FEEDBACK CONTROL OF MANY DIFFERENTIAL-DRIVE ROBOTS WITH UNIFORM CONTROL INPUTS

BY

CEM ONYUKSEL

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Adviser:

Assistant Professor Timothy Bretl

Abstract

We describe a method of feedback position control for an ensemble of robots with unicycle kinematics under the constraint that every robot receives exactly the same global control inputs. Each robot in the ensemble scales its given control inputs by a bounded model parameter and thus may achieve a turning rate and forward speed which are unique. Exploiting inhomogeneities in robot execution of control inputs, we derive a globally asymptotically stabilizing feedback control policy to regulate the position of each robot. This policy scales linearly for any number of robots, and it stabilizes the system asymptotically to the goal position even with Gaussian noise in actuation. Computer simulations and hardware experiments are used to validate the policy. Additionally, we propose methods for trajectory tracking and obstacle avoidance. Finally, we show an example of multi-robot object manipulation and assembly.

To my parents for their love and support

Acknowledgments

First and foremost, I would like to thank my advisor Timothy Bretl for his constant support and for always pushing me to be my best. Also, I would like to thank the members of RMS Lab for helping me brainstorm ideas and work through problems. You are all both supportive colleagues and good friends. I extend a special thanks to Aaron Becker for his massive contributions to my work in ensemble control; we make a great team! Thanks to Dan Block for his guidance in the lab as well as for letting me use his robots for experiments.

Table of Contents

Chapter 1

Introduction

We develop a globally asymptotically stabilizing feedback control policy that enables position control of a collection of differential-drive robots, which all receive the same control signal. The method presented is robust to standard models of noise and is globally asymptotically stable. Simulations involving thousands of robots and hardware experiments with four to six robots demonstrate that the control policy is viable. Our control policy can be used to control systems of micro- and nano-robots in which robots exhibit unicycle kinematics and all robots in the system receive the same global inputs. While not all micro- or nano-robot systems have the same unicycle kinematics described by our model, our method can be adapted to specific systems by adding constraints on the inputs or goal positions. This thesis draws from and extends much of the work presented in the author's previous publication [1].

1.1 Robust Control

Robust control is a mature field with a large body of work. Fundamentals of robust control — state space models, linear analysis, stabilization, optimal control, model uncertainty, and feedback — can be found in textbooks [2, 3]. The control policies presented in this thesis are feedback control policies to control multiple differential-drive robots with bounded model perturbations, and we draw from previous work on these topics.

Differential-drive robots are a class of robots that have wheels which can be independently controlled. Often, there are two drive wheels with additional unpowered castor wheels to balance the robot. These robots are nonholonomic systems with unicycle kinematics. Usually, lateral motion is not allowed, and thus differential-drive robots can be modeled as having a

forward velocity and turning rate, as we model our robots in Chapter 2.

Previous work analyzed the kinematics of a differential-drive robot with two drive wheels and a third castor wheel which has no wheel slip to developed PID (proportional + integral + derivative) feedback controllers for stable path-following by using nonlinear feedback linearization techniques [4]. Later work examined trajectory tracking by decoupling position control from velocity control to reduce internal and external sources of error and achieve accurate tracking [5]. Lucibello et al. provide methods for regulating position and orientation of a single robot with unicycle kinematics to achieve exponential convergence to the goal, rejecting disturbances [6]. We draw on these techniques to control the linear and angular velocity of the differential-drive robots used in the hardware experiments in Chapter 4.

The control methods described in this thesis are designed to control systems of not one, but many robots. Multi-agent systems have gained popularity as flocking, swarming, schooling, and distributed control have become more widely-studied; often, methods of multi-robot control combine low-level controllers for each robot that define group dynamics with high-level controllers that specify overall group tasks. The CMUnited-98 robot soccer champion team utilized control and planning methods that allowed the team of five robots to avoid moving obstacles in a dynamic environment while working cohesively as a team to achieve a high-level goal [7]. On their team, each differential-drive robot had its own on-board control algorithm to steer it along desired trajectories, while the high-level controller defined trajectories that would achieve the high-level goals of passing the ball and shooting on the opponents' goals. Egerstedt and Hu designed a method for controlling formations of robots where the low-level controller's goal is to keep the robots in a rigid formation, and one robot is designated as the leader [8]. In their method, the high-level controller moves the leader along desired trajectories; the formation is treated as a rigid body, and thus the entire formation moves along the desired trajectories.

In some systems, the individual robots can perform advanced tasks beyond staying in formation. An underwater de-mining system, developed by Tan, consists of a swarm of robots where each individual robot sweeps for mines while avoiding obstacles and other robots [9]. The entire swarm is controlled through statistical methods where the mean configuration and variance of the swarm is controlled by the high-level controller, resulting in

a hybrid controller that balances individual robot tasks with swarm tasks. Some multi-agent systems allow the group to break apart and re-form. In flocking methods, there is no need for a leader, and robots will rejoin a flock if they are separated from it (for example to go around an obstacle). Low-level controllers on each robot try to keep uniform distance between members of the flock while maintaining the same velocity and heading as neighbors, and the entire flock works together to perform a high-level task [10]. The multi-agent methods presented in this thesis draw from these methods, but individual robots do not need any on-board intelligence. The global controller we present takes both low-level objectives (trajectory control for each robot) and high-level objectives (tasks such as assembly) into account to send one uniform control signal to all robots in the system.

1.2 Sensorless Manipulation

The robotic systems explored in this thesis are made of robots that differ from each other by a bounded model parameter, but that parameter may not always be known. Sensorless manipulation can position and orient objects unambiguously without the need for sensing, and often without knowing specific information about the objects. Thus, it is possible to create open-loop paths that position and orient objects which are robust to different sizes, weights, and starting configurations of objects. Erdmann et al. designed a tray manipulator that could orient objects by tilting the tray and using the walls to force the objects inside the tray into certain orientations [11]. For certain shapes of objects, the resulting orientations could be completely determined, but for other shapes of objects, the number of possible orientations could only be reduced. In another system, built by Böhringer et al., a vibrating platform that could sort objects places on the platform by making use of frictional forces between objects and the platform [12]. Vibration patterns create a two-dimensional force field on objects, and chaining together sequences of vibrations allows objects to be positioned and oriented in the plane with no sensor feedback from the object. In yet another system, designed by Akella et al., polygonal parts are manipulated with a suction-gripping robot arm on a conveyor belt [13]. At one end of the conveyor belt, a fence forces the object to align one of its faces with the fence, and through

a series of rotations with the robot arm and alignments with the fence, it is possible to generate a sensorless plan to orient parts. Akella's system can orient parts within size and shape tolerances. Sensorless manipulation techniques show that open-loop methods can control the position and orientation of objects even if certain properties of the objects are unknown, similar to how the wheel size of our differential-drive robots may be unknown.

Often, it is desirable to control robots where a dynamic or kinematic model is not known. Closely related to sensorless manipulation, control policies which are robust to model perturbations allow the steering of robots where the kinematic or dynamic properties are unknown. Our systems consist of robots with various, bounded model parameters, so in addition to being robust to external disturbances, we need our controllers to be robust to model perturbations. Cheah and Slotine present a method of adaptively controlling robots where the kinematic and dynamic properties are unknown using sensing feedback and updating estimates of the robots' properties online as it moves [14]. In Sections 4.3 and 4.4, we present our method to update the kinematic model for each of our robots as they move and verify our method in hardware experiments. If the dynamic and kinematic parameters are both unknown and time-varying, then it may not be possible to learn them. Mao shows that such systems can be robustly stabilized with feedback control if their time-varying parameters are polytopic — within a set of possible values [15]. We show that our method works for unknown, uncalibrated, and even incorrect parameter values, as long as they are within the bounds of our model. The textbook Mobile Robots presents a feedback controller which guarantees exact asymptotic convergence for a single robot with the same type of model perturbations we consider — scaling of angular and linear velocities [16, Chap. 11.6.2]. We extend this work by showing that we can control many robots with different model parameters simultaneously.

1.3 Ensemble Control

Ensemble control is a framework in which instead of controlling each agent in a group individually, one controls the entire group at once in such a way that all the agents in the group are steered to their goal positions. Brockett and Khaneja developed methods to control the spin of ensembles of nuclear

particles with the objective of achieving the correct energy levels for certain measurements and imaging [17, 18]. Similar to our differential-drive robots, their nuclear particle systems are nonholonomic, and they have model parameters that scale system characteristics; in their case, interactions with other particles can cause the system to be at different energy levels, and they use ensemble control techniques to steer particles to the desired energy level. Li and Khaneja extend this work to controlling quantum ensembles using compensating pulse sequences, again for nuclear magnetic resonance imaging [19, 20]. Because the quantum particles have no intelligence and all the particles in an ensemble receive the same control inputs, it is necessary to design these global control inputs so as to control all the particles in the ensemble.

Li formally introduced the idea of ensemble control as a class of control problems beyond those of nuclear and quantum particles [21]. He also provided necessary and sufficient conditions for determining if an ensemble is controllable. The controllability of ensembles can be understood by taking high-order Lie brackets of an ensemble system and examining its vector fields to make a polynomial approximation of the infinite-dimensional system [22, 23]. Optimal control inputs can be calculated to drive an ensemble to a goal by approximation of the minimum-energy control law [24]. Ensemble control techniques can also be applied to finite-dimensional systems, similar to our systems of multiple robots, even if those systems are time-varying [25]. The conditions for controllability in this case are similar to those of the non-time-varying infinite case.

Building upon Li's work, Becker et al. demonstrated that open-loop position control of a unicycle with an unknown parameter could be achieved by applying ensemble control methods to the system [26]. By steering an entire ensemble of possible robots (which encompasses all possible parameter values) to the goal, one can guarantee that the robot is steered to its goal, even though its parameter is unknown. Specifically, motion primitives can be concatenated to generate arbitrarily accurate paths for the unicycle; error decreases exponentially with the number of primitives used. It is not possible to control the orientation of an ensemble of robots with unicycle kinematics [27], so ensemble control of unicycles is limited to position control. Nonetheless, position control can be very useful in robotic systems. As shown in simulation in Chapter 5, an ensemble of robots can be used as

a set of manipulators to move objects around and build larger structures. Also, Chapter 3 shows how position control allows trajectory tracking, which enables a host of applications where orientation is not critical.

An ensemble of unicycles is an example of an underactuated system — a system with more degrees of freedom than control inputs. One study of an underactuated system examined a two-link arm where only the bottom link was controlled [28]. To balance the arm in the vertical position, a swing-up algorithm was developed based on the total energy in the system, and the system switched to a linear controller when the arm was near the top. Improvements to the control design widen the basin of attraction by treating velocity of the second link as an unknown value (instead of trying to stabilize it at zero) [29]. So one approach to attempting to control underactuated systems is to exploit system properties such as total energy. Another approach is to only control a controllable subspace of the configuration space. Oriolo et al. demonstrate that although it is not possible to control a robot with unactuated joints to a single point, it is possible to control such a robot to a manifold of points [30]. Similarly, although we are not able to control the entire configuration of our ensembles of unicycles, we are able to control ensembles to a subspace (orientation is not fixed) and thus we show that we can control the position of an ensemble of robots with unicycle kinematics.

Groups of robots can manipulate objects by pushing them around the workspace. Lynch derived necessary and sufficient conditions for stable pushing — the object remains fixed to the pusher [31]. These conditions take into account friction between the pusher and the object and the trajectory of the pusher with respect to the orientation of the object. Pusher trajectories are shown in Figure 1.1. Drawing from this work, we can put constraints on an ensemble of pushing robots such that none of the robots in the ensemble violate the necessary conditions, and thus we could achieve stable pushing. Other methods of object manipulation involve surrounding objects with a formation of robots that traps the object, also called caging [32]. Algorithms for cooperative object manipulation by formations of robots may be decentralized [33]. We aim to be able to apply more sophisticated cooperative object manipulation algorithms to our ensemble of differential-drive robots, which would allow us to have guarantees about stability and success of manipulation.

Unlike many previous ensemble control methods, the policies presented

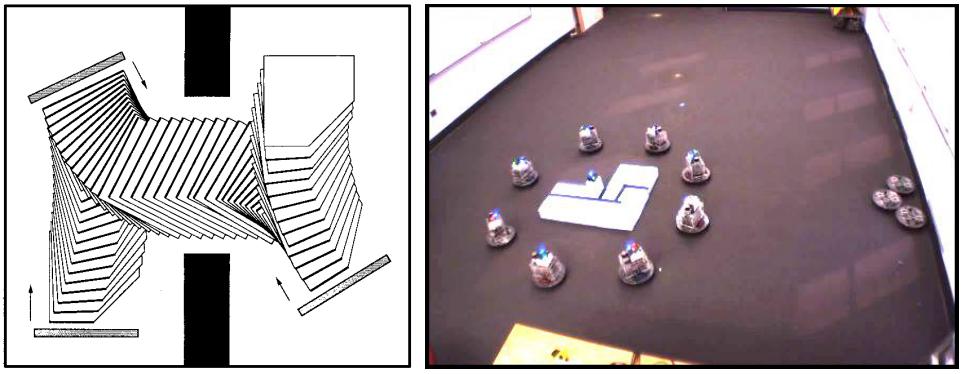

Figure 1.1: On the left, a stable pushing trajectory [31], ©1999 IEEE. On the right, object manipulation by caging [33], ©2008 IEEE.

in this thesis are closed-loop policies with feedback. As demonstrated in computer simulations and hardware experiments, extending open-loop ensemble control methods by adding feedback allows the system to be robust to noise and disturbances. This is especially important when applying ensemble control to real systems because there may be imperfections in actuation or sensing which would cause open-loop methods to perform poorly. In some cases, as shown in Chapters 3 and 4, noise in the system can actually improve performance when using a feedback ensemble control policy.

1.4 Micro- and Nano-Robots

The target of this work is to provide a framework which can be used to control micro- and nano-robot systems where it is not possible to send commands to individual robots. The control policies derived in Chapter 2 are for robots with unicycle kinematics. These include the three systems shown in Figure 1.2: light-driven nanocars, scratch-drive micro-robots, and radio-controlled differential-drive robots.

Nanometer-sized transporters, or nanocars, have properties similar to the differential-drive robots used in this thesis in simulations and hardware experiments. Initially, the cars were thermally powered [37], but modern iterations aim to be powered by light instead. The light-driven nanocar [34] is a synthesized molecule 1.7×1.38 nm in size containing a uni-directional molecular motor, actuated by a certain wavelength of light. Future work by Tour et al. aims to add controllable steering to this molecule.

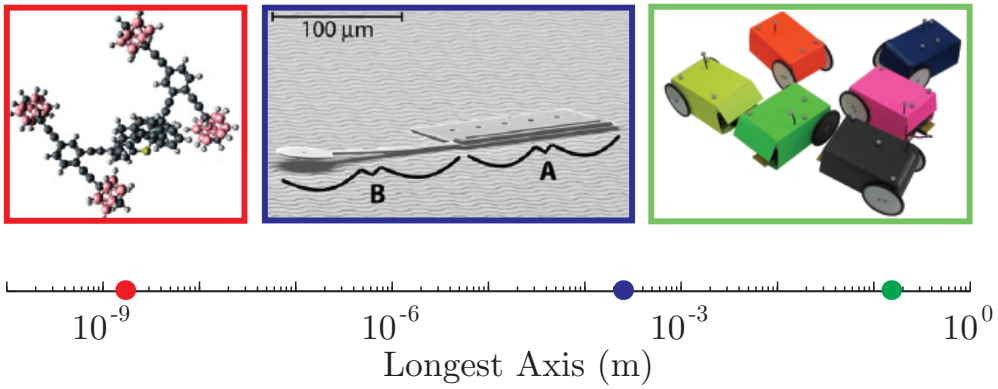

Figure 1.2: Three robotic systems with uniform inputs. On the left, light-driven nanocars, image adapted with permission from [34], ©2012 American Chemical Society. In the middle, scratch-drive micro-robots [35], ©2008 IEEE. On the right, six differential-drive Segbot robots [36].

The scratch-drive micro-robot, from Donald and Paprotny et al. [38], is a device 60×250 μm in size actuated by varying the electric potential across a substrate; multiple scratch-drive robots on the same substrate are controlled by this single uniform control input. To independently control each micro-robot, the system is designed with unique robots such that individual robots can be actuated while the others are immobilized or spin in place. Scratch-drive micro-robots can be used for assembly tasks by controlling each scratch-drive robot, in turn, into a formation [35]. In contrast, we show an example of an assembly task done by actuating all of the robots in our ensemble simultaneously in Chapter 5. Our approach can be adapted to scratch-drive micro-robots because small imperfections in each robot cause each one to move and turn at slightly different rates, which is precisely the model perturbation that we exploit to allow us to simultaneously actuate all the robots, yet steer each one toward a desired goal.

Our approach is related to recent work by Sitti et al. [39,40]. They manipulate the 2D coordinates of multiple geometrically-dissimilar cuboid permanent magnets (Mag-μBots) by exploiting heterogeneity in their dimensions and geometry to actuate groups of robots with magnetic fields. Their magnetic robots each respond differently to input fields, and the field can be selected to actuate individual or subgroups of robots. Similarly, we exploit heterogeneity in the wheel sizes of our robots; however, unlike the Mag μ-Bots, we require that all of the robots in our ensemble always act on every

input to the system; we do not do any switching between robots but rather actuate all of them at once.

To move objects at the micro- and nano-scale, very precise manipulators have been developed. Sebastian et al. developed a robust control algorithm for precise motion control of piezoactuators from atomic force microscopes (AFM) [41]. Piezoactuators are used to move nanoscopic objects in the workspace. Another nonlinear control technique for piezoactuators was developed by Bashash et al. [42]. AFM actuation methods are very precise, but manipulation with AFM is difficult because each object needs to be pushed one at a time; we propose a method for manipulating multiple robots, and thus multiple objects simultaneously.

Other methods of nano-manipulation include physical, fluidic, magnetic, and optical methods. Expanding on methods of sensorless manipulation and parts-handing, Böringer et al. use vibration patterns to manipulate and sort micro-scale parts on a plane [43]. Electrowetting on dielectrics (EWOD) is a fluidic method that controls micro-robots by altering properties of the fluid which transports them [44]. Both of these methods allow multiple objects to be controlled simultaneously. A magnetic system, PolyMites, can be manufactured to have different linear velocities, but currently, they are steered such that all robots in the workspace have the same orientation, so ensemble control cannot immediately be applied to this system [45]. Cells implanted with magnetic diodes, magnetotaxis, can be steered with a magnetic field, but current work does not allow their velocity to be controlled [46]. However, magnetotaxis in the same workspace share a global control signal, similar to the unicycle systems discussed in this thesis. Ensemble control policies can be leveraged to control multiple magnetotaxis simultaneously as long as the turning rates are different or there is sufficient noise in the system, as shown in Chapter 3.

Hu et al. developed an optical system where bubbles in oil are manipulated by focusing patterns of light in the workspace [47]. This creates a thermal gradient to actuate the bubbles. This system was used with multiple micro-robots (bubbles) to manipulate six robots simultaneously (in two groups of three), and the robots themselves were used as manipulators to manipulate beads in the oil [48]. Different-sized bubbles react to light patterns with different forces, so this system can be modeled as an ensemble with bounded model parameters. Applying ensemble control to this work could allow many

bubbles to be independently actuated while applying a uniform global control input to the system. This may allow more complex manipulation tasks.

Inspired by previous work on robust control and ensemble control, this thesis *(1)* provides a globally asymptotically stabilizing feedback control policy to control an ensemble of differential-drive robots, *(2)* demonstrates its convergence under a standard noise model in simulation and hardware experiments, with constraints on inputs and around obstacles, *(3)* shows that the policy still works when the parameter values are incorrectly specified or if all robots are identical, and *(4)* gives an example of a simple assembly task that can be performed with an ensemble as a robotic manipulator. This work is directly applicable to micro- and nano-robot systems that have unicycle kinematics and may be applied to other types of micro- and nano-robots with some adaptation.

Chapter 2

Controlling an Ensemble of Unicycles

In this chapter, a globally asymptotically stabilizing feedback controller for an ensemble of unicycles is developed through Lyapunov analysis [49]. The resulting controller regulates the position of each robot in the ensemble such that it is robust to external disturbances and Gaussian noise in actuation, as demonstrated in later chapters. The derived control policy will move robots closer to their goals when it is possible to do so by controlling the linear velocity, $u_1(t)$, of the ensemble; it will never increase the average distance of the ensemble to the goal. There exist configurations where no $u_1(t)$ can decrease the position error; however, it is proven that for any such configuration, except the origin, the ensemble can always be rotated in place such that there exists some $u_1(t)$ which will decrease the position error.

2.1 System Kinematics

Consider a single nonholonomic unicycle that rolls without slipping. Its configuration is described by its 2-dimensional position and orientation, $q = [x, y, \theta]^\top$, and its configuration space is $\mathcal{Q} = \mathbb{R}^2 \times \mathbb{S}^1$. The control inputs are the forward speed $u_1(t) \in \mathbb{R}$ and turning rate $u_2(t) \in \mathbb{R}$. The kinematics of the unicycle are given by

$$\dot{q}(t) = u_1(t) \begin{bmatrix} \cos\theta \\ \sin\theta \\ 0 \end{bmatrix} + u_2(t) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{2.1}$$

Given $q(0), q_{\text{goal}} \in \mathcal{Q}$, the control problem for regulating the position of a

single robot is to find inputs $u_1(t)$ and $u_2(t)$ such that for any $q(0)$ and q_{goal},

$$\lim_{t\to\infty} \left\| \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} (q(t) - q_{\text{goal}}(t)) \right\|_2 = 0.$$

If such inputs always exist, then we say that the system is globally asymptotically stabilizable.

To extend the system to an ensemble, the control problem is solved under model perturbations which scale $u_1(t)$ and $u_2(t)$ by some unknown, bounded constant $\epsilon \in [1-\delta, 1+\delta]$ for some $0 \leq \delta < 1$. The ensemble control policy steers an uncountably infinite collection of unicycles parametrized by ϵ, each one governed by

$$\dot{q}_\epsilon(t) = \epsilon \left(u_1(t) \begin{bmatrix} \cos\theta_\epsilon(t) \\ \sin\theta_\epsilon(t) \\ 0 \end{bmatrix} + u_2(t) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right), \tag{2.2}$$

where θ_ϵ is the orientation of the robot with parameter ϵ.

The main control input to the system is $u_1(t)$, and the next section will describe how to choose $u_1(t)$ at every time t. However, $u_2(t)$ must also be specified such that the ensemble's orientation is not static. So one possible function is $u_2(t) = 1$. Then $\theta = \theta_\epsilon^0 + \epsilon t$, where θ_ϵ^0 is the starting orientation of the robot with parameter ϵ. Substituting into (2.2), the resulting system is defined by:

$$\begin{aligned} \dot{x}_\epsilon(t) &= \epsilon u_1(t) \cos\left(\theta_\epsilon^0 + \epsilon t\right) \\ \dot{y}_\epsilon(t) &= \epsilon u_1(t) \sin\left(\theta_\epsilon^0 + \epsilon t\right). \end{aligned} \tag{2.3}$$

2.2 Designing a Control Policy

A control policy which globally asymptotically stabilizes the position of an ensemble to its goal can be found by using a control-Lyapunov function. Without loss of generality, assume the goal position for each robot in the ensemble is $(0, 0)$. A suitable candidate Lyapunov function must be continuous, positive-definite, and radially unbounded in order to guarantee global

asymptotic stability; one such function is the sum-squared distance of the ensemble from the origin, weighted by ϵ:

$$V(t,x,y) = \int_{1-\delta}^{1+\delta} \frac{1}{2\epsilon} \left(x_\epsilon^2(t) + y_\epsilon^2(t)\right) d\epsilon. \tag{2.4}$$

Theorem 1. *The ensemble* (2.3) *with* $0 \leq \delta < 1$ *is globally asymptotically stabilizable.*

Proof. Begin by taking the derivative of the candidate Lyapunov function, $V(t,x,y)$.

$$\begin{aligned}
V(t,x,y) &= \int_{1-\delta}^{1+\delta} \frac{1}{2\epsilon} \left(x_\epsilon^2(t) + y_\epsilon^2(t)\right) d\epsilon \\
\dot{V}(t,x,y) &= \int_{1-\delta}^{1+\delta} \frac{1}{\epsilon} \left(x_\epsilon(t)\dot{x}_\epsilon(t) + y_\epsilon(t)\dot{y}_\epsilon(t)\right) d\epsilon \\
&= u_1(t) \int_{1-\delta}^{1+\delta} \left(x_\epsilon(t)\cos\left(\theta_\epsilon^0 + \epsilon t\right) + y_\epsilon(t)\sin\left(\theta_\epsilon^0 + \epsilon t\right)\right) d\epsilon \\
&= u_1(t) F(t,x,y)
\end{aligned}$$

Here, $F(t,x,y)$ is the integral term which is finite as long as $x_\epsilon(t)$ and $y_\epsilon(t)$ are square integrable over ϵ. Note that $V(t,x,y)$ is positive definite and radially unbounded, and $V(t,x,y) \equiv 0$ only at the origin, where the origin is defined as

$$(x_\epsilon(t), y_\epsilon(t)) = (0,0), \quad \forall \epsilon.$$

Next, choose

$$\begin{aligned}
u_1(t) &= -F(t,x,y) \\
&= -\int_{1-\delta}^{1+\delta} \left(x_\epsilon(t)\cos\left(\theta_\epsilon^0 + \epsilon t\right) + y_\epsilon(t)\sin\left(\theta_\epsilon^0 + \epsilon t\right)\right) d\epsilon.
\end{aligned} \tag{2.5}$$

With this choice of $u_1(t)$,

$$\dot{V}(t,x,y) = -\left(F(t,x,y)\right)^2.$$

Note that $\dot{V}(t,x,y) \leq 0$, but there exists a subspace of $(x_\epsilon(t), y_\epsilon(t))$ such that $\dot{V}(t,x,y) = 0$. Because $\dot{V}(t,x,y)$ is negative semi-definite, one can only claim stability, not asymptotic stability.

To gain a proof of asymptotic stability, we use an approach similar to that of Beauchard et al. [50] to extend LaSalle's invariance principle [51] to this infinite-dimensional system. The proof proceeds by showing the invariant set — a set which once entered the function $V(t, x, y)$ will never leave — contains only the origin.

Let the set S be all the $(x_\epsilon(t), y_\epsilon(t))$ configurations where there is no $u_1(t)$ which will decrease the position error of the system:

$$\begin{aligned} S &= \left\{ x_\epsilon(t), y_\epsilon(t) \middle| \dot{V}(t, x, y) = 0 \right\} \\ &= \left\{ x_\epsilon(t), y_\epsilon(t) \middle| -\left(F(t, x, y)\right)^2 = 0 \right\} \\ &= \left\{ x_\epsilon(t), y_\epsilon(t) | F(t, x, y) = 0 \right\}. \end{aligned}$$

Let S_{inv} be the invariant set; it is a subset of S where $F(t, x, y) = 0$ for all orientations of the ensemble. When the ensemble reaches a position in S_{inv}, there is no orientation, $\theta_\epsilon^0 + \epsilon u_2(t) = \theta_\epsilon^0 + \epsilon t$, such that the ensemble can move to reduce error. In other words, once the ensemble is at a position in S_{inv}, there will never exist a $u_1(t)$ which will move the ensemble to a position with lower error, regardless of the orientation.

The invariant set is defined by the following expression:

$$\begin{aligned} S_{inv} = \Bigg\{ x_\epsilon, y_\epsilon \Bigg| \int_{1-\delta}^{1+\delta} \big(& x_\epsilon \cos\left(\theta_\epsilon^0 + \epsilon t\right) \\ & + y_\epsilon \sin\left(\theta_\epsilon^0 + \epsilon t\right) \big) d\epsilon = 0, \quad \forall t \Bigg\}. \end{aligned} \tag{2.6}$$

To show that the only point in S_{inv} is the origin, assume for the sake of contradiction that there exists a point in S_{inv} such that $(x_\epsilon, y_\epsilon) \neq (0, 0)$ for some $\epsilon \in [1 - \delta, 1 + \delta]$.

The Fourier transform operation is defined as

$$\mathcal{F}\left[f(t)\right] = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{\mathrm{i}\omega t} dt.$$

Then, apply a Fourier transform to $F(t, x, y)$ from time domain t to frequency domain w.

$$\mathcal{F}\left[\int_{1-\delta}^{1+\delta}\left(x_\epsilon\cos\left(\theta_\epsilon^0+\epsilon t\right)+y_\epsilon\sin\left(\theta_\epsilon^0+\epsilon t\right)\right)d\epsilon\right]\{\omega\}=\mathcal{F}\left[0\right],\quad\forall\omega$$

$$\int_{1-\delta}^{1+\delta}\left(\mathcal{F}\left[x_\epsilon\cos\left(\theta_\epsilon^0+\epsilon t\right)\right]\{\omega\}\right.$$

$$\left.+\mathcal{F}\left[y_\epsilon\sin\left(\theta_\epsilon^0+\epsilon t\right)\right]\{\omega\}\right)d\epsilon=0,\quad\forall\omega$$

$$\int_{1-\delta}^{1+\delta}e^{-\mathrm{i}\theta_\epsilon^0}\sqrt{\frac{\pi}{2}}(x_\epsilon\left(\underline{\boldsymbol{\delta}}(-\epsilon+\omega)+\underline{\boldsymbol{\delta}}(\epsilon+\omega)\right)$$

$$+\mathrm{i}y_\epsilon\left(\underline{\boldsymbol{\delta}}(-\epsilon+\omega)-\underline{\boldsymbol{\delta}}(\epsilon+\omega)\right))d\epsilon=0,\quad\forall\omega,\qquad(2.7)$$

where $\underline{\boldsymbol{\delta}}(\cdot)$ is the Dirac-delta operator. The Dirac-delta operator is non-zero only when $\epsilon=\pm\omega$. Thus, integrating (2.7) yields

$$x(\omega)\pm\mathrm{i}y(\omega)=0,\quad\forall\omega,$$

and because x and y are both real-valued, it reduces to

$$x_\epsilon=0,y_\epsilon=0,\quad\forall\epsilon\in[1-\delta,1+\delta].$$

So there is a contradiction because we initially assumed that $(x_\epsilon,y_\epsilon)\neq(0,0)$ for some $\epsilon\in[1-\delta,1+\delta]$. Therefore, the only point in S_{inv} is the origin, and LaSalle's invariance principle holds.

The Lyapunov function, $V(t,x,y)$, is positive-definite and radially unbounded, its derivative, $\dot{V}(t,x,y)$, is negative semi-definite, and the only invariant point where $\dot{V}=0$ is the origin. Therefore, the origin of the system (2.3) is globally asymptotically stable under the control policy

$$\begin{aligned}u_1(t)=&-\int_{1-\delta}^{1+\delta}\left(x_\epsilon(t)\cos\left(\theta_\epsilon^0+\epsilon t\right)+y_\epsilon(t)\sin\left(\theta_\epsilon^0+\epsilon t\right)\right)d\epsilon\\u_2(t)=&1.\end{aligned}\qquad(2.8)$$

□

2.3 A Finite Ensemble in Continuous Time

The previous section demonstrates that an ensemble with infinitely-many robots can be controlled in continuous time such that it globally asymptotically stabilizes to its goal position. However, most real-life applications involve a finite number of robots. We call an ensemble with a finite number of robots a finite ensemble.

To model a finite ensemble of n robots, redefine the system kinematic model from (2.3) by subscripting the state and ϵ with i, representing the ith robot in the ensemble:

$$\begin{aligned}\dot{x}_i =& \epsilon_i u_1(t)\cos\left(\theta_i(t)\right)\\ \dot{y}_i =& \epsilon_i u_1(t)\sin\left(\theta_i(t)\right)\\ \dot{\theta}_i =& \epsilon_i u_2(t).\end{aligned} \tag{2.9}$$

To adapt to discrete time, the control policy (2.8) is modified such that the integration over ϵ is replaced by a finite sum from 1 to n:

$$\begin{aligned}u_1(t) =& -\frac{1}{n}\sum_{i=1}^{n}\left(x_i(t)\cos\left(\theta_i^0+\epsilon_i t\right)+y_i(t)\sin\left(\theta_i^0+\epsilon_i t\right)\right)\\ u_2(t) =& 1,\end{aligned} \tag{2.10}$$

where for the ith robot, ϵ_i is the variable parameter, $(x_i(t), y_i(t))$ is the position at time t, and $\theta_i(t) = \theta_i^0 + \epsilon_i t$ is the orientation at time t.

Theorem 2. *The finite ensemble 2.9 under control law 2.10 is globally asymptotically stable.*

Proof. A suitable Lyapunov function is the mean squared distance of the

finite ensemble from the origin, weighted by ϵ:

$$\begin{aligned} V(t,x,y) =& \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2\epsilon_i}\left(x_i^2+y_i^2\right) \\ \dot{V}(t,x,y) =& \frac{1}{n}\sum_{i=1}^{n}\frac{1}{\epsilon_i}\left(x_i\dot{x}_i+y\dot{y}_i\right) \\ =& u_1(t)\frac{1}{n}\sum_{i=1}^{n}\left(x_i\cos\left(\theta_i^0+\epsilon_i t\right)+y_i\sin\left(\theta_i^0+\epsilon_i t\right)\right) \\ =& u_1(t)F(t,x,y) \end{aligned} \tag{2.11}$$

Once again, the invariant set S_{inv} is the set of positions $(\mathbf{x},\mathbf{y})$ — with $\mathbf{x}$ and $\mathbf{y}$ each being vectors of the positions of n robots — where there is no orientation such that the ensemble can move to decrease error in position:

$$S_{inv}=\left\{\mathbf{x},\mathbf{y}\middle|\frac{1}{n}\sum_{i=1}^{n}\left(x_i\cos\left(\theta_i^0+\epsilon_i t\right)+y_i\sin\left(\theta_i^0+\epsilon_i t\right)\right)=0,\quad \forall t\right\}. \tag{2.12}$$

To show that the only point in S_{inv} is the origin, assume for the sake of contradiction that there exists a point in S_{inv} such that $(\mathbf{x},\mathbf{y})\neq(0,0)$.

As in section 2.2, then apply the Fourier transform to $F(t,x,y)$.

$$\begin{aligned} \frac{1}{n}\sum_{i=1}^{n}e^{-\mathrm{i}\theta_\epsilon^0}\sqrt{\frac{\pi}{2}}(&x_i\left(\underline{\boldsymbol{\delta}}(-\epsilon_i+\omega)+\underline{\boldsymbol{\delta}}(\epsilon_i+\omega)\right) \\ &+\mathrm{i}y_i\left(\underline{\boldsymbol{\delta}}(-\epsilon_i+\omega)-\underline{\boldsymbol{\delta}}(\epsilon_i+\omega)\right))=0,\quad\forall\omega. \end{aligned} \tag{2.13}$$

Again x_i and y_i are real-valued. By setting $\omega\pm\epsilon_i$ for $i\in[1,n]$ it follows that in the invariant set

$$(x_i,y_i)=(0,0)\quad\forall i\in[1,n],$$

so there is a contradiction, and $(\mathbf{x},\mathbf{y})$ must be the origin. Therefore, the finite ensemble is globally asymptotically stabilizable with control policy (2.10). □

2.4 A Finite Ensemble in Discrete Time

Sometimes it is more natural to control systems in discrete-time, such as when feedback is received at a certain sampling rate. To implement the feedback control policy (2.10) on a robotic testbed with actuation and sensing at discrete times, we break time into discrete increments of size ΔT and in each time-step we perform two stages of actuation. During the first stage we apply a linear velocity, and during the second stage we command the robots to turn in place.

$$\begin{aligned} k =& \frac{t}{\Delta T} - \operatorname{mod}(t, \Delta T) \\ F(k) =& \frac{1}{n} \sum_{i=1}^{n} \left(x_i(k) \cos(\theta_i(k)) + y_i(k) \sin(\theta_i(k)) \right) \\ \Big[u_1(k), u_2(k) \Big] =& \begin{cases} [-F(k), 0] & \text{stage 1} \\ [0, \phi] & \text{stage 2} \end{cases} \end{aligned} \tag{2.14}$$

We can then write the kinematics for the position as

$$\begin{bmatrix} x_i(k+1) \\ y_i(k+1) \end{bmatrix} = \begin{bmatrix} x_i(k) \\ y_i(k) \end{bmatrix} + \begin{bmatrix} \epsilon_i \cos(\theta_i(0) + \epsilon_i k\phi) \\ \epsilon_i \sin(\theta_i(0) + \epsilon_i k\phi) \end{bmatrix} u_1(k), \tag{2.15}$$

for $i = 1, 2, \ldots, n$ and $k \in \mathbb{Z}$.

Eq. (2.15) is a discrete-time linear time-varying system. As $\Delta T \to 0$, the discrete-time ensemble (2.15) approaches the continuous-time model (2.9). To prove (2.14) stabilizes (2.15), we show the system is uniformly k-step controllable, as in The Control Handbook [52, chap 25.3]. In standard notation, (2.15) is written as

$$q_i(k+1) = A_i(k) q_i(k) + B_i(k) u(k). \tag{2.16}$$

Here $A_i(k)$ is the identity matrix for all i, k. We can calculate $B_i(k)$ as

$$
\begin{aligned}
B_i(0) &= \begin{bmatrix} \cos(\theta_i(0)) \\ \sin(\theta_i(0)) \end{bmatrix} \\
B_i(1) &= \begin{bmatrix} \cos(\theta_i(0) + \epsilon_i \phi) \\ \sin(\theta_i(0) + \epsilon_i \phi) \end{bmatrix} \\
&\vdots \\
B_i(k) &= \begin{bmatrix} \cos\left(\theta_i(0) + \epsilon_i k \phi\right) \\ \sin\left(\theta_i(0) + \epsilon_i k \phi\right) \end{bmatrix} \\
\mathbf{B}(k) &= \begin{bmatrix} B_1(k) \\ B_2(k) \\ \vdots \\ B_n(k) \end{bmatrix}
\end{aligned}
$$

The controllability matrix $\mathcal{C}_k$ is defined as

$$
\mathcal{C}_k = \left[\mathbf{B}_0, \mathbf{B}_1 \dots \mathbf{B}_{k-1}\right].
$$

The finite ensemble with n robots has $2n$ degrees of freedom. To control each robot's x, y position requires $\mathcal{C}$ to be rank $2n$. This matrix is almost always full rank provided that $k \gg 2n$ and a suitable choice of ϕ. In our simulations and hardware experiments we use $\phi = \frac{\pi}{2}$. If $\mathcal{C}_k$ is full rank, then for any starting state $\mathbf{q}_0$ and desired final state $\mathbf{q}_1$, the control sequence is derived by solving in the least squares sense the overdetermined system of equations

$$
\mathcal{C}_k \mathbf{u}_{[0,\dots,k-1]} = (\mathbf{q}_1 - \mathbf{q}_0). \tag{2.17}
$$

We note that for $k = 2n$, $\mathcal{C}$ is almost always ill-conditioned, leading to very large control commands and poor convergence. Better results are obtained for $k = 5n$, as shown in Figure 2.1, with control effort 15 orders of magnitude less than that for $k = 2n$ and exact convergence to the goal.

In this chapter, methods for controlling ensembles of robots using feedback control were presented, and it was proven that these methods globally asymptotically stabilize an ensemble to a goal position. These control policies can be used to steer collections of infinitely-many robots in continuous-time and collections of finitely-many robots in both continuous- and discrete-time.

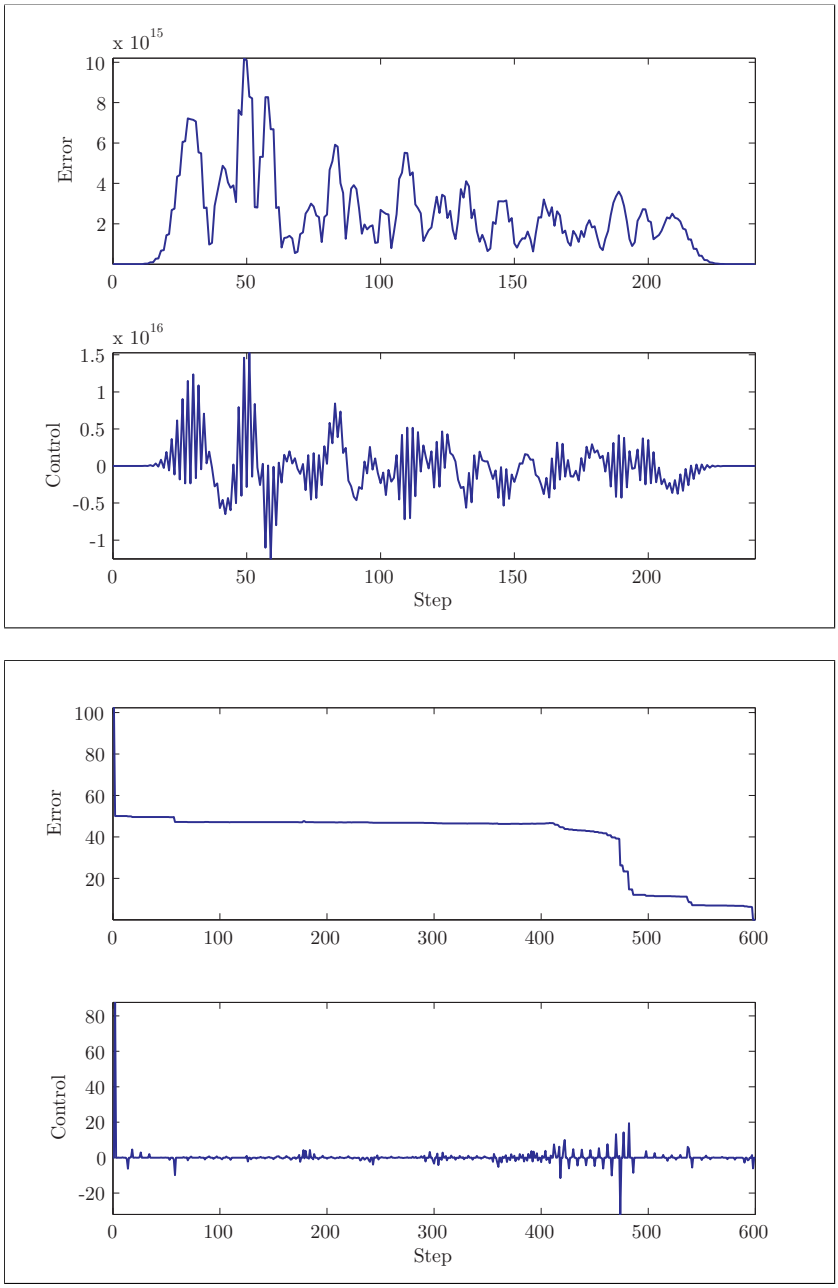

Figure 2.1: Error and control effort of a discrete-time finite ensemble of $n = 120$ robots under control (2.17). At top are results for $k = 2n$. The controllability matrix $\mathcal{C}_k$ is ill conditioned, leading to poor convergence and large control efforts. The bottom plot shows $k = 5n$, leading to control effort 15 orders of magnitude less and convergence to the goal. The initial error for each simulation is 100, but with $k = 2n$ the final error is 58, while the final error for $k = 5n$ is zero.

Chapter 3

Implementation and Extensions

This chapter describes specific implementations of the ensemble control methods described in Chapter 2 and simulations of ensembles in continuous- and discrete-time. Also, we discuss constraints on the inputs, apply a standard noise model to discrete-time systems, and show how ensembles can track paths. The implementations and extensions described in this chapter are beneficial for applying the theoretical control policies to real systems, such as in Chapter 4.

3.1 Simulation of Continuous-Time Systems

We simulated the finite ensemble described in (2.9) with the control policy shown in (2.10). Simulations were conducted in MATLAB, using ODE-45 to simulate $n = \{1000, 2000\}$ robots in continuous time for two different test cases. For these tests $\delta = 1/2$, and $\epsilon_i = 1 - \delta + \frac{2\delta}{n} i$. For the continuous-time simulations, $u_2(t) = \cos(\sqrt{t})$ because a finite ensemble poorly approximates an infinite ensemble for large t when $u_2(t) = 1$. The error plots show the average distance of a single robot from its goal over time. The source code for all discrete-time MATLAB simulations is in Appendix A.1.1.

3.1.1 Point to Point

Robots are initialized to $(x_i, y_i, \theta_i) = (1, 1, 0)$ and steered to the origin. Results are shown in Figure 3.1. As the error plot shows, error never increases, and the error quickly and asymptotically approaches zero. This simulation verifies that our continuous-time finite ensemble control policy works for a large number of robots with $\epsilon \in [1 - \delta, 1 + \delta]$ that all start in the same configuration.

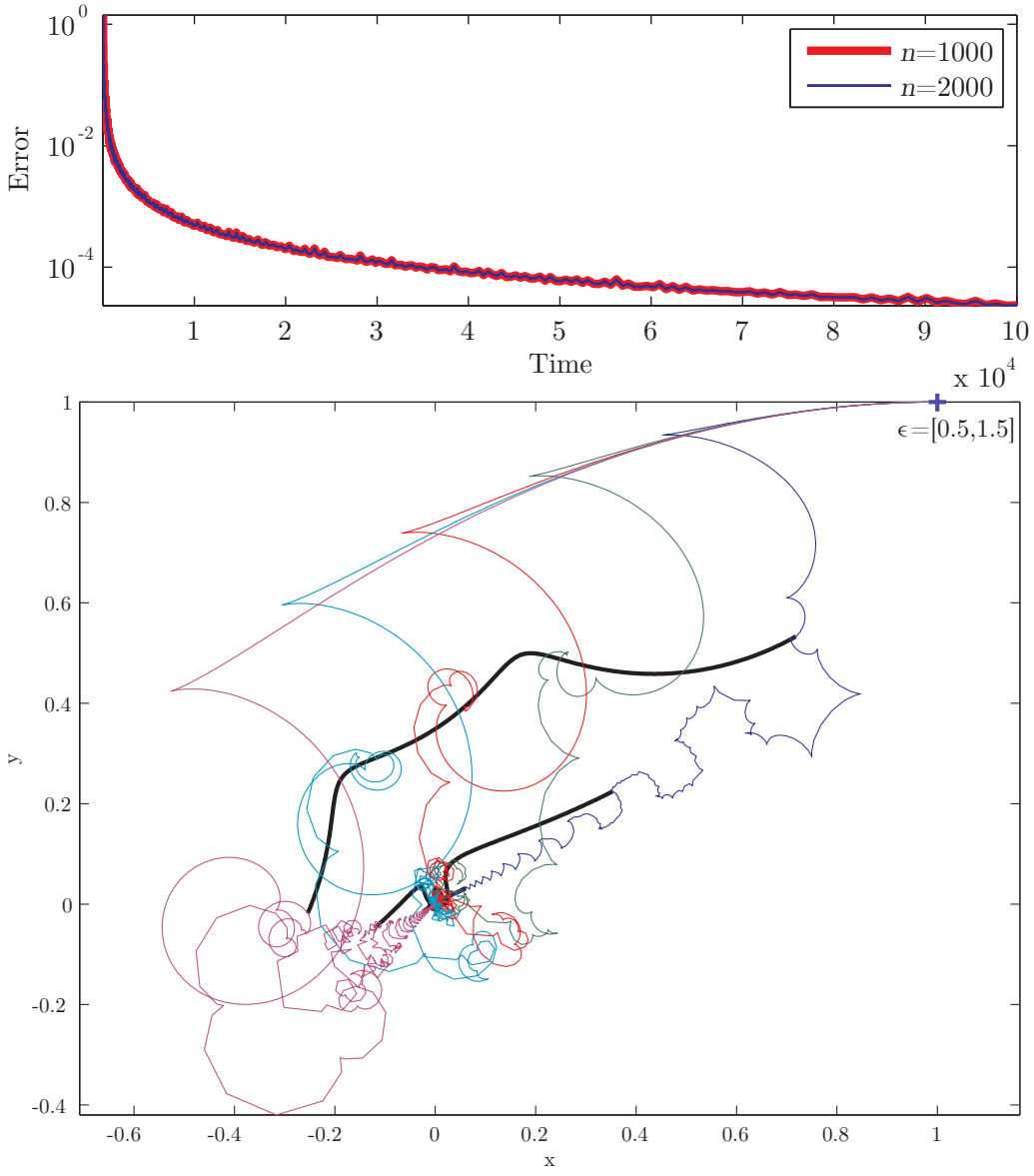

Figure 3.1: Continuous-time simulation of n robots, with $\epsilon \in [0.5, 1.5]$, all initialized to $(1, 1)$ and steered to $(0, 0)$ using control policy (2.10) and $u_2(t) = \cos(\sqrt{t})$. The simulation was run with $n = \{1000, 2000\}$. Each trial achieved the same error, as shown in the top plot. State trajectories of the system are shown in the bottom plot. Lines show the path followed for five particular values of ϵ. Thick black lines show the entire ensemble at instants of time.

3.1.2 Path to Point

Robots are initialized to $\theta_i = 2\pi i/n$, $(x_i, y_i) = (\cos(\theta_i), \sin(\theta_i))$, a circle of radius 1, and steered to the origin. Results are shown in Figure 3.2. Again, the error plot shows a monotonic, asymptotic decrease. This simulation verifies that our continuous-time finite ensemble control policy works for a large number of robots with $\epsilon \in [1-\delta, 1+\delta]$ that all start in different configurations.

From these simulations, we see that under our control policy, the error converges asymptotically to zero. Additionally, the system errors and trajectories for $n = 1000$ and 2000 are identical, suggesting that this level of discretization accurately represents the infinite ensemble ($n = \infty$) kinematics.

3.2 Extension to Unidirectional Vehicles

Some systems, including the nanocar and scratch-drive micro-robot, have unidirectional constraints on their inputs. To handle linear velocity constraints, we modify (2.8) to be non-negative in $u_1(t)$:

$$\begin{aligned} u_1(t) &= \max\left(0, -\int_{1-\delta}^{1+\delta} \left(x_\epsilon(t)\cos\left(\theta_\epsilon^0 + \epsilon t\right) + y_\epsilon(t)\sin\left(\theta_\epsilon^0 + \epsilon t\right)\right) d\epsilon\right) \\ u_2(t) &= \cos\left(\sqrt{}(t)\right). \end{aligned} \tag{3.1}$$

Recall that for continuous-time simulations, $u_2(t)$ is modified to get a more accurate representation of an infinite ensemble. This policy can be extended to robots with minimum turning radius (e.g. [35,38]) by redefining the robot center as the center of rotation; because our control policy requires only non-negative turns, it can be extended to vehicles with a minimum turning radius by the following transformation: given a minimum turning radius r_{min}, define the new robot center to be the center of rotation. This is a translation $(-r_{min}, 0)$ in the robot reference frame, as shown in Figure 3.3.

In simulation, robots are initialized in the same manner as 3.1.1, but simulated with the bidirectional control policy (2.10) as well as the unidirectional control policy (3.1). From the results shown in Figure 3.4, one can conclude that constraining inputs to only be positive slows down convergence of the

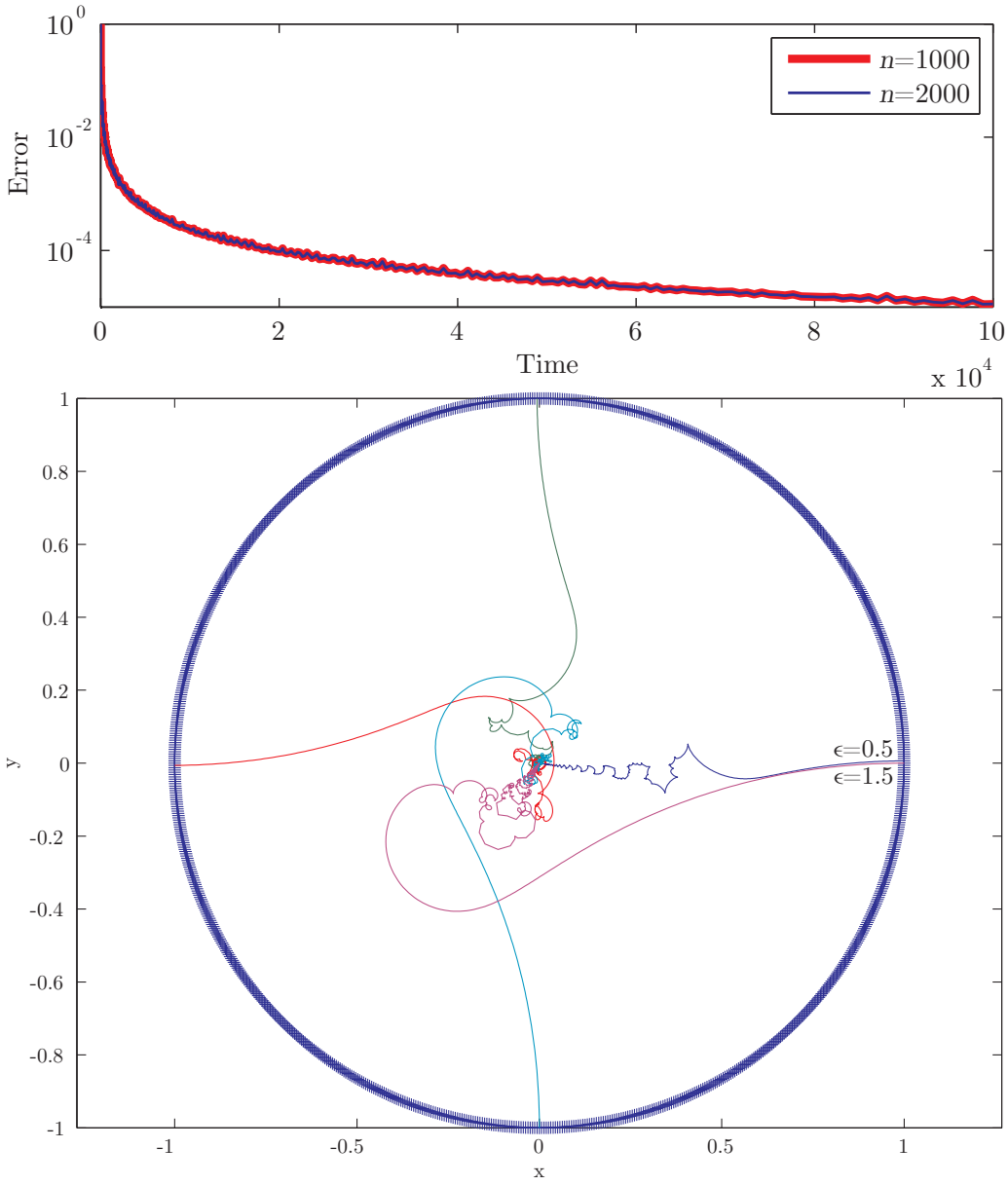

Figure 3.2: Continuous-time simulation of n robots, with $\epsilon \in [0.5, 1.5]$, initially evenly distributed about the unit circle and steered to $(0, 0)$ using control policy (2.10) and $u_2(t) = \cos(\sqrt{t})$. The simulation was run with $n = \{1000, 2000\}$. Each trial achieved the same ending error, as shown in the top plot. State trajectories of the system are shown in the bottom plot. Lines show the path followed for five particular values of ϵ. Thick black lines show the entire ensemble at instants of time.

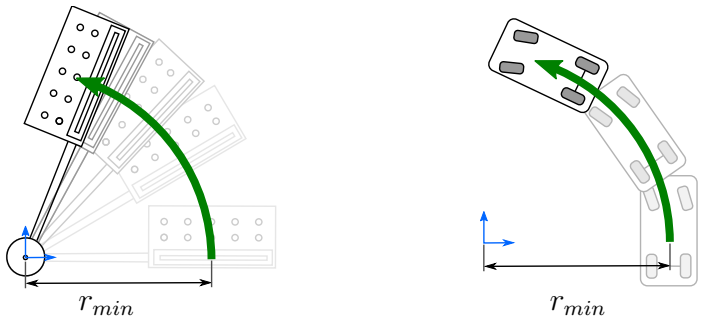

Figure 3.3: Because our control policy requires only non-negative turns, it can be extended to vehicles with a minimum turning radius (e.g. scratch-drive robots and automobiles) by defining the robot center to be the center of rotation.

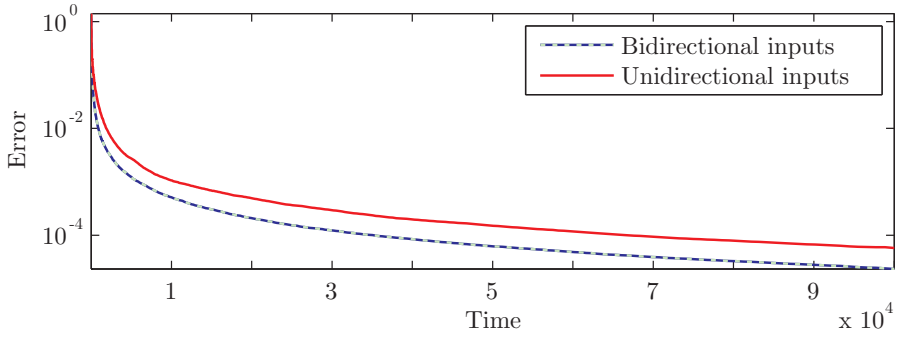

Figure 3.4: Lyapunov function for a continuous-time simulation of $n = 1000$ robots, with $\epsilon \in [0.5, 1.5]$, all initialized to $(1, 1)$ and steered to $(0, 0)$ using the bidirectional control policy (2.10) and the unidirectional control policy (3.1).

ensemble to its goal, but even with unidirectional inputs, the ensemble still successfully asymptotically converges to its goal.

3.3 Discrete-Time Simulations with a Standard Noise Model

To model noise that is natural in a real system, we apply the noise model in Probabilistic Robotics by Thrun et al. [53, Chap. 5.4.2]. This model defines each discrete-time motion as a rotation, a translation, and a second rotation. It uses the four parameters $\alpha_1, \alpha_2, \alpha_3$, and α_4 to weight the correlation of noise between rotation and translation actions. If the desired rotation, translation,

and second rotation are given by $(\delta_{\text{rot1}}, \delta_{\text{trans}}, \delta_{\text{rot2}})$, then the actual actions, after noise is applied, are given by

$$\begin{aligned}
\hat{\delta}_{\text{rot1}} &= \delta_{\text{rot1}} - \text{sample}(\alpha_1 \delta_{\text{rot1}}^2 + \alpha_2 \delta_{\text{trans}}^2) \\
\hat{\delta}_{\text{trans}} &= \delta_{\text{trans}} - \text{sample}(\alpha_3 \delta_{\text{trans}}^2 + \alpha_4 \delta_{\text{rot1}}^2 + \alpha_4 \delta_{\text{rot2}}^2) \\
\hat{\delta}_{\text{rot2}} &= \delta_{\text{rot2}} - \text{sample}(\alpha_1 \delta_{\text{rot2}}^2 + \alpha_2 \delta_{\text{trans}}^2),
\end{aligned} \tag{3.2}$$

where sample(x) generates a random sample from the zero-centered normal distribution with variance x.

An important note for the application of control policies under the noise model is that these are feedback control policies, and the orientation of each robot in the ensemble is constantly being measured. This means that in control policies (2.8), (2.10), and (2.14), the term for the calculated orientation $\theta_i^0 + \epsilon_i t$ should be replaced by a term that is the measured orientation $\theta_i(t)$ in simulations with noise or in hardware experiments.

To test the discrete-time control policy (2.14) with our noise model, we simulated a collection of 120 robots in MATLAB under various levels of noise with both differing and identical values of ϵ. Sample trajectories are shown in Figure 3.5. For each discrete-time simulation, the goal of the ensemble is to move from spelling the word "ROBOTICS" to spelling the word "ILLINOIS", and error is measured as the average distance of one robot to its goal. The source code for all discrete-time MATLAB simulations is in Appendix A.1.2.

3.3.1 Simulating Different ϵ Values with Noise

Simulating with a bounded range of $\epsilon \in [1-\delta, 1+\delta]$, we found that with no noise, the position error of our robot collection converged exponentially to zero error. When the noise model (3.2) was applied to add Gaussian noise in actuation, the error converged to a non-zero value for small values of noise, and diverged for large values of noise, as shown in Figure 3.6.

In this simulation, all values of α are held equal for each level of noise, so for example when the noise is set to 0.1, $\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = 0.1$. This simulation shows that if noise affects both translation and rotation equally, then it is best to have the least amount of noise possible in actuation.

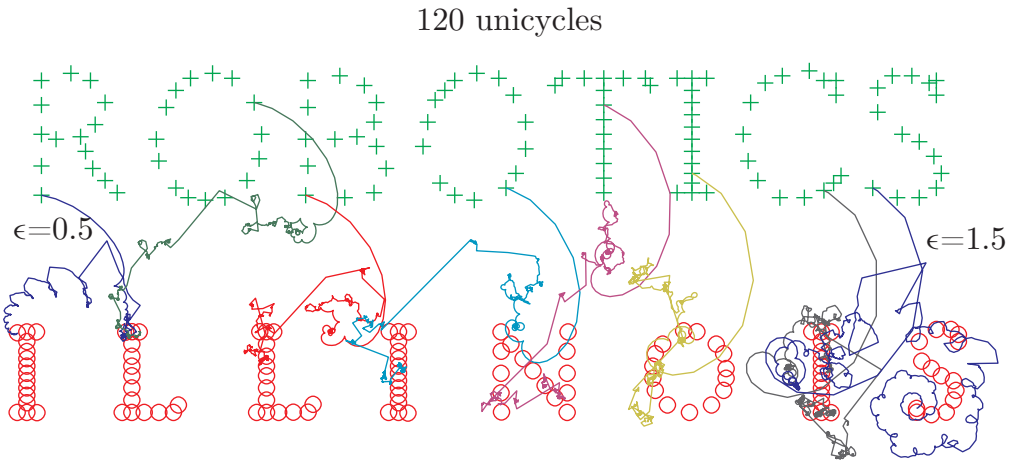

Figure 3.5: Simulation results from applying the control policy (2.14) for 120 robots with unicycle kinematics. Wheel size (ϵ) was evenly distributed from 0.5 to 1.5. The plot shows the starting '+' and ending '∘' positions along with 8 selected state trajectories.

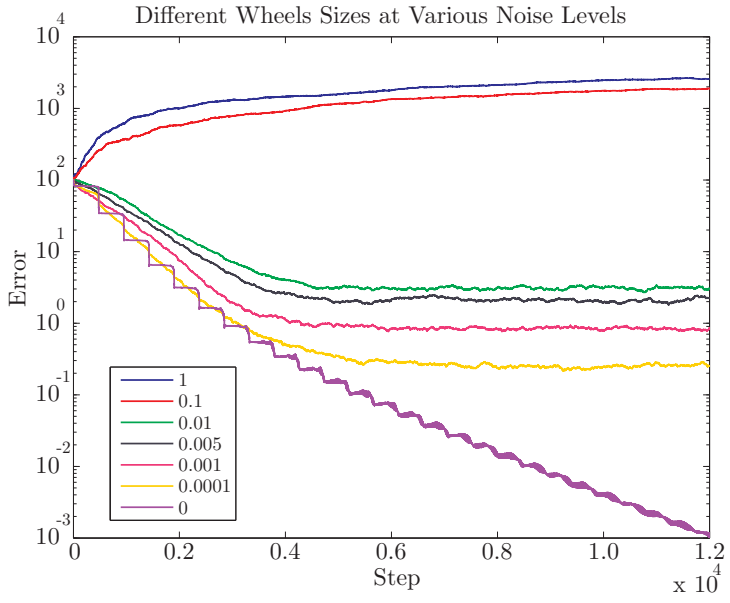

Figure 3.6: Error of a discrete-time, finite collection of 120 robots simulated under a standard noise model (3.2). The convergence of the position error, where $\epsilon \in [0.5, 1.5]$, with different levels of noise parametrized by α; all α are equal.

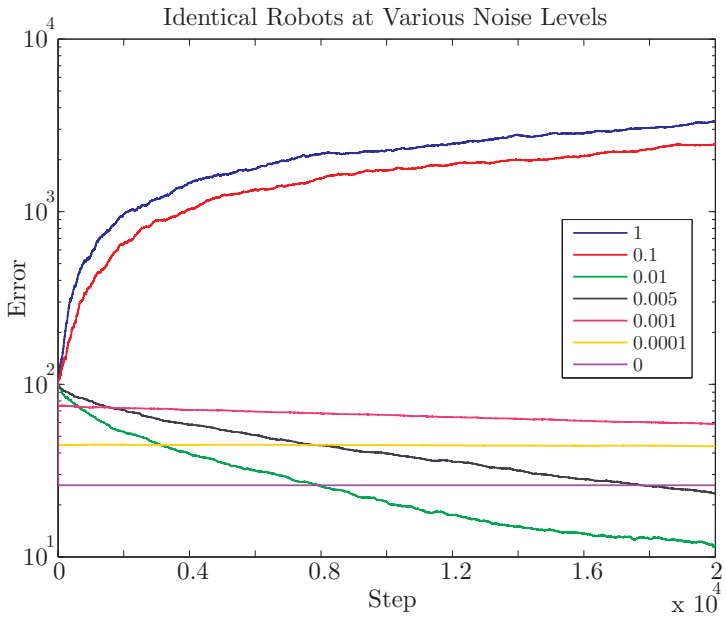

Figure 3.7: Error of a discrete-time, finite collection of 120 robots simulated under a standard noise model (3.2). The convergence of the position error, where all ϵ are set to 1, with different levels of noise parametrized by α; all α are equal.

3.3.2 Identical Robots

In this simulation, all 120 robots are identical, and they attempt to complete the same task as in Section 3.3.1. Interestingly, the best performance is not achieved with a very large or small amount of noise in actuation, but rather the least error is achieved within a specific intermediate range of noise values. Again, all values of α are held equal for each level of noise. Large α values caused the error to diverge, while small α values led to very slow convergence. With no noise at all, the error gets stuck at a constant value and does not continue to decrease. This result is shown in Figure 3.7.

This simulation suggests that for an ensemble completing a task, there is an optimal level of noise such that the ensemble converges to the goal and does so quickly. Too much noise in actuation outstrips the feedback control policy, while too little noise does not generate sufficient new orientations for good selections of $u_1(t)$.

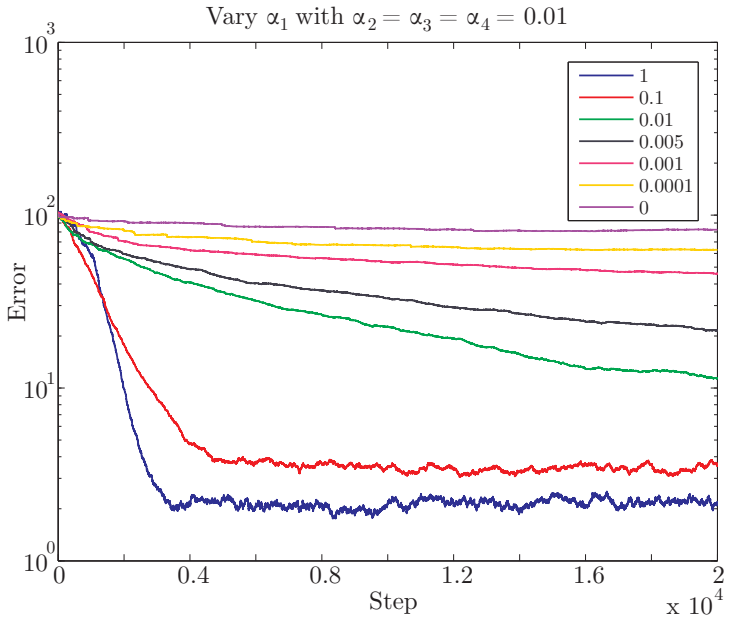

Figure 3.8: Error of a discrete-time, finite collection of 120 robots simulated under a standard noise model (3.2). The convergence of the position error, where all ϵ are set to 1, with different levels of noise parametrized by α. Focusing the noise in the rotation (α_1) improves convergence with identical robots.

3.3.3 Effect of Rotational Noise

Again with 120 identical robots attempting the same task as Sections 3.3.1 and 3.3.2, we held the translational and cross-term noise at 0.01 ($\alpha_2 = \alpha_3 = \alpha_4 = 0.01$. This value was chosen because it performed the best in the identical robot simulation of Section 3.3.2. But in this simulation, we varied the rotational noise parameter, α_1. We found that convergence rate increased proportionally with α_1, up to a limit of approximately $\alpha_1 = 1$, where increasing α_1 beyond 1 did not change performance. This result is shown in Figure 3.8.

These results are interesting because they show that noise is necessary for a finite collection of identical robots to be controllable. Particularly, systems perform the best if they have a minimal amount of noise in translational actuation but a large amount of noise in rotational actuation. This is a subset of a larger class of problems for which noise is beneficial, or even necessary, for stability and control, and it can help inform robotic system designers that if they would like good performance from ensemble control

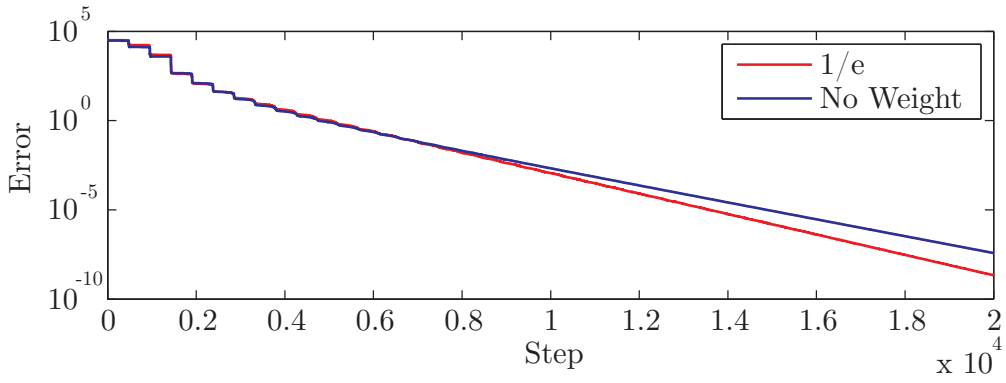

Figure 3.9: Error plots for a discrete-time simulation of 120 robots, with $\epsilon \in [0.5, 1.5]$, comparing control policies (2.14) and (3.3). Weighting by $1/\epsilon$ improves convergence.

methods, it is better to have noisy turning than noisy linear movement.

3.4 Using $1/\epsilon$ to Improve Convergence

The control laws presented in Chapter 2 are globally asymptotically stabilizing, but in practice there are ways to improve performance of ensembles. One method is to modify the control policy to take into account known values of ϵ for each robot and to weight the summed error accordingly. Modifying (2.14) to incorporate this change yields

$$\begin{aligned} k =& \frac{t}{\Delta T} - \mathrm{mod}\,(t, \Delta T) \\ F(k) =& \frac{1}{n} \sum_{i=1}^{n} \frac{1}{\epsilon_i} \left(x_i(k) \cos(\theta_i(k)) + y_i(k) \sin(\theta_i(k)) \right) \\ \Big[u_1(k), u_2(k) \Big] =& \begin{cases} [-F(k), 0] & \text{stage 1} \\ [0, \phi] & \text{stage 2} \end{cases} \end{aligned} \tag{3.3}$$

In (3.3), robots with larger values of ϵ are viewed as closer than they actually are because they can move farther in the same time as a robot with a smaller value of ϵ. Policies (2.14) and (3.3) are compared in Figure 3.9, and it is evident that adding the $1/\epsilon$ weighting improves performance.

3.5 Collision Avoidance with Potential Functions

When steering an ensemble, it may be desirable to avoid obstacles in the workspace. One method of obstacle-avoidance is potential fields, as presented in Principles of Robot Motion [54, Chap. 4]. In this method, robots avoid obstacles by being repelled from them by an artificial potential field. For the ith robot in an ensemble, one possible repulsive field function is:

$$u_i^{\text{rep}}(t) = \begin{cases} \frac{1}{2}\eta cos(\phi_i(t)) \left(\frac{1}{D(x_i(t),y_i(t))} - \frac{1}{Q^\star} \right)^2 & D(x_i(t), y_i(t)) \leq Q^\star \\ 0 & D(x_i(t), y_i(t)) > Q^\star, \end{cases} \tag{3.4}$$

where $D(x_i(t), y_i(t))$ is distance of robot to the closest obstacle, $\phi_i(t)$ is angle from robot to the closest obstacle, $Q^\star$ is the maximum effective distance of the potential field, and η is a gain.

Then the control policy for a finite ensemble (2.10) is modified to incorporate the repulsive field:

$$u_1(t) = -\frac{1}{n} \sum_{i=1}^{n} \left(x_i(t) \cos\left(\theta_i(t)\right) + y_i(t) \sin\left(\theta_i(t)\right) - u_i^{\text{rep}}(t) \right). \tag{3.5}$$

Potential fields have the possibility to get stuck in local minima, as highlighted by Koren and Borenstein [55], so we can no longer guarantee that our ensemble is globally asymptotically stable under (3.5), but in practice, ensembles are able to reach their goals if given more time. Figure 3.10 shows a simulation with an ensemble avoiding two obstacles using potential fields.

3.6 Trajectory Tracking

Ensembles can track trajectories by slowly varying the goal position of each robot over time. Figure 3.11 shows a simulation of six robots following six different trajectories simultaneously, all responding to the same control signal. Trajectory tracking allows the ensemble control framework presented in this document to be combined with high-level motion planning; a motion-planner can be used to generate trajectories for each robot, and then ensemble control can control the ensemble such that each robot tracks its trajectory.

As this chapter details, feedback ensemble control is a framework that can

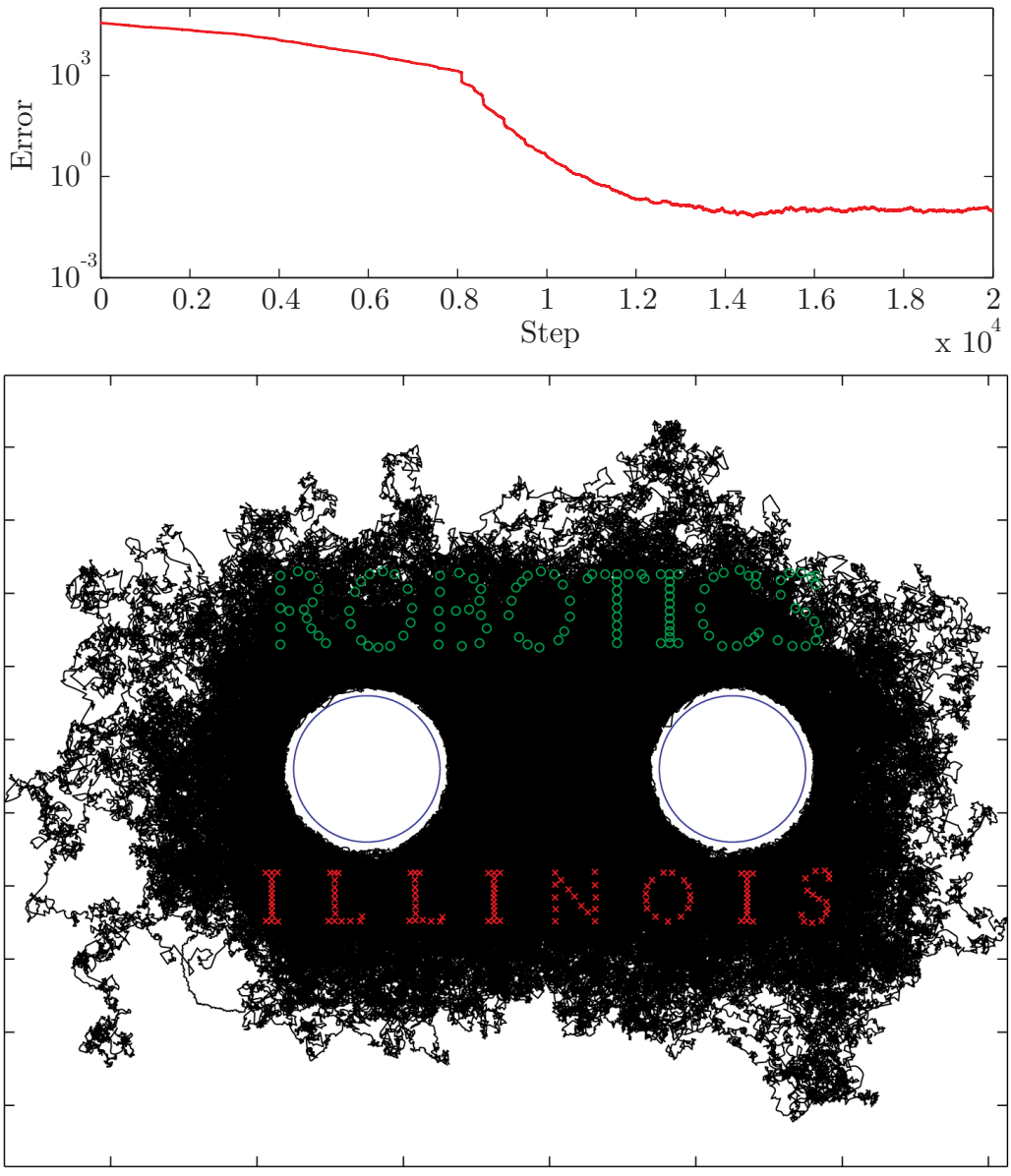

Figure 3.10: Obstacle avoidance with potential fields, using control policy (3.5). The top plot shows the error over time, and the bottom plot shows the paths traced by all 120 robots. In this simulation, $\eta = 20$ and $Q^\star = 30$.

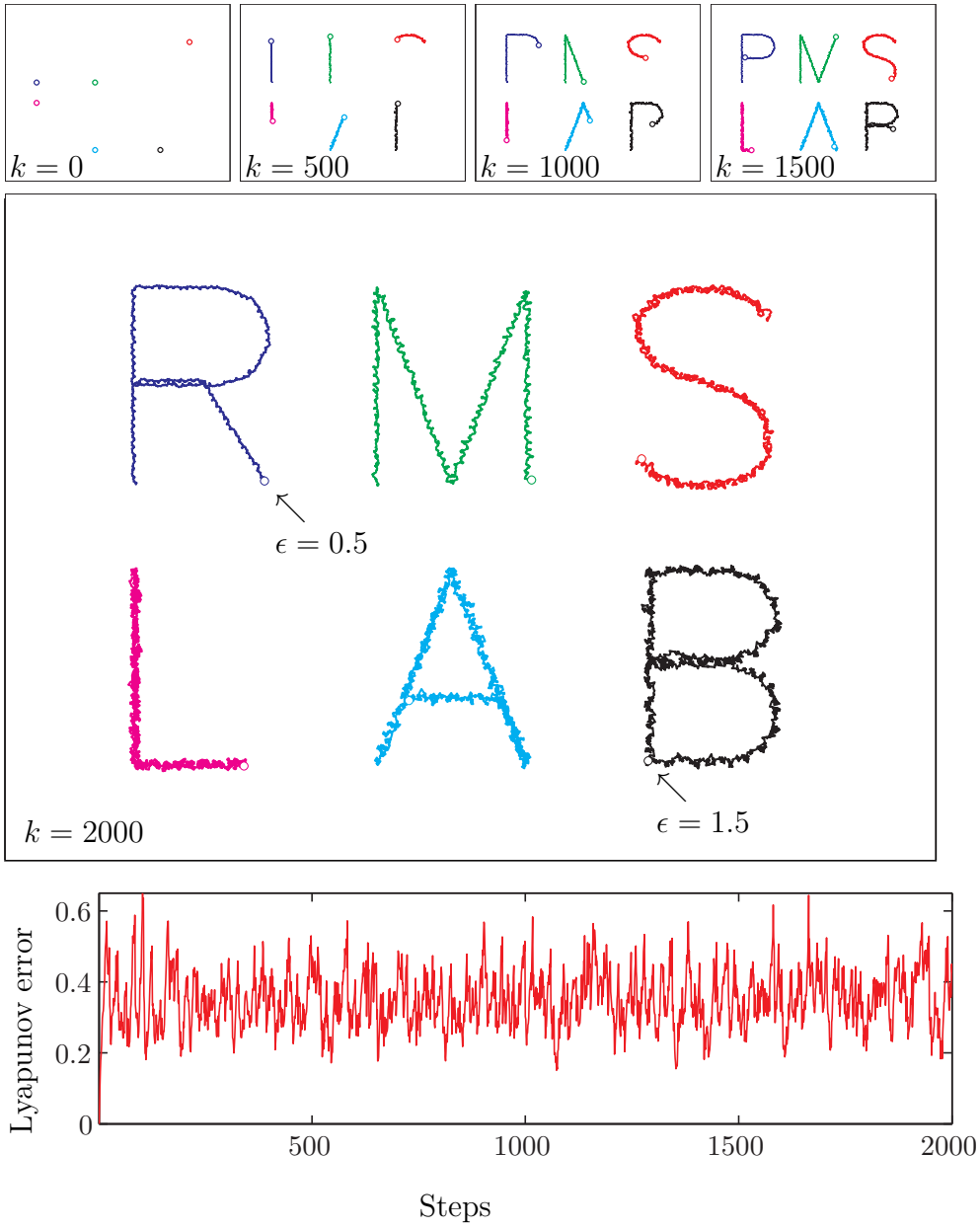

Figure 3.11: Simulation of trajectory-following. Six differential-drive robots with wheel sizes ranging from 0.5 to 1.5 of nominal are steered with a common control signal to follow trajectories that spell out 'RMSLAB'. The top left robot (R) has the smallest wheels while the lower right robot (B) has the largest wheels. The bottom plot shows that the Lyapunov function stabilizes around 0.37.

be extended to meet the requirements of many systems. Ensembles can be controlled even with noise in sensing and actuation and around obstacles as well as along trajectories.

Chapter 4

Hardware Experiments

To verify the feedback ensemble control methods presented in Chapter 2, computer simulations were supplemented with hardware experiments on differential-drive robots, with an optical motion-capture system for position and orientation feedback. Hardware experiments did, in fact, verify that the control policies described in this thesis are effective.

4.1 Differential-Drive Robots

Our differential robots are the Segbots, courtesy of Dan Block [36]. They have two large direct-drive wheels in the back, and a free-wheeling ball caster in the front, as shown in Figure 4.1. The robots have very little on-board intelligence; the motor controllers simply translate linear and angular velocities to voltages to be applied with friction compensation. The robots have no notion of goals or trajectories to follow. In the experiments shown in this paper, we use wheels with diameters in the set $\{102, 108, 127, 152\}$ mm. Trials with identical-size wheels all used 102mm wheels.

4.2 System Overview

A block diagram of our system in Figure 4.2 shows the relevant hardware. The robots are commanded to either move linearly or turn in place in units of encoder ticks. These commands are broadcast over 900MHz radio using an AeroComm 4490 card. The robots have no notion of what wheel size they have or where they are; they simply respond to the global control signals.

Four to five optical tracking dots are fixed to the top of each robot. Position and orientation data for each vehicle are uniquely measured by an 18-camera NaturalPoint Optitrack system with reported sub-millimeter accuracy. A

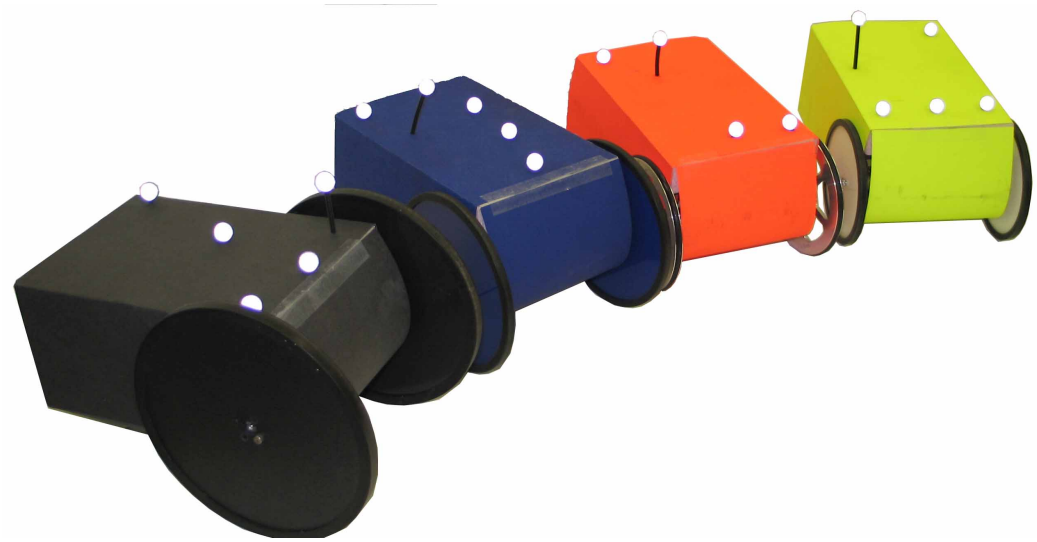

Figure 4.1: Four differential-drive robots with wheel diameters in the set $\{102, 108, 127, 152\}$ mm, courtesy of Dan Block [36]. Each robot receives the same broadcast control signal, but the different wheel sizes scale the commanded linear and angular velocities.

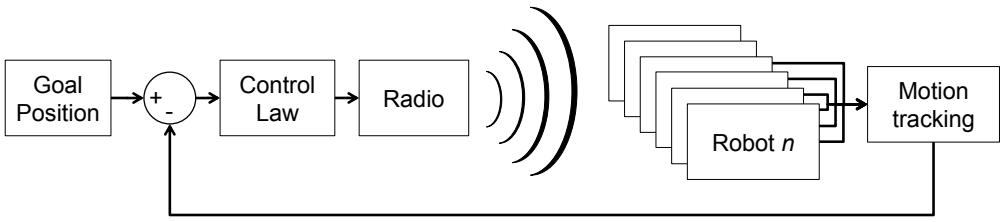

Figure 4.2: Block diagram for steering a multi-robot system with a global signal.

MATLAB program receives feedback from the Optitrack system and computes the control policy (2.14) before sending the global control signals to each robot simultaneously at each discrete time-step.

4.3 Online Calibration

Calibration is not necessary for successful implementation of the controller, but it improves performance. There are very accurate methods to perform calibration of differential-drive robots such as Borenstein's method [56], but they require a separate calibration step. We instead use an online calibration method. While online calibration may not be as accurate, we demonstrate in Section 4.4.2 that calibration of ϵ values is helpful but not critical.

In our hardware experiments, for every translation command $u(k)$, we record beginning and ending positions to calculate d_i, the distance traveled, and update each ϵ_i value according to the following rule:

$$\epsilon_i(k+1) = \epsilon_i(k) + K\frac{|u(k)|}{M}\left(\frac{d_i}{|u(k)|} - \epsilon_i(k)\right).$$

K is the weighting we give new measurements of ϵ, and M is the maximum possible distance we may command the robot to move. For the experiments shown here $K = 0.1$ and $M = 0.7$.

4.4 Experiments

A series of experiments were conducted to show that the derived control policy converges in a real system. Results for unique wheel sizes with online calibration, for unique wheel sizes without online calibration, and for robots with identical wheels are shown. In each experiment, goals are defined in the workspace, and the ensemble controller is left to steer the robots to their goals. Then, the goals are moved, and the controller steers the robots to the new set of goals. Error is measured individually for each robot as its distance to its goal in meters and overall as the sum of all individual errors. The source code for the MATLAB program as well as the program running on the robots is in Appendix A.2.

4.4.1 Unique Wheel Sizes with Online Calibration

In this experiment, each robot is initially assumed to have $\epsilon = 1$, and the actual values of ϵ were learned through online calibration. The robots were successfully commanded from a horizontal line, to a box formation, to a vertical line, and finally to a tight box formation. The results in Figure 4.3 show convergence both in position and in ϵ values. Online calibration requires persistent excitation, so convergence slows as the robots approach their targets.

The error plots show that the robots steadily approach their goals. Contrary to the perfect (no noise in sensing or actuation) theoretic control policy, the robots do actually move away from their goals on occasion. This is caused by a combination of imperfect sensing from the optical tracking system as well as imperfect execution of given commands by the robots (noise in actuation). The formations shown below the error plots are what the optical tracking system sees at each time specified.

4.4.2 Unique Wheel Sizes without Calibration

Surprisingly, it is not necessary to know or to learn the ϵ values. For this entire experiment ϵ was set to 1. Four robots were successfully commanded from a horizontal line to a box formation, and then to a vertical line. For each formation, error converged to less than half a meter, as shown in Figure 4.4. With no calibration, the robots took longer to converge to their goals, but they were still successful.

This experiment shows that even with incorrect ϵ values, our control policy can still control the ensemble. This is primarily because it is a feedback policy, so it constantly adjusts to compensate for errors in the robot positions.

4.4.3 Identical Wheel Sizes

In this experiment, all the robots were fitted with identical wheels, effectively making all the robots identical. Again, the robots were commanded to move from a horizontal line to a box and then to a vertical line formation. The ensemble does not converge as tightly to the goals as in previous experiments, and the first formation change takes longer than usual; however, each robot

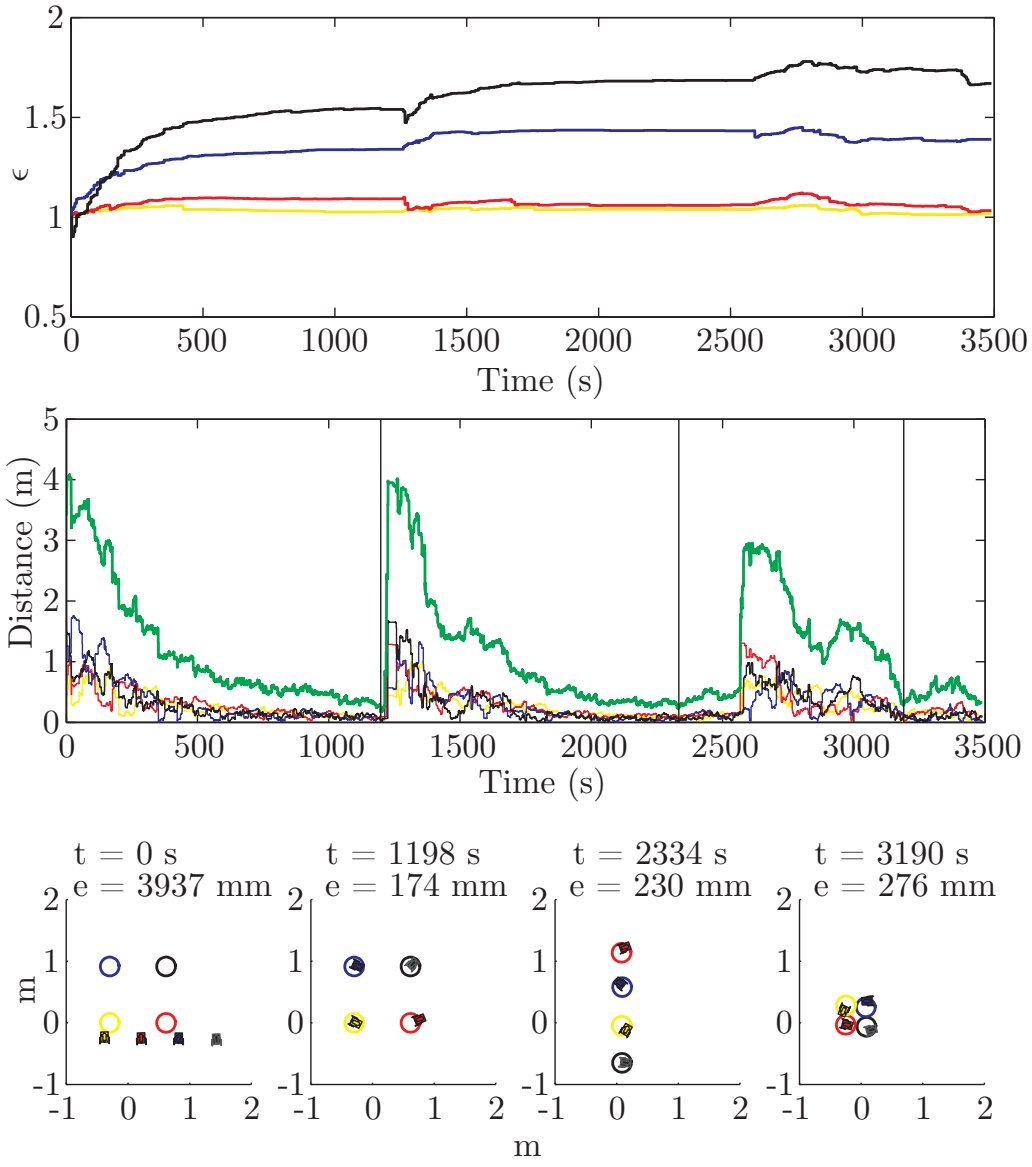

Figure 4.3: Hardware experiment with unique wheel sizes and online calibration. The top plot shows ϵ values estimated by online calibration. The bottom plot shows the summed distance error as the robots were steered through the sequence of formations shown.

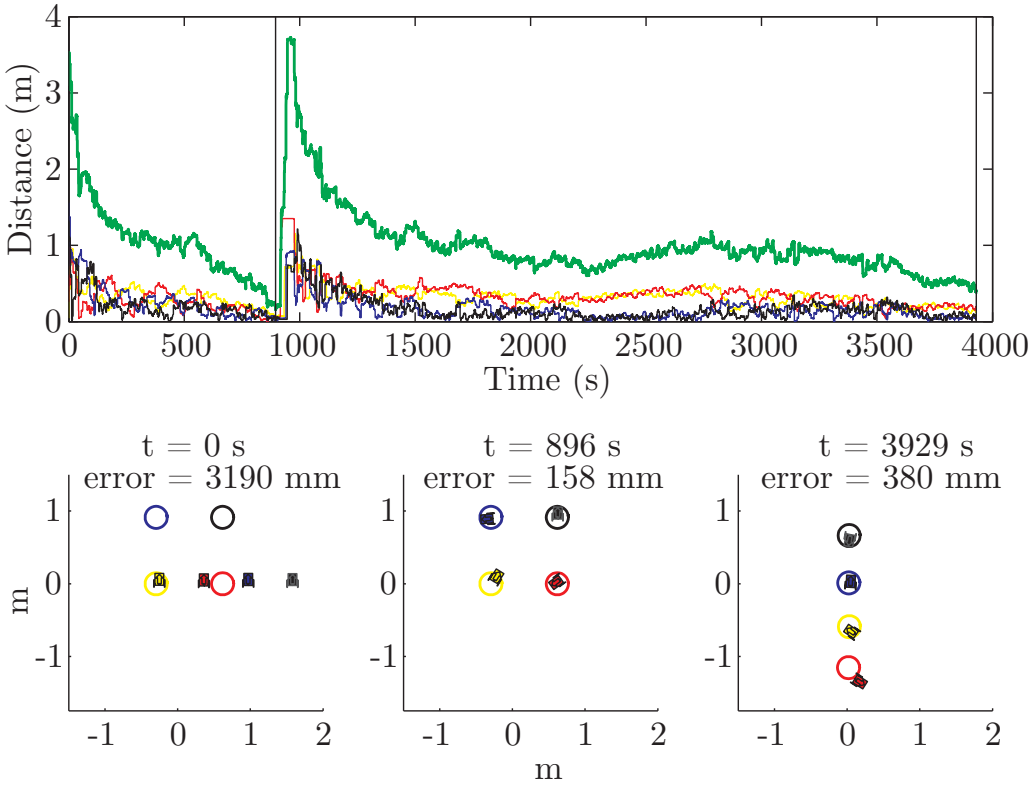

Figure 4.4: Hardware experiment with unique wheel sizes and no calibration. The plot shows the summed distance error as the robots were steered through the sequence of formations shown.

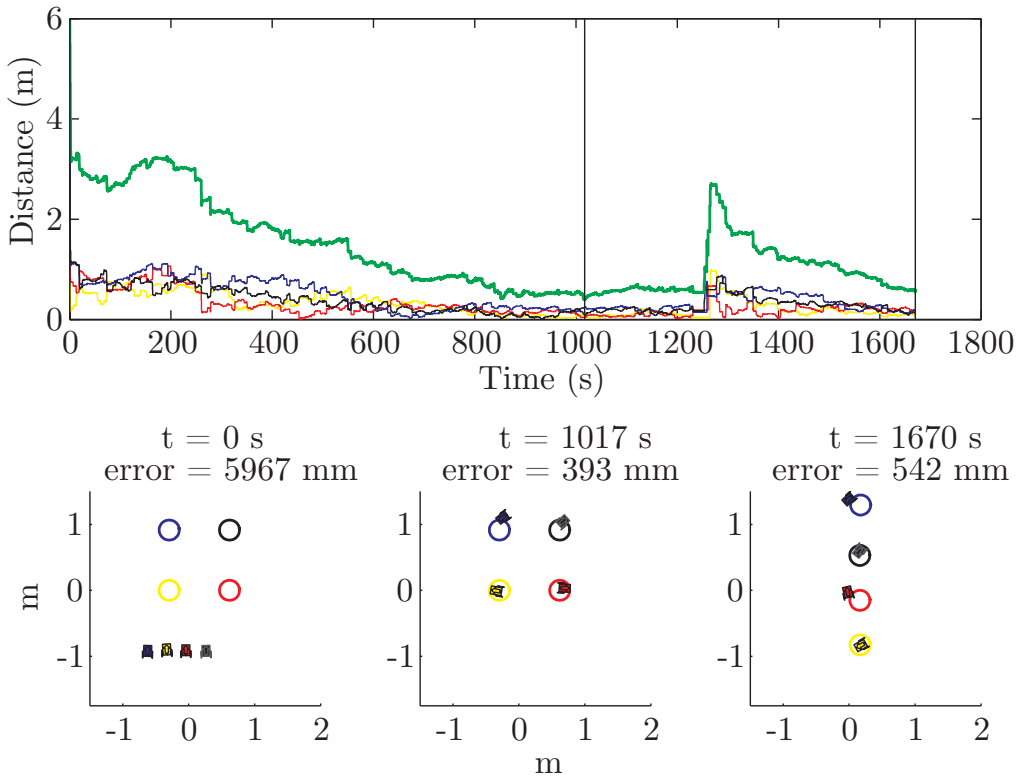

Figure 4.5: Hardware experiment with identical wheel sizes. The plot shows the summed distance error as the robots were steered through the sequence of formations shown.

in the ensemble does get quite close to its goal, and this experiment shows that even an ensemble of identical robots can be successfully steered to a goal. Figure 4.5 shows the experiment.

As with the simulation in Section 3.3.2, an ensemble of identical robots will approach its goal to as near as actuation noise in the system will allow. Coincidentally, it is noise in actuation that allows the ensemble to converge at all. With no noise, an ensemble of ensemble robots would get stuck far from the goal, as shown in Figure 3.7. But a real hardware system, such as the differential-drive robots used in this experiment, has sufficient noise in actuation so that the ensemble can approach its goal. However, it is undesirable to have so much noise in actuation that the motion of the robots is not responsive to the control inputs.

The hardware experiments verify that the control law for a finite ensemble in discrete-time, (2.14), does steer robots to their goals. The error plots show that convergence is asymptotic to zero error. The system is also robust

against external disturbances. In Figure 4.6, the robots were tasked with coming together to one location in the room. Even when we moved the robots far from each other, the system recovered from the perturbation. The experiments conducted with differential-drive robots suggest that our control policies would also be effective on micro- or nano-robot systems with similar unicycle kinematics.

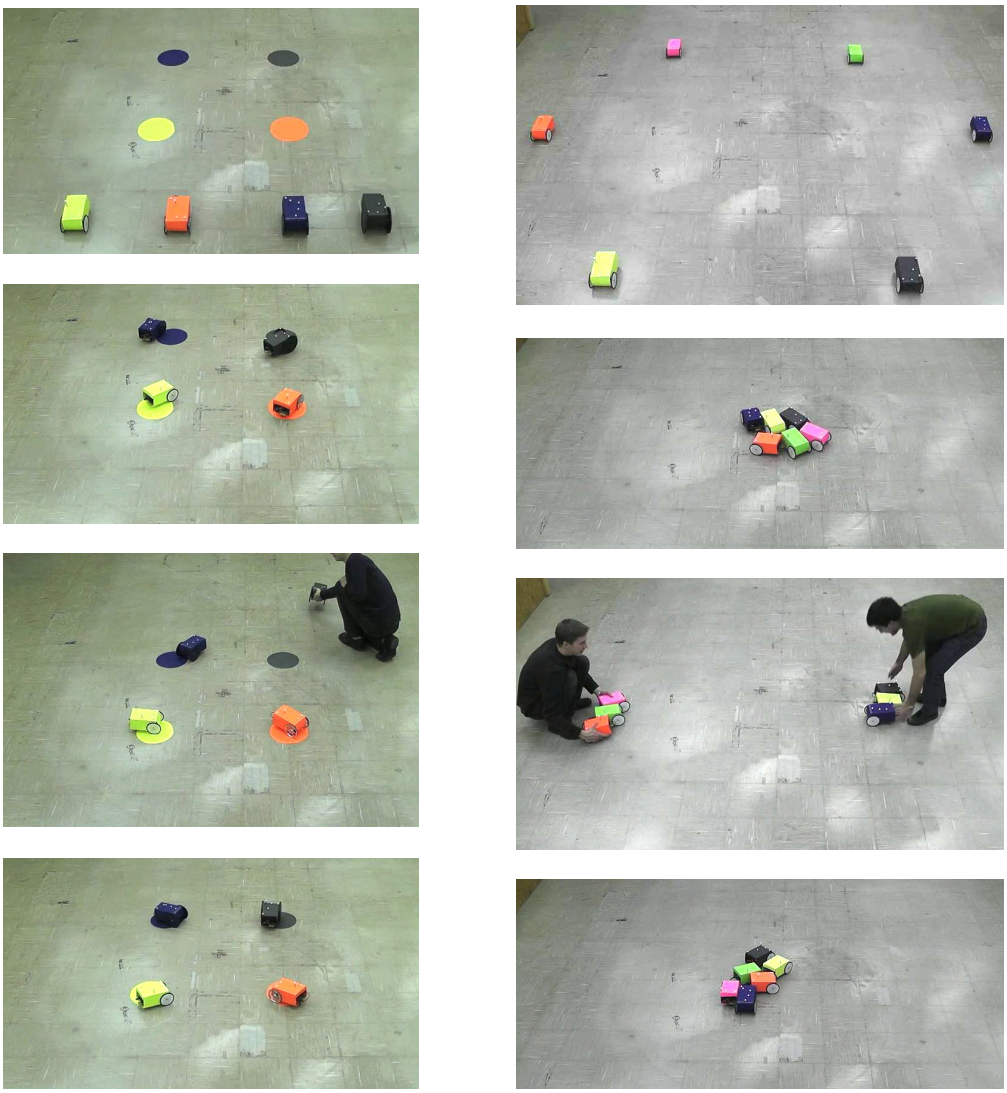

Figure 4.6: Photographs of hardware experiments steering four differential-drive robots with different wheel sizes (left column) and six differential-drive robots with identical wheel sizes (right column). The robots are initialized in a straight line and all receive the same control input from a wireless signal. A motion capture system is used for feedback to steer the four robots to the colored targets and the six robots to rendezvous. In the third frame a disturbance is injected by moving a single robot away from its target (left) and by splitting the ensemble (right).

Chapter 5

Conclusion

This thesis investigates ensembles of unicycles that share a uniform control input. Through Lyapunov analysis, we derived a globally asymptotic stabilizing controller for a continuous-time, infinite ensemble. We extended this controller to finite ensembles of unicycles in continuous and discrete time. In simulation, it was shown that a discrete-time, finite ensemble of unicycles converges asymptotically and rejects disturbances from a standard noise model. Also, obstacle-avoidance and trajectory-following methods were proposed, with accompanying simulations to verify their feasibility. Hardware experiments demonstrated online calibration which learned the unknown parameter for each robot. These experiments led to encouraging results that *(1)* our controller still works when all wheel sizes are wrong and *(2)* if actuation is imperfect, our controller works even when all wheel sizes are the same.

5.1 Applications of Feedback Ensemble Control

The ability to control position enables many tasks. For example, Chapter 4 demonstrates robot gathering using six robots with identical-sized wheels. Robot gathering robustly collects all the robots to one position; to achieve robot gathering, the goal position of each robot is set to the mean position of the ensemble.

Dispersion is the opposite of gathering. To achieve dispersion, the goal position of each robot is set to the mean of the ensemble, but the control policy is set to $u(t) = F(t)$, which will repel robots from each other. Dispersion may be useful for distributing micro- and nano-robots over a substrate. Other tasks include forming subgroups, path-following, and pursuit/avoidance. Each can be implemented by a suitable selection of goal trajectories.

Two applications that are the focus of much micro- and nano-robot research are nano-manipulation and assembly [57, 58]. Nano-manipulation is simply manipulating objects at the nano-scale, and assembly is building structures from smaller components. With the ability to control position and track trajectories, micro- and nano-robots in an ensemble can be used as a manipulator. In Figure 5.1, we show how we are able to move six unicycle robots to assemble a structure from smaller components. The simulation was generated with the Box2D game environment, which has realistic collision detection and physics. The source code for this simulation is in Appendix A.1.3. One advantage of using an ensemble control method over other methods such as using an atomic force microscope tip is that with ensemble methods, multiple robots, and thus multiple objects, can be manipulated at once.

5.2 Future Work

This work shows that an ensemble of nonholonomic unicycles with uniform inputs to all robots can be regulated to arbitrary positions, reject disturbances from a standard noise model, and converge to goals with global asymptotic stability. This work may be particularly relevant to systems of micro- and nano-robots, which are often constrained to uniform inputs. The control policies described focus on unicycle kinematics, but future work could adapt these policies to other types of robots with different kinematics or dynamics.

Future work could seek to add guarantees to using an ensemble as a set of manipulators — when can we be sure we will not lose hold of the object? When can we guarantee that we will place the object within a certain precision? Also, it may be beneficial to combine high-level motion-planning principles with low-level ensemble trajectory tracking to move in more intelligent paths. The policies presented are not optimal, and adding optimality to the ensemble control policies could improve the execution speed of tasks.

The ensemble control policies in this thesis are best applied to systems where robots do not have on-board intelligence or controlling each robot individually is infeasible; micro- and nano-robotic systems often have these characteristics, and as small robots become more prolific, there will be more applications for the control framework presented here.

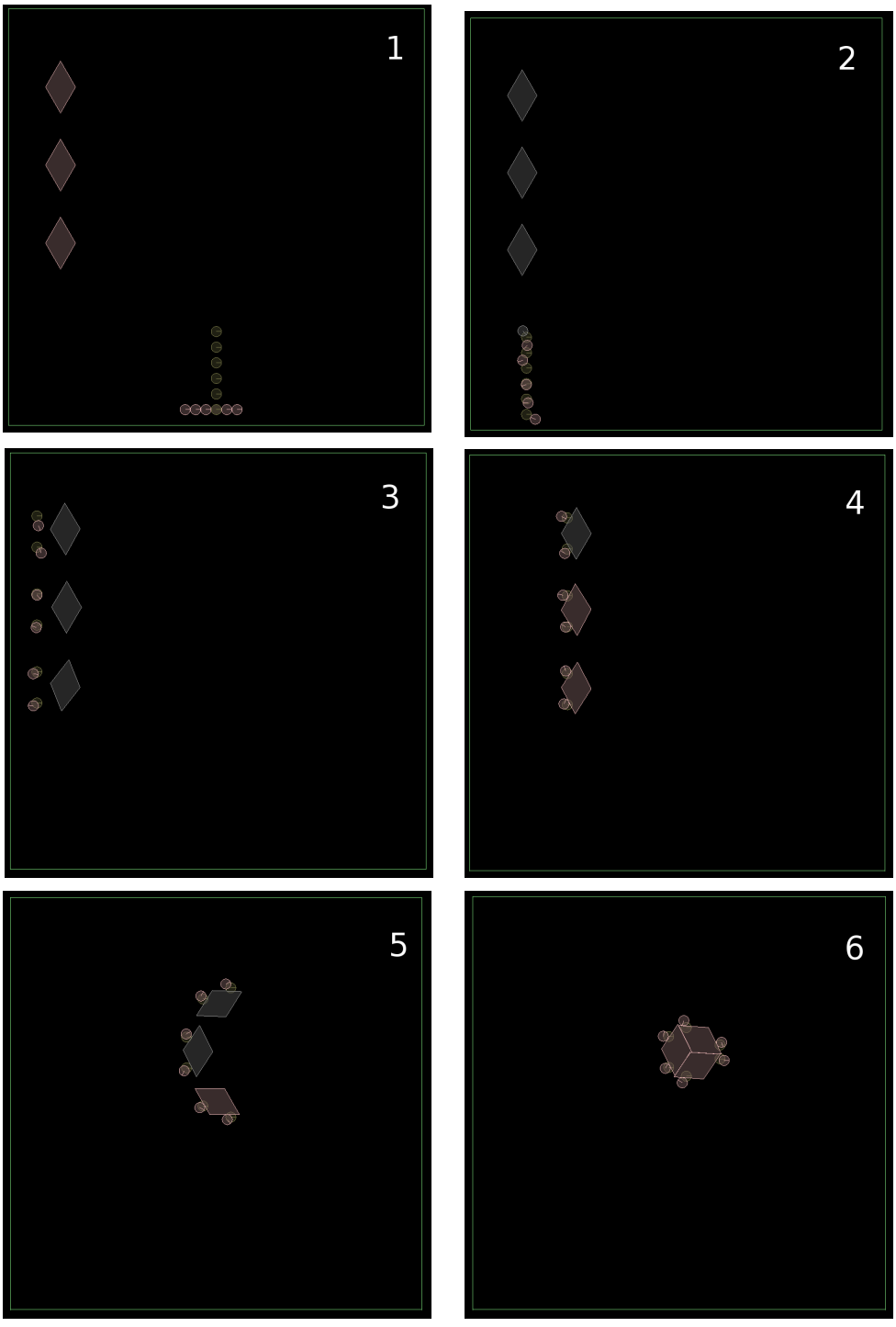

Figure 5.1: Images from an assembly and manipulation simulation. Robots are red circles, and objects are diamonds. The goal trajectories are represented by the yellow circles. The robots approach the objects, push them into position, and then orient them to assemble the final object.

Appendix A

Source Code

A.1 Simulations

A.1.1 Continuous-Time Matlab Simulations

```
%  simUnicyclesCirclePt.m
%
%  Cem Onyuksel and Aaron Becker 2012
%
%  run simulations of unicycle in continuous-time

% in the mathematica file, you see that control authority over the
% continuum drops as theta grows.  (the continuum theta is
% being wrapped around and around the unit circle.
% A better control policy would be to flip u2 every pi deg.
format compact
saveState = true;

global eps
global invEps
global unidirectional

%clc
nRob = 50;
delta = .5;
eps = linspace(1-delta, 1+delta, nRob)';
invEps = false;
unidirectional = false;
```

```
%%%INITIALIZE TO START AT {1,1,0}
%state0 = [ones(nRob,1);ones(nRob,1);zeros(nRob,1);];

%%% INITIALIZE  TO START IN CIRCLE
sp = (1:nRob)*2*pi/nRob;
state0 = [cos(sp)';sin(sp)';sp'];



tic
display('Starting Simulation')
options = odeset('RelTol',1e-7,'AbsTol',1e-7);
tms = logspace(0,5,1000);
[T,Y] = ode45(@unicycleMinAveCont,tms,state0,options);
invEps = true;
[T2,Y2] = ode45(@unicycleMinAveCont,tms,state0,options);
%[T,Y] = ode45(@unicycleMinAveCont,(0:500:10000),state0,options);
%[T,Y] = ode45(@unicycleMinAveCont,[0,100],state0,options);
%[T,Y] = ode45(@unicycleMinAveCont,[0,50],state0,options);
toc
%  50 robots, 1000 seconds sim time, 110 seconds compute

%   Plotting the columns of the returned array Y versus T
% shows the solution
if saveState
    replines = round(linspace(1, nRob,5));
    display(['plotting lines ',num2str(replines)]);
    figure(1)%%%%%%%%%%%%%%%%%%%%%%%%%%  XY PLOT %%%%%%%%%%%%%%%
    clf

    plot(Y(1,1:nRob),Y(1,nRob+1:2*nRob),'b+',Y(end,1:nRob), ...
        Y(end,nRob+1:2*nRob),'ro')
    hold on
    nFreeze = 5;
    FreezePts = round(linspace(1,numel(T),nFreeze));

    plot(Y(:,replines),Y(:,nRob+replines))
    title(['XY plot for n = ',num2str(nRob),' unicycles'])
    axis equal
```

```
    xlabel('x')
    ylabel('y')
    text(Y(1,1),Y(1,nRob+1),['\epsilon=',num2str(min(eps))])
    text(Y(1,nRob),Y(1,2*nRob),['\epsilon=',num2str(max(eps))])

    figure(2)%%%%%%%%%%%%%%%%%%%%%%%%%  STATE/TIME PLOT %%%%%%%%
    clf

    plot(T,Y(:,replines),'-')
    hold on
    plot(T,Y(:,nRob+replines),'--')

    xlabel('time')
    ylabel('state values')
    title(['State Evolution for n = ',num2str(nRob),' unicycles'])
end

figure(3)%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% LYAPUNOV FUNC VALUES %%%%%
clf
Yt = Y';
errorD = sum((Y(:,1:nRob)'.^2+Y(:,nRob+1:2*nRob)'.^2).^.5)'./nRob;

Yt2 = Y2';
errorD2 = sum((Y2(:,1:nRob)'.^2+ ...
    Y2(:,nRob+1:2*nRob)'.^2).^.5)'./nRob;

%semilogy(T,sum(Y(:,1:2*nRob)'.^2)'./nRob,'r')
semilogy(T,errorD,'r'), hold on
semilogy(T2,errorD2,'b')

xlabel('time')
ylabel('Error')
title(['Lyapunov function V for n = ',num2str(nRob),' unicycles'])
set(gca,'Ytick',[1e-6,1e-4,1e-2,1e-0])
```

```
% unicycleMinAveCont.m
%
% Aaron Becker and Cem Onyuksel 2012
%
% Simulate in continuous time, a finite ensemble of kinematic
% unicycles with unique parameters that scale their
% linear and angular velocity

function dy = unicycleMinAveCont(t,y)
% UNICYCLEMINAVECONT evolve the system one timestep.
% t = time, y = state,
% invEps adds a 1/eps scaling to F if true, unidirectional
% makes the control input unidirectional if true

%v1p2 uses structure [x1,..,xn,y1,...,yn,h1,...,hn]
global eps
global invEps
global unidirectional

% display('In unicycleMinAveCont')
% display(y)
% display(eps)

dy = zeros(size(y));    % a column vector
unicycles = numel(y)/3;



%u2 = 1; %constant turning
u2 = cos(t.^.5);

%u2 = cos(t/max(eps));
%u2 = cos(t/max(eps));%periodic turning

if invEps
    %  x * cos(theta)  +  y *sin(theta)
    F = sum((eps.^-1).*(y(1:unicycles).* ...
        cos(y(2*unicycles+1:end)) + ...
        y(unicycles+1:2*unicycles).* ...
```

```
        sin(y(2*unicycles+1:end)) ));
else
    F = sum((y(1:unicycles).* ...
        cos(y(2*unicycles+1:end)) + ...
        y(unicycles+1:2*unicycles).* ...
        sin(y(2*unicycles+1:end)) ));
end

if unidirectional
   u1 = max(0,-F/unicycles);
else
    u1 = -F/unicycles;
end

dy(1:unicycles)             = u1*eps.*cos(y(2*unicycles+1:end));
dy(unicycles+1:2*unicycles) = u1*eps.*sin(y(2*unicycles+1:end));
dy(2*unicycles+1:end)       = u2*eps;
```

A.1.2 Discrete-Time MATLAB SIMULATIONS

```
function simDT()
% This code simulates N unicycles in discrete time moving from
% start to end configurations.  End configuration can be a vector
% parameterized by time for a moving goal or path following.
%
% N is an integer number of robots.
% start is an Nx3 matrix of robot start positions
% goal is a Nx3xT matrix of robot goal positions, where T is the
% time dimension.  Note that we can not control orientation, so
% the third element of the third dimension is ignored in the
% control law.
%
% Aaron Becker, March 1, 2012
% Cem Onyuksel April 12, 2012
%
%%%%%%%%%%%%%%%%%%%%%%%%

% select which simulations to run
```

```
trajectory = false;
varynoise = false;
varynoiseIdentical = false;
varya1Identical = false;
unidirectional = false;
obsavoid = true;
epsilontest = false;

tic

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Effects of 1/eps
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if epsilontest
    % set up the simulation
    timesteps = 20000;
    [start, goals] = roboticsIL(timesteps);
    N = size(start,1); % number of robots
    T = 1:size(goals,3); % time
    S = zeros(size(goals)); % state of robots
    delta = 0.5; % spread of ensemble wheel sizes
    eps = linspace(1-delta, 1+delta, N)'; % ensemble parameter
    PN = 0.000; % noise levels

    % initialize
    S(:,:,1) = start;
    % iterate!
    for t = 2:length(T)
        S(:,:,t) = ApplyControlLaw(S(:,:,t-1),goals(:,:,t),...
            eps,PN,PN,PN,PN,true);
    end

    % plot error with 1/eps
    figure(11);
    x = zeros(N,length(T));
    y = zeros(N,length(T));
```

```
    x(:,:) = (S(:,1,:)-goals(:,1,:)).^2;
    y(:,:) = (S(:,2,:)-goals(:,2,:)).^2;
    error = sum(x + y)./N;
    subplot(2,1,1);
    semilogy(T,error,'Color','r','LineWidth',1);
    hold on

    S(:,:,1) = start;
    % iterate!
    for t = 2:length(T)
        S(:,:,t) = ApplyControlLaw(S(:,:,t-1),goals(:,:,t),eps,...
            PN,PN,PN,PN,false);
    end

    % plot no 1/eps error
    x = zeros(N,length(T));
    y = zeros(N,length(T));
    x(:,:) = (S(:,1,:)-goals(:,1,:)).^2;
    y(:,:) = (S(:,2,:)-goals(:,2,:)).^2;
    error = sum(x + y)./N;
    semilogy(T,error,'Color','b','LineWidth',1);
    hold off
    xlabel('Step')
    ylabel('Error')
    title('Effects of 1/\epsilon')
    legend({'1/\epsilon','No Weighting'},'Location','NorthEast')
    axis([0 timesteps 10^-10 10^5])

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Obstacle Avoidance
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if obsavoid
    % set up the simulation
    timesteps = 20000;
```

```
[start, goals] = roboticsIL(timesteps);
N = size(start,1); % number of robots
T = 1:size(goals,3); % time
S = zeros(size(goals)); % state of robots
delta = 0.5; % spread of ensemble wheel sizes
eps = linspace(1-delta, 1+delta, N)'; % ensemble parameter
PN = 0.0001; % noise levels
% obstacles of form [x,y,radius]
%obs = [75,120,50;200,120,50;325,120,50];
obs = [75,120,50;325,120,50]; % obstacles of form [x,y,radius]


% initialize
S(:,:,1) = start;
% iterate!
for t = 2:length(T)
    S(:,:,t) = ApplyObstacleControlLaw(S(:,:,t-1),...
        goals(:,:,t),obs,eps,PN,PN,PN,PN);
end

% plot error
figure(9);
x = zeros(N,length(T));
y = zeros(N,length(T));
x(:,:) = (S(:,1,:)-goals(:,1,:)).^2;
y(:,:) = (S(:,2,:)-goals(:,2,:)).^2;
error = sum(x + y)./N;
subplot(2,1,1);
semilogy(T,error,'Color','r','LineWidth',1);
xlabel('Step')
ylabel('Error')
title('Obstacle Avoidance')
axis([0 timesteps 10^-3 10^5])
set(gca,'Ytick',[1e-3,1e0,1e3])

% plot paths
figure(10);
for i=1:N
```

```
        plot(reshape(S(i,1,:),1,timesteps),-reshape(S(i,2,:),1,...
            timesteps), 'k','MarkerSize',1);
        hold on; axis equal;
    end

    % plot start and end configurations

    plot(start(:,1,1),-start(:,2,1),'go','MarkerSize',3);
    plot(S(:,1,end),-S(:,2,end),'rx','MarkerSize',3);
    % plot obstacles
    for i=1:size(obs,1)
        p = linspace(0,2*pi,1000);
        plot(obs(i,3)*cos(p)+obs(i,1),...
            -obs(i,3)*sin(p)-obs(i,2),'b');
    end
    hold off;
end


%%%%%%%%%%%%%%%%%%%%
% TRAJECTORY FOLLOWING
%%%%%%%%%%%%%%%%%%%%

if trajectory
    % set up the simulation
    timesteps = 10000;
    N = 3; % number of robots

    % trajectories
    s1 = [100*cos(linspace(0,pi,timesteps))', ...
        100*sin(linspace(0,pi,timesteps))',zeros(timesteps,1)];
    s2 = s1 + 50;
    s3 = [linspace(0,100,timesteps)',zeros(timesteps,1),...
        zeros(timesteps,1)];
    goals = zeros(N,3,timesteps);
    goals(1,:,:) = s1';
    goals(2,:,:) = s2';
    goals(3,:,:) = s3';
```

```
start = [s1(1,:);s2(1,:);s3(1,:)];

T = 1:size(goals,3); % time
S = zeros(size(goals)); % state of robots
delta = 0.5; % spread of ensemble wheel sizes
eps = linspace(1-delta, 1+delta, N)'; % ensemble parameter
PN = 0.0001; % noise levels

% initialize
S(:,:,1) = start;
% iterate!
for t = 2:length(T)
    S(:,:,t) = ApplyControlLaw(S(:,:,t-1),goals(:,:,t),eps,...
        PN,PN,PN,PN,true);
end

% plot trajectories
figure(1)
for j = 1:N
    x = zeros(timesteps,1);
    y = zeros(timesteps,1);
    x(:) = S(j,1,:);
    y(:) = S(j,2,:);
    plot(x,y,'.'); hold on
    plot(x(1),y(1),'go');
    plot(x(end),y(end),'rx');
end
axis equal; hold off
title('Trajectory Following');

% plot error
figure(2);
x = zeros(N,length(T));
y = zeros(N,length(T));
x(:,:) = (S(:,1,:)-goals(:,1,:)).^2;
y(:,:) = (S(:,2,:)-goals(:,2,:)).^2;
error = sum(x + y)./N;
plot(T,error,'Color','r','LineWidth',1);
```

```
    title('Trajectory Following Error');
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Bidirectional vs Unidirectional Inputs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if unidirectional
    % set up the simulation
    timesteps = 20000;
    [start, goals] = roboticsIL(timesteps);
    N = size(start,1); % number of robots
    T = 1:size(goals,3); % time
    S = zeros(size(goals)); % state of robots
    delta = 0.5; % spread of ensemble wheel sizes
    eps = linspace(1-delta, 1+delta, N)'; % ensemble parameter
    PN = 0.000; % noise levels

    % initialize bidirectional
    S(:,:,1) = start;
    % iterate!
    for t = 2:length(T)
        S(:,:,t) = ApplyControlLaw(S(:,:,t-1),goals(:,:,t),eps,...
            PN,PN,PN,PN,true);
    end

    % plot error
    figure(3);
    x = zeros(N,length(T));
    y = zeros(N,length(T));
    x(:,:) = (S(:,1,:)-goals(:,1,:)).^2;
    y(:,:) = (S(:,2,:)-goals(:,2,:)).^2;
    error = sum(x + y)./N;
    semilogy(T,error,'Color','r','LineWidth',1);
    hold on

    % unidirectional
    S(:,:,1) = start;
```

```
    % iterate!
    for t = 2:length(T)
        S(:,:,t) = ApplyControlLawUni(S(:,:,t-1),goals(:,:,t),...
            eps,PN,PN,PN,PN);
    end

    % plot unidirectional error
    x = zeros(N,length(T));
    y = zeros(N,length(T));
    x(:,:) = (S(:,1,:)-goals(:,1,:)).^2;
    y(:,:) = (S(:,2,:)-goals(:,2,:)).^2;
    error = sum(x + y)./N;
    semilogy(T,error,'Color','b','LineWidth',1);
    hold off
    xlabel('Step')
    ylabel('Error')
    title('Bidirectional vs Unidirectional inputs')
    legend({'bidirectional','Unidirectional'},'Location',...
        'NorthEastOutside')

    % plot start and end configurations
    figure(4);
    plot(start(:,1,1),-start(:,2,1),'go'); hold on; axis equal;
    plot(S(:,1,end),-S(:,2,end),'rx');
    hold off;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Vary Noise with different wheel sizes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if varynoise
    % system parameters and variables
    timesteps = 10000;
    [start, goals] = roboticsIL(timesteps);
    N = size(start,1); % number of robots
    T = 1:size(goals,3); % time
    S = zeros(size(goals)); % state of robots
```

```
    delta = 0.5; % spread of ensemble wheel sizes
    eps = linspace(1-delta, 1+delta, N)'; % ensemble parameter
    noise=[1,0.1,0.01,0.005,0.001,0.0001,0]; % noise levels

    % plotting parameters
    colors = distinguishable_colors(length(noise));%['b','g',...
        'r','m','c','y','k'];
    l = cell(length(noise),1);

    % show start and end configurations
%     figure(5);
%     plot(start(:,1,1),-start(:,2,1),'go'); hold on; axis equal;
%     plot(goals(:,1,end),-goals(:,2,end),'rx');
%     hold off;

    % plot noise levels for different robots
    for j = 1:length(noise)

        PN = noise(j);
        display(['Simulating for Noise level: ' num2str(PN)]);

        % initialize robots
        S(:,:,1) = start;
        % iterate!
        for t = 2:length(T)
            S(:,:,t) = ApplyControlLaw(S(:,:,t-1),goals(:,:,t),...
                eps,PN,PN,PN,PN,true);
        end

        % plot error
        figure(6);
        x = zeros(N,length(T));
        y = zeros(N,length(T));
        x(:,:) = (S(:,1,:)-goals(:,1,:)).^2;
        y(:,:) = (S(:,2,:)-goals(:,2,:)).^2;
        error = sum(x + y)./N;
        semilogy(T,error,'Color',colors(j,:),'LineWidth',1);
        hold on
```

```
        l{j} = num2str(noise(j));

    end
    hold off
    xlabel('Step')
    ylabel('Error')
    title('Different Wheels Sizes at Various Noise Levels')
    legend(l,'Location','NorthEastOutside')
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Vary Noise with identical robots
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if varynoiseIdentical
    % system parameters and variables
    timesteps = 10000;
    [start, goals] = roboticsIL(timesteps);
    N = size(start,1); % number of robots
    T = 1:size(goals,3); % time
    S = zeros(size(goals)); % state of robots
    delta = 0; % spread of ensemble wheel sizes
    eps = linspace(1-delta, 1+delta, N)'; % ensemble parameter
    noise=[1,0.1,0.01,0.005,0.001,0.0001,0]; % noise levels

    % plotting parameters
    colors = distinguishable_colors(length(noise));
    l = cell(length(noise),1);

    % show start and end configurations
%     figure(5);
%     plot(start(:,1,1),-start(:,2,1),'go'); hold on; axis equal;
%     plot(goals(:,1,end),-goals(:,2,end),'rx');
%     hold off;

    % plot noise levels for different robots
    for j = 1:length(noise)
```

```
        PN = noise(j);
        display(['Simulating for Noise level: ' num2str(PN)]);

        % initialize robots
        S(:,:,1) = start;
        % iterate!
        for t = 2:length(T)
            S(:,:,t) = ApplyControlLaw(S(:,:,t-1),goals(:,:,t),...
                eps,PN,PN,PN,PN,true);
        end

        % plot error
        figure(7);
        x = zeros(N,length(T));
        y = zeros(N,length(T));
        x(:,:) = (S(:,1,:)-goals(:,1,:)).^2;
        y(:,:) = (S(:,2,:)-goals(:,2,:)).^2;
        error = sum(x + y)./N;
        semilogy(T,error,'Color',colors(j,:),'LineWidth',1);
        hold on
        l{j} = num2str(noise(j));

    end
    hold off
    xlabel('Step')
    ylabel('Error')
    title('Identical Robots at Various Noise Levels')
    legend(l,'Location','NorthEastOutside')
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Vary \alpha_1 (rotational noise) w/ identical robots
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if varya1Identical
    % Plot change in rotation error vs translation error for
    % identical robots
    timesteps = 10000;
```

```
[start, goals] = roboticsIL(timesteps);
N = size(start,1); % number of robots
T = 1:size(goals,3); % time
S = zeros(size(goals)); % state of robots
delta = 0; % spread of ensemble wheel sizes
eps = linspace(1-delta, 1+delta, N)'; % ensemble parameter
a1s=[1,0.1,0.01,0.005,0];%0.01,0.005,0.001,0.0001,0]; % noise
PN = 0.001;

% plotting parameters
colors = distinguishable_colors(length(a1s));%['b','g','r',...
    'm','c','y','k'];
l = cell(length(a1s),1);

for j = 1:length(a1s)

    a1 = a1s(j);
    display(['Simulating for a1 = ' num2str(a1)]);
    % initialize robots
    S(:,:,1) = start;
    % iterate!
    for t = 2:length(T)
        S(:,:,t) = ApplyControlLaw(S(:,:,t-1),goals(:,:,t),...
            eps,a1,PN,PN,PN,true);
    end

    % plot error
    figure(8);
    x = zeros(N,length(T));
    y = zeros(N,length(T));
    x(:,:) = (S(:,1,:)-goals(:,1,:)).^2;
    y(:,:) = (S(:,2,:)-goals(:,2,:)).^2;
    error = sum(x + y)./N;
    semilogy(T,error,'Color',colors(j,:),'LineWidth',1);
    hold on
    l{j} = num2str(a1s(j));
end
hold off
```

```
    xlabel('Step')
    ylabel('Error')
    title({['Vary \alpha_1 with \alpha_2 = \alpha_3 = ...
        \alpha_4 = ' num2str(PN)]; 'with identical robots'})
    legend(l,'Location','NorthEastOutside')
end

toc

end

function S = ApplyControlLaw(S,G,eps,a1,a2,a3,a4, invEps)
    % Control Law with noise
    % Each action consists of a turn, a move, and another turn
    % We have two actions: turn and move forward
    % a1: rotation->rotation
    % a2: translation->rotation
    % a3: translation->translation
    % a4: rotation->translation
    % invEps: whether to add the 1/eps term or not
    N = size(S,1); % number of robots

    % First action: turn, assume we want to try to turn pi/2
    r1 = pi/2; % how much we want to turn for the first turn
    %trans = 0; % no translation or 2nd rotation
    %r2 = 0;

    % apply r1 with noise
    %u2 = r1*ones(N,1) - sample(a1*r1^2 + a2*trans^2,N);
    u2 = r1*ones(N,1) - sqrt(a1*r1^2)*randn(N,1);
    S(:,3) = S(:,3) + eps.*u2; % theta
    % apply trans with noise
    %u1 = trans*ones(N,1) - sample(a3*trans^2 + a4*r1^2 + ...
    % a4*r2^2,N);
    u1 = -sqrt(a4*r1^2)*randn(N,1);
    S(:,1) = S(:,1) + u1.*eps.*cos(S(:,3)); % x
    S(:,2) = S(:,2) + u1.*eps.*sin(S(:,3)); % y
    % apply r2 with noise
```

```
    %u3 = r2 - sample(a1*r2^2 + a2*trans^2, N);
    %S(:,3) = S(:,3) + eps.*u3;

    % Second action: Translation
    % our requested translation is the amount that minimizes
    % the average error, capped at magnitude 20.  negative
    % because error is negative
    % trans = -sum(eps*(x*cos(theta) + y*sin(theta)))/unicycles,
    % where x and y are the distance to each robot's goal
    %r1 = 0;
    %r2 = 0;
    if invEps
        trans = -sum((eps.^-1).*((S(:,1)-G(:,1)).*cos(S(:,3)) +...
        (S(:,2)-G(:,2)).*sin(S(:,3))))/N; % negative of error
    else
        trans = -sum(((S(:,1)-G(:,1)).*cos(S(:,3)) + ...
        (S(:,2)-G(:,2)).*sin(S(:,3))))/N; % negative of error
    end
    % saturate input command to 20px
    trans = sign(trans).*min(20,abs(trans));

    % apply r1 with noise
    %u2 = r1*ones(N,1) - sample(a1*r1^2 + a2*trans^2,N);
    u2 = -sqrt(a2*trans^2)*randn(N,1);
    S(:,3) = S(:,3) + eps.*u2; % theta
    % apply trans with noise
    %u1 = trans*ones(N,1) - sample(a3*trans^2 + a4*r1^2 +
    %   a4*r2^2,N);
    u1 = trans*ones(N,1) - sqrt(a3*trans^2)*randn(N,1);
    S(:,1) = S(:,1) + u1.*eps.*cos(S(:,3)); % x
    S(:,2) = S(:,2) + u1.*eps.*sin(S(:,3)); % y
    % apply r2 with noise
    %u3 = r2 - sample(a1*r2^2 + a2*trans^2, N);
    u3 = -sqrt(a2*trans^2)*randn(N,1); % optimized for speed
    S(:,3) = S(:,3) + eps.*u3;

end
```

```
function S = ApplyControlLawUni(S,G,eps,a1,a2,a3,a4)
    % *Unidirectional* Control Law with noise
    % Each action consists of a turn, a move, and another turn
    % We have two actions: turn and move forward
    % a1: rotation->rotation
    % a2: translation->rotation
    % a3: translation->translation
    % a4: rotation->translation
    N = size(S,1); % number of robots

    % First action: turn, assume we want to try to turn pi/2
    r1 = pi/2; % how much we want to turn for the first turn

    % apply r1 with noise
    u2 = r1*ones(N,1) - sqrt(a1*r1^2)*randn(N,1);
    S(:,3) = S(:,3) + eps.*u2; % theta
    % apply trans with noise
    u1 = -sqrt(a4*r1^2)*randn(N,1);
    S(:,1) = S(:,1) + u1.*eps.*cos(S(:,3)); % x
    S(:,2) = S(:,2) + u1.*eps.*sin(S(:,3)); % y

    % Second action: Translation
    % our requested translation is the amount that minimizes
    % the average error, capped at magnitude 20.  negative
    % because error is negative
    % trans = -sum(eps*(x*cos(theta) + y*sin(theta)))/unicycles,
    % where x and y are the distance to each robot's goal
    trans = -sum((eps.^-1).*((S(:,1)-G(:,1)).*cos(S(:,3)) + ...
        (S(:,2)-G(:,2)).*sin(S(:,3))))/N; % negative of error
    trans = max(0,min(20,trans)); % saturate input command

    % apply r1 with noise
    u2 = -sqrt(a2*trans^2)*randn(N,1);
    S(:,3) = S(:,3) + eps.*u2; % theta
    % apply trans with noise
    u1 = trans*ones(N,1) - sqrt(a3*trans^2)*randn(N,1);
    S(:,1) = S(:,1) + u1.*eps.*cos(S(:,3)); % x
    S(:,2) = S(:,2) + u1.*eps.*sin(S(:,3)); % y
```

```
    % apply r2 with noise
    u3 = -sqrt(a2*trans^2)*randn(N,1); % optimized for speed
    S(:,3) = S(:,3) + eps.*u3;
end

function S = ApplyObstacleControlLaw(S,G,O,eps,a1,a2,a3,a4)
    % Obstacle-Avoiding Control Law with noise
    % Each action consists of a turn, a move, and another turn
    % We have two actions: turn and move forward
    % a1: rotation->rotation
    % a2: translation->rotation
    % a3: translation->translation
    % a4: rotation->translation
    N = size(S,1); % number of robots

    nu = 20; % gain
    Qs = 30; % max effective distance

    % First action: turn, assume we want to try to turn pi/2
    r1 = pi/2; % how much we want to turn for the first turn

    % apply r1 with noise
    u2 = r1*ones(N,1) - sqrt(a1*r1^2)*randn(N,1);
    S(:,3) = S(:,3) + eps.*u2; % theta
    % apply trans with noise
    u1 = -sqrt(a4*r1^2)*randn(N,1);
    S(:,1) = S(:,1) + u1.*eps.*cos(S(:,3)); % x
    S(:,2) = S(:,2) + u1.*eps.*sin(S(:,3)); % y

    % Second action: Translation
    % our requested translation is the amount that minimizes
    % the average error, capped at magnitude 20.  negative
    % because error is negative
    % trans = -sum(eps*(x*cos(theta) + y*sin(theta)))/unicycles,
    % where x and y are the distance to each robot's goal
    trans = -sum((eps.^-1).*((S(:,1)-G(:,1)).*cos(S(:,3)) + ...
        (S(:,2)-G(:,2)).*sin(S(:,3))))/N; % negative of  error
    trans = sign(trans).*min(20,abs(trans)); % saturate input
```

```
% look at where we'd be if we evolved the system with no noise
Test = S;
Test(:,1) = S(:,1) + trans*eps.*cos(S(:,3)); % x
Test(:,2) = S(:,2) + trans*eps.*sin(S(:,3)); % y

% for each robot, calculate distance to closest obstacle
% and find repulsive fields of the evolved system
urep = zeros(N,1);
for r=1:N
    xdist = O(:,1)-Test(r,1);
    ydist = O(:,2)-Test(r,2);
    a = cos(S(r,3) - atan2(ydist,xdist)); % angles to objects
    % threshold to zero
    obsdist = max(sqrt(xdist.^2 + ydist.^2) - O(:,3),0);
    D = min(obsdist);
    if D <= Qs
        urep(r) = nu*a(find(obsdist == D,1))*(nu/D - nu/Qs)^2;
    else
        urep(r) = 0;
    end
end
repulsion = mean(urep);
% saturate repulsive field to trans
repulsion = sign(repulsion)*min(abs(trans),abs(repulsion));
trans = trans - repulsion;
 % if we can't go anywhere (repulsive fields of inf and -inf)
if isnan(repulsion)
    trans = 0;
end

% saturate input (again)
trans = sign(trans).*min(20,abs(trans));

% apply r1 with noise
u2 = -sqrt(a2*trans^2)*randn(N,1);
S(:,3) = S(:,3) + eps.*u2; % theta
% apply trans with noise
```

```
    u1 = trans*ones(N,1) - sqrt(a3*trans^2)*randn(N,1);
    S(:,1) = S(:,1) + u1.*eps.*cos(S(:,3)); % x
    S(:,2) = S(:,2) + u1.*eps.*sin(S(:,3)); % y
    % apply r2 with noise
    u3 = -sqrt(a2*trans^2)*randn(N,1); % optimized for speed
    S(:,3) = S(:,3) + eps.*u3;
end
```

A.1.3 Assembly Simulation

```
 /*
 * Aaron & Cem's ensemble control of assemble
 */

#ifndef ASSEMBLE_H
#define ASSEMBLE_H

class Assemble : public Test
{
    int subT; // timer to trigger next t
    int t;  // which timestep (in terms of goals) we're on
    // in a 2-step control policy, are we rotating or translating
    int rotORtrans;
    static const int numrobots = 6;
    static const int timesteps = 5000;
public:
    Assemble()  {
        subT = 0;
    t = 0;
        rotORtrans = 0;
    m_controlState = WAIT; // wait
        robot_mass = 0;
        robot_inertia = 0;

    // body for bounding box
    b2BodyDef bd;
        bd.position.Set(0.0f, 0.0f);
        b2Body* body = m_world->CreateBody(&bd);
```

```
// define bounding box
    b2Vec2 vs[4];
    vs[0].Set(-40.0f, 0.0f);
    vs[1].Set( 40.0f, 0.0f);
    vs[2].Set( 40.0f, 80.0f);
    vs[3].Set(-40.0f, 80.0f);
    b2ChainShape loop;
    loop.CreateLoop(vs, 4);
    b2FixtureDef fd;
    fd.shape = &loop;
    fd.density = 0.0f;
    fd.friction = 0.0f; // no friction
    body->CreateFixture(&fd);

    // make a friction joint between world and objects
    b2FrictionJointDef frictionJointDef;
    frictionJointDef.localAnchorA.SetZero();
    frictionJointDef.localAnchorB.SetZero();

// body for friction joint
    b2BodyDef bodyDef;
    m_groundBody = m_world->CreateBody(&bodyDef);
    frictionJointDef.bodyA = m_groundBody;
    frictionJointDef.maxForce = FRICTION;
    frictionJointDef.maxTorque = FRICTION;
    frictionJointDef.collideConnected = true;


for (int i=0; i < 3; i++) {
        b2Vec2 vs2[4];
      vs2[0].Set(-5.0f, 0.0f);
      vs2[1].Set( 0.0f, -5.0/sqrt(3));
      vs2[2].Set( 5.0f, 0.0f);
      vs2[3].Set( 0.0f, 5.0/sqrt(3));
  b2PolygonShape p;
  p.Set(vs2, 4);
```

```
      b2BodyDef bd2;
        bd2.type = b2_dynamicBody;
        bd2.position.Set(-30, 35+15*i);
        bd2.angle = -PI/2;
        bd2.gravityScale = 0.0f;
        b2Body* body2 = m_world->CreateBody(&bd2);

        frictionJointDef.bodyB = body2;
        m_world->CreateJoint( &frictionJointDef );
        body2->CreateFixture(&p, 0.2f);
}

    // create the goals
// Phase 1 - move left
int phase1 = 300;
int phase2 = 800;
int phase3 = 3000;
int phase4 = 4500;
for (int j = 0; j < timesteps; j++) {
      for(int i = 0; i < numrobots; i++) {
    float32 s;
    if (j < phase1) {
      s = (float) j/phase1;
            robGoals[i][0][j] = s*-35;
            robGoals[i][1][j] = 3+3*i;
    } else if (j < phase2) {
      s = (float) (j-phase1)/(phase2-phase1);
            robGoals[i][0][j] = -35;
            robGoals[i][1][j] =
                (1-s)*robGoals[i][1][phase1-1] +
                s*(32 + 6*i + (i/2)*3);
    } else if (j < phase3) {
      s = (float) (j-phase2)/(phase3-phase2);
            robGoals[i][0][j] = robGoals[i][0][phase2-1] +
                s*25;
            robGoals[i][1][j] = robGoals[i][1][phase2-1];

    } else if (j < phase4) {
```

```
  s = (float) (j-phase3)/(phase4-phase3);
  if (i == 0) {
          robGoals[i][0][j] = robGoals[i][0][phase3-1] +
            18*cos(-PI/2+s*PI/2.1);
          robGoals[i][1][j] = robGoals[i][1][phase3-1] +
            18*sin(-PI/2+s*PI/2.1) + 18;
  }
  if (i == 1) {
          robGoals[i][0][j] = robGoals[i][0][phase3-1] +
            12*cos(-PI/2+s*PI/2.5);
          robGoals[i][1][j] = robGoals[i][1][phase3-1] +
            12*sin(-PI/2+s*PI/2.5) + 11;
  }
  if (i == 2) {
          robGoals[i][0][j] = robGoals[i][0][phase3-1] +
            s*8;
          robGoals[i][1][j] = robGoals[i][1][phase3-1];
  }
  if (i == 3) {
          robGoals[i][0][j] = robGoals[i][0][phase3-1] +
            s*8;
          robGoals[i][1][j] = robGoals[i][1][phase3-1];
  }
  if (i == 4) {
          robGoals[i][0][j] = robGoals[i][0][phase3-1] +
            12*cos(-PI/2+s*PI/2.5);
          robGoals[i][1][j] = robGoals[i][1][phase3-1] -
            12*sin(-PI/2+s*PI/2.5) - 11;
  }
  if (i == 5) {
          robGoals[i][0][j] = robGoals[i][0][phase3-1] +
            18*cos(-PI/2+s*PI/2.1);
          robGoals[i][1][j] = robGoals[i][1][phase3-1] -
            18*sin(-PI/2+s*PI/2.1) - 18;
  }

} else {
        robGoals[i][0][j] = robGoals[i][0][j-1];
```

```
            robGoals[i][1][j] = robGoals[i][1][j-1];
    }
      }
}

    // make all our robots and their targets
for(int i =0; i<numrobots; i++) {
  float32 e = 1-DELTA + 2*DELTA*i/numrobots + DELTA/numrobots;
  float32 x_start = -6+2*i;
  float32 y_start = 3;

        // make each robot
  m_Robot[i] = new Robot(m_world,e,i,x_start,y_start);

  // connect robot to world friction
        b2Body* body = m_Robot[i]->m_body;
        frictionJointDef.bodyB = body;
        m_world->CreateJoint( &frictionJointDef );

  // draw kinematic bodies for trajectories/goals
  m_Target[i] = new Target(m_world,robGoals[i][0][0],
    robGoals[i][1][0]);

    }

// assume all robots have same mass
if (numrobots > 0) {
  robot_mass = m_Robot[0]->m_body->GetMass();
  robot_inertia = m_Robot[0]->m_body->GetInertia();
}

}

void Keyboard(unsigned char key) {
switch (key) {
        case 'a' : m_controlState |= ROBOT_LEFT; break;
        case 'd' : m_controlState |= ROBOT_RIGHT; break;
        case 'w' : m_controlState |= ROBOT_FWD; break;
```

```
          case 's' : m_controlState |= ROBOT_BWD; break;
    case 'g' : m_controlState &= ~WAIT; break;
          default: Test::Keyboard(key);
  }
}

void KeyboardUp(unsigned char key) {
  switch (key) {
          case 'a' : m_controlState &= ~ROBOT_LEFT; break;
          case 'd' : m_controlState &= ~ROBOT_RIGHT; break;
          case 'w' : m_controlState &= ~ROBOT_FWD; break;
          case 's' : m_controlState &= ~ROBOT_BWD; break;
          default: Test::Keyboard(key);
  }
}

  void Step(Settings* settings) {

  static float desired_dist = 0.0f;
  static float impulse = 0.0f;
  static float tf = 0.0f;

  if (m_controlState &= WAIT) // wait for keypress
    return;

      subT += 1;

      if( subT >= (settings->hz)*tf) { // this waits one second
          subT = 0; // reset subsecond counter

    if (t < timesteps-1) // increment as long as we're in range
      t += 1;

    // Set the control to be applied
          if( rotORtrans == CONTROL_ROT ) {
      // If turning, apply impulse to turn robots
              m_controlState |= IMPULSE_ROT;
      impulse = sqrt(PI*FRICTION*robot_inertia);
```

```
        tf = (1+DELTA)*impulse/FRICTION;
            } else {
                // Else, apply Control Law on linear velocity
        m_controlState |= IMPULSE_TRANS;

                // calculate control law
                desired_dist = 0.0f;
                for(int i =0; i<numrobots; i++) {
                    float32 ang = m_Robot[i]->m_body->GetAngle();
                    b2Vec2 pos = m_Robot[i]->m_body->GetPosition();
                    float32 eps = m_Robot[i]->epsilon;
                    desired_dist += -1/eps*((pos.x -
                        robGoals[i][0][t])*cos(ang)
+
                        (pos.y - robGoals[i][1][t])*sin(ang));
                }
                desired_dist /= numrobots;

        impulse = SIGN(desired_dist)*sqrt(2*ABS(desired_dist)
          *0.8*FRICTION*robot_mass);
        tf = (1+DELTA)*impulse/FRICTION;
            }
            rotORtrans ^= 1; // switch from rotating to translating
        } else {
            m_controlState &= ~(IMPULSE_ROT|IMPULSE_TRANS);
        }

    //show some useful info
    //m_debugDraw.DrawString(5, m_textLine,
    //  "Press w/a/s/d to control the robots" );
    //m_textLine += 15;
        char mystr[30];
        /*sprintf(mystr, "time = %i,(%f,%f)",
      t,robGoals[0][0][t],robGoals[0][1][t]);*/
    sprintf(mystr, "time = %i, d = %4.2f",t, desired_dist);
        m_debugDraw.DrawString(5, m_textLine, mystr );
    //TODO temporary manual-only mode
      //m_controlState &= ~(IMPULSE_ROT|IMPULSE_TRANS);
```

```
    // apply the control policy to all robots
        for(int i =0; i<numrobots; i++) {
            m_Robot[i]->update(m_controlState, impulse);
      m_Target[i]->update(robGoals[i][0][t],robGoals[i][1][t]);
        }
    // parent step function
    Test::Step(settings);

  }

    static Test* Create()
    {
        return new Assemble;
    }

    int m_controlState;
    Robot* m_Robot[numrobots];
    Target* m_Target[numrobots];
    float32 robGoals[numrobots][2][timesteps];
  float32 robot_mass;
  float32 robot_inertia;
};

#endif
```

A.2 Hardware Experiments

This code was used in the experiments in Chapter 4.

A.2.1 Matlab Program

```
function EOHtrackAndCommandRobots()

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Robot Demo
```

```
% Aaron Becker & Cem Onyuksel
% February 2012
%
% This program is designed to control an ensemble of segbots.
%
% This code loads UDP code to get robot pose from OptiTrack,
% sets up and
uses serial connections to each robot, enables
% manual or automatic control
% of the robots, and stores data from each run to a unique file
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    scoms = {'com1','com15','com14','com17'};

    import java.io.*
    import java.net.DatagramSocket
    import java.net.DatagramPacket
    import java.net.InetAddress
    format compact



    running = 1;

    %% Open a data file to store data runs
    fid = fopen(['RobotRuns\Robot-',datestr(now, ...
            'yyyy-mm-dd-HH-MM-SS'),'.txt'], 'w');

    % runs optiTrack UDP server in background
    system('motrack_udp_MultiRB_one_client.py&');
    % Open socket
    port = 3500;
    timeout = 500;
    packetLength = 200;
    try
        socket = DatagramSocket(port);
        socket.setSoTimeout(timeout);
        socket.setReuseAddress(1);
        packet = DatagramPacket(zeros(1,packetLength,'int8'),...
            packetLength);
```

```
catch Error
    display(Error);
    display(Error.message);
    try
       socket.close;
   catch Error
        display(Error.message);
   end
end % try


%%%%%%%%%%%%%%%%%%%% OPEN SERIAL CONNECTIONS %%%%%%%%%%%%%%%%
ncoms = numel(scoms);
serials = cell(ncoms,1);

closeopenserial  %clean up serial connections
for k = 1:ncoms
    serials{k} = serial(scoms{k},'Baudrate',57600, ...
        'DataBits',8, 'StopBits',1,'Timeout',0.3);
    fopen(serials{k});
    fprintf(serials{k},'');
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Initialize a joystick %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear JoyMEX
JoyMEX('init',1);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tic; %start a timer

% Create figure for displaying the robots w/ close function
mycolor = ['y','r','b','k','g','c','m'];

nRobots = ncoms;
Epsilon = ones(nRobots,1);  % variation in forward speed.
EpsilonCal.Gain = 10/100; % Low pass filter
EpsilonCal.IS_ON = false;
EpsilonCal.cmddist = 0.0;
EpsilonCal.maxDist = 2.53; % Maximum distance
```

```
if EpsilonCal.IS_ON
    EpsilonCal.fig = figure(3);
    clf
    set(EpsilonCal.fig,'CloseRequestFcn',@onClose);
    EpsilonCal.pHandles = zeros(nRobots,1);
    for i = 1:nRobots
        EpsilonCal.pHandles(i) = plot(0, Epsilon(i), ...
            'color',mycolor(i));
        hold on
    end
    xlabel('time (s)')
    ylabel('Epsilon values')
    title('Online Calibration of \epsilon')
end



% initialize robots and workspace
robots = zeros(nRobots,3);
RobOutlineY = [0,-2.375,-2.375,-2.75,-2.75,-3,-3,-2.75, ...
    -2.75,-2.375,-2.375,2.375, 2.375, 2.75, 2.75, 3, 3, ...
    2.75, 2.75, 2.375, 2.375, 0,0,-1,-1,-2,0,2,1,1,0]/12;
RobOutlineX = [-1,-1,0,0,-2,-2,2,2,0,0,5.5, 5.5,0,0, 2, ...
    2,-2,-2,0,0, -1,-1, 0,0,3,3,5,3,3,0,0]/12;
hRobots = zeros(nRobots,1);
hRobotsDes = zeros(nRobots,1);
hTargets= zeros(nRobots,1);

f1 = figure(1);
clf;
set(f1,'CloseRequestFcn',@onClose);
xLim = 12;   yLim = 16;% Set the size of the robot arena
plot([-xLim, xLim, xLim, -xLim, -xLim]/2,[yLim, yLim, ...
    -yLim, -yLim, yLim]/2,'r','linewidth',2)
hold on

% starting goal positions
thTargets = 0:pi/8:2*pi;
```

```
ctrTargets = [-1 2 -1 2;0 0 3 3]';
radTargets = 1/2;

for i = 1:nRobots
    % draw targets and make them draggable
    hTargets(i)= patch(ctrTargets(i,1)+ ...
        radTargets*cos(thTargets),ctrTargets(i,2)+ ...
        radTargets*sin(thTargets),mycolor(i));
    set(hTargets(i),'LineWidth',1,'EdgeColor',mycolor(i));
    draggable(hTargets(i),'none', ...
        [-xLim/2,xLim/2,-yLim/2,yLim/2]);
end
f1Title = title('Robot position');
for i =1:nRobots
    % draw robot locations and the desired locations
    th = robots(i,3);
    rx = cos(th)*RobOutlineX -sin(th)*RobOutlineY+ robots(i,1);
    ry = sin(th)*RobOutlineX +cos(th)*RobOutlineY+ robots(i,2);
    hRobotsDes(i) = plot(rx,ry,mycolor(i));hold on
    hRobots(i) = patch(rx,ry,mycolor(i));
    if(mycolor(i) == 'k')
        set(hRobots(i),'EdgeColor',[0.2,0.2,0.2]);
    end
end



axis equal
% Force update of plot
drawnow

% Create figure to graph error/get user input w/ close function
S.f2 = figure(2);
set(S.f2,'CloseRequestFcn',@onClose);
clf;
tlast = toc;
S.hErrors = zeros(nRobots+1,1);
S.hErrors(1) = plot(tlast,sum(sum((robots(:,1:2) - ...
    ctrTargets).^2)),'-r','linewidth',2); hold on
```

```
for i =1:nRobots
    % draw robot locations and the desired robot locations
    S.hErrors(i+1) = plot(tlast,sum((robots(i,1:2) - ...
        ctrTargets(i,1:2)).^2),'-','color' ,mycolor(i));
end
xlabel('Time (s)')
ylabel('Distance Error^2 (ft^2)')
S.title = title('Error Plot');
S.manualMode = 1;
S.pp = uicontrol(   'style','pop',...
             'unit','pix',...
             'position',[70 410 120 10],...
             'backgroundc',get(S.f2,'color'),...
             'fontsize',12,'fontweight','bold',...
             'string',{'Manual';'Automatic';'Calibrate'},...
             'value',S.manualMode);
set(S.pp, 'callback',{@pp_call,S});  % Set the callback.


function [] = pp_call(varargin)
    % Callback for popupmenu.
    S = varargin{3};  % Get the structure.
    S.manualMode = get(S.pp,'val'); % Get the users choice
end

S.pb = uicontrol('style','push',...
             'units','pix',...
             'position',[420 390 90 30],...
             'fontsize',12,...
             'string','Reset',...
             'callback',{@pb_call,S});
set(S.pb, 'callback',{@pb_call,S});  % Set the callback.

function [] = pb_call(varargin)
    % Callback for 'reset' button
    S = varargin{3};  % Get the structure.
    for ic =1:numel(S.hErrors)
```

```
        % draw robot locations and the desired robot locations
        set(S.hErrors(ic),'Xdata',[],'Ydata',[]);
    end
end

% Set control state.
CL.turnwait = 3.0; % seconds to wait after a turn command.
CL.state = 1; % 1 = go straight, 2 = turn, 3 = wait for turn
CL.start = toc;
% state does not switch from drive straight to turn until this
CL.dThresh = 1/12;
CL.sendcontrol = 0;
CL.straightwait = 3.0; % seconds to wait after a straight
CL.straightcounter = 0;
CL.CAL.state = 0;
CL.CAL.wait = 2.5;  %seconds
ft2rev  = 1; %540 ticks per revolution
oneTurn = 1; %2/12 radius wheel wheelbase = 5.5/12


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%   MAIN LOOP   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
while(running)

    try %read the data from OptiTrack
        socket.receive(packet);
        UDPmssg = packet.getData;
        UDPmssg = UDPmssg(1:packet.getLength);
        Pose = reshape(typecast(UDPmssg,'single'),9,[])';

        EA = SpinCalc('QtoEA213',Pose(:,5:8),1e-4,0);
        %display([Pose(:,[1,2,3,4]),EA(:,1)])

        data(1:size(Pose,1),:) = [Pose(:,[1,2,4]) ,EA(:,1)];
        data(:,2) = data(:,2)*3.2808399;  %convert from m to ft
        data(:,3) = -data(:,3)*3.2808399; %convert from m to ft
        data(:,4) = (90+data(:,4))*pi/180;%convert to radians
```

```
    for i = 1:size(data,1)
        id = data(i,1);
        if id > 0 && id<=nRobots && ...
            abs(data(i,2)) < xLim && abs(data(i,3)) < yLim
            robots(id,:) = data(i,2:4); %update position
        end
    end
    reDrawRobotPos(hRobots, robots, 0,0,RobOutlineX, ...
        RobOutlineY);
catch Error
end % try

tcurr = toc; % read current time
if  (tcurr > tlast + 0.5)
    tlast = tcurr;
    for i =1:nRobots   %compute the target locations
        ctrTargets(i,:) = [mean(get(hTargets(i),'Xdata'));
        mean(get(hTargets(i),'Ydata'))];
    end
    % update error graph
    xdata = [get(S.hErrors(1),'Xdata'),tlast];
    totalErr = sum(sum((robots(:,1:2) - ctrTargets).^2));
    set(S.hErrors(1),'Ydata', [get(S.hErrors(1), ...
        'Ydata'), totalErr],'Xdata', xdata) % total error
    set(S.title, 'string',['Total Error = ', ...
        num2str(totalErr,'%.2f'),', Time = ', ...
        num2str(xdata(end)-xdata(1),'%.1f')]);
    set(f1Title, 'string',['Total Error = ', ...
        num2str(totalErr,'%.2f'),', Time = ', ...
        num2str(xdata(end)-xdata(1),'%.1f')]);

    for i =1:nRobots % update graph of robot position error
        set(S.hErrors(i+1),'Ydata', [get(S.hErrors(i+1),...
            'Ydata'),  sum((robots(i,1:2) - ...
            ctrTargets(i,1:2)).^2)],'Xdata', xdata);
    end
end
```

```
% save the robot position and goal
fprintf(fid,[sprintf('%5.3f, ',toc), ...
    num2str(S.manualMode),', ',sprintf('%5.3f, ...
    ',[robots,ctrTargets,Epsilon]),'\n']);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% CONTROLLER %%%%%%%%%
if  S.manualMode == 1
    [~, ab] = JoyMEX(1); % Query button state of joystick 1
    CL.cur = toc;
    if  CL.cur > CL.start + 0.2

        if sum(ab(1:4)) % if a button is pressed
            CL.start = CL.cur;
            dpos =(+ab(4) - ab(2));  %bt 4 and 2 fwd/bwd
            dth  = (+ab(1) - ab(3)); %bt 1 and 3 turning
            reDrawRobotPos(hRobotsDes, robots, dth, ...
                dpos,RobOutlineX,RobOutlineY);
            Fs = [ft2rev*dpos, oneTurn*dth];
            sendCommand(Fs,serials,ncoms);
        end
    end
elseif S.manualMode == 2   %automatic control
    % Control law:  send fw/bw command every .5 second
    % mag < dThresh, then switch and send turn
    % command to turn pi/2.  Wait then switch
    % to turn command

    if CL.state == 0 %wait afer a straight command
        CL.cur = toc;
        if  CL.cur > CL.start + CL.straightwait
            CL.state = 2;
            if EpsilonCal.IS_ON && ...
                abs(EpsilonCal.cmddist) > 0.084

                % Low Pass filter:  Eps(k+1) = Eps(k) +
                %   Gain*( MeasEps(k) - Eps(k) )
                MeasEps = (sum( (robots(:,1:2)- ...
```

```
                EpsilonCal.startpos).^2, 2).^.5)/...
                abs(EpsilonCal.cmddist);
            Epsilon = Epsilon +EpsilonCal.Gain*...
                (abs(EpsilonCal.cmddist)/...
                EpsilonCal.maxDist)*...
                (  MeasEps - Epsilon);

            %Plot the new Epsilon values
            xdata = [get(EpsilonCal.pHandles(1), ...
                'Xdata'),tlast];
            for i = 1:nRobots
                ydata = [get(EpsilonCal.pHandles(i),...
                    'Ydata'),Epsilon(i)];
                set(EpsilonCal.pHandles(i),'Xdata',...
                    xdata,'Ydata',ydata);
            end
        end
    end
elseif CL.state == 1   %straight command
    dth = 0;
    dpos = controlLaw(nRobots, robots, ...
        ctrTargets,Epsilon);

    if EpsilonCal.IS_ON
        EpsilonCal.cmddist = min(EpsilonCal.maxDist,...
            abs(dpos)); % record data for calibration
        EpsilonCal.startpos = robots(:,1:2);
    end
    CL.state = 0;
    CL.straightcounter = CL.straightcounter+1;
    CL.start = toc;
    CL.sendcontrol = 1;
    display(['mode = 1,dist = ',num2str(dpos)])
elseif CL.state == 2  %Turn command
    dpos = 0;
    dth = pi/2;
    CL.state = 3;
    CL.start = toc;
```

```
                CL.sendcontrol = 1;
                display(['mode = 2,turn = ',num2str(dth)])
            elseif CL.state == 3 %Wait during turn
                CL.cur = toc;
                if  CL.cur > CL.start + CL.turnwait
                    CL.state = 1;
                end
            end

            if CL.sendcontrol == 1
                CL.sendcontrol = 0; %reset control law
                reDrawRobotPos(hRobotsDes, robots, dth, dpos, ...
                    RobOutlineX,RobOutlineY);
                Fs = [dpos*ft2rev, dth*oneTurn];
                sendCommand(Fs,serials,ncoms);
            end
        elseif S.manualMode == 3
            % Calibrate Mode:record position,command robots to move
            % forward 1ft, record distance travelled,
            % command robots to move
backwards 1ft,
            %record distance travelled.  Repeat, and save
            % median value to Epsilon

            if CL.CAL.state == 0   %initialize
                CL.CAL.state = 1;
                CL.CAL.count = 0;
                CL.CAL.dists = zeros(nRobots,10);

            elseif CL.CAL.state == 1   %straight command

                CL.CAL.count = CL.CAL.count+1;
                dth = 0;
                dpos = (-1)^CL.CAL.count;
                CL.CAL.startpos = robots(:,1:2);
                CL.CAL.start = toc;
                CL.CAL.state = 2;
```

```
            reDrawRobotPos(hRobotsDes, robots, dth, dpos,...
                RobOutlineX,RobOutlineY);
            Fs = [dpos*ft2rev, dth*oneTurn];
            sendCommand(Fs,serials,ncoms);

        elseif CL.CAL.state == 2  %Wait command
            CL.CAL.cur = toc;
            if  CL.CAL.cur > CL.CAL.start + CL.CAL.wait
                % record position change
                CL.CAL.dists(:,CL.CAL.count) = ...
                    sum( (robots(:,1:2)- ...
                    CL.CAL.startpos).^2, 2).^.5;
                CL.CAL.state = 1;
                if CL.CAL.count == 10
                    CL.CAL.state = 3;
                end
            end
        elseif CL.CAL.state == 3 %set distances
            Epsilon = median(CL.CAL.dists,2);
            display(CL.CAL.dists)
            display(['Num dists < 0.25 = ', ...
                num2str(sum(sum(CL.CAL.dists < 0.25))), ...
                ' out of ',num2str(numel(CL.CAL.dists))])
            display(Epsilon)
            CL.CAL.state = 0;
            S.manualMode = 1;
            set(S.pp, 'value',S.manualMode);
        end

    end
    % Force update of plot
    drawnow %expose update
end  %END MAIN WHILE LOOP
%CLEAN UP BEFORE CLOSING PROGRAM
%clear the serial ports
for k = 1:ncoms
    fclose(serials{k});
    delete(serials{k})
```

```
end
%close the file
fclose(fid);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%% FUNCTIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function dist = controlLaw(nRobots, robots, ctrTargets,Epsilon)
    F = sum((Epsilon.^(-1)).*( cos(robots(:,3)).* ...
            (robots(:,1)-ctrTargets(:,1))+ ...
            sin(robots(:,3)).*(robots(:,2)-ctrTargets(:,2)) ));
    dist = -F/nRobots;
end


function sendCommand(Fs,serials,ncoms)
        ByteArray = uint8(zeros(1,5));
        ByteArray(1) = 253;
        if Fs(1) > 0
            ByteArray(2) = 'F';
            ByteArray(3) = min(252,round((Fs(1)*100 )));
        elseif Fs(1) < 0
            ByteArray(2) = 'B';
            ByteArray(3) = min(252,round((abs(Fs(1))*100 )));
        elseif Fs(2) > 0
            ByteArray(2) = 'L';
            ByteArray(3) = min(252,round((Fs(2)*100)));
        else
            ByteArray(2) = 'R';
            ByteArray(3) = min(252,round((abs(Fs(2))*100)));
        end
        ByteArray(4) = ByteArray(3);%ERROR cHECKING
        ByteArray(5) = 255;
        for kc = 1:ncoms
            fwrite(serials{kc},ByteArray); %send to each robot
        end
```

```
    end


    function onClose(src,evt) %#ok<INUSD>
       % When user tries to close the figure, end the while loop
       % Clear MEX-file to release joysticks
       clear JoyMEX

       % close UDP socket
       try
           socket.close;
       catch Error
            display(Error.message);
       end
       running = 0;
       delete(src);
       %close the python script
       system('taskkill /IM cmd.exe');
       %close all other figures
       set(0,'ShowHiddenHandles','on')
       delete(get(0,'Children'))
    end
end

function reDrawRobotPos(hRobots, robots, dth, dpos, ...
                        RobOutlineX,RobOutlineY)
         for i =1:numel(hRobots)  %redraw desired robot position
            th = robots(i,3)+dth;
            rx = cos(th)*RobOutlineX -sin(th)*RobOutlineY+ ...
                robots(i,1)+dpos*cos(th);
            ry = sin(th)*RobOutlineX +cos(th)*RobOutlineY+ ...
                robots(i,2)+dpos*sin(th);
            set(hRobots(i), 'XData',rx, 'YData',ry );
         end
end

function closeopenserial()
    % close any open ports
```

```
    ins = instrfind;
    for i=1:size(ins,2)
        if(strcmp('open',ins(i).Status))
            s = ins(i);
            fprintf(['Closing: ',s.Port,'\n']);
            fclose(s);
            delete(s);
        end
    end
end
```

A.2.2 Optitrak Data Parsing

```
 ######################################################################
#   motrack_udp_MultiRB_one_client.py
#  Version motrack_udp_MultiRB_one_client.py is used to stream to a
#  MATLAB reader.
#
#  This code written to interface with Matlab Software
#
#   v1.0.0
#   written by Miles Johnson, Aaron Phelps, and Cem Onyuksel
#
#   reads 6-DOF on multiple trackables from a running Tracking
#   Tools (go to Streaming Pane and check the box for 'Broadcast
#   Frame Data'
in NaturalPoint Streaming Engine, For "Network
#   Interface Selection",
under "Local Interface" set to
#   <opti_ip> ) and sends the data to
<drone_ip>, the IP of a
#   laptop.
#
#
#    Note that the order of IP addresses in drone_ip corresponds to
#    trackable numbers.  These must match!
#
```

```
#    Copies of this code are found on the svn at
# https://subversion.cs.illinois.edu/svn/ae483/trunk/
#   optitrack/motrack_udp_multi.py
# ------------------------------------------------------------------
#   Change Log:
#
#
#####################################################################

import socket
import struct
import threading
import time



#####################################################################
######## Configuration Options ######################################
#####################################################################

# Drone Computer IP's - First IP responds to first trackable, etc.
# To increase number of drones,
# add IP's (run ifconfig on Linux machines)
#drone_ip = "128.174.192.70" #Mechtronics PC, 2nd from end
drone_ip = socket.gethostbyname(socket.gethostname()) #Optitrack PC

# Drone Computer UDP Port
drone_port = 3500

# OptiTrack Computer IP address
# [Start->type 'cmd' in command window, type IPconfig]
opti_ip =socket.gethostbyname(socket.gethostname()) #This is the
                            #same address that the computer
                            # uses to connect to the Internet.

# Data Port (Set in Optitrack Streaming Properties)
opti_port = 1511
```

```
# Multicast Interface (in Optitrack Streaming Properties)
multicastAdd = "239.255.42.99"

#########################################################################
#########################################################################
# DO NOT EDIT ANYTHING BENEATH THIS LINE!!!!
#########################################################################
#########################################################################

class FMClientUDP(threading.Thread): #{{{1
    def __init__(self,address=('localhost',drone_port)):
        threading.Thread.__init__(self)
        self.setDaemon(True)
        self.address = address
        self.sock = None
        self.lock = threading.Lock()

    def start(self):
        threading.Thread.start(self)

    def stop(self):
        if self.sock:
            self.sock.close()

    def sendMessage(self, msg):
        self.lock.acquire()
        data = msg
        if self.sock:
            try:
                #print "sending msg = ",data
                self.sock.sendto(data,self.address)
                pass
            except socket.error:
                #print "socket error"
                pass
        self.lock.release()

    def run(self):
```

```
        try:
            self.sock = socket.socket(socket.AF_INET,
            socket.SOCK_DGRAM)
            #print "ClientUDP connected"
        except socket.error:
            #print "ClientUDP socket error."
            return



def unPack(data): #{{{1
    trackableState = []
    byteorder='@'
    PacketIn = data
    major = 2
    minor = 0
    offset = 0
    # message ID, nBytes
    messageID, nBytes = struct.unpack(byteorder+'hh',
        PacketIn[offset:offset+4])
    offset += 4
    print 'messageID=',messageID,' number of bytes=',nBytes
    if (messageID == 7):
        frameNumber,nMarkerSets = struct.unpack(byteorder+'ii',
            PacketIn[offset:offset+8])
        offset += 8
        #print 'Markersets=', nMarkerSets
        i=nMarkerSets
        while (i > 0):
            ns = PacketIn[offset:offset+255]
            szNamelen=ns.find('\0')
            #print ns, szNamelen
            szName = struct.unpack(byteorder+str(szNamelen)+'s',
                PacketIn[offset:offset+szNamelen])[0]
            offset += szNamelen+1 # include the C zero char
            #print 'Modelname=',szName
            # markers
            nMarkers = struct.unpack(byteorder+'i',
```

```
        PacketIn[offset:offset+4])[0]
    offset += 4
    #print 'Markercount=',nMarkers
    j=nMarkers
    while (j>0):
        x,y,z = struct.unpack(byteorder+'fff',
            PacketIn[offset:offset+12])
        offset += 12
        j=j-1
    i=i-1

#unidentified markers
nUMarkers = struct.unpack(byteorder+'i',
    PacketIn[offset:offset+4])[0]
offset += 4
#print 'Unidentified Markercount=',nUMarkers
i = nUMarkers
while (i > 0):
    ux,uy,uz = struct.unpack(byteorder+'fff',
        PacketIn[offset:offset+12])
    offset += 12
    i=i-1

# rigid bodies
nrigidBodies = struct.unpack(byteorder+'i',
    PacketIn[offset:offset+4])[0]
nr = nrigidBodies
print nr
offset += 4
#print 'Rigid bodies=',nrigidBodies
while (nr > 0):
    ID = struct.unpack(byteorder+'i',
        PacketIn[offset:offset+4])[0]
    offset += 4
    rbx,rby,rbz = struct.unpack(byteorder+'fff',
        PacketIn[offset:offset+12])
    offset += 12
    rbqx,rbqy,rbqz,rbqw = struct.unpack(byteorder+'ffff',
```

```
    PacketIn[offset:offset+16])
offset += 16

trackableState.extend([ID,frameNumber, rbx, rby, rbz,
    rbqx, rbqy, rbqz, rbqw])

print '\nID=',ID
print 'pos:',rbx,rby,rbz
print 'ori:',rbqx,rbqy,rbqz,rbqw
# associated marker positions
nRigidMarkers = struct.unpack(byteorder+'i',
    PacketIn[offset:offset+4])[0]
offset += 4
#print 'Marker count=',nRigidMarkers
md = []
markerID = []
markersize = []
for i in range(0,nRigidMarkers):
    md.extend(struct.unpack(byteorder+'fff',
        PacketIn[offset:offset+12]))
    offset += 12
if major >= 2:
    for i in range(0,nRigidMarkers):
        markerID.append(struct.unpack(byteorder+'I',
            PacketIn[offset:offset+4])[0])
        offset += 4
    for i in range(0,nRigidMarkers):
        markersize.append(struct.unpack(byteorder+'f',
            PacketIn[offset:offset+4])[0])
        offset += 4
    for i in range(0,nRigidMarkers):
        pass
else:
    for i in range(0,nRigidMarkers):
        pass

# marker errors
if major >= 2:
```

```
                markerError = struct.unpack(byteorder+'f',
                    PacketIn[offset:offset+4])[0]
                offset += 4
                #print 'Mean marker error=',markerError

            nr = nr-1 # next rigid body

    return nrigidBodies, trackableState

if __name__ == '__main__': #{{{1

    # Initialize Multicast Socket
    mreq = struct.pack('4sl',socket.inet_aton(multicastAdd),
        socket.INADDR_ANY)

    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,1)

    s.bind((opti_ip, opti_port))

    s.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)

    ###### create data sender ####
    clientaddress = (drone_ip,drone_port)
    client = FMClientUDP(address=clientaddress)
    client.start()

    # Receive data
    print "begin recv'ing"

    frame_counter = 1.
    while (True):
        data, addr = s.recvfrom(20240)

        if data:
            #print "FRAME NUMBER ", frame_counter,
            #print
```

```
numRB, state = unPack(data)
#print 'There are ',numRB,' rigid bodies detected!'
#time.sleep(2)
msg = ''
for rb in range(numRB):
    ID = state.pop(0)
    fn = state.pop(0)
    x  = state.pop(0)
    y  = state.pop(0)
    z  = state.pop(0)
    q0 = state.pop(0)
    q1 = state.pop(0)
    q2 = state.pop(0)
    q3 = state.pop(0)

    msg_rb = struct.pack('fffffffff',ID,x,y,z,
        q0,q1,q2,q3,frame_counter)
    #x == 0 and y == 0 and z == 0 and
    if q0 == 0 and q1 == 0 and q2 == 0 and q3 ==0 :
        pass
    else:
        msg = msg+msg_rb
    # I'd like to just send ID, x,y,theta, framecounter
    #print msg

client.sendMessage(msg)

    #print

    #else:
        #print "NO MORE TRACKABLES DETECTED"
        #print

frame_counter = frame_counter + 1
```

A.2.3 Differential-Drive Robot Program

```
/*****************************************************************
 user_mainEOH.c
 MSP430F2272

 Aaron and Cem
 Turns 2 wheel robot into RC car


 College of Engineering Control Systems Lab
 University of Illinois at Urbana-Champaign
 *****************************************************************/

// need to make sure
// theta is between -pi and pi.

#include "msp430x22x2.h"
#include "UART.h"
#include "LS7266.h"

#define PI 3.141592654

#define FWD 0x4
#define REV 0x1
#define LEFT 0x2
#define RIGHT 0x8

////////  FUNCTION PROTOTYPES ////////////////////////////////////
void Init_PWM(void);
//void Motor_PWM(char motor,float u);
void MotorPI(float vref, float tref, float *DISTpos,
    float *THETApos,
char zeroAllIntegrals);
//////// END FUNCTION PROTOTYPES ////////////////////////////////

char newprint = 0;
unsigned long timecnt = 0;
```

```
char msgindex = 0, txindex = 0;
char started = 0, newmsg = 0;

char wireless_state = 0;
char wall_follow = 0, right_wall = 1;
unsigned state_counter = 0;
int rw_dist = 0, fw_dist = 0;
int ADC[3];

int rw_ref = 800, fw_near = 600, fw_far = 800, fw_ref = 900;
float Kp_fw = 0.005, Kp_rw = 0.003;

float vref = 0, tref = 0;

char WiFicmdFlag = 0; //0 == no move, 1 == straight move, 2 == turn
float cmdnum = 0;
float dist = 0;
float turn = 0;
float THETApos = 0;
float DISTpos = 0;
char zeroAllIntegrals = 0;
float ang_err = 0.0;
float distTOGO = 0.0;

void main(void) {

    WDTCTL = WDTPW + WDTHOLD; // Stop WDT

    if (CALBC1_16MHZ == 0xFF || CALDCO_16MHZ == 0xFF)
        while (1)
            ;

    DCOCTL = CALDCO_16MHZ; // Set uC to run at approximately 16 Mhz
    BCSCTL1 = CALBC1_16MHZ;

    P1DIR |= 0x1; // Default LED output

    // Timer A Config
```

```
TACCTL0 = CCIE; // Enable interrupt
TACCR0 = 20000; // period = 5ms
TACTL = TASSEL_2 + MC_1 + ID_2; // source SMCLK, up mode

// ADC10 Config
ADC10CTL0 = SREF_0 + ADC10SHT_1 + ADC10ON + ADC10IE + MSC;
// Ref-(V+=Vcc,V-=GND), SHT=8xADC10CLk, enable interrupt,
// multiple sample/convert
ADC10CTL1 = INCH_2 + CONSEQ_1 + ADC10SSEL_0 + SHS_0;
// Start at A2, sequence of channels, source ADC10OSC,
// use SC bit for trigger
ADC10DTC0 = 0;
ADC10DTC1 = 3; // Number of conversions in a block
ADC10SA = (unsigned int) ADC; // Start address pointer for DTC
ADC10AE0 = 0x5; // Disable port pin buffers for A0,A2

// Inputs from wireless keyfob
P3SEL &= ~0xF; // DI/O on P3.0-3.3
P3DIR &= ~0xF; // Input Direction
P3REN &= ~0xF; // Disable internal pull-up/dn resistors

Init_UART(57600, 1); // Initialize UART for 57600 baud serial
Init_Encoders(); // Initialize encoders
Init_PWM(); // Set up Timer B for PWM output on TB1 and TB2

// Wireless modem control pins
P2SEL &= ~0xC0; // Digital I/O
P2OUT |= 0x80; // CMD/Data pin high for data
P2DIR |= 0x80; // Output direction for CMD/data pin (P2.7)
P2DIR &= ~0x40; // Input direction for CTS pin (P2.6)

ADC10CTL0 |= ENC; // Enable ADC conversions

_BIS_SR(GIE);
// Enable global interrupt

while (1) {
```

```
        if (newmsg) {
                //read position and theta set point
                my_scanf(rxbuff, &dist, &turn);
                THETApos = 0.0;
                DISTpos = 0.0;
                if (dist != 0.0F)
                {   WiFicmdFlag = 1;}
                else if(turn != 0.0F)
                {   WiFicmdFlag = 2;}
            newmsg = 0;
        }

        if (newprint) {
            P1OUT ^= 0x1;
            UART_send(3,(float)DISTpos,
                (float)THETApos,(float)cmdnum);
            newprint = 0;
        }

    }
}

// Timer A0 interrupt service routine
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A(void) {
    timecnt++; // Keep track of time for main while loop.

    if ((timecnt % 10) == 0) { // 50ms sample rate

        zeroAllIntegrals = 0; //reset
        if (1 == WiFicmdFlag) // go the prescribed distance
                {
            tref = 0.0;
            distTOGO = (dist - DISTpos);

            if (fabsf(distTOGO) < 0.025) //0.05 ~ 1/2 inch
                    { //declare success
                vref = 0.0F;
```

```
                WiFicmdFlag = 0; //reset the flag
                zeroAllIntegrals = 1; // we have PID controller
            } else {
                vref = distTOGO;
            }
        } else if (2 == WiFicmdFlag) // turn the prescribed angle
                {
            //no sign problems since THETApos never wraps around pi
            ang_err = (turn - THETApos);
            if (fabsf(ang_err) < 0.05) { //declare success
                WiFicmdFlag = 0;
                tref = 0.0F;
                zeroAllIntegrals = 1;
            } else {
                tref = 0.35 * ang_err;
            } //was 0.5, but overshot, 0.25 undershot
            vref = 0.0F;
        } else { // no command -- don't move
            tref = 0.0F;
            vref = 0.0F;
            zeroAllIntegrals = 1;
        }

        MotorPI(vref, tref, &DISTpos, &THETApos, zeroAllIntegrals);
    }

    if ((timecnt % 20) == 0) { // 100 ms
        newprint = 1; // .5 seconds passed
    }

    ADC10CTL0 |= ADC10SC; // Trigger ADC every 5ms
}

// ADC 10 ISR - Called when conversions (A7-A0) have completed
#pragma vector=ADC10_VECTOR
__interrupt void ADC10_ISR(void) {

    rw_dist = 1023 - ADC[2]; // Channel A0 results
```

```
    fw_dist = 1023 - ADC[0]; // Channel A2 results
    // re-initialize DTC block start address
    ADC10SA = (unsigned int) ADC;

}

// USCI Transmit ISR - Called when TXBUF is empty
#pragma vector=USCIAB0TX_VECTOR
__interrupt void USCI0TX_ISR(void) {

    if (IFG2 & UCA0TXIFG) { // USCI_A0 requested TX interrupt
        if (printf_flag) {
            if (currentindex == txcount) {
                senddone = 1;
                printf_flag = 0;
                IFG2 &= ~UCA0TXIFG;
            } else {
                UCA0TXBUF = printbuff[currentindex];
                currentindex++;
            }
        } else if (UART_flag) {
            if (!donesending) {
                UCA0TXBUF = txbuff[txindex];
                if (txbuff[txindex] == 255) {
                    donesending = 1;
                    txindex = 0;
                } else
                    txindex++;
            }
        }

        IFG2 &= ~UCA0TXIFG;
    }

    if (IFG2 & UCB0TXIFG) { // USCI_B0 requested TX interrupt

        IFG2 &= ~UCB0TXIFG; // clear IFG
    }
```

```
}

// USCI Receive ISR - Called when shift register has been
// transferred to RXBUF
// Indicates completion of TX/RX operation
#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR(void) {

    if (IFG2 & UCB0RXIFG) { // USCI_B0 requested RX interrupt

        IFG2 &= ~UCB0RXIFG; // clear IFG
    }

    if (IFG2 & UCA0RXIFG) { // USCI_A0 requested RX interrupt

        if (!started) { // Haven't started a message yet
            if (UCA0RXBUF == 253 || UCA0RXBUF == 126) {
                started = 1;
                newmsg = 0;
            }
        } else { // In process of receiving a message
            if ((UCA0RXBUF != 255) &&
                (msgindex < (MAX_NUM_FLOATS * 5))) {
                rxbuff[msgindex] = UCA0RXBUF;

                msgindex++;
            } else { // Stop char received or too much data
                if (UCA0RXBUF == 255) { // Message completed
                    newmsg = 1;
                    rxbuff[msgindex] = 255; // "Null"
                }
                started = 0;
                msgindex = 0;
            }
        }
        IFG2 &= ~UCA0RXIFG;
    }
}
```

```
/* control.c
*
* Cem Onyuksel and Aaron Becker 2012
*/

#include "msp430x22x2.h"
#include "LS7266.h"

// initializes Timer B for PWM output at 20kHz
// initial duty cycle is 0% for both channels
void Init_PWM(void) {

    TBCTL = TBSSEL_2 + MC_1;      // Source SMCLK, Up Mode
    TBCCR0 = 800;                 // Set up PWM freq. of 20kHz
    TBCCTL0 = 0;                  // No interrupt, no output
    TBCCR1 = 0;                   // Initialize TB1 duty cycle (0%)
    TBCCTL1 = CLLD_1 + OUTMOD_7;  // Update CCR1 on TB=0, set mode
    TBCCR2 = 0;                   // Initialize TB2 duty cycle (0%)
    TBCCTL2 = CLLD_1 + OUTMOD_7;  // Update CCR2 on TB=0, set mode

    P4SEL |= 0x06;    // Enable TB1 & TB2
    P4DIR |= 0x06;    // Output TB1 & TB2 to P4.1 & P4.2

    P4SEL &= ~0x1; P4DIR |= 0x1;            // Phase 1 - P4.0
    P2SEL &= ~0x20; P2DIR |= 0x20;          // Phase 2 - P2.5
}


void Motor_PWM(char motor, float u) {
    signed char dir = 0;

    if((motor!=1) && (motor!=2)) return;

    // Saturation control
    if(u > 10) u = 10;
    if(u < -10) u = -10;

    if(u >= 0) dir = 1;
```

```
    if(u < 0) dir = -1;

    u = fabsf(u);

    if(motor == 1) {
        if(dir == 1) P4OUT |= 0x1;
        else P4OUT &= ~0x1;

        TBCCR1 = (unsigned int)(TBCCR0*((10.0-u)/10.0) + 0.5);
    }
    else {
        if(dir == 1) P2OUT |= 0x20;
        else P2OUT &= ~0x20;

        TBCCR2 = (unsigned int)(TBCCR0*((10.0-u)/10.0) + 0.5);
    }
}

float Kp=2.08, Ki = 7.32, Ka = 1, Kp_turn = 4; Ki_turn = 1.0;
float enc1 = 0, enc2 = 0;
float I1 = 0, I2 = 0, v1 = 0, v2 = 0, e1 = 0, e2 = 0;
const float cpos = 0.2198, vpos = 2.424;
const float cneg = -0.386, vneg = 2.177;
float u1 = 0, u2 = 0;
float olde1 = 0, olde2 = 0, e_turn = 0, v1temp = 0, v2temp = 0;
float oldenc1 = 0, oldenc2 = 0;
float Iturn = 0.0, olde_turn = 0.0;


// vref and tref are the angular velocity and turn rate
// (difference in angular velocities) setpoints in rad/s.
// MotorPI changes the duty cycle of the 2
// motors according to a PI control law
void MotorPI(float vref, float tref,float *DISTpos,
    float *THETApos, char zeroAllIntegrals) {

    if(zeroAllIntegrals)
    {I1 =0.0; I2 = 0.0; Iturn = 0.0;}
```

```
  oldenc1 = enc1; oldenc2 = enc2; //Store previous velocity calc

  enc1 = Read_Enc_Counter(0);     // Read encoder 1 count
  enc2 = Read_Enc_Counter(1);     // Read encoder 2 count

  v1temp = v1;  v2temp = v2;  // Store in case of rollover

  // velocity (ft/s)
  v1 = (((enc1-oldenc1)*(2.0*PI/564.0))/0.05)*(2.0/12.0);
  v2 = -(((enc2-oldenc2)*(2.0*PI/564.0))/0.05)*(2.0/12.0);

  if(fabsf(v1)>50) v1 = v1temp;       // rollover protection
  if(fabsf(v2)>50) v2 = v2temp;       // rollover protection

  *DISTpos = *DISTpos + (v1+v2)*0.025;
  *THETApos = *THETApos- (v1-v2)* 0.10975;
  // Store previous errors
  olde1 = e1; olde2 = e2; olde_turn = e_turn; /

e_turn = tref - v2 + v1;         // Steering error: v2-v1 -> tref

  // Integrate e1 using trapezoid rule
  Iturn += (e_turn+olde_turn)*0.10;



  e1 = vref - v1 - Kp_turn*e_turn - Ki_turn*Iturn;
  e2 = vref - v2 + Kp_turn*e_turn + Ki_turn*Iturn;

  // Integrate using trapezoid rule
  I1 += ((e1+olde1)/2.0)*0.05;
  I2 += ((e2+olde2)/2.0)*0.05;

  // Compute control effort (PI control)
  u1 = Kp*e1 + Ki*I1;
  u2 = Kp*e2 + Ki*I2;

  // Friction compensation
  if(v1 > 0) u1 += 0.6*(cpos + vpos*v1);
```

```
    else if(v1 < 0) u1 += 0.6*(cneg + vneg*v1);

    if(v2 > 0) u2 += 0.6*(cpos + vpos*v2);
    else if(v2 < 0) u2 += 0.6*(cneg + vpos*v2);

  // Handle saturation
  if(u1 > 10) {
    I1 *= 0.90;
    //I1 -= Ka*(u1-10.0);
    u1 = 10;
  }
  else if(u1 < -10) {
    I1 *= 0.90;
    //I1 -= Ka*(u1+10.0);
    u1 = -10;
  }

  if(u2 > 10) {
    I2 *= 0.90;
    //I2 -= Ka*(u2-10.0);
    u2 = 10;
  }
  else if(u2 < -10) {
    I2 *= 0.90;
    //I2 -= Ka*(u2+10.0);
    u2 = -10;
  }

    Motor_PWM(1,u1);
    Motor_PWM(2,-u2);
}
```

References

[1] A. Becker, C. Onyuksel, and T. Bretl, "Feedback control of many differential-drive robots with uniform control inputs," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2012 (under review).

[2] G. E. Dullerud and F. G. Paganini, *A Course in Robust Control Theory: A Convex Approach*. [Online]. Available: http://www.loc.gov/catdir/enhancements/fy0816/99046358-d.html: New York: Springer, 2000, vol. 36.

[3] K. Zhou and J. Doyle, *Essentials of Robust Control*. Prentice Hall Inc, 1998.

[4] R. M. DeSantis, "Modeling and path-tracking control of a mobile wheeled robot with a differential drive," *Robotica*, vol. 13, no. 04, pp. 401–410, 1995. [Online]. Available: http://dx.doi.org/10.1017/S026357470001883X

[5] Y. Chung, C. Park, and F. Harashima, "A position control differential drive wheeled mobile robot," *Industrial Electronics, IEEE Transactions on*, vol. 48, no. 4, pp. 853 –863, Aug. 2001.

[6] P. Lucibello and G. Oriolo, "Robust stabilization via iterative state steering with an application to chained-form systems," *Automatica*, vol. 37, pp. 71–79, 2001.

[7] M. Bowling and M. Veloso, "Motion control in dynamic multi-robot environments," in *Computational Intelligence in Robotics and Automation, 1999. CIRA '99. Proceedings. 1999 IEEE International Symposium on*, 1999, pp. 168 –173.

[8] M. Egerstedt and X. Hu, "Formation constrained multi-agent control," *Robotics and Automation, IEEE Transactions on*, vol. 17, no. 6, pp. 947 –951, Dec. 2001.

[9] Y. C. Tan and B. Bishop, "Evaluation of robot swarm control methods for underwater mine countermeasures," in *System Theory, 2004. Pro-*

ceedings of the Thirty-Sixth Southeastern Symposium on, 2004, pp. 294 – 298.

[10] R. Olfati-Saber, "Flocking for multi-agent dynamic systems: algorithms and theory," *Automatic Control, IEEE Transactions on*, vol. 51, no. 3, pp. 401 – 420, Mar. 2006.

[11] M. Erdmann and M. Mason, "An exploration of sensorless manipulation," *IEEE J. Robot. Autom.*, vol. 4, no. 4, pp. 369–379, Aug. 1988.

[12] K. F. Böhringer, V. Bhatt, B. R. Donald, and K. Goldberg, "Algorithms for sensorless manipulation using a vibrating surface," *Algorithmica*, vol. 26, no. 3, pp. 389–429, 2000. [Online]. Available: http://dx.doi.org/10.1007/s004539910019

[13] S. Akella and M. T. Mason, "Orienting toleranced polygonal parts," *The International Journal of Robotics Research*, vol. 19, no. 12, pp. 1147–1170, 2000. [Online]. Available: http://ijr.sagepub.com/content/19/12/1147.abstract

[14] C. C. Cheah, C. Liu, and J. J. E. Slotine, "Adaptive tracking control for robots with unknown kinematic and dynamic properties," *The International Journal of Robotics Research*, vol. 25, no. 3, pp. 283–296, 2006. [Online]. Available: http://ijr.sagepub.com/content/25/3/283.abstract

[15] W.-J. Mao, "Robust stabilization of uncertain time-varying discrete systems and comments on "An improved approach for constrained robust model predictive control," *Automatica*, vol. 39, no. 6, pp. 1109 – 1112, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0005109803000694

[16] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, "Mobile robots," in *Robotics: Modelling, Planning and Control*, 2nd ed., ser. Advanced Textbooks in Control and Signal Processing. Springer, 2009, ch. 11.

[17] R. W. Brockett and N. Khaneja, "On the stochastic control of quantum ensembles," in *System Theory: Modeling, Analysis and Control*, T. Djaferis and I. Schick, Eds. Kluwer Academic Publishers, 1999.

[18] N. Khaneja, "Geometric control in classical and quantum systems," Ph.D. dissertation, Harvard University, 2000.

[19] J.-S. Li and N. Khaneja, "Ensemble controllability of the bloch equations," in *IEEE Conf. Dec. Cont.*, San Diego, CA, Dec. 2006, pp. 2483–2487.

[20] J.-S. Li and N. Khaneja, "Control of inhomogeneous quantum ensembles," *Physical Review A (Atomic, Molecular, and Optical Physics)*, vol. 73, no. 3, p. 030302, 2006.

[21] S. Li, "A new perspective on control of uncertain complex systems," in *IEEE Conf. Dec. Cont.*, Dec. 2009, pp. 708–713.

[22] J.-S. Li, "Control of inhomogeneous ensembles," Ph.D. dissertation, Harvard University, May 2006.

[23] J.-S. Li and N. Khaneja, "Ensemble control of bloch equations," *IEEE Trans. Autom. Control*, vol. 54, no. 3, pp. 528–536, Mar. 2009.

[24] J.-S. Li and N. Khaneja, "Ensemble control of linear systems," in *IEEE Conf. Dec. Cont.*, New Orleans, LA, USA, Dec. 2007, pp. 3768–3773.

[25] J.-S. Li, "Ensemble control of finite-dimensional time-varying linear systems," *IEEE Trans. Autom. Control*, vol. 56, no. 2, pp. 345–357, Feb. 2011.

[26] A. Becker and T. Bretl, "Motion planning under bounded uncertainty using ensemble control," in *RSS, Zaragosa Spain*, 2010.

[27] A. Becker and T. Bretl, "Approximate steering of a unicycle under bounded model perturbation using ensemble control," *IEEE Trans. Robot.*, vol. 28, no. 3, pp. 580–591, 2012.

[28] M. Spong, "The swing up control problem for the acrobot," *Control Systems, IEEE*, vol. 15, no. 1, pp. 49 –55, Feb. 1995.

[29] X. Xin and M. Kaneda, "A new solution to the swing up control problem for the acrobot," in *SICE 2001. Proceedings of the 40th SICE Annual Conference. International Session Papers*, 2001, pp. 124 –129.

[30] G. Oriolo and Y. Nakamura, "Free-joint manipulators: motion control under second-order nonholonomic constraints," in *Intelligent Robots and Systems '91. 'Intelligence for Mechanical Systems, Proceedings IROS '91. IEEE/RSJ International Workshop on*, Nov. 1991, pp. 1248 –1253 vol.3.

[31] K. Lynch, "Locally controllable manipulation by stable pushing," *Robotics and Automation, IEEE Transactions on*, vol. 15, no. 2, pp. 318 –327, Apr. 1999.

[32] P. Song and V. Kumar, "A potential field based approach to multi-robot manipulation," in *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, vol. 2, 2002, pp. 1217 –1222.

[33] J. Fink, M. Hsieh, and V. Kumar, "Multi-robot manipulation via caging in environments with obstacles," in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, May 2008, pp. 1471 –1476.

[34] P.-T. Chiang, J. Mielke, J. Godoy, J. M. Guerrero, L. B. Alemany, C. J. Villagómez, A. Saywell, L. Grill, and J. M. Tour, "Toward a light-driven motorized nanocar: Synthesis and initial imaging of single molecules," *ACS Nano*, vol. 6, no. 1, pp. 592–597, Feb. 2011.

[35] B. Donald, C. Levey, and I. Paprotny, "Planar microassembly by parallel actuation of MEMS microrobots," *Microelectromechanical Systems, Journal of*, vol. 17, no. 4, pp. 789–808, Aug. 2008.

[36] D. Block, "The segbots (two-wheeled balancing robots)," 2012, Control Systems Laboratory, University of Illinois, Urbana, IL. [Online]. Available: http://coecsl.ece.illinois.edu/projects.html

[37] Y. Shirai, A. J. Osgood, Y. Zhao, K. F. Kelly, and J. M. Tour, "Directional control in thermally driven single-molecule nanocars," *Nano Letters*, vol. 5, no. 11, pp. 2330–2334, Feb. 2005.

[38] B. Donald, C. Levey, C. McGray, I. Paprotny, and D. Rus, "An untethered, electrostatic, globally controllable MEMS micro-robot," *J. of MEMS*, vol. 15, no. 1, pp. 1–15, Feb. 2006.

[39] S. Floyd, E. Diller, C. Pawashe, and M. Sitti, "Control methodologies for a heterogeneous group of untethered magnetic micro-robots," *I. J. Robotic Res.*, vol. 30, no. 13, pp. 1553–1565, Nov. 2011.

[40] E. Diller, S. Floyd, C. Pawashe, and M. Sitti, "Control of multiple heterogeneous magnetic microrobots in two dimensions on nonspecialized surfaces," *IEEE Trans. Robot.*, vol. 28, no. 1, pp. 172–182, Feb. 2012.

[41] A. Sebastian and S. Salapaka, "Design methodologies for robust nano-positioning," *Control Systems Technology, IEEE Transactions on*, vol. 13, no. 6, pp. 868 – 876, Nov. 2005.

[42] S. Bashash and N. Jalili, "Robust multiple frequency trajectory tracking control of piezoelectrically driven micro/nanopositioning systems," *IEEE Trans. Control Syst. Technol.*, vol. 15, pp. 867 – 878, Sept. 2007.

[43] K.-F. Bohringer, K. Goldberg, M. Cohn, R. Howe, and A. Pisano, "Parallel microassembly with electrostatic force fields," in *Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on*, vol. 2, May 1998, pp. 1204 –1211 vol.2.

[44] E. Schaler, M. Tellers, A. Gerratt, I. Penskiy, and S. Bergbreiter, "Toward fluidic microrobots using electrowetting," in *IEEE International Conference on Robotics and Automation*, May 2012.

[45] H.-W. Tung, D. R. Frutiger, S. Pan, and B. J. Nelson, "Polymer-based wireless resonant magnetic microrobots," in *IEEE International Conference on Robotics and Automation*, May 2012.

[46] Y. Ou, D. H. Kim, P. Kim, M. J. Kim, and A. A. Julius, "Motion control of tetrahymena pyriformis cells with artificial magnetotaxis: Model predictive control (mpc) approach," in *IEEE International Conference on Robotics and Automation*, May 2012.

[47] W. Hu, K. S. Ishii, and A. T. Ohta, "Micro-assembly using optically controlled bubble microrobots in saline solution," in *IEEE International Conference on Robotics and Automation*, May 2012.

[48] K. S. Ishii, W. Hu, and A. T. Ohta, "Cooperative micromanipulation using optically controlled bubble microrobots," in *IEEE International Conference on Robotics and Automation*, May 2012.

[49] A. M. Lyapunov, translated and edited by A.T. Fuller, *The General Problem of the Stability of Motion*. London: Tayor & Francis, 1992.

[50] K. Beauchard, J.-M. Coron, and P. Rouchon, "Time-periodic feedback stabilization for an ensemble of half-spin systems," in *IFAC Sym. Nonlin. Cont. Sys.*, Bologna: Italy, Sept. 2010.

[51] J. P. LaSalle, "Some extensions of Liapunov's second method," *IRE Transactions on Circuit Theory*, vol. CT, no. 7, pp. 520–527, Dec. 1960.

[52] W. S. Levine, Ed., *The Control Handbook*. United States of America: CRC Press, Inc., 1996, ch. 25.3 Discrete-Time Linear Time-Varying Systems, pp. 459–463.

[53] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, Sept. 2005.

[54] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, and S. Thrun, *Principles of Robot Motion*. The MIT Press, 2005.

[55] Y. Koren and J. Borenstein, "Potential field methods and their inherent limitations for mobile robot navigation," in *Proceedings of the IEEE Conference on Robotics and Automation*, April 1991, pp. 1398–1404.

[56] J. Borenstein and L. Feng, "Measurement and correction of systematic odometry errors in mobile robots," *IEEE Trans. Robot. Autom.*, vol. 12, no. 6, pp. 869–880, Dec. 1996.

[57] M. Sitti, "Survey of nanomanipulation systems," in *Proceedings of the 2001 1st IEEE Conference on Nanotechnology*, 2001, pp. 75–80.

[58] M.-F. Yu, "Fundamental studies of nanoscale sensing and actuation based on nanomanipulation and assembly," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2003, pp. 2365–2370.