EXPERIMENTS IN ROBOT FORMATION CONTROL WITH AN EMPHASIS ON DECENTRALIZED CONTROL ALGORITHMS

by

Joshua Cook

B.S., Kansas State University, 2008

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Mechanical Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2011

Approved by:

Major Professor
Dr. Guoqiang Hu

Copyright

Joshua Cook

2011

Abstract

In this thesis, several algorithms and experiments involving the control of robot formations are presented. The algorithms used were focused on decentralized control. The experiments were implemented on two different experimental testbeds consisting of teams of wheeled mobile robots. The robots used are described along with their sensors and supporting hardware. Also, there is a discussion of the programming framework used to build the control software.

The first control algorithm and experiment uses a robust consensus tracking algorithm to control a formation of robots to track a desired trajectory. The robots must maintain the correct formation shape while the formation follows the trajectory. This task is complicated by limited communication between the robots, and disturbances applied to the information exchange. Additionally, only a subset of the robots have access to the reference trajectory. In the second experiment, the same algorithm was re-implemented in a decentralized way, which more effectively demonstrated the goals of the algorithm.

The second algorithm involves controlling a formation of robots without a global reference frame. In order to accomplish this, the formation description is reformulated into variables describing the relative positions of the robots, and vision-based measurements are used for control. A homography-based technique is used to determine the relative positions of the robots using a single camera. Then a consensus tracking controller similar to the one used previously is used to distribute the measured information to all of the robots. This is done despite the fact that different parts of the information are measured by different agents.

Table of Contents

Table of Contents iv

List of Figures vi

List of Tables vii

1 Introduction 1
1.1 Overview . . . 1
1.2 Project Areas . . . 2
1.3 Literature Review . . . 4

2 Experimental Testbed 7
2.1 Robot Platforms . . . 7
2.1.1 AmigoBot . . . 7
2.1.2 Pioneer 3-DX . . . 9
2.2 Sensors . . . 10
2.2.1 Encoders . . . 10
2.2.2 Sonar . . . 10
2.2.3 Cameras . . . 11
2.3 Software . . . 12
2.4 Previous Work . . . 15
2.5 Preliminary Experiments . . . 16
2.5.1 Simultaneous Robot Control . . . 16
2.5.2 Sonar Characterization . . . 17

3 Robust Consensus Formation Tracking Control Experiment 20
3.1 Problem Description . . . 20
3.2 Theoretical Development . . . 21
3.2.1 Robust Consensus Tracking . . . 21
3.2.2 Motion Controller . . . 23
3.3 Experiment . . . 25
3.3.1 Hardware Implementation . . . 25
3.3.2 Software Implementation . . . 26
3.3.3 Implementation Challenges . . . 27
3.3.4 Experimental Parameters . . . 29
3.3.5 Experimental Results . . . 32
3.4 Distributed Experiment . . . 34
3.4.1 Hardware Implementation . . . 34

3.4.2 Software Implementation 35
3.4.3 Implementation Challenges 36
3.4.4 Experimental Parameters 38
3.4.5 Experimental Results 38

4 Formation State Description and Measurement 42
4.1 Motivation 42
4.2 Formation State Description 43
4.3 Measurement Rules and Assumptions 46
4.4 Image Measurement 48

5 Vision-Based Formation Tracking Experiment With Consensus 54
5.1 Problem Description 54
5.2 Consensus Tracking Protocol 55
5.2.1 Model 56
5.2.2 Control Objective 57
5.2.3 Consensus Protocol Design and Error System 57
5.2.4 Stability Analysis 60
5.3 Robot Motion Control 63
5.3.1 Relative Agent Dynamics 64
5.3.2 Low-Level Motion Controller 66
5.4 Hardware Implementation 66
5.5 Software Implementation 68
5.6 Implementation Challenges 69
5.7 Experimental Setup 71
5.8 Experimental Results 74

6 Conclusion 80

Bibliography 85

A Code For Distributed Formation Tracking Experiment 86

List of Figures

2.1 AmigoBot Robot . . . 8
2.2 Pioneer 3-DX Robot . . . 9

3.1 General Experimental Framework . . . 21
3.2 Snapshot of Formation Shape . . . 24
3.3 Wheeled Mobile Robot Schematic . . . 25
3.4 Experimental Hardware Configuration . . . 26
3.5 Control Software Structure . . . 27
3.6 Information Exchange Graph . . . 30
3.7 Consensus Tracking Results $p_{xci}(t), p_{yci}(t)$ . . . 33
3.8 Consensus Tracking Results θ_{ci} . . . 33
3.9 Trajectory Tracking Results . . . 34
3.10 Formation Tracking Error . . . 35
3.11 Distributed Experimental Hardware Configuration . . . 36
3.12 Distributed Control Software Structure . . . 37
3.13 Consensus Tracking Results $p_{xci}(t), p_{yci}(t)$ . . . 39
3.14 Consensus Tracking Results θ_{ci} . . . 40
3.15 Trajectory Tracking Results . . . 40
3.16 Formation Tracking Error . . . 41

4.1 Fully Described Robot Formation . . . 45
4.2 Image Measurement Coordinate Frames . . . 49
4.3 Relationships Between Relative Position Orientation and States . . . 53

5.1 General Experimental Framework . . . 55
5.2 Relative Agent Dynamics . . . 64
5.3 Hardware Configuration . . . 67
5.4 Formation Initial Configuration . . . 72
5.5 Formation Final Configuration . . . 73
5.6 Consensus Tracking Results d_{12}, d_{23} . . . 75
5.7 Consensus Tracking Results β_{12}, β_{23} . . . 76
5.8 Consensus Tracking Results θ_{12}, θ_{23} . . . 77
5.9 Visual Formation Tracking Error . . . 77
5.10 Visual Formation Consensus Error . . . 78
5.11 Visual Formation Consensus Estimator . . . 78
5.12 Robot Velocity Command . . . 79
5.13 Robot Angular Velocity Command . . . 79

List of Tables

3.1 Experimental Control Gains 31
3.2 Consensus Initial Conditions 31
3.3 Postion Initial Conditions 32

4.1 Relationships Between Coordinate Frames 50

5.1 Experimental Control Gains 74

Chapter 1

Introduction

1.1 Overview

As robots become ever more useful in today's society, the demand for them to do more continues to grow. Some of the potential applications of robots do not always require a single, incredibly complex robot. Instead, sometimes, a task can be best achieved by a team of simpler robots working together. These teams working together in the real world present several new and interesting challenges. Working together requires communication and coordination that can be difficult in real environments. Because of the challenges and potential benefits, control of teams of robots is an active area of research. As new theories are developed for coordination and control of teams of robots, experiments need to be done to evaluate their effectiveness.

In this thesis several algorithms for the control of formations of mobile robots are presented. Then experiments and their results are given to demonstrate the effectiveness of the proposed algorithms. To conduct these experiments, a experimental test bed consisting of several wheeled mobile robots was used. In fact two different test beds were used. One test bed had a larger number of smaller simpler robots, while the other had a fewer number of larger more advanced robots. Both test beds are equipped with a variety of sensors, different sensors were used in different experiments. Some of the algorithms and experiments used cameras as sensors.

One of the concepts which has gained significant support and interest in the field of multi-robot control is decentralized control. In decentralized control, there is no central global decision making, and the control decisions are made in a distributed manner based on locally available information. There are many benefits of decentralized control over centralized control. One of the commonly cited benefits is it can make a system robust to failure of part of the system. In a centralized control system, if the central controller or the communication from the controller to the rest of the system fails, the entire system fails. In a decentralized control system, if part of the system fails it will not necessarily bring down the rest of the system. Another benefit is that such a system can reduce the load and requirements on the communication system. Due to the benefits of decentralized control, the algorithms in this thesis focus on a decentralized approach.

1.2 Project Areas

Over the course of the project, several different areas of research were explored. Some became the focus of this document, and others served as a basis for later work. Other areas of research were explored briefly and abandoned to focus on the main topic of research. This section briefly outlines some of the areas of work that were explored. It also discusses some projects which were begun or investigated, but did not fit with the rest of the research.

One of the earliest projects focused on practical implementation. The robots that were going to be used in experiments had been purchased, and some work had been done on controlling individual robots. However, it was desired to be able to control multiple robots simultaneously from one control program. This goal lead directly into later experiments which required controlling a team of robots. This task was accomplished relatively easily, but required gaining a greater understanding of the robot control software, and served as a basis for later projects.

Once a team of robots could be controlled simultaneously, work was done to try and apply this to coordination of multiple robots. There are many different types of coordi-

nation. Some examples include: follow-the-leader algorithms, formation control, confining an uncooperative agent, etc. This led to the evaluation of the robot's sensors, and their usefulness in these sort of tasks. Eventually it was decided to focus on the formation control problem, and consensus-based algorithms in particular.

Another area of research which was investigated was human control of robots. In particular, using brain waves or electrical impulses from a human face to control a robot. One of the obvious applications of technology such as this include controlling a robotic wheelchair. To investigate this concept a commercial device was purchased. This device, called a Neural Impulse Actuator by the manufacturer, can measure electrical and brain wave data from sensors on an operator's face, and then translate this to key strokes. The sensors were mounted to a headband worn by the user. This headband was connected to a device that communicated with software installed on a computer. There was some success in controlling a robot with the device, but the product's proprietary interface limited the work that could be done, and eventually efforts were focused elsewhere.

Eventually the primary focus of the work became implementing consensus algorithms. In particular, robust consensus-based formation control algorithms. This was built on the earlier work on controlling multiple robots. An experiment was developed demonstrating a robust consensus tracking algorithm for formation control that was robust to disturbances in the information exchange. Later, this experiment was re-implemented using network communication for the consensus exchange instead of controlling all of the robots from the same application.

It was decided to incorporate cameras into the experiments. Initially, there was a plan to use a camera in a localization system. The camera would be fixed in some location where it would be able to see the space where the robots were operating. Then, using image processing, accurate locations for each of the robots could be determined. This configuration, where a fixed camera is used to control the motion of something in its field of view, is referred to as a camera-to-hand configuration. The other popular configuration is where the camera

is fixed to the object it is being used to control. This configuration is known as camera-in-hand. It was decided that camera-in-hand provided a better opportunity to further the main focus of the research, so work on the camera-to-hand configuration was abandoned. This switch did require acquiring new experimental hardware to accommodate the cameras mounted to the robots.

In order to sort out implementation challenges and prove that the experimental hardware worked, the first experiment using a camera mounted to the robot was fairly simple. The camera was used to make the robot follow a desired trajectory and approach a visual target. An existing vision-based control algorithm based on adaptive control and homography techniques was used. The lessons learned were of vital importance for later vision-based control experiments.

Finally, vision-based control was incorporated with the formation control and consensus algorithms. This new experiment used cameras and homography techniques for measurements, and robust consensus for distributing information. It then used this information to control a team of robots into a desired formation.

1.3 Literature Review

There are many applications where cooperating mobile robots could be a good solution. A few of these have been discussed in the literature. One example presented in the paper [1] uses a team of robotic highway safety cones to mark out construction sites. Each cone contains a simple robot which must work with the others to close down a lane for highway maintenance or other activities. Papers such as [2] and others examine the problem of controlling formations of spacecraft such as satellites. Another area of application involves distributed sensor networks where there are many agents each equipped with sensors. The paper [3] is an example of this, where a formation of robots tracks the maximum or minimum of some sensed variable in the environment. Another example of a distributed sensing application is the paper [4] which looks into the problem of distributing optimally agents

with sensors in an environment. There are many other potential applications.

Several different approaches have been explored for controlling formations of robots. One approach is the behavior-based method which is discussed in papers such as [5–7]. In this approach different goals such as formation keeping, obstacle and collision avoidance, and navigation are implemented as separate simple behaviors. These behaviors are usually simple rule-based controllers. Their outputs are then averaged together using weights to determine the motion of the robots. This approach has several drawbacks, one of which is that it is very difficult to analyze or guarantee the stability of the formation.

Another approach is the virtual structure method as discussed in [8, 9]. A virtual structure is a concept where the desired positions of the robots are considered to be part of a structure. This structure is then moved or changed to move or change the formation, and each robot simply attempts to match it's position with its desired position in the structure. One of the drawbacks to this method is that the virtual structure needs to be calculated centrally and then communicated to the robots.

Leader-follower methods such as described in [10–12] have been studied extensively. In the leader follower framework one or more robots are designated as leaders, and others as followers. This means that separate controllers have to be developed for the leaders and followers. There a few drawbacks to this technique, in particular, the leaders often serve as a central point of failure for the formations.

The paper [13] discusses a synchronization approach to robot formation control. The idea is that each robot attempts to move to its own desired location while synchronizing its motion with its neighbors. This helps with maintaining formation shape and other constraints during a change in formation.

Another approach which has been studied is consensus-based formation control. In consensus, the robots make control decisions based on the states of their neighbors, and there is no central coordination of the formation. Often concepts from graph theory are used to model the connections between the robots, and to help analyze the stability of

the control algorithms. The paper [14] provides an overview of many of the concepts in consensus. In [15] the authors looked at proving the stability of a formation using graph-theory-based concepts to model the communications and measurements made by the robots. Then, they used a Nyquist-based criterion to examine the stability of the formations. In [16] the authors apply consensus to a leader follower problem, and develop a discontinuous controller to ensure convergence in finite time. The authors in [17] examine several different consensus algorithms to track a time-varying reference signal. This reference signal then serves as a virtual leader for the formation. There are many more examples of papers dealing with consensus.

Cameras and computer vision algorithms can provide a wealth of information as sensors. For this reason, they are often used in robotics for control. Several papers such as [10, 18] have used cameras and vision for robot formation control. Despite the benefits of cameras as sensors, they have several drawbacks. Such as limited field of view, and re-constructing three dimensional coordinates from a two dimensional image.

Chapter 2

Experimental Testbed

2.1 Robot Platforms

Two different types of robots were used in the experiments. Both were manufactured by MobileRobots inc. Specifically, the two models used were AmigoBots and the Pioneer 3-DX. Both are non-holonomic, wheeled, mobile robots. They have two independently controlled drive wheels and a rear caster for balance. Both have built in microcontrollers, but require an external computer to send them commands.

2.1.1 AmigoBot

The AmigoBot (pictured in Figure 2.1) is the smaller and simpler robot of the two used in the experiments. Its small size lends it to being used in experiments on multi-agent systems. However, its small size limits its payload carrying capacity significantly. Because of this, they are usually used in conjunction with a separate computational system such as an off-board computer.

The robots themselves have simple microcontrollers that control all of their hardware. The motion commands sent to the robot are executed using a PID controller [19]. The microcontroller uses optical encoders on the wheel motors to calculate the position and velocity of the robot. Also connected to the on-board controller is a WiBox wireless network adapter. This adapter connects to the controller through a standard serial interface, and is capable

Figure 2.1: *AmigoBot Robot*

of forwarding serial information through an 802.11b wireless Ethernet network. Through this interface the robot can send and receive packets of information. In the communication model used, the robot sets itself up as a server which sends out Server Information Packets (SIP). Then separate clients can receive these packets, and send out their own packets known as command packets. The client in our case is a PC running robot control software. The separate client is where the high level thinking and control takes place. The client (PC running software) receives information from the server (AmigoBot), decides what it wants the robot to do, and sends it commands in a command packet. This process continues while there is a connection. For this purpose, an Ad-Hoc wireless network is setup. Then both the robot, and the PC running the client software connect to this network. The client software then attempts to establish a TCP connection to the server. After they are connected they can begin to exchange packets.

The microcontroller used in the AmigoBots is a 32-bit Renesas SH2-7144 RISC microprocessor [19]. They are equipped with two built in 12 Volt, sealed, lead-acid batteries that

provide the robot and any accessories with a total of 55.2 watt-hours.

2.1.2 Pioneer 3-DX

The Pioneer 3-DX, pictured in Figure 2.2, has many of the same features as the AmigoBots. Unlike the AmigoBots they do not have the built in wireless adapter, they must communicate with the controlling computer through the serial port. They are larger, have a bigger payload capacity, and better battery life. They have larger tires, and can handle rougher terrain.

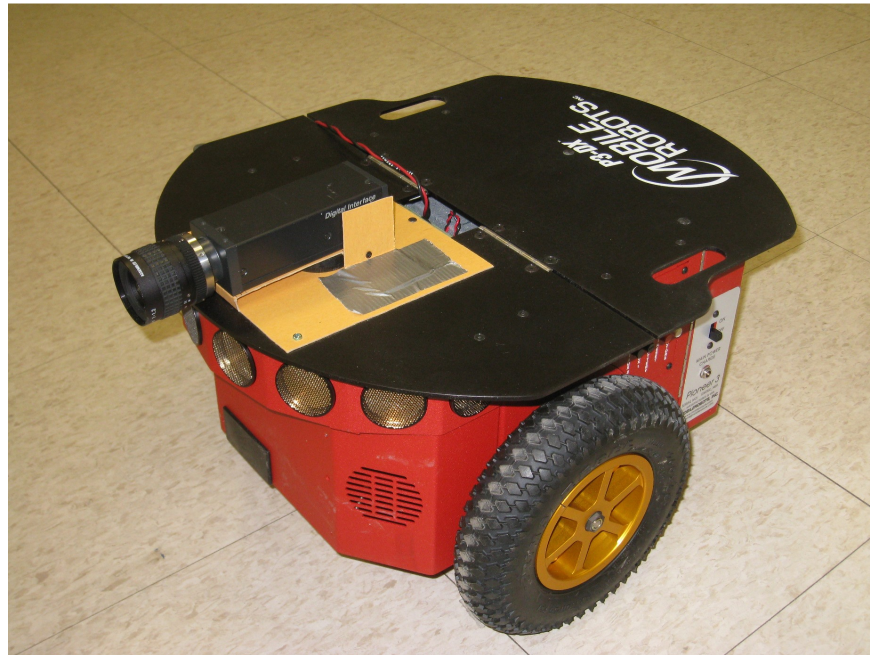

Figure 2.2: *Pioneer 3-DX Robot*

To connect to the computers a USB to RS-232 adapter was used. These have been known to cause problems with dropped information from time to time. These had to be used because the laptops did not have serial ports.

The Pioneer 3-DX comes equipped with the same basic microcontroller as is used in the AmigoBots. For power, the Pioneer 3-DX robots can have up to three hot-swappable batteries installed at a time. The batteries are 12 Volt sealed lead acid batteries, and with three installed, they provide a total of 252 watt-hours for the robot and accessories [20].

2.2 Sensors

The AmigoBot robots come with two installed sensors. The motor encoders, and the sonar array. The encoders are used to calculate the robot's approximate displacement and heading from a starting position, and its velocity. The sonar currently are only used for obstacle avoidance routines. The P3-DX robots also came with two sensors, wheel encoders and a sonar array. Additionally cameras were mounted to the P3-DX robots to take visual measurements.

2.2.1 Encoders

Each robot has two linear encoders, one attached to each motor shaft. These are used for odometery to estimate the robot's position and orientation relative to its starting location. They are also used to calculate the velocity feedback for the on-board velocity control loop. The robot's microcontroller calculates the distance traveled and the velocity from the encoder readings. It then makes this information to any client software connected to the robot. The raw encoder data is also available if necessary.

2.2.2 Sonar

Each robot is equipped with an array of eight sonar sensors. Each sensor works by sending out a pulse of sound, which bounces off objects, and returns to the sensor. By timing how long it takes for the reflected sound to be picked up by the sensor, one can estimate the distance to the object off which the sound bounced. The sonar sensors are controlled by a specialized board inside the robot that relays the information to the robot's microcontroller. The microcontroller can then pass this information on to the client software. The sonar sensors fire in sequence, with a small delay between each firing. One sensor fires, then listens for a return, then the next sensor in the sequence fires. There are a few software controllable parameters for the sonar array. First, the sequence of firing for the sensors can be set. Second, the firing rate can be also be set in software. Lowering the firing rate gives

less time between successive readings on the same sensor, but also reduces the maximum range of the sensors.

The sonar sensors are very useful for determining the range from the robot to some object or obstacle in its environment. In order to determine the objects position relative to the robot more information is needed than just the range. To solve this problem, an array of sensors is used. Each sensor can only detect obstacles in a narrow region directly in front of the sensor. Using an array of such sensors, it is possible to determine the relative position of an object based on the range reading, and which sensor detected it. This is useful in tasks such as obstacle avoidance where the goal is simply to avoid driving too closely to any object in the environment.

There are several limitations of the sonar sensor. In particular it only gives range to an object, and gives us little to no information about that object. All we can really say with certainty is that it reflects sound waves. This is compared to cameras which can give many different pieces of information about an object, but have their own limitations, as well. Also, because of the narrow sensing region for the sonar, blind spots can occur where no sensor would detect an object. Certain objects do not reflect sound well or deflect it to a direction other than the one it came from. This results in certain objects such as small-diameter, vertical cylinders being invisible to the sonar sensors.

2.2.3 Cameras

Cameras can be very powerful sensors in robotics. They are capable of discerning a great deal about the environment, yielding a lot of information from each image. For the experiments where cameras were used, the cameras were mounted on top of the Pioneer 3-DX robots. The cameras used were Sony XCD-SX910 digital machine vision cameras. They communicate with computers over the firewire interface also knows as IEEE 1394. The XCD-SX910 is a black and white camera with a maximum image resolution of 1376x1024 with a maximum frame rate of 15 fps [21]. There are several selectable modes with different image resolutions,

and frame rates.

The cameras are designed to draw their power from the firewire cable. Unfortunately most laptop computers are not equipped with 6-pin powered firewire ports. Instead most only have the unpowered 4-pin firewire ports. This makes using the cameras along with on-board laptops problematic. One solution is to power the camera separately. The Pioneer 3-DX robots are designed to be expanded with attachments, so they are designed to provide power to external devices. The motor power board provides conditioned 5 Volts direct current at 1.5 Amps total and unconditioned battery power at approximately 12 Volts direct current for accessories [20]. The power specifications for the Sony XCD-SX910 specify that it requires from 8 to 30 Volts supplied by the firewire cable. Its power consumption is listed as 3.5 Watts at 12 Volts [21]. This implies that the unconditioned battery voltage supplied by the robots should be sufficient for powering the camera. To power the camera and have it connected to the computer at the same time, a 4 to 6-pin firewire connector was modified to tie the power pins on the 6-pin side to separate external supply wires. These supply wires were then connected to the robot's motor power board's accessory connections.

2.3 Software

Included with the robots was a variety of software for controlling the robots and for developing custom robot-control software. One particular software package of interest is ARIA or Advanced Robotics Interface for Applications, which serves as a programing interface for the robots. It's a set of C++ programs and header files which make it easy to write programs that interface with and control the robots. ARIA is open source, and is licensed under the GPL or General Public License. Also included was a program called MobileSim. MobileSim is a robot and environment simulator, allowing control software to be tested on virtual robots inside a virtual environment. Another API or Application Programming Interface that was used was OpenCV or Open Computer Vision. OpenCV is used for communicating with the cameras, and for most of the image processing tasks.

For each task required of the robots, a new control program was typically developed. These control programs are sometimes referred to as robot client software because of the way they communicate with the robot. As was mentioned before, the ARIA API was used for communicating with the robots, and many other tasks. These libraries are all written in C++ and make extensive use of object oriented programming. Each robot that the client will connect to is represented by an ArRobot class object. The connection to the robot is represented by an ArTcpConnection object which is then associated with the robot object. In some of the programs, the ArTcpConnection is used indirectly, and other classes such as ArRobotConnector are used to setup and establish the connection. The client software can easily connect to several robot servers by creating multiple ArRobot objects with a ArTcpConnection object for each one.

The ARIA software has three basic levels of commands it can send to the robot. The first are low-level, direct commands which directly set parameters on the robot. The second level is direct motion commands. These are more user friendly, but still fairly basic. They do things such as specify velocities for the wheels, or command the robot to drive a certain distance, etc. The third level is actions. Actions are more sophisticated commands that can contain logic and produce several different motions. You can also stack actions, and the ARIA software will decide what motion commands to send based on the motions requested by each action. The action structure proved to be useful for developing control programs, although the stacking feature was less useful.

These actions are custom classes which are children of the ArAction class. Actions are class objects which can store data, and have code made up of direct motion commands and logic. After an instance of an action is created, it is "attached" to a robot object. Afterward, whenever the robot communication loop runs, it executes that action's code. The code makes a decision on what direct motion commands it would like to send to the robot, such as a certain forward velocity, and returns this desired command to the "action resolver." The action resolver is a class object of ArActionResolver or a class inherited

from it, and its task is to take the desired commands from several actions and combine them, relying on a weighting system, into one command. This command is then sent to the robot. The multiple actions with weights system allows for several very simple actions to be combined to get more complicated behavior from the robot.

One important feature for some of the robot control software developed is multi-threading. Multi-threading allows a program to execute multiple blocks of code in parallel. The different tasks may not actually execute simultaneously depending on the number of processor cores in the computer, and other programs running at the time. ARIA includes several tools for working on multi-threaded programs. Its classes handle all of the platform specific details of creating and managing threads, making it easy to develop cross-platform, multi-threaded code. The most common example of threads in ARIA are robot objects. To communicate with and control the robots, a loop waits for a communication packet from the robot, then executes a series of tasks before sending a response packet to the robot and resumes waiting. Those tasks include interpreting the sensor data, any actions that have been added to the robot, and others. This loop can be run either synchronously in the thread where the object was originally created (usually the program's main thread) or asynchronously in it's own new thread. If this loop is run synchronously in the main thread, the no other tasks can be executed in that thread until it disconnects from the robot. Whereas if it is run asynchronously, the main thread can continue on executing other tasks such as connecting to other robots, or image processing.

A major issue that comes up in the development of multi-threaded programs is data access. If two threads running in parallel try to access or write to the same space in memory at the same time, the results will be unpredictable at best. Such an occurrence might crash the program or worse. To keep this from happening, the concept of locking is used. Locking allows for multiple threads to access the same data while preventing them from doing so at the same time. It is accomplished using an object known as a "mutex" which is short for mutual exclusion. The mutex has a state that is either locked or unlocked, and its state can

be changed using a function call. Before accessing any data, a thread will attempt to lock the mutex associated with that data, and unlock it after it is done. If the mutex has been locked by another thread, it will not allow the thread that attempted to lock it to proceed until the lock has been removed. In this way, as long as the threads always attempt to lock the mutex before accessing data, no two threads can access the data at the same time.

OpenCV was used to handle the access to the cameras. Also it provided a convenient interface for working with images, and matrices. Most importantly though, OpenCV includes many built in image processing algorithms. These include simple tasks such as image filtering, and scaling, and more complicated things like optical flow calculations, which is a method for tracking a series of points through successive images. Using several assumptions and the locations of the points in a previous image, it finds the points in a new image. This allows for the tracking of specific feature points in a video stream; giving the ability to track an object in the camera's field of view as the camera and object move.

2.4 Previous Work

Previously, some work had already been done with the experimental hardware. In particular, some work had been done with the AmigoBots described in Section 2.1.1. Most of this work focused on setting up the wireless network communications between the robots and a computer. First, the WiBox wireless Ethernet to serial adapters on board the robots were configured to use an "Ad-Hoc" network with a particular name. This process involved connecting to the wireless adapter from a computer over a serial connection. Then a Belkin 802.11g wireless USB network adapter was installed in a desktop computer, and configured to use the same Ad-Hoc network. This adapter was also configured to have a static IP address on the same subnet as the wireless adapters on the robots.

Most of the other preliminary efforts revolved around establishing a base for robot control program development. This included preparing a development environment on a computer with all of the appropriate libraries and ARIA installed. Then, several example programs

that came included with the ARIA source code were compiled. Once this development environment was established and tested, some of the example programs were modified to demonstrate the ability to create custom control programs. In particular, a simple program was created that gave a robot a series of direct motion commands.

2.5 Preliminary Experiments

Early on there were several tests and experiments that were focused on gaining understanding of the experimental hardware and software. These preliminary experiments also served as a base from which later experiments and code were built.

2.5.1 Simultaneous Robot Control

The simultaneousRobotControl program was designed to demonstrate the ability to send commands simultaneously to two robot servers from one client application. The goal was to have two different physical robots executing different and independent tasks while being controlled by a single application. This is made possible by the threaded nature of the program. The packet sending/receiving and action command processing for each robot is done in its own thread separate from the main program thread. This is accomplished by calling the ArRobot member function runAsync(). At that point the main program starts a thread for each robot, then waits for the robot threads to end. The robot threads are where the actual processing is performed. This concept would later be extended to controlling multiple robots, and allowing them to work collaboratively.

The concept of Actions in ARIA was used in this program. Each robot was given a set of two default actions, one to recover from stalls, and one to avoid obstacles in front of the robot. The obstacle avoidance action relied on the robot's sonar sensors to detect and measure the distance to obstacles. Additionally, the first robot was given an action that commanded it to simply drive in a circle continuously. This action was given a lower priority than the other two, so that the robot would stop turning if it stalled or encountered

an obstacle. Similarly, the second robot was given an action that has it drive a set distance, turn around, then repeat the process. These tasks demonstrated the use of the action framework, and clearly illustrates the ability to connect to multiple robots and give them distinct commands.

The application was a success, although several problems were encountered during the development. Up to this point most of the work had used ARIA helper classes to simplify parsing options from the command line and establishing the network connection to the robot. Unfortunately, these helper classes assumed that the application would connect to only one robot. After a careful search of the ARIA documentation and code, the underlying tools used were identified, and the command line parsing and network connections were implemented in a way that accommodated several robots. Also, there were issues in programing the action that drove back and forth in a straight line. The odometery from the wheel encoders was used to determine when the appropriate distance had been traveled, and the current angle of the robot. This proved to be somewhat unreliable, particularly the angle. As a result, the robot would not always turn around a full 180 degrees, thereby not following the same path every time as was intended.

2.5.2 Sonar Characterization

In order to evaluate the usefulness of the sonar sensors installed on the robots, several experiments were conducted to measure their performance. Of particular interest was the feasibility of using them to measure the locations of other robots or similar objects in their environment. Some of the factors which would affect this include the maximum and minimum range of the sensors, and how often the data is updated. Also important is how narrow of a sector can each sensor see. This information can be used to determine if the sonar array has blind spots, and how accurately the position of an object can be determined.

The sonar data collected by the robot is sent to the client computer in each SIP or Server Information Packet. When the client receives a SIP it starts its robot task processing cycle.

In order to save the sensor readings from the sonar array, a “Sensor Interpretation Task” was added to this cycle that writes the raw sensor readings and the times they were taken to a file. These readings are thereby saved for later analysis.

The setup for the range verification tests was fairly simple. A single amigoBot was placed on a flat surface, and a vertical sonar target was placed in the path of one of the sonar sensors a known distance from the robot. The robot and its sonar array were then activated, and the returning data was collected for several seconds. The target was moved to a different distance from the robot, and the process was repeated. Both the robot and the target remained stationary during the data collection. The experiment was first run with the default sonar polling rate of 40ms between each of the eight sensors. At this rate each individual sensor obtains a new reading every 320ms. The sonar data is sent in a packet from the robot server to our client software at a rate of 100ms, so only every third or forth packet contained updated readings. To get a better data sampling rate, a polling rate of 12ms was used next. This corresponds to a new individual sensor reading every 96ms; therefore, each packet from the robot should contain an updated sensor reading. The same experiment was repeated with this new sample rate.

From these tests, it was evident that the polling rate has a significant affect on the maximum range. For the default rate of 40ms the maximum range is approximately 5 meters while at a rate of 12ms the maximum range is decreased to approximately 2 meters. This is a significant reduction in the maximum range capabilities of the sonar sensors. The minimum range appears to be unaffected by polling rate, and was found to be approximately 0.2m. The sonar still detects objects closer than this, but it reports their range as the minimum.

The setup for the angular width determination test was very similar to the setup above. The primary difference in this experiment was that the target was at a fixed range from the sensor, and instead of increasing its range it was moved in a circular arc with the sensor at the arc’s center. First the target was placed directly in front of the sensor then moved slowly off to the side increasing the distance between its edge and a line extending straight

out from the sensor. The object had a square corner, and the front face of the target was kept normal to a line extending from the sensor to this corner. Again, both the robot and target were stationary during data collection. At each target configuration, perpendicular distance between the target corner and the line extending out from the sensor was recorded. The default sonar polling rate of 40ms was used in the tests.

The data was analyzed to determine the largest angle at which the sonar sensor still returned a valid range. Then the next highest angle for which data was recorded was determined. These two angles bound the maximum angle away from its normal at which at which a single sensor can still detect an object. Data was taken at two different ranges. One set of data was taken at approximately .6 meters and the other at approximately 1.8m. At 0.6m the maximum angle was between 5.85 and 7.03 degrees. At 1.8m the maximum angle was between 6.32 and 6.72 degrees. The second set of measurements yielded a more accurate estimate of the angle, but both sets agree. The data shows that the maximum angle from the sensor normal at which the sensor can pick up an object is approximately $\pm$ 6.5 degrees or a total angular width of 13 degrees.

From the results of these two tests, we can construct a graph of the regions around the robot where it can detect an object with the sonar. We can also identify any blind spots that the sonar sensors may have. From the robot parameter file we can obtain the (x,y) position and heading of each sensor.

Chapter 3

Robust Consensus Formation Tracking Control Experiment

3.1 Problem Description

In this chapter, an algorithm and results are presented for multi-robot formation control. Some of the work in this chapter was presented in [22]. We consider a team of wheeled mobile robots that can move in a plane. The goal is for these robots to maintain a desired formation shape while the formation follows a time-varying desired trajectory. Each of the robots or agents makes independent control decisions. To coordinate the agents' actions, a robust consensus tracking algorithm is used. Each agent implements the consensus algorithm to track the formation's desired trajectory. From the formation's desired trajectory, each agent calculates its own desired trajectory using a previously known desired formation shape. Finally, each robot implements a motion controller to follow this desired trajectory.

The problem is complicated by a limited information exchange between the robots. Each robot can only share or receive information from its neighbors in the formation. Also, the time-varying, actual, desired trajectory for the formation is only provided to a subset of the agents. Disturbances that represent both external disturbances and unmodeled dynamics are applied to the information exchange dynamics, which is why a robust consensus algorithm is needed.

Initially, this experiment was implemented with all of the agents controlled by one in-

stance of the control software running on one computer. This simplified the experiment, and removed the need for multiple computers. Having all the agents controlled by one computer is inherently centralized, and removed possible network effects. To address these problems, the control software was rewritten to be implemented so that each agent was controlled by a different computer. This brought up several challenges that had to be addressed.

3.2 Theoretical Development

Before the experiment was conducted, the supporting theory was developed. Figure 3.1 gives a general overview of the algorithm's structure. What follows is the theoretical background for the experiment, and an overview of the stability proof.

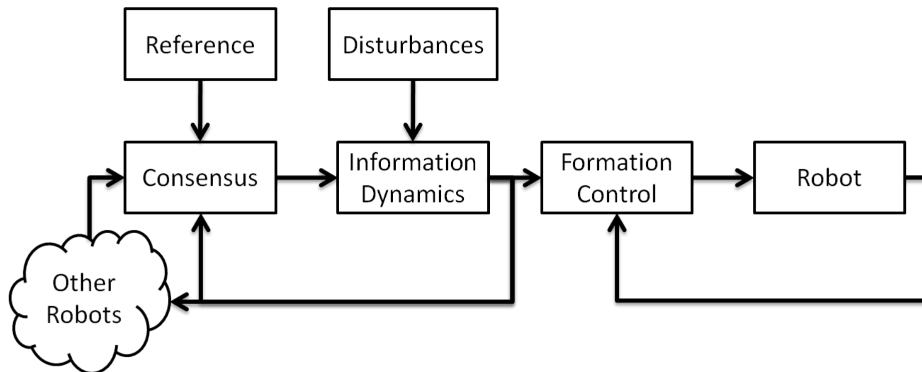

Figure 3.1: *General framework for the experiment*

3.2.1 Robust Consensus Tracking

The detailed theoretical development for the robust consensus tracking controller was previously published in [23].

The consensus controller deals with the information states for each agent. Equation 3.1 shows the information state dynamic model including disturbances.

$$\dot{x}_i = u_i + f_{ai} + f_{bi}(x_i), \quad i = 1, \ldots, N \tag{3.1}$$

In this equation $x_i(t) \in \mathbb{R}^n$ is agent i's information state, $u_i(t) \in \mathbb{R}^n$ is a control input to be designed, $f_{ai}(t) \in \mathbb{R}^n$ and $f_{bi}(x_i) \in \mathbb{R}^n$ are disturbances that are a function of time and the state respectively.

The control objective is that all of the information states converge and track the desired trajectory x_d.

The information exchange graph topology is modeled with an adjacency matrix $A \in \mathbb{R}^{N\times N}$ where $a_{ij} \neq 0$ if agent i receives information from agent j and N represents the number of agents. Access to the reference trajectory is represented by the diagonal matrix $B = diag\{b_1, b_2, \dots, b_N\} \in \mathbb{R}^{N\times N}$ where $b_i \neq 0$ if agent i has access to the reference signals. For this experiment it is assumed that the information exchange is an undirected graph. Meaning that if agent i receives information from agent j then agent j also receives information from agent i. This results in the A matrix being symmetric.

The control algorithm is presented below. First, a consensus tracking error for agent i is defined in Equation 3.2. Where x_i, x_j, and x_d are the agent's, the neighbor's, and the reference state respectively.

$$e_{fi} = \sum_{j=1}^{N} a_{ij}\left(x_i - x_j\right) + b_i\left(x_i - x_d\right) \tag{3.2}$$

Based on this error, a non-linear estimator term is designed to estimate the unknown disturbance terms in Equation 3.1.

$$\hat{f}_i = k_1\left(e_{fi} - e_{fi}(0)\right) + \int_0^t \left(k_2 \mathrm{sgn}(e_{fi}) + e_{fi}\right) d\tau \tag{3.3}$$

In this controller, the signum function $\mathrm{sgn}(e_{fi})$ gives the sign of the error, and is defined element-wise. k_1, and k_2 are positive constant control gains. Finally, these pieces are combined to design the control input u_i.

$$u_i = -\hat{f}_i(t) + b_i\dot{x}_d - k_c e_{fi} \tag{3.4}$$

This controller has three terms. The first term is the estimate of the disturbances to remove them from the dynamics. The second term is a feed-forward, velocity term based on the

reference signals. This term is only present if the agent has access to the reverence trajectory, as is indicated by the b_i. The last term is simply a proportional error term where k_c is another positive constant control gain.

3.2.2 Motion Controller

Once the agents determine the desired trajectory for the formation using the consensus algorithm, each agent calculates its own desired trajectory using a formation shape vector. This formation shape vector gives the position of that agent relative to the formation center. Equation 3.5 gives this calculation, and Equation 3.6 gives the desired speeds.

$$\begin{bmatrix} p_{xdi} \\ p_{ydi} \end{bmatrix} = \begin{bmatrix} p_{xci} \\ p_{yci} \end{bmatrix} + \begin{bmatrix} \cos(\theta_{ci}) & -\sin(\theta_{ci}) \\ \sin(\theta_{ci}) & \cos(\theta_{ci}) \end{bmatrix} \begin{bmatrix} d_{xi} \\ d_{yi} \end{bmatrix} \tag{3.5}$$

$$\begin{bmatrix} \dot{p}_{xdi} \\ \dot{p}_{ydi} \end{bmatrix} = \begin{bmatrix} \dot{p}_{xci} \\ \dot{p}_{yci} \end{bmatrix} + \dot{\theta}_{ci} \begin{bmatrix} -\sin(\theta_{ci}) & -\cos(\theta_{ci}) \\ \cos(\theta_{ci}) & -\sin(\theta_{ci}) \end{bmatrix} \begin{bmatrix} d_{xi} \\ d_{yi} \end{bmatrix} \tag{3.6}$$

Figure 3.2 shows the desired formation shape along with the reference trajectory for this experiment. The formation is a square that travels in a circle in such a way that the formation is aligned with the circle.

The robots used in these experiments are non-holonomic wheeled mobile robots. A simple schematic of the robots is shown in Figure 3.3. The kinematic model for robot i is given in Equation 3.7.

$$\begin{aligned} \dot{p}_{xoi} &= v_i \cos(\theta_i) \\ \dot{p}_{yoi} &= v_i \sin(\theta_i) \\ \dot{\theta}_i &= \omega_i \end{aligned} \tag{3.7}$$

Where v_i and ω_i are the control inputs to the robot. To deal with the non-holonomic constraint, the position of the robot's heading point is controlled instead of the position of the robot's center. The kinetics of the heading point relative to the robot's center are given in Equation 3.8.

$$\begin{bmatrix} \dot{p}_{xi} \\ \dot{p}_{yi} \end{bmatrix} = \begin{bmatrix} v_i \cos(\theta_i) - l\omega_i \sin(\theta_i) \\ v_i \sin(\theta_i) + l\omega_i \cos(\theta_i) \end{bmatrix} \triangleq \begin{bmatrix} u_{xi} \\ u_{yi} \end{bmatrix} \tag{3.8}$$

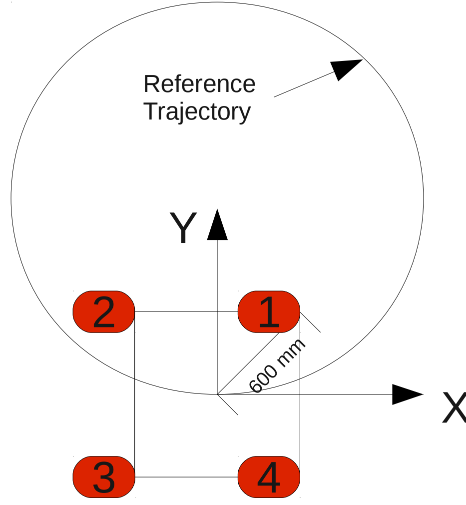

Figure 3.2: *Snapshot of formation shape along with reference trajectory*

Where l is the distance between the center and the heading point. $u_{xi}, u_{yi} \in \mathbb{R}$ are auxiliary control inputs that can be designed to achieve the trajectory tracking goal. Once these control inputs are determined, the actual robot control inputs are back calculated as follows.

$$\begin{bmatrix} v_i \\ \omega_i \end{bmatrix} = \begin{bmatrix} \cos(\theta_i) & \sin(\theta_i) \\ -\frac{1}{l}\sin(\theta_i) & \frac{1}{l}\cos(\theta_i) \end{bmatrix} \begin{bmatrix} u_{xi} \\ u_{yi} \end{bmatrix} \tag{3.9}$$

These auxiliary control inputs are designed to track each robot's desired trajectory. A proportional-integral-derivative or PID controller with a velocity feed forward term is used for this. This controller is given in Equation 3.10.

$$\begin{aligned} u_{xi} &= \dot{p}_{xdi} - k_{Px}\left(p_{xi} - p_{xdi}\right) - k_{Dx}\left(\dot{p}_{xi} - \dot{p}_{xdi}\right) - k_{Ix}\int\left(p_{xi} - p_{xdi}\right)d\tau \\ u_{yi} &= \dot{p}_{ydi} - k_{Py}\left(p_{yi} - p_{ydi}\right) - k_{Dy}\left(\dot{p}_{yi} - \dot{p}_{ydi}\right) - k_{Iy}\int\left(p_{yi} - p_{ydi}\right)d\tau \end{aligned} \tag{3.10}$$

Where $k_{Px}, k_{Py}, k_{Dx}, k_{Dy}, k_{Ix}, k_{Iy}$ are constant control gains.

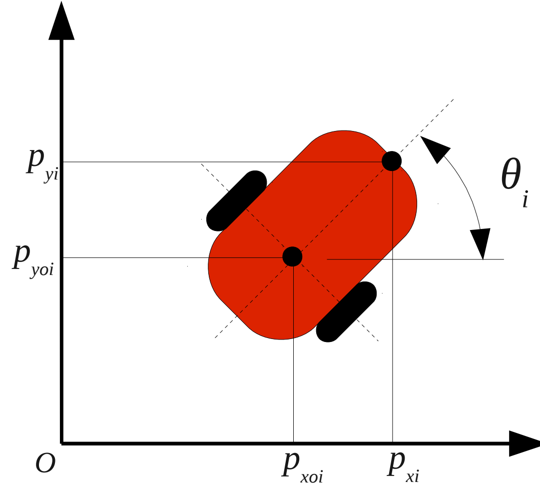

Figure 3.3: *Wheeled mobile robot schematic showing the heading point*

3.3 Experiment

Once the control algorithm was developed, an experiment was developed to demonstrate its effectiveness. What follows are the details and results for this experiment.

3.3.1 Hardware Implementation

The robots used for this experiment were AmigoBots, which were introduced in Section 2.1. The robots made use of their built in linear encoders to keep track of the their global position, and for velocity control feedback. The control algorithm was implemented in software on a remote workstation, and velocity commands were sent to the robots over an Ad-Hoc wireless IEEE 802.11b network. All four robots were controlled by the same computer as shown in Figure 3.4. The control computer was a simple desktop workstation running the Microsoft Windows XP operating system. It communicated with the network using a USB wireless Ethernet adapter.

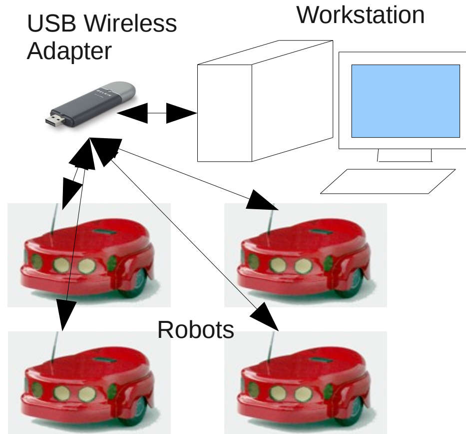

Figure 3.4: *Experimental Hardware Configuration*

3.3.2 Software Implementation

The control software for these experiments was written in C++ using the ARIA robot API. ARIA was previously discussed in Section 2.3. The control loops for each robot were implemented in separate threads. This allowed the control calculations to be done in parallel. A central data storage thread modeled the network interaction and calculated the reference trajectory. The main thread of the program was responsible for initializing communication with robots, and spawning all the other threads. Figure 3.5 shows a graphical representation of the program's structure, and how these different threads interact. The robot control threads were responsible for the communication with the robots. These threads would wait for a communication packet from the robot, perform the necessary calculations, and respond with a command packet. This resulted in the time step for the control calculations being dependent on the robot communication.

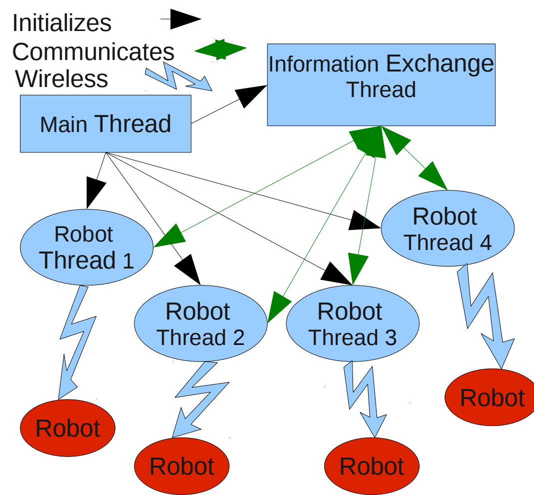

Figure 3.5: *Control Software Structure*

3.3.3 Implementation Challenges

There were a great many challenges to be overcome in the implementation of this experiment. Most of them were related to writing the control software, and overcoming the practical limitations imposed by real systems.

In order to simplify the process of fixing issues with the software, the code was tested in stages. First, the trajectory tracking code was tested and initial PID gains were set before the consensus algorithm was implemented. Initially, all the robots were placed in their desired formation shape. This resulted in small or zero initial error in the robots' positions, removing a small aspect of the difficulty of the experiment. When the consensus algorithm was implemented it initially did not have any disturbances, and was given initial conditions identical to the desired signal's initial conditions. This allowed for initial setting of the consensus gains. After the experiment worked at each stage, another level of difficulty (such as disturbances) was added, and any new issues addressed. Also, at each stage, the MobileSim robot simulation program that came with the robots was used to test for problems

before the experiment was run with real robots.

Some issues were easy to fix, but difficult to correctly identify. During the tuning of the motion controller, the robots were given a specific circular desired trajectory without any consensus. When the experiment was run the robots would drive in a circle, but of the wrong radius. Initially, it was believed that poor tuning or some issue with the communication was at fault. Later, it was discovered that the robot was expecting an angular velocity command in degrees per second, while the control algorithm was calculating it in radians per second. A simple unit conversion solved the problem that was very hard to identify.

The largest issue with the experiment involved noisy output from the consensus controller. The trajectory tracking controller calculates the desired trajectory for the robot from the the consensus state and its derivative. The derivative of the consensus state for the experiment is calculated as a combination of the consensus control input and the disturbances. It was found that the consensus trajectory and particularly the derivative were very noisy. This caused the trajectory tracking controller to fail. The experiment would appear to be working at the start, but eventually one of the robots would turn in the wrong direction literally spiraling out of control, and not be able to recover. Sometimes they would recover, but seldom would the experiment run for long.

Several steps were taken to eliminate the problem of the noisy consensus states. The main issue was that part of the control law involved the signum function of the error. This discontinuous function was responsible for a large portion of the noise in the consensus states. First, the consensus control gains were changed to try and reduce that function's influence on the noise. This had limited benefit because changing the gains lowered the tracking performance. Next, a low pass filter was applied to the computed trajectory before being used in the tracking controller. This helped reduce some of the noise, but not all. More aggressive filtering could have removed more of the noise, but would have began to affect the underlying signal leading to steady-state errors in the tracking. Slowing down the reference trajectory also helped to reduce the chance of the experiment failing, and it

increased the odds of the robots recovering from a fault.

Despite all of these attempts to improve performance, the experiment continued to fail. Finally, it was decided to replace the discontinuous signum function in the control law with a continuous approximation. This did not remove all the noise, but it reduced the noise enough that the previous improvements were enough to solve the problem. The hyperbolic tangent function, like the signum function, returns 1 for large positive numbers and returns -1 for large negative numbers, but at the origin it has a smooth transition from -1 to 1 instead of a sharp step. In the implementation, before the error was passed to the hyperbolic tangent function, it was multiplied by a positive constant. Increasing this constant made the hyperbolic tangent function approximate the signum function better, of course it also increased the noise on the consensus state. This allowed the control law to be adjusted to reduce the noise while still maintaining performance. Once the consensus control gains were re-adjusted this allowed the experiment to proceed.

3.3.4 Experimental Parameters

The reference trajectory was selected as a circle where the formation rotates to stay aligned with a tangent to the circle. A circle was selected to ensure the robots would be able to execute their trajectories without any problems. Also, it allowed the experiment to run for a reasonable amount of time without facing the space constraints of the test environment. Equation 3.11 gives the derivatives of the reference states that were integrated to generate the reference trajectory.

$$\begin{aligned} \dot{p}_{xc}^d &= v_c^d \cos(\theta_c^d) \\ \dot{p}_{yc}^d &= v_c^d \sin(\theta_c^d) \\ \dot{\theta}_c^d &= \omega_c^d \end{aligned} \tag{3.11}$$

The reference velocities for the experiment were $v_c^d = 20[mm/sec]$ and $\omega_c^d = \frac{\pi}{150}[rad/sec]$.

The initial conditions used for the integration of the reference states were:

$$\begin{bmatrix} p_{xc}^d(0) \\ p_{yc}^d(0) \\ \theta_c^d(0) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.12}$$

As was shown in Figure 3.2 the formation shape for this experiment is a square centered on the reference trajectory with a robot at each corner. The formation shape vectors for robot i was calculated as follows.

$$\begin{aligned} d_{xi} &= 600 \cos\left(\frac{\pi}{2} i - \frac{\pi}{4}\right) [mm], \quad i = 1, \ldots, 4 \\ d_{yi} &= 600 \sin\left(\frac{\pi}{2} i - \frac{\pi}{4}\right) [mm] \end{aligned} \tag{3.13}$$

The information exchange graph details are shown graphically in Figure 3.6. For the experiment only the first agent has access to the reference signals, and the information exchange between the agents is undirected. Mathematically the information exchange topology

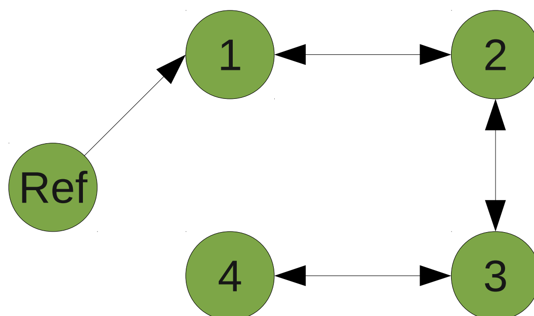

Figure 3.6: *Undirected information exchange graph showing connections between agents*

Gain	Value
k_1	0.01
k_2	0.7
k_c	11
$k_{Px} = k_{Py}$	0.1
$k_{Dx} = k_{Py}$	0.001
$k_{Ix} = k_{Iy}$	0.01

Table 3.1: *Experimental Control Gains*

$x_1(0)$	$x_2(0)$	$x_3(0)$	$x_4(0)$
-1000 [mm]	500 [mm]	1000 [mm]	-500 [mm]
-500 [mm]	1500 [mm]	500 [mm]	1000 [mm]
1 [rad]	0.75 [rad]	0.5 [rad]	0.25 [rad]

Table 3.2: *Consensus Initial Conditions*

is given the the following A and B matrices.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.14}$$

To ensure that each agent had different disturbances on the information exchange dynamics, the applied disturbances were a function of the agent number. The dynamics along with the disturbances are given in Equation 3.15 below.

$$\dot{x}_i = u_i + 0.1 \sin\left(\frac{i}{7}t\right) + 0.1\cos(x_i), \quad i = 1, \ldots, 4 \tag{3.15}$$

Values for the control gains from Equations 3.3, 3.4, and 3.10 are all listed in Table 3.1.

To better illustrate the tracking capabilities of the consensus tracking controller, each robot was given non-zero initial consensus state. The initial conditions used are given in Table 3.2.

The robots were not initially placed in the desired formation. Instead, they were arranged in a line. This helped illustrate the capabilities of the trajectory tracking motion controller. Additionally, it helped avoid collisions between the robots during initial transients. The initial conditions for position are given in Table 3.3.

Robot	$p_{xoi}(0)$ [mm]	$p_{yoi}(0)$ [mm]	θ_i [rad]
1	-150	750	0
2	-150	250	0
3	-150	-250	0
4	-150	-750	0

Table 3.3: *Position Initial Conditions*

3.3.5 Experimental Results

After the control software was written, the control gains were tuned, and all of the issues previously mentioned were solved, the experiment was run to get the results. After the experiment was run for an appropriate amount of time, the data collected was saved and analyzed. Then graphs were generated to illustrate the performance of the algorithm.

First, the consensus tracking results were analyzed. Figure 3.7 shows the consensus tracking results for the coordinates of the center of the formation. Each robot's understanding of the state is graphed along with the desired reference signal. Despite the initial conditions, the consensus states quickly converge to the desired value. Because of the disturbances and noise, the tracking is not perfect; however, it is very close, and the error is small relative to the signal.

Figure 3.8 shows the consensus tracking results for the desired angle for the formation. Each robot's understanding is graphed along with the desired reference signal. Because the desired signal in this case is a ramp, the difference between the robot's states and the reference state is also graphed. Again the states quickly converge to the desired value, and remain close despite the disturbances.

The trajectories of the robots' heading points are graphed in Figure 3.9. Also shown is the shape of the formation at 50 seconds and 200 seconds into the experiment. This graph clearly shows that the initial positions were not in the desired formation. It takes the agents several seconds to begin tracking their respective trajectories, but after 50 seconds the formation is almost the desired shape. After 200 seconds, the formation has already

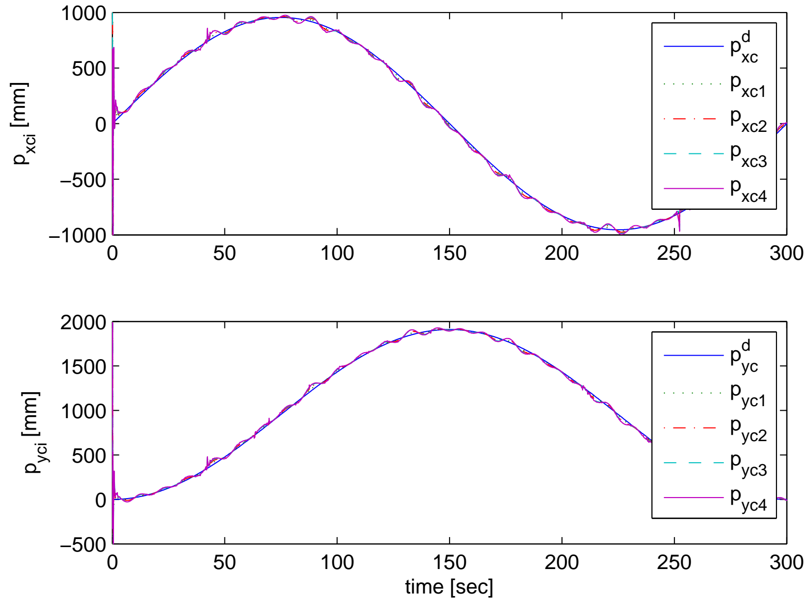

Figure 3.7: *Consensus tracking results for states* $p_{xci}(t), p_{yci}(t), i = 1, \ldots, 4$ *and reference states* p^d_{xc}, p^d_{yc}

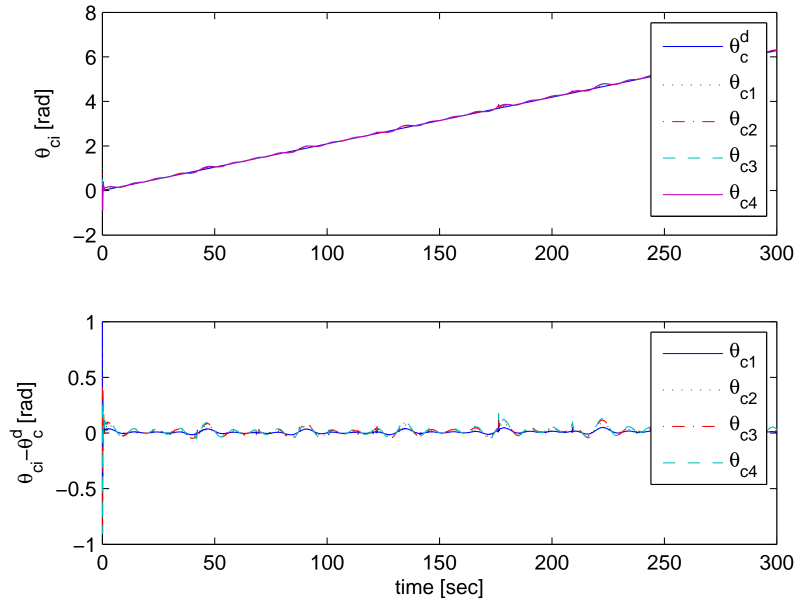

Figure 3.8: *Consensus tracking results and error for state* $\theta_{ci}, i = 1, \ldots, 4$ *and reference state* θ^d_c

achieved the desired shape, and is clearly tracking its desired path.

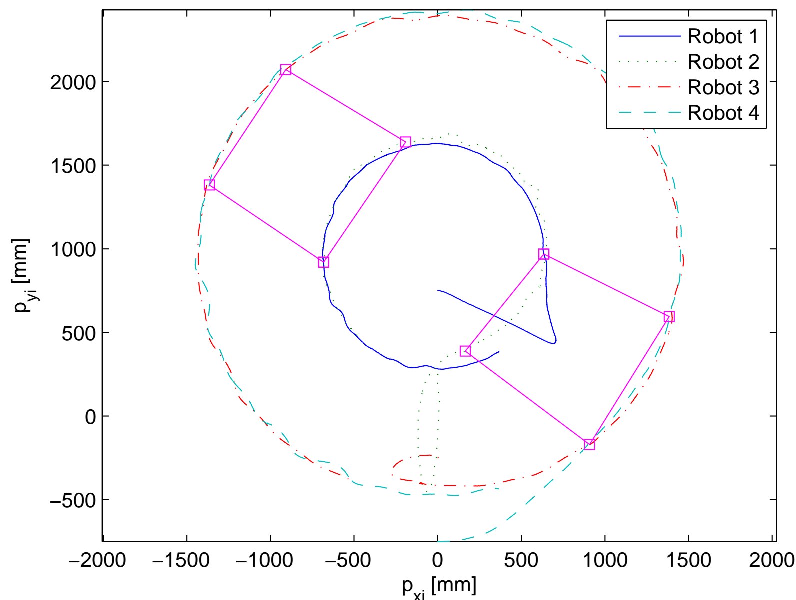

Figure 3.9: *Actual trajectory of the robots with formation snapshots at 50 seconds and 200 seconds*

Shown in Figure 3.10 is the position tracking error for one of the robots. This error converges to zero.

3.4 Distributed Experiment

As was mentioned earlier, there were several aspects of the experiment, which despite the distributed nature of the algorithm, made it centralized. To address these issues the experiment was re-designed. What follows are the details and results for this reworked experiment.

3.4.1 Hardware Implementation

The use of one computer for control is one of the main centralized aspects of the experiment, so for the distributed experiment, one computer was used for each agent (4 to be exact). Figure 3.11 shows the hardware setup for the distributed version of the experiment. The

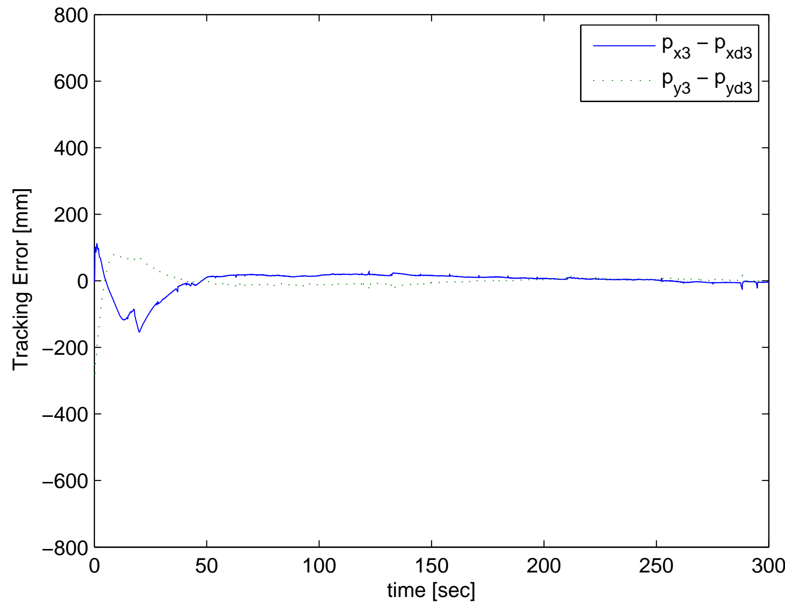

Figure 3.10: *Formation tracking error for Robot 3*

computers used were four laptops running a Linux operating system. The same Ad-Hoc network was used for communicating with the robots, but instead of external adapters, the laptops were all equipped with built in wireless networking cards. The same network was also used for the inter-agent communication required by the consensus algorithm.

3.4.2 Software Implementation

Most of the changes between the original and distributed versions of the experiment were in software implementation. Instead of one control program, a different instance of the control software was run for each agent. All these control programs were running on different computers, and using the wireless network for communication. This allowed the experiment to be decentralized. Figure 3.12 shows the new structure of the control software.

What was previously a central data storage thread became the consensus control thread. The consensus tracking controller code was separated from the motion controller code and implemented in this new thread. This allowed greater control over the execution rate of

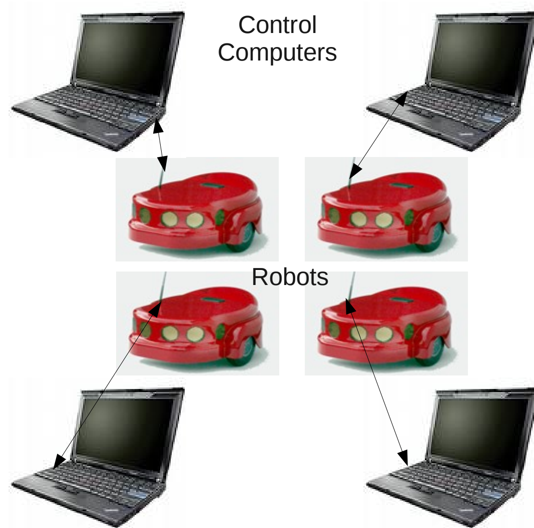

Figure 3.11: *Distributed Experimental Hardware Configuration*

the consensus algorithm that improved its stability. Instead of executing at the robot communication rate which is about 10 Hz the control loop could execute at a much faster rate of approximately 100 Hz. The consensus thread also handles initializing and monitoring network connections to the other agents in the formation. Only the agents that are adjacent have communication links established. An extra initialization phase was added to the program where it waits for all necessary connections to be made between the agents before connecting to the robots, or starting the control algorithms. The source code for this experiment is listed in Appendix A.

3.4.3 Implementation Challenges

There were several challenges in implementing the distributed version. The initial failure of the consensus algorithm proved to be a major obstacle. Even for a simple case, the consensus information quickly grew too large, due to delays in the information, and numerical issues arising from large step sizes.

Originally all the control calculations for both the consensus and motion control were

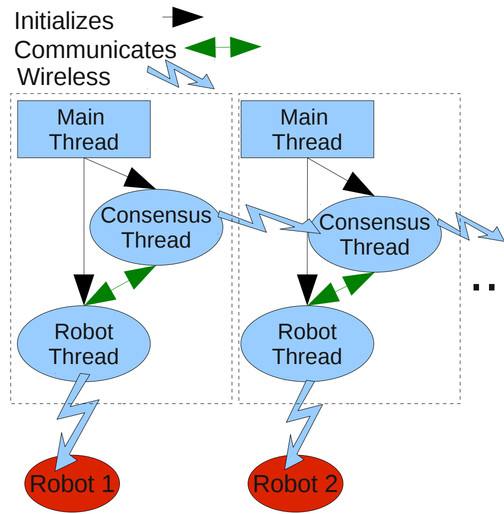

Figure 3.12: *Distributed Control Software Structure*

done in the robot control loop in a custom "action". Periodically the robot sends a packet containing status and sensor information to the control computer. This triggers an iteration of the robot control loop. During each iteration several tasks are executed including the custom action containing the control code. Then a response packet containing the new control inputs is sent by the control computer to the robot. This process happens approximately every 100 ms. This relatively large step size coupled with a delay in receiving information from the other agents was causing the consensus calculations to diverge quickly.

The sample rate for the packets sent by the robots could have been increased, but this would have added additional network load. Instead, the consensus calculations were run in a separate thread in parallel to the robot control thread. They were moved to the data storage thread, along with the code to handle the exchange of consensus information between the agents over the network. Because this thread did not depend on receiving a packet from the robot to trigger its execution, it could run much faster with smaller time steps solving the stability issues.

3.4.4 Experimental Parameters

Many of the parameters from the original experiment remained unchanged for the distributed version. One of the major differences was in the speed of the reference trajectory. Decreasing the time step for the consensus calculations improved the performance of both the consensus controller and the motion controller. This allowed for the motion controller to track a faster trajectory. The new reference trajectory parameters were $w_c^d = \frac{\pi}{50}[rad/sec]$ and $v_c^d = 60[mm/sec]$.

3.4.5 Experimental Results

As in Section 3.3.5, after the control program was complete, the experiment was run and data collected. What follows are graphs showing that the performance is maintained despite the changes. In fact, the performance is actually improved somewhat.

Figure 3.13 shows the first two consensus states and their reference states. The figures in this section have the same time scale as the figures for the centralized experiment. One of the changes between the previous experiment and this one was increasing the speed of the formation by increasing the reference state parameters. Despite the increase in speed, and the disturbances, the consensus algorithm is still able to track the reference states quite well.

The third consensus state and its corresponding reference state are graphed in Figure 3.14. Just as in the previous figure, the tracking is good. Unlike Figure 3.8 the error was not shown. Because of the distributed nature of the experiment, the reference state has a different time scale than the consensus states. This made it difficult to compute the error accurately.

The actual paths of the four robots during the experiment are shown in Figure 3.15. As before, the robots begin the experiment without the desired configuration. After a few seconds, the robots move into the desired formation shape, and the formation tracks the desired trajectory. The exact position tracking is not quite as good as it was previously.

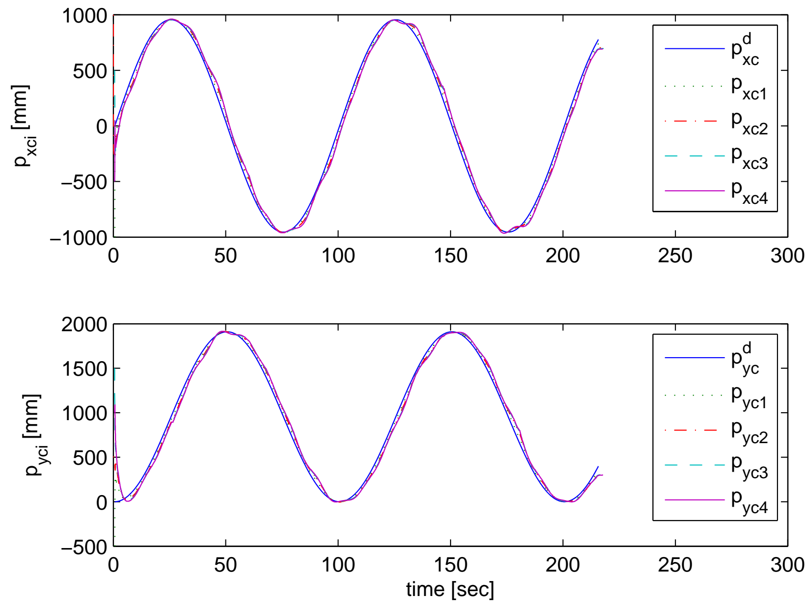

Figure 3.13: *Consensus tracking results for states* $p_{xci}(t), p_{yci}(t), i = 1, \ldots, 4$ *and reference states* p^d_{xc}, p^d_{yc}

This is likely due to the greatly increased speed of the formation. The increase in speed was to demonstrate that the improved consensus performance allowed the tracking of faster trajectories.

The position tracking error for the third robot is shown in Figure 3.16. Again, the tracking is not perfect but is close. It takes several seconds to stabilize to the desired values. This is partially because of the increased speed of the formation.

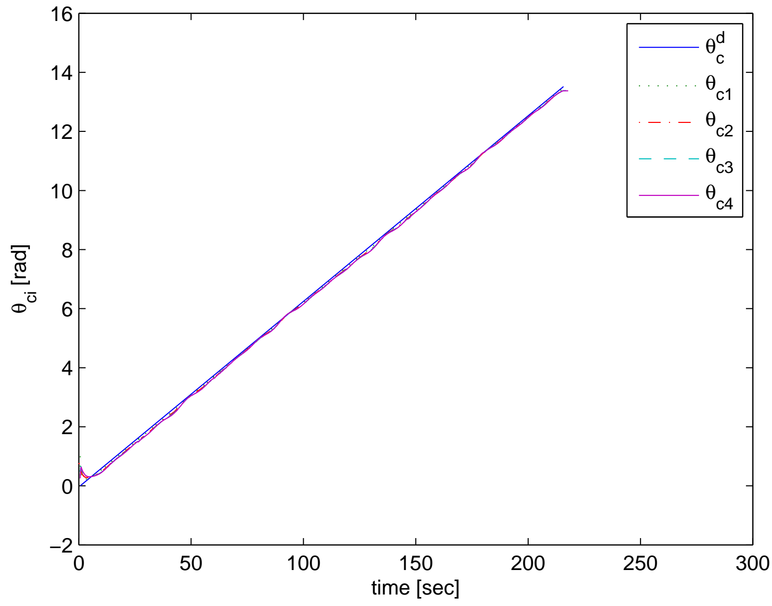

Figure 3.14: *Consensus tracking results and error for state $\theta_{ci}, i = 1, \ldots, 4$ and reference state θ_c^d*

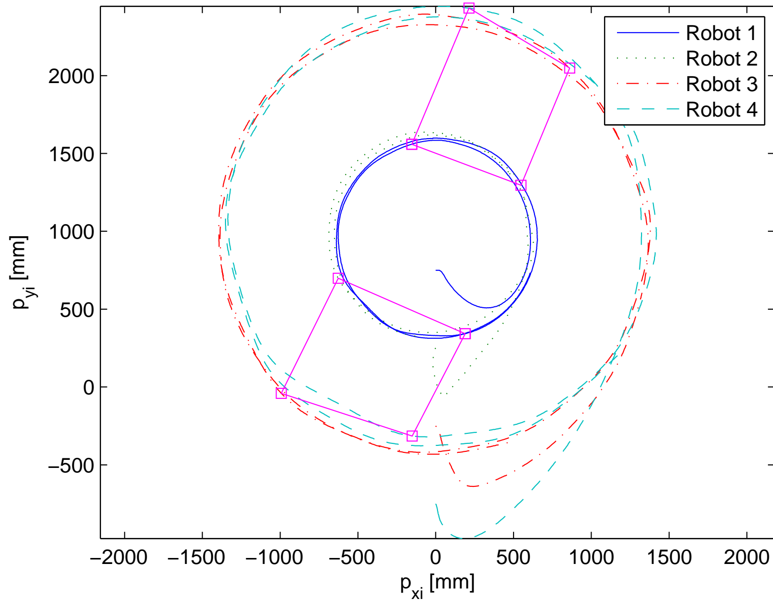

Figure 3.15: *Actual trajectory of the robots with formation snapshots at 50 seconds and 200 seconds*

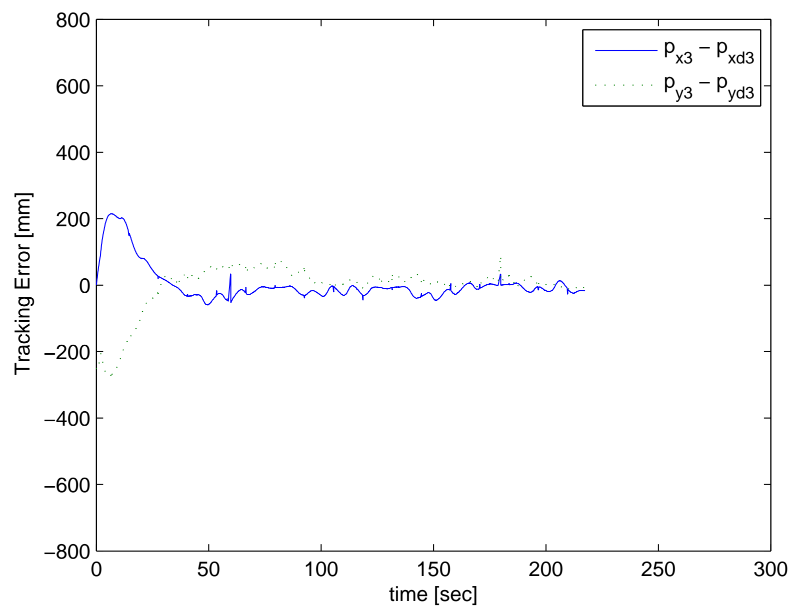

Figure 3.16: *Formation tracking error for Robot 3*

Chapter 4

Formation State Description and Measurement

4.1 Motivation

In the previous chapter, the positions of the agents in the formation were described in terms of two pieces of information. The global position and orientation of the formation's center, and the relative locations of the agents from this center. This approach requires each agent to know its own position relative to some common, global, reference frame. The inclusion of a global reference frame presents several challenges. First, a single reference for all the agents leads to a centralized framework. Second, each agent must measure its position accurately. In all the previous experiments (Sections 3.3 and 3.4), the position and orientation of the robots were measured by wheel encoder odometery. This process leads to slight errors accumulating over time, and wheel slip can cause significant drift in the measurements. Also, the initial positions of the robots must be known. Another traditional solution to this problem is to use Global Positioning System or GPS measurements. Unfortunately, GPS measurements may not always be available, such as in indoor environments.

Removing the dependence on a global reference frame gives a more decentralized approach. It also removes the problems associated with needing the global position and orientation of each agent. New problems arise with the removal of the global reference. In particular, it requires new ways of describing and measuring the agent's positions in the

formation. Instead of global positions, only relative measurements between two agents, or between an agent and its environment are possible. This leads toward describing the state of the formation in terms of the relative positions of the agents, or in terms of the geometry of the formation. Using these tools it is possible to build up a state that completely describes the configuration of the agents in a formation. Sections 4.2 and 4.3 addresses this problem.

One of the ways to make relative measurements within a formation of robots is to use cameras mounted to the robots. Cameras and computer vision algorithms are capable of making relative measurements with some constraints. Field of view constraints for most cameras, and other practical considerations, limit their measurement capabilities. Related to the formation control problem, it is unlikely that every agent will be able to make all the measurements necessary to completely describe the formation. In the same way, it is unlikely that any one agent will be able to obtain this information by itself. The assumption that may be made more safely, is that if all of the information being measured was combined somehow, a complete description of the formation's state could be developed. Section 5.2 describes a way to combine measurements made by different agents into a complete description of the formation. This is done in a distributed manner using a consensus tracking algorithm based on the one used in section 3.2.

4.2 Formation State Description

To begin with, only a formation consisting of three mobile robots will be considered. The shape of such a formation is obviously a triangle. The goal is to describe this formation without reference to a global frame. To describe the shape of three points in a plane uniquely, a minimum of four variables, and a local coordinate frame are needed. One of these four should relate to the coordinate frame. Then three more pieces of information that relate the shape to a global frame are needed to completely describe the shape in the global reference frame. Again these three must be independent.

Considering the fact that the robots have size and therefore orientation makes this a

formation of robots. This can be accounted for by considering a coordinate frame aligned with each robot and fixed to each point. Because one coordinate frame already exists, only two have to be added. This corresponds to adding two more pieces of information to fully describe the formation. These two pieces of information don't have to take the form of coordinate frames, but they do have to relate the orientation of two different robots to the formation. This brings the number of independent variables up to 6, three to describe the shape of the formation, and three to relate the orientations of the robots to the formation. It is not necessary to think of any of these latter three in terms of coordinate frames at all, but it may prove to be convenient. These must also be independent.

For example, a formation of three robots can be described with the following independent set of 6 variables: two of the distances between the robot centers; two of the bearing angles; and two of the relative rotations between robots. Specifically, d_{ij} the distance from the center of (origin of) robot i to the center of (origin of) robot j. There are three of these distances because $d_{ij} = d_{ji}$ and $d_{ii} = 0$. The bearing angle β_{ij} is the bearing angle from robot i to robot j, it is the angle measured positive counter clockwise, from the coordinate frame of robot i to the line connecting the center of robot i to the center of robot j. β_{ij} is defined on $(-\pi, \pi]$ and is measured from the axis pointing out of the front of the robot. β_{ii} is undefined, and $\beta_{ij} \neq \beta_{ji}$. θ_{ij} is the relative angle of rotation from robot i to robot j. It is the angle measured positive counter clockwise between one local axis of robot i and the same local axis of robot j. It can also be defined as $\theta_{ij} = \theta_j - \theta_i$ where θ_i and θ_j are the orientation angles of the two robots measured in some global frame. Figure 4.1 shows a formation of three robots described with these variables.

Relating this example back to the earlier discussion, these variables can be thought of as describing a triangle with local coordinate frames at each vertex. The distances correspond to the lengths of two of the sides of the triangle. The bearing angles give the relative orientation of two of the coordinate frames to adjacent sides of the triangle. Then, each relative rotation angle tells the angle between two coordinate frames. With these distances

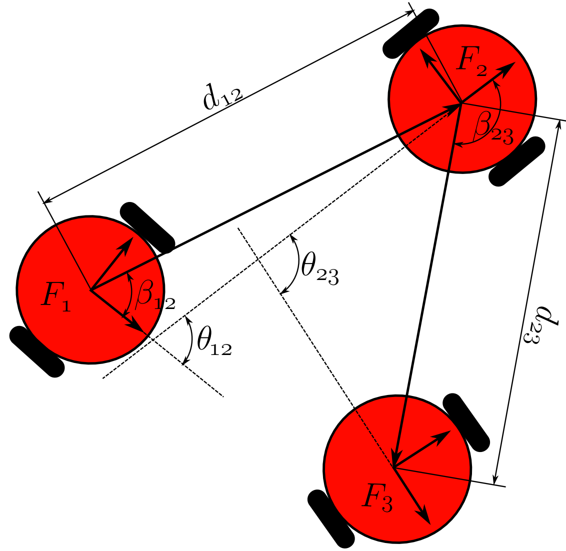

Figure 4.1: *Fully Described Robot Formation*

and angles, it is possible to calculate any other variable that could describe the formation. For example using the bearing angles, and the relative rotations, the internal angle where the two known sides of the triangle meed could be determined. Then, using this angle and the known lengths, the length of the third side could be determined using the law of cosines. It is clear that these 6 variables serve as a basis set for all variables that can describe the formation. Any descriptor of the formation can be calculated from these using geometric relationships. This basis set is not unique, any complete set of independent variables describing the geometry of the formation will do.

This concept can be generalized to larger formations with more robots. For each additional robot in the formation, the number of states required to completely describe the formation increases by the number of degrees of freedom for the robot. For example to completely describe a formation of four robots with each having three degrees of freedom, 9 variables are required. As previously mentioned, a similar formation with only three robots would require 6 variables. In general the positions of robots in a formation irrespective of

a global frame can be described by the number of degrees of freedom for the robots times the number of robots minus the number of degrees of freedom for the global frame. For the formation of four robots this equates to 3 degrees of freedom times 4 robots minus 3 degrees of freedom for the global frame, or 9 variables total.

4.3 Measurement Rules and Assumptions

Three states exactly define the relative displacement between two robots with three degrees of freedom on a plane. That is one robot can be uniquely placed relative to another using only a three-dimensional state. Therefore, a robot who has only one robot visible to it, can make at most 3 independent measurements. Also, it means that if a robot has two robots visible, it can make 6 independent measurements. An example of these 3 independent measurements would be distance d_{ij} and the two bearing angles β_{ij} and β_{ji}. Or, the distance d_{ij} one bearing angle β_{ij} and the relative rotation θ_{ij}. Note that if robot i makes three independent measurements of robot j, no measurements of robot i made by robot j will be independent of the original three.

Considering the three independent measurements of one robot from another as a basis set for all possible measurements allows some freedom. For example, given three of these measurements from robot i to robot j, they can easily be converted into the equivalent measurements from robot j to robot i.

Assuming three measurements are possible, the following measurement rules were developed:

- Robot j is visible to robot i if robot j is in the field of view of robot i's camera. This is true if $|\beta_{ij}| < \sigma_i/2$, where σ_i is the view angle of robot i's camera.
- If a third robot is blocking robot i's view of robot j, robot j is not visible.
- If a robot is visible, it can be uniquely identified.

- If robot j is visible to robot i, robot i can accurately measure three independent states describing their relative locations, such as β_{ij}, d_{ij}, and θ_{ij}.

With these rules and assuming a three robot formation there are 36 possible measurements, and for each robot that is visible, three of these are measured. However, remembering that the independent measurements form a basis set; relations can be developed such as the following: $d_{ij} = d_{ji}$, $\theta_{ij} = -\theta_{ji}$, $\beta_{ij} = \theta_{ij} + \pi + \beta_{ji}$. Using relations such as these, for each robot that is visible, six of the possible measurements can be calculated. If only three of these measurements are used, that is, d_{12} is considered a measurement, and not d_{21}. This reduces the total number of measurements from 36 to 9 without causing too many problems because when a robot is visible 6 variables are calculated then the 3 that are not desired measurements are discarded.

So, for the case of a three robot formation there are 9 possible measurements. Any 6 of these are independent, and with any 6 we can completely define the shape of the formation as discussed in Section 4.2. These measurements are made by the robots three at a time.

For the experiments, a particular measurement configuration for the formation will be assumed. This ensures that all of the states are measurable. It will be assumed that robot 2 is always visible to robot 1, and robot 3 is always visible to robot 2. Using the measurement rules defined above, this allows the selection of a state made up of variables directly measurable. Because of the chosen formation case, the distance, bearing angle, and relative rotation between robots 1 and 2, and robots 2 and 3 will be the states used.

These definitions give the following state (Equation 4.1) that describes the shape of the formation and the orientation of the robots irrespective of a global reference frame.

$$x = \begin{bmatrix} d_{12} \\ \beta_{12} \\ \theta_{12} \\ d_{23} \\ \beta_{23} \\ \theta_{23} \end{bmatrix} \tag{4.1}$$

4.4 Image Measurement

Homography techniques allow the comparison of two images taken of the same plane and can determine the translation and rotation required to transform one image into the other. This gives the translation and rotation of the camera's coordinate frame as it moves from the reference configuration where the first image was taken to the configuration where the second image was taken. Or the camera remains still and the plane translates and rotates. In the case of two robots the motion will likely be a combination of these two. One image can be taken to represent a reference configuration where the relative positions of the robots are known, and a second image to represent a configuration where the relative positions of the robots are unknown. Then using homography, and certain information known about the reference configuration, the relative positions and orientations of the two robots can be determined. This process is simplified slightly by the fact that the robots are constrained to move in a plane, essentially removing two rotational and one translational degrees of freedom.

The two papers [24, 25] discuss similar development for image based measurements using homography.

Consider two wheeled mobile robots Robot 1 and Robot 2 that are moving in a plane. Robot 1 has a camera rigidly attached to it pointing towards the front of the robot. Robot 2 has a plane π containing feature points rigidly attached to it facing rearward. Figure 4.2 shows the robots, several coordinate frames, and vectors in two configurations. The first configuration is a static reference configuration. The coordinate frames for Robot 1 and Robot 2 are F_{1r} and F_{2r} respectively. There is also a camera coordinate frame I_r. After undergoing some translations and rotations, the robots arrive at the second configuration where their coordinate frames are F_1 and F_2. The camera coordinate frame in this configuration is I.

The rotations and translations between various coordinate frames shown in Figure 4.2 are summarized in Table 4.1. $R_r, R'_r, R, R', R_I \in SO(3)$ represent rotation matrices between the various coordinate frames. $x_{fr}, x'_{fr}, x_f, x'_f, x_I \in \mathbb{R}^3$ are the relative positions of the

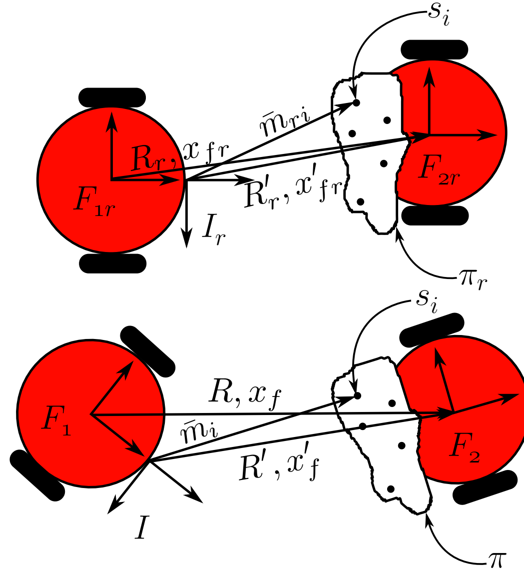

Figure 4.2: *Image Measurement Coordinate Frames*

various coordinate frames. The subscript r indicates a variable or coordinate frame in the reference configuration. Also the use of $'$ indicates rotations or translations relative to the camera coordinate frame.

$$x_{fr} = \begin{bmatrix} x_r & y_r & z_r \end{bmatrix} \tag{4.2}$$

$$x_f(t) = \begin{bmatrix} x(t) & y(t) & z(t) \end{bmatrix} \tag{4.3}$$

R_I and x_I are not pictured in the figure, but they relate the location and orientation of the camera to the robot to which it is attached. The constant Euclidean coordinates of the ith feature point on the plane π described in Robot 2's coordinate frame are denoted s_i. $\bar{m}_{ri} \in \mathbb{R}^3$ is the Euclidean coordinates of the ith feature point on the plane π_r described in I_r. Also, $\bar{m}_i \in \mathbb{R}^3$ is the Euclidean coordinates of the ith feature point on the plane π described in I. That is they are the coordinates of the feature points in the camera coordinate frame for the reference and second configurations respectively.

Rotation	Translation	Coordinate Frames
R_r	x_{fr}	F_{2r} to F_{1r} in F_{1r}
R'_r	x'_{fr}	F_{2r} to I_r in I_r
R	x_f	F_2 to F_1 in F_1
R'	x'_f	F_2 to I in I
R_I	x_I	I to F_1 in F_1

Table 4.1: *Relationships Between Coordinate Frames*

$$\bar{m}_{ri} = \begin{bmatrix} x_{ri} & y_{ri} & z_{ri} \end{bmatrix} \tag{4.4}$$

$$\bar{m}_i(t) = \begin{bmatrix} x_i(t) & y_i(t) & z_i(t) \end{bmatrix} \tag{4.5}$$

The camera follows the pinhole camera model represented by the Equation 4.6.

$$p_i = Am_i \tag{4.6}$$

Where p_i represents the image space pixel coordinates of some point, A is the intrinsic camera matrix, and m_i is the normalized version of $\bar{m}_i$. The normalized coordinates are given as follows.

$$m_{ri} = \begin{bmatrix} \frac{x_{ri}}{z_{ri}} & \frac{y_{ri}}{z_{ri}} & 1 \end{bmatrix} \tag{4.7}$$

$$m_i(t) = \begin{bmatrix} \frac{x_i(t)}{z_i(t)} & \frac{y_i(t)}{z_i(t)} & 1 \end{bmatrix} \tag{4.8}$$

Through camera calibration, the intrinsic camera parameters represented by the matrix A are known. Also through calibration, the relative position and orientation of the camera coordinate frame to the first robot's frame x_I and R_I respectively are known.

From the geometry of the problem shown in Figure 4.2, the following relationships can be developed.

$$x_{fr} = x_I + R_I x'_{fr} \qquad R_r = R_I R'_r \tag{4.9}$$

$$x_f = x_I + R_I x'_f \qquad R = R_I R' \tag{4.10}$$

The relationships in Equation 4.9 lead to expressions to calculate the translations and rotations from the camera coordinate frame. Given the extrinsic camera calibration parameters, and the exact relative locations of the robots in the reference configuration, these parameters can be calculated as follows.

$$
\begin{aligned}
x_{fr} &= x_I + R_I x'_{fr} \\
R_I x'_{fr} &= x_{fr} - x_I \\
x'_{fr} &= R_I^T \left(x_{fr} - x_I \right)
\end{aligned}
\tag{4.11}
$$

$$
\begin{aligned}
R_r &= R_I R'_r \\
R'_r &= R_I^T R_r
\end{aligned}
\tag{4.12}
$$

The feature point locations relative to the camera can be described as follows.

$$\bar{m}_{ri} = x'_{fr} + R'_r s_i \tag{4.13}$$

$$\bar{m}_i = x'_f + R' s_i \tag{4.14}$$

These can be combined to eliminate the constant coordinates s_i.

$$
\begin{aligned}
\bar{m}_i &= x'_f + R' R_r'^T \left(\bar{m}_{ri} - x'_{fr} \right) \\
\bar{m}_i &= x'_f - R' R_r'^T x'_{fr} + R' R_r'^T \bar{m}_{ri} \\
\bar{m}_i &= x'_n + R'_n \bar{m}_{ri}
\end{aligned}
\tag{4.15}
$$

Where the auxiliary variables $x'_n \in \mathbb{R}^3$ and $R'_n \in SO(3)$ are defined as follows.

$$x'_n = x'_f - R'_n x'_{fr}, \qquad R'_n = R' R_r'^T \tag{4.16}$$

For further analysis, $d_r \in \mathbb{R}$ and $n_r \in \mathbb{R}^3$ are defined, such that:

$$d_r = n_r^T \bar{m}_{ri} \tag{4.17}$$

Where d_r represents the perpendicular distance from I to the plane π, and n_r is a unit vector normal to π pointing toward I. Substituting Equation 4.17 into Equation 4.15 yields the following.

$$\begin{aligned}
\bar{m}_i =& \frac{x'_n n_r^T \bar{m}_{ri}}{d_r} + R'_n \bar{m}_{ri} \\
\bar{m}_i =& \left(\frac{x'_n}{d_r} n_r^T + R'_n \right) \bar{m}_{ri} \\
m_i z_i =& \left(\frac{x'_n}{d_r} n_r^T + R'_n \right) m_{ri} z_{ri} \\
m_i =& \frac{z_{ri}}{z_i} \left(\frac{x'_n}{d_r} n_r^T + R'_n \right) m_{ri} \\
m_i =& \alpha_i H m_{ri}
\end{aligned} \tag{4.18}$$

Where

$$H = \frac{x'_n}{d_r} n_r^T + R'_n, \qquad \alpha_i = \frac{z_{ri}}{z_i} \tag{4.19}$$

From corresponding feature points in the reference and second image, and equation 4.18 it is possible to calculate H. Then, using a homography decomposition algorithm, it is possible to calculate $\frac{x'_n}{d_r}$ and R'_n if n_r^T is given. Because they are constant, and related to the feature point locations in the reference configuration n_r^T and d_r can be calculated beforehand. Once x'_n and R'_n are obtained, x_f and R can be calculated using equations 4.16, and 4.10.

$$x_f = x_I + R_I \left(x'_n + R'_n x'_{fr} \right), \qquad R = R_I \left(R'_n R'_r \right) \tag{4.20}$$

After calculating these relative positions and orientations, it is necessary to calculate the states used for formation control. Specifically d_{ij}, β_{ij}, and θ_{ij}. These quantities are pictured in Figure 4.3. θ_{ij} can be calculated from R. Because the robots are constrained to a plane, R represents the rotation about robot j's z axis that would align it with robot i. The angle of this rotation is the same as θ_{ij}, so to obtain the angle, the rotation matrix is transformed into a quaternion, or rotation vector representation from which the angle of rotation is extracted. The distance between the robots d_{ij} is simply the magnitude of x_f. The slightly more complicated calculation is determining β_{ij}. It can be found with the

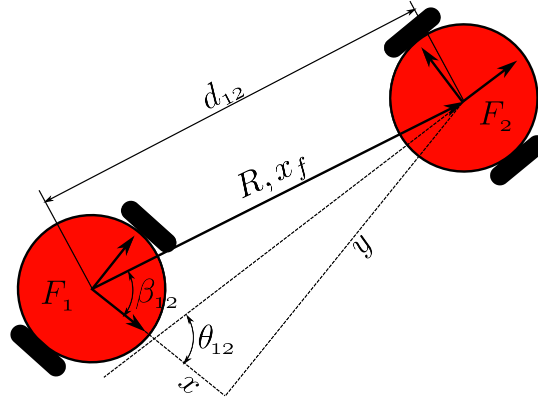

Figure 4.3: *Relationships Between Relative Position Orientation and States*

following equation.

$$\beta_{ij} = \tan^{-1}\left(\frac{y}{x}\right) \tag{4.21}$$

Calculating angles with the inverse tangent function can cause issues if the vector is in the second or third quadrant. In this case however, it is assumed that the angle of view of the camera is much less than 180°; therefore, the x coordinate will always be positive. If it were not, the other robot would not be visible, and no measurement would have been made.

Chapter 5

Vision-Based Formation Tracking Experiment With Consensus

5.1 Problem Description

The goal in this chapter is to develop an algorithm and experiment using the formation description developed in Chapter 4. That state description, and measurement method will be applied to a team of three robots with cameras mounted on them. The robots will use that information to achieve and maintain a desired formation shape.

As mentioned before, there needs to be a method for combining and distributing the information measured by different agents. A consensus algorithm similar to the one used in Section 3.2 can be used to accomplish this task. Previously this algorithm was used to distribute a time-varying reference signal from one leader to the rest of the formation while ensuring that all the agents had the same understanding of this signal. Additionally the algorithm was robust to disturbances in the information exchange, and the agents had limited information exchange. In this new case, the algorithm will be used to combine measured information collected from multiple agents, and distribute it to all the agents ensuring they all have the same information. Again, this will be accomplished despite limited information exchange.

Finally after the robots have an accurate picture of the formation they need a way to design their movements to achieve the desired formation shape. To accomplish this,

first a desired formation state derivative is calculated. Second, this desired derivative is transformed into desired velocities for the robots using an inverse Jacobian matrix.

All of these individual pieces are combined into an experiment implemented on real hardware with real robots and cameras. A general framework for the algorithm used is shown in Figure 5.1. In the experiment, the robots, who do not share any common reference frame, must achieve a desired formation shape based on measurements made with on-board cameras. Additionally, no single agent is able to measure all of the required information, instead the information is measured by several agents. This information is exchanged over a wireless network with limited communication, that is, not all of the agents communicate with each other.

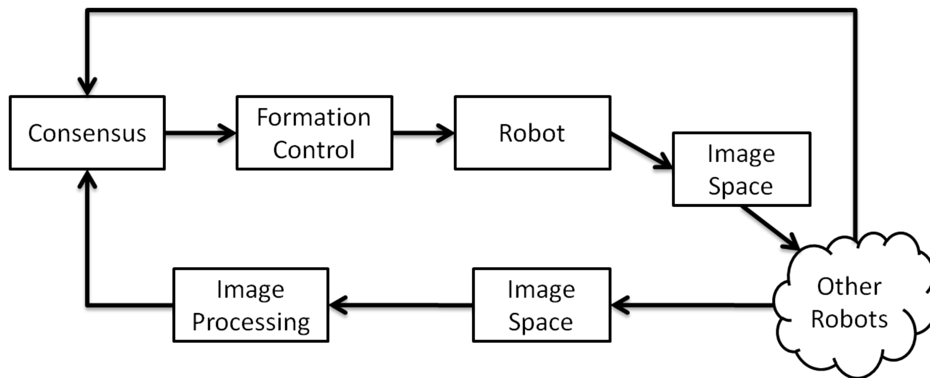

Figure 5.1: *General framework for the experiment*

5.2 Consensus Tracking Protocol

Each robot can only measure part of the relative state information, while the rest of the information is measured by other robots. In order for the robots to make a control decision, they need this information from the other robots in the formation. In order to share this information, a consensus tracking protocol is designed to estimate all the states. This protocol is similar to the one presented in Section 3.2 with a few key differences. The most significant difference is that instead of a single reference state being provided by one or a

subset of the agents, different pieces of a measured state are provided by different agents.

The consensus algorithm is designed to be generic to the number of states and agents. This allows this same algorithm to be applied to other problems with more or less states, and more or less agents. There is little restriction on the states as well, so they could easily represent any information that is measured by several agents in a distributed manner. Because of this, the consensus algorithm could have applications to distributed sensor networks as well.

5.2.1 Model

The goal of this consensus algorithm is to provide a consistent estimate of the formation state to all of the agents. The actual formation state is the current relative positions of the robots in the formation. This state is then measured in a distributed way by several of the agents in the formation. It is the job of the consensus algorithm to update each robot's estimate or understanding of the actual state using these measurements and the understandings of its neighbors. Many consensus algorithms are designed to reach an agreement between several agents on a constant value. This algorithm, however, must track a time-varying state. In order to track this state effectively, the dynamics of the state must be considered.

The current configuration of the robot formation, which is a function of the actual locations of the robots, is given by the state $x \in \mathbb{R}^n$. This state is treated as the desired value that the consensus algorithm must track, but it is essentially unknown to the robots. Instead portions of this state are measured, and that information is used to update the consensus states. The dynamics of x, given in Equation 5.1, are related to the dynamics of the robots. They are a function of the motions of the robots, and any real-world disturbances they encounter such as wheel slip or uneven terrain.

$$\dot{x} = J(x)v + f_a(x, v, t) = f(x, v, t) \tag{5.1}$$

In this equation, $v \in \mathbb{R}^{2N}$ is a vector containing the angular and linear velocities of all the robots. $J(x) \in \mathbb{R}^{n \times 2N}$ is the Jacobian matrix that relates the angular and linear velocities

of the robots to the derivatives of the states, and is a non-linear function of the states. The unknown function $f_a(x, v, t) \in \mathbb{R}^n$ represents the unknown real world disturbances on the motion of the robots. It also incorporates any unmodeled dynamics of the robots. For convenience, $f(x, v, t) \in \mathbb{R}^n$ is the sum of these two terms. This derivative is not measurable by any individual robot. Even the portion that is a function of the robot control inputs is mostly unknown because each robot is assumed to only know its own velocity.

As previously mentioned, each robot has an information state $x_i \in \mathbb{R}^n$ that represents robot i's understanding of the current formation configuration. This state is an internal control variable, and as such it can be assigned dynamics that facilitate the later control development. Because this state is meant to track x, it would be beneficial to assign dynamics to this state that exactly match that of the formation. Unfortunately, as previously mentioned, most of the dynamics are unknown, so instead the state will be given simple first order dynamics with a control input to be designed later. The dynamics of each robot's information state are chosen as follows in Equation 5.2.

$$\dot{x}_i = u_i, \quad i = 1, \ldots, N \tag{5.2}$$

5.2.2 Control Objective

The control objective is to design a control law $u_i(t)$ such that each robot's information state $x_i(t)$ tracks the current configuration of the formation $x(t)$ in the sense that

$$x_i(t) \to x(t) \quad \text{as} \quad t \to \infty$$

5.2.3 Consensus Protocol Design and Error System

Define a consensus and measurement error for robot i as:

$$e_{ci} = \sum_{j=1}^{N} a_{ij}(x_i - x_j) + B_i(x_i - x), \quad \in \mathbb{R}^n \tag{5.3}$$

Where x_i is the information state of robot i, and x is the state as measured by robot i. The scalars a_{ij} are the elements of the adjacency matrix A that describes the network connections

between the agents. N is the number of agents and n (which does not appear) is the number of states. The matrix $B_i \in \mathbb{R}^{n \times n}$ is a diagonal matrix that represents what states in x are measurable to robot i. All of the off diagonal terms are zero, and the diagonal terms are equal to 1 if robot i can measure the corresponding state, and zero otherwise.

The control algorithm is defined as follows:

$$u_i = -k_c e_{ci} - \hat{f}_i \tag{5.4}$$

$$\hat{f}_i = k_1 \left(e_{ci} - e_{ci}\left(t_0\right)\right) + \int_0^t \left(k_2 \mathrm{sgn}(e_{ci}) + k_3 e_{ci}\right) d\tau \tag{5.5}$$

In 5.4 and 5.5 the scalars $k_1, k_2, k_3, k_c \in \mathbb{R}$ are positive constant control gains. In 5.4 the term $-\hat{f}_i$ serves to estimate a feed-forward term.

To enable the analysis of the entire system as a whole, concatenated vectors are defined as follows.

$$X = \left[x_1^T, \ldots, x_N^T\right]^T \tag{5.6}$$

$$U = \left[u_1^T, \ldots, u_N^T\right]^T \tag{5.7}$$

$$E_c = \left[e_{c1}^T, \ldots, e_{cN}^T\right]^T \tag{5.8}$$

$$\hat{F} = \left[\hat{f}_1^T, \ldots, \hat{f}_N^T\right]^T \tag{5.9}$$

$$B = blockdiag(B_1, B_2, \ldots, B_N) \tag{5.10}$$

Where $X, U, E_c, \hat{F} \in \mathbb{R}^{nN}$ and $B \in \mathbb{R}^{nN \times nN}$. The concatenated error can be written as:

$$E_c = \left(L \otimes I_n\right) X + B\left(X - \mathbf{1} \otimes x\right) \tag{5.11}$$

Where $L \in \mathbb{R}^{N \times N}$ is the Laplacian matrix of the communication graph. For convenience we define a matrix $H = (L \otimes I_n) + B$. Using this definition, the error becomes

$$E_c = HX - B\left(\mathbf{1} \otimes x\right) \tag{5.12}$$

Also

$$\hat{F} = k_1 \left(E_c - E_c\left(t_0\right)\right) + \int_0^t \left(k_2 \mathrm{sgn}(E_c) + k_3 E_c\right) d\tau \tag{5.13}$$

$$\text{sgn}\left(E_c\right) = \left[\text{sgn}\left(e_{11}\right), \ldots, \text{sgn}\left(e_{Nn}\right)\right]^T$$

$$U = -k_c E_c - \hat{F} \tag{5.14}$$

Taking the time derivatives of the concatenated error and estimate law.

$$\begin{aligned} \dot{E}_c =& H\dot{X} - B\left(\mathbf{1} \otimes \dot{x}\right) \\ =& HU - B\left(\mathbf{1} \otimes f\right) \\ =& H\left(-k_c E_c - \hat{F}\right) - B\left(\mathbf{1} \otimes f\right) \end{aligned} \tag{5.15}$$

$$\dot{\hat{F}} = k_1 \dot{E}_c + k_2 \text{sgn}\left(E_c\right) + k_3 E_c \tag{5.16}$$

For further analysis, define an auxiliary error $r \in \mathbb{R}^{nN}$

$$\begin{aligned} r &= H^{-1}\dot{E}_c + k_c E_c \\ &= H^{-1}\left(H\left(-k_c E_c - \hat{F}\right) - B\left(\mathbf{1} \otimes f\right)\right) + k_c E_c \\ &= -k_c E_c - \hat{F} - H^{-1}B\left(\mathbf{1} \otimes f\right) + k_c E_c \\ &= -\hat{F} - H^{-1}\left(H - \left(L \otimes I_n\right)\right)\left(\mathbf{1} \otimes f\right) \\ &= -\hat{F} - \left(\mathbf{1} \otimes f\right) + H^{-1}\left(L \otimes I_n\right)\left(\mathbf{1} \otimes f\right) \\ &= -\hat{F} - \left(\mathbf{1} \otimes f\right) + H^{-1}\left(L\mathbf{1} \otimes f\right) \\ &= -\hat{F} - \left(\mathbf{1} \otimes f\right) = -\hat{F} - G \end{aligned} \tag{5.17}$$

In the above analysis, a property of the Kroncker product ($\otimes$) and a property of the Laplacian matrix L were used. Specifically it is a property of the Laplacian that $L\mathbf{1} = 0$. Also $G = \left(\mathbf{1} \otimes f\right)$. Taking the time derivative of this auxiliary error

$$\begin{aligned} \dot{r} &= -\dot{\hat{F}} - \dot{G} \\ &= -k_1 \dot{E}_c - k_2 \text{sgn}\left(E_c\right) - k_3 E_c - \dot{G} \\ &= -k_3 E_c - k_2 \text{sgn}\left(E_c\right) - \dot{G} - k_1 \dot{E}_c \\ &= -k_3 E_c - k_2 \text{sgn}\left(E_c\right) - \dot{G} - k_1\left(Hr - k_c H E_c\right) \\ &= -k_1 Hr - k_3 E_c - k_2 \text{sgn}\left(E_c\right) - \dot{G} + k_1 k_c H E_c \\ &= -k_1 Hr - k_3 E_c - k_2 \text{sgn}\left(E_c\right) + \Psi \end{aligned} \tag{5.18}$$

Where

$$\Psi = -\dot{G} + k_1 k_c H E_c \tag{5.19}$$

Now for further analysis, define a desired Ψ_d.

$$\Psi_d = -\dot{G} = -\left(\mathbf{1} \otimes \dot{f}\right) \tag{5.20}$$

$$\tilde{\Psi} = \Psi - \Psi_d = k_1 k_c H E_c \tag{5.21}$$

5.2.4 Stability Analysis

To show that the consensus controller will achieve the control objective, Lyapunov stability analysis is used. First a matrix and a function are shown to be positive definite.

Lemma 5.2.1. *The matrix H will be positive definite if the undirected communication graph is connected, and every state is measured by at least one agent.*

Proof. $N \in \mathbb{R}$ is the number of agents. $n \in \mathbb{R}$ is the number of states. The connectivity of the graph is given by the Laplacian matrix $L \in \mathbb{R}^{N \times N}$

For each agent i there is a diagonal matrix $B_i \in \mathbb{R}^{n \times n}$ that describes which states are measured by that agent. The jth diagonal element of B_i is denoted $b_{ij} \in \mathbb{R}$. If $b_{ij} \neq 0$ agent i measures state j. Define a block diagonal matrix $B = \text{blkdiag}\left(B_1, B_2, \cdots, B_N\right) \in \mathbb{R}^{Nn \times Nn}$

For each state j there is a diagonal matrix $P_j \in \mathbb{R}^{N \times N}$ that describes which agents measure that state. The ith diagonal element of P_j is denoted $p_{ji} \in \mathbb{R}$. If $p_{ji} \neq 0$ state j is measured by agent i.We define a block diagonal matrix $P = \text{blkdiag}\left(P_1, P_2, \cdots, P_n\right) \in \mathbb{R}^{Nn \times Nn}$

Note that $b_{ij} = p_{ji}$ however $B_i \neq P_i$ and $B \neq P$. The format of the matrix B is that it contains blocks for each agent with the rows and columns of those blocks corresponding to different states. The matrix P however contains blocks for each state with the rows and columns of those blocks corresponding to different agents. It is useful to define a transformation matrix $T \in \mathbb{R}^{Nn \times Nn}$ such that:

$$T^T B T = P$$

In the matrix $H = (L \otimes I_n) + B$ the term $(L \otimes I_n)$ has the same basic format as B in that it contains blocks for each agent with rows and columns of those blocks corresponding to different states. Because of this similarity, it is clear that

$$T^T (L \otimes I_n) T = (I_n \otimes L)$$

If this transformation is applied to H, a block diagonal matrix is obtained.

$$\begin{aligned} T^T H T =& T^T ((L \otimes I_n) + B) T = (I_n \otimes L) + P \\ =& \text{blkdiag} (L + P_1, L + P_2, \cdots, L + P_n) \end{aligned} \tag{5.22}$$

If the undirected graph $\mathcal{G}_1$ is connected and at least one agent measures state j, then the matrix $L + P_j$ is positive definite.

Since at least one agent has access to the desired trajectory, not all p_{ji}'s are equal to zero. Say $p_{jm} > 0, m \in \{1, ..., N\}$.

Denote the N eigenvalues of L as $\lambda_1, \lambda_2, ..., \lambda_N \in \mathbb{R}$ with $\lambda_1 = 0$ and $\lambda_i > 0, i \in \{2, ..., N\}$. A set of N orthogonal nonzero eigenvectors associated with the N eigenvalues of L are represented as $\xi_i \in \mathbb{R}^N$ with $\xi_1 = \mathbf{1}$. An arbitrary nonzero vector $y \in \mathbb{R}^N$ can be represented as

$$y = a_1 \mathbf{1} + \sum_{i=2}^{N} a_i \xi_i$$

with at least one $a_i \neq 0, i \in \{1, ..., N\}$. If $a_1 = 0$ and $a_j \neq 0$ for at least one $j \in \{2, ..., N\}$, then

$$y^T (L + P_j) y = y^T L y + y^T P_j y \geq \sum_{i=2}^{N} a_i^2 \lambda_i \xi_i^T \xi_i + p_{jm} y_m^2 \geq a_j^2 \lambda_j \xi_j^T \xi > 0$$

If $a_1 \neq 0$ and $a_j = 0$ for all $j \in \{2, ..., N\}$, then

$$y^T (L + P_j) y = y^T L y + y^T P_j y = 0 + p_{jm} y_m^2 > 0$$

If the above conditions are met for all of the states, all of the blocks in $T^T H T$ will be positive definite, and the combined matrix itself will be positive definite. Because T is an invertible transformation, H is also positive definite. □

Lemma 5.2.2. *The function $s(t) \in \mathbb{R}$ defined as follows, is positive definite.*

$$s(t) \triangleq E_c(t_0)^T k_2 sgn(E_c(t_0)) - E_c(t_0)^T \Psi_d(t_0) - q(t) \tag{5.23}$$

$$\dot{q}(t) = r^T H^T (\Psi_d(t) - k_2 sgn(E_c)) \in \mathbb{R}. \tag{5.24}$$

Proof. The integration term $q(t)$ is given by

$$\begin{aligned} q(t) &= \int_{t_0}^{t} r^T H^T \left(\Psi_d(t) - k_2 \mathrm{sgn}(E_c)\right) d\tau \\ &= \int_{t_0}^{t} \left(\dot{E}_c + k_c H E_c\right)^T \left(\Psi_d(t) - k_2 \mathrm{sgn}(E_c)\right) d\tau \\ &= \int_{t_0}^{t} k_c E_c^T H^T \left(\Psi_d - k_2 \mathrm{sgn}(E_c)\right) d\tau + \int_{t_0}^{t} \dot{E}_c^T \Psi_d d\tau - \int_{t_0}^{t} \dot{E}_c^T k_2 \mathrm{sgn}(E_c) d\tau \end{aligned}$$

Integrating the last two terms

$$\begin{aligned} \int_{t_0}^{t} \dot{E}_c^T \Psi_d d\tau &= E_c^T \Psi_d \big|_{t_0}^{t} - \int_{t_0}^{t} E_c^T \dot{\Psi}_d d\tau \\ \int_{t_0}^{t} \dot{E}_c^T k_2 \mathrm{sgn}(E_c) d\tau &= E_c k_2 \mathrm{sgn}(E_c)\big|_{t_0}^{t} \end{aligned}$$

$$\begin{aligned} q(t) &= \int_{t_0}^{t} k_c E_c^T H^T \left(\Psi_d - k_2 \mathrm{sgn}(E_c)\right) d\tau + E_c^T \Psi_d \big|_{t_0}^{t} - \int_{t_0}^{t} E_c^T \dot{\Psi}_d d\tau - E_c k_2 \mathrm{sgn}(E_c)\big|_{t_0}^{t} \\ &= \int_{t_0}^{t} k_c E_c^T H^T \left(\Psi_d - \frac{1}{k_c} H^{-T} \dot{\Psi}_d - k_2 \mathrm{sgn}(E_c)\right) d\tau + E_c^T \Psi_d - E_c\left(t_0\right) \Psi_d\left(t_0\right) \\ &\quad - E_c^T k_2 \mathrm{sgn}\left(E_c\right) + E_c^T\left(t_0\right) k_2 \mathrm{sgn}\left(E_c\left(t_0\right)\right) \\ &\leq \int_{t_0}^{t} k_c ||H E_c|| \left(||\Psi_d|| + \frac{1}{k_c} ||H^{-T} \dot{\Psi}_d|| - k_2\right) d\tau \\ &\quad + ||E_c|| \left(||\Psi_d|| - k_2\right) + E_c^T\left(t_0\right) k_2 \mathrm{sgn}\left(E_c\left(t_0\right)\right) - E_c^T\left(t_0\right) \Psi_d\left(t_0\right) \end{aligned}$$

If $\frac{1}{k_c}||H^{-T}\dot{\Psi}_d|| - k_2 < 0$ then the integral satisfies the following inequality

$$\int_{t_0}^{t} q\left(\tau\right) d\tau \leq E_c^T\left(t_0\right) k_2 \mathrm{sgn}\left(E_c\left(t_0\right)\right) - E_c^T\left(t_0\right) \Psi_d\left(t_0\right)$$

Therefore

$$s\left(t\right) = E_c(t_0)^T k_2 \mathrm{sgn}(E_c(t_0)) - E_c(t_0)^T \Psi_d(t_0) - \int_{t_0}^{t} q(\tau) d\tau \geq 0$$

□

Theorem 5.2.1. *The consensus protocol will converge if the control gains are chosen appropriately.*

Proof. Define a Lyapunov function candidate as:

$$V = \frac{1}{2}k_3 E_c^T E_c + \frac{1}{2}r^T H r + s \tag{5.25}$$

Taking the time derivative

$$\begin{aligned}
\dot{V} =& k_3 E_c^T \dot{E}_c + r^T H \dot{r} + \dot{s} \\
=& k_3 E_c^T \left(Hr - k_c H E_c\right) + r^T H \left(-k_1 H r - k_3 E_c - k_2 \mathrm{sgn}\left(E_c\right) + \Psi\right) \\
& - r^T H \left(\Psi_d - k_2 \mathrm{sgn}\left(E_c\right)\right) \\
=& - k_c k_3 E_c^T H E_c - k_1 r^T H^2 r + k_3 E_c^T H r \\
& - k_3 r^T H E_c - k_2 r^T H \mathrm{sgn}\left(E_c\right) + k_2 r^T H \mathrm{sgn}\left(E_c\right) + r^T H \Psi - r^T H \Psi_d \\
=& - k_c k_3 E_c^T H E_c - k_1 r^T H^2 r + r^T H \tilde{\Psi} \\
=& - k_c k_3 E_c^T H E_c - k_1 r^T H^2 r + r^T H k_1 k_c H E_c \\
=& - k_c k_3 \left(E_c^T H E_c - \frac{1}{k_3} k_1 r^T H^2 E_c + \left(\frac{-1}{2k_3} k_1 H^T r\right)^T H \left(\frac{-1}{2k_3} k_1 H^T r\right)\right) \\
& - k_1 r^T H^2 r + \frac{1}{4k_3} k_1^2 k_c r^T H^3 r \\
=& - k_c k_3 \left(E_c - \frac{k_1 H^T r}{2k_3}\right)^T H \left(E_c - \frac{k_1 H^T r}{2k_3}\right) - k_1 r^T H^2 \left(I - \frac{k_1 k_c}{4k_3} H\right) r
\end{aligned} \tag{5.26}$$

From 5.26 it is apparent that the derivative of the Lyapunov function will be negative semi-definite provided that k_c, k_1, k_3 are selected such that

$$I - \frac{k_1 k_c}{4k_3} H > 0 \text{(i.e., positive definite).} \tag{5.27}$$

□

5.3 Robot Motion Control

Once the current state of the formation is fully determined using the consensus algorithm, the objective becomes controlling the robots to achieve the desired state x_d. Just like the

robots discussed in Section 3.2.2 the robots for this experiment are non-holonomic. Their two control inputs are the forward velocity and angular velocity.

5.3.1 Relative Agent Dynamics

The time derivative of the formation state is related to the velocities of the robots in the formation. To find the derivative of the state, the kinetics of two robots and their relative velocities will be examined. The development is as follows.

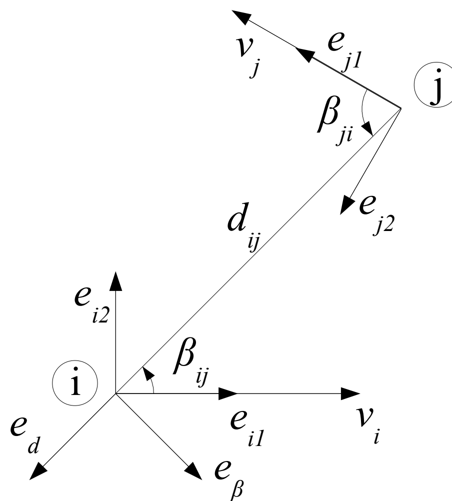

Figure 5.2: *Relative Agent Dynamics*

Consider 2 robots i and j in a formation with velocities and angular velocities of $\vec{v}_i$ $\vec{\omega}_i$ and $\vec{v}_j$ $\vec{\omega}_j$ respectively. The formation shape is defined in terms of d_{ij} θ_{ij} and β_{ij}. Note that β_{ji} is not a state, but is a function of the states. For robot i, define a local coordinate system with basis unit vectors $\vec{e}_{i1}$ and $\vec{e}_{i2}$ where $\vec{e}_{i1}$ points to the front of the robot. In the same way, define a local coordinate system for robot j. Define an auxiliary polar coordinate system relative to j with unit vectors $\vec{e}_d$ and $\vec{e}_\beta$. This configuration is depicted in Fig.5.2

The velocity of robot i can be written as:

$$\vec{v}_i = \vec{v}_j + \vec{v}_{i/j}$$

$$v_i \vec{e}_{i1} = v_j \vec{e}_{j1} + \dot{d}_{ij} \vec{e}_d + d_{ij} \omega_{ij} \vec{e}_\beta$$

Where ω_{ij} is the angular speed of the line connecting i to j. Noting that:

$$\vec{e}_{i1} = -\cos(\beta_{ij}) \vec{e}_d + \sin(\beta_{ij}) \vec{e}_\beta$$

$$\vec{e}_{j1} = \cos(\beta_{ji}) \vec{e}_d - \sin(\beta_{ji}) \vec{e}_\beta$$

The velocity can be rewritten as:

$$v_i \left(-\cos(\beta_{ij}) \vec{e}_d + \sin(\beta_{ij}) \vec{e}_\beta\right) = v_j \left(\cos(\beta_{ji}) \vec{e}_d - \sin(\beta_{ji}) \vec{e}_\beta\right) + \dot{d}_{ij} \vec{e}_d + d_{ij} \omega_{ij} \vec{e}_\beta$$

Equating like terms on either side of the equals sign gives

$$\vec{e}_d : -v_i \cos(\beta_{ij}) = v_j \cos(\beta_{ji}) + \dot{d}_{ij}$$

$$\vec{e}_\beta : v_i \sin(\beta_{ij}) = -v_j \sin(\beta_{ji}) + d_{ij} \omega_{ij}$$

Rearranging these equations, the equations for the following derivatives are obtained

$$\begin{aligned} \dot{d}_{ij} &= -v_i \cos(\beta_{ij}) - v_j \cos(\beta_{ji}) \\ \omega_{ij} &= \frac{v_i}{d_{ij}} \sin(\beta_{ij}) + \frac{v_j}{d_{ij}} \sin(\beta_{ji}) \end{aligned}$$

Based on the geometry of the problem, $\omega_{ij} = \omega_i + \dot{\beta}_{ij}$ and $\beta_{ji} = \beta_{ij} - \theta_{ij} - \pi$ thus the derivatives are obtained as:

$$\dot{d}_{ij} = -v_i \cos(\beta_{ij}) - v_j \cos(\beta_{ij} - \theta_{ij} - \pi)$$

$$\dot{\beta}_{ij} = -\omega_i + \frac{v_i}{d_{ij}} \sin(\beta_{ij}) + \frac{v_j}{d_{ij}} \sin(\beta_{ij} - \theta_{ij} - \pi)$$

From the definition of the relative angle:

$$\begin{aligned} \theta_{ij} &= \theta_j - \theta_i \\ \dot{\theta}_{ij} &= \dot{\theta}_j - \dot{\theta}_i \\ \dot{\theta}_{ij} &= \omega_j - \omega_i \end{aligned}$$

Finally all of the state derivatives are given by:

$$\begin{aligned}\dot{d}_{ij} &= -v_i \cos(\beta_{ij}) + v_j \cos(\beta_{ij} - \theta_{ij}) \\ \dot{\beta}_{ij} &= -\omega_i + \frac{v_i}{d_{ij}} \sin(\beta_{ij}) - \frac{v_j}{d_{ij}} \sin(\beta_{ij} - \theta_{ij}) \\ \dot{\theta}_{ij} &= \omega_j - \omega_i\end{aligned} \tag{5.28}$$

And the combined state derivative is:

$$\dot{x} = \begin{bmatrix} \dot{d}_{12} \\ \dot{\beta}_{12} \\ \dot{\theta}_{12} \\ \dot{d}_{23} \\ \dot{\beta}_{23} \\ \dot{\theta}_{23} \end{bmatrix} = \begin{bmatrix} -v_1 \cos(\beta_{12}) + v_2 \cos(\beta_{12} - \theta_{12}) \\ -\omega_1 + \frac{v_1}{d_{12}} \sin(\beta_{12}) - \frac{v_2}{d_{12}} \sin(\beta_{12} - \theta_{12}) \\ \omega_2 - \omega_1 \\ -v_2 \cos(\beta_{23}) + v_3 \cos(\beta_{23} - \theta_{23}) \\ -\omega_2 + \frac{v_2}{d_{23}} \sin(\beta_{23}) - \frac{v_3}{d_{23}} \sin(\beta_{23} - \theta_{23}) \\ \omega_3 - \omega_2 \end{bmatrix} \tag{5.29}$$

5.3.2 Low-Level Motion Controller

Equation 5.29 can be rewritten as a product of a Jacobian matrix J and a vector of control inputs v.

$$\dot{x} = \begin{bmatrix} -\cos(\beta_{12}) & 0 & \cos(\beta_{12} - \theta_{12}) & 0 & 0 & 0 \\ \frac{1}{d_{12}} \sin(\beta_{12}) & -1 & -\frac{1}{d_{12}} \sin(\beta_{12} - \theta_{12}) & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -\cos(\beta_{23}) & 0 & \cos(\beta_{23} - \theta_{23}) & 0 \\ 0 & 0 & \frac{1}{d_{23}} \sin(\beta_{23}) & -1 & -\frac{1}{d_{23}} \sin(\beta_{23} - \theta_{23}) & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \\ v_3 \\ \omega_3 \end{bmatrix} \tag{5.30}$$

Suppose that J is invertible and x_d and $\dot{x}_d$ are provided for feedback, then the controller $[v_i,\ \omega_i]^T$ can be designed as

$$v = J^{-1} \left(-k \left(x - x_d \right) + \dot{x}_d + \hat{f} \right) \tag{5.31}$$

If J is singular, an estimate J_d based on desired information x_d will be used instead.

5.4 Hardware Implementation

The robots used for this experiment were three Pioneer 3-DX robots. Each one was equipped with an on board computer mounted to the top of the robot. Also mounted to the top of the

robots where three Sony XCD-SX910 firewire cameras. These were mounted with a custom cardboard bracket. The brackets were designed to hold the cameras in place roughly level while allowing some motion in the event of a collision. This, less than rigid, mounting dampened some vibrations, and helped protect the cameras. Each camera was equipped with a Pentax TV Lens with a focal length of 6 mm. This focal length was chosen to give a reasonable field of view for the cameras. A robot with the attached computer and camera is shown in Figure 5.3.

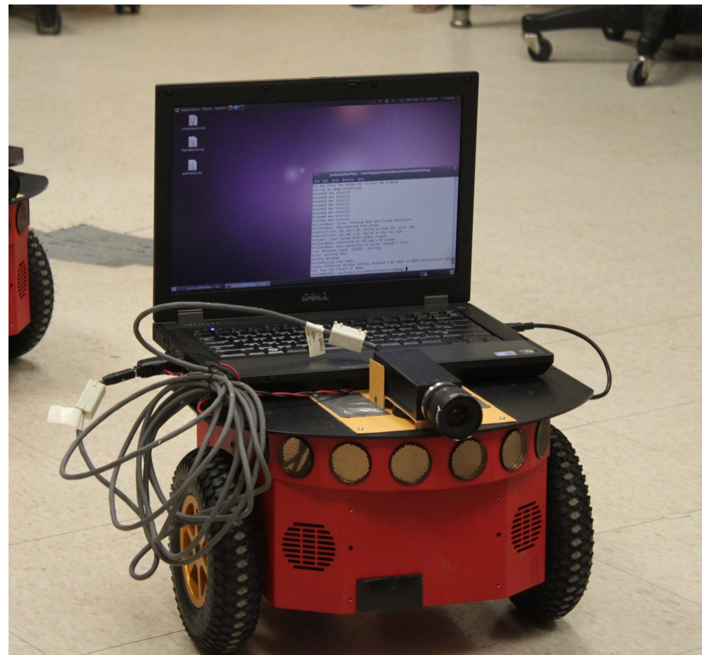

Figure 5.3: *Hardware configuration showing: robot, laptop, camera, mounting bracket, and modified cable*

The cameras draw their power from the firewire cable. One of the laptops used is equipped with a 6-pin powered firewire port that can supply the required power. The other two laptops, however, only have non-powered 4-pin firewire ports. To provide power to those cameras, special cables were constructed that allowed the power pins to be tied into the robot's power distribution board. The robot's power distribution boards are equipped with screw terminals for powering external accessories. Referencing the data sheet for the

camera and the robot, the power supplied at these terminals was found to be appropriate for powering the cameras without any modification. So the two power pins on a 6-pin firewire connector were tied to these terminals using some auxiliary wires, while the four data pins were connected to a standard 4-pin connector.

Each laptop was positioned so that the top of the lid was facing the rear of the robot. The target pattern for the image processing was attached to the lid of the laptop allowing it to be easily visible to the robot following, and making its angle adjustable to ensure that it is vertical. The target pattern used was a chessboard with four squares to a side centered on a white sheet of paper. The chessboard pattern was chosen to simplify the image processing tasks. OpenCV already had built in functions that could locate the internal corners of a chessboard pattern in an image. Also the internal corners of the chessboard provided good high contrast points for tracking by optical flow algorithms.

5.5 Software Implementation

The structure is similar in some ways to the distributed formation tracking experiment. The consensus code in particular, is based on the code from the distributed experiment. It only required a few modifications. Mainly, it does not calculate the reference trajectory any more, instead the reference signals come from the image measurements. The control action follows the same basic underlying structure as was used previously, except the control algorithm has been changed significantly.

The most drastic change was the addition of the image processing code. The image processing is done inside the main function. After the initialization of all of the other components, and the establishment of the network connections the image processing is initialized. Then there is a image processing loop that waits for the robot to be stopped to end. Actually, it is waiting for the user to press the escape key that triggers the key handler attached to the robot and signals the program to end.

Within this loop, images are captured and processed to extract the desired feature points.

Initially a function that searches the image for a chessboard pattern is used to locate the feature points. Unfortunately, this function takes a long time to find the chessboard, so if it was used exclusively, the image processing loop would execute slower than the image capture rate of the camera. To solve this problem, after the points have been found, an optical flow algorithm tracks them from one image to the next. The optical flow algorithm has less computational overhead, so it can locate the feature points in a new image faster than the camera is capturing images. Then using these feature points, and ones extracted from a reference image, the homography matrix is composed. Then homography decomposition and the algorithm developed in Section 4.4 are used to calculate the states. These states are then used to update the desired values of the consensus algorithm.

5.6 Implementation Challenges

As with previous experiments, there were several challenges to the implementation. Some challenges arose from the decentralized nature of the experiment. Most came from the added complexity of incorporating visual measurements. Some problems were avoided by relying on experience from the previous experiments. For example, there were very few issues with the network communication between the robots because most of those issues were solved in the previous experiments.

One challenge was choosing an appropriate final goal for the experiment. The motion controller developed for the robots involves calculating the inverse of a Jacobian matrix. This is not possible if that matrix is singular. It was found that certain combinations of states would make this matrix singular. This limited the possible final configurations for the robots because the motion controller could not drive the robots to a singular configuration. In particular it became apparent that the final configuration could not have all the robots with the same heading. Looking at the physical interpretation of these singular states, they arise when non-zero robot velocities result in zero formation change. For example, if all of the robots have the same heading, and the same non-zero forward velocity, the

formation shape will not change despite the fact that the robots are moving. The solution is to design a formation shape where any non-zero robot velocities would result in a change in the formation states. This is how the final formation shape was chosen for the experiment.

The biggest challenge for this experiment was losing visibility between robots. In order for the experiment to work properly, the robots making measurements must continue to do so. Unfortunately, often one robot or the other would move in such a way that the target would leave the camera's field of view. This posed several challenges. At first when this happened, the experiment essentially failed. There is no algorithm implemented to restore the formation to a visible state after this happens. So, when the visibility was lost, there was no guarantee that it would be restored. Also, often the optical flow algorithm would continue to return points for a frame or two after the points it was tracking were lost. This resulted in drastically incorrect values of the state being passed to the consensus algorithm. Another issue arose from the image processing algorithm. When the target would be lost, the image processing algorithm would revert to searching the image for a chessboard pattern. As previously mentioned, this made the image processing loop slower than the image capture rate, but the camera continued to capture images at the same rate. This resulted in a delay accumulating between when an image was captured and when it was processed. The issue being that if the image processing did manage to re-acquire the target, it would be out of the field of view again before the image processing could catch up.

Many different steps were required to alleviate the issues with the loss of visibility. The main goal was to prevent the visibility from being lost in the first place. Also, when it was lost, the algorithm needed to be more robust and more likely to recover. The first action taken was to saturate the control inputs to the robot, particularly the angular velocity. This prevented spikes in the inputs from causing drastic rotations that would cause the visibility to be lost. Other steps included turning down the motion control gains, and making the final configuration more stable. This made a significant improvement in maintaining the visibility. Another change that had a positive improvement was tuning the consensus controller to track

the measured states more accurately. Transients and delays in the consensus, particularly in the beginning, were causing the motion controller to give the robots incorrect commands. To improve the robustness, several sanity checks were implemented in the code that calculates the states. This helps prevent unreasonable states from being passed along to the consensus algorithm. This has the effect of freezing the desired value for the consensus algorithm at the last known good measurement until a new measurement is made. Finally, the image processing code was modified so that when it is searching the whole image for the chessboard, it only operates on every other image. This helped to prevent the delay that was being experienced. All of these steps combined allowed the experiment to function correctly. It also made it more tolerant to faults in the image processing.

5.7 Experimental Setup

To setup the experiment, all the key parameters were set in advance, then a starting configuration was chosen. The staring configuration was chosen to be far from the final configuration while still maintaining visibility between the robots. The possible starting configurations were limited by the constraint that the visibility between the appropriate robots had to be established at the beginning of the experiment.

The starting configuration of the robots used in the experiment is shown in Figure 5.4.

For the measurement configuration, the first robot was considered to be able to see the second robot and the second robot could see the third robot. Given the measurement assumptions, the first robot would measure the first three states and the second robot would measure the last three states. In this configuration, the third robot does not measure any states. Mathematically this configuration can be described with the B_i matrices mentioned

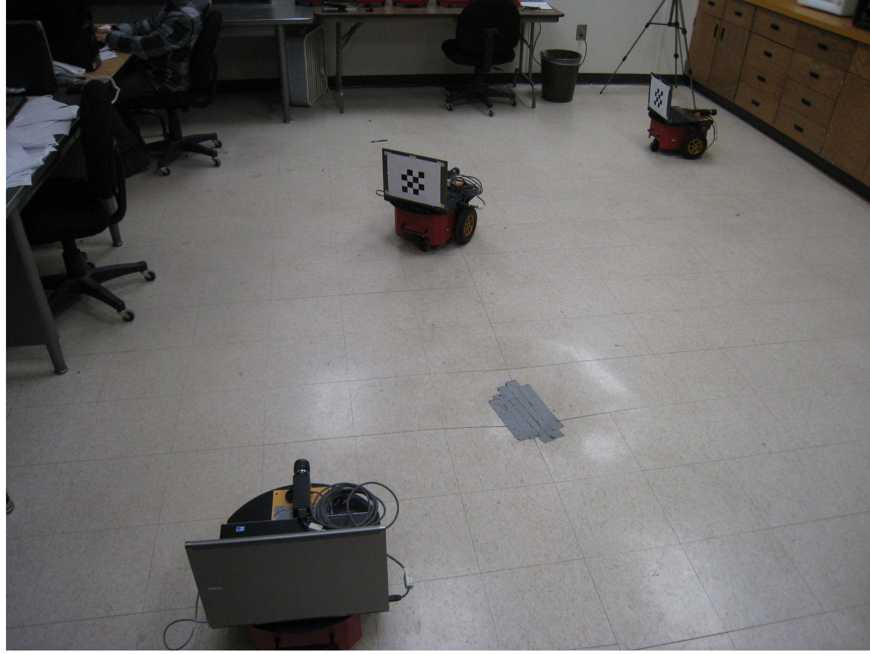

Figure 5.4: *Formation initial configuration*

in Equation 5.3, these matrices are given for each robot in Equation 5.32.

$$B_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad B_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad B_3 = [0] \tag{5.32}$$

The desired reference configuration is given in Equation 5.33.

$$x_d = \begin{bmatrix} 1[m] \\ -\frac{\pi}{12}[rad] \\ -\frac{\pi}{6}[rad] \\ 1[m] \\ \frac{\pi}{12}[rad] \\ \frac{\pi}{6}[rad] \end{bmatrix} \tag{5.33}$$

Also Figure 5.5 shows the robots in this configuration. The information exchange between the agents was specified by an adjacency matrix A, which is given in Equation 5.34.

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{5.34}$$

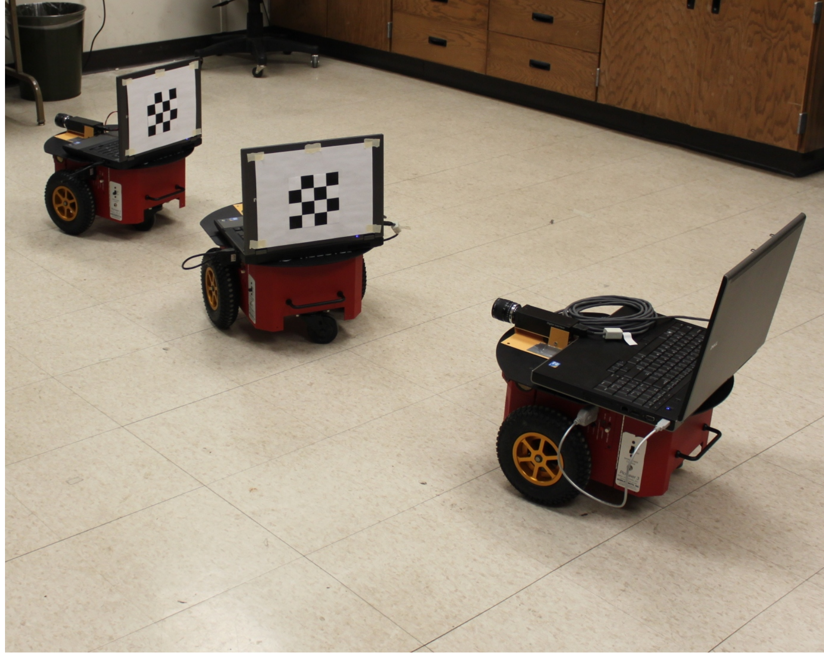

Figure 5.5: *Formation final configuration*

The initial conditions for the consensus states were all set to zero. Additionally, the desired or measured state for the consensus algorithm was initialized to zero as well. This was important because the consensus calculations began running before the image processing began. Setting both the initial conditions and the measured state to zero prevented the consensus state from changing before the image processing had started taking valid measurements. Also, an all zero state resulted in a singularity in the motion control. This was actually helpful because it prevented the motion controller from calculating any control inputs until after the consensus states were non-zero. This ensured that the robots did not start moving until the network communications were established, and the image processing had started.

The control gains from Equations 5.4, 5.5, and 5.31 are given in Table 5.1.

Gain	Value
k_1	1
k_2	7
k_3	10
k_c	11
k	$= diag(0.2)$

Table 5.1: *Experimental Control Gains*

5.8 Experimental Results

After the experiment was run successfully, the collected data was consolidated and analyzed to evaluate the performance. This task was somewhat complicated by the fact that most of the data had different time scales. Data from the different robots started at different times, and data from the consensus threads started at different times from the data collected in the robot control thread. Also, the time steps between data points were not always consistent, particularly in the robot control data. Using corresponding points in the communications data, the time arrays for each data set were shifted to have a common reference point. Then the data was graphed.

Figure 5.6 shows the measured value and each robot's consensus understanding of two states. These two states represent the distance between the robots as measured by the image processing. As previously stated the consensus states and the measured values are initially zero. Then when the image measurements begin, there is a step change. There is a noticeable amount of overshoot in the consensus states. Though the states quickly settle down and track the measurements quite well.

The two measured bearing angles and the corresponding consensus understandings are graphed in Figure 5.7. These states represent the angle measured relative to the robot's heading to the next robot in the formation. The overshoot is less of an issue for these states. These measured states show more noise than the distances. Much of this noise is actually filtered by the consensus algorithm giving smoother states for the control of the robot.

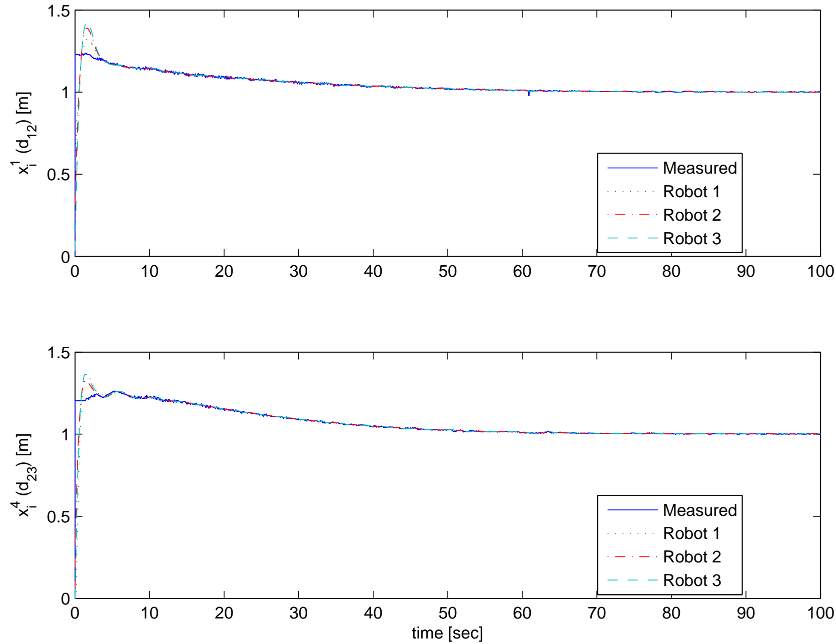

Figure 5.6: *Consensus tracking results for states d_{12}, d_{23} and their reference states*

The final two consensus states are graphed in Figure 5.8. Again, the measured states are graphed along with each robot's consensus understanding. These states are the relative rotation between the robots' headings. In particular, the relative rotation between the first and second, and the second and third robots. The measured values show the same noise seen in the plot of the bearing angles, and after an initial transient, the tracking is quite good.

Figure 5.9 demonstrates the accomplishment of the experiment's primary goal. That is to drive the state of the formation to a desired value. The figure shows the error between the measured states and the desired formation shape. These all eventually converge nicely to zero. A few states appear to have some slight steady state error. This is likely due to the robot rounding control inputs to whole numbers. The result being that angular velocity commands less than 1 degree per second do not get executed. It is noticeable in this figure that the second robot's image processing started a few seconds before the first robot's.

The consensus error for the first robot is plotted in Figure 5.10. These results are typical

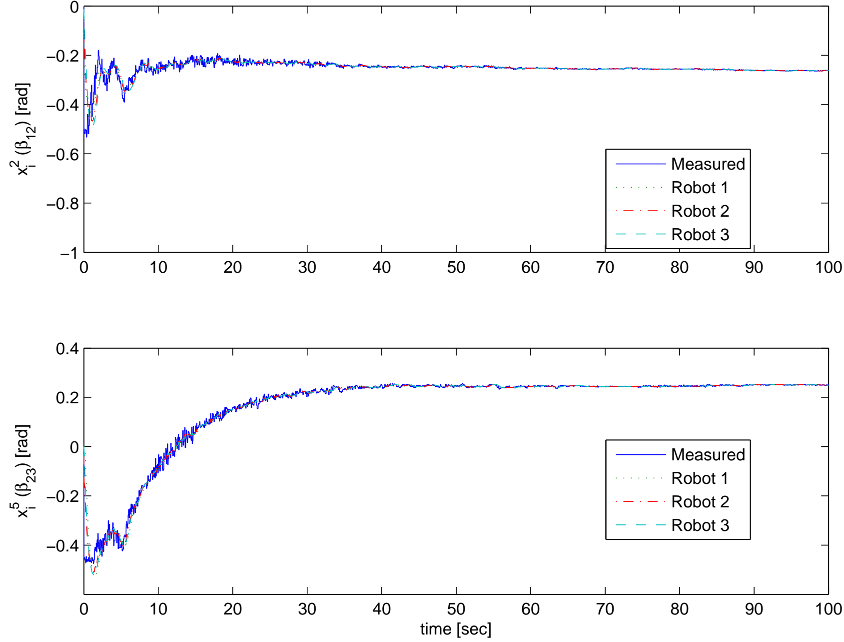

Figure 5.7: *Consensus tracking results for states β_{12}, β_{23} and their reference states*

for all there robots.

Figure 5.11 shows the non-linear estimation term for all six states for the first robot. These results are typical for all three robots. Initially, there is a large spike growing from zero. These spikes are particularly noticeable for the two distance states. They are the result of the step change and overshoot seen at the beginning of the consensus states.

Each robot's forward velocity command is shown in Figure 5.12. The velocity commands were actually saturated before being sent to the robot, but this figure shows the calculated un-saturated velocities. Within the first few seconds the calculated velocity is quite large. This portion is due in part to the transients in the consensus states. After this initial phase, the velocities settle down nicely.

Figure 5.13 shows the calculated angular velocity commands for each robot. Like the forward velocity, the angular velocity commands were saturated before being sent to the robot. Because of the significant saturation many of the spikes in the first few seconds were filtered out.

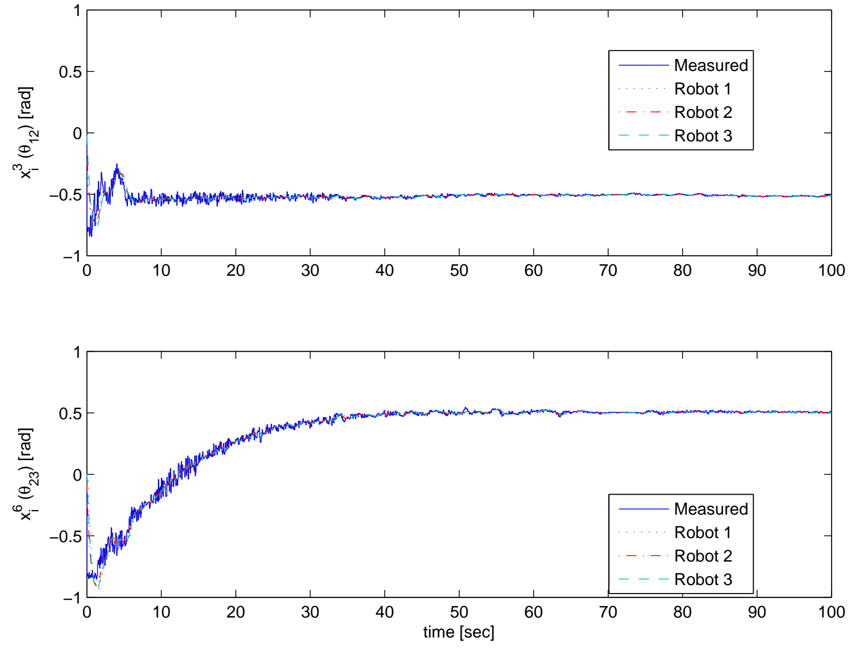

Figure 5.8: *Consensus tracking results for states θ_{12}, θ_{23} and their reference states*

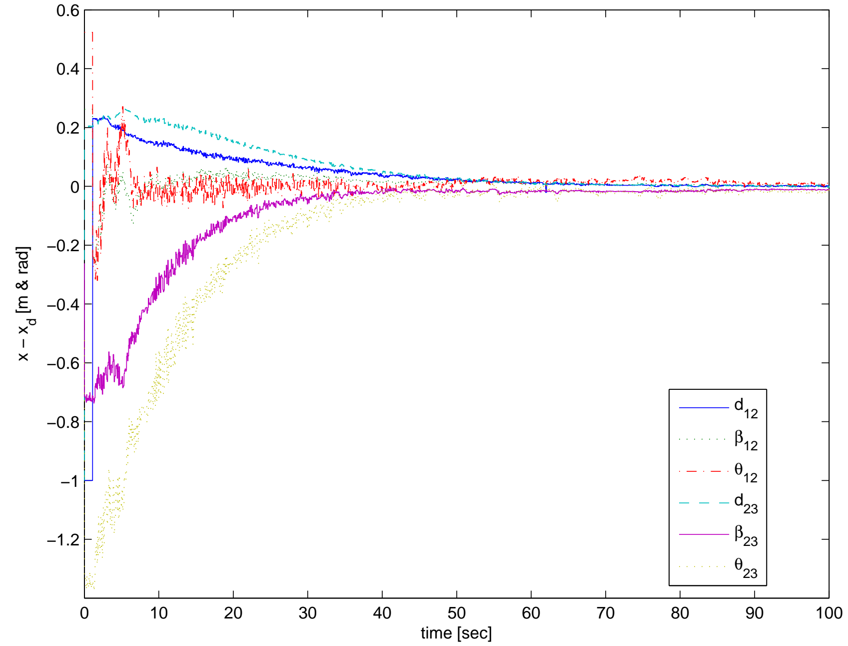

Figure 5.9: *Visual formation tracking error showing convergence to desired values*

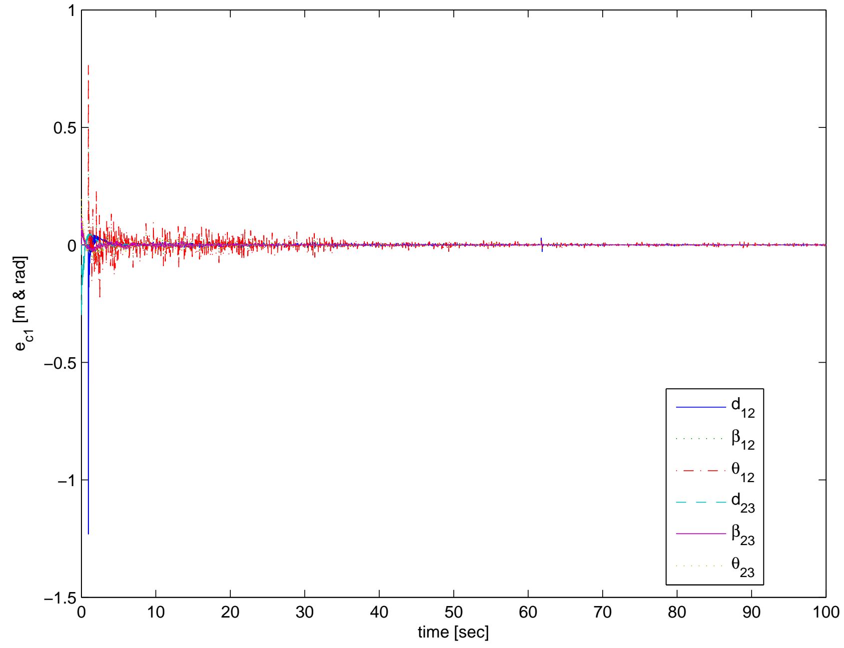

Figure 5.10: *Vision based formation consensus tracking error for agent 1*

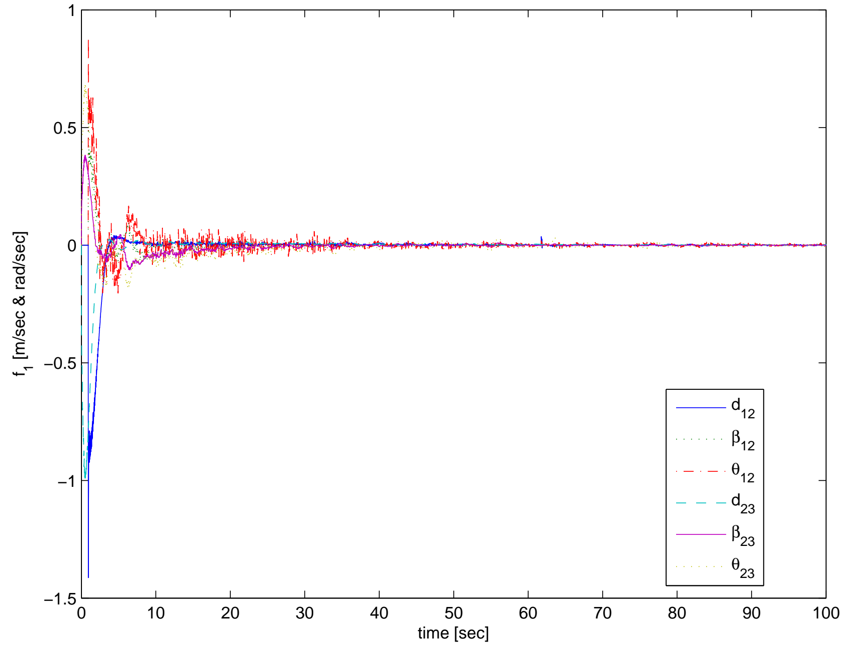

Figure 5.11: *Vision based formation consensus derivative estimate for agent 1*

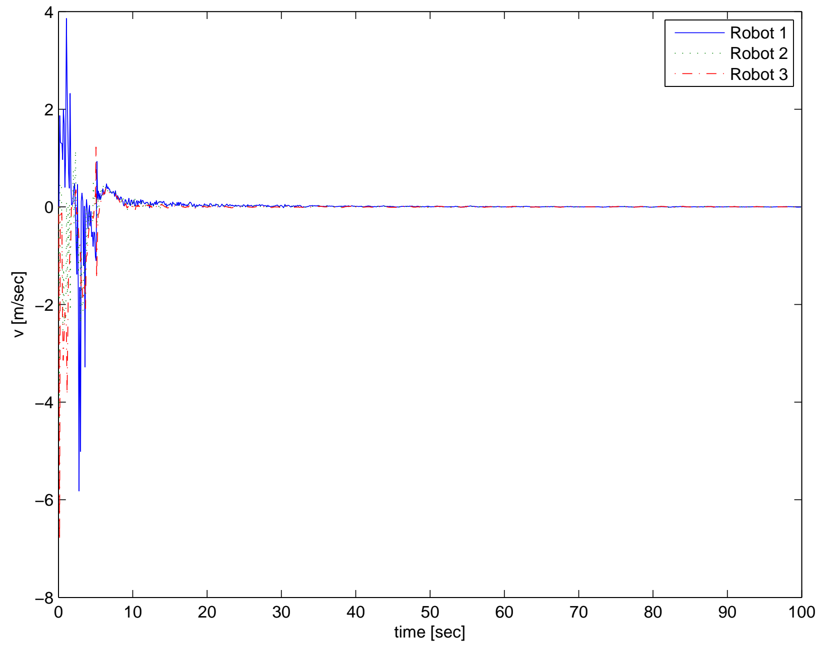

Figure 5.12: *Robot forward velocity commands*

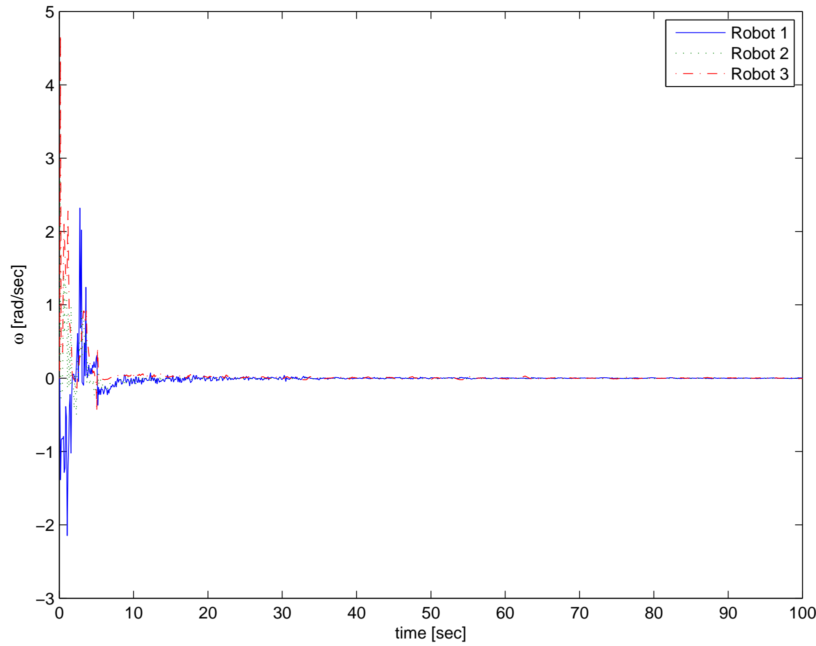

Figure 5.13: *Robot angular velocity commands*

Chapter 6

Conclusion

In this thesis, several robot formation control algorithms were presented. Experiments were conducted to demonstrate the effectiveness of those algorithms. Results of these experiments were presented demonstrating their success.

The first main experiment focused on robust consensus formation control. The algorithm for this experiment was discussed in Section 3.2. The primary focus of the algorithm is robust consensus tracking. All of the agents came to agree on a time-varying reference signal despite disturbances applied to their information exchange. It compensated for the disturbances with a special non-linear term that estimates them. For the motion control of the robots, they compute a desired trajectory from the consensus state, and implement a trajectory tracking controller to follow it. To compensate for the non-holonomic constraint on the robots, the position of the front of the robots is controlled instead of the robot's center. The trajectory tracker uses a PID controller to design the velocity of this point; then computes the required linear and angular velocity inputs for the robot.

After the algorithm was developed, the AmigoBot robots described in Section 2.1.1 were used to demonstrate it in the experiment. The robots were able to achieve the desired formation shape of a square, and the formation was able to track the desired trajectory. Results from the experiment are in Section 3.3.5. All of the robots in the experiment were controlled by a single computer, and all of the information exchange for the consensus algorithm was modeled using a central data storage.

The previous experiment showed the consensus algorithm working, but poorly demonstrated the distributed information exchange between the robots. In the development it was intended that robots could only access the consensus states of their neighbors, and that this data would be communicated over a network. In the original experiment, limited information exchange between the agents had to be artificially enforced, and possible complications from the network were not modeled. To try and better capture these complications, the experiment was re-implemented using a separate computer to control each robot. This second experiment is discussed in Section 3.4. In order to solve issues arising from the network communication, the software program had to be reorganized separating the consensus controller from the motion controller. This had the added benefit of improving the performance of the consensus algorithm. The results from this experiment are all in Section 3.4.5.

It was decided to use on-board cameras as the primary sensor for the next experiment, so a new approach had to be formulated to incorporate visual measurements. Previously, all of the robots were told their initial starting locations, and used encoders on the motors to keep track of their positions. Instead of this approach, which relied on a common global reference frame, it was decided to reformulate the problem to rely on relative measurements made with vision. First in Section 4.2, a formulation was developed which could describe the robot formation in terms of only relative measurements. Then, in Section 4.4, a process was developed to make relative measurements with cameras mounted to the robots. This process involved relating a current image to a previously captured reference image.

One challenge to the new formation description was that a complete description of the formation would not be measured by any one robot. Also, the different pieces of information were measured by different robots, and some robots took no measurements at all. To incorporate all of the data, and ensure that all of the robots had access to it, a consensus tracking controller was developed. This controller is similar to the one used previously with a few key differences. The development of this controller can be found in Section 5.2. This consensus controller, instead of tracking a desired signal calculated by a leader, tracks the

measurements made by the cameras. Also, there is no single leader for the information, instead, each agent serves as a leader for the information it measures. The convergence and stability of this algorithm is shown using a Lyapunov based stability analysis. This controller could have applications beyond the task it was designed for; in particular, it allows the fusion of distributed measurements made by agents with limited communication.

A motion controller for the formation was designed based on the formation state description previously developed. A controller was designed to stabilize the formation state derivatives and drive the state to a desired final value. Then an inverse Jacobian matrix was used to relate these state derivatives to the velocity control inputs of the robots. There were some issues that had to be resolved relating to singularities in the Jacobian matrix, but solutions were found.

Finally, a new experiment using these tools was run involving a formation of three robots with on-board cameras. They were able to achieve the final desired formation using only visual measurements after starting at previously unspecified initial positions. Section 5.8 shows the results of this experiment.

Overall, several successful experiments were conducted with formations of robots. These experiments demonstrated several formation control algorithms that focused on distributed control. The derivation of some of these algorithms were also detailed. In addition, a method of obtaining measurements describing the relative positions of robots using on-board cameras was developed. Along with this measurement scheme, a process for describing a formation of robots in terms of these relative measurements and also a method using these to control the formation was developed. These pieces combined to result in a insightful examination into robot formation control experiments with distributed control.

Bibliography

[1] X. Shen, J. Dumpert, and S. Farritor, "Design and control of robotic highway safety markers," *Mechatronics, IEEE/ASME Transactions on*, vol. 10, no. 5, pp. 513–520, 2005.

[2] J. R. Lawton and R. W. Beard, "Synchronized multiple spacecraft rotations," *Automatica*, vol. 38, no. 8, pp. 1359–1364, Aug. 2002.

[3] P. Ogren, E. Fiorelli, and N. Leonard, "Cooperative control of mobile sensor networks:Adaptive gradient climbing in a distributed environment," *Automatic Control, IEEE Transactions on*, vol. 49, no. 8, pp. 1292–1302, 2004.

[4] J. Cortes, S. Martinez, T. Karatas, and F. Bullo, "Coverage control for mobile sensing networks," *Robotics and Automation, IEEE Transactions on*, vol. 20, no. 2, pp. 243–255, 2004.

[5] T. Balch and R. Arkin, "Behavior-based formation control for multirobot teams," *Robotics and Automation, IEEE Transactions on*, vol. 14, no. 6, pp. 926–939, 1998.

[6] L. Parker, "ALLIANCE: an architecture for fault tolerant multirobot cooperation," *Robotics and Automation, IEEE Transactions on*, vol. 14, no. 2, pp. 220–240, 1998.

[7] J. Lawton, R. Beard, and B. Young, "A decentralized approach to formation maneuvers," *Robotics and Automation, IEEE Transactions on*, vol. 19, no. 6, pp. 933–941, 2003.

[8] M. A. Lewis and K. Tan, "High precision formation control of mobile robots using virtual structures," *Autonomous Robots*, vol. 4, no. 4, pp. 387–403, Oct. 1997.

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

[10] A. Das, R. Fierro, V. Kumar, J. Ostrowski, J. Spletzer, and C. Taylor, “A vision-based formation control framework,” *Robotics and Automation, IEEE Transactions on*, vol. 18, no. 5, pp. 813–825, 2002.

[11] J. Huang, S. Farritor, A. Qadi, and S. Goddard, “Localization and follow-the-leader control of a heterogeneous group of mobile robots,” *Mechatronics, IEEE/ASME Transactions on*, vol. 11, no. 2, pp. 205–215, 2006.

[12] J. Shao, G. Xie, and L. Wang, “Leader-following formation control of multiple mobile vehicles,” *Control Theory & Applications, IET*, vol. 1, no. 2, pp. 545–552, 2007.

[13] D. Sun, C. Wang, W. Shang, and G. Feng, “A synchronization approach to trajectory tracking of multiple mobile robots while maintaining Time-Varying formations,” *Robotics, IEEE Transactions on*, vol. 25, no. 5, pp. 1074–1086, 2009.

[14] R. Olfati-Saber, J. Fax, and R. Murray, “Consensus and cooperation in networked Multi-Agent systems,” *Proceedings of the IEEE*, vol. 95, no. 1, pp. 215–233, 2007.

[15] J. Fax and R. Murray, “Information flow and cooperative control of vehicle formations,” *Automatic Control, IEEE Transactions on*, vol. 49, no. 9, pp. 1465–1476, 2004.

[16] S. Khoo, L. Xie, and Z. Man, “Robust Finite-Time consensus tracking algorithm for multirobot systems,” *Mechatronics, IEEE/ASME Transactions on*, vol. 14, no. 2, pp. 219–228, 2009.

[17] W. Ren, “Consensus tracking under directed interaction topologies: Algorithms and experiments,” *Control Systems Technology, IEEE Transactions on*, vol. 18, no. 1, pp. 230–237, 2010.

[18] N. Moshtagh, N. Michael, A. Jadbabaie, and K. Daniilidis, "Vision-Based, distributed control laws for motion coordination of nonholonomic robots," *Robotics, IEEE Transactions on*, vol. 25, no. 4, pp. 851–860, 2009.

[19] *Team AmigoBot Operations Manual*, MobileRobots Inc., January 2007.

[20] *Pioneer 3 Operations Manual*, MobileRobots Inc., July 2007.

[21] *Digital Video Camera Module Technical Manual, XCD-SX910*, Sony Corporation, 2003.

[22] J. Cook and G. Hu, "Experimental verification and algorithm of a multi-robot cooperative control method," in *Advanced Intelligent Mechatronics (AIM), 2010 IEEE/ASME International Conference on*, July 2010, pp. 109 –114.

[23] G. Hu, "Robust consensus tracking for an integrator-type multi-agent system with disturbances and unmodelled dynamics," *International journal of control*, vol. 84, no. 1, pp. 1–8, 2011.

[24] G. Hu, S. Mehta, N. Gans, and W. Dixon, "Daisy chaining based visual servo control part i: Adaptive quaternion-based tracking control," in *Control Applications, 2007. CCA 2007. IEEE International Conference on*, Oct. 2007, pp. 1474 –1479.

[25] G. Hu, N. Gans, S. Mehta, and W. Dixon, "Daisy chaining based visual servo control part ii: Extensions, applications and open problems," in *Control Applications, 2007. CCA 2007. IEEE International Conference on*, Oct. 2007, pp. 729 –734.

Appendix A

Code For Distributed Formation Tracking Experiment

```
/*
 * formationTracking.cpp
 *
 * Written by: Joshua Cook
 * Date: 01/19/10
 *
 * Program to perform formation and consensus tracking and control with
     multiple robots.
 * Each robot is controlled by a different instance of the program.
 */

// Include all the Aria headers etc
#include "Aria.h"
#include "ArNetworking.h"
#include "ActionFormation.h"
#include "ConsensusThread.h"


int main(int argc, char **argv)
{
    // variables
    char* host;
    char* host1;
    char* host2;
    char* host3;
    char* host4;
    std::vector<std::string> hosts;
    int port;
    int port1;
```

```
int port2;
int port3;
int port4;
std::vector<int> ports;
int IDnumber;
int ret;
std::string str;
bool argSet = false;

// Other objects
ArKeyHandler keyHandler;

// Robot's objects
ArRobot robot;                      // The robot
ArTcpConnection connection; // The tcp connection for the robot
ArSonarDevice sonar;                // Sonar for the robot

// Initialize Aria
Aria::init();

//
// Parse out all of the command line options.
//

// Create and argument parser to parse out the command line options
ArArgumentParser parser(&argc, argv);
parser.loadDefaultArguments();

// Determine my robot's ID number in the formation
argSet = false;
parser.checkParameterArgumentInteger("-IDnumber", &IDnumber, &
   argSet);

// Figure out connection information for the robots
// User can specify hostname and port number at command line
host = parser.checkParameterArgument("-rh");      // My robot's
   remote host name
if(!host) host = "localhost";
host1 = parser.checkParameterArgument("-rh1");  // Host name for
   remote server
if(!host1) host1 = "localhost";
host2 = parser.checkParameterArgument("-rh2");  // Host name for
   remote server
if(!host2) host2 = "localhost";
host3 = parser.checkParameterArgument("-rh3");  // Host name for
```

```
    remote server
if(!host3) host3 = "localhost";
host4 = parser.checkParameterArgument("-rh4");   // Host name for
    remote server
if(!host4) host4 = "localhost";

// Set default ports
port = 8101;
port1 = 7272;
port2 = 7272;
port3 = 7272;
port4 = 7272;
if(strcmp(host1, host2) == 0 )
{
    // same host, it must be using two ports (but can still
        override below with -rp2)
    port2++;
}
if(strcmp(host2, host3) == 0 )
{
    // same host, it must be using two ports (but can still
        override below with -rp3)
    port3 = port2 + 1;
}
if(strcmp(host3, host4) == 0 )
{
    // same host, it must be using two ports (but can still
        override below with -rp4)
    port4 = port3 + 1;
}

// Check to see if ports were specified
argSet = false;
parser.checkParameterArgumentInteger("-rp", &port, &argSet);
if(!argSet) parser.checkParameterArgumentInteger("-rrtp", &port);
argSet = false;
parser.checkParameterArgumentInteger("-rp1", &port1, &argSet);
if(!argSet) parser.checkParameterArgumentInteger("-rrtp1", &port1);
argSet = false;
parser.checkParameterArgumentInteger("-rp2", &port2, &argSet);
if(!argSet) parser.checkParameterArgumentInteger("-rrtp2", &port2);
argSet = false;
parser.checkParameterArgumentInteger("-rp3", &port3, &argSet);
if(!argSet) parser.checkParameterArgumentInteger("-rrtp3", &port3);
argSet = false;
```

```
parser.checkParameterArgumentInteger("-rp4", &port4, &argSet);
if(!argSet) parser.checkParameterArgumentInteger("-rrtp4", &port4);

// Store hostnames and ports
hosts.push_back(host1);
hosts.push_back(host2);
hosts.push_back(host3);
hosts.push_back(host4);
ports.push_back(port1);
ports.push_back(port2);
ports.push_back(port3);
ports.push_back(port4);

// Check to see of the help option was given at the command line,
// and warn about unparsed arguments
if (!parser.checkHelpAndWarnUnparsed())
{
    ArLog::log(ArLog::Terse, "Usage: formationTracking -IDnumber <
        number> [-rh <hostname>] [-rp <port>]\n"\
            "[-rh1 <hostname1>] [-rh2 <hostname2>] [-rh3 <hostname3
                >] [-rh4 <hostname4>]\n"\
            "[-rp1 <port1>] [-rp2 <port2>] [-rp3 <port3>] [-rp4 <
                port4>]\n"\
            "\t<number> Is the zero based index of the robot in the
                 formation."
            "\t<hostname> Is the network host name of the robot.
                Default is localhost (for the simulator).\n"\
            "\t<port> Is the TCP port number of the local robot.
                Default is 8101.\n"\
            "\t<hostnameN> Is the network host name of the Nth
                robot's server. Default is localhost.\n"\
            "\t<portN> Is the TCP port number of the Nth robot's
                server. Default is 7272.\n");
    Aria::exit(0);
}

//
// Create more objects now that we have obtained the settings from
   the command line.
//

// Setup the reference trajectory thread
ConsensusThread X_r(hosts, ports, IDnumber);
// Starting position of the robot
ArPose robotStart1(-150, 250*(3-IDnumber*2), 0);
```

```
// Position controller
ActionFormation formation(robotStart1, 600*cos((IDnumber*2+1)*M_PI
   /4), 600*sin((IDnumber*2+1)*M_PI/4), &X_r, IDnumber);//
   Formation Tracking

//
// Get things going and connect to the robots
//

// Connect to the robots if this fails exit
ArLog::log(ArLog::Normal, "Connecting to local robot at %s:%d...",
   host, port);
if ((ret = connection.open(host, port)) != 0)
{
    str = connection.getOpenMessage(ret);
    printf("Open failed on local robot: %s\n", str.c_str());
    Aria::exit(1);
}

// add the sonar to the robot
robot.addRangeDevice(&sonar);

// set the device connection on the robot
robot.setDeviceConnection(&connection);

// try to connect, if we fail exit
if (!robot.blockingConnect())
{
    printf("Could not connect to robot 1... exiting\n");
    Aria::exit(1);
}

// turn on the motors and disable the sonar
robot.enableMotors();
robot.comInt(ArCommands::SONAR, 0);

// add actions
robot.addAction(&formation, 55);

// Create a ArKeyHandler and attach it to the robot so the program
   will exit when Escape is pressed
Aria::setKeyHandler(&keyHandler);
robot.attachKeyHandler(&keyHandler);

// Start the consensus calculations running in their own thread
```

```
    X_r.runAsync();

    // Start running the robot in it's own thread
    robot.runAsync(true);

    // Don't end the main program thread until the robot is done
    robot.waitForRunExit();

    // Stop the consensus thread
    X_r.stopRunning();
    do {
        ArUtil::sleep(250);
    } while(X_r.getRunningWithLock());

    // Shut down Aria
    Aria::shutdown();
    return 0;
}

#ifndef ACTIONFORMATION_H
#define ACTIONFORMATION_H

#include "ArAction.h"
#include "ArRobot.h"
#include "ConsensusThread.h"
#include <vector>
#include <fstream>

/*
 * Action that has the robot follow other robots in formation
 *
 * Written By: Joshua Cook
 */
class ActionFormation : public ArAction
{
public:
    // Constructor
    ActionFormation(ArPose& startPose, double formationX, double
       formationY, ConsensusThread* data, int number);
    // Destructor
    virtual ~ActionFormation(void);
    // called by the action resolver to obtain this action's requested
       behavior
    virtual ArActionDesired *fire(ArActionDesired currentDesired);

protected:
```

```
    /* Our current desired action: fire() modifies this object and
       returns
      to the action resolver a pointer to this object.
      This object is kept as a class member so that it persists after
         fire()
      returns (otherwise fire() would have to create a new object each
         invocation,
      but would never be able to delete that object).
    */
    ArActionDesired myDesired;

    ConsensusThread* myData;
    ArPose myStartPose;
    ArTime myTzero;
    int myNumber;
    double myD;
    // Control Variables
    double t, t_minus;
    double t_ref;
    double p_xo, p_yo, theta; // theta in rad
    double p_x, p_y;
    double d_x, d_y;
    double p_xd, p_yd;
    double p_xd_dot, p_yd_dot;
    double p_xd_dot_minus, p_yd_dot_minus;
    double p_xd_minus, p_yd_minus;
    double filter_tau;
    double p_x_minus, p_y_minus;
    double integralError_x, integralError_y;
    double ux, uy;
    double v, omega;
    // Consensus variables
    double p_xc, p_yc, theta_c; // i'th robot's understanding of
       reference trajectory, theta_c in rad
    double p_xc_dot, p_yc_dot, theta_c_dot;
    // Control gains
    double kx, ky, kix, kiy, kdx, kdy;
    bool myStart, myStart2;

    // File object
    std::ofstream myFile;
    // File name
    std::string myFileName;
};
```

```
#endif /*ACTIONFORMATION_H*/

// ActionFormation.cpp
#include "ActionFormation.h"

// Constructor
ActionFormation::ActionFormation(ArPose& startPose, double formationX,
   double formationY, ConsensusThread* data, int number) : ArAction("
   Formation", "Try to have the robot follow a formation")
{
     myStartPose = startPose;
     myData = data;
     d_x = formationX;
     d_y = formationY;
     myNumber = number;

     // Open a file for data collection
     int tmp = myNumber + '0';
     myFileName = "formation";
     myFileName += tmp;
     myFileName += ".csv";//"C:\\Documents and Settings\\Joshua\\My
        Documents\\Data\\" + file;
     printf("Opening file: %s\n", myFileName.c_str());
     myFile.open(myFileName.c_str());

     myFile << " t, t_ref, p_xo, p_yo, theta, p_x, p_y, p_xd, p_yd,
        p_xd_dot, p_yd_dot, ux, uy, v, omega,"
         << " p_xc, p_yc, theta_c, p_xc_dot, p_yc_dot, theta_c_dot" <<
             std::endl;

     // Initialize variables
     myStart = true;
     myStart2 = true;
     myD = 150.; // Distance to front of robot mm
     // Control variables
     t = t_minus = 0.;
     t_ref = 0.;
     p_xo = p_yo = 0.;
     theta = 0.;
     p_x = p_y = 0.;
     p_xd = p_yd = 0.;
     p_xd_dot = p_yd_dot = 0.;
     p_xd_dot_minus = p_yd_dot_minus = 0.;
     p_xd_minus = p_yd_minus = 0.;
     filter_tau = 5.;
```

```
    p_x_minus = p_y_minus = 0.;
    integralError_x = integralError_y = 0.;
    ux = uy = 0.;
    v = omega = 0.;
    // Consensus variables
    p_xc = p_yc = theta_c = 0.;
    p_xc_dot = p_yc_dot = theta_c_dot = 0.;

    // Control Gains
    kx = 0.1;
    ky = 0.1;
    kix = 0.001;
    kiy = 0.001;
    kdx = 0.01;
    kdy = 0.01;
}

// Destructor
ActionFormation::~ActionFormation(void)
{
    printf("Closing file: %s\n",myFileName.c_str());
    myFile.close();
}

// Fire
ArActionDesired *ActionFormation::fire(ArActionDesired currentDesired)
{
    myDesired.reset();

    // Do some initialization the first time through
    if (myStart)
    {
        myTzero.setToNow();
        myRobot->moveTo(myStartPose);
        myStart = false;
    }

    // Get the most current position of the robot
    myRobot->getPose().getPose(&p_xo, &p_yo, &theta);
    theta = theta*M_PI/180;      // convert theta to rad.
    p_x = p_xo + myD*cos(theta);
    p_y = p_yo + myD*sin(theta);

    // Update the local time variable
    t = (double)myTzero.mSecSince()/1000.;
```

```
// Get understanding of reference trajectory from consensus thread
double tmp[STATES];
myData->getX_i(myNumber, tmp);
p_xc = tmp[0];
p_yc = tmp[1];
theta_c = tmp[2];
myData->getX_i_dot(myNumber, tmp);
p_xc_dot = tmp[0];
p_yc_dot = tmp[1];
theta_c_dot = tmp[2];
t_ref = myData->getT();

// Calculate the desired position and velocity for this robot
p_xd = p_xc + d_x*cos(theta_c) - d_y*sin(theta_c);
p_yd = p_yc + d_x*sin(theta_c) + d_y*cos(theta_c);
p_xd_dot = p_xc_dot + theta_c_dot*(-d_x*sin(theta_c) - d_y*cos(
    theta_c));
p_yd_dot = p_yc_dot + theta_c_dot*( d_x*cos(theta_c) - d_y*sin(
    theta_c));

// Do some low pass filtering on the desired velocities
p_xd_dot = p_xd_dot*(t - t_minus)/(filter_tau + (t - t_minus)) +
    p_xd_dot_minus*filter_tau/(filter_tau + (t - t_minus));
p_yd_dot = p_yd_dot*(t - t_minus)/(filter_tau + (t - t_minus)) +
    p_yd_dot_minus*filter_tau/(filter_tau + (t - t_minus));
p_xd = p_xd*(t - t_minus)/(filter_tau + (t - t_minus)) + p_xd_minus
    *filter_tau/(filter_tau + (t - t_minus));
p_yd = p_yd*(t - t_minus)/(filter_tau + (t - t_minus)) + p_yd_minus
    *filter_tau/(filter_tau + (t - t_minus));

// Calculate integral error
integralError_x += (p_x - p_xd)*(t - t_minus);
integralError_y += (p_y - p_yd)*(t - t_minus);

// Command velocities of a point on the front of the robot using
    PID controller
ux = p_xd_dot - kx*(p_x - p_xd) - kix*integralError_x - kdx*( (p_x
    - p_x_minus)/(t - t_minus) - p_xd_dot);
uy = p_yd_dot - ky*(p_y - p_yd) - kiy*integralError_y - kdy*( (p_y
    - p_y_minus)/(t - t_minus) - p_yd_dot);

// Actual control inputs
v = ux*cos(theta) + uy*sin(theta);
omega = 180/M_PI*(-1/myD*ux*sin(theta) + 1/myD*uy*cos(theta));
```

```
    // Save actual and desired positions etc. to file
    myFile << t << "," << t_ref << "," << p_xo << "," << p_yo << "," <<
        theta << ","
        << p_x << "," << p_y << "," << p_xd << "," << p_yd << ","
        << p_xd_dot << "," << p_yd_dot << ","
        << ux << "," << uy << "," << v << "," << omega << ","
        << p_xc << "," << p_yc << "," << theta_c << ","
        << p_xc_dot << "," << p_yc_dot << "," << theta_c_dot << std::
            endl;

    // Do some checking just in case
    if (v > myRobot->getAbsoluteMaxTransVel())
    {
        v = myRobot->getAbsoluteMaxTransVel();
        printf("Exceeded max velocity\n");
    }
    if (omega > myRobot->getAbsoluteMaxRotVel() )
    {
        omega = myRobot->getAbsoluteMaxRotVel();
        printf("Exceeded max rotational velocity\n");
    }

    // Set the desired inputs
    myDesired.setVel(v);
    myDesired.setRotVel(omega);

    // Save off previous values
    p_x_minus = p_x;
    p_y_minus = p_y;
    p_xd_dot_minus = p_xd_dot;
    p_yd_dot_minus = p_yd_dot;
    p_xd_minus = p_xd;
    p_yd_minus = p_yd;
    t_minus = t;

    return &myDesired;
}

#ifndef CONSENSUSTHREAD_H
#define CONSENSUSTHREAD_H

#include "ArASyncTask.h"
#include "ArMutex.h"
#include "ariaUtil.h"
#include "ArNetworking.h"
```

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>

/* ConsensusThread
 * Implements a robust consensus tracking algorithm.
 * Also it calculates the reference trajectory for a robot formation
 * and provides it to the robots.
 */

// Some constants for array sizes
#define NUM_ROBOTS 4
#define STATES 3

class ConsensusThread : public ArASyncTask
{
public:
    // Constructor/Destructor
    ConsensusThread();
    ConsensusThread(std::vector<std::string> hosts, std::vector<int>
       ports, int ID);
    virtual ~ConsensusThread();

    // Functions
    virtual void* runThread(void*);
    void lock();
    void unlock();
    void getX_d(double (&x)[STATES]);
    void getX_d_dot(double (&x_dot)[STATES]);
    void getX_i(int id, double (&x)[STATES]);
    void setX_i(int id, double (&x)[STATES]);
    void getX_i_dot(int i, double (&x)[STATES]);
    double getT();
    double a(int i, int j);
    double b(int i);
    void getState(ArServerClient *client, ArNetPacket *packet);
    void clientGetState(ArNetPacket *packet);

private:
    // Mutex to lock the thread
    ArMutex myMutex;

    // File objects
    std::ofstream myFile;
```

```
    std::ofstream myPacketFile;
    // File names
    std::string myFileName;
    std::string myPacketFileName;

    // Server and Client Objects for network communication
    ArServerBase myServer;
    int myID;
    std::vector<std::string> myHosts;
    std::vector<int> myPorts;
    std::vector<ArClientBase*> myClients;
    ArFunctor2C<ConsensusThread, ArServerClient *, ArNetPacket *>
       myGetStateCB;
    ArFunctor1C<ConsensusThread, ArNetPacket*> myClientUpdateCB;

    // Data
    bool start, myStart2; // Used for initialization
    ArTime myTimeZero;
    double t, t_minus;
    double v_c_d, w_c_d;

    // Consensus variables
    double e_f[STATES]; // error for consensus algorithm
    double e_f0[STATES]; // Beginning error for consensus algorithm
    double fhat[STATES];     // estimate of disturbance force
    double fhat_int[STATES]; // Integral part of estimate
    double u[STATES];    // control input for consensus
    double x_i_dot[STATES];
    double x_c_d[STATES];    // actual reference trajectory, angles in
       rad
    double x_c_d_dot[STATES];
    // Control gains
    double k1, k2[STATES], k3, k4[STATES], kc;  // consensus control
       gains

    double A[NUM_ROBOTS][NUM_ROBOTS];    // Adjacency matrix
    double B[NUM_ROBOTS];                // Vector giving access to
       reference trajectory
    double X[STATES][NUM_ROBOTS];        // Matrix of robot x_i's
};

#endif /*CONSENSUSTHREAD_H*/

// ConsensusThread.cpp

#include "ConsensusThread.h"
```

```
// Constructor/Destructor
ConsensusThread::ConsensusThread()
{
}

ConsensusThread::ConsensusThread(std::vector<std::string> hosts, std::
    vector<int> ports, int ID) :
        myID(ID), myHosts(hosts), myPorts(ports), myGetStateCB(this, &
            ConsensusThread::getState),
        myClientUpdateCB(this, &ConsensusThread::clientGetState)
{

    // Initialize variables
    start = true;
    myStart2 = true;
    t = t_minus = 0.;    // s


    // Reference trajectory parameters
    v_c_d = 3*20;         // mm/s
    w_c_d = 3*M_PI/150; // rad/s

    // Consensus Control Gains
    k1 = .01;
    k2[0] = .7*1000;
    k2[1] = .7*1000;
    k2[2] = .7;
    k3 = 1;
    k4[0] = 0.01;
    k4[1] = 0.01;
    k4[2] = 1;
    kc = 11;

    // Consensus variables
    for (int i = 0; i < STATES; i++)
    {
        e_f[i] = 0.;
        e_f0[i] = 0.;
        fhat[i] = 0.;
        fhat_int[i] = 0.;
        u[i] = 0.;
        x_i_dot[i] = 0.;
        x_c_d[i] = 0.;  // actual reference trajectory, angles in rad
        x_c_d_dot[i] = 0.;
```

```
}

// Initialize matrices
for (int i = 0; i < NUM_ROBOTS; i++)
{
    B[i] = 0.;
    for (int j = 0; j < NUM_ROBOTS; j++)
    {
        A[i][j] = 0.;
    }
}
for (int i = 0; i < STATES; i++)
{
    for (int j = 0; j < NUM_ROBOTS; j++)
    {
        X[i][j] = 0.;
    }
}

// Information-Exchange
B[0] = 1;
B[1] = 0;
B[2] = 0;
B[3] = 0;
A[0][1] = 1;
A[1][0] = 1;
A[1][2] = 1;
A[2][1] = 1;
A[2][3] = 1;
A[3][2] = 1;

// Consensus Initial Conditions
X[0][0] = -1000.; X[0][1] =  1000.; X[0][2] =   500.; X[0][3] =
    -500.;
X[1][0] =  -500.; X[1][1] =   500.; X[1][2] =  1500.; X[1][3] =
    1000.;
X[2][0] =     1.; X[2][1] =   0.75; X[2][2] =    0.5; X[2][3] =
    0.25;

// Open files for data collection
int tmp = myID + '0';
myFileName = "consensus";
myFileName += tmp;
myFileName += ".csv";//"C:\\Documents and Settings\\Joshua\\My
    Documents\\Data\\" + file;
```

```
std::cout << "Opening file: " << myFileName << std::endl;
myFile.open(myFileName.c_str());

myPacketFileName = "packets";
myPacketFileName += tmp;
myPacketFileName += ".csv";
std::cout << "Opening file: " << myPacketFileName << std::endl;
myPacketFile.open(myPacketFileName.c_str());

// Write a headers to the files
myFile << "t, ";
for (int j = 0; j < NUM_ROBOTS; j++)
{
    for (int i = 0; i < STATES; i++)
    {
        myFile << "X[" << i << "][" << j << "], ";
    }
}
for (int i = 0; i < STATES; i++) myFile << "x_c_d[" << i << "], ";
for (int i = 0; i < STATES; i++) myFile << "x_c_d_dot[" << i << "],
    ";
for (int i = 0; i < STATES; i++) myFile << "e_f[" << i << "], ";
for (int i = 0; i < STATES; i++) myFile << "fhat[" << i << "], ";
for (int i = 0; i < STATES; i++) myFile << "u[" << i << "], ";
for (int i = 0; i < STATES; i++) myFile << "x_i_dot[" << i << "], "
   ;
myFile << std::endl;

myPacketFile << "t, n, x, y, theta" << std::endl;

// Start server running
if ( !myServer.open(myPorts[myID],myHosts[myID].c_str()) )
{
    std::cout << "Could not open server." << std:: endl;
}
myServer.addData("getstate", "Returns the consensus state.",
        &myGetStateCB, "none", "Return Description");

myServer.runAsync();

// Fill up the vector with client objects, so I can use their
   position as their ID
for (int i = 0; i < NUM_ROBOTS; i++)
{
    myClients.push_back(new ArClientBase());
```

```
    }

    // Make all of the client connections
    bool connected = false;
    while ( !connected )
    {
        connected = true;
        for (int i = 0; i < NUM_ROBOTS; i++)
        {
            // Only start a client if we receive information from that
                agent.
            if ( a(myID, i) != 0. )
            {
                if ( ((myClients[i])->getState() == ArClientBase::
                    STATE_NO_CONNECTION)
                        || ((myClients[i])->getState() == ArClientBase
                            ::STATE_FAILED_CONNECTION))
                {
                    connected = false;
                    // Attempt to connect the client to the server
                    const char* host = myHosts.at(i).c_str();
                    if ( !(myClients[i])->blockingConnect(host, myPorts
                        [i], true) )
                    {
                        std::cout << "Could not connect to server " <<
                            host << " at port " << myPorts[i] << std::
                            endl;
                    }
                    else
                    {
                        (myClients[i])->addHandler("getstate", &
                            myClientUpdateCB);
                        (myClients[i])->request("getstate", 100);
                    }
                }
            }
        }
    }

}

ConsensusThread::~ConsensusThread()
{
    // Empty the vector of client objects
    for (int i = 0; i < NUM_ROBOTS; i++)
```

```
    {
        delete myClients[i];
    }
    myClients.clear();

    // Close the files
    std::cout << "Closing file: " << myFileName << std::endl;
    myFile.close();
    std::cout << "Closing file: " << myPacketFileName << std::endl;
    myPacketFile.close();
}

// Functions
// Main thread loop
void* ConsensusThread::runThread(void*)
{
    // Run this thread until another thread requests it stop
    while ( this->getRunningWithLock() )
    {
        // lock mutex while we are changing variables
        lock();
        // First time through do initialization
        if ( start )
        {
            myTimeZero.setToNow();
            start = false;
        }
        // Update the local time variable
        t = (double)myTimeZero.mSecSince()/1000.;

        // Calculate actual reference trajectory
        x_c_d[2] = w_c_d*t;
        x_c_d[0] = v_c_d/w_c_d*sin(x_c_d[2]);
        x_c_d[1] = -v_c_d/w_c_d*cos(x_c_d[2]) + v_c_d/w_c_d;
        x_c_d_dot[0] = v_c_d*cos(x_c_d[2]);
        x_c_d_dot[1] = v_c_d*sin(x_c_d[2]);
        x_c_d_dot[2] = w_c_d;

        // Consensus Error
        for (int i = 0; i < STATES; i++) e_f[i] = 0.;
        for (int j = 0; j < NUM_ROBOTS; j++)
        {
            for (int i = 0; i < STATES; i++) e_f[i] += a(myID,j)*(X[i][
                myID] - X[i][j]);
        }
```

```
for (int i = 0; i < STATES; i++) e_f[i] += b(myID)*(X[i][myID]
    - x_c_d[i]);
if (myStart2)
{
    for (int i = 0; i < STATES; i++) e_f0[i] = e_f[i];
    myStart2 = false;
}

// Disturbance estimation
for (int i = 0; i < STATES; i++) fhat_int[i] += (k2[i]*tanh(k4[
    i]*e_f[i]) + k3*e_f[i])*(t-t_minus);//((e_fx>0)-(e_fx<0))
for (int i = 0; i < STATES; i++) fhat[i] = k1*(e_f[i] - e_f0[i
    ]) + fhat_int[i];

// Consensus Control input
for (int i = 0; i < STATES; i++) u[i] = -fhat[i] + b(myID)*
    x_c_d_dot[i] - kc*e_f[i];

// Calculate understanding of reference trajectory (with
    disturbance)
x_i_dot[0] = u[0] + 100*sin((myID+1)/7.*t) + 100*cos(X[0][myID
    ]/1000);
x_i_dot[1] = u[1] + 100*sin((myID+1)/7.*t) + 100*cos(X[1][myID
    ]/1000);
x_i_dot[2] = u[2] + 0.1*sin((myID+1)/7.*t) + 0.1*cos(X[2][myID
    ]);
for (int i = 0; i < STATES; i++) X[i][myID] += x_i_dot[i]*(t -
    t_minus);

// Write data to file
myFile << t << ",";
for (int i = 0; i < NUM_ROBOTS; i++)
{
    for (int j = 0; j < STATES; j++)
        myFile << X[j][i] << ",";
}
for (int i = 0; i < STATES; i++) myFile << x_c_d[i] << ",";
for (int i = 0; i < STATES; i++) myFile << x_c_d_dot[i] << ",";
for (int i = 0; i < STATES; i++) myFile << e_f[i] << ",";
for (int i = 0; i < STATES; i++) myFile << fhat[i] << ",";
for (int i = 0; i < STATES; i++) myFile << u[i] << ",";
for (int i = 0; i < STATES; i++) myFile << x_i_dot[i] << ",";
myFile << std::endl;

unlock();
```

```
        // Run all of the client loops once if they are supposed to be
            connected
        for (int i = 0; i < NUM_ROBOTS; i++)
        {
            // Only run a client if we receive information from that
                agent.
            if ( a(myID, i) != 0. )
            {
                (myClients[i])->loopOnce();
            }
        }

        // Save off previous values
        t_minus = t;
        // Loop keeps running to keep thread alive
        ArUtil::sleep(10); // sleep the thread for 10 ms
    }

    return NULL;
}

// Lock the mutex to prevent multiple threads accessing data
    simultaneously
void ConsensusThread::lock()
{
    myMutex.lock();
}

// Unlock the mutex
void ConsensusThread::unlock()
{
    myMutex.unlock();
}

// Get the actual reference trajectory
void ConsensusThread::getX_d(double (&x)[STATES])
{
    lock();
    for (int i = 0; i < STATES; i++)
    {
        x[i] = x_c_d[i];
    }
    unlock();
    return;
```

```
}

// Get the actual reference trajectory derivative
void ConsensusThread::getX_d_dot(double (&x_dot)[STATES])
{
    lock();
    for (int i = 0; i < STATES; i++)
    {
        x_dot[i] = x_c_d_dot[i];
    }
    unlock();
    return;
}

// Get the local version of the consensus state for agent i
void ConsensusThread::getX_i(int id, double (&x)[STATES])
{
    if ( id < 0 || id >= NUM_ROBOTS )
        return;
    lock();
    for (int i = 0; i < STATES; i++)
    {
        x[i] = X[i][id];
    }
    unlock();
    return;
}

// Set the local version of the consensus state for agent i
void ConsensusThread::setX_i(int id, double (&x)[STATES])
{
    if ( id < 0 || id >= NUM_ROBOTS )
        return;
    lock();
    for (int i = 0; i < STATES; i++)
    {
        X[i][id] = x[i];
    }
    unlock();
    return;
}

// Get the local version of the consensus state derivative for agent i
// Only works for the local agent
void ConsensusThread::getX_i_dot(int i, double (&x)[STATES])
```

```
{
    if ( i < 0 || i >= NUM_ROBOTS )
        return;
    lock();
    for (int i = 0; i < STATES; i++)
    {
        x[i] = x_i_dot[i];
    }
    unlock();
    return;
}

// Get the local time variable
double ConsensusThread::getT()
{
    double ret;
    lock();
    ret = t;
    unlock();
    return ret;
}

// Get element of adjacency matrix
double ConsensusThread::a(int i, int j)
{
    if (( i < 0 || i >= NUM_ROBOTS ) || ( j < 0 || j >= NUM_ROBOTS ))
        return 0.;

    double ret = 0.;

    lock();
    ret = A[i][j];
    unlock();

    return ret;
}

// Get element of the vector B
double ConsensusThread::b(int i)
{
    if ( i < 0 || i >= NUM_ROBOTS )
        return 0.;

    double ret = 0.;
```

```
    lock();
    ret = B[i];
    unlock();

    return ret;
}

// Server packet handling call-back function
void ConsensusThread::getState(ArServerClient *client, ArNetPacket *
  packet)
{
    ArNetPacket sending;

    // lock the thread to prevent simultaneous access
    lock();

    // Load the data into the packet
    sending.byte2ToBuf(myID);
    for (int i = 0; i < STATES; i++)
    {
        sending.doubleToBuf(X[i][myID]);
    }

    // Unlock
    unlock();

    client->sendPacketTcp(&sending);
}

// Client packet handling call-back function
void ConsensusThread::clientGetState(ArNetPacket *packet)
{
    int id;
    double x[STATES] = {0.0};

    // Extract the data from the packet
    id = packet->bufToByte2();
    for (int i = 0; i < STATES; i++)
    {
        x[i] = packet->bufToDouble();
    }

    myPacketFile << getT() << "," << id << ",";
    for (int i = 0; i < STATES; i++)
    {
```

```
        myPacketFile << x[i] << ",";
    }
    myPacketFile << std::endl;

    // Update stored values
    setX_i(id, x);
}
```