", pp. 1059–1076 (Springer, 2018).

Garrido-Jurado, S., R. Muñoz-Salinas, F. Madrid-Cuevas and R. Medina-Carnicer, "Generation of fiducial marker dictionaries using mixed integer linear programming", Pattern Recognition **51** (2015).

Gillespie, D. T., "Stochastic simulation of chemical kinetics", Annu. Rev. Phys. Chem. **58**, 35–55 (2007).

Go, C. K., B. Lao, J. Yoshimoto and K. Ikeda, "A reinforcement learning approach to the shepherding task using SARSA", in "International Joint Conference on Neural Networks", pp. 3833–3836 (2016).

Gordon, S., "Talking to Martians: Communications with Mars Curiosity Rover", `https://sandilands.info/sgordon/communications-with-mars-curiosity`, URL `https://sandilands.info/sgordon/communications-with-mars-curiosity`, accessed: 2019 (2012).

Graesser, L. and W. L. Keng, *Foundations of deep reinforcement learning: theory and practice in Python* (Addison-Wesley Professional, 2019).

Gutin, G. and A. Punnen, "The traveling salesman problem and its variations", Springer Science and Business Media **12** (2006).

Hagberg, A., D. Schult and P. Swart, "Exploring network structure, dynamics, and function using networkx", SCIPY 08 (2008).

Heiken, G. H., D. T. Vaniman and B. M. French, "Lunar sourcebook-a user's guide to the moon", Research supported by NASA,. Cambridge, England, Cambridge University Press, 1991, 753 p. No individual items are abstracted in this volume. (1991).

Heintz, B., "Training a neural network to detect gestures with opencv in python", `https://towardsdatascience.com/training-a-neural-network-to-detect-gestures-with-op` URL `https://towardsdatascience.com/training-a-neural-network-to-detect-gestures-wit` (2018).

Huber, P. J., "Robust estimation of a location parameter", Ann. Math. Statist. **35**, 1, 73–101, URL `https://doi.org/10.1214/aoms/1177703732` (1964).

Hunter, J. D., "Matplotlib: A 2d graphics environment", (2007).

Hüttenrauch, M., S. Adrian and G. Neumann, "Deep reinforcement learning for swarm systems", Journal of Machine Learning Research **20**, 54, 1–31 (2019).

Iandola, F. N., S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size", arXiv preprint arXiv:1602.07360 (2016).

Ji, M., G. Ferrari-Trecate, M. Egerstedt and A. Buffa, "Containment control in mobile networks", IEEE Transactions on Automatic Control **53**, 8, 1972–1975 (2008).

JPL, N., "Moon trek", `https://trek.nasa.gov/moon/index.html`, URL `https://trek.nasa.gov/moon/index.html`, accessed: 2018-7-30 (2018).

Kamalapurkar, R., P. Walters, J. Rosenfeld and W. Dixon, *Reinforcement learning for optimal feedback control* (Springer, 2018).

Karavas, G. K., D. T. Larsson and P. Artemiadis, "A hybrid BMI for control of robotic swarms: Preliminary results", in "2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)", pp. 5065–5075 (IEEE, Vancouver, BC, 2017), URL `http://ieeexplore.ieee.org/document/8206390/`.

Kingma, D. P. and J. Ba, "Adam: A method for stochastic optimization", (2017).

Kipf, T. N. and M. Welling, "Semi-supervised classification with graph convolutional networks", (2017).

Koenig, N. P. and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator.", in "IEEE/RSJ Int'l. Conf. on Intelligent Robots and Systems", pp. 2149–2154 (2004).

Kullback, S. and R. A. Leibler, "On information and sufficiency", Ann. Math. Statist. **22**, 1, 79–86, URL `https://doi.org/10.1214/aoms/1177729694` (1951).

Lipton, Z. C., K. Azizzadenesheli, A. Kumar, L. Li, J. Gao and L. Deng, "Combating reinforcement learning's sisyphean curse with intrinsic fear", (2018).

Lu, D., D. Hershberger and W. Smart, "Layered costmaps for context-sensitive navigation", IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) pp. 709–715 (2014).

Mahoney, E., "Resource prospector", `https://www.nasa.gov/resource-prospector`, URL `https://www.nasa.gov/resource-prospector`, accessed: 2018 (2018).

Matthey, L., S. Berman and V. Kumar, "Stochastic strategies for a swarm robotic assembly system", in "2009 IEEE International Conference on Robotics and Automation", pp. 1953–1958 (IEEE, 2009).

Mesbahi, M. and M. Egerstedt, *Graph theoretic methods in multiagent networks*, vol. 33 (Princeton University Press, 2010).

Mnih, V., A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning", (2016).

Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, "Playing atari with deep reinforcement learning", arXiv preprint arXiv:1312.5602 (2013).

Music, S., G. Salvietti, D. Prattichizzo and S. Hirche, "Human-Multi-Robot Teleoperation for Cooperative Manipulation Tasks using Wearable Haptic Devices", IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) p. 8 (2017).

Nagi, J., A. Giusti, L. M. Gambardella and G. A. Di Caro, "Human-swarm interaction using spatial gestures", in "2014 IEEE/RSJ International Conference on Intelligent Robots and Systems", pp. 3834–3841 (IEEE, 2014).

Nair, V. and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines", in "ICML", (2010).

Nguyen, H. T., T. D. Nguyen, M. Garratt, K. Kasmarik, S. Anavatti, M. Barlow and H. A. Abbass, "A deep hierarchical reinforcement learner for aerial shepherding of ground swarms", in "Neural Information Processing", edited by T. Gedeon, K. W. Wong and M. Lee, pp. 658–669 (Springer International Publishing, Cham, 2019).

Nguyen, H. T., T. D. Nguyen, V. P. Tran, M. Garratt, K. Kasmarik, S. Anavatti, M. Barlow and H. A. Abbass, "Continuous deep hierarchical reinforcement learning for ground-air swarm shepherding", (2020).

Pal, A. and R. Tiwari, "Communication constraints multi-agent territory exploration task", Applied Intelligence **3**, 38, 357–383 (2013).

Paranjape, A. A., S.-J. Chung, K. Kim and D. H. Shim, "Robotic herding of a flock of birds using an unmanned aerial vehicle", IEEE Transactions on Robotics **34**, 4, 901–915 (2018).

Pierson, A. and M. Schwager, "Controlling noncooperative herds with robotic herders", IEEE Transactions on Robotics **34**, 2, 517–525 (2017).

ROBOTIS, "Turtlebot3", `http://www.robotis.us/turtlebot-3/`, URL `http://www.robotis.us/turtlebot-3/` (2019).

Romero Ramirez, F., R. Muñoz-Salinas and R. Medina-Carnicer, "Speeded up detection of squared fiducial markers", Image and Vision Computing **76** (2018).

Romero-Ramirez, F. J., R. Muñoz-Salinas and R. Medina-Carnicer", "Speeded up detection of squared fiducial markers", Image and Vision Computing **76**, 38–47, URL `http://www.sciencedirect.com/science/article/pii/S0262885618300799` (2018).

Šošić, A., A. M. Zoubir and H. Koeppl, "Reinforcement learning in a continuum of agents", Swarm Intelligence **12**, 1, 23–51 (2018).

Sutton, R. S. and A. G. Barto, *Reinforcement learning: An introduction* (MIT Press, 2018).

Team, Q. D., "Qgis", https://www.qgis.org, URL https://www.qgis.org, accessed: 2018 (2002).

Zahi M Kakish, "pheeno_ros ROS packages", https://github.com/acslaboratory/pheeno_ros, URL https://github.com/acslaboratory/pheeno_ros, accessed: 2019 (2018).

Zahi M Kakish, "Herding OpenAI Gym Environment", https://github.com/acslaboratory/gym-herding (2019).

Zahi M Kakish, "Multi Robot Trainer (MRT)", https://github.com/zmk5/multi_robot_trainer (2020a).

Zahi M Kakish, "ROS2 Robotarium", https://github.com/zmk5/ros2_robotarium (2020b).

Zahi M Kakish, K. Elamvazhuthi and S. Berman, "Using reinforcement learning to herd a robotic swarm to a target distribution", arXiv:2006.15807v2 [cs.RO] (2020).

Zahi M Kakish, K. Elamvazhuthi and S. Berman, "Using reinforcement learning to herd a robotic swarm to a target distribution", Autonomous Collective Systems Laboratory YouTube channel, https://youtu.be/py3Pe24YDjE (2020).

Zahi M Kakish, K. Elamvazhuthi and S. Berman, "Leader-based swarm herding using deep reinforcement learning algorithms trained on mean-field models", Swarm Intelligence (*in preparation*) (2021).

Zahi M Kakish, F. Rodríguez-Lera, D. Bischel, A. Mosquera, R. Boumghar, S. Kaczmarek, T. Seabrook, P. Metzger and J. L. Galanche, "Open-source AI assistant for cooperative multi-agent systems for lunar prospecting missions", in "8th European Conference for Aeronautics and Space Sciences (EUCASS)", (2019).

Zahi M Kakish, S. Vedartham and S. Berman, "Towards decentralized human-swarm interaction by means of sequential hand gesture recognition", in "2020 IEEE International Conference on Robotics and Automation (*submitted*)", (2020).

Thomas, D., W. Woodall and E. Fernandez, "Next-generation ROS: Building on DDS", (2014).

Walt, S., S. Colbert and G. Varoquaux, "The numpy array: A structure for efficient numerical computation", Computing in Science and Engineering **13**, 2, 22–30 (2011).

Wang, L., A. D. Ames and M. Egerstedt, "Safety barrier certificates for collisions-free multirobot systems", IEEE Transactions on Robotics **33**, 3, 661–674 (2017).

Warmerdam, F., "The geospatial data abstraction library", Open Source Approaches in Spatial Data Handling pp. 87–104 (2008).

Williams, R. J., "Simple statistical gradient-following algorithms for connectionist reinforcement learning", Machine learning **8**, 3-4, 229–256 (1992).

Wilson, S., R. Gameros, M. Sheely, M. Lin, K. Dover, R. Gevorkyan, M. Haberland, A. Bertozzi and S. Berman, "Pheeno, a versatile swarm robotic research and education platform", IEEE Robotics and Automation Letters **1**, 2, 884–891 (2016).

Wilson, S., P. Glotfelter, L. Wang, S. Mayya, G. Notomista, M. Mote and M. Egerstedt, "The Robotarium: Globally impactful opportunities, challenges, and lessons learned in remote-access, distributed control of multirobot systems", IEEE Control Systems Magazine **40**, 1, 26–44 (2020).

Yang, Y., R. Luo, M. Li, M. Zhou, W. Zhang and J. Wang, "Mean field multi-agent reinforcement learning", in "International Conference on Machine Learning", pp. 5567–5576 (2018).

Zhi, J. and J.-M. Lien, "Learning to herd agents amongst obstacles: Training robust shepherding behaviors using deep reinforcement learning", (2020).

Zhou, J., G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li and M. Sun, "Graph neural networks: A review of methods and applications", (2019).