Experiments in Distributed Multi-Robot Coordination

Larry Dale Ballard

Utah State University

EXPERIMENTS IN DISTRIBUTED MULTI-ROBOT COORDINATION

by

Larry Dale Ballard

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Electrical Engineering

Approved:

Dr. Wei Ren Major Professor	Dr. YangQuan Chen Committee Member
Dr. Jacob Gunther Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2008

Abstract

Experiments in Distributed Multi-Robot Coordination

by

Larry Dale Ballard, Master of Science

Utah State University, 2008

Major Professor: Dr. Wei Ren
Department: Electrical and Computer Engineering

Consensus control algorithms for multi-agent systems are an area of much research. Several consensus control laws are experimentally validated on a multi-robot testbed in this thesis. A graphical user interface (GUI) is developed that simplifies use of the testbed, as well as allows the execution of the testbed programs to be divided across multiple computers. This not only provides a more powerful computing environment, but also a more realistic communication environment for the testbed. A method for a time-varying or dynamic formation is both proposed and experimentally validated on the testbed. This research also explores a method for dynamic group resizing, i.e. addition or removal of members of the formation. Also, a new control law for synchronized oscillations is validated. Finally, a testbed for multiple cooperative Unmanned Air Vehicles (UAV) is developed for the Procerus UAV.

(102 pages)

To my wife, Sheila, and my children

Acknowledgments

I would like to acknowledge and thank my major professor, Dr. Wei Ren, for his direction and encouragement. I am very grateful for his willingness to give help and suggestions when I needed them. I would also like to acknowledge the other members of my committee, Dr. YangQuan Chen and Dr. Jacob Gunther.

I would also like to acknowledge the influence of all of the members of CSOIS and their constant excitement for research. It was a great motivation to be around and associated with such great minds every day.

I am grateful for my beautiful wife and family who were always interested in my research and work, and who were always patiently behind me in my studies encouraging me to be my best.

Finally, I am especially grateful to my parents who brought me up and taught me to be curious and love learning.

Larry Ballard

Contents

List of Tables

List of Figures

Chapter 1
Introduction

1.1 Motivation

There has been a large amount of research done in the last two decades with respect to autonomous mobile robots. With every increase in available technology bigger and more complex tasks become possible with smaller and more effective systems. As smaller, more reliable, and longer range communication systems became available, it suddenly became not only possible, but very useful, to implement cooperative groups of robots to perform a certain task rather than one monolithic robot. The saying, "The whole is greater than the sum of the parts" has a lot of truth when dealing with autonomous robots. Cost is generally lower for several small robots than it would be for one large robot. Also, a group of robots is capable of many tasks that a single independent robot could never perform. For instance, there are countless useful applications for distributed mobile sensor networks. Also, a group of mobile robots with robotic arms attached to them would be able to perform complex tasks that would be impractical for a single robot to do. Formations of Unmanned Air Vehicles (UAVs) could be used to perform dangerous tasks to improve the chance of success even if some of the UAVs are destroyed.

There has been a large amount of effort directed toward finding efficient and robust methods for controlling these groups of mobile agents [1–4]. Although there has been a great deal of theory developed for multi-agent control, few researchers have gone as far as to test theory with a physical system. Implementation on an physical system often can reveal hidden difficulties and can sometimes lead to unexpected discoveries.

Probably one of the most promising directions for research in this area is consensus-based coordination. It has been shown by many people that consensus can be used to bring groups with limited communication to an agreement on a value of interest [5–7]. Because of its robustness with limited and even time varying communication between agents [8], these cooperative control algorithms could effectively be used in a wide range of applications from mobile robots, UAVs, autonomous underwater vehicles (AUVs), and satellites.

1.2 Contributions

1.2.1 Multi-Robot Testbed

The multi-robot testbed was designed to facilitate the implementation of consensus-based control algorithms on the AmigoBot mobile-robot platform. The testbed also allows the simulation of time varying communication topologies among the robots. In this thesis we expand this multi-robot testbed with a graphical user interface (GUI). The purpose of the GUI is to both simplify and automate the procedure for running experiments or simulations on the testbed. The process is automated by allowing settings such as communication topologies, experiment runtime, and other algorithm specific settings to be stored for repeated use. For simulations, the simulator program, MobileSim, can be opened and the poses of the robots automatically set from the GUI. The GUI also provides a way for the execution of the testbed software to be distributed across multiple computers, thus distributing the processing load, as well as providing a more realistic communication environment where communication between robots is wireless.

1.2.2 Dynamic Formation Control

In exploring the literature on cooperative formation control it appears that the shape or structure of the formation has nearly always been fixed or predefined and time invariant with a fixed number of points or agents [4, 9]. There has been little written with respect to a dynamically changing or time varying virtual structure. The idea of changing the shape of the formation for the purpose of obstacle avoidance has been addressed, but the formations are still predefined [10, 11]. If the structure or formation were able to change dynamically and in a controlled fashion, group object avoidance could be more effective. That is, it would be possible for a wide formation to temporarily form a single file line, or a narrow formation, in order to pass through a very narrow opening, or a very tight formation could expand in order to let the obstacle pass through the formation. This functionality might be useful for a formation of robots in a building where it might be required to pass through narrow doorways and avoid people or other obstacles.

Another new feature that this algorithm provides is the ability to dynamically change the number of agents in the formation. Previously if one or more agents lost communication with the rest of the formation either through a blocked communication path or by being disabled, the rest of the formation would continue as if the others were still there, with no knowledge of their absence. This is because each agent has some specific knowledge of its own position in the formation. That is, each robot in the formation had a specific place in the formation that it would always occupy. If the lost agent were able to re-establish communication, it would be able to continue on the mission as before. If the agent had been destroyed or disabled it would be very difficult to replace because of the need for individual knowledge of position in the formation. In situations where several groups of mobile robots might be sent to the same location to accomplish a mission, a formation that can add or remove agents dynamically would make it possible to shift agents of one group to another group as needed. For example in fighting a fire, several groups could be sent to fight the fire from different positions. If one area were identified as very important, then agents from another group could be transferred to the more critical group. Also in the case that a large formation of agents were sent on a mission, if a part of the formation lost its line of communication with the rest of the group, as long as there existed at least one leader with knowledge of the mission objective in each group, both groups would be able to reform the formation and continue on the mission.

The purpose of this research is to develop and experimentally validate a new method allowing time varying formation parameters in a group of mobile robots allowing the group to more easily avoid or adapt to obstacles in the environment. This research also provides a method for varying the number of agents in a given formation. All of this requires only an undirected connected communication topology.

1.2.3 Coupled Harmonic Oscillation Control

Synchronization phenomena are common in nature. An important avenue of study in synchronization focuses on coupled oscillators. One classical example is the Kuramoto model [12], which assumes full connectivity of the network. Recent works generalize the Kuramoto model to nearest neighbor interaction [13–15]. In the context of multi-agent systems, Paley, Leonard, and Sepulchre [16, 17] study connections between phase models of coupled oscillators and kinematic models of self-propelled particle groups and provide feedback control laws that stabilize symmetric formations of multiple, unit speed particles on closed curves. In papers by Chopra and Spong [18], passivity-based control is studied for the problem of synchronization of multi-agent systems. Related to synchronization are consensus problems in multi-agent systems. Consensus means that a team of agents reaches an agreement on a common value by negotiating with their neighbors (see [6] for recent surveys).

The coupled harmonic oscillators examined by Ren [19] are second-order linear oscillators, which distinguish them from others [13–17]. The coupled harmonic oscillators studied by Ren are also related to the second-order consensus algorithms studied in articles by Ren and Wang [20,21]. However while the consensus equilibrium for state derivatives is a nonzero constant or zero, the states using the coupled harmonic oscillators are synchronized to achieve oscillatory motions. Therefore, the coupled harmonic oscillators can be used to achieve cooperative scanning of an area with multiple robotic vehicles.

The purpose of this research is to experimentally validate the coupled harmonic oscillators presented by Ren [19] using a team of mobile robots, in particular to explore and implement a new decentralized strategy for symmetric formations using the coupled harmonic oscillators. The purpose of this control strategy is for distributed groups of mobile robots to move in a synchronized manner. Both simulation and experimental results are shown. Based on the results of both simulation and experimentation this strategy is an effective method for synchronizing the motion of mobile robots. While distributed cooperative control has been an active area of research, experimental results are rare due to numerous practical challenges (see "Decentralized cooperative control: A multivehicle platform for research in networked embedded systems" [22] and references therein for some existing multi-vehicle testbeds). The experimental results in this research are of significance for real-world applications.

1.2.4 Cooperative UAV Control

UAVs have begun to become a popular area of research as technology makes them a more feasible solution for many problems. As with wheeled or ground robots, multi-UAV formations are much more useful and economical than a single UAV trying to accomplish a similar task. The motivation for this research is the need for a platform with which cooperative control laws can be tested on UAVs. The goal is to build a testbed platform, initially for Procerus UAVs, but that will eventually be expanded to support multiple types of UAVs such as XBow and Paparazzi. The requirements for this research were to build a testbed similar in function as the AmigoBot testbed described in Chapter 2 where control algorithms for multi-UAV formations can be implemented and tested with time varying communication topologies.

1.3 Organization

The remainder of this thesis is organized as follows. In Chapter 2 the process of designing the GUI for the AmigoBot testbed is explained. Also, in this chapter a short tutorial for using the AmigoBot testbed with the GUI is presented. Chapter 3 introduces a new approach for implementing a virtual structure formation with time varying formation parameters. This dynamic formation scheme is implemented on the AmigoBot testbed, and results are given. Chapter 4 shows the experimental validation of the coupled harmonic oscillator control law developed by Ren [19]. Chapter 5 follows the design and building of a multi-UAV testbed, and Chapter 6 gives concluding remarks.

Chapter 2
Aria GUI

2.1 Introduction

This chapter presents the development of the GUI for the AmigoBot mobile robot testbed. Also included in this chapter is an example of the usage of the GUI. The GUI was developed as the next step in building a useful testbed for cooperative control algorithms. The purpose being that it should provide a concise way of creating and storing repeatable experiments. A major motivation for this was also the fact that previously all changes to any experiment had to be done in code, the entire testbed had to be re-compiled and then run from the command line. This process could become very complicated and time consuming.

Another obstacle of the testbed is that previously it was necessary to execute all of the parts of the testbed on a single computer. We desired to be able to spread the work load across multiple computers. Doing this would allow more complex algorithms to be implemented in the future as well as to allow experiments to include more and more mobile robots without exceeding the processing capabilities of a single computer. Previously, because of the limited processing power of our computers, the largest group of robots possible was four; now the limiting factor is the size of the lab and the number of robots available.

2.2 Design Requirements

The first important requirement for the GUI, and one of the main reasons for creating it, was to be able to design, store, and run experiments without having to modify the code and recompile. The reason that the settings were hard coded originally is that it was not reasonable to require that such a large number of settings be entered in the command line.

The settings that need to be modified for different experiments are:

- Path - Several predefined paths are available, such as a circle or figure 8, etc.
- Communication Matrix - Determines the time-varying communication topology between robots.
- Leader Array - Determines both the leaders and followers as well as which control law to use.
- Dynamic Formation Parameters (for dynamic formation experiments) - Determines parameters specific for the dynamic formation controller (Chapter 3).
- Coupled Harmonic Oscillator Parameters (for coupled harmonic oscillator experiments) - Determines parameters specific for the harmonic oscillator controller (Chapter 4).

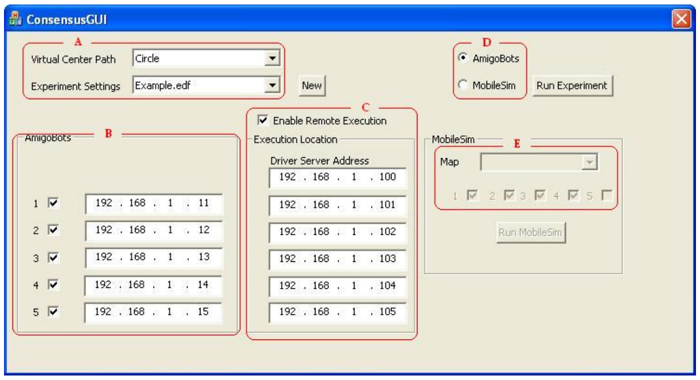
These parameters should be able to be entered in an organized manner so that they can be saved and recalled for use at any time. Gain values were not included in the set of parameters to be stored because they are rarely changed once a good value has been found for them, and including them in the experiment parameters would unnecessarily complicate the GUI.

A second requirement for the GUI design is to allow the user to run either experiments on the AmigoBots or simulations on MobileSim. MobileSim is a powerful graphic simulator for ARIA that is useful for preliminary testing. See Appendix A for more information about ARIA and MobileSim. The GUI should be able to open MobileSim and place the simulated robots in the correct locations for simulation. This not only will save time by programmatically placing the robots in MobileSim, but the robots can also be placed more exactly than by manually moving them with a mouse.

Finally, the last major design requirement was to be able to distribute the processing between multiple computers. There are three distinct executables that comprise the mobile robot testbed. The first is “DriverServer.exe.” The purpose of DriverServer.exe is to supply the desired position or path to the leader or leaders. Only one instance of this program needs to be run. The other two executables are linked to the robot, so one instance of each needs to be running for each robot in the formation. “ClientTest.exe” encapsulates the controller implementation and handles the communication between neighboring robots, as well as the “DriverServer.exe” if that particular robot is a leader. “ServerTest.exe” links to the physical robot and supplies “ClientTest.exe” with feedback from the robot’s sensors. It also sends the control commands from “ClientTest.exe” to the robot. One instance of “ServerTest.exe” and “ClientTest.exe” must be running for each robot. Having all of these programs running on one computer can tax the CPU to the point that one computer is not capable of running the testbed if there are very many robots in the group. Allowing the “ClientTest.exe” and “ServerTest.exe” instances for each robot to be run on different computers would free up processing power to be dedicated to more complex algorithms as well as allowing more robots to be controlled at once. Another benefit of this setup is that it models communication between robots more realistically. Instead of simply passing information from one place to another on a single machine, the data now has to be passed across a wireless network as it would in a more real world situation.

2.3 Graphical Layout

The GUI is written as a dialog based C++ MFC program. The main dialog is shown in fig. 2.1. Section A in fig 2.1 is where the user can select the path and the set of stored parameters for the experiment. The path can be chosen from a list of predefined paths such as a line, a circle or a figure 8. The experiment settings are user defined and will be explained shortly. Section B is where the user can define how many robots will be used in the experiment by selecting them with the corresponding check box. Each AmigoBot has its own IP address for wireless communication. By specifying the IP address of each AmigoBot here, it is possible to select which AmigoBots will be used. By selecting the check box in section C, the location of execution for each robot’s controller and the path planner can be specified by IP address. By deselecting this check box, everything will be run on the local computer. Section D is where the user can decide whether to run the experiment on the



Virtual Center Path: Circle

Experiment Settings: Example.edf

[✓ AmigoBots] MobileSim

[✓] Enable Remote Execution

		AmigoBots	Driver Server Address
			192 . 168 . 1 . 100
1	✓	192 . 168 . 1 . 11	192 . 168 . 1 . 101
2	✓	192 . 168 . 1 . 12	192 . 168 . 1 . 102
3	✓	192 . 168 . 1 . 13	192 . 168 . 1 . 103
4	✓	192 . 168 . 1 . 14	192 . 168 . 1 . 104
5	✓	192 . 168 . 1 . 15	192 . 168 . 1 . 105

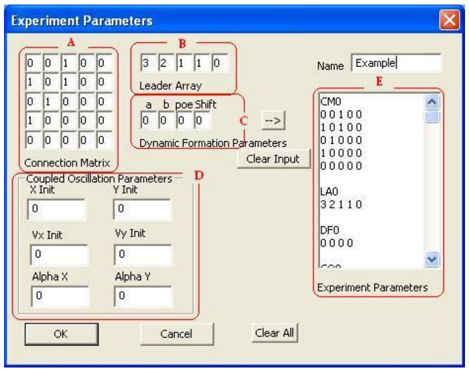
MobileSim: Map

1 ✓ 2 ✓ 3 ✓ 4 ✓ 5

Fig. 2.1: Main dialog for testbed GUI.

AmigoBots or MobileSim. When MobileSim is selected in section D, section E is enabled. Section E works in a way very similar to section B. By clicking on check boxes, robots can be added or removed from the simulation. If desired, MobileSim can be opened with a map that can be selected from the drop down menu. The button "Run MobileSim" will execute Mobilesim with the selected number of robots and map. The button "Run Experiment" begins the experiment for either the AmigoBots or MobileSim depending on which one has been selected. Finally the "New" button opens the dialog in fig. 2.2 for creating a new set of experiment parameters.

The "Experiment Parameters" window shown in fig. 2.2 is used to enter all of the parameters for a desired experiment. The connection matrix is set in section A. Section B is where leader array is set. Section C and D are for parameters specifically for dynamic formation experiments and coupled harmonic oscillator experiments. These are explained in detail in Chapters 3 and 4. Once all of the parameters have been entered into the text boxes, they are added to the experiment by clicking the "–>" button. All of the parameters can be time varying. Up to nine variations can be stored in a single experiment. Each time "–>" is pressed, the current set of parameters in the text boxes are added to the experiment. Section E displays all of the parameters that have been added. When the experiment is run, the testbed will change from one set of parameters to the next every ten seconds.



Connection Matrix

0	0	1	0	0
1	0	1	0	0
0	1	0	0	0
1	0	0	0	0
0	0	0	0	0

Leader Array: 3 2 1 1 0

Dynamic Formation Parameters

a	b	poe	Shift
0	0	0	0

Coupled Oscillation Parameters

X Init: 0	Y Init: 0
Vx Init: 0	Vy Init: 0
Alpha X: 0	Alpha Y: 0

Name: Example

Experiment Parameters

```
CM0
0 0 1 0 0
1 0 1 0 0
0 1 0 0 0
1 0 0 0 0
0 0 0 0 0

LA0
3 2 1 1 0

DF0
0 0 0 0
```

Fig. 2.2: Experiment parameter settings for the GUI.

2.4 Functionality

This section explains the details of the important functionality. We will show how experiment parameter files (edf) are constructed, stored, and loaded. Then we will examine the method used for creating new processes from within the GUI. This will include both loading the testbed executables as well as starting MobileSim. Last we will explain how we are able to execute programs on remote computers from the GUI.

2.4.1 Saving and Loading

The first step in creating a new set of experiment parameters is to fill in all of the fields in fig. 2.2 sections A,B,C, and D. Then clicking the "–>" button will add this set of parameters to the experiment. The parameters are time varying, so the parameters entered in the fields can be changed and added again. Nine sets of parameters must be added to the experiment regardless of whether or not the parameters change. The method for saving the experiment parameters is quite simple. Each time the "–>" button is pressed a section of formatted text is concatenated to the end of a CString variable the holds the entire set of experiment parameters. The format of this text is shown in table 2.1. The four sections shown in table 2.1 are: "Connection Matrix" (CM), "Leader Array" (LA), "Dynamic Formation" (DF), and "Coupled Oscillator" (CO). The # is some integer ranging from 1 to 9.

Once all nine sets of parameters have been added the CString variable is written to a new file with the .edf suffix. It is written in ASCII format so it is possible to modify

Table 2.1: Experiment parameter file (edf) format.

```
CM#
x x x x x
x x x x x
x x x x x
x x x x x
x x x x x

LA#
x x x x x

DF#
x x x x

CO#
x x x x x x
```

these files or even create new files with a text editor. Algorithm 2.1 shows the code that is used to save the experiment data to a file. Lines 1–5 construct the full file name with extension and open the file stream to write to the file. Lines 6 and 7 prepare a character buffer to output the file data to the file. Lines 8–10 write to the file and close it. Once the file is successfully saved, the "Experiment Parameters" window is closed. The experiment settings box from section A of fig. 2.1 contains a list of all the experiment data files that are available and has to be reloaded. The code snippet in algorithm 2.2 shows the method that was used to build a list of all the previously saved experiment data files. Line 2 shows a mask string to search for only files with the suffix ".edf". Lines 3–9 make use of an MFC class, CFileFinder, to obtain a list of files in the current execution directory that match the mask. A loop is then used to enter all of the files found into the dropdown box list, where m_ExpSet is the CComboBox variable for the drop-down box.

2.4.2 Loading Executables

To run a simulation using MobileSim, MobileSim needs to be running before the simulation starts. When the "Run MobileSim" button is pressed the GUI will start MobileSim

Algorithm 2.1 Code for writing the new experiment data to a file.

```
ofstream fout;
CString name = m_Name + _T(".edf");
const char * namebuf = new char[name.GetLength()];
namebuf = (char*)name.GetBuffer(sizeof(namebuf));
fout.open(namebuf,ios::trunc);
wchar_t *buf = new wchar_t[m_ExpmntData.GetLength()];
buf = (wchar_t*)m_ExpmntData.GetBuffer(sizeof(buf));
fout.write((char*)m_ExpmntData.GetBuffer(m_ExpmntData.GetLength()),
    m_ExpmntData.GetLength());
fout.close();
```

Algorithm 2.2 Code for finding and listing all of the .edf files that have been created.

```
CString maskDll, fileName;
maskDll = _T("*.edf");
CFileFind finder;
BOOL looking = finder.FindFile(maskDll);
while(looking){
    looking = finder.FindNextFile();
    fileName = finder.GetFileName();
    m_ExpSet.AddString(fileName);
}
```

with a map, if one is selected, and with as many robots as are indicated by the check boxes in fig. 2.1 section E. The numbers near the check boxes indicate the robot number (not the number of robots). That means that for five robots to be loaded in MobileSim, all five check boxes must be marked, and the robots will have ID numbers 1-5. The same is true when using AmigoBots. The code shown in algorithm 2.3 shows how MobileSim is started from the GUI. Lines 1–5 show the declaration of variables that are not specifically used here, but are necessary for the "CreateProcess" function in line 25. Lines 6–22 creates a CString variable containing the same command line string that would be entered at the command prompt to run MobileSim. This variable is then passed to the "CreateProcess" function in line 25. Then the function "CreateProcess" uses the string to run the executable. The other variables passed in to this function are necessary for the function, but of no use to us so they are not explained here.

When MobileSim is started, each of the robots are by default placed in a vertical line

Algorithm 2.3 Code for finding and listing all of the .edf files that have been created.

```
    UpdateData(TRUE);
    PROCESS_INFORMATION ProcessInfo;
    STARTUPINFO StartupInfo;
    ZeroMemory(&StartupInfo, sizeof(StartupInfo));
    StartupInfo.cb = sizeof StartupInfo;
    CString mobilesim = _T("C:\\Program Files\\MobileRobots\\
        MobileSim\\Mobilesim.exe");
    if(m_CheckSim1.GetCheck())
        mobilesim += " -r amigo-sh";
    if(m_CheckSim2.GetCheck())
        mobilesim += " -r amigo-sh";
    if(m_CheckSim3.GetCheck())
        mobilesim += " -r amigo-sh";
    if(m_CheckSim4.GetCheck())
        mobilesim += " -r amigo-sh";
    if(m_CheckSim5.GetCheck())
        mobilesim += " -r amigo-sh";
    if(m_SelectedMap == "")
        mobilesim += " -nomap";
    else
        mobilesim += _T(" -m \"C:\\Program Files\\MobileRobots\\MobileSim\\")
        + m_SelectedMap + _T("\"");

    CreateProcess(0,(LPSTR)(LPCTSTR)mobilesim,NULL,NULL,FALSE,0,NULL,
        NULL,&StartupInfo,&ProcessInfo);
```

at intervals of 1m. They are not arranged into any specific formation until the program “ServerTest.exe” is started. “ServerTest.exe” arranges the robots by sending a special packet to MobileSim. An example of this packet is shown in algorithm 2.4 where the robots are placed in regular intervals around a circle of radius “RADIUS”. The variables of type ArTypes::Byte4, x, y, and th, on lines 1–4, are used to set the (x, y) position and the orientation angle of the robots. The packet ID 224 on line 5 tells MobileSim that the packet contains position coordinates for a robot. The packet is sent to MobileSim via the connection opened for the robot. In this manner each robot updates its own initial position in MobileSim.

When all of the options and parameters have been correctly chosen, the experiment can be run. There is very little difference in what happens programmatically when a simulation is started for MobileSim as compared to an Experiment for the AmigoBots. The

Algorithm 2.4 Code for setting the position and orientation of the robots in MobileSim.

```
double ang = (teamSize%2 == 1 ? 0 : M_PI/teamSize) − (2*M_PI*i/teamSize);
ArTypes::Byte4 x = ArTypes::Byte4(cos(ang)*RADIUS);
ArTypes::Byte4 y = ArTypes::Byte4(sin(ang)*RADIUS);
ArTypes::Byte4 th = 0;
unsigned char command = 224; // Command for SIM_SET_POSE
ArRobotPacket packet;
packet.empty();
packet.setID(command);
packet.uByteToBuf(0);
packet.byte4ToBuf(x);
packet.byte4ToBuf(y);
packet.byte4ToBuf(th);

packet.finalizePacket();

ArDeviceConnection* conn = robot.getDeviceConnection();
conn->write(packet.getBuf(),packet.getLength());
```

only difference being that for MobileSim the testbed connects the the simulated robots in MobileSim and for the AmigoBots, the testbed connects directly to the AmigoBots. More details about the differences in running simulations as opposed to actual experiments on the AmigoBots are provided in Appendix A.

The “Run Experiment” button causes the GUI to begin loading the testbed. The first of the three testbed programs to be started is “ServerTest.exe”. As stated previously, “ServerTest.exe” provides the interface to the robot, sending motion commands to, and receiving sensor feedback from the robot. “ServerTest.exe” expects a list of three command line parameters. The first parameter “-rh” is used to specify the address of the robot. For MobileSim the address is “localhost” and an additional parameter, “-rrtp”, is required to specify the simulated robot’s port on localhost. In order for “ServerTest.exe” to communicate sensor data back to “ClientTest.exe” as well as receive command data, the instance of “ServerTest.exe” for each robot implements a server that listens on a different port. The next parameter “-sp” indicates which port this particular instance of “ServerTest.exe” will use to listen for “ClientTest.exe” connections. As a side note, this is also how communication between robots is implemented. If communication is allowed between two robots, then “ClientTest.exe” for one robot is able to connect to “ServerTest.exe” of another. The last parameter is “-ts”, which indicates the team size or the total number of robots selected by the check boxes. Algorithm 2.5 shows an example of how an instance of “ServerTest.exe” is executed.

The next program is “DriverServer.exe”. There are only three command line parameters for “DriverServer.exe”. The first, “-sp” is the server port and is used in the same way as it was for “ServerTest.exe”. “DriverServer.exe” acts as a Server that allows the leader robots to connect to it and request path information. The next parameter is “-pfile”. This parameter passes the name of the .edf file to “DriverServer.exe”. The last parameter is “-vcpath”. This parameter specifies which of the predefined paths to use.

The last program to start is “ClientTest.exe”. “ClientTest.exe” is a client program that connects to both “DriverServer.exe” and “ServerTest.exe”. “ClientTest.exe” contains the controller implementation. It takes up to seven parameters. The first two are always required. The first, “-num”, is the robot’s number. This identifies the individual robots. For example, if the second parameter “-ts”, or team size, were set to 5, then “-num” might be an integer between 1 and 5. The last five parameters are only required if remote execution is enabled, in which case, they are “-rl1”,“-rl2”,“-rl3”,“-rl4”,“-rl5”, and represent the IP addresses of remote execution locations for each robot’s “ServerTest.exe” and “ClientTest.exe” instances. “ClientTest.exe” needs to know these addresses so that it is still able to open a connection to “DriverServer.exe” and “ServerTest.exe”.

2.4.3 Remote Execution

The purpose of allowing remote execution is to reduce the amount of computing needs to be performed on one computer. By allowing “DriverServer.exe” and the “ClientTest.exe”, “ServerTest.exe” pair for each robot to be executed on different computers more complex experiments can be run. Also, the communication between robots is more realistic because the communication between robots is forced to go through a wireless network.

Remote execution requires two programs. The first program, a windows service, must be running on each computer that is to host a remote execution. Also, all of the programs and files necessary to run them must be located on the remote computer. The job of this

Algorithm 2.5 Code for starting an instance of ServerTest.exe.

```
m_IPAddress1.GetAddress(b1, b2, b3, b4);
itoa(b1,buffer1,10);
itoa(b2,buffer2,10);
itoa(b3,buffer3,10);
itoa(b4,buffer4,10);
itoa(sp++,spbuf,10);
server = serverBase + _T(" -rh " + buffer1 + "." + buffer2 + "." +
    buffer3 + "." + buffer4 + " -sp " + spbuf + _T(" -ts ") + tsbuf);
if(m_EnableRemoteExec.GetCheck())
    RemoteExec(rl1,server);
else
    CreateProcess(0,(LPSTR)(LPCTSTR)server,NULL,NULL,FALSE,0,NULL,
        NULL,&StartupInfo,&ProcessInfo);
```

service is to listen on a port, and when it receives a message on this port it will execute the command contained in the message. Algorithm 2.6 shows the initialization of the service. On line 2 of this code snippet a socket is opened on port 9987, and line 3 tells the service begins listening. If a connection is attempted the connection is passed to the "ProcConnectionFunc" function in line 16 which is executed in an asynchronous thread so that subsequent connections are not forced to wait for the "ProcConnectionFunc" to finish before they can be handled. "ProcConnectionFunc" is shown in algorithm 2.7. Its purpose is to read data from the socket and create a new process using the data. The data received in the packet is the command line arguments sent to the server from the RemoteExec function implemented in the GUI. The service then starts the program in the same way that MobileSim was started in algorthm. 2.3.

As previously mentioned, the service must be installed on each computer that is to run the remotely executed programs. To install this service, the executable, rexecService.exe, must be copied to any directory on the computer. Then in a command prompt window and in the directory where "rexecService.exe" has been saved, enter the command "*rexecService -i*" at the command prompt as illustrated in fig. 2.3(a). This will install the service. Once the service has been installed it must be configured and started. The service can be configured and started from "Services Manager" (from the Administrative Tools in the Control Panel) by making sure that "Allow service to interact with desktop" has been

Algorithm 2.6 Service initialization.

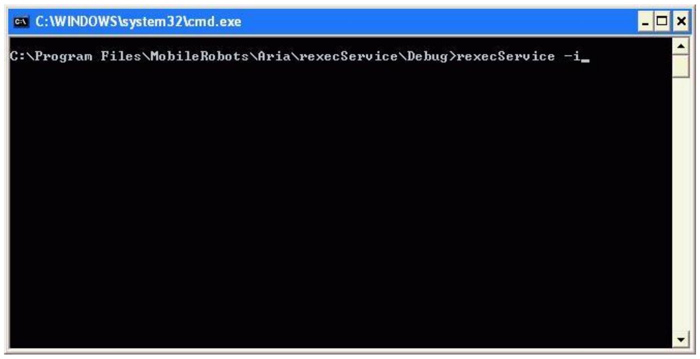
```
CAsyncSocket SrvSock;
SrvSock.Create(9987);
SrvSock.Listen();
CAsyncSocket CliSock;
SOCKADDR addr;
int nSize = sizeof(SOCKADDR);
// Loop to maintain the search while the application is running.
while (!m_pStop->Lock(0))
{
    // Check to see if a connection is being attempted.
    if(SrvSock.Accept(CliSock, &addr, &nSize))
    {
        SOCKET hSock = CliSock.m_hSocket;
        CliSock.Detach();
        //Let a seperate thread handle the individual connections:
        AfxBeginThread(ProcConnectionFunc, (LPVOID)hSock);
    }
    Sleep(1);
}
```

checked as illustrated in fig. 2.3(b). Once this is complete the service must be restarted, and it will be ready to work.

The second program required for remote execution is included in the GUI. To run part of the testbed on another computer the GUI needs to send a packet containing the command line command to port 9987, the port that the service is listening on. This functionality is encapsulated in the “RemoteExec” function that was seen earlier in algorithm 2.5. The “RemoteExec” implementation is shown in algorithm 2.8. Note in line 1 that there are two arguments. The first is the IP address of the computer where the command is to be executed. The second is a string containing the command to be executed. In lines 4–17, the function opens a socket to the given IP address on port 9987. The packet is data is sent to the remote computer in line 20 where CString variable, “cmdline”, contains the command line statement to be executed on the remote computer.

2.5 Usage

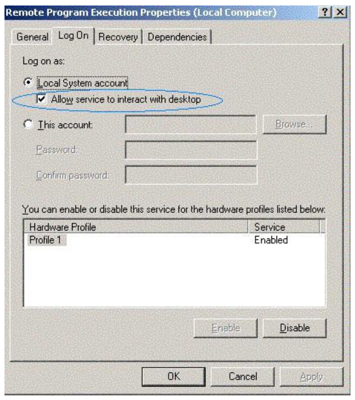
The example in this section is a short tutorial for using the GUI. For this example we will assume that we wish to run a new simulation on MobileSim. The first thing that we



C:\WINDOWS\system32\cmd.exe

```
C:\Program Files\MobileRobots\Aria\rexecService\Debug>rexecService -i_
```

(a) Service installation.



Remote Program Execution Properties (Local Computer)

Log on as:

[✓ Local System account]

[✓ Allow service to interact with desktop]

This account:

Password:

Confirm password:

You can enable or disable this service for the hardware profiles listed below:

Hardware Profile	Service
Profile 1	Enabled

(b) Service setup.

Fig. 2.3: Installation and setup of the remote execution service.

need to do once the GUI has been started is to click on the "New" button to create a new experiment data file. This will open the window shown in fig. 2.2. For this example we will implement a simple time invariant communication topology, the one shown in fig. 2.4. This communication topology can be entered into the Communication Matrix field of the GUI as is shown in fig. 2.5. Also for this example we will let robot 1 be the only leader and the rest of the robots will be followers. Therefore the Leader Array fields can be set as in fig. 2.5. The rest of the fields can be set to 0 for this example. Once all the fields, including the name, have been filled the "->" must be pushed nine times, and the experiment data can be saved by clicking "OK".

For this example we will have our formation follow a circular path so we will select "Circle" from the "Virtual Center Path" drop-down box, and we will choose our "Examples.edf" experiment data file from the "Experiment Settings" drop-down box. as shown in fig. 2.6. By selecting the MobileSim radio button the controls for MobileSim are enabled. Since four robots are selected by default there is no need to change that. The next step is to click the "Run MobileSim" button. Once MobileSim is running and four robots are visible we can start our simulation by clicking "Run Experiment". This example could be run just as easily on the AmigoBots by selecting the "AmigoBots" radio button and entering the correct AmigoBot IP addresses.

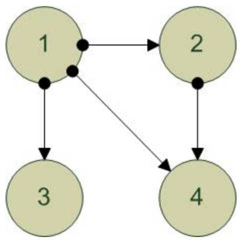
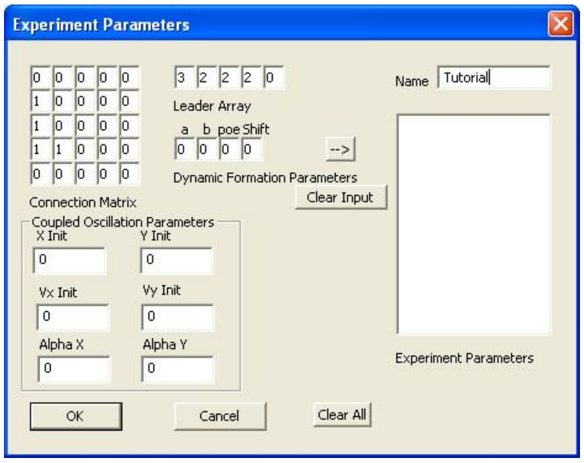


Fig. 2.4: Communication topology used in this example.



Connection Matrix

0	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	1	0	0	0
0	0	0	0	0

Leader Array: 3 2 2 2 0

Dynamic Formation Parameters

a	b	poe	Shift
0	0	0	0

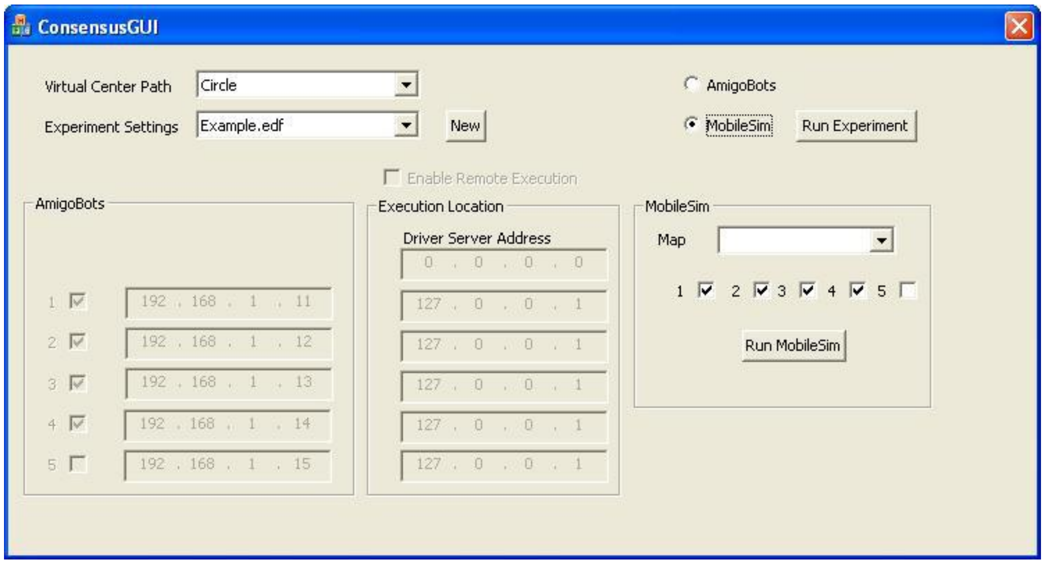
Name: Tutorial

Coupled Oscillation Parameters

X Init: 0	Y Init: 0
Vx Init: 0	Vy Init: 0
Alpha X: 0	Alpha Y: 0

Experiment Parameters

Fig. 2.5: Experiment data for this example.



Virtual Center Path: Circle

Experiment Settings: Example.edf

( ) AmigoBots [✓ MobileSim]

☐ Enable Remote Execution

AmigoBots		Execution Location: Driver Server Address
		0 . 0 . 0 . 0
1 ✓	192 . 168 . 1 . 11	127 . 0 . 0 . 1
2 ✓	192 . 168 . 1 . 12	127 . 0 . 0 . 1
3 ✓	192 . 168 . 1 . 13	127 . 0 . 0 . 1
4 ✓	192 . 168 . 1 . 14	127 . 0 . 0 . 1
5 ☐	192 . 168 . 1 . 15	127 . 0 . 0 . 1

MobileSim

Map:

1 ✓ 2 ✓ 3 ✓ 4 ✓ 5 ☐

Fig. 2.6: Experiment data for this example.

Algorithm 2.7 ProcConnectionFunc implementation.

```
UINT ProcConnectionFunc(LPVOID pParam)
{
    SOCKET hSock = SOCKET(pParam);
    CAsyncSocket CliSock;
  CliSock.Attach(hSock);

    CHAR pInData[1028];

    int nTotalBytes = 0;
    int nRec = 0;
    while (1)
    {
        int nRec = CliSock.Receive(&pInData[nTotalBytes], 1028 - nTotalBytes);
        if (nRec == SOCKET_ERROR || nRec == 0)
            break;
        if (GetApp()->m_pStop->Lock(0))
            break;

        nTotalBytes+=nRec;
    }

    CString strDta(pInData, nTotalBytes);

    STARTUPINFO         StartInfo;
    PROCESS_INFORMATION ProcInfo;

    memset(&StartInfo, 0, sizeof(STARTUPINFO));
    StartInfo.cb = sizeof(STARTUPINFO);

    CString strCmdLine = strDta; //set the command line.

    BOOL fOk = CreateProcess(NULL, strCmdLine.GetBuffer(0), NULL, NULL, FALSE,
        NORMAL_PRIORITY_CLASS, NULL, NULL, &StartInfo, &ProcInfo);

    CliSock.Close();
    return 1;
}
```

Algorithm 2.8 RemoteExec function implementation.

```
int RemoteExec(CString addr, CString cmdline){
    int nRetCode = 0;

    if (!AfxSocketInit())
    {
        return 1;
    }

    CSocket cliSock;
    if(!cliSock.Create())
    {
        return 2;
    }
    if(!cliSock.Connect(addr, 9987))
    {
        return 3;
    }
    //Build up the arguments:

    cliSock.Send(cmdline, cmdline.GetLength());

    return nRetCode;
}
```

Chapter 3

Dynamic Formation Algorithm

3.1 Introduction, Motivations, and Uses

Presented in this chapter is a novel solution for dynamically changing or time varying formation parameters. That is, a decentralized method of handling time varying formation parameters such as the number of agents in the formation and formation shape. By implementing this algorithm a consensus will be reached for the number of agents in the formation as well as for the parameters that control the shape of the formation.

In previous research consensus has been used mainly to achieve a virtual center of the formation. The number of agents as well as the individual positions in the formation remain fixed. If the formation were able to change dynamically and in a controlled fashion, object avoidance could be more effective. That is, it would be possible for a wide formation to temporarily form a single file line in order to pass through a very narrow obstacle, or for a very tight formation to expand in order to let the obstacle pass through the formation. This functionality might be useful for a formation of robots in a building where groups of robots were required to pass through narrow doorways and avoid people or other obstacles.

Another new feature is the ability to dynamically change the number of agents in the formation. Previously if one or more agents lost communication with the rest of the formation either from blocked line of communication or by being disabled, the rest of the formation would continue as if the others were still there, with no knowledge of their absence. If the lost agents were able to re-establish communication, they would be able to continue on the mission as before. If the agent had been destroyed or disabled, the only way to replace it would be with another agent with the previous agent's same knowledge of its position in the formation.

In some situations it might be desirable to send several groups to the same location to accomplish a mission. With a formation that can add or remove agents dynamically, it would then be possible to shift agents of one group to another. For example, in fighting a fire several groups could be sent to fight the fire from different positions. If one area were identified as a very important area, then agents from other groups could be transferred to the critical group.

3.2 Problem Statement

To achieve this consensus-based, time-varying formation, three problems were identified.

1. How to determine the number of agents in the formation.
2. How to determine the shape of the formation.
3. How to organize the agents within the formation.

One of the goals was to allow each agent to decide for itself where it should fit in the formation. This means that all required knowledge of the formation will be common among all agents, and each agent's knowledge of its own unique desired position in the formation will be derived from data shared by its neighbors. The trade off for this ability, and for having a formation where the number of agents is not fixed, is that each agent must be able to find out from its neighbors how many agents are in the formation. Since there is not required to exist a direct line of communication between each agent, it is not possible for the agents to simply count the other agents in the group. A consensus must then be reached by all of the agents for how many are in the group.

The formation should be able to accommodate the addition of new agents as well as their removal in an organized fashion. It should also be able to accommodate a large variety of shapes, with as small an amount of data as possible required to specify the formation. The information specifying the formation shape and size will be shared between neighbors and will then need to be used along with the agent's distinct identifier, such as an IP address or serial number, to determine its position in the formation.

3.3 Algorithm Development

In this section we develop the algorithms necessary for this dynamic formation. Starting with an algorithm for determining how many agents are part of the formation. Also, we develop an algorithm for determining the organization and shape of the formation.

3.3.1 Existence Algorithm

Initially, the idea of letting each agent maintain a list of the other agents seemed somewhat simpler than it actually was. We first explored the idea of having an ID list that would be passed from each agent to its neighbors, where essentially each agent would add its neighbors' IDs to the list and pass the list on. We found that, for adding to the list, this method would be very easy to implement. The problem with this method is with removing an ID from the list. It became evident that this method would be very complicated because of the possibility loops in the graph, and would require that more information be passed between agents than just an ID list to determine if the ID should be kept in, added to, or removed from the list. For instance, as a robot enters into the robot's neighbors would add this robot's ID to the ID list and pass the list on. In this way any robot that is not able to communicate directly with another will still eventually be able to obtain a complete list of the robots in the group. If a robot then leaves the group, the neighbors of this robot will receive an ID list with the lost robot's ID and they will assume that the lost robot is still somehow connected to the group. To solve this problem flag bits would have to be associated with each robot's ID to determine if the ID should be added, kept in, or removed from the list. The logic required to set and reset these flags became very complicated and prone to logic errors because of the possibility of having loops in the communication topology.

Instead of the ID list for the existence consensus we devised a better method that gives each agent a confidence array of the form $C = \begin{bmatrix} c_1 & \cdots & c_n \end{bmatrix}$. If an agent has direct communication with agent j then $c_j = 1$. This represents the idea that the agent knows that agent j exists with no doubt. If there is no connection then the initial value is $c_j = 0$. Each agent in the group will maintain its own confidence array, C, which it will share with its neighbors. During each communication cycle, each agent updates its own confidence array based on the average of its own confidence array with the confidence arrays that it receives from each of its neighbors. Only the values that are less than 1 will be updated since a 1 represents a direct communication link and should only be changed if the communication link is broken. If agent j is part of the group, then the value c_j will grow to 1 over time. After this c_j reaches a defined threshold, the agent is assumed to exist within the group. When a confidence value starts to decrease, this indicates that a agent has been lost or removed from the group. Once the confidence value drops below a lower threshold value, the agent is assumed to no longer be part of the group. When the confidence value begins to decrease however, a 0 has to be added to the average to account for the confidence value of 0 for the lost agent and to drive the average to 0. Otherwise the average can converge to any value between 0 and 1, and cannot be guaranteed to go below the lower threshold value. The only requirement for this algorithm to work is that the communication graph contain an undirected spanning tree. A directed spanning tree is not enough The reason for the undirected requirement is that not even the leader or path planner knows the correct number of agents in the group. Therefore, each member must be able to receive information either directly or indirectly from every other member in the group, and the only way to reasonably ensure this for a changing communication topology is to provide undirected communication between the agents.

Algorithm 3.1 demonstrates the existence algorithm for agent i in pseudo code, where $A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ a_{2,1} & \cdots & a_{2,n} \\ \vdots & \cdots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix}$ is the communication matrix derived from the communication topology, where $a_{i,j} = 1$ if agent i receives data from j and 0 otherwise, and $C^k = \begin{bmatrix} c_1^k & \cdots & c_n^k \end{bmatrix}$ is agent k's confidence array. After the calculation each agent in the group counts the confidence values greater than a set upper threshold and uses that as the number of members in the group. Figure 3.2 shows the confidence values of agents 1, 2, and 4 as the connection to agent 3 changes over time. The communication matrices are shown in (3.1).

Algorithm 3.1 Existence algorithm.

```
If $a_{i,j} = 1 \rightarrow c_j^i = 1$
Else $c_j^i = \frac{\sum_{k=1}^{n} a_{i,k} * c_j^k}{\sum_{k=1}^{n} a_{i,k}}$
If the previous $c_j^i <$ the current $c_j^i \rightarrow c_j^i = \frac{\sum_{k=1}^{n} a_{i,k} * c_j^k}{1 + \sum_{k=1}^{n} a_{i,k}}$
```

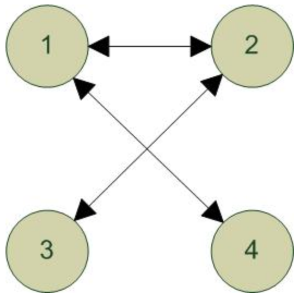
Figure 3.1 shows the progression of the communication graph. From fig. 3.2 we can see that quickly all three agents come to an agreement that agent 3 does, in fact, belong to the group. Then at iteration 30 the link connecting agent 3 to the group is broken. The consensus quickly converges to 0 meaning that all agents agree that agent 3 is no longer part of the group. The non-monotonic nature of the decreasing curve is a result of the fact that the agent that loses the link to agent 3 no longer “knows” whether or not agent 3 exists, so its confidence value drops immediately to 0, then based on the confidence value of the other agents it raises to a non-zero value again which raises the calculated average for the next iteration. This oscillation rapidly decays to 0. At iteration 60 two new links are created to agent 3. As a result the remaining agent that does not have direct link to agent 3 agrees very quickly that agent 3 is back in the group. This setup creates a loop in the connection graph. Loops caused a great deal of difficulty and complication with other algorithms that were tested. This algorithm however had no trouble reaching the correct answer in the presence of loops as is shown in fig. 3.2 from iteration 60 to the end of the simulation. At iteration 90 the loop is broken, but agent 3 is left in the group, and finally at iteration 120 a loop is broken again, this time by removing both links to agent 3, thus removing it from the group. In all of these cases the algorithm behaved as expected. If more speed is required the upper threshold that determines when the confidence level is high enough to consider the agent to be part of the group, and the lower threshold that determines when the confidence level is low enough to determine that the agent is no longer part of the group, can be moved closer to .5. This is similar to adjusting the noise margin in a digital circuit.

$$\begin{aligned} A \;=\; & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \\ \rightarrow & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{aligned} \tag{3.1}$$

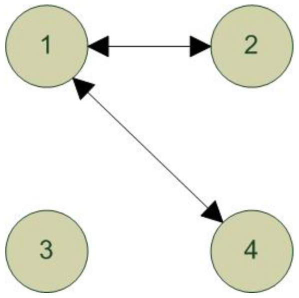
3.3.2 Dynamic Formation Scheme and Agent Organization

Keeping in mind the requirements for the formation algorithm, that there be a minimal amount of data required to specify the formation thereby minimizing the amount of data passed between agents, that there be a large variety of possible formation shapes, and that no unique knowledge, except the agent's own unique ID number, should be required, an ellipse was chosen as the base shape for the formation. An ellipse was chosen as an ideal foundation for the formation because it can be described using only two parameters, the semi-major and semi-minor axis. By specifying these two parameters, any elliptical form from a line to a circle can be specified. By specifying the number of agents to place on the surface of the ellipses, virtually any formation shape can be achieved. Finally by specifying a shift parameter, the formation is able rotate inside of the ellipse, i.e. the formation can be rotated in elliptical coordinates to achieve an even greater flexibility of the formation shape and maneuverability. So there are a total of four parameters required to completely specify the formation.

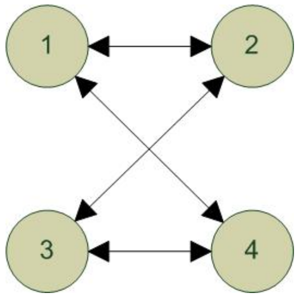
Also, part of this algorithm is the task of determining how and where each agent should be placed in the formation. In order for the shape of the formation and position of



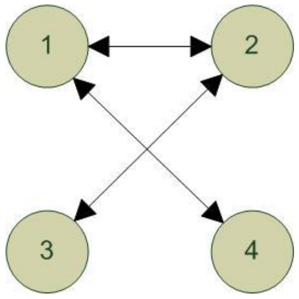
(a) This graph contains an undirected spanning tree where node 3 communicates with only one neighbor.



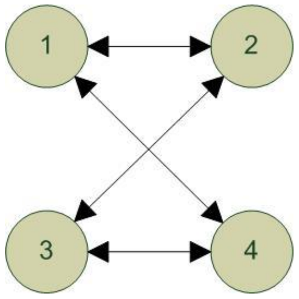
(b) The link to node 3 has been lost at iteration 30.



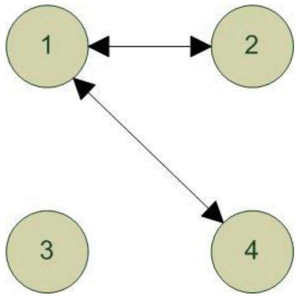
(c) The link to node 3 has been restored and a loop that includes node 3 has been created at iteration 60.



(d) The loop has been lost by removing one of the links to node 3 at iteration 90.



(e) The loop has been recreated by restoring the link to node 3 at iteration 120.



(f) Both links to node 3 have been lost at iteration 150.

Fig. 3.1: Communication graph progression for existence algorithm simulation.

the agents to be more easily calculated, the agents needed to be spaced evenly around the surface of the ellipse independent of the shape of the ellipse. Therefore, agents should be placed at even arc length intervals and not at equal angle intervals. When an odd number of agents are placed on the surface of the ellipse, the first agent will be placed at 0° and the rest spaced at equal distances along the surface of the ellipse. When an even number of agents is placed on the perimeter, then the first agent's place is shifted $\frac{1}{2}$ desired arc length from 0° and the rest are again placed at equal intervals along the surface. The one exception to this rule is when only two agents are placed on the surface. This is to facilitate a single-file configuration. Once all the available positions on the surface of the ellipse are filled, additional agents will be placed on imaginary straight lines that connect consecutive robots on the surface of the ellipse. The extra positions will be created by dividing the

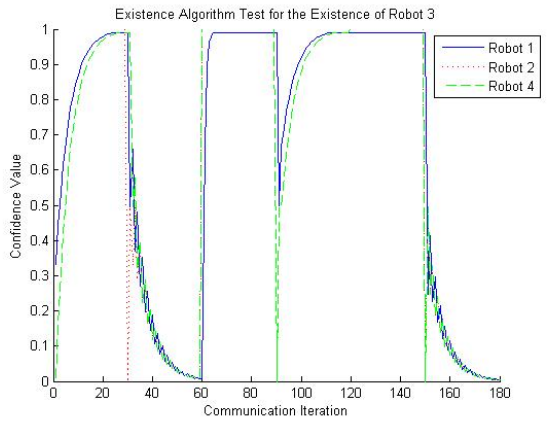


Fig. 3.2: Confidence values.

space on the imaginary lines evenly between the remaining agents. Figure 3.3(a) shows the addition of points in a sequence up to 9 points where 4 points are allowed on the ellipse. Figure 3.3(b) shows a similar sequence where only 3 points are allowed on the ellipse.

The first major obstacle with an elliptical formation is the mathematical complexity of an ellipse. It is necessary to derive a method for calculating the (x, y) position on the ellipse from a given distance around the surface of the ellipse so that each agent will be able to dynamically calculate it's own position in the formation. The ellipse equation is shown in

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1, \tag{3.2}$$

where a is the semi-major axis and b is the semi-minor axis. To calculate the perimeter of an ellipse exactly requires an infinite sum, but a good estimate is found using Ramunujen's Approximation in (3.3).

$$P = \pi \left(3(a+b) - \sqrt{(3a+b)(a+3b)}\right) \tag{3.3}$$

$$\int_j^k dS = \int_j^k \sqrt{1+\dot{y}^2}dx \tag{3.4}$$

With the perimeter known it is a simple matter to calculate the desired arc length between agents. It is also necessary to be able to calculate the end point of the given arc where the desired arc length and starting point are known. As mentioned earlier, the first agent is placed on the x-axis at $0°$ when there is an odd number of positions on the surface of the ellipse. The position of the first agent acts as the starting point for finding the position of the second agent and so on. For even numbers of agents, the starting point is still on the x-axis at $0°$, but the first agent is placed at one half the desired arc length distance along the ellipse from the starting point. From there the second is placed the desired distance from the first, and so on. Arc lengths of an ellipse can be found from the equation in (3.4). This equation requires an integral that cannot be found symbolically, so the integration must be done numerically. Several numeric solving techniques were tested, as well as using the Taylor Series expansion to solve an estimation of the integral symbolically. The Taylor Series expansion of an order high enough to provide sufficient accuracy was too difficult to find. The numerical limit of our computers were reached after finding the 32^{nd} coefficient of the Taylor series. Even with this many coefficients, this method was not nearly accurate enough. Figure 3.4 shows the large error of the Taylor expansion. It also shows that the more circular the ellipse, the more error was produced in this approximation. And finally this method, although it required no loops to calculate, required a large number of multiplications. In choosing a numerical solver, no great speed advantage was found by using any particular numerical solver over another. The Simpson method is used for 0 shown in this thesis.

To solve (3.4) we must first find $\dot{y}$. Solving (3.2) for y results in $y = b\sqrt{1 - \frac{x^2}{a^2}}$, then taking the derivative yields $\dot{y} = \frac{bx}{\sqrt{a^4 - a^2x^2}}$ and can be substituted into (3.4) to get the final arc length equation

$$S = \int_j^k \sqrt{1 + \frac{b^2x^2}{a^4 - a^2x^2}} dx. \tag{3.5}$$

Now it is possible to numerically solve where the limits of integration j and k are the initial and final x-axis values, respectively. This equation will be used to solve for k.

Now that the necessary equations have all been derived, the method for finding the coordinates for points on the ellipse can be explained. We begin by dividing the perimeter of the ellipse into equal segments using Ramanujen's approximation of the perimeter of the ellipse and the number of points that will be placed on the surface of the ellipse. For simplification, the center of the ellipse is always set at the origin with the semi-major axis on the x-axis. When calculating the arc length the first point can be selected arbitrarily by selecting any value for $-a < x < a$, where a is the length of the semi-major axis. Note, however, that (3.5) is infinity when $x = \pm a$, so the in initial x can be chosen to be a very small distance from $-a$. In this discussion the starting point will always be taken to be $-.999a$. The value for y is calculated from the ellipse equation once x is known. The end points of the arc are estimated using a binary search method for finding the upper bound of the integration, k, in (3.5), where $x = .999a$ is used as the initial upper bound. Once the end point is found, it is used as the starting point for the next arc, or the lower limit of integration, j, in (3.5) and the initial upper bound, k, is again set to $x = .999a$ is used as the initial upper limit. The arc length is calculated using these bounds and compared to the desired arc length. After each calculation the upper limit is changed based on the rules of a binary search until the calculated arc length is as close as necessary to the desired arc length. At this point the upper limit is the next x value and the y value can easily be calculated. This process continues until all of the desired points are found.

There may be other faster search techniques, but for the purposes of this thesis a binary search was simple to implement and accomplished the requirements sufficiently well. Using the binary search method it was found that in nearly all cases no more than 15 iterations were necessary to reach a suitable estimate for x.

Since the formation will be changing, it is important to control the rate at which the formation changes to avoid saturation. It would not be desirable, for example, to tell a UAV to go from an altitude of 100 to 300 feet instantly. There are three parameters that can affect the size and shape of the formation. They are: semi-major axis, semi-minor axis, and shift. To find a way limit the rate at which these parameters can change let a represent a parameter. Then $\Delta a = a_f - a_0$, where a_0 is the initial value and a_f is the final value. Also let $\Delta t = t_f - t_0$, where t_0 is the initial time and t_f is the final time, and let $\alpha = \frac{\Delta a}{\Delta t}$, where α is the maximum rate of change and can be chosen for a given parameter. Now the fastest that a can go from $a_0 \rightarrow a_f$ is given by $\Delta t = \frac{\Delta a}{\alpha}$. If we let $t_0 = 0$, then $t_f = \frac{\Delta a}{\alpha}$. Now let T be the sampling period. The value found for t_f must be rounded up to the nearest $n * T$, where $n \in \mathbf{Z}$ and the amount of change, ϵ in a per sampling period, T, is calculated by $\epsilon = \frac{\Delta a}{nT}$. So ϵ is the ideal amount that a will be required to change after each sample period and $a_f = a_0 + \sum_{i=0}^{n} \epsilon$.

3.4 Dynamic Formation Algorithm Implementation for AmigoBot Platform

The implementation of the algorithms on the AmigoBot testbed in this chapter will be explained in some detail in this section. Some background knowledge may be required about the inner workings of the AmigoBot platform, or ARIA. Appendix A provides a general overview of the AmigoBot and ARIA, and Appendix B provides a general overview of our AmigoBot testbed.

The algorithm was placed in “ClientTest.exe”. The job of “ClientTest.exe” is to gather data from the robot’s sensors and from the robot’s neighbors, as well as from the path planner if the particular robot is a leader, then use this data in the control algorithm. This cycle happens every 100 ms. See Appendix B for more detailed information about the testbed itself.

The first thing that must happen in this cycle is that the robot must calculate its confidence array. The confidence array is stored in matrix form with the confidence arrays of each of the robot’s neighbors. The robot must first find out who are its neighbors. For each link that it has with a neighbor, the corresponding confidence value is set to 1. The next step is to sum its confidence values with those of its neighbors. Based on the robots past confidence values and the new confidence values the robot determines if a 0 needs to be added to the average. Recall that if the average is found to be decreasing a 0 must be added to the average to account for a link that has been lost.

The next step is to estimate the formation’s virtual center, and the ellipse parameters. The virtual center is the location of the center of the formation or the (x, y) position that the path planner sends to the leader. Each of the followers in the formation must estimate the parameters given by the path planner to the leaders using the data passed to them by their neighbors. Over time a consensus is reached by all robots in the group. The algorithm for consensus is the same for each of the parameters. Let a_i be a parameter for robot i, then

$$a[k] = a[k-1] + \frac{\sum_{n \in \mathbf{N}_i(t)} da_n}{|\mathbf{N}_i|}, \tag{3.6}$$

where k is the discrete-time index, $\mathbf{N}_i$ is the set of robot i's neighbors, da_n is the change in a_n over time or the derivative of a_n, and $|N_i|$ is the number of neighbors that robot i has.

Once each AmigoBot has an estimate of the virtual center and the ellipse parameters, it uses them to calculate the formation positions of each of the robots. Because of the relatively large amount of processing that must be done to compute the positions, they are only calculated if any of the estimates of the four ellipse parameters have changed. In this process the center of the ellipse is placed at the origin and the robot positions on the surface of the ellipse are calculated first using the ellipse equations derived in this chapter. Once all of the positions on the surface of the ellipse have been found it is an easy matter of placing the rest of the points on straight lines connecting the points on the ellipse. In this step much care had to be taken to make sure that it was never possible to try to place a point exactly on the x-axis, as (3.5) is undefined where $x = a$. We found that reasonable results could be reached by letting $-.999a \leq x \leq .999a$. Care was also taken to find an acceptable accuracy for the binary search. That is, how close to the actual desired arc length did the binary search have to get before it would stop? The trade off was between the number of iterations necessary to reach the final result and the exactness of the positions in the formation. We settled on an accuracy of .01, meaning that the binary search would continue until the end point put the arc length within $\pm.01$ of the desired arc length. This gave very accurate placement on the ellipse and generally took no more than fifteen iterations to finish. Once all of the points are found the the center of the formation is translated to the position of the estimated virtual center, and the formation is rotated to the orientation of the virtual center.

Each of the positions in the formation are found in a sequential order, and therefore each of the robots can compare their own unique ID with the ID, that is linked to the confidence array, of the other robots. In this way the robots are able to place themselves in the same sequential order in the formation without ever being assigned a fixed place in the formation.

3.5 Experiments and Results

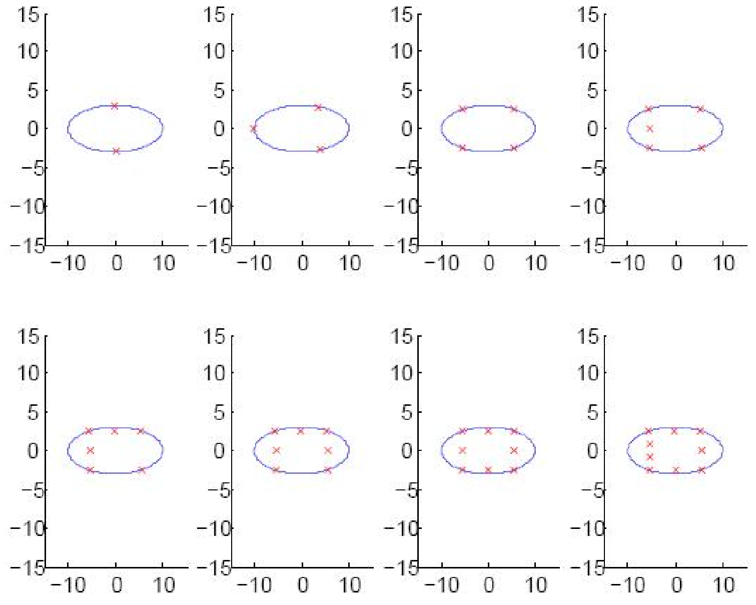
With the implementation of the dynamic formation controller finalized, the GUI is used to run the experiments on the testbed. Figure 3.5 shows an example of a new experiment being set up for the dynamic formation controller. Most of the setup is the same as was described in Chapter 2. The difference is that for these experiments values must be specified for the ellipse parameters a, b, *poe*, and $shift$, where a is the semi-major axis, b is the semi-minor axis, *poe* is the number of positions on the surface of the ellipse, and $shift$ is the rotation of the formation in elliptical coordinates. The $shift$ parameter works by shifting the positions on the ellipse the given arc distance around the surface of the ellipse. The values of a, b, and $shift$ are expected in tenths of meters. The reason for this scaling was to avoid unnecessary complications in programming, and so that it would be possible to allow only numerical values as inputs to the GUI, but still allow values smaller than 1 meter as inputs.

For the experiment shown in fig. 3.6 each of the ellipse parameters are changed in time to demonstrate the capabilities of the dynamic formation algorithm developed in this chapter. The communication topology used for this experiment is that given by the graph shown in fig. 3.7. Initially, all four robots go from their initial positions to form the triangular formation shown in fig. 3.6(a). In this case the ellipse is a circle because both the semi-major and semi-minor axes are equal, and with only three positions allowed on the surface of the ellipse, three of the robots position themselves to form the triangle. The fourth robot then takes its place at the midpoint between two of the robots on the ellipse. Then the path planner sends a new set of ellipse parameters to the leader of the formation. The new parameters place four robots on the surface of the ellipse. The new square formation is shown in fig. 3.6(b). Next the semi-major axis is increased to from 1m to 3m in figs. 3.6(c) and 3.6(d). Note the amount of time this takes to occur. The semi-major axis parameter grows slowly until it reaches the final value so that the robot motors are never saturated. From this rectangular formation the robots then form a single file line by changing the number of positions on the ellipse from four to two as shown in fig. 3.6(e). Finally, the shift parameter is demonstrated in fig. 3.6(f). To see the effects of shift parameter compare figs. 3.6(e) with 3.6(f). Note how the formation has been rotated inside of the ellipse.

The results of the next experiment, shown in fig. 3.8, demonstrate the ability of the dynamic formation algorithm to handle the addition and loss of members of the formation. The addition and removal of a robot from the formation is done simply by changing the communication matrix between the communication matrices shown in fig. 3.9. In this experiment all of the ellipse parameters are all held constant. The experiment begins with only three robots as shown in fig. 3.8(a) using the connection matrix in fig. 3.9(a). The connection matrix is then changed to the one shown in fig. 3.9(b), which includes a fourth robot into the formation as shown in fig. 3.8(b). And finally a robot is removed from the formation by changing the connection matrix to the one shown in fig. 3.9(c) resulting in formation in fig. 3.8(c). Note how the robots are able to reform the formation when robots are added and removed.

3.6 Conclusion

From the work that has been shown in this chapter, the ellipse has been shown to be a workable solution to the problem of dynamically changing formations. Even though the solution requires numeric integration to find the arc lengths of an ellipse, the calculations were still easily able to be completed in time, and did not require very much more processing power than the fixed formation algorithm. However, future work could be done to find a less computationally intensive method for finding the robot positions. Additional future research could be done to implement feedback to the path planner. This would allow the driver to make decisions both for obstacle avoidance as well as path planning to account agents in the formation that are not able to keep up with the rest of the formation. For object avoidance, some direction could be taken from Qu, Wang, and Plaisted [23] to implement object avoidance with a formation of robots.


(a) Adding members to a formation with four members allowed on the ellipse.

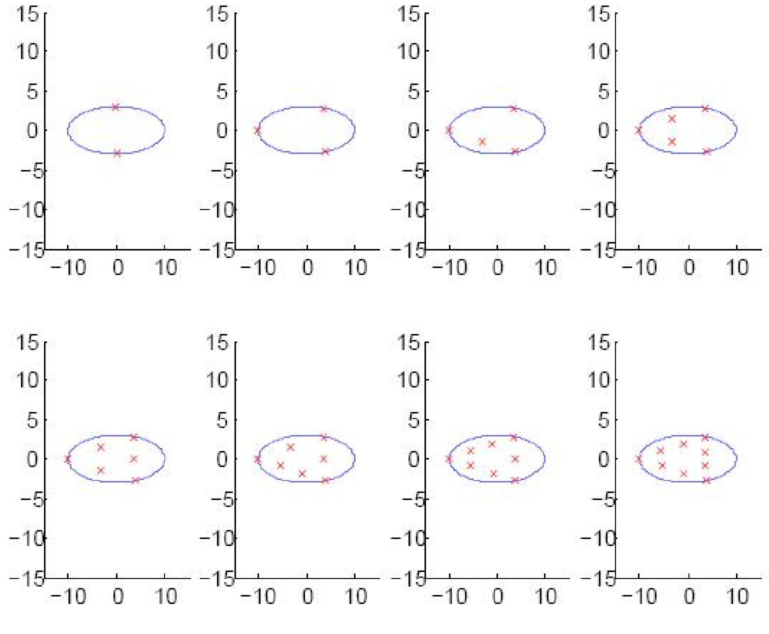

(b) Adding members to a formation with three members allowed on the ellipse.

Fig. 3.3: Demonstration of two different formations using the described method for organizing robots in formation.

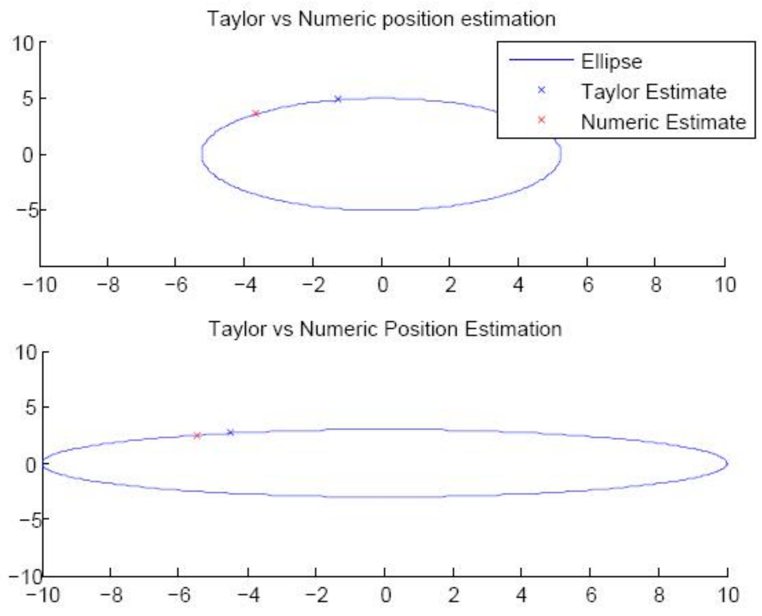
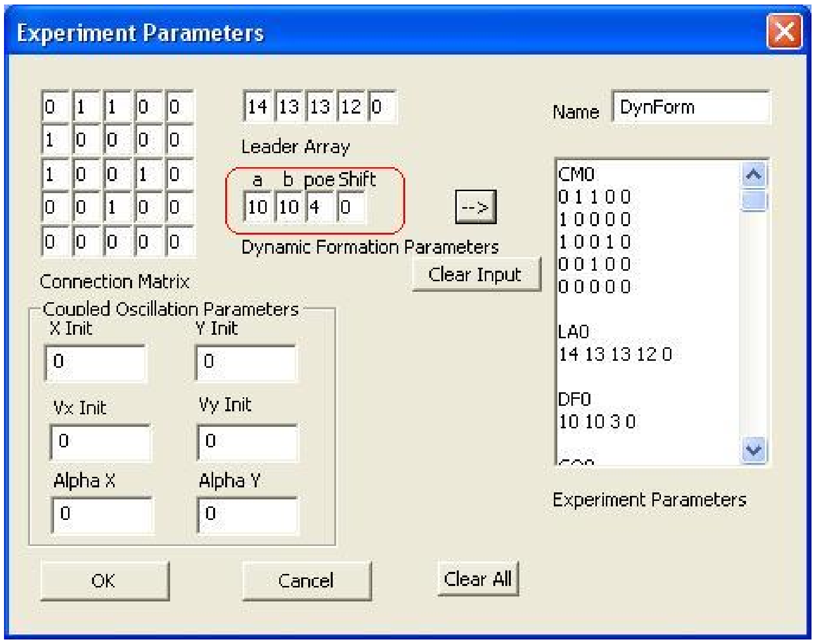


Fig. 3.4: Taylor estimate vs. numeric estimate.



Connection Matrix

0	1	1	0	0
1	0	0	0	0
1	0	0	1	0
0	0	1	0	0
0	0	0	0	0

Leader Array: 14 13 13 12 0

Dynamic Formation Parameters

a	b	poe	Shift
10	10	4	0

Name: DynForm

Coupled Oscillation Parameters

X Init	Y Init
0	0

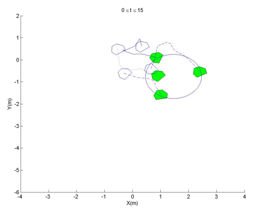
Vx Init	Vy Init
0	0

Alpha X	Alpha Y
0	0

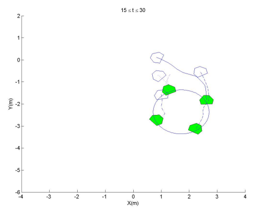
Experiment Parameters

```
CM0
0 1 1 0 0
1 0 0 0 0
1 0 0 1 0
0 0 1 0 0
0 0 0 0 0

LA0
14 13 13 12 0

DF0
10 10 3 0
```

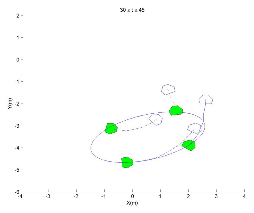
Fig. 3.5: Dynamic formation experiment setup.



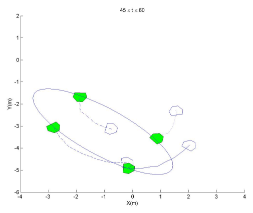
(a) From the starting position four robots go to a triangular formation. $a = 10$, $b = 10$, $poe = 3$, and $shift = 0$.



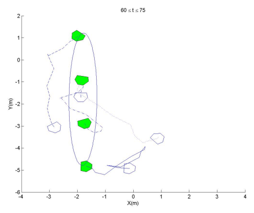
(b) From the triangle formation four robots go to a square formation. $a = 10$, $b = 10$, $poe = 4$, and $shift = 0$.



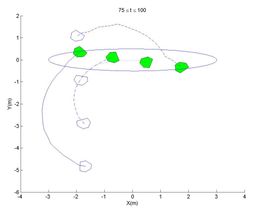
(c) From the square formation four robots begin to for a rectangular formation. $a = 30$, $b = 10$, $poe = 4$, $shift = 0$.



(d) Robots finish forming the rectangular formation $a = 30$, $b = 5$, $poe = 4$, $shift = 0$.



(e) From the rectangular formation the robots form a single file line. $a = 30$, $b = 5$, $poe = 2$, $shift = 0$.



(f) The shift parameter is used to rotate the line inside the ellipse $a = 30$, $b = 5$, $poe = 2$, $shift = 13$.

Fig. 3.6: Dynamically changing formation.

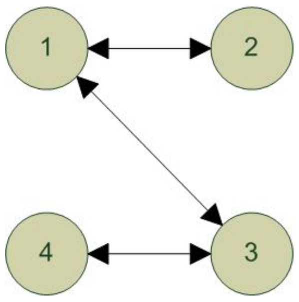
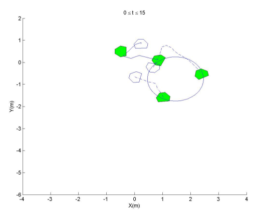
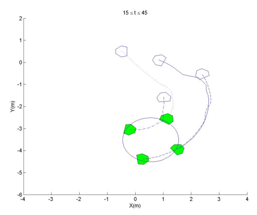


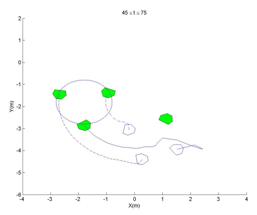
Fig. 3.7: Communication topology graph used in the experiment demonstrating the ellipse parameters.



(a) From the starting position three of the four robots go to a triangular formation. $a = 10$, $b = 10$, $poe = 4$, and $shift = 0$.

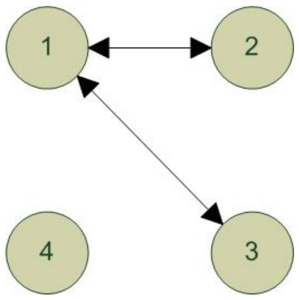


(b) The fourth robot is added to the formation and the robots assume a square formation. $a = 10$, $b = 10$, $poe = 4$, and $shift = 0$.

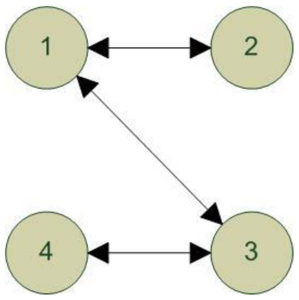


(c) A robot leaves the formation and the remaining three robots assume a triangle formation again. $a = 30$, $b = 10$, $poe = 4$, and $shift = 0$.

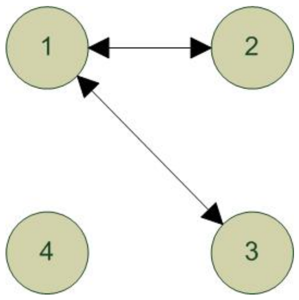
Fig. 3.8: Depictions of the addition and removal of a robot from the formation.



(a) Communication topology graph for first time period.



(b) Communication topology graph for second time period.



(c) Communication topology graph for third time period.

Fig. 3.9: Communication topology graphs used in the experiment demonstrating addition and removal of robots in the formation.

Chapter 4

Harmonic Oscillation Algorithm

4.1 Introduction

In this chapter we show simulation and experimental results for another new distributed control algorithm. The new algorithm presented in this chapter is derived from the coupled harmonic oscillator. In physics a harmonic oscillator is a system that exerts or experiences a restoring force proportional to its displacement when moved from its equilibrium position. This is according to Hooke's law. A coupled harmonic oscillator is a system with multiple harmonic oscillators where each oscillator's motion is coupled with or effected by the motion of the other oscillators. Coupled harmonic oscillators are interesting because in a system of identical harmonic oscillators the frequency and amplitude of oscillation for each of the harmonic oscillators will converge to the same value over time, independent of the initial displacement of any of the oscillators. That is the oscillators naturally reach a group consensus with only the data received from their neighbors defined by the coupling. Our goal is to exploit this natural consensus to create an algorithm for repetitive synchronized movement. Such an algorithm could be used in patrolling or mapping tasks.

In this chapter we will approach the development of the control algorithm by adapting coupled harmonic oscillator equations so that the coupling is defined by the communication topology. Then by expanding this uncontrolled algorithm to a leader follower case so that the final oscillation can be controlled by a path planner and not simply a result of the initial states of the robots. We will then discuss Matlab simulation results followed by the setup and usage of the testbed for our implementation and experiments of the coupled harmonic oscillator controller. Finally, we will show the results of these experiments and our conclusions based on both simulated and experimental results.

4.2 Algorithm Development

We begin this section by applying the coupled harmonic oscillator equations directly to our control problem. Some Matlab simulation is given to demonstrate that the implementation works. Subsequently, we show the expansion of this strategy to a leader follower strategy that provides a means for the parameters of the harmonic oscillation to be exactly controlled by a path planner. Sections 4.2.1 and 4.2.2 are based on a publication by Ren [19].

4.2.1 Coupled Harmonic Oscillators

A coupled harmonic oscillator occurs when two masses are connected by a damper with coefficient b and are each attached to springs with equal spring constants k. The equations describing this situation are

$$m\ddot{x}_1 + kx_1 + b(\dot{x}_1 - \dot{x}_2) = 0, \tag{4.1}$$

$$m\ddot{x}_2 + kx_2 + b(\dot{x}_2 - \dot{x}_1) = 0, \tag{4.2}$$

where $x_i \in \mathbf{R}$ denotes the position of the ith mass. Following from (4.1) and (4.2) we propose n coupled harmonic oscillators with local interaction of the form

$$\ddot{x}_i + \alpha x_i + \beta \sum_{j \in N_i(t)} (\dot{x}_i - \dot{x}_j) = 0, \tag{4.3}$$

where α and β are positive gains, the index $i = 1, ..., n$, and the set $N_i(t)$ is the time varying set of neighbors of the ith oscillator. Let the states of the ith oscillator be represented by ξ_i and ζ_i, where $\xi_i = x_i$ and $\zeta_i = \dot{x}_i$.

Ren [19] shows that if there exists at least one oscillator with a directed path to all other oscillators in the network topology, then (4.3) guarantees that both ξ_i and ζ_i achieve oscillatory trajectories and that both ξ_i and ζ_i are synchronized (i.e. ξ_i approaches ξ_j and ζ_i approaches ζ_j). We use the directed graph $G = (V, \epsilon)$, where $V = \{1, \ldots, n\}$ is the node set and $\epsilon \subseteq V \times V$ is the edge set to model the interaction among the oscillators. Let $\pounds$

denote the Laplacian matrix associated with G. Also, let $p \in \mathbf{R}^n$ satisfy $p \geq 0$, $\mathbf{1}^T p = 1$, where $\mathbf{1}$ denotes the $n \times 1$ column vector of all ones, and $p^T \mathcal{L} = 0$. Ren [19] also shows that for large t

$$\xi_i(t) \to \xi_j(t) \to \cos(\sqrt{\alpha}t)p^T\xi(0) + \frac{1}{\sqrt{\alpha}}\sin(\sqrt{\alpha}t)p^T\zeta(0), \tag{4.4}$$

$$\zeta_i(t) \to \zeta_j(t) \to -\sqrt{\alpha}\sin(\sqrt{\alpha}t)p^T\xi(0) + \cos(\sqrt{\alpha}t)p^T\zeta(0), \tag{4.5}$$

where $\xi(0) = [\xi_1(0), \dots, \xi_n(0)]^T$ and $\zeta(0) = [\zeta_1(0), \dots, \zeta_n(0)]^T$.

4.2.2 Leader Follower Harmonic Oscillators

Let there be a reference oscillator, labeled as oscillator 0, defined by

$$\dot{\xi}_0 = \zeta_0, \tag{4.6}$$

$$\dot{\zeta}_0 = -\alpha\xi_0, \tag{4.7}$$

where α is a positive gain. Solving the differential equations (4.6) and (4.7) it can be shown that

$$\xi_0(t) = \cos(\sqrt{\alpha}t)\xi_0(0) + \frac{1}{\alpha}\sin(\sqrt{\alpha}t)\zeta_0(0), \tag{4.8}$$

$$\zeta_0(t) = -\sqrt{\alpha}\sin(\sqrt{\alpha}t)\xi_0(0) + \cos(\sqrt{\alpha}t)\zeta_0(0). \tag{4.9}$$

Now the coupled harmonic oscillator control equations can be given as

$$\dot{\xi}_i = \zeta_i, \tag{4.10}$$

$$\dot{\zeta}_i = -\alpha\xi_i - \beta \sum_{j \in N_i(t)} (\zeta_i - \zeta_j) - b_{i,0}(\zeta_i - \zeta_0), \tag{4.11}$$

where $b_{i,0}$ is defined as 1 if oscillator i receives information directly from the reference oscillator 0 (i.e. the oscillator is a leader) and $b_{i,0}$ is 0 otherwise. When the leader has a directed path to all the other oscillators, Ren [19] shows that (4.4) and (4.5) can be applied directly to (4.10) and (4.11) to yield $\xi_i(t) \to \xi_0(t) \to \cos(\sqrt{\alpha}t)\xi_0(0) + \frac{1}{\alpha}\sin(\sqrt{\alpha}t)\zeta_0(0)$ and $\zeta_i(t) \to \zeta_0(t) \to -\sqrt{\alpha}\sin(\sqrt{\alpha}t)\xi_0(0) + \cos(\sqrt{\alpha}t)\zeta_0(0)$, for large t.

4.3 Experimentation and Results

We will consider a group of four mobile robots with communication topology depicted in fig. 4.1 throughout the rest of this chapter. An arrow from node j to node i indicates that robot i can receive information from robot j.

Simulations for this coupled harmonic oscillator control strategy were carried out using Matlab/Simulink. Matlab/Simulink simulates a continuous-time implementation. The algorithm was then implemented using the testbed for multi-vehicle cooperative control. By the nature of the testbed, this implementation must be in discrete-time. Therefore, this chapter will show both continuous and discrete-time results.

For the simulations and experiments contained in this paper two values of α are used, namely, $\alpha = .0044$ and $\alpha = .071$.

4.3.1 Matlab Simulation (Continuous-Time)

To model the AmigoBots used in the testbed, we used a model for a two-wheel differential drive robot. The nonlinear state equation that was used in Simulink to model robots is defined as $\begin{bmatrix} \dot{x}_i \\ \dot{y}_i \\ \dot{\theta}_i \end{bmatrix} = \begin{bmatrix} V_i\cos(\theta_i) \\ V_i\sin(\theta_i) \\ \omega_i \end{bmatrix}$, where ($x_i$,$y_i$) is the cartesian position, θ_i is the heading, and V_i and ω_i, respectively, are the translational and rotational velocity

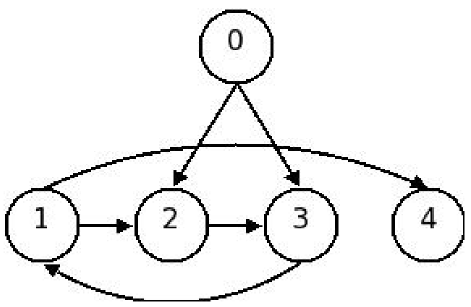


Fig. 4.1: Communication topology.

inputs corresponding to the ith robot. To avoid the nonholonomic constraint, we consider a fixed point off center of the wheel axis denoted by (X_{hi}, Y_{hi}), where $x_{hi} = x_i + L_i \cos(\theta_i)$ and $y_{hi} = y_i + L_i \sin(\theta_i)$, where L_i is the distance from the center of the wheel axis to the hand position, or a fixed control point off the wheel axis. It follows that $\begin{bmatrix} \dot{x}_{hi} \\ \dot{y}_{hi} \end{bmatrix} = \underbrace{\begin{bmatrix} \cos(\theta_i) & -L_i \sin(\theta_i) \\ \sin(\theta_i) & L_i \cos(\theta_i) \end{bmatrix}}_{T} \begin{bmatrix} V_i \\ \omega_i \end{bmatrix}$. Letting,

$$\begin{aligned} \begin{bmatrix} V_i \\ \omega_i \end{bmatrix} &= T^{-1} \begin{bmatrix} U_{xi} \\ U_{yi} \end{bmatrix} \\ &= \begin{bmatrix} \cos(\theta_i) & \sin(\theta_i) \\ -\sin(\theta_i)/L_i & \cos(\theta_i)/L_i \end{bmatrix} \begin{bmatrix} U_{xi} \\ U_{yi} \end{bmatrix}, \end{aligned}$$

it follows that $\dot{x}_{hi} = U_{xi}$ and $\dot{y}_{hi} = U_{yi}$.

The control scheme used in both the simulations and the experiments is illustrated in fig. 4.2. The driver section shown in fig. 4.2 provides the reference oscillator signal and has outputs ζ_{x0}, ξ_{x0}, ζ_{y0}, and ξ_{y0}. The control scheme consists of two parts, the coupled oscillator and the linear feedback controller. Note that corresponding to each robot there exists a coupled oscillator controller and a linear feedback controller. Each coupled oscillator controller guarantees that its oscillator state is synchronized with the state of the reference oscillator, while each linear feedback controller guarantees that the corresponding robot tracks the state of its corresponding oscillator. Let ζ_{xi}, ξ_{xi}, ζ_{yi}, and ξ_{yi} denote the states from the coupled oscillator controller. We use (4.10) and (4.11) to design the coupled oscillator controller. In particular, we let ξ_{*i} and ζ_{*i} [1] satisfy (4.10) and (4.11). We define our linear feedback controller as

$$U_{xi} = \zeta_{xi} - (x_i - \xi_{xi}), \tag{4.12}$$

[1]Here * denotes x or y.

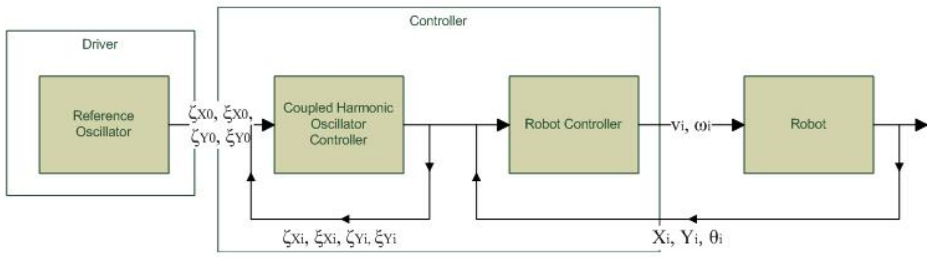


Fig. 4.2: Control scheme consisting of the coupled oscillator controller and the linear feedback controller.

$$U_{yi} = \zeta_{yi} - (y_i - \xi_{yi}). \tag{4.13}$$

Note that the coupled oscillator controller guarantees that $\xi_{*i}(t) \to \xi_{*j}(t) \to \cos(\sqrt{\alpha}t)\xi_{*0}(0) + \frac{1}{\alpha}\sin(\sqrt{\alpha}t)\zeta_{*0}(0)$ and $\zeta_{*i}(t) \to \zeta_{*j}(t) \to -\sqrt{\alpha}\sin(\sqrt{\alpha}t)\xi_{*0}(0) + \cos(\sqrt{\alpha}t)\zeta_{*0}(0)$, also that the linear feedback controller guarantees that $x_{hi}(t) \to \xi_{xi}(t)$ and $y_{hi}(t) \to \xi_{yi}(t)$. Combining these two arguments means that $x_{hi}(t) \to x_{hj}(t) \to \cos(\sqrt{\alpha}t)\xi_{x0}(0) + \frac{1}{\alpha}\sin(\sqrt{\alpha}t)\zeta_{x0}(0)$ and $y_{hi}(t) \to y_{hj}(t) \to \cos(\sqrt{\alpha}t)\xi_{y0}(0) + \frac{1}{\alpha}\sin(\sqrt{\alpha}t)\zeta_{y0}(0)$ for large t. Throughout the simulations and experiments in this chapter $\xi_{x0}(0) = 1.5$, $\xi_{y0}(0) = 0$, $\zeta_{x0}(0) = 0$ and $\zeta_{y0}(0) = .1$ so that the reference path will be a circle of radius 1.5m and the translational velocity of the robot will be .1m/s for large values of t.

The communication topology is given by fig. 4.1. The initial values for the reference oscillator states, $\xi_0(0)$ and $\zeta_0(0)$, are the same that were calculated previously in this section. The initial values for other coupled harmonic oscillators, $\xi_i(0)$ and $\zeta_i(0)$ were chosen arbitrarily to show convergence over time to the reference oscillator states. Figure 4.3 shows the state of each of the coupled oscillators where $j = 0$ is the reference oscillator. In fig. 4.3 ζ converges very quickly, and because of this ξ converges very slowly. Both states should ideally converge at similar rates. In this case the value for β used in (4.11) was 1. The effects of β on the speed of convergence are illustrated in fig. 4.4. This figure shows that while ζ converges slower, ξ is able to converge faster and as a result the system converges faster. A reasonable value for β seems to be $\beta = 10\alpha$. The gain α also effects the the rate at which the states will converge. A comparison of fig. 4.5 and fig. 4.3 shows that

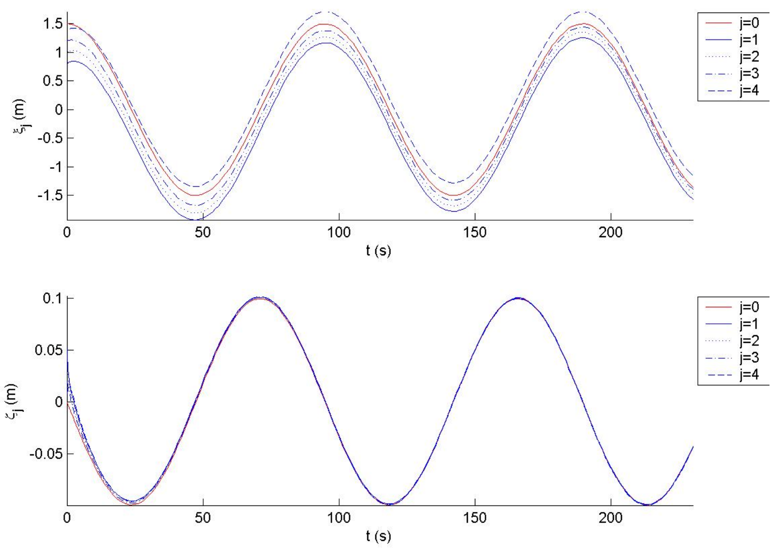


Fig. 4.3: Matlab simulation with $\alpha = .0044$ and $\beta = 1$.

for a larger α ($\alpha = .071$) the states shown in fig. 4.5 converge long before those of fig. 4.3. Finally letting $\beta = 10\alpha$ for this second case, we can see by comparing fig. 4.5 and fig. 4.6 that the states are able to converge faster.

Adding the robot model into the simulation for the case where $\alpha = .0044$ and where $\beta = 10\alpha$, we are able to see that the local controller is able to track the input from the harmonic oscillator controller. Figure 4.7 shows the trajectories and snapshots of the four robots at different time periods and fig. 4.8 shows the complete trajectories of the four robots with snapshots of the robots. Here we have introduced a deviation to both X and Y components of the states of the oscillators. Note that the motion of the four robots becomes synchronized as they move around their circular paths. Figures 4.9 and 4.10 show the x and y axis positions and the velocities versus time of the four robots together with the reference. Note that all of the four robots' states synchronize with the reference states.

4.3.2 AmigoBot Experiment (Discrete-Time)

In order to be able to change the oscillation parameters easily for any given experiment, the GUI includes the section highlighted in fig. 4.11. These fields represent the initial

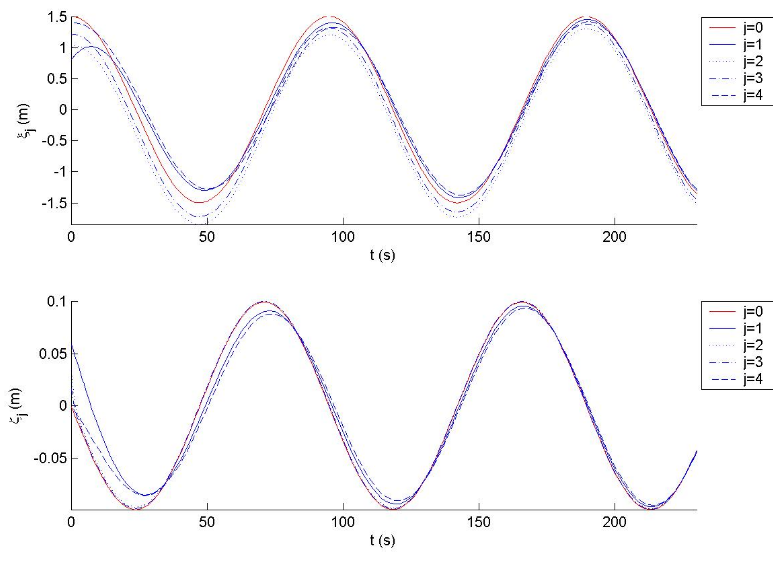


Fig. 4.4: Matlab simulation with $\alpha = .0044$ and $\beta = 10\alpha < 1$.

conditions of the reference oscillator. The x Init and y Init fields correspond to $\xi_{x0}(0)$ and $\xi_{y0}(0)$, Vx Init and Vy Init correspond to $\zeta_{x0}(0)$ and $\zeta_{y0}(0)$, and Alpha X and Alpha Y both correspond to α. A distinction was made between the x and y directions for α for added generality.

As we mentioned previously, the testbed is an implementation of a discrete-time distributed networked control system. Since the network delay is much less than the nominal 100ms sample time of the ActivMedia AmigoBots we know the system delay is close to 100ms. To remove this instability the reference oscillator was implemented as $\xi_{*0}[k] = \cos(\sqrt{\alpha}kT)\xi_{*0}[0] + \frac{1}{\alpha}\sin(\sqrt{\alpha}kT)\zeta_{*0}[0]$ and $\zeta_{*0}[k] = -\sqrt{\alpha}\sin(\sqrt{\alpha}kT)\xi_{*0}[0] + \cos(\sqrt{\alpha}kT)\zeta_{*0}[0]$, where k denotes the discrete-time index and T denotes the sampling period. Due to the discrete-time nature of the system, the controller was implemented using zero order hold as

$$\xi_i\,[k+1] = \xi_i\,[k] + T\zeta_i\,[k] + \frac{1}{2}T^2 u_i\,[k]\,, \tag{4.14}$$

$$\zeta_i\,[k+1] = \zeta_i\,[k] + Tu_i\,[k]\,, \tag{4.15}$$

where $\dot{\zeta}_{*i}[k]$ is the kth sample of $\dot{\zeta}_{*i}(t)$ in (4.11).

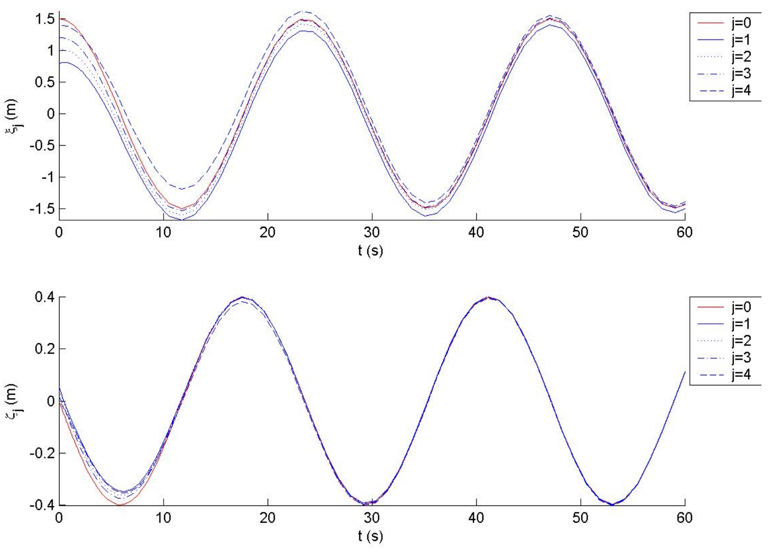


Fig. 4.5: Matlab simulation with $\alpha = .071$ and $\beta = 1$.

Our experiments show that the control scheme works just as well in discrete-time as it does in continuous-time. Figure 4.12 shows the trajectories and snapshots of the four robots over four time periods and fig. 4.13 shows the complete trajectories of the four robots. These two figures can be compared to figs. 4.7 and 4.8, respectively. Note that in both cases the snapshots of the robots become aligned over time.

Figures 4.14 and 4.15 show the x and y axis positions and velocities with respect to time of the four robots and the reference. See how each of the robots converge to the reference position and velocity. These plots can also be compared to figs. 4.9 and 4.10.

4.4 Conclusion

In this chapter we have proposed and implemented a distributed strategy for the synchronization of the motions of multiple mobile robots. The results of our simulations and experiments have shown that we are able to control the shape and speed of the path. Our results also show robustness to discretized implementations. This research also shows that using coupled harmonic oscillators is an effective solution to synchronized motion useful for many repetitive and patrol applications.

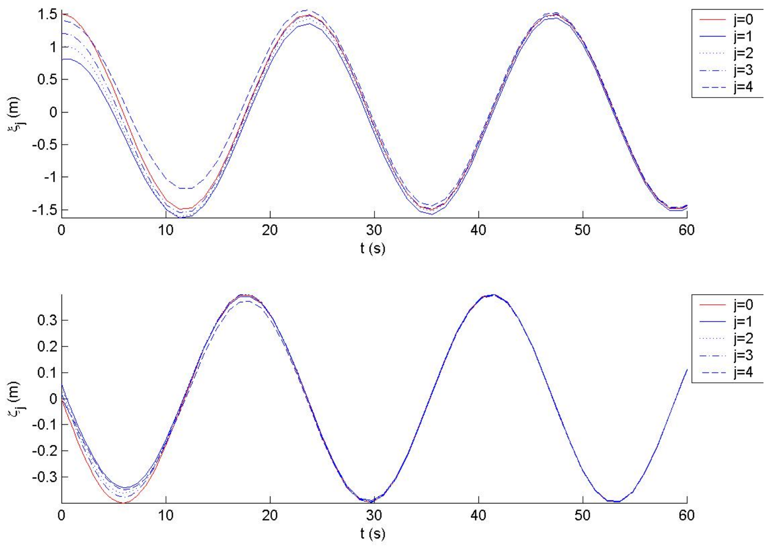


Fig. 4.6: Matlab simulation with $\alpha = .071$ and $\beta = 10\alpha$.

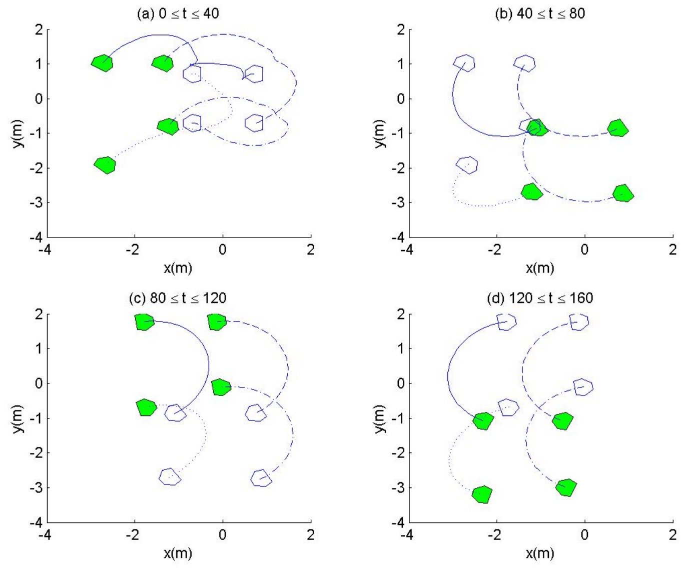


Fig. 4.7: Matlab simulation: Robot paths and snapshots for four consecutive time periods of 40 second intervals starting at 0 s and ending at 160 s. The white snapshot indicates the starting positions.

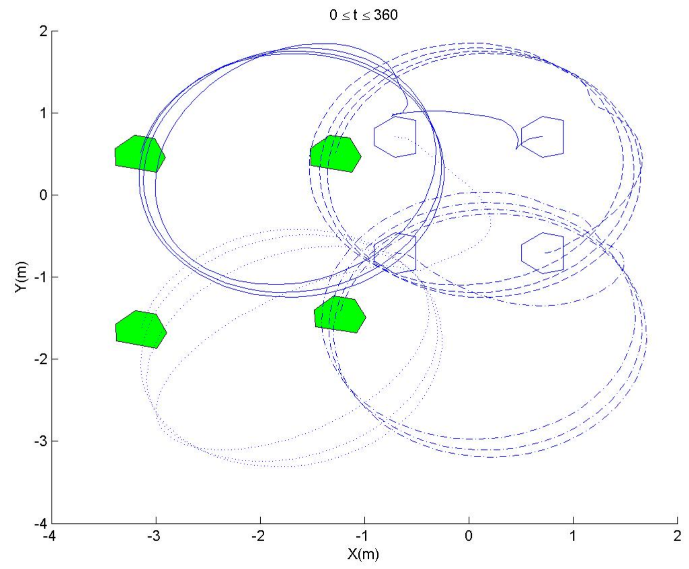


Fig. 4.8: Matlab simulation: Robot paths for an entire simulation of 360 second with snapshots at the beginning and end. The white snapshot indicates the starting positions.

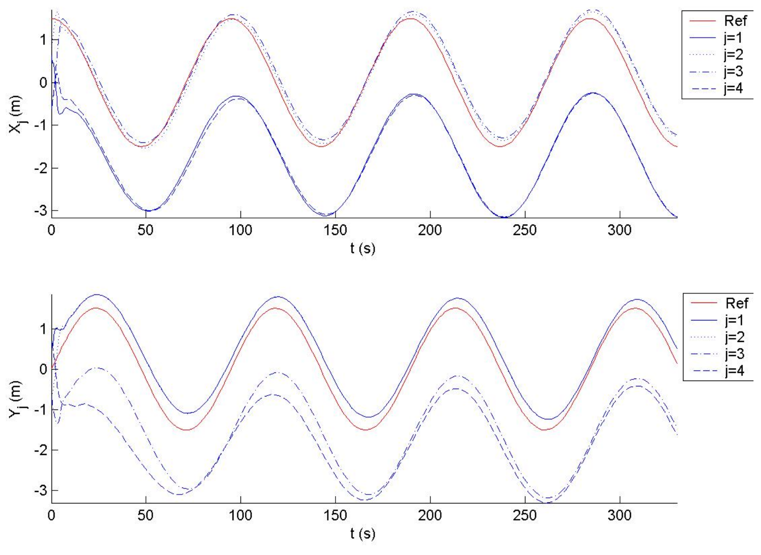


Fig. 4.9: Matlab simulation: Robot (x, y) position vs. time.

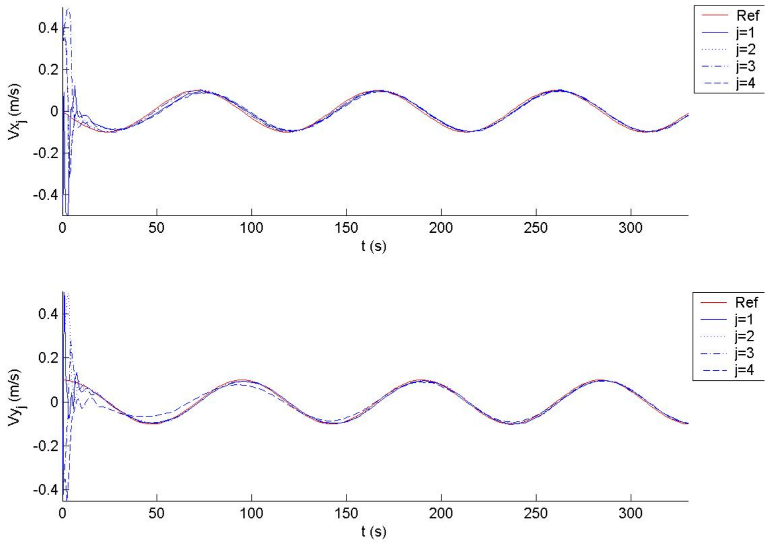
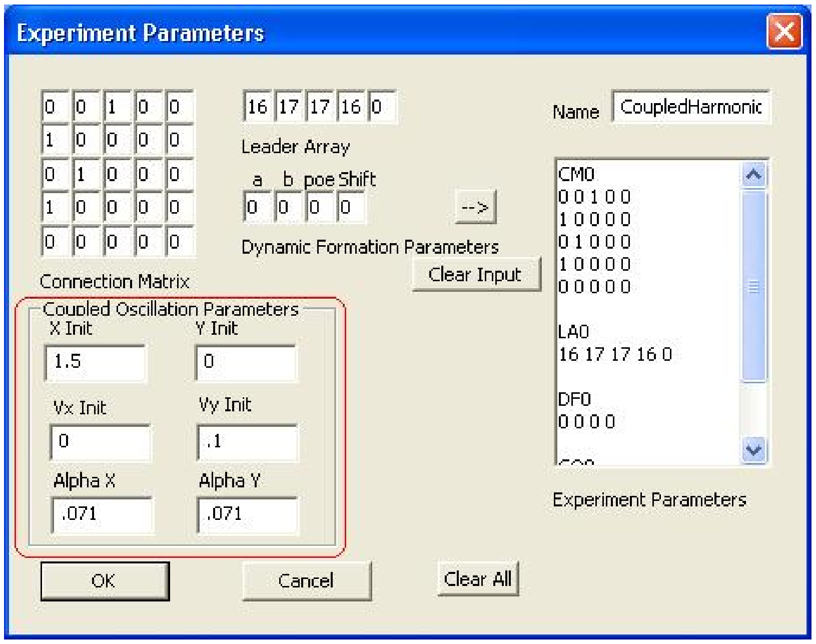


Fig. 4.10: Matlab simulation: Robot velocities in the (x, y) directions vs. time.



Experiment Parameters

0	0	1	0	0
1	0	0	0	0
0	1	0	0	0
1	0	0	0	0
0	0	0	0	0

Connection Matrix

16 17 17 16 0

Leader Array

a	b	poe	Shift
0	0	0	0

Dynamic Formation Parameters

Name: CoupledHarmonic

```
CM0
0 0 1 0 0
1 0 0 0 0
0 1 0 0 0
1 0 0 0 0
0 0 0 0 0

LA0
16 17 17 16 0

DF0
0 0 0 0
```

Experiment Parameters

Coupled Oscillation Parameters

X Init	Y Init
1.5	0

Vx Init	Vy Init
0	.1

Alpha X	Alpha Y
.071	.071

Fig. 4.11: Shows the usage of the GUI for the coupled harmonic oscillator controller.

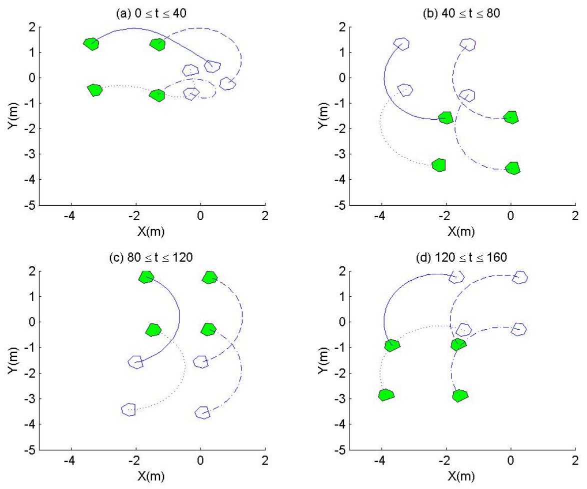


Fig. 4.12: AmigoBot experiment: Robot paths and snapshots for four consecutive time periods of 40 second intervals. The white snapshot indicates the starting positions.

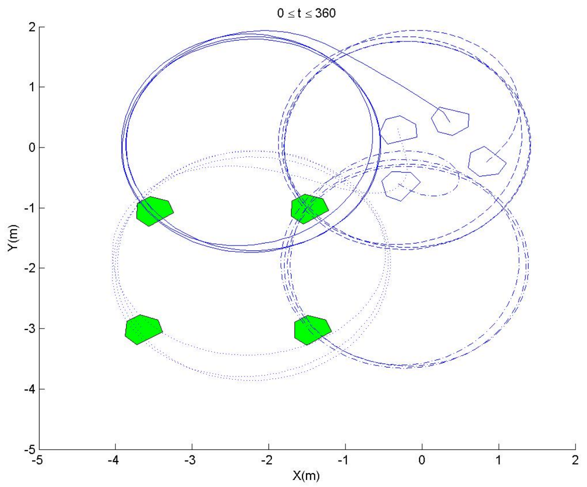


Fig. 4.13: AmigoBot experiment: Robot paths for an entire simulation of 360 second with snapshots at the beginning and end. The white snapshot indicates the starting positions.

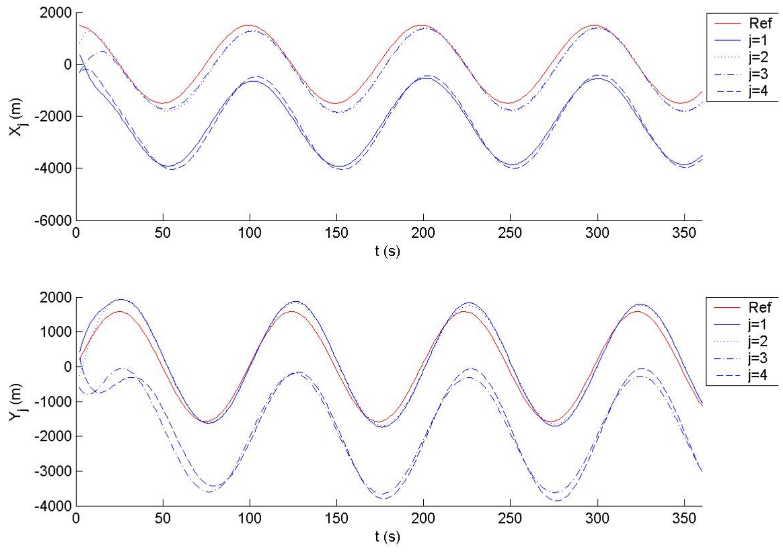


Fig. 4.14: AmigoBot experiment: Robot (x, y) position vs. time.

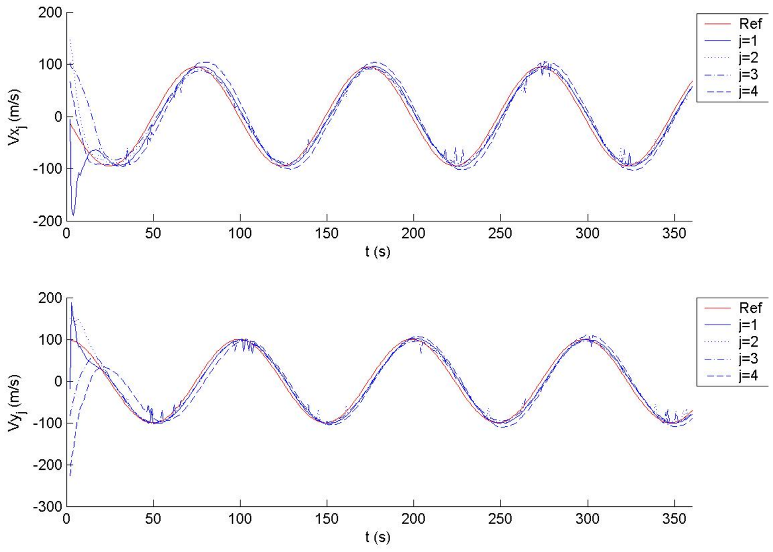


Fig. 4.15: AmigoBot experiment: Robot velocities in the (x, y) directions vs. time.

Chapter 5
Procerus UAV Formation

5.1 Introduction

As batteries have become lighter and their capacity has increased, and all other electronic components have grown ever smaller and lighter, electric unmanned aerial vehicles (UAVs) have entered the spotlight. And with so much work being done on the cooperative control of ground robots, the next logical step is to apply this knowledge to multiple UAVs. There are many uses for formations of UAVs, from atmospheric sampling where multiple UAVs could take air samples or pollution measurements from the same latitude and longitude at different altitudes to achieve a more accurate picture of the pollution distribution in the atmosphere, to extremely hazardous missions where the loss of one or more UAVs is possible. This chapter presents work that has been done in constructing a testbed for coordinated control of UAVs.

The testbed in this chapter is developed for two Procerus UAVs with the Kestrel autopilot with the goal of being able to expand it in the future to include more UAVs and to incorporate several other types of autopilots, such as the Xbow and Paparazzi. This chapter is organized as follows. First we will give an overview of the Procerus UAV platform, then we discuss the design requirements for the testbed. From there we will go through the GUI layout and design its functionality. Lastly, we will give our conclusions.

5.2 Procerus UAV Platform Overview

The Procerus UAVs were selected as the initial platform for the testbed because they can be purchased ready to fly. Little effort is required in the initial set-up and calibration of the UAVs. They are also quite rugged and durable. The platform introduction presented in this section is based on Procerus manuals [24].

5.2.1 Airframe, Sensors, and Actuators

The Procerus airframe is a delta wing made from an EPP foam as shown in fig. 5.1. The UAVs are propelled by very efficient brushless DC motors, and are powered by three LiPo batteries. Two small servo motors control the ailerons. Sensors include two pressure sensors, one to determine the air-speed, and the other to determine the altitude, GPS, 3-axis gyro and accelerometer, and a dipole antenna for communication. The Kestrel autopilot is the on-board electronics that handles the low-level control of the UAV.

5.2.2 CommBox

Communication with the UAV is handled by the CommBox shown in fig. 5.2. The CommBox is the communications gateway between the Virtual Cockpit, RC transmitter, and multiple Kestrel Autopilots. The virtual cockpit communicates through the CommBox via the CommBox's RS-232 serial port. The CommBox parses the packets of data sent by the virtual cockpit and passes them on to the address specified by the packet.

An RC transmitter can be connected to the CommBox to allow the user to override the control of the Kestrel autopilot. The RC transmitter can control only one UAV at a time, but by changing the RC destination address, any UAV can be controlled at any time. Signals from the RC Transmitter are sent to the UAV at a rate of 13 Hz.

Finally, the autopilot on-board the UAVs transmits and receives communication to and from the CommBox via the Aerocomm modem. Each packet the CommBox receives is checked for errors. If no errors are found, the packet is passed on to the virtual cockpit through the serial connection.

5.2.3 Autopilot

The Kestrel autopilot, shown in fig. 5.3, is a piece of hardware that resides on-board the UAV and processes all of the data from the sensors and communication with the Virtual Cockpit and CommBox. It also handles all of the low-level control of the UAV's actuators,

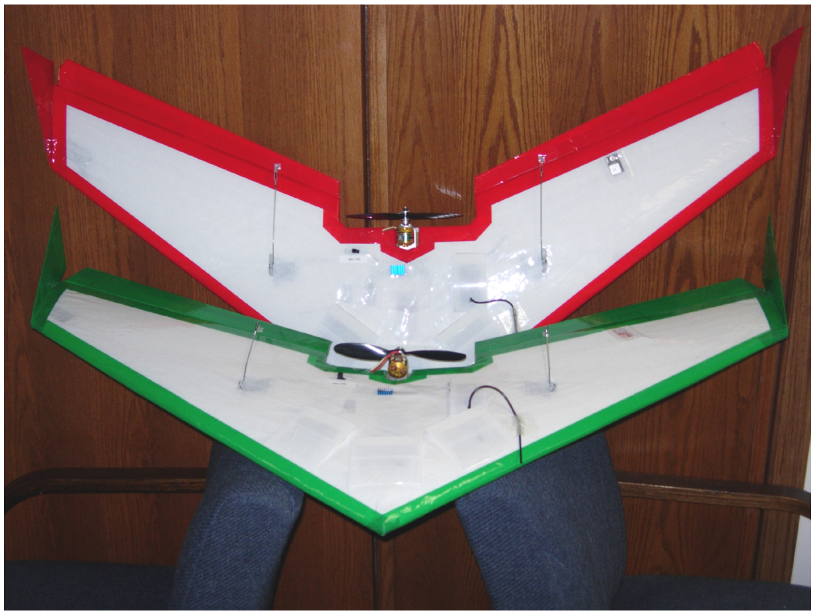


Fig. 5.1: Procerus UAVs.

Fig. 5.2: CommBox.

such as propeller speed control and aileron angle. Mid-level control is also handled by the autopilot. This mid-level control includes numerous modes, such as, take-off, land, loiter, and navigation modes. The navigation mode is the mode that we will be using in this chapter to implement our high-level control algorithms. In navigation mode the autopilot flies the UAV toward waypoints that are sent from the virtual cockpit. Our control algorithms will generate these waypoints based on the positions of other UAVs. The waypoints are then sent to the appropriate UAVs.

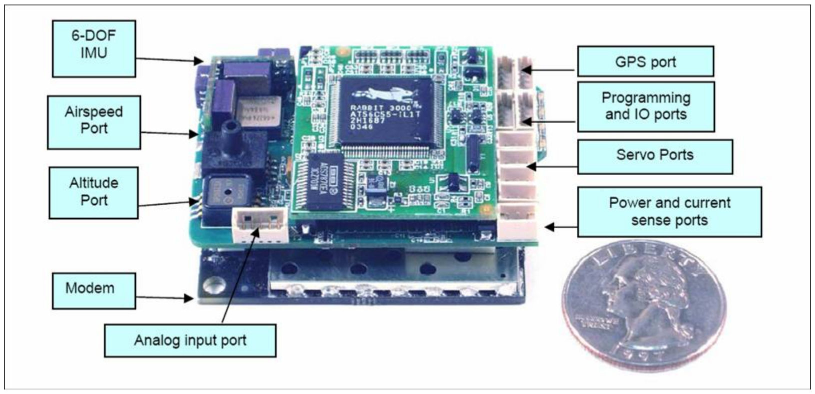


Fig. 5.3: Kestrel autopilot.

5.2.4 Virtual Cockpit

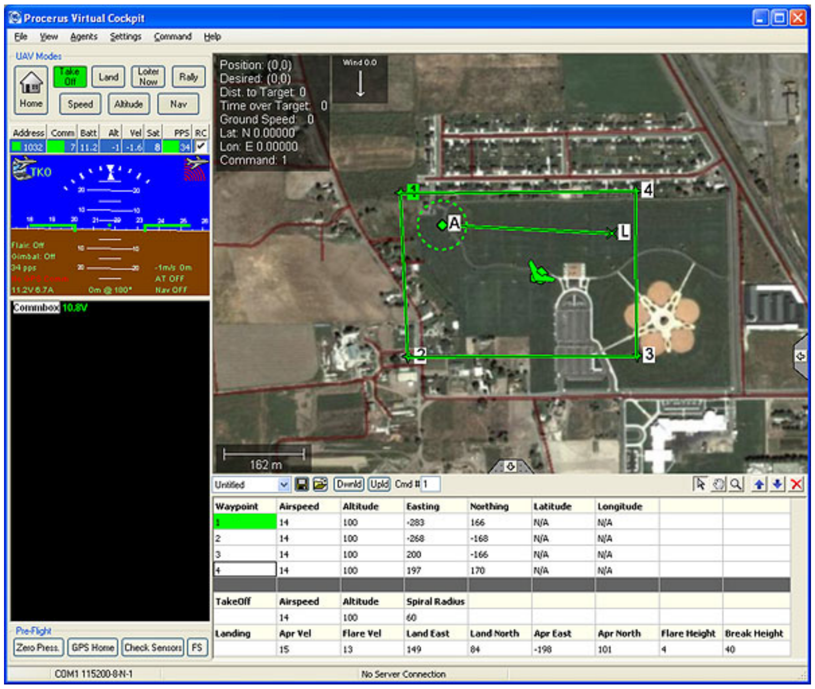
The Virtual Cockpit, shown in fig. 5.4, is a Windows-based GUI for the Kestrel autopilot, and is very easy to use. It is able to control and monitor multiple UAVs. However, each UAV is controlled individually simply by setting lists of waypoints for each UAV. The UAVs do not communicate to each other through the Virtual Cockpit. Our goal is to implement cooperative control algorithms for UAVs using these tools. This goal is made possible through the software development kit supplied by Kestrel. This kit allows the developer to develop software that extends the Virtual Cockpit's functionality.

5.3 Design Requirements

The objective of the research in this chapter is to develop multi-UAV cooperative control algorithms and the software for the testbed similar to the testbed designed for the AmigoBots in Chapter 2. This UAV test bed will aid in much future research.

Specific design goals include:

- Implement pre-flight checks and settings.
  - Zero pressure - sets the current air pressure as the reference. Speed and altitude is then calculated us the difference measured from this reference.
  - Check sensors - verifies that all sensor readings are within reasonable bounds and working properly.



Address	Comm	Batt	Alt	Vel	Sat	PPS	RC
1032	7	11.2	-1	-1.6	8	34	✓

Flair: Off
Gimbal: Off
34 pps
No GPS Comm
11.2V 6.7A
0m @ 100°
-1m/s 0m
AT OFF
Nav OFF

Position: (0,0)
Desired: (0,0)
Dist. to Target: 0
Time over Target: 0
Ground Speed: 0
Lat: N 0.00000
Lon: E 0.00000
Command: 1

Cmd # 1

Waypoint	Airspeed	Altitude	Easting	Northing	Latitude	Longitude		
1	14	100	-283	166	N/A	N/A		
2	14	100	-268	-168	N/A	N/A		
3	14	100	200	-166	N/A	N/A		
4	14	100	197	170	N/A	N/A		
TakeOff	Airspeed	Altitude	Spiral Radius					
	14	100	60					
Landing	Apr Vel	Flare Vel	Land East	Land North	Apr East	Apr North	Flare Height	Break Height
	15	13	149	84	-198	101	4	40

Fig. 5.4: Kestrel virtual cockpit.

- Fail safe - uploads predetermined actions for the UAV to take in the case of catastrophic errors or loss of communication.
- Set GPS home - Sets the current GPS coordinates as the home position where it will return and land.

- Set UAV autopilot modes.
  - Take-off mode - uploads a waypoint that the UAV will go to upon successful take-off and initiates the take-off sequence.
  - Land mode - uploads the specific landing parameters and initiates the landing sequence on the autopilot.
  - Loiter mode - tells the autopilot to begin circling its current position.
  - Navigation mode - causes the autopilot to follow a flight plan or list of waypoints.
  - Manual mode - allows the UAV to be controlled manually by RC transmitter.
  - Follower mode - causes the leader follower algorithm developed later on in this chapter to be enabled.

- Upload/Download waypoints - used to manually enter waypoints and send them to the UAV as well as to find out what waypoints are currently on the UAV's flight plan.
- Data logging - logs all telemetry data for both the leader and follower, as well as all waypoints supplied to the follower by the follower mode controller.

5.4 GUI Design and Functionality

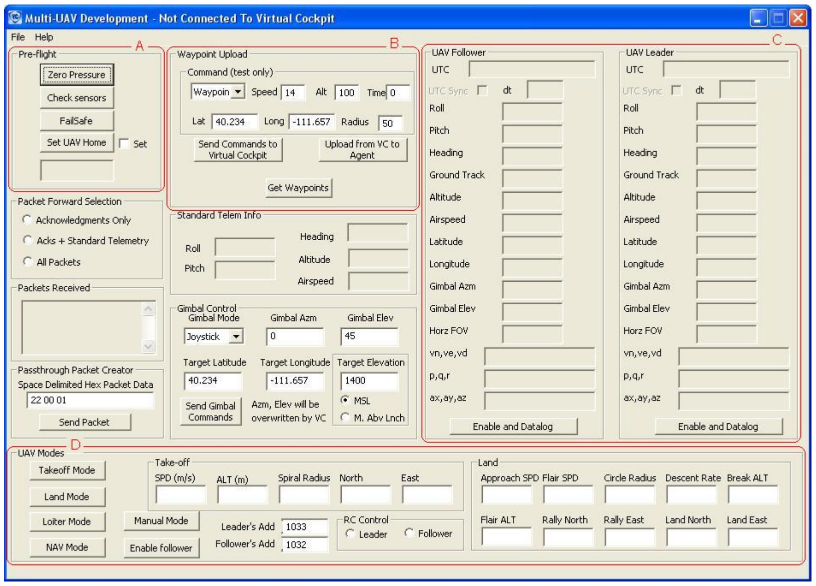
The GUI and all of its functionality are implemented using Visual C++. The GUI design, shown in fig. 5.5, is implemented using only a single dialog box. At the time of this writing, the GUI is used mainly to implement the control algorithm and control the single follower UAV, but will easily be extended to include more UAVs. The reason for this design is that we wanted to get a proof of concept as quickly as possible, and also the fact that we had only two Procerus UAVs available. The leader UAV simply follows a flight path from the Virtual Cockpit. The controller will take telemetry data from the leader and plot waypoints for the follower.

5.4.1 Preflight

The preflight functionality is grouped together in section A of fig. 5.5. The "Zero Pressure" button sends a packet of data to the autopilot containing the command to set the differential and absolute pressures to zero to calibrate them. The "Check Sensors" button sends a packet to the autopilot requesting a sensor check.

A reading from each of the sensors, the 3-axis gyros and accelerometers, the differential and absolute pressure sensors, GPS, temperature and the pitch and roll estimates, is taken and a bitmask is returned that indicates a good or bad status for each of the sensors' readings. When the return packet arrives from the autopilot a message box opens alerting the user of any sensor values that were found to be out of range. Note, the pressures must have been zeroed before checking the sensors.

The "Failsafe" button causes a set of important failsafe functions to be be enabled on the autopilot. There are a large number of failsafes available on the Kestrel autopilot, but



Command: Waypoin Speed 14 Alt 100 Time 0
Lat 40.234 Long -111.657 Radius 50
Space Delimited Hex Packet Data: 22 00 01
Gimbal Mode: Joystick Gimbal Azm: 0 Gimbal Elev: 45
Target Latitude: 40.234 Target Longitude: -111.657 Target Elevation: 1400
[✓ MSL]
Leader's Add 1033
Follower's Add 1032

Fig. 5.5: GUI designed for the cooperative UAV testbed.

not all of them need to be set. The "Failsafe" button sets the failsafe actions for loss of GPS, low battery, critical battery, loss of communication, and altitude. A packet is built containing a bitmask that indicates which failsafe functions are being enabled by the packet. Failure values for each failsafe are also added to the packet. For example, to set the low battery failsafe, the "Low Battery" bit must be set in the bitmask, and a minimum battery voltage value must be specified in the packet. When this failsafe is set on the autopilot, the autopilot will execute a predefined action when the battery voltage drops below the minimum voltage. This action will override all other actions in order to avoid damage due to a low battery.

The "Set UAV Home" button sets the UAV's GPS home position. The GPS home position is a reference for several functions including many failsafe functions, so that the UAV can attempt to return home. GPS home is also used as a reference point for calculating distances. This function uses the UAV's current GPS location to set the GPS home when this button is pushed. Note that for this reason the GPS must have a signal for GPS home to be set. If the GPS home is set successfully, the checkbox next to the "Set GPS Home" button will be checked.

5.4.2 Waypoints

The purpose of the waypoints section, section B in fig. 5.5, is to allow the user to upload waypoints manually to the follower UAV while the cooperative control algorithm is disabled, and also to allow the user to download the UAV's current waypoint. Downloading the UAV's current waypoint is mainly for convenience in debugging control algorithms. A waypoint consists of an altitude, airspeed, latitude, longitude, and radius. The radius is the distance from the actual latitude and longitude of the waypoint that the UAV must be before it considered to have reached the waypoint. To set the waypoint, a packet must be sent to the UAV with this information, likewise when a waypoint is downloaded from the UAV a request packet is sent to the UAV and a similar packet is returned from the UAV containing the waypoint information.

5.4.3 Data Logging

Data logging is done for both the leader UAV and the follower UAV as well as for the control algorithm when it is enabled. The standard telemetry packet that is sent twice per second. An important note is that the polling mode must be used and not the broadcast mode when more than one UAV is being controlled by the same ground station. In broadcast mode the autopilot broadcasts telemetry automatically at 6 Hz. In polling mode the autopilot sends packets only when they are requested. This is necessary to avoid packet collisions. When polling mode is enabled the Virtual Cockpit coordinates the polling. The virtual cockpit by default polls each UAV's telemetry at 2 Hz in polling mode.

The telemetry data is recorded in a file in ASCII format so that it can be easily loaded into Matlab for analysis. Each file is given a unique name with the date included. The same data is recorded for both leader and follower. Section C in fig. 5.5 shows the graphical output of the leader and follower telemetry data log. A waypoint log file is also created to log output from the controller. The waypoint log only applies to the follower and only logs the waypoints sent to the follower from the cooperative control algorithm. The waypoint log files are written in the same format as the telemetry log files and include the altitude, airspeed, latitude, longitude, and radius of the waypoints.

5.4.4 UAV Modes

The UAV modes in section D of fig. 5.5 are used to command the UAV to perform specific functions. Take-off mode must be set in multiple steps. First a special waypoint must be set for the UAV to go to when when take off is complete. Once the waypoint has been successfully sent and an acknowledgment received from the autopilot, a command packet can be sent to change the autopilot mode to take-off mode. Note that if the command packet is sent before the acknowledgment packet is received, the mode will not change. Land mode is very similar to take-off mode. The main difference being that more parameters are required for the special landing waypoint to specify the landing parameters such as the descent rate and the rally point or starting point for the landing sequence.

The other modes are less complicated and only require sending the command to change the mode. Loiter mode causes the UAV to begin circling at its current location. Manual mode turns control of the UAV over to the RC transmitter. And Navigation mode causes the UAV to follow the flight plan, going from one waypoint to the next.

The button labeled "Enable Follower" enables the cooperative controller and sets the UAV mode to Navigation. When the cooperative controller is enabled, first the Virtual Cockpit begins polling the autopilots for their telemetry, and all of the data loggers are initialized if they were not already. The cooperative controller is set to sample the leader UAV's telemetry every second and calculate a new waypoint for the follower UAV. Each time a new waypoint is calculated by the controller, the follower UAV's flight plan is modified by the new waypoint. Since the follower UAV is in navigation mode, it will immediately begin moving toward this new waypoint. However, care must be taken not to send a waypoint to a UAV that is still in the process of uploading a previous waypoint. This can cause collisions and packet errors. To avoid this each time the cooperative control algorithm begins, it checks to see if an acknowledgment packet has been returned from the autopilot indicating that it is ready to receive the next control input.

Also in section D of fig. 5.5, there is a radio button group labeled "RC Control". This is where the user can select which UAV the RC will control when in Manual Mode. The destination of the RC controller is handled by the Virtual Cockpit. Each time one of these radio buttons is pushed a packet is sent to the Virtual Cockpit with the address of the new RC control destination. The addresses of the leader and follower are set by the "Leader Add" and "Follower Add" fields.

5.5 Conclusion

The documentation for the development kit supplied by Kestrel is quite extensive in the description of the more than 75 different packet commands available, but the usage of these packets was at times difficult to decipher and at times required much guessing and trial and error. The main difficulty encountered was the timing of the packet delivery. If packets are sent too quickly in succession so that the first packet is not completely received by the autopilot before the second packet is sent, the second packet will be lost. Our solution to this was to wait for the acknowledgment packet from the autopilot before sending subsequent packets. This required setting flags each time a packet was sent and resetting the flags each time the packets were acknowledged. This does not apply to special packets that are guaranteed by the Virtual Cockpit to be sent. Once these guaranteed packets are sent by the user, the Virtual Cockpit will continue to attempt to send them until they are sent successfully.

While it is possible to directly control the angle of the ailerons and the propeller speed, the controller output is in the form of waypoints and not motor speed and aileron angles. While this sort of low-level control may be done in the future, we were more interested in achieving a proof of concept. The low-level control of the UAV would require a significant amount of time to adequately model the UAV and then to test the controller for safety before any kind of in flight test would be possible. From the point to which we have brought the test bed now, it will be a much simpler task in the future to develop more complex control algorithms.

With the current state of the test bed it should be a simple task to expand it to control more UAVs as they are acquired using the existing code.

Chapter 6
Conclusion

6.1 Summary of Results

The purpose of the research presented in this thesis is to implement and experimentally validate new cooperative control schemes in a physical environment as well as to explore a new technique for handling formation control in the presence of obstacles and the loss or addition of members.

To facilitate the implementation and testing of the control schemes and others on the AmigoBot testbed Chapter 2 details the design and implementation of a graphical user interface. This GUI is a convenient means of storing and recalling experiment parameters. An important feature of this GUI is the ability to distribute the execution of the testbed programs across multiple computers. This ability not only lessens the processing load required from each computer, but also provides a more realistic communication environment. Because each AmigoBot has been fitted with a small on-board computer, the controller can now be executed on each robot's on-board computer; therefore, becoming a more realistic implementation of a distributed system.

Formation control of multiple agents is a very active field of research. In Chapter 3 a new method for dynamically determining the shape of the formation is presented. That is, this scheme allows the formation to change shape and size in order to avoid obstacles or accommodate new mission objectives or terrain. This method also makes it possible for members of the formation to leave and return, or for new members to be added and the formation will adapt to accommodate these changes. As a result cooperative distributed systems can have the ability adapt and change according to their environment and their members.

Coupled harmonic oscillators can be found in many places in nature. The repetitive nature of harmonic oscillators makes them useful for many applications such mapping or perimeter patrolling. Coupled harmonic oscillators lend themselves well to distributed control applications as demonstrated in Chapter 4. A coupled harmonic oscillator control scheme is developed and implemented in both discrete-time and continuous-time. The results show that this method works well for systems with a sparse communication topology and is robust in both continuous-time and discrete-time.

UAVs are gaining much popularity as batteries and other components become lighter and smaller. Small, low-cost UAVs are beginning to become available in the same way that small, low-cost wheeled robots became available years ago. As a result groups of small UAVs can now be used to perform tasks much more efficiently and economically than single monolithic UAVs are able. Chapter 5 describes the implementation of a cooperative multi-UAV testbed for the Procerus Platform. This testbed provides a proof of concept for a distributed multi-UAV controller.

6.2 Future Work

For all of the consensus algorithms discussed in this thesis, the leaders of the formation only receive data from the path planner. An interesting topic of research would be to find an efficient way provide feedback to the path planner. If one or more members of the group were not able to keep in formation, the path planner would be able to adjust so that all members of the formation would be able to follow. This could also be applied to the dynamic formation algorithm in Chapter 3 to allow the members of the formation to act as distributed sensors, and by feeding sensor data back to the path planner, the path planner would be able to effect obstacle avoidance using the capabilities of the dynamic formation.

For the UAVs there is a great deal of interesting work to be done. The testbed can be expanded to control more UAVs, as well as expanded to control UAVs with different autopilots such as the XBow or Paparazzi. Much work still needs to be done to optimize the controllers as well. Cooperative UAV control is such a relatively new area that the possibilities for new work are countless, and the testbed can serve as a base for implementation.

6.3 Conclusion

We have developed and added on to testbeds for two different platforms, the AmigoBot UGV and the Procerus UAV. These testbeds provide a valuable resource for implementing and testing distributed cooperative control algorithms on physical systems. Using these testbeds we have implemented two new cooperative control schemes and compared them to simulations done in Matlab. A new dynamically changing formation algorithm for distributed cooperative systems was developed and shown to work on the AmigoBot testbed. Also, a coupled harmonic oscillator control scheme was developed and experiments done on the testbed showed it to be implementable and robust.

References

[1] B. D. Anderson, C. Yu, S. Dasgupta, and A. S. Morse, "Control of a three-coleader formation in the plane," *Science and Control Letters*, vol. 56, no. 9–10, pp. 573–578, 2007.

[2] M. Arcak, "Passivity as a design tool for group coordination," *IEEE Transactions on Automatic Control*, vol. 52, no. 8, pp. 1380–1390, Aug. 2007.

[3] P. K. C. Wang, "Navigation strategies for multiple autonomous mobile robots moving in formation," *IEEE/RSJ International Workshop on Intelligent Robots and Systems*, pp. 486–493, 1989.

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

[5] W. Ren and R. W. Beard, *Distributed Consensus in Multi-vehicle Cooperative Control*, ser. Communications and Control Engineering Series. London: Springer-Verlag, 2008.

[6] W. Ren, R. W. Beard, and E. M. Atkins, "Information consensus in multivehicle cooperative control: Collective group behavior through local interaction," *IEEE Control Systems Magazine*, vol. 27, no. 2, pp. 71–82, Apr. 2007.

[7] R. Olfati-Saber, J. A. Fax, and R. M. Murray, "Consensus and cooperation in networked multi-agent systems," *Proceedings of the IEEE*, vol. 95, no. 1, pp. 215–233, Jan. 2007.

[8] W. Ren and N. Sorensen, "Distributed coordination architecture for multi-robot formation control," *Robotics and Autonomous Systems*, vol. 56, no. 4, pp. 324–333, Apr. 2008.

[9] W. Ren and R. W. Beard, "Virtual structure based spacecraft formation control with formation feedback," in *AIAA Guidance, Navigation, and Control Conference*, Monterey, CA, Aug. 2002, AIAA Paper No. AIAA-2002-4963.

[10] Y. Li and X. Chen, "Leader-formation navigation using dynamic formation pattern," in *IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pp. 1494–1499, Jul. 2005.

[11] S. Mastellone, D. M. Stipanovic, and M. W. Spong, "Remote formation control and collision avoidance for multi-agent nonholonomic systems," *IEEE International Conference on Robotics and Automation*, pp. 1062–1067, 2007.

[12] Y. Kuramoto, *Chemical Oscillations, Waves and Turbulence.* Springer: Berlin, 1984.

[13] A. Jadbabaie, N. Motee, and M. Barahona, "On the stability of the Kuramoto model of coupled nonlinear oscillators," in *American Control Conference*, vol. 5, pp. 4296–4301, Boston, MA, Jul. 2004.

[14] N. Chopra and M. W. Spong, "On synchronization of Kuramoto oscillators," in *Conference on Decision and Control and European Control Conference*, pp. 3916–3922, Seville, Spain, Dec. 2005.

[15] A. Papachristodoulou and A. Jadbabaie, "Synchronization in oscillator networks: Switching topologies and non-homogeneous delays," in *Conference on Decision and Control and European Control Conference*, pp. 5692–5697, Seville, Spain, Dec. 2005.

[16] D. A. Paley, N. E. Leonard, and R. Sepulchre, "Oscillator models and collective motion: Splay state stabilization of self-propelled particles," in *Conference on Decision and Control and European Control Conference*, pp. 3935–3940, Seville, Spain, Dec. 2005.

[17] D. A. Paley, N. E. Leonard, and R. Sepulchre, "Collective motion of self-propelled particles: Stabilizing symmetric formations on closed curves," in *Conference on Decision and Control*, pp. 5067–5072, San Diego, CA, Dec. 2006.

[18] N. Chopra and M. W. Spong, "Passivity-based control of multi-agent systems," in *Advances in Robot Control: From Everyday Physics to Human-Like Movements*, S. Kawamura and M. Svinin, Eds., pp. 107–134. Berlin: Springer-Verlag, 2006.

[19] W. Ren, "Synchronization of coupled harmonic oscillators with application to motion coordination," in *Automatica*, 2008, (to appear).

[20] W. Ren and E. M. Atkins, "Distributed multi-vehicle coordinated control via local information exchange," *International Journal of Robust and Nonlinear Control*, vol. 17, no. 10–11, pp. 1002–1033, Jul. 2007.

[21] G. Xie and L. Wang, "Consensus control for a class of networks of dynamic agents," *International Journal of Robust and Nonlinear Control*, vol. 17, no. 10-11, pp. 941–959, Jul. 2007.

[22] D. Cruz, J. McClintock, B. Perteet, O. Orqueda, Y. Cao, and R. Fierro, "Decentralized cooperative control: A multivehicle platform for research in networked embedded systems," *IEEE Control System Magazine*, vol. 27, no. 3, pp. 58–78, Jun. 2007.

[23] Z. Qu, J. Wang, and C. E. Plaisted, "A new analytical solution to mobile robot trajectory generation in the presence of moving obstacles," *IEEE Transactions on Robotics*, vol. 20, no. 6, pp. 978–993, Dec. 2004.

[24] *Kestrel Autopilot System, Communication Protocol for Virtual Cockpit and Kestrel Autopilot Firmware*, Procerus Technologies [Online]. Available: `http://www.procerusuav.com`.

Appendices

Appendix A
Manual of Aria Usage

A.1 Introduction to the P3-DX and AmigoBot Platform

The robots in Mobile Robots Lab include two P3-DX and five AmigoBot mobile robots from ActivMedia Robotics shown in fig. A.1. All of the robots in the lab are fundamentally similar. The main differences between the P3-DX and AmigoBot are size and capacity. The P3-DX is larger and has a larger capacity for sensor and actuator payload. Each robot has a differential drive system with rear castor, high precision wheel encoders, and eight sonar positioned around the frame of the robot. Finally, each robot is equipped with an 802.11b wireless card for communication with the host PC.

The encoders can be used quite reliably for dead reckoning (or to keep track of the absolute X, Y, θ position). The encoder provides 123 tics per millimeter. It is important to remember that each time a wheel slips or skids the error in this measurement increases. It is always important to keep in mind, when using any sensor, how that sensor interacts with object that it senses.

Sonar is a very useful for finding the robot's position relative to its surroundings. It is, however, a little trickier to use than the encoders. The sonar on these robots have a maximum range of 5000mm (5 meters) and a minimum range of 100mm (10 centimeters). Although the sonar can sense objects in its path in this range, it is most accurate between 100mm and about 2000mm. The most common problem when using sonar is letting the robot get too close to an object. Anything closer than 10cm cannot be seen because the sonar ping echoes back to the robot before it begins listening for it. Other common issues with sonar happen when the robot is in a tightly enclosed space where the sonar can echo off multiple walls before returning to the robot, and thus giving a false reading. Also if the

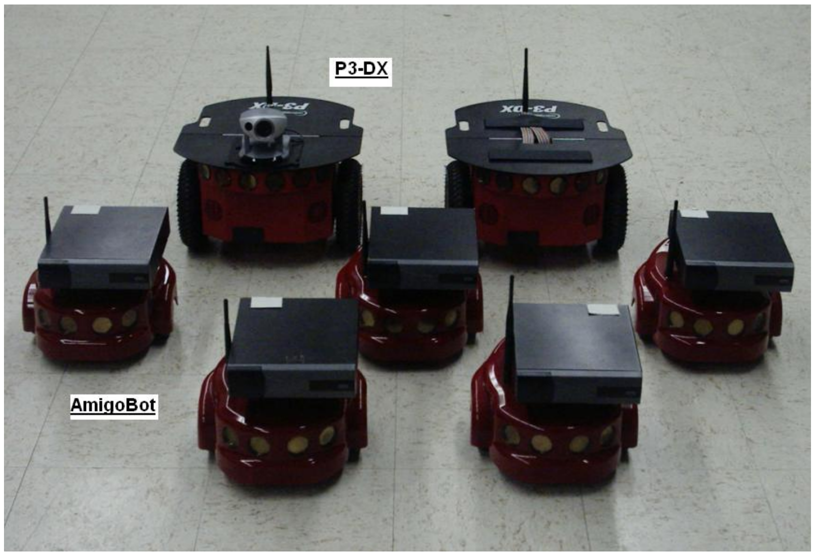


Fig. A.1: Two P3-DX and five AmigoBot mobile robots.

robot is in close proximity to another robot, it is possible for the robot to mistake a ping from a nearby robot as its own.

All of the robots have 16-bit micro-controllers that supports all of the sensors, wheel drives and other functionality of the robot. However, the actual control program is run remotely. It is helpful to think of the PC as part of the robot. The PC executes the programs and sends the control commands and receives the sensor data to and from the robot's micro-controller via the 802.11b wireless connection.

A.2 Overview of Aria

Aria is an open source Software Development Kit (SDK) that provides all the functionality, sending and receiving commands and information to and from the robot. It is written in C++ and compiled as a dll that can be included in other projects. There are several fundamental classes that can be used to implement the majority of any Aria-based program. I will go over each of them and give a description of what they do and how they are used, but this will not be a complete listing of all the classes and functions in Aria. There are many useful, more advanced, classes and functions included in the SDK.

A.2.1 ArRobot Class

A logical starting point is with the ArRobot class. This is a very important class. It is used to represent the actual robot in your program. The class handles all communication between the PC and the robot. That means that you will never have to worry about deciphering the command codes used to send and request data. This class also makes sure that the most recent sensor data from the robot is always available for use. To add a robot to your program, all that needs to be done is to declare a variable of type ArRobot. Once the variable is declared, it can be used to send commands to, and receive sensor data from, the robot. Multiple robots can be controlled by a single program simply by declaring multiple instances of the ArRobot class.

A.2.2 ArFunctor Class

The ArFunctor class can be a bit difficult to understand, but it is really quite simple. All it is, is a pointer to a function. Thus the name, FUNCtion poinTOR. The thing that makes a functor different from a regular pointer to a function in C++ is that if a regular pointer points to a class member function, it will not have access to the class' "this" pointer. A functor does have access to the "this" pointer. This is important. Functors very useful for "callbacks." This will be explained in detail in section 5 (Using ArNetworking Library).

ArFunctor is the base class for quite a few Functor implementations. This manual will cover the most used and most important ones. To find a list of all functor classes, look in the ArFunctor.h file, in the AriaDLL project of the SDK. The following are examples of several different functor types.

- ArFunctorC<CallbackContainer> functor1(cb,&CallbackContainer::callback1);
- ArFunctor1C<CallbackContainer, int> functor2(cb,&CallbackContainer::callback2);
- ArRetFunctor1C<bool, CallbackContainer, const char *>
  functor3(cb,&CallbackContainer::callback3);

In the first example, a functor named functor1 is declared. ArFunctorC functors point to a class member function that returns no value and takes no arguments. Inside the angle brackets (<>), CallBackContainer is the name of the class that contains that function that is being pointed to. The ArFunctorC constructor takes two arguments, cb, which is a pointer to a variable whose "this" pointer will be used, and the address of the function that it will point to.

The second example is very similar. ArFunctor1C points to a class member function that takes one argument. The type of this argument is specified by the second parameter in angle brackets.

Finally the last example is, again, very similar to the first two, with the only difference being that it returns a value. The type of the return value is specified by the first argument in angle brackets.

Once a functor is initialized, it can be used as an argument to other functions so that a function can call another function. This is where they become very important for callbacks; because we can create a list of functors, then whenever an event that is related to this list of functors occurs, it is possible to iterate through this list invoking each of the functions that the functors point to. The usefulness of this will become clearer in section 5.

A.2.3 ArAction Class

The ArAction class is very useful for designing specific sets of actions that the robot can perform. As with any action in real life, each ArAction has a priority. For example, in a car, the driver's "Stop" action should have a higher priority than the "Go Forward" action, so that the driver will stop if there is an obstacle blocking his path. This is, in fact, how the ArAction class is used. An ArActionStop and an ArActionGo class can be derived from the ArAction class. By giving the ArActionStop action a higher priority than the ArActionGo action, the robot can go forward and stop before hitting an obstacle. The code examples in this section are based on the Aria SDK *actionExample* code. Notice in algorithm A.1 that there are three virtual functions declared. The only function that must be overridden is "virtual ArActionDesired* fire(ArActionDesired currentDesired);". This is where the behavior for this action is implemented. To use this new class once it has been implemented, all that needs to be done is to declare an instance of it and add that instance

Algorithm A.1 ArActionGo definition.

```
classActionGo : public ArAction
{
public:
    ActionGo(double maxSpeed, double stopDistance);
    virtual  ActionGo(void) ;
    virtual ArActionDesired *fire(ArActionDesired currentDesired);
    virtual void setRobot(ArRobot *robot);
protected:
    ArRangeDevice *mySonar;
    ArActionDesired myDesired;
    double myMaxSpeed;
    double myStopDistance;
};
```

to the robot's actions.

The addAction function shown in algorithm A.2, is a member of ArRobot, and takes two arguments. The first is the address of the instance of the action. The second argument is a priority value from 0 to 100. The ArActionStallRecover, which is an action class included in the SDK, is also used in this example. Another very useful ArAction subclass included in the Aria SDK is ArActionInput. This class provides methods to set the velocity and rotation velocity of a robot just as you would using ArRobot. The difference is that since it is an ArAction subclass, it has a priority. This makes it possible to implement object avoidance or other behaviors separately from the main control algorithm. In this way robot will be protected from running into obstacles or ensured to perform other high priority actions, no matter what the control algorithm says. For example, the algorithm A.3 shows how this might be used.

In algorithm A.3, there are two ArActions. The stop action is given a higher priority than ArActionInput. This ensures that robot will stop when it reaches a wall or obstacle. The robot's velocity is then set using the ArActionInput instance.

There are several other ArAction subclasses available in the Aria SDK, that will not be mentioned here, but they can be found in the AriaDLL project of the SDK. Also for a

Algorithm A.2 ArAction usage.

```
ActionGo go(500, 350);
ArActionStallRecover recover;

robot.addAction(&recover, 100);
robot.addAction(&go, 50);
```

Algorithm A.3 ArActionInput usage.

```
ArActionInput myActionInput;
ArActionStop myActionStop;

robot.addAction(&myActionInput,50);
robot.addAction(&myActionStop,99);

myActionInput.setVel(300);
```

deeper understanding of how to use ArActions, the *actionExample* in the examples folder section of the SDK is very helpful.

A.3 Creating a New Project

Creating a new project using the Aria SDK is not difficult, but it can be for those who don't have a good understanding of how all the pieces of the project fit together. So before doing any programming, it is important to know how to create a project. It is essentially the same as creating a regular project in Microsoft Visual Studio.NET. The difficulties lie in making sure that all the settings are correct. The following steps will result in a working project. Note, the settings for the project need to be adjusted to match those required to access the Aria platform code. Settings may also have to be modified based on where you put the project. The following is assumes that the project folder is placed in the Aria directory.

1. Start an instance of Microsoft Visual Studio.NET.

2. Open the New Project Wizard. File→New→Project.

3. Select "Win32 Console Project" from the available templates.

4. Enter a project name in the appropriate field.
5. Select the Aria directory for the location field. The default Aria directory is "C:\Program Files\MobileRobots\Aria", but it may differ.
6. Click OK.
7. Add a new .cpp file to the project by right clicking the project name in the Solution Explorer, then select Add→Add New Item, and follow the prompts to create a new file.
8. Modify Settings.
   (a) Right click on the new project name in the Solution Explorer.
   (b) Select Properties from the context menu.
   (c) Select the General tab in the properties dialog box.
   (d) Enter ../bin in the Output Directory field.
   (e) Enter ../obj/debug in the Intermediate Directory field.
   (f) Select the General tab under the C/C++ folder.
   (g) Enter ..\include in the Additional Include Directories field.
   (h) Select the General tab under the Linker folder.
   (i) Enter ../bin/$(ProjectName).exe in the Output File field.
   (j) Enter ../lib in the Additional Library Directories field.
   (k) Select the Input tab under the Linker folder.
   (l) Enter the following libraries in the Additional Dependencies field: Aria.lib, ArNetworking.lib, wsock32.lib winmm.lib, and advapi32.lib. Note, that not all are needed for every project.
   (m) Click OK.

Note that once your project is created using these steps, you still need to make sure that the entire Aria solution has been built and that the Aria.dll and ArNetworking.dll are both in the Additional Library Directories that you specified previously. A simple way to avoid confusion for beginners is to copy the "demo" project from the "Aria/Examples" folder into a new folder within the "Aria" directory. Then rename to project file and add it to the Aria solution.

Once these steps are complete the project will be ready to compile and run.

A.4 Writing a Simple Program

The simplest way to begin a new program is to modify the contents of *demo.cpp*; however, *demo.cpp* contains a lot of code that is not necessary for the AmigoBots or the P3-DXs in the lab. Knowing what is necessary or not can result in smaller more efficient code. This section will show a break-down of the *demo.cpp* code that is necessary for our lab's robots in algorithm A.4.

The first thing that has to be done in any program is to include the Aria.h header file. Next, inside the main function, Aria must be initialized before using anything else from the Aria SDK. Aria::init() initializes Aria's thread layer and the signal handling methods. For Windows it also initializes the socket layer.

After Aria has been initialized the variables can be declared and initialized. ArArgument parser parses the command line arguments passed in through *main()*. ArSimpleConnector is used to connect the PC to the robot or simulator. Notice ArRobot. This means that demo will be controlling one robot. Finally, ArSonarDevice provides a means of objectifying the robot's sonar.

After the variables have been declared, the next step is to parse the command line arguments and load any default arguments, so that they can be used in initialization. Default arguments are not often used, so they will not be covered any further here. The parser.checkHelpAndWarnUnparsed() function makes sure that all of the arguments are valid. If there are any invalid command line arguments this function also prints out a warning message and returns false.

Following the parsing of the arguments a keyHandler is declared and initialized so that if the user pushes the Esc key, the program will exit. This is not necessary for most code to run, but it is good practice so that Aria can close down normally.

Next, our *robot* is given access to *sonarDev.* This just makes the robot start collecting sonar readings and making them available to the rest of the program. After the *robot* has been initialized, simpleConnector.connectRobot(&robot) will connect the PC to the actual robot, and robot.runAsync() starts the *robot*'s thread in an asynchronous mode. In other words, its execution will not by synchronized with the main program thread of execution.

Finally, the last step is to initialize the robot's motors with the robot.comInt(ArCommands::ENABLE, 1) command, and add the behavioral code to control the robot. For example the ArActionGo action could be activated here.

A.5 Using ArNetworking Library

The ArNetworking library is just another part of the Aria SDK. This portion of the SDK makes it possible to create servers and clients in your program, so that multiple programs on multiple computers can communicate with each other. This is useful in a variety of situations, such as, to spread the work load over multiple computers if there are a large number of robots to control. The most important use for ArNetworking is that it gives the ability to simulate real life communication between robots. It is used extensively this way in our research of consensus algorithms.

As with the Aria library, there are several classes within the ArNetworking library that are more important than the others, and will be used in anything you might want to do with ArNetworking. These classes are: ArServerBase, ArClientBase, and ArNetPacket.

To understand how to use these classes well, it is important to understand their relationships with each other. ArServerBase represents a server. A server's job is to open a port, or line of communication. The server does not actively make a connection with anything, but waits for a client to connect to it and request some sort of data or action from it. For example, when a browser is opened and connects to a site on the internet, the

Algorithm A.4 demo.cpp.

```
#include "Aria.h"
int main(int argc, char** argv)
{
        Aria::init();
        ArArgumentParser parser(&argc, argv);
        ArSimpleConnector simpleConnector(&parser);
        ArRobot robot;
        ArSonarDevice sonarDev;
        parser.loadDefaultArguments();
        if (!Aria::parseArgs() —— !parser.checkHelpAndWarnUnparsed())
        {
            Aria::logOptions();
            exit(1);
        }
        ArKeyHandler keyHandler;
        Aria::setKeyHandler(&keyHandler);
        robot.attachKeyHandler(&keyHandler);
        printf("You may press escape to exit\n");
        robot.addRangeDevice(&sonarDev);
        if (!simpleConnector.connectRobot(&robot))
        {
            printf("Could not connect to robot... exiting\n");
            Aria::exit(1);
        }
        robot.runAsync(true);
        robot.lock();
        robot.comInt(ArCommands::ENABLE, 1);
        // TODO Add behavior here
        robot.unlock();
        robot.waitForRunExit();
        Aria::exit(0);
        return 0;
}
```

browser connects to a server that has the data for the page that the browser can request from it.

ArClientBase represents the client. In the example of the internet above, the browser is the client. The client must know where to find the server and which port the server is listening on. Once the client is connected to the server, it can request data from the server. A client can connect to only one server at a time, but a server can have multiple clients connected to it at a time.

There must be a standard way to send data between the client and server. This is the purpose of ArNetPacket. The packet holds the data that is to be transferred between the server and client.

A.5.1 ArServerBase Example

In this example shown in algorithms A.5 and A.6 you will see how to implement a server using ArServerBase. ArServerBase requires a class to handle the data output when a client sends it a request. This is a special type of class commonly referred to as a handler class. The handler class makes heavy use of callbacks and therefore functors. The idea of a callback is important to understand. They allow both the client and server to process other things while waiting for data to be sent or received.

Algorithm A.5 shows the handler class definition. It has a function called outputData that has two arguments, an ArServerClient* and an ArNetPacket*. This function will be used as the callback. Each time a client that is connected to the server requests the server's data (in this case the myData variable) this function will handle the request. The OutputHandler class also has a functor called myOutputDataCB. Notice that the pointer will point to a function that is a member of this class, and will take two arguments of type ArServerClient* and ArNetpacket*.

The implementation of the OutputHandler class is shown in Algorithm A.6. The first thing to notice is that the functor is initialized in the call to the constructor, and is set to point to the outputData member function. Next, notice that there is a line in the constructor that calls a function of the ArServerBase class, called addData. This function

Algorithm A.5 Handler class definition.

```
classOutputHandler
{
public:
    OutputHandler(ArServerBase *server, double* data);
    virtual  OutputHandler(void);
    void outputData(ArServerClient *client, ArNetPacket *packet);
protected:
    ArServerBase *myServer;
    double *myData;
    ArNetPacket myPacket;
    ArFunctor2C¡OutputHandler, ArServerClient *, ArNetPacket *¿
    myOutputDataCB;
};
```

adds our functor to a list of functors that will be invoked each time the server receives a request for "getData". The arguments are: the name used in the request for the data, a description of the function that is pointed to by the functor, a pointer to the functor, a description of the arguments that the callback function is expecting to be contained in the request packet, and a description of the data that will be returned by the callback function in the reply packet.

All the outputData function has to do in this example, add the myData variable to the reply packet, and send it. The myPacket.empty() function is there simply to make sure that there is no data already in the packet. If the client had sent data with the request to the server, the server could retrieve the data with a line similar to this: clientData = packet→bufToDouble();.

Algorithm A.7 shows the final steps for creating a server with the SDK. There should not be anything very unexpected here. First, a variable of type ArServerBase is declared. Then, ArServerSimpleOpener is used to open the port that the server will listen on. ArServerSimple Opener is a class that simplifies opening the server port when the port is specified in the command line arguments. It is possible to use *server.open(int port);* to open the server port. After the port has been opened, the runAsync() function tells *server*

Algorithm A.6 Handler class implementation.

```
OutputHandler::OutputHandler(ArServerBase *server, double* data) :
        myOutputDataCB(this, &OutputHandler::outputData)
{
        myServer = server;
        myData = data;
        myServer-¿addData(”getData”, ””, &myOutputDataCB, ”none”, ”double”);
}

OutputHandler:: OutputHandler(void)
{}
void OutputHandler::outputData(ArServerClient *client, ArNetPacket *packet)
{
        myPacket.empty();
        myPacket.byteToBuf(*myData);
        myPacket.finilize();
        client→sendPacketTcp(&myPacket);
}
```

to begin listening for client connections. And lastly, an instance of the OutputHandler class needs to be declared so that it can begin handling the requests to the server.

A.5.2 ArClientBase Example

There are very few differences in setting up a Server from a Client as you will see. As with the server example, the client needs a handler to handle the sending requests to the server and receiving the data. Algorithm A.8 shows the class definition of this handler. The only difference between this class definition and the OutputHandler class definition for the server is that the functor, instead of pointing to a function with two arguments, points to a function with only one argument.

Algorithm A.9 shows the implementation of the client's handler class. Again there are very few changes from the server's handler class. The most important change is in the constructor. To send a request, the handler uses ArClientBase's addHandler and Request or RequestOnce functions. The addHandler function tells *myClient* what functor to invoke each time myClient sends a request to the server for “getData”. The request function tells *myClient* to send a request to the server for “getData” every 100 milliseconds.

It will not be surprising that starting the client, shown in algorithm A.10 is nearly the same as starting a server. In fact the only thing that is really noteworthy that you have not seen yet is the way client is connected to the server. Instead of opening the client, the client *connects* to the server, using the address where the server can be found and the port that it is listening on. To learn more about the ArNetworking library look at Aria’s *serverDemo* and *clientDemo*.

A.6 Running Code on the MobileSim Simulator

MobileSim is a very useful and very powerful simulator for the mobile robot platform. It should be the starting point for testing new programs or algorithms for the robots.

To open MobileSim, first open a command prompt. Then, after changing to MobileSim’s directory (C:\Program Files\MobileRobots\MobileSim), type “mobilesim.” That will run MobileSim with its default settings. You will see a dialog box that lets you choose a map to load in MobileSim or to continue with no map. Maps can be created for MobileSim using Mapper3-basic. By default MobileSim opens with only one P3-DX available. To open MobileSim with more than one robot or with multiple types of robots use the following argument format in the command prompt: “mobilesim -r amigo-sh -r amigo-sh -r p3dx -r p3dx.” This would open MobileSim with two AmigoBots and two P3-DXs.rectory (C:\Program Files\MobileRobots\MobileSim), type “mobilesim.” That will run MobileSim with its default settings. You will see a dialog box that lets you choose a map to load in MobileSim or to continue with no map. Maps can be created for MobileSim using Mapper3-basic. By default MobileSim opens with only one P3-DX available. To open MobileSim with more than one robot or with multiple types of robots use the following argument format in the command prompt: “mobilesim -r amigo-sh -r amigo-sh -r p3dx -r p3dx.” This would open MobileSim with two AmigoBots and two P3-DXs.

To connect a program to the robots in MobileSim, the program needs to be executed from the command prompt so that you can take advantage of the command line options. When MobileSim loads a robot it makes that robot available on a certain port. The first robot to be loaded, is loaded on port 8101. Each additional robot is loaded on the next consecutive port. So, if you wanted to run *demo.exe* on the first P3-DX that was loaded in MobileSim, you would type the following in the command prompt: "demo -rrtp 8103." For a complete list of command line options for demo type: "demo -h."

A.7 Running Code on the Robots

Running your program on a robot is very similar to running the code on MobileSim. The difference is that instead of connecting to some port on MobileSim, you must connect directly to the robot's IP address. In the lab, the robots each have a static local IP address assigned to them. To run *demo.exe* on one of the AmigoBots, whose IP address happens to be 192.168.1.11, you would type "demo -rh 192.168.1.11" in the command prompt. For each robot you wish to run the program on, you would repeat this process, changing the address to the appropriate address for the robot.

It is also possible, with the robots in the lab, to use an external IP address to connect to them. This is possible because each robot has a port forwarded to it. For example, the following would connect to the robot whose port is 8101: "demo -rh 129.123.4.197 -rrtp 8101."

Each of the five AmigoBots and one of the P3-DXs also have on board computers installed on them. The AmigoBot computers run run Windows, and the P3-DX's computer runs a version of Debian LINUX. By logging onto the on board computer using Remote Desktop Connection for the Windows computers or some remote access program such as Putty for LINUX, you can also run programs on the robot from its own on board computer. The computers are connected to the robot via serial connection. This means that wanted to run *demo.exe* from the on board computer, you would only have to type: "demo" from the command prompt. As long as MobileSim is not running when you do this, *demo.exe* will connect to the robot over its serial connection. The IP addresses for the on board computers are also listed in table A.1.

Table A.1: Command line arguments for connecting to the lab robots.

Robot	Internal Address	External Address	On board Computer IP Address
P3-DX 1	-rh 192.168.1.10	-rh 129.123.4.197 -rrtp 8106	
AmigoBot 1	-rh 192.168.1.11	-rh 129.123.4.197 -rrtp 8101	192.168.1.41
AmigoBot 2	-rh 192.168.1.12	-rh 129.123.4.197 -rrtp 8102	192.168.1.42
AmigoBot 3	-rh 192.168.1.13	-rh 129.123.4.197 -rrtp 8103	192.168.1.43
AmigoBot 4	-rh 192.168.1.14	-rh 129.123.4.197 -rrtp 8104	192.168.1.44
AmigoBot 5	-rh 192.168.1.15	-rh 129.123.4.197 -rrtp 8105	192.168.1.45
P3-DX 2		-rh 129.123.4.197 -rrtp 8107	192.168.1.16

Algorithm A.7 ArServerBase initialization.

```
void main(int argc, char** argv)
{
    Aria::init();
    ArServerBase server;
    double* data;
    ArArgumentParser parser(&argc, argv);
    ArSimpleConnector simpleConnector(&parser);
    ArServerSimpleOpener simpleOpener(&parser);
    parser.loadDefaultArguments();
    if(!simpleConnector.parseArgs() || !simpleOpener.parseArgs() ||
        !parser.checkHelpAndWarnUnparsed())
    {
        simpleConnector.logOptions();
        simpleOpener.logOptions();
        exit(1);
    }
    if(!simpleOpener.open(&server))
    {
        printf("Could not open server port.\n");
        exit(1);
    }
    server.runAsync();
    OutputHandler outputHandler(&server, &data);
    while(1)
    {
        // It is possible to change the data variable here
        Sleep(1);
    }
    Aria::exit(1)
}
```

Algorithm A.8 Handler class definition.

```
classOutputHandler
{
public:
    OutputHandler(ArClientBase *client, double* data);
    void handleData(ArNetPacket *packet);
    ArFunctor1C¡DriverOutputHandler, ArNetPacket *¿ myHandleDataCB;
protected:
    ArClientBase *myClient;
    double* myData;
};
```

Algorithm A.9 OutputHandler implementation.

```
OutputHandler:: OutputHandler(ArClientBase *client, double* data) :
      myHandleDataCB(this, &OutputHandler::handleData)
{
      myClient = client;
      myData = data;
      myClient→addHandler(”getData”, &myHandleDataCB);
      myClient→request(”getData”, 100);
}

void OutputHandler::handleData(ArNetPacket *packet)
{
      *myDriverOutput = packet→bufToDouble();
}
```

Algorithm A.10 ArClientBase initialization.

```
void main(int argc, char **argv)
{
    Aria::init();
    ArArgumentParser parser(&argc, argv);
    ArClientBase client;
    double* data;
    if(!client.blockingConnect(”localhost”, 7300))
    {
        printf(”Failed to connect to client server\n”);
        exit(1);
    }
    printf(”Connected to client servers.\n”);
    OutputHandler outputHandler(&client, &data);
    client.runAsync();
    while(client.isConnected())
    {
        // Do something with data
        ArUtil::sleep(100);
    }
}
```

Appendix B
Overview of AmigoBot Testbed

B.1 Introduction to the Testbed

In order to research cooperative control systems, we have assembled a testbed for multi-vehicle cooperative control. This testbed was used in all of the simulations and experiments in this thesis. We will give an overview of this testbed as background for the experiments. The testbed includes five AmigoBots with on board computers. The software for the testbed is written in C++ using the Aria SDK. The full testbed software package includes five different executables that will be explained in detail in the following appendix. Three programs comprise the core of the testbed software, “DriverServer.exe”, “ClientTest.exe”, and “ServerTest.exe”. These three programs work together to form a distributed communication network between the AmigoBots shown in fig. B.1, where the communication topology as well as many other control parameters are time varying. One program is the GUI that is used developed in Chapter 2, and the last program, “RexecService.exe” is a windows service that allows “DriverServer.exe”, “ClientTest.exe” and “ServerTest.exe” to be executed on remote computers from the GUI. This appendix will focus only on the three core programs, as the other two are covered in detail in Chapter 2.

B.2 Testbed Design

In network topology illustrated in fig. B.1, the top level, “DriverServer.exe”, represents a path planner. The communication between the path planner and other robots can be limited to as few as one or as many as all of the robots. The robots with communication links to the path planner are the leaders. Those with communication links only to other robots are followers. The communication between individual robots can also be time varying. This

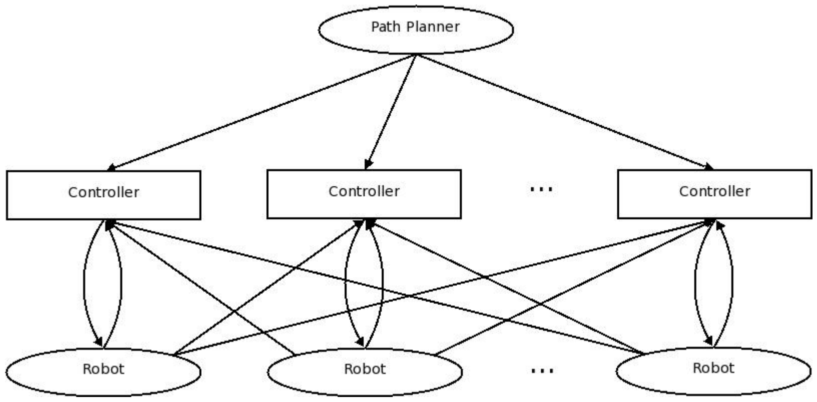


Fig. B.1: Network topology implemented in testbed to simulate communication between robots.

simulates a limited or even changing communication topology.

“DriverServer.exe” implements an ArNetworking server object (See Appendix A for details on ArNetworking and the ArServerBase class). As clients connect to the server data is made available to them through request. “DriverServer.exe” implements a timer that updates the path and time varying data that is made available to the robots. The communication matrix and leader array, as well as the dynamic formation parameters for dynamic formation experiments, are updated at ten second intervals. This is how the time varying communication topology and formation parameters are implemented.

B.2.1 Connection Matrix

The communication matrix is what determines the communication topology between robots. Every robot, leader or follower, must continuously request the communication matrix from the server. The values of the communication matrix cells can be 1’s or 0’s. A 1 in cell (2,3) means that robot 2 receives data from robot 3. A 0 means that it does not.

B.2.2 Virtual Center

The path planner has several built in functions to plot the desired path of formation. The position that the path planner plots for the formation is called the virtual center and includes the (x, y) position as well as the heading, θ. Only leaders are allowed to use this data. Currently the path plotting functions provide linear, circular, and figure 8 paths, as well as a stationary rotating path.

B.2.3 Leader Array

The leader array serves two purposes. First, the leader array indicates which of the robots are leaders, or have access to the virtual center and other mission information other than the connection matrix from the path planner. Second, it determines which control algorithm will be used for the experiment. This and the communication matrix are the only two things available to all robots from "DriverServer.exe". At the time that this appendix was written, the available leader array cell values are those shown in table B.1.

B.2.4 Dynamic Formation Parameters

The four time varying formation parameters a, b, poe, and $shift$, are made available to the leaders. "DriverServer.exe" also implements an algorithm to limit the rate at which these parameters can change. Chapter 3 explains these parameters and the algorithm in detail.

B.3 ClientTest

The principle purpose of "ClientTest.exe" is to implement the control algorithms. As per the definition of a distributed control system, each robot implements its own controller. Therefore, each robot has its own individual instance of "ClientTest.exe". In fig. B.1 "ClientTest.exe" is represented by the middle layer. "ClientTest.exe" implements the ArNetworking client (see Appendix A for details on ArNetworking and ArClientBase class). To achieve communication with "DriverServer.exe", "ClientTest.exe" must have a client object for that connection. There must also be a client object for each of the "ServerTest.exe" server to handle communication between robots.

Each of the controllers shown in table B.1 is implemented in "ClientTest.exe" as a C++ class. The program runs an infinite loop that cycles at 100 ms intervals. At the beginning of each interval the client to "DriverServer.exe" requests the communication matrix and

Table B.1: Leader array.

Value	Description
0	No connection.
1	Simple follower for consensus controller.
2	Intelligent follower for consensus controller.
3	Leader for consensus controller.
4	Velocity consensus controller.
5	Simple attitude controller.
6	Attitude 1 follower controller.
7	Attitude 1 leader controller.
8	Attitude 2 follower controller.
9	Attitude 2 leader controller.
10	Attitude 3 follower controller.
11	Attitude 3 leader controller.
12	Dynamic formation simple follower controller.
13	Dynamic formation intelligent follower controller.
14	Dynamic formation leader controller.
15	Couple harmonic oscillator follower controller.
16	Coupled harmonic oscillator leader controller.

leader array. The leader array is used to determine which controller class implementation to use, and the communication matrix is used in the control algorithm. “ClientTest.exe” also collects data from each of the “ServerTest.exe” instances and makes it available to the controller class.

B.4 ServerTest.exe

The purpose of the “ServerTest.exe” is to act as a software representation of an AmigoBot. It is represented by the bottom layer in fig. B.1. “ServerTest.exe” handles all of the communication with the physical robot by sending control commands to, and sensor feedback from, the robot. Another ArNetwork server is implemented here and a client from each of the “ClientTest.exe” instances will connect to it to request sensor feedback from the robot. Only the client that corresponds to the particular robot will send control commands, however.