DISTRIBUTED CONTROL FOR ROBOTIC SWARMS USING CENTROIDAL VORONOI TESSELLATIONS

by

Shelley Rounds

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

of

MASTER OF SCIENCE

in

Electrical Engineering

Approved:

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

UTAH STATE UNIVERSITY
Logan, Utah

2008

Abstract

Distributed Control for Robotic Swarms Using
Centroidal Voronoi Tessellations

by

Shelley Rounds, Master of Science
Utah State University, 2008

Major Professor: Dr. YangQuan Chen
Department: Electrical and Computer Engineering

This thesis introduces a design combining an emerging area in robotics with a well established mathematical research topic: swarm intelligence and Voronoi tessellations, respectively. The main objective for this research is to design an economical and robust swarm system to achieve distributed control. This research combines swarm intelligence with Voronoi tessellations to localize a source and create formations. Extensive software coding must be implemented for this design, such as the development of a discrete centroidal Voronoi tessellation (CVT) algorithm.

The ultimate purpose of this research is to advance the existing Mobile Actuator and Sensor Network (MASnet) platform to eventually develop a cooperative robot team that can sense, predict, and finally neutralize a diffusion process. Previous work on the MASnet platform has served as a foundation for this research. While growing closer to the MASnet goal, results also provide stimulating discoveries for mathematical and swarm research areas.

(144 pages)

This work is dedicated to my family and friends for their unfailing support and the Electrical and Computer Engineering Department for their help and guidance.

Acknowledgments

I would like the thank Dr. YangQuan Chen who has given me the opportunity to work on this platform and to continue my education beyond undergraduate experience. His continual hard work allowed me to work with the Center for Self-Organizing and Intelligent Systems (CSOIS) conducting research under the National Science Foundation Research Experience for Undergraduates (NSF REU). His keen eye and reservoir for new and beneficial research topics also helped me recognize the potential in swarm engineering.

I would also like to thank William Bourgeous for his guidance and time spent helping me build a strong research foundation. He is responsible for showing me how exciting swarm engineering can be. As the graduate researcher, he took me under his wing and spent more time needed helping me with my work. He is truly a hard worker and great mentor.

Special thanks go to Florian Zwetti, whom I have worked closely with on the MASnet platform. His long hours of working allowed us to continue our own MASnet projects. His passion and hard work has driven him to a well-earned degree which has also contributed to advancing the platform.

Finally, this thesis must recognize the REU students under my leadership: Jonathan Diaz, Edvier Cabassa, Jordan Wirth, Angel Cortes, and Sean Wolf, who all completed various projects for the MASnet platform; and the PhD student, Haiyang Chao, who helped me with understanding and creating CVTs. Without their work, I could not have finished this thesis.

Shelley Rounds

Contents

[1]The original version of this user's manual can be found in the Appendix of William Bourgeous' thesis [1].

List of Tables

List of Figures

Acronyms

ADC	Analog-to-Digital Converter
CPS	Cyber-Physical System
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSMA/CA	Carrier Sense Multiple Access/Collision Avoidance
CSOIS	Center for Self-Organizing and Intelligent Systems
CVT	Centroidal Voronoi Tessellation
DAC	Digital-to-Analog Converter
EEPROM	Electrically Erasable Programmable Memory
EM	Electromagnetic
GUI	Graphical User Interface
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronic Engineers
IR	Infrared
MAC	Medium Access Control
MASmote	Mobile Actuator and Sensor Mote
MASnet	Mobile Actuator and Sensor Network
MFC	Microsoft Foundation Classes
NSF	National Science Foundation
PAR	Photosynthetically Active Radiation
PCB	Printed Circuit Board
pGPS	Pseudo Global Positioning System
REU	Research Experience for Undergraduates
RF	Radio Frequency
TSR	Total Solar Radiation
WSN	Wireless Sensor Network

Chapter 1
Introduction

1.1 Overview

This research is a contribution to the MASnet project, which includes developing algorithms for phototaxis and formation control using CVTs. The objective is for a swarm of mobile robots to communicate with each other on the MASnet plaform to achieve these tasks. In the process, many discoveries have been uncovered and the MASnet platform system has been improved.

This chapter serves as an introduction of ideas and tools used to create the desired algorithms. Chapter 2 explains the MASnet platform in detail as well as improvements on the platform robots. The platform setup and more communication details for phototaxis are discussed in Chapter 3. In-depth explanations of phototaxis and formation control algorithms are given in Chapters 4 and 5, respectively. Finally, a list of contributions along with suggestions of future work is presented in Chapter 6.

1.2 Motivation

Unfortunately, war and military threats continue to rise at home and abroad. Security and defense efforts are sorely needed in the face of warfare and terrorism. However, surveillance, search, and rescue tactics are extremely hazardous and tedious for soldiers. The CSOIS has successfully developed robots to perform these tasks. The MASnet platform of CSIOS is intended to help the security effort. Previously, CSIOS has focused on single robots; currently, MASnet brings robot swarms to the foreground.

The ultimate goal of the platform is to sense, model, predict, and control a diffusion process. The robots should cooperate with each other to neutralize a toxic area till humans are safe to enter. Simulation results on this subject have been achieved [2, 3].

For this research, a team of robots successfully track a dynamic non-diffusing source. This can be extremely useful in locating chemical, radiological, and nuclear weapons or any source that can be detected by sensors. This research also presents an extremely flexible and self-healing formation control algorithm. Formations for search efforts and battle tactics are crucial. Finally, this thesis evaluates the effect of robot number, initial starting positions, and sensor number on the performance of the swarm. In critical moments, it is important for the robots to work efficiently. This research delivers robot results for new and advanced techniques with these applications in mind.

1.3 Swarm Intelligence and Emergent Behavior

Swarming behavior in biological systems is an astonishing phenomenon. Many species adopt swarming behaviors to complete specific tasks such as foraging, guarding, traveling, or simply surviving. Figure 1.1 shows examples of swarms in nature. Perhaps the most extraordinary observation of swarming behavior is the mass global behavior that emerges from a group of agents, although each agent is only aware of itself and nearest neighbors [4].

A *swarm* can be defined as a group of distributed agents—such as ants, birds, crickets or robots—which have only local knowledge and communication. These local interactions and each agent's individual behavior cause a global *emergent behavior*. As previously mentioned, this emergent behavior is the miracle and most powerful tool for any swarm system.

In recent years, engineers have begun to apply this powerful tool to robotics. Formerly,

(a) A flock of geese (U.S. Fish and Wildlife Service `http://www.fws.gov/midwest/swanlake`)

(b) A school of fish (IEEE CDC 2004 at Paradise Island, Bahamas)

Fig. 1.1: Swarm examples in nature.

robots have been built to sense, actuate, and explore alone, requiring large communications ranges, complicated code and some human control to traverse the terrain. Using swarm intelligent robots, however, can drastically reduce communication ranges, programming code, and human control. Applying swarms to mobile robotic networks can eliminate any need for human control, creating a fully autonomous system. The culmination of these attributes leads to a much more robust and cost-effective system.

Since the emergence of swarm robotics, an abundance of applications have been suggested. Some of which include:

- Environmental monitoring,
- Mine detection,
- Herding,
- Mapping.

Such applications can be implemented using swarms in a variety of ways. For example, each swarm agent may use behavioral, control, consensus, and mathematical algorithms or any combination of these algorithms. This research plans to experiment with a mathematical Voronoi tessellation algorithm to monitor the environment and distribute each agent in specified patterns.

1.4 Wireless Sensor Networks

The idea of using a swarm of simple robots in place of a single sophisticated robot offers many advantages in military, industrial, and commercial applications as discussed in sec. 1.3. With a group of sensors and actuators it is possible to sense and act on a distributed environment such as temperature, electromagnetic waves, or a cloud of toxic gas. By regularly updating these sensors and actuators, a closed-loop system, also known as cyber-physical systems (CPS), can also characterize and track dynamic environments.

These sensors and actuators must be networked together in order to coordinate with each other to monitor and control the environment. This network is called wireless sensor network (WSN). A WSN is comprised of nodes which typically include sensors and/or actuators, microprocessors, and wireless communication modules [2]. Nodes in a WSN are designed to have low-power consumption, low-cost, mesh networking, and low-data throughput. Many communication protocols are designed specifically for WSNs which emphasize power efficiency, for longer battery life, and are less concerned with bandwidth use. Some current application examples of WSNs include:

- Industrial control and monitoring,
- Security monitoring,
- Agriculture and field observation,
- Patient monitoring.

More information on WSN can be found in *Wireless Sensor Networks: Architectures and Protocols* [5]. For this thesis, a group of heterogeneous nodes will communicate in a WSN to calculate Voronoi tessellations.

1.5 Voronoi Tessellations

A Voronoi tessellation refers to a region, containing p generating points, separated into cells where each cell contains one generating point and every point in the cell is closest to its generating point [6]. Voronoi tessellations are mathematically defined as follows.

Given a region $\Omega \in \mathbb{R}^N$and a set of generating points $\{p_i\}_{i=1}^{k} \subset \Omega$, let the Voronoi cell V_i corresponding to the generator p_i be

$$V_i = \{\ q \in \boldsymbol{\Omega} |\ |q - p_i| < |q - p_j| \quad j = 1, \ldots, k,\ j \neq i\} \tag{1.1}$$

$$i = 1, \ldots, k,$$

where the set of Voronoi cells $\{V_i\}_{i=1}^{k}$ creates a Voronoi tessellation on Ω. While the Euclidean norm $|q - p_i|$ is defined as

$$|q - p_i| = \sqrt{(q_1 - p_{i1})^2 + (q_2 - p_{i2})^2 + \ldots + (q_N - p_{iN})^2}. \tag{1.2}$$

Equation (1.1) simply compares the distance between points on the region, q, and generators, p. If a point q is closest to the generator p_i, then that point belongs to the Voronoi cell V_i. This concept can be used to create a discrete Voronoi tessellation. An example of Voronoi tessellations created by a random distribution of points can be seen in fig. 1.2. Notice that each generator's cell is only affected by its nearest neighbors. Therefore, in order to construct any Voronoi tessellation, each generator should only be aware of its nearest neighbors.

1.5.1 Centroidal Voronoi Tessellations

Given a density over the region of interest, a CVT is a Voronoi tessellation in which the generators are the centroids (centers of mass according to the given density) of its

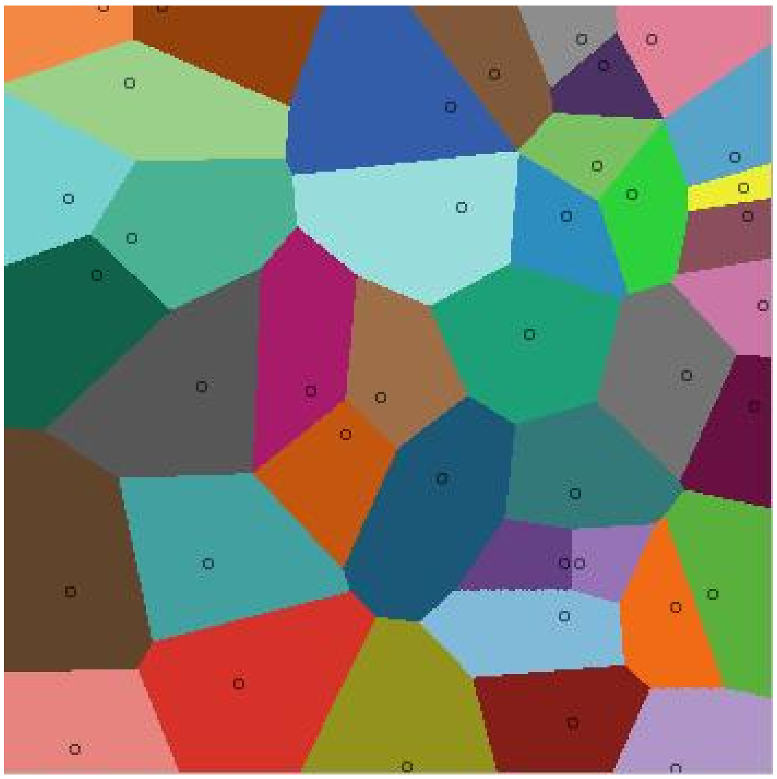


Fig. 1.2: Random Voronoi tessellation created in NetLogo, a multi-agent simulation software.

corresponding cell [7]. The mathematical definition is as follows.

Given a density function $\rho(q) \geq 0$ defined on Ω, the mass centroid p_i^* for each Voronoi cell V_i is given by

$$p_i^* = \frac{\int_{V_i} \rho(q) q \mathrm{d}q}{\int_{V_i} \rho(q) \mathrm{d}q} \text{ for } i = 1, \ldots, k, \tag{1.3}$$

where q is a point in the cell V_i. The discrete version is given by

$$p_i^* = \frac{\sum_{V_i} \sum_{j=1}^{N} \rho_j q_j}{\sum_{V_i} \sum_{j=1}^{N} \rho_j} \text{ for } i = 1, \ldots, k, \tag{1.4}$$

where N is the number of sampled points in the Voronoi cell V_i. The tessellation constructed by (1.3) is a centroidal Voronoi tessellation, provided that $\int_{V_i} \rho(q) \mathrm{d}q \geq 0$, if and only if

$$p_i = p_i^* \text{ for } i = 1, \ldots, k.$$

In other words, the points p_i are not only the generators for the Voronoi cells V_i, but also the mass centroids for those cells [8].

These diagrams are extremely useful in many fields for their geometric properties. CVTs typically create elegant diagrams where the concentration of generators can be controlled by the given density function. CVTs have many applications which include territorial animal behavior, image and data compression, multi-dimensional integration, partial differential equations, and optimal sensor and actuator locations [7]. Figure 1.3 shows a diagram of mouthbreeder fish territories with a centroidal Voronoi tessellation fit.

1.5.2 Energy Function

CVTs minimizes the energy function which is a useful property for many applications. The equation for the energy function is below.

$$\mathcal{H}_V(p) = \int_{V_i} |q - p_i|^2 \rho(q) \mathrm{d}q \text{ for } i = 1, \ldots, k \tag{1.5}$$

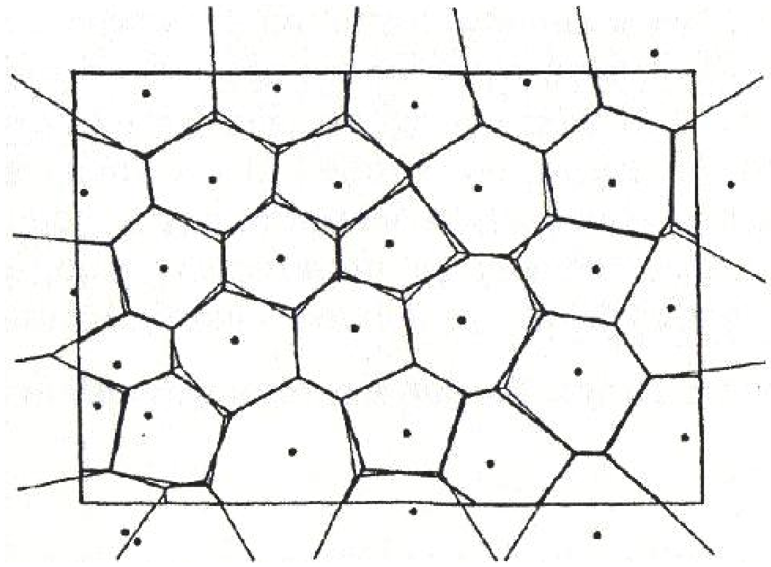


Fig. 1.3: Mouthbreeder territories with a Voronoi tessellation fit (from Spatial Tessellations: Concepts and Applications of Voronoi Diagrams).

The discrete version is given by

$$\mathcal{H}_V(p) = \sum_{V_i} \sum_{j=1}^{N} |q_j - p_i|^2 \rho_j, \tag{1.6}$$

which is also known as the variance, cost, or error function [7]. This energy function evaluates the location error of the generators, p_i, according to the density $\rho(q)$. The following proof shows that a CVT is a necessary condition to minimize the energy function in (1.5).

Proof. Evaluate a small variation on the generating point, p_j

$$\mathcal{H}_{V_j}(p_j + \epsilon) - \mathcal{H}_{V_j}(p_j) = \int_{V_i} \rho(q) \Big\{ |q - p_j - \epsilon|^2 - |q - p_j|^2 \Big\} \, \mathrm{d}q.$$

Expand and simplify the expression in brackets

$$\mathcal{H}_{V_j}(p_j + \epsilon) - \mathcal{H}_{V_j}(p_j) = \int_{V_j} \rho(q) \epsilon \left\{ \epsilon + 2(p_j - q) \right\} \mathrm{d}q.$$

Divide by ϵ

$$\frac{\mathcal{H}_{V_j}(p_j+\epsilon)-\mathcal{H}_{V_j}(p_j)}{\epsilon}=\int_{V_j}\rho(q)\{\epsilon+2(p_j-q)\}\,\mathrm{d}q.$$

Take the limit as $\epsilon \to 0$

$$\mathcal{H}^{'}_{V_j}(p_j)=2\int_{V_j}\rho(q)(p_j-q)\mathrm{d}q.$$

Now, distribute the terms

$$\mathcal{H}^{'}_{V_j}(p_j)=2p_j\int_{V_j}\rho(q)\mathrm{d}q-2\int_{V_j}q\rho(q)\mathrm{d}q.$$

For the minimum solution, the derivative must be zero. Therefore,

$$p_j=\frac{\int_{V_j}q\rho(q)\mathrm{d}q}{\int_{V_j}\rho(q)\mathrm{d}q},$$

which corresponds to the mass centroid equation (1.3) [9]. □

1.5.3 CVT Algorithms

CVT algorithms are typically used to locate static sources. However, if the source or distribution moves slower than the CVT convergence rate, dynamic tracking can be achieved. It follows that the speed of the target is limited to the convergence rate. Because the target is unknown, the convergence rate must be increased to track quicker moving targets.

Perhaps the most common and basic algorithm to construct discrete CVTs is the *Lloyd's algorithm.* This algorithm is a clear-cut iteration between building Voronoi tessellations and computing their centroids [10]. *Lloyd's algorithm* is described below [8].

Given a region Ω, a density function $\rho(x, y)$ defined for all $x \in \Omega$, and a positive integer k,

1. Select an initial set of k points $\{p_i\}_{i=1}^{k}$ as the generators;

2. Construct the Voronoi cells $\{V_i\}_{i=1}^{k}$ associated with the generators;

3. Find the mass centroid of each Voronoi cell. These centroids become the new set of generators;

4. If the new generating points meet a given convergence criterion, terminate; otherwise, return to step 2.

Lloyd's method requires few iterations, but each iteration is expensive to calculate the precise Voronoi tessellation and mass centroid. A second commonly used method for computing CVTs is the *MacQueen algorithm.* This algorithm does not require any precise construction of Voronoi tessellations or mass centroids; thus taking advantage of discrete CVTs. The *MacQueen's algorithm* is below [10].

Given a region Ω, a density function $\rho(x, y)$ defined for all $x \in \Omega$, and a positive integer k,

1. Select an initial set of k points $\{p_i\}_{i=1}^{k}$ as the generators; set the integer array $J_i = 1$ for $i = 1, \ldots, k$;

2. Pick a random point $q \in \Omega$;

3. Find the generator p_i closest to the point q; denote the index of that p_i as i^*;

4. Set $p_{i^*} \leftarrow \frac{J_{i^*} p_{i^*} + q}{J_{i^*} + 1}$ and $J_{i^*} \leftarrow J_{i^*} + 1$;

5. p_{i^*}, along with the unchanged points $\{p_i\}_{i=1, i \neq i^*}^{k}$ are the new set of generating points;

6. If the new generating points meet a given convergence criterion, terminate; otherwise, return to step 2.

Notice that the integer array J_i keeps track of the number of updates for p_i. Despite the absence of these calculations the algorithm still converges to a CVT [11]. However, each iteration only moves one generator and many iterations are needed for convergence. A combination of the few iterations of Lloyd's method and the cheap computation of the MacQueen's method can create a faster converging CVT algorithm for robots.

1.6 MAS2D

A new simulation platform called `MAS2D`, derived from `Diff-MAS2D` [12], is used to test these algorithms. `MAS2D` is designed to receive any moving or static distribution over the region $\Omega = [0,1]^2$.

Robots are modeled as particles by second order dynamics [13]

$$\ddot{p}_i = u_i, \tag{1.7}$$

where u_i is the control signal. To minimize the function in (1.5), the control law is set to follow a CVT

$$u_i = k_p(p_i - p_i^*) - k_d\dot{p}_i, \tag{1.8}$$

where p_i^* is the mass centroid of V_i, and both k_p and k_d are positive constants. The final term in (1.8) introduces viscous damping [14]. k_d is the damping coefficient and $\dot{p}_i$ is the velocity of robot i. This term eliminates possible oscillation as the robot approaches its destination. `MAS2D` and `Diff-MAS2D` act as the MASnet platform to observe robot behavior given certain static, dynamic, or diffusing densities.

Chapter 2
MASnet Platform

A WSN combined with a platform encapsulates the ongoing MASnet project that began in 2003 at CSOIS. Several robots, which act as the wireless sensor and actuator mobility nodes, can move on top of the MASnet platform. The general purpose for the platform is to study and research swarm engineering tasks such as formation building, environmental monitoring, and tracking. To keep the requirements of swarm engineering, the robots have limited communication and sensing capabilities. Despite restricted communication, robots are able to coordinate with each other to perform these tasks. A concept of the platform used for the this research is shown in fig. 2.1.

The MASnet system is made up of five to six elements.

1. 2.5 x 4 x 0.15 m Plexiglas® surface with wooden supports
2. Sensor array (optional)
3. Sensed element (fog, light, etc.)
4. Pseudo-GPS (pGPS) camera
5. MicaZ robots
6. Base-station

The robots execute commands, from the base-station, on the platform; the pGPS camera monitors the robots position; the camera information is displayed on the base station computer; and the base station sends commands back to the robots. The sensor array is used only for the CVT phototaxis experiment. In the phototaxis experiment, a mobile light source is held above the platform while the sensors, under the platform, measure the light distribution.

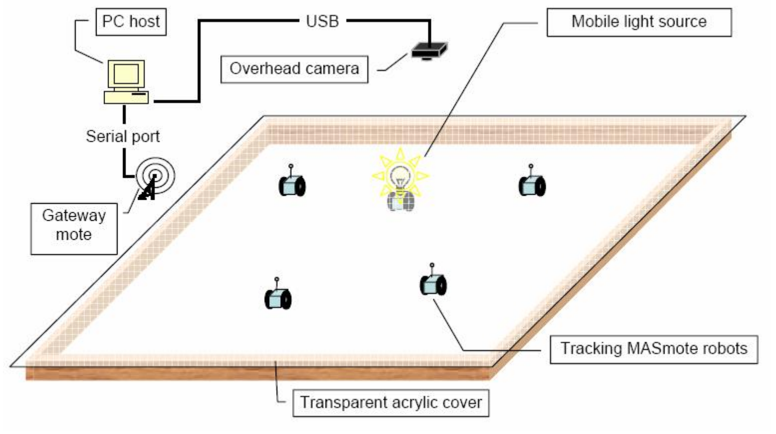


Fig. 2.1: Concept of the MASnet platform.

The base station functionality is coded in a program called RobotCommander written in C++ exclusively for MASnet. The platform is made of off-the-shelf products and open source software to keep the system flexible and low-cost. More details on MASnet platform development is described in PungYu Chen's and Zhongmin Wang's theses [2, 3, 15]. See fig. 2.2 for a picture of the actual MASnet platform.

2.1 Original MASmote Robots

Each robot is a small, two-wheel, differentially driven robot built of mainly commercial, off-the-shelf parts which can be easily redesigned; see fig. 2.3. The robots are assembled to be simple, compact, and inexpensive for swarm research. These robots are battery powered (using four AA batteries), self-contained, and can easily communicate with the base station and other robots. The dimensions of the robots are only 9.5 x 9.5 x 6.5 cm [16]. Red markers with individual patterns are placed on top of each robot for pGPS detection and identification.

Each robot is also equipped with the following parts:

- 1 MicaZ programming board,
- 2 servo motors,
- 2 encoders,

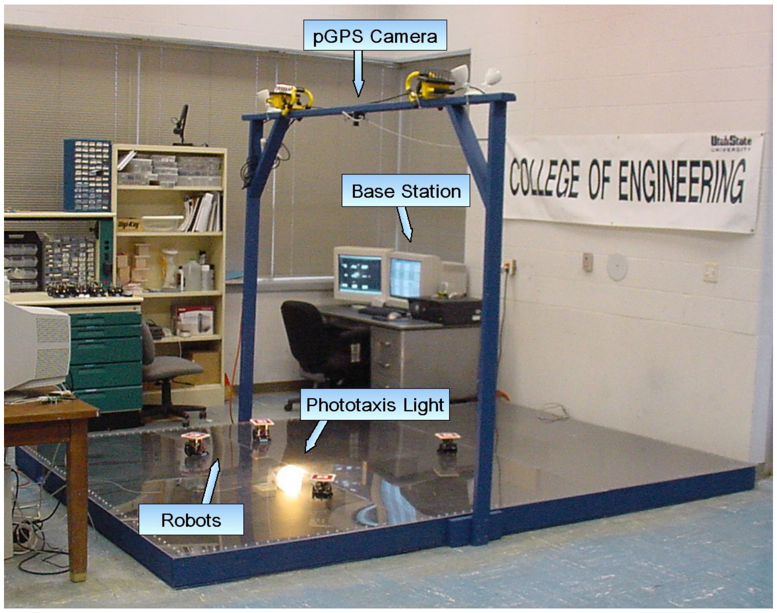


Fig. 2.2: MASnet platform.

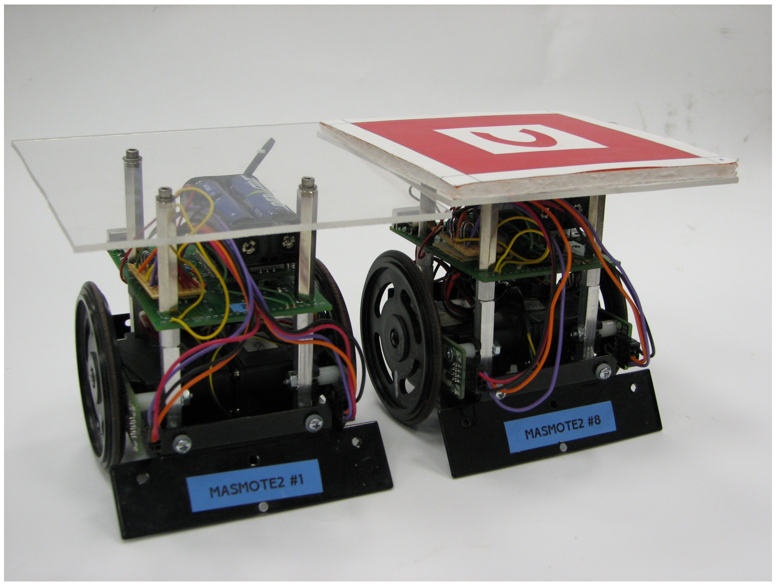


Fig. 2.3: MASnet robots (MASmotes).

- 2 photo-diodes,
- 2 IR sensors.

The next sections provide descriptions of the robot parts.

2.1.1 MicaZ Motes

The MicaZ programming board, or MicaZ mote, is developed by Crossbow for communication, sensing, and computation of the individual robots [17]. See fig. 2.4 for a picture of the 58 x 32 x 13 mm MicaZ mote.

The board is equipped with an 8 MHz ATmega 128L main CPU with 128KB programmable flash memory, 4KB EEPROM and 512KB flash memory to store sensor readings. It also has changeable pulse-width modulation outputs with eight 10-bit ADC channels. A CC2420 RF transceiver chip handles the wireless communication at 2.4 GHz with a maximum communication rate of 250 kbps.

The mote can be interfaced to a sensor board or a programming station by a 51-pin connector. The mote operates on a 3 V supply given by two AA batteries mounted on the mote. The mote is also equipped with three LEDs for status display and debugging purposes.

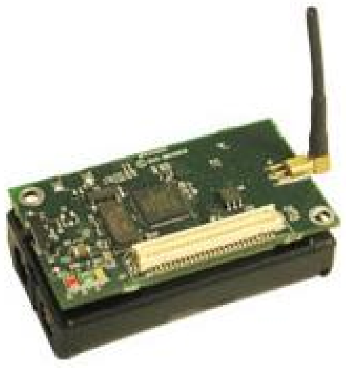


Fig. 2.4: MicaZ mote.

2.1.2 Servo Motors and Encoders

The robots are driven by two customized Futaba S9254 servo motors. These motors are controlled by a full duty cycle pulse-width modulated signal specified in the robot code. At 100% duty cycle, the robots can move at a maximum speed of approximately 57.6 cm/sec. The servo motors produce an estimated torque and speed of 47.2 oz.-in. and 0.06 sec/60, respectively, with an input voltage of 4.8 V [18].

Two high resolution encoders called Wheel Watchers from Nubotics [19] are used to calculate the position of the robots. These encoders have an angular resolution of 128 counts per revolution.

2.1.3 Photo-Diodes

OPT101 integrated circuits currently serve as the photo sensors in the MASnet project and return light intensity readings by 10-bit (4-digit) numbers [20]. Two sensors are placed on the front and back of the robots for sensing below the platform. See fig. 2.5 for a picture of the light sensors. Notice that the sensors are surrounded by black rubber tubing. The tubing minimizes interference from external light sources.

These sensors are chosen for their built-in transimpedance amplifier and monolithic photo-diode. Previous models of the MASnet robots include unreliable photoresistors that

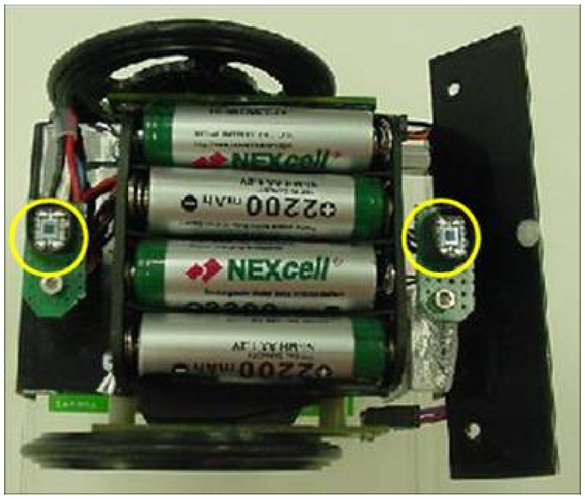


Fig. 2.5: Robot photo sensors (bottom view).

require additional op-amp feedback circuitry. These photoresistors respond sluggishly in which the output depends on the input voltage. Ultimately, it was decided to use a more consistent, accurate, and much less complicated photo-diode circuit. Characterization for the photo-diode circuit is presented in fig. 2.6. To test for a constant output voltage, the supplied voltage is varied with increments of 0.5 V while measuring a constant light source. From the plot, the output is fairly constant for input voltages above 3.7 V (instead of the 2.7 V given in the data sheet); however, because the robots cannot operate below 4 V, the photo sensors will not operate in the nonlinear region.

2.1.4 IR Sensors

The robots are also equipped with two IR sensors which can detect distances from other objects. The sensors are mounted in front of the robot to avoid colliding with each other and other nearby objects. See fig. 2.7 for the IR sensor locations.

These GP2D120 IR sensors made by Sharp return 10-bit (4-digit) numbers according to the distance from the object sensed. These sensors also have a maximum saturation range between 2.5 and 4 cm [21]. An experiment has been conducted to characterize the IR sensors with the weakest case scenario. Because IR sensors respond weakest to black objects, sensor readings are taken while approaching a black wall. See fig. 2.8 for the characterization results.

The robot approached the black wall in increments of 0.5 cm. For each increment,

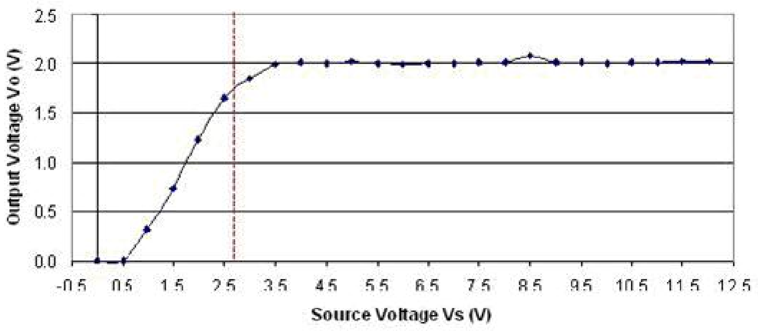
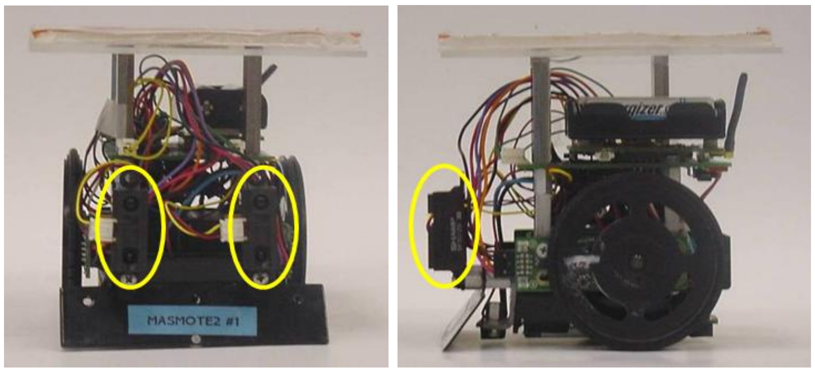


Fig. 2.6: Photo sensor characterization (courtesy of Heather Nelson).

several readings are recorded and the average is plotted in the figure. Observe that reliable readings occur beyond 4 cm and an object can be sensed as far as 50 cm, which is much larger than the 30 cm range specified in the data sheet.

2.2 New MASmote Robots (MASmote Gen-II)

MASnet has undergone many upgrades and changes during the course of the project. This section explains the evolution of the robot design on the platform, where 10 MASmotes, or robots, were originally designed for the Mica2 mote [16]. Once Crossbow released the MicaZ technology, four robots were upgraded in hardware and software to work with the MicaZ. Since then, these four robots have been altered for other MASnet experiments leaving them nonfunctional for CVT experiments.



(a) Front view (b) Side view

Fig. 2.7: Robot IR sensors.

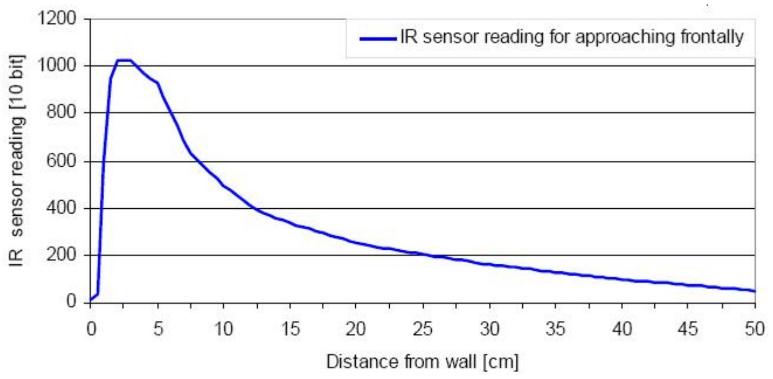


Fig. 2.8: IR sensor characterization (courtesy of Florian Zwetti).

Before any CVT experiments can be tested on the MASnet platform, all remaining robots need to be upgraded and designed to work with the MicaZ mote. Although there is substantial documentation on building original MASmotes [22], documentation on building MicaZ MASmotes was nonexistent before this project. Additionally, the physical design was messy and difficult to understand with poorly soldered components and floating protoboards, where shorts happened often.

2.2.1 Design Improvements

The old MASmotes have been updated and improved by reverse engineering the original MicaZ design. See Table 2.1 for a list of the parts needed to build a second generation MASmote from scratch. Most parts in the list are also used on the first generation MicaZ MASmotes. However, there are several fundamental differences in design between previous and current MASmotes designs. For clarification, the original robots designed for the Mica2 mote are called Mica2 MASmotes; the first robots designed for the MicaZ mote are the first generation MicaZ MASmotes; and the new robots designed here are the second generation MicaZ MASmotes.

Table 2.1: Parts list.

Part Number	**Qty.**	**Description**
R157-BLACK-ORING-WHEEL	2	servo driven wheels from Acroname Robotics [23]
R238-WW01-KIT	2	wheel encoder kits by Nubotics [19]
S03N-2BB	2	servo motors from GWS [24]
R146-GP2D120	2	IR proximity sensors from Sharp [21]
OPT101	2	photo-diodes from Burr-Brown [20]
MPR2400CA	1	MicaZ mote from Crossbow [17]
N/A	1	MARK III chassis kit from Junun [25]
495-1047-1-ND	2	$10nf$ capacitors from Digi-Key
P1318-ND	1	$1000\mu f$ capacitor from Digi-Key
N/A	1	protoboard
N/A	4	$2k\Omega$ resistors
N/A	4	$3.3k\Omega$ resistors

Battery Pack Attachment

The backbone of the design begins with the chassis which includes a scoop to balance the robot, a base to support the robot and battery pack, and a small Plexiglas® support to eventually mount a photo-diode. The MARK III comes with Velcro to attach the battery pack to the base. The Velcro was originally used for this purpose for the Mica2 MASmotes [22]. However, the Velcro attachment, deemed too weak and unreliable for the MASmotes, was replaced by bolts joining the battery pack to the base with 3.5 cm standoffs. See fig. 2.9 for an illustration.

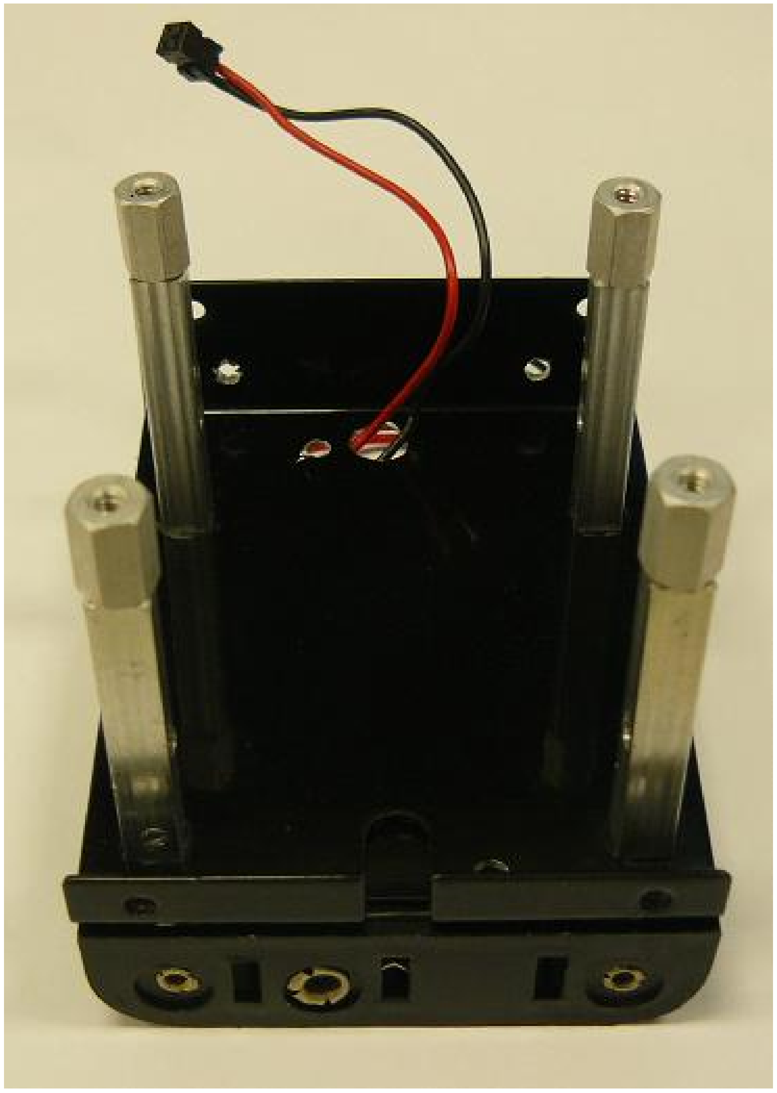


Fig. 2.9: Battery pack attached to the base.

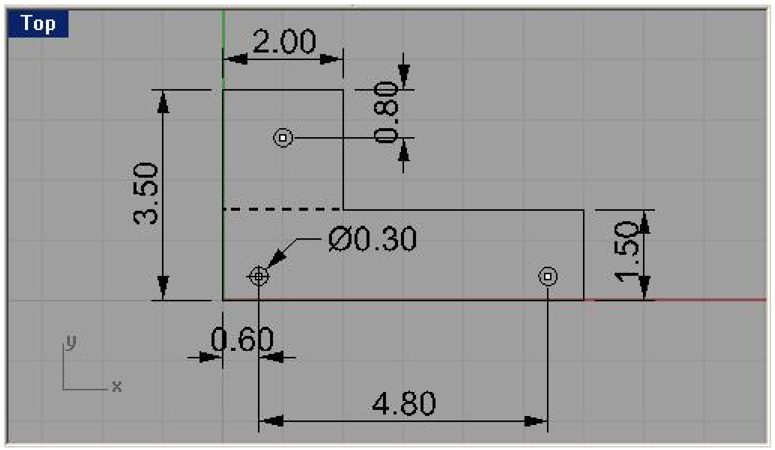


Fig. 2.10: Plexiglas® support dimensions (courtesy of Angel Cortes and Jordan Wirth).

Photo-Diode Plexiglas® Support

A Plexiglas® support holds the new photo-diode described in sec. 2.1.3 to the front of the robot. Figure 2.10 shows the dimensions of the Plexiglas®. A soldering iron softens the Plexiglas®, allowing the sheet to bend 90° along the dotted line in the figure. The top flap is bolted to a 1.5 x 0.5 in protoboard, which holds the photo-diode circuit while the bottom portion is attached to the chassis base.

Voltage Divider Mount

The first generation MASmote held a precarious protoboard of voltage dividers to limit the encoder input voltage. The board was not mounted to any part of the robot, the poor soldering caused sporadic shorts, and the unkempt wires made it extremely difficult to maintain. An example of a first generation voltage divider circuit can be seen in fig. 2.11(a).

The second generation MASmote design now has a mounted protoboard 6 x 2 cm. This board holds the voltage divider circuit and is mounted to the front of the MASmote with 1 cm standoffs. Observe fig. 2.11(b) for a picture of the mounted circuit. Compare the first and second generation voltage divider circuits and notice how the second generation is much more organized. The soldered connection is much more cohesive, the board is mounted and sturdy, and the circuit itself is more organized. Currently, there has been no shorts or loose wires with the new voltage divider mount.

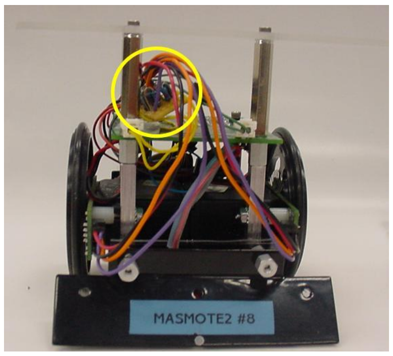
Servo Motors

The first generation MicaZ MASmote used servo motors from Futaba. The second generation simply "recycled" the GWS servo motors from old Mica2 MASmotes. A table comparing the size, weight, speed and torque of both motors is shown in Table 2.2. Notice that both motors are extremely similar with the exception of speed. The Futaba motor moves about four times faster than the GWS motor at 4.8 V.

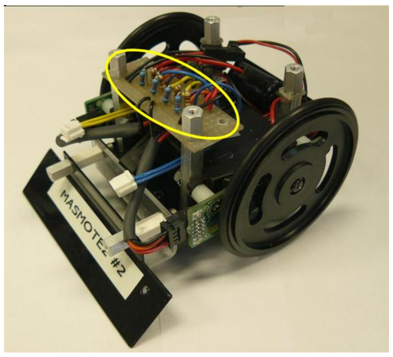
The higher speed may seem an advantage for the robots; however, experiments show that robots with the GWS motor are much more accurate and quickly converge to their desired locations without overshoot. Because of the latency of the GPS camera and receiving encoder information from the robots, faster motors tend to overshoot and oscillate around desired positions.

PCB

The printed circuit board connects all the robots electrical components (motors, encoders, photo-diodes, and IR sensors) to the MicaZ, which processes the data. Slight changes



(a) First generation MASmote voltage divider



(b) Second generation MASmote voltage divider

Fig. 2.11: First and second generation voltage divider circuits.

Table 2.2: Specifications for the GWS and Futaba motors.

Model	**Size**	**Weight**		**Speed @ 4.8V**	**Torque @ 4.8V**
	L x W x H mm	g	oz	sec/60°	oz-in
(GWS) S03N 2BB	40 x 20 x 40	41	1.44	0.23	47
(Futaba) S9254	41 x 20 x 36	49	1.7	0.06	47.2

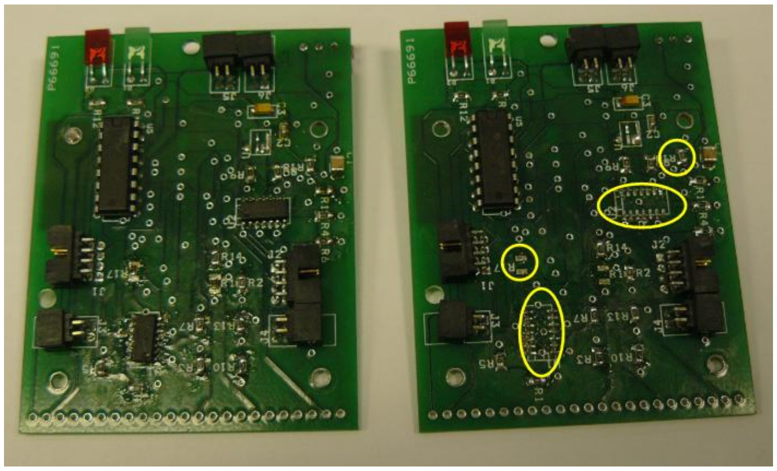


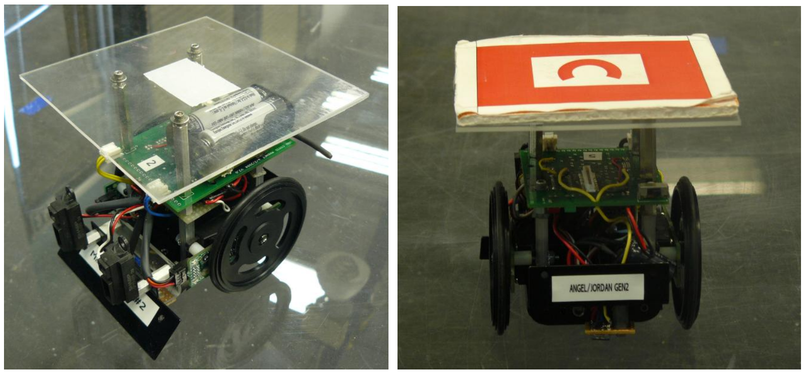
Fig. 2.12: Left: Mica2 PCB; Right: MicaZ PCB.

must also be made to the PCB to facilitate the changes in other parts and designs discussed in this chapter. A picture of the old Mica2 PCB and new MicaZ PCB can be seen in fig. 2.12. The yellow circles indicate the parts that should be taken out for the new PCB design. The modifications are simple; remove the two IC chips and resistors 17 and 18.

For convenience, wires connecting the PCB to robots components are separated by connectors. It is difficult to analyze and maintain the first generation MASmotes without having to solder and swim around wires. Separating the PCB by connectors makes it possible to disconnect parts quickly and easily with no soldering necessary.

2.2.2 Final Design

Once all the improvements and modification from sec. 2.2.1 are finished, the final robot can be seen in fig. 2.13. Compare this to the first generation MicaZ MASmotes in fig. 2.3. Notice that the new generation is much more organized. The new, more stable design also improves robot position accuracy and ease of maintenance where circuit shorts no longer occur. For complete documentation on building a second generation MASmote, refer to *Generation II MAS-Motes Construction Manual* [26].



(a) Perspective view (b) Back view

Fig. 2.13: Second generation MicaZ MASmote.

2.3 Software Description

2.3.1 TinyOS and nesC

The MASmote system uses TinyOS, an event-driven operating system designed for wireless sensor networks with limited memory. The TinyOS system can be valuable for multi-agent applications for its low coding and memory requirements, which can reduce time and cost for building individual agents and the system as a whole [27].

The TinyOS system is developed in nesC, a concurrent extension of C, created at the University of California at Berkley [17, 27–30]. This programming language is primarily used for embedded systems such as the WSN on the MASnet platform [29]. nesC simplifies accessing hardware through interrupts and low-level control. The programming language is attractive for projects which require executing several tasks for that very reason. Tmote, discussed in the next chapter, and MicaZ motes are both programmed with nesC.

2.3.2 RobotCommander

The base-station computer is programmed to read and process information coming from the base-station mote (gateway mote) and pGPS camera, and send commands through the gateway. A program written for this purpose, RobotCommander, is developed in C++ with Microsoft Foundation Classes (MFC) for a user-friendly GUI application. See fig. 2.14 for a screen capture of the RobotCommander GUI.

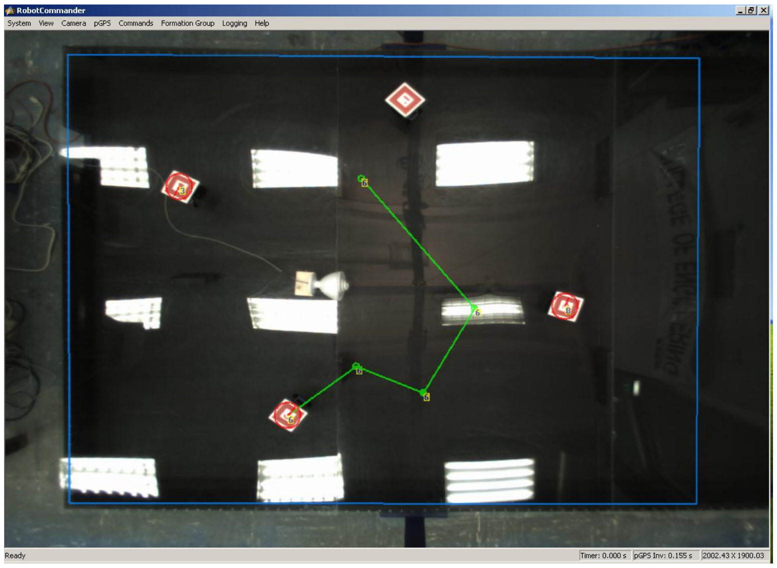


Fig. 2.14: RobotCommander GUI.

Image processing—which includes lens radial distortion compensation, robot marker detection, and screen rendering—occurs continually in the main application while all other functions are event driven [1]. The primary functions of RobotCommander are listed as follows [2].

- Real-Time Image Processing
  - Control camera and video stream
  - Capture and analyze pGPS images
  - Transform marker positions from pGPS image to the MASnet platform coordinates
- Communication
  - Receive messages from motes through the gateway
  - Send commands to robots
- Logging (i.e. robot positions, sensor readings, etc.)
- Providing a GUI

- Show pGPS image, robot, and communication information, etc.
- Issue commands

RobotCommander finds the position of each robot in the pGPS image by identifying unique robot markers. Each marker is assigned to a certain robot. See fig. 2.15 for four robots with their assigned markers. These markers are detected by a modified version of the AR-ToolKit [2,31], which detects the red frame first and then recognizes the robot by identifying the symbol. Once a robot is detected, its id, position, and angle information can be logged and broadcast to all robots.

The RobotCommander GUI shows the pGPS image and marks detected robots by red circles. Among other functions, the user can point and click on any place in the GUI image for robots to follow. These point-to-point commands are indicated by green circles in fig. 2.14. For information on operating the MASnet platform, refer to Appendix B. For a list of all files that comprise RobotCommander, see Appendix C.

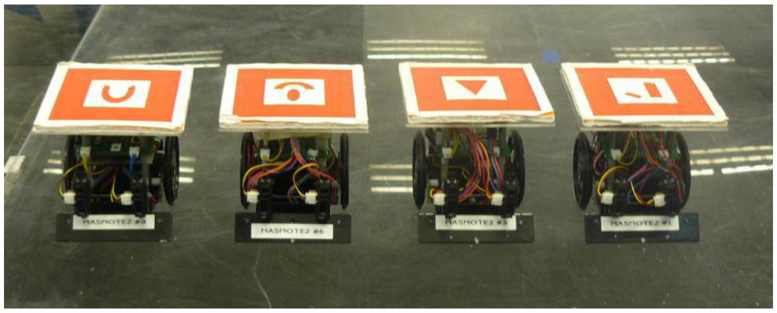


Fig. 2.15: Robots with red markers.

Chapter 3

Heterogeneous Swarm System

3.1 System Setup

This particular setup is used solely for the CVT phototaxis experiment which is discussed in detail in Chapter 4. For this experiment an array of at least nine light sensors are evenly distributed over the platform. These sensors measure light intensity at their discrete positions. Light data is sent through the radio to the base-station for CVT analysis. Meanwhile, at least four robots moving atop the platform periodically send encoder position information, and receive pGPS information and movement commands according to the RobotCommander CVT analysis.

3.1.1 Base-Station

As described in the setup, the base-station carries a heavy computation and communication load. At least 13 motes must communicate with the base-station simultaneously with multiple messages. In the future, the setup can be more decentralized, where the individual robots calculate their own CVTs. However, this experiment relies on the base-station for communication handling and CVT calculations. To perform all these tasks, the base-station is split up into three layers that run concurrently [22].

SerialForwarder passes messages from the gateway mote to the RobotCommander program and vice-versa.

Image Processing captures and analyzes the pGPS image for robot identification in RobotCommander.

Robot Analysis analyzes given data and initiates robot commands in RobotCommander.

Even considering these three divide-and-conquer layers, the gateway involved with the first layer must send and receive several messages in a short amount of time. If not handled carefully, congestion and packet-loss may occur often. To alive communication, instead of sending pGPS messages for each robot individually, RobotCommander packs the messages in an array, which is sent out periodically. The array is broadcast to all robots, but each robot only reads its own pGPS message and ignores the rest of the array. An additional remedy to packet-loss is controlling how often messages are sent from the base-station, robots, and light sensors. This technique is discussed in sec. 4.4.1.

3.1.2 Light Sensors

The base-station must receive wireless data from the robots and light sensors. Ten robots are available; however, light sensors must be provided. It has been decided to use Tmote Sky motes from Sentilla, previously Moteiv, for their on-board light sensors, compact size, and easy on-board USB programming. Unfortunately, Tmotes need to pass messages to the MicaZ gateway mote and cross-communication between these two motes has not yet been accomplished. This research introduces a heterogeneous swarm system consisting of first generation and second generation robots using MicaZ motes and Tmote Sky light sensors. The following sections describe the Tmote Sky module and how to setup cross-communication.

3.2 Tmote Sky

Tmote Sky is a low-power wireless mote designed specifically for WSNs. The Tmote Sky, produced by Sentilla, replaces the old Telos mote, produced by Crossbow. For compatibility with sensors, actuators, and other wireless mote modules, Tmote Sky is compliant with USB and IEEE 802.15.4 industry standards. This mote includes temperature, humidity, visible light, and IR light sensors with 16 pins for on-board expansion. These on-board sensors, with an option to expand, increase robustness and adaptability while decreasing mote cost. Tmote Sky also comes with TinyOS support for easy integration with wireless protocols and accessible open source software [32]. See fig. 3.1 for a front and back view of

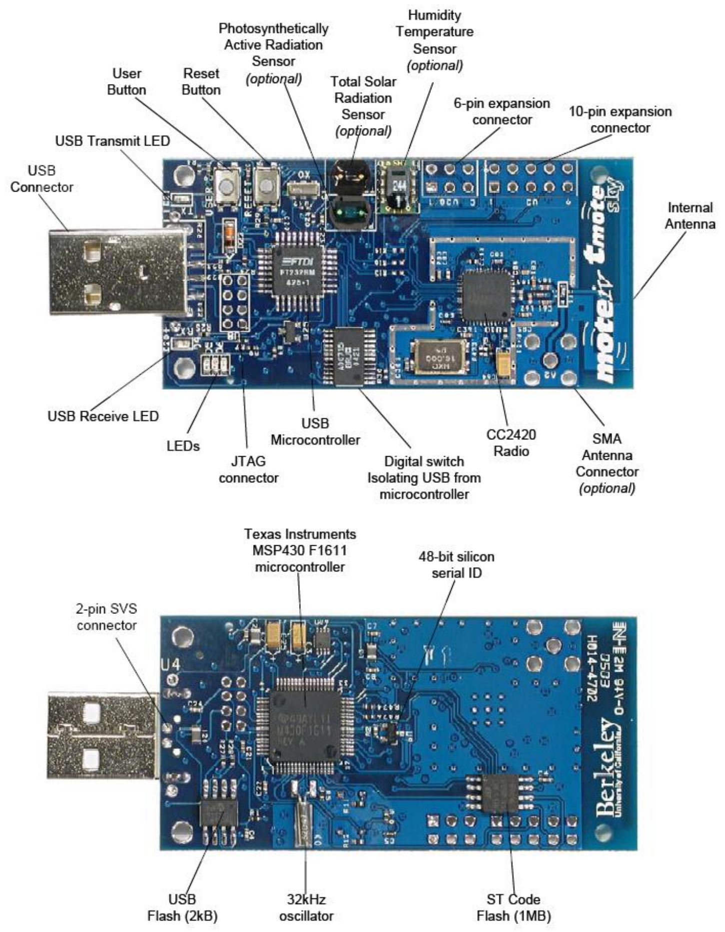

Humidity Temperature Sensor (optional)
Photosynthetically Active Radiation Sensor (optional)
User Button
Reset Button
Total Solar Radiation Sensor (optional)
6-pin expansion connector
10-pin expansion connector
USB Transmit LED
USB Connector
Internal Antenna
USB Receive LED
LEDs
JTAG connector
USB Microcontroller
Digital switch Isolating USB from microcontroller
CC2420 Radio
SMA Antenna Connector (optional)
Texas Instruments MSP430 F1611 microcontroller
48-bit silicon serial ID
2-pin SVS connector
USB Flash (2kB)
32kHz oscillator
ST Code Flash (1MB)

Fig. 3.1: Front and back of Tmote Sky (from Tmote Sky datasheet).

the Tmote Sky with labeled components. The nominal dimensions of the Tmote without battery pack and antenna are 1.26 x 2.58 x 0.26 in.

A list of Tmote Sky parts and features is given below [32].

- 250kbps 2.4GHz IEEE 802.15.4 Chipcon wireless transceiver
- Interoperability with other IEEE 802.15.4 devices
- 8MHz Texas Instruments MSP430 microcontroller (10k RAM, 48k Flash)
- Integrated ADC, DAC, supply voltage supervisor, and DMA controller
- Integrated on-board antenna with 50m range indoors / 125m range outdoors

- Integrated humidity, temperature, and light sensors
- Fast wakeup from sleep ($< 6\mu$s)
- Hardware link-layer encryption and authentication
- Programming and data collection via USB
- 16-pin expansion support and optional SMA antenna connector
- TinyOS support: mesh networking and communication implementation

The Tmote Sky operates between 2.1 and 3.6 V provided by two AA batteries. When connected to a computer, the Tmote operates on 3 V from that computer and does not require batteries. The programming language for the Tmote Sky is nesC 1.2 while the MicaZ is programmed with an older version, nesC 1.1. See *nesC 1.2 Language Reference Manual* for a list of changes between nesC 1.1 and 1.2 [28]. Once the programming is finished the Tmote can operate independent of any computer [33].

Currently, the Tmote Sky comes with two light sensors from the Hamamatsu Corporation [32]. The two photo-diodes are S1087 for sensing visible light spectrum (PAR sensor) and S1087-01 for sensing IR and visible light (TSR sensor). These sensors are surrounded by a ceramic package which blocks interference light entering the active area from the back or side for more accurate readings. A table of basic specification can be seen in Table 3.1 [34].

Any photo-diode can be used with the Tmote Sky as long as the dimensions are similar to the PAR and TSR sensors. However, the present sensors are sufficient for this research. It has been decided to use the less sensitive PAR sensor for current experiments on the MASnet platform. The PAR sensor can filter interfering light and concentrate on the light source more easily.

Table 3.1: Specifications for the PAR and TSR sensors.

Part Number	**Size**	**Spectral Response Range**	**Peak Sensitivity Wavelength**
	L x W x H mm	nm	nm
S1087 (PAR)	5 x 6 x 13.5	320 - 730	560
S1087-01 (TSR)	5 x 6 x 13.5	320 - 1100	960

3.3 MicaZ and Tmote Cross-Communication

The goal is for Tmotes to sense and send PAR readings while the MicaZ gateway mote receives PAR data for analysis. In order for this to occur, the MicaZ must understand the messages coming from the Tmotes. Although both MicaZs and Tmotes operate with TinyOS, their messages sent through the radio are fundamentally different. While operating on the same frequency channel, Tmote message bytes do not match MicaZ message bytes. The CRC check fails on the receiving MicaZ and the message is dropped.

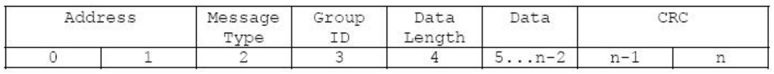
However, the Telos mote by Crossbow can communicate with the MicaZ as well as other Crossbow developed motes. Fortunately, because of similar design, Tmote Sky's can be programmed as Telos motes allowing cross-communication between MicaZs and Tmotes. The procedure is simple. While a Tmote is connected to a computer, in the Cygwin environment under the directory with the desired application, type the following.

```
make telosb install.[node number]
```

3.3.1 TinyOS Messages

TinyOS messages or packets that travel through the radio are called active messages. If motes operate on the same channel and protocol, but use different active message formats, they cannot communicate. The TinyOS message format implemented by Crossbow consists of an address, message type, group ID, data length, payload data, and CRC. See fig. 3.2 and Table 3.2 for a diagram, separated by bytes, and explanation of the TinyOS active message format.

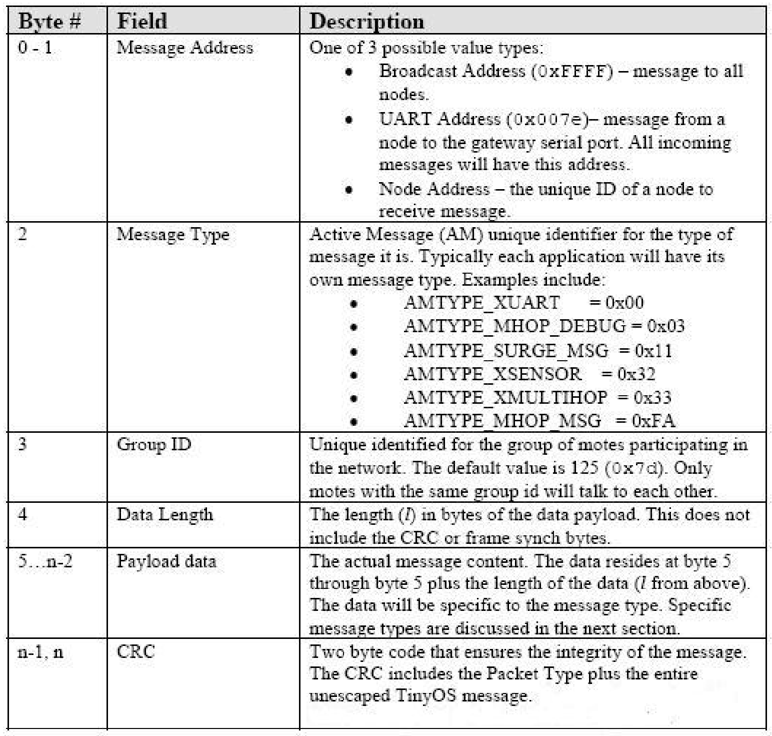
Other fields are also required in TinyOS messages, but these fields are not sent or received over the radio. They are only used internally for each individual node. These fields include signal strength (2 bytes), acknowledge signal (1 byte), and a timestamp (2



Address		Message Type	Group ID	Data Length	Data	CRC	
0	1	2	3	4	5...n-2	n-1	n

Fig. 3.2: TinyOS message format (from Octave Technology).

Table 3.2: TinyOS message description (from Octave Technology).



Byte #	Field	Description
0 - 1	Message Address	One of 3 possible value types: • Broadcast Address (0xFFFF) – message to all nodes. • UART Address (0x007e)– message from a node to the gateway serial port. All incoming messages will have this address. • Node Address – the unique ID of a node to receive message.
2	Message Type	Active Message (AM) unique identifier for the type of message it is. Typically each application will have its own message type. Examples include: • AMTYPE_XUART = 0x00 • AMTYPE_MHOP_DEBUG = 0x03 • AMTYPE_SURGE_MSG = 0x11 • AMTYPE_XSENSOR = 0x32 • AMTYPE_XMULTIHOP = 0x33 • AMTYPE_MHOP_MSG = 0xFA
3	Group ID	Unique identified for the group of motes participating in the network. The default value is 125 (0x7d). Only motes with the same group id will talk to each other.
4	Data Length	The length (*l*) in bytes of the data payload. This does not include the CRC or frame synch bytes.
5…n-2	Payload data	The actual message content. The data resides at byte 5 through byte 5 plus the length of the data (*l* from above). The data will be specific to the message type. Specific message types are discussed in the next section.
n-1, n	CRC	Two byte code that ensures the integrity of the message. The CRC includes the Packet Type plus the entire unescaped TinyOS message.

bytes) [35]. The purpose for the acknowledge signal, called "ack," is explained in the next section.

3.3.2 MAC Protocol

The MAC protocol that transmits and receives these active messages is the Carrier Sense Multiple Access—a contention based protocol designed to accommodate several nodes communicating on the same channel—with collision avoidance (CSMA/CA) [36, 37]. To send a packet, the transmitter checks to see if the receiver is free. If the channel is busy the packet is delayed and retries later; otherwise, the transmitter sends a preamble followed by the active message. The receiver, after detecting a preamble, accepts the active message and finally transmits an "ack" signal to inform the transmitter that the message was received. For an illustration of this procedure, see fig. 3.3.

There are several advantages in using the CSMA/CA protocol for this WSN. Networks can be easily scalable since resources are allocated on-demand. CSMA/CA is also much

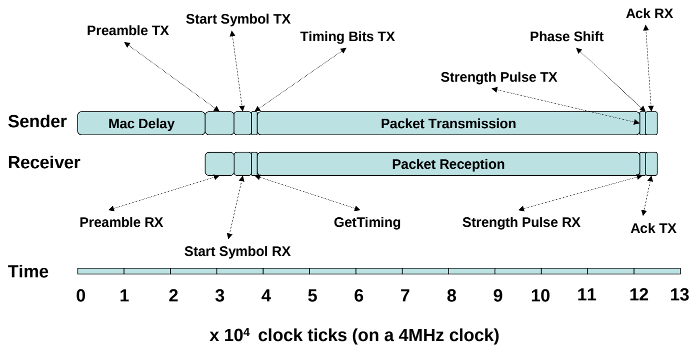


Fig. 3.3: Timing diagram of network send/receive (from Mica High Speed Radio Stack).

more flexible than scheduled protocols such as Time Division Multiple Access (TDMA) or Frequency Division Multiple Access (FDMA) because there are no communication clusters, or precise synchronization with time required, and direct node-to-node communication is possible [36]. This protocol has also influenced the development of many new WSN specific protocols. Some of these protocols include S-MAC [38], LPRT [39], B-MAC, RT-Link, MMSN, HyMAC [40], and SIP [41]. Optimizing the communication protocol is not the focus of this work but may be included in future work.

3.4 Implementation

The previous section explains a simple way for a MicaZ to receive Tmote data by the CSMA/CA protocol and answers how messages are structured. Tmote messages are then forwarded to RobotCommander in the first base-station layer; see sec. 3.1.1. This section shows specifically how a Tmote message can be understood in RobotCommander and presents code for Tmotes and RobotCommander to do so.

3.4.1 nesC

The original program used for the Tmotes was an altered application of “Oscilloscope” developed at UC Berkeley. This application gathered all sensor data—humidity, temperature, TSR, PAR, internal temperature, and internal voltage—by a state machine over four second intervals, but only sent the PAR reading over the radio. This program wastes power and time by gathering unnecessary readings through a state machine. Tmotes programmed with this application would stop sending messages within hours (with 100% duty cycle). A second application written to gather and send only PAR readings, which lasts an average of seven days (with 100% duty cycle), replaces the original application. The interval between sent messages is determined by one variable instead of a state machine.

This application includes an interface called “Oscope” which sends sensor data in a structured format. The definition of the Oscope interface can be found in the `.../opt/moteiv/tinyos-1.x/tos/lib/Oscope` folder in the Cygwin directory. An excerpt of Oscope.h is below.

```
enum
{
  OSCOPE_BUFFER_SIZE = 10,

  AM_OSCOPEMSG = 10,

};

typedef struct OscopeMsg
{
  uint16_t sourceMoteID;
  uint16_t lastSampleNumber;
  uint16_t channel;
  uint16_t data[OSCOPE_BUFFER_SIZE];
} OscopeMsg_t;
```

This `OscopeMsg` is the payload data sent wirelessly (see Table 3.2), where

`sourceMoteID` specifies the node number programmed for each mote.

`lastSampleNumber` timestamps when the last sample was taken.

`channel` specifies which sensor the data corresponds to (PAR, TSR, temperature, etc.).

`data` indicates an array of 10 sensor readings.

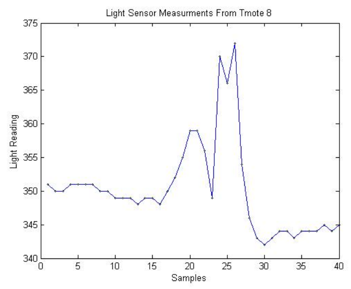
The message is sent as soon as ten sensor readings are gathered. Finally, the variable `AM_OSCOPEMSG` is the unique message type which is set to 10 (0x0A). As described in Table 3.2, the message type variable identifies what message will be received.

3.4.2 RobotCommander

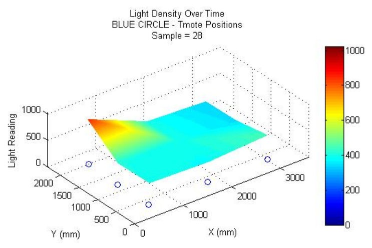
Once the Tmotes send reliable PAR sensor messages, RobotCommander must receive and analyze the data. First, the structure and message type are defined in a header file, such as the Oscope.h code above. For all messages coming into RobotCommander (robot sensor messages, pGPS, light readings, etc.) the message type is compared in a switch statement, which executes code for that certain message. For a Tmote PAR reading, the variables in the OscopeMsg structure are saved in an array for CVT calculation and displayed in the Communication Dialog box. This dialog box, for debugging purposes, shows the incoming and outgoing messages that are received/sent.

A Tmote logging function has also been added to the RobotCommander GUI that executes in the switch statement mentioned above. A second program written in MATLAB, called "Read Tmote," reads Tmote log files, produces light reading plots for each Tmote, and provides an animation of light intensity over time. Figs. 3.4(a) and 3.4(b) show a plot and screen capture from the program.

The animation is intended to simulate the MASnet platform. Light readings are in



(a)



(b)

Fig. 3.4: Plot and screen capture from Read Tmote program.

digital units with a maximum of 1024 and minimum of 0. Locations and coordinates match the actual platform. Because of the accuracy of the Read Tmote program, it is a powerful development tool for robot CVTs. For information on setting up the heterogeneous swarm system, refer to Appendix B.

Chapter 4
Multi-Robot Phototaxis Using Centroidal Voronoi Tessellations

There are many scenarios and techniques possible for controlling a distributed environment considering the sensors and actuators may be stationary or mobile, connected, or independent. Researchers have experimented with chemical plum tracking [42], dynamic diffussion [8, 43], boundary estimation [44], etc. However, these experiments solely use source information for convergence. In this project, an array of stationary sensors characterize the environment to build a centroidal Voronoi tessellation, while actuators move according to the tessellation. CVTs include nearest neighbor information, which make collision avoidance, cooperative control, and dynamic target tracking possible in a single algorithm. CVT is also a non-model based mathematical method that asymptotically converges to a field emitting source [45].

Previous work on CVTs briefly examines the effect of fewer sensors on convergence [3]. Exploring ways to decrease the number of sensors needed for convergence dramatically reduces cost and setup time for a sensor array. It is also impractical to have several hundred sensors setup in an indoor or relatively small outdoor setting. Fewer sensors can easily dissolve in the background, making a livable monitored environment. A new light estimation algorithm is introduced and included in the CVT algorithm to assure predictable robot convergence despite fewer sensors.

CVT-based taxis methods can be used in many applications such as chemotaxis, nuclear hazard detection, electromagnetic (EM) radio jammer localization, hot spot detection in forest fire mop-ups, etc. CVTs have not yet been applied to actual robots before to the author's best knowledge. All previous work appears in simulation or partial hardware application. This work advances swarm engineering and Voronoi tessellation algorithms as a whole by applying CVTs to actual robots on a real platform. This research also introduces the light estimation algorithm as a method to accommodate fewer sensors.

4.1 Swarm Design Approach

Global swarm behavior can be complex and volatile. Scientific procedures for swarm behavior have been introduced to break down the complicated process for much more predictable results. The first swarm procedure, proposed by S. Kazadi, includes two steps [46]:

1. Generate a swarm condition,
2. Fabricate behaviors that satisfy the swarm condition.

These steps are simple and practical. To create global swarm behavior, define the desired condition (foraging, flocking, etc.) and create behaviors to satisfy it. One may argue this procedure is too basic and does not help simplify the difficult swarm task.

For this project, a second, more detailed swarm design approach is used based on Kazadi's original procedure [47]:

1. Identify desired emergent behavior,
2. Select or devise a set of behaviors and motivators,
3. Choose appropriate input for the above behavior motivators,
4. Generate an algorithm to combine behavior motivators,
5. Simulate global behavior,
6. Apply global behavior to robots.

All steps in this process, with the exception of the first, depends on the previous steps. If one step does not work, the previous step should be modified for a successful swarm. This ladder dependency organizes a swarm design and helps identify problem areas in the design.

These procedures were originally created for behavioral algorithms. Mathematical models were previously believed to be too difficult to develop because of the complex interactions and unknown variables of global swarm behavior. Swarming requires general behaviors instead of the exact calculations mathematical models bring [47]. However, this project successfully follows the six-step swarm design approach using a mathematical non-model based algorithm without compromising general behaviors or complexity. The remainder of this chapter will explain the phototaxis CVT design through each step of the swarm design approach and compare results with a behavioral algorithm.

4.2 CVT Phototaxis Design

4.2.1 Identify Emergent Behavior

Step one of the design process defines the overall desired robot behavior. Examples of swarm behavior which can be combined to create an emergent behavior include [47]:

- Aggregation,
- Flocking,
- Foraging,
- Following,
- Grouping,
- Docking,
- Dispersion,
- Homing,
- Herding,
- Shoaling, etc.

A clearly defined task is needed to find required behavior motivators for the second step. The main goal of this project is to find and track a moving light source where the location is unknown. Specific tasks to achieve this goal are:

1. *Rendezvous* at a light source,
2. *Follow light gradient* towards light source,
3. *Collision avoidance.*

With these behaviors in mind selecting motivators is straightforward whether using behavioral or mathematical algorithms.

4.2.2 Select Behavior Motivators

Behavior motivators are algorithms that can be run by the individual agents or that allow each agent to react to local knowledge. Motivators constitute the individual agent's behavior and can react to a single or multiple inputs. A combination of motivators creates an emergent swarm behavior [47].

As mentioned in Chapter 1, the CVT algorithm has many behaviors "built in." For this project the robots act as the generators for CVTs. Because Voronoi cells are convex polygons that never overlap, one robot or generator cannot collide with another. This inherent property of CVTs covers collision avoidance [48, 49].

Also, if a region of interest is given a concentrated density, the CVT will cause the robots to aggregate or rendezvous toward the maximum peak of that density. During this aggregation the robots' paths follow the gradient of the density. This behavior satisfies the first and second motivator given the maximum of the density function is placed at the location of the light.

An additional observation shows that the robots can also converge to the light source simultaneously. The proof in sec. 1.5.2 shows that to minimize the energy function in (1.5) the generators simultaneously approach a CVT configuration. Granted, each generator may approach their CVT location at different rates depending on how far they are from that location and other generators. However, if the robot kinematics respond well enough to commands, a simultaneous rendezvous can be achieved. Considering these advantages to CVTs only one rendezvous motivator is needed for cooperative phototaxis.

4.2.3 Choose Inputs

Currently a CVT is calculated on the base-station computer that can communicate with all robots and sensors. In the future, a more decentralized design will be extremely beneficial. Specific inputs will change as the algorithm gradually becomes more decentralized. For now, inputs required for the rendezvous motivator with CVTs include:

1. Array of light readings,
2. Position at each light reading,
3. Position of each robot.

Light readings are gathered by wireless sensors. Robots positions are gathered by a combination of pGPS and encoders.

4.2.4 Phototaxis CVT Algorithm

CVT Algorithm for Robots

An array of sensors is placed evenly over a platform, which serves as the region of interest. The number of sensors is not a concern at this point. These sensors measure light density over the platform at their corresponding positions to construct the discrete density function $\rho(x, y)$. The light readings and robot positions, p_i, are gathered to build a Voronoi Tessellation. Note that the density function $\rho(x, y)$ is only sampled at certain points q_j, where $j = 1, \ldots, l$, and l is the total number of sensors. The combined Lloyd-MacQueen method for robots is as follows.

Given a region Ω, a density function $\rho(x, y)$ defined for all $x \in \Omega$, and positive integers k and l,

1. Select an initial set of k points $\{p_i\}_{i=1}^{k}$(robot starting positions) as the generators;
2. Select the sampled points $\{q_j\}_{j=1}^{l} \in \Omega$ where $\rho(q_j) = \rho(x_j, y_j)$;

3. Find the generator p_i closest to the point q_j for each sampled point; assign the set of points q_j with closest generator p_i to V_i (this builds the discrete approximation of the Voronoi cells$\{V_i\}_{i=1}^{k}$);

4. Find the discrete mass centroid of each Voronoi cell. These centroids become the new set of generators;

5. Give the robots command to move to the new generating points;

6. If the new generating points meet a given convergence criterion, terminate; otherwise, return to step 2.

Step three of this algorithm can be obtained by looping though sensor and robot data. This step is also the application of (1.1) which compares the distances between generators, p_i, and points, q to assign the Voronoi cell V_i. Notice that a single iteration of this new Lloyd-MacQueen method changes all generators p_i at once, as in the Lloyd method, while an explicit computation of Voronoi tessellations is not needed, much like the MacQueen method. The best of both algorithms are now combined for faster convergence.

Density Function

Before the CVT algorithm can be applied, a density $\rho(x, y)$ is required. Originally, the readings from the sensors served as the density. However, interference from other light sources and sensor noise caused sporadic results. Secondly, the light source could be too broad or asymmetrical for the robots to aggregate neatly around the source. Third, robots will follow unique light sources in different ways. Each light source has its own distribution characterized by intensity and drop-off rate.

The desired response requires robots to find and track any light source identically, regardless of the light characteristics. To accomplish this, the density is modeled after a Gaussian distribution. The center of the distribution, (x_c, y_c), occurs at the location of the light. Below is the Gaussian function

$$\rho(x, y) = c \exp^{-\sigma\left[(x-x_c)^2+(y-y_c)^2\right]}, \tag{4.1}$$

where c is the intensity of the Gaussian, and σ is the drop-off rate. A large σ is desired for concentrated densities. A plot of a Gaussian density can be seen in fig. 4.1 where $c = 1$, $\sigma = 500$, $x_c = 0.5$, and $y_c = 0.5$. The general Gaussian density in (4.1) achieves extremely robust rendezvous behavior. Results of this density can be seen in the following sections.

Light Estimation Algorithm

In order to plot the density function, the location of the light source must be known. Unfortunately, the number of sensors is limited and the source may not lie directly over any sensor. Therefore, a light estimation algorithm is needed to approximate the location of the light source. A precise characterization of the light is typically done to create an

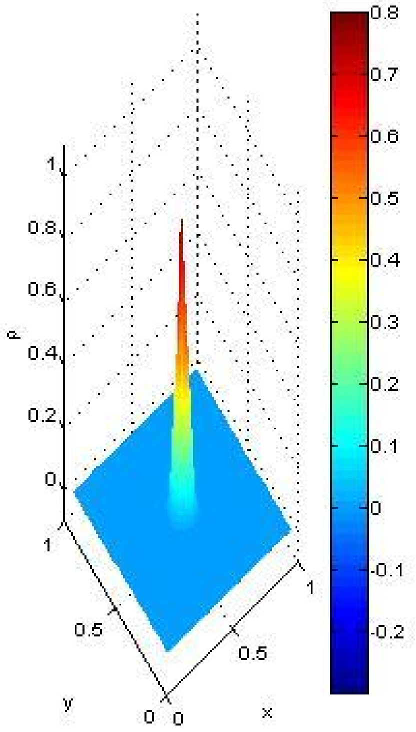


Fig. 4.1: Plot of the Gaussian function for rendezvous.

equation fit, piece-wise function, or zone intensity levels of the light distribution. However, such exact calculations are only useful for that particular light. Assuming the light source is unknown, the light estimation algorithm must find any light source within the region of interest.

This project introduces such an algorithm. Instead of characterizing the distribution of a specific light, consider the general intensity equation for the spreading of light [50],

$$i(x,y) = \frac{c}{[(x-x_c)^2 + (y-y_c)^2]^{\sigma/2}}. \tag{4.2}$$

Again, c is the intensity and σ is the drop-off rate. See fig. 4.2 for a plot of the light intensity equation (4.2) where $c = 1$, $\sigma = 0.3$, $x_c = 0.5$, and $y_c = 0.5$.

In this case, the intensity at each sensor reading, $\{i(x_j, y_j)\}_{j=1}^{l}$ is known. Through signal processing and least mean square techniques the intensity, c, the drop-off rate, σ,

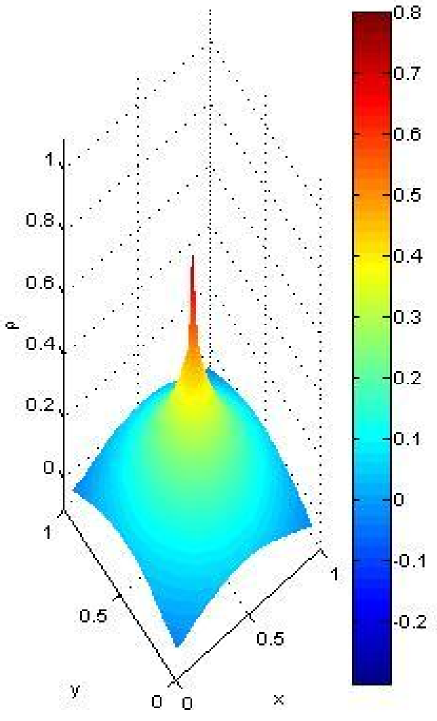


Fig. 4.2: Plot of the general light intensity distribution.

and the location of the light, (x_c, y_c) can be found. First, take the log of the intensity equation

$$\lambda(x,y) = \log c - \frac{1}{2}\sigma \log\left[(x-x_c)^2 + (y-y_c)^2\right],$$

$$\text{where } \lambda = \log(i).$$

Combine all known sample readings in matrix form

$$\begin{bmatrix} \lambda(x_1,y_1) \\ \lambda(x_2,y_2) \\ \vdots \\ \lambda(x_l,y_l) \end{bmatrix} = \begin{bmatrix} 1 & -\frac{1}{2}\log\left[(x_1-x_c)^2+(y_1-y_c)^2\right] \\ 1 & -\frac{1}{2}\log\left[(x_2-x_c)^2+(y_2-y_c)^2\right] \\ \vdots & \vdots \\ 1 & -\frac{1}{2}\log\left[(x_l-x_c)^2+(y_l-y_c)^2\right] \end{bmatrix} \begin{bmatrix} \log c \\ \sigma \end{bmatrix}$$

$$\Rightarrow \bar{\lambda} = \mathbf{A}(q_c)\bar{\mathbf{b}}(c,\sigma),$$

$$\text{where } q_c = (x_c, y_c).$$

Because we are dealing with measured values, noise and interference are introduced and the squared error $\left\|\bar{\lambda} - \hat{\mathbf{A}}(q_c)\hat{\bar{\mathbf{b}}}(c,\sigma)\right\|^2$, where a hat indicates an estimated matrix or vector, must be minimized:

$$\min_{\hat{\mathbf{b}},q_c} \left\|\bar{\lambda} - \hat{\mathbf{A}}(q_c)\hat{\bar{\mathbf{b}}}(c,\sigma)\right\|^2 = \min_{\hat{q}_c}\left(\bar{\lambda}^T\mathbf{P}(q_c)\bar{\lambda}\right) = \min_{\hat{q}_c}\mathbf{E}(q_c), \tag{4.3}$$

$$\begin{aligned} \text{where} \quad & \mathbf{P}(\hat{q}_c) = \mathbf{I} - \hat{\mathbf{A}}(\hat{\mathbf{A}}^T\hat{\mathbf{A}})^{-1}\hat{\mathbf{A}}^T = \mathbf{I} - \hat{\mathbf{A}}\left(\hat{\mathbf{A}}^{-1^*}\right), \\ \text{and} \quad & \hat{\mathbf{A}}^{-1^*} \text{ is the pseudoinverse of } \hat{\mathbf{A}}. \end{aligned}$$

The right hand side of (4.3) finds the x and y positions where $\mathbf{E}(q_c)$ is an absolute minimum. The minimum value can be found by iterating through x and y positions and calculating $\mathbf{E}(q_c)$ for each iteration. Finally, light intensity c and drop-off rate σ are derived from the matrix $\hat{\mathbf{A}}$ associated with the minimum value

$$\hat{\mathbf{b}} = \begin{bmatrix} \log \hat{c} \\ \hat{\sigma} \end{bmatrix} = \left(\hat{\mathbf{A}}^{-1^*}\right) \bar{\lambda}. \tag{4.4}$$

Because the phototaxis project only requires an estimation of the location of the light (x_c, y_c), (4.4) is not used, but can be extremely useful in identifying different light sources.

Unfortunately, iterating through the entire region of interest may take too much time. The algorithm can be much faster if only a fraction of all x and y positions are evaluated. A recursive light position estimation algorithm helps reduce the number of iterations needed. The algorithm focuses or "zooms in" on the critical area and ignores the rest of the region.

Given a region Ω, an array of light intensity readings, $\{i_j\}_{j=1}^{l}$ and position at each reading q_j,

1. Find the log of each reading, $\lambda_j(q) = \log\left[i_j(q)\right];$
2. Begin with a large step size Δ and *critical area* Ω;
3. Iterate through x and y positions by step size Δ over the *critical area* to find the minimum $\mathbf{E}(q)$;
   (a) $\hat{\mathbf{A}} = \begin{bmatrix} 1 & -\frac{1}{2}\log\left[(x_1-x)^2+(y_1-y)^2\right] \\ 1 & -\frac{1}{2}\log\left[(x_2-x)^2+(y_2-y)^2\right] \\ \vdots & \vdots \\ 1 & -\frac{1}{2}\log\left[(x_l-x)^2+(y_l-y)^2\right] \end{bmatrix};$
   (b) $\mathbf{P} = \mathbf{I} - \hat{\mathbf{A}}(\hat{\mathbf{A}}^T\hat{\mathbf{A}})^{-1}\hat{\mathbf{A}}^T = \mathbf{I} - \hat{\mathbf{A}}\left(\hat{\mathbf{A}}^{-1^*}\right);$
   (c) $\mathbf{E}(q) = \min\left(\bar{\lambda}^T\mathbf{P}\bar{\lambda}\right);$
   (d) $\Delta \leftarrow \frac{\Delta}{10}$; *critical area* $\leftarrow q \pm 5\Delta$;
   (e) If step size Δ reached the minimum step, continue to step 4; otherwise, repeat step 3.
4. Return the estimated light position $q_c \leftarrow q$.

By combining light location estimation, the Gaussian density function and the basic CVT algorithm for robots a successful cooperative phototaxis can be achieved. To test the performance of these algorithms, they are first placed in a simulated environment. Once kinks are solved in simulation, the phototaxis algorithm is ready to apply to physical robots.

4.3 Simulation

Simulation is an important step for swarm design that allows the designer to study and adjust agent behaviors relatively quickly compared to hardware implementation. Simulation can also be used to differentiate between algorithm mistakes and hardware problems once in the hardware phase.

4.3.1 Results

In this simulation nine static sensors are evenly placed over Ω to show phototaxis performance with a limited amount of sensors. Four robots are also evenly placed over the Ω. The control law for the robots is set to

$$u_i = 3(p_i - p_i^*) - 3\dot{p}_i.$$

The light source is placed at the bottom left corner $(0, 0)$ with the distribution:

$$i(x, y) = \frac{1}{(x^2 + y^2)^{0.3/2}}.$$

The Gaussian density function for CVT rendezvous is set to

$$\rho(x, y) = \exp^{-100\left[(x-x_c)^2+(y-y_c)^2\right]}.$$

After the center of the light source is calculated, fig. 4.3 shows the Gaussian density with the center of the estimated light location at $(0, 0)$.

The time step is set to 0.05 seconds. Robots compute desired positions every 0.2 seconds. Progression of the simulation is shown in fig. 4.4. The X's indicate sensors, O's

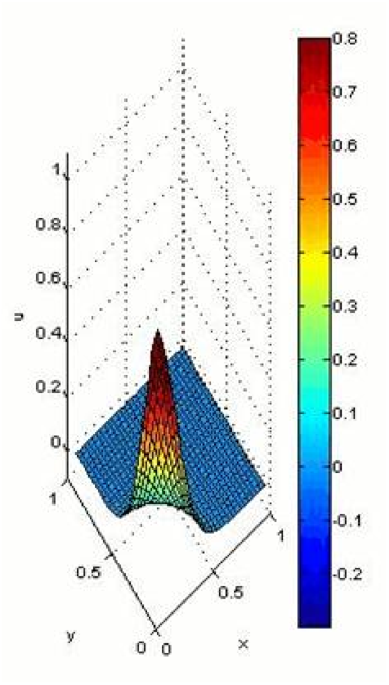


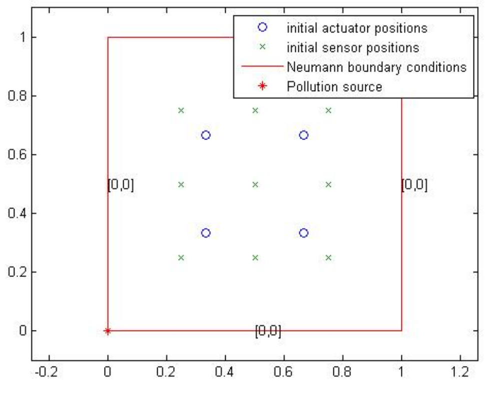
Fig. 4.3: Plot of the Gaussian function with estimated light location $(0, 0)$.

indicate robot paths and the red asterisk indicates the estimated position of the light source. Notice how the robots drive toward the source while keeping their square formation. This is achieved by the nature of CVTs; no formation control consensus algorithms are used. The robots converge to a CVT and arrive simultaneously at the source after 5 seconds. Similar behaviors occur at different light locations and with dynamic light sources. For the robots to gather closer to the light, the Gaussian density parameter σ should be increased.

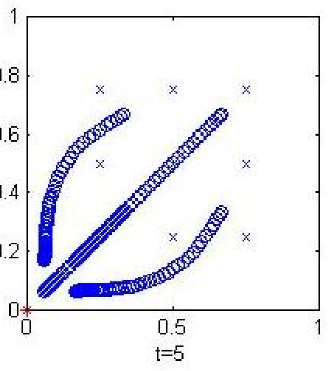
4.3.2 Proof of Convergence

`MAS2D` and `Diff-MAS2D` assume the passive, second order dynamics described in (1.7) with the control law in (1.8). The control input preserving the zero dynamics ($\dot{p}_i = 0$) is also assumed to be $u_i = 0$. The desired result is for the robots to converge to a CVT. In other words, the robot positions must approach their mass centroids

$$p_i \rightarrow p_i^*.$$



(a) Initial postitions



(b) Robot motion

Fig. 4.4: Progression of simulated robots for phototaxis.

With this in mind, position control laws which should approach a CVT can be considered a tracking problem where p_i tracks p_i^*. Lyapunov functions provide proof of convergence for such tracking problems.

Proposition 4.3.1. *For the closed-loop system in equation (1.8), the robots converge asymptotically to a particular centroidal Voronoi tessellation, assuming the tessellation set is finite.*

Proof. For convenience, the control law in equation (1.8) is rewritten here

$$\ddot{p}_i = u_i = k_p(p_i - p_i^*) - k_d\dot{p}_i.$$

Consider the following Lyapunov function for the control law

$$V(p_i) = \frac{1}{2}k_p \left\|p_i - p_i^*\right\|^2 + \frac{1}{2}\dot{p}_i^2.$$

Find the derivative of the Lyapunov function

$$\dot{V}(p_i) = k_p(p_i - p_i^*)\dot{p}_i + \dot{p}_i\ddot{p}_i.$$

Factor out $\dot{p}_i$ and substitute the control law for $\ddot{p}_i$

$$\dot{V}(p_i) = \dot{p}_i \left[k_p(p_i - p_i^*) - k_p(p_i - p_i^*) - k_d\dot{p}_i\right].$$

Cancel terms and simplify the expression

$$\dot{V}(p_i) = -k_d\dot{p}_i^2 \leq 0.$$

Therefore, the Lyapunov derivative is negative semi-definite. However, $\dot{V}(p_i) = 0$ if and only if $\dot{p}_i = 0$. Now, examine the control law in this special case given the zero dynamics assumption

$$u_i = 0 = k_p(p_i - p_i^*)$$

$$\Rightarrow p_i = p_i^*.$$

Therefore, by LaSalle's principle, the robots asymptotically converge to a specific centroidal Voronoi tessellation ($p_i = p_i^*$) if the tessellation set is finite. □

4.4 Robot Implementation

Now that phototaxis works in simulation, the algorithm is ready to be tested on physical robots.

4.4.1 MASnet Challenges

The transition from simulation to hardware can be challenging considering the imperfect responses of the robots and heavy communication traffic. Remember, the robots are cheap and each one has its own idiosyncrasies while nine Tmotes and four robots are communicating simultaneously with the base-station. Problems of packet loss and processor lock-ups can occur. Optimizing controllable variables can lessen, if not eliminate these problems. A discussion of these challenges follows.

First, it was difficult to find an adequately concentrated Gaussian without returning a zero density in at least one Voronoi cell. Each cell must have a density greater than zero to find the mass centroid. Several numbers were tested for c and σ from the Gaussian equation (4.1). The optimal response occurs when $c = 1024$, which is the maximum light reading and $\sigma = 10^{-5}$, where the region of interest is 3000 x 2300 mm. The mass centroid can be calculated and the robots converge inches from the light source.

Convergence rate was also a problem on the platform. CVTs can be calculated several times a second, but the light estimation took approximately 1 minute when each x and y position was evaluated with a step size of 10 mm. This algorithm was replaced with the recursive algorithm which can estimate the light in less than a second, beginning with a step size of 1000 mm.

Once the entire CVT algorithm could be calculated within a second, several mass centroid commands had to be sent to the robots every minute. Frequent CVT recalculation is needed for dynamic environments. Unfortunately, too much communication caused packet-loss and processor freezes. The robots could not move or send and receive any data. After experimenting, the optimal rate for CVTs without impairing the four robots is 2 seconds.

4.4.2 Results

Nine experiments were conducted with a stationary light source at different locations on the platform. For these experiments, an incandescent light was held above the platform by hand. The initial configuration is setup similar to the simulation. See fig. 4.5(a) for the initial startup from RobotCommander. Robots are indicated by red circles, Tmotes are indicated by teal squares, the region of interest is indicated by a blue rectangle, and the light sits on the top right corner. The average time it takes to surround the light source is 13.5 seconds. See Table 4.1 for a table of average convergence times according to four different light locations.

Table 4.1: Average convergence time according to light locations.

Location	Middle	Corners	Edges	Overall
Convergence Time	10 s	18.5 s	14 s	13.5 s

Similar convergence results occur for dynamic CVTs provided the light does not move faster than the algorithm reconfigures. Screen shots of one particular experiment can be seen in fig. 4.5. The upper left robot was intentionally impaired to show the robust nature of the algorithm. It is difficult to tell the exact location of the light from the image because of the pGPS camera's inherent distorted view. However, phototaxis behavior can still be observed. The phototaxis CVT algorithm is also easily scalable. Robots can be added at any time and include themselves in the surrounding group. Visit the YouTube channel `http://www.youtube.com/user/MASnetPlatform` for videos of static and dynamic CVT-based phototaxis experiments on the MASnet platform. The nesC code for CVT-based phototaxis can be found in Appendix A.

4.4.3 Proof of Convergence

Unfortunately, robots on the MASnet platform do not use a position control law and a Lyapunov function cannot prove convergence. Instead, movement is controlled by heading. Therefore, a general mathematical proof will show that applications using any variation of the Lloyd algorithm to compute CVTs—any algorithm that iterates between computing Voronoi tessellations and moving generators—converges to the desired tessellation.

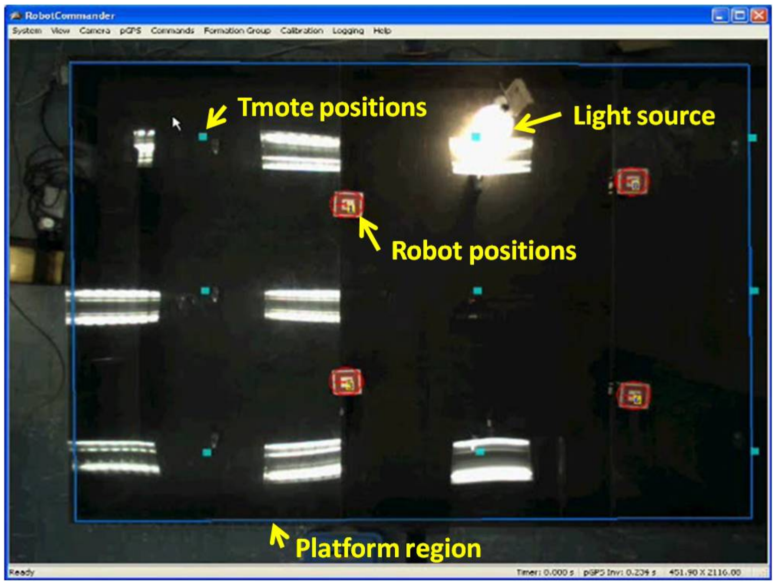
For clarity, the energy function in (1.5) is redefined as a function of the pair $(\mathbf{Q}, \mathbf{P})$:

$$\mathcal{H}_{\mathcal{V}}(\mathbf{Q}, \mathbf{P}) = \sum_{i=1}^{k} \int_{V_i(\mathbf{Q})} |q - p_i|^2 \rho(q) \mathrm{d}q,$$

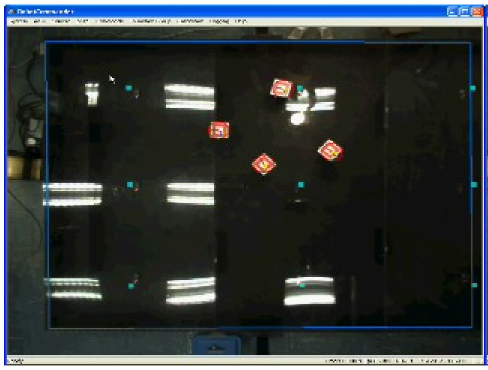
where $\mathbf{Q}$ is the set of any points within each cell, $\mathbf{Q} = \{q_i\}_{i=1}^{k}$; $\{V_i(\mathbf{Q})\}_{i=1}^{k}$ are the Voronoi regions with respect to the points $\{q_i\}_{i=1}^{k}$; and $\mathbf{P}$ is the set of distinct generators. The set of mass centroids is defined as $\mathbf{T}(\mathbf{P}) = \{p_i^*\}_{i=1}^{k}$.

Some properties of the energy function, discussed in *Convergence of the Lloyd algorithm for computing centroidal Voronoi tessellations*, must be described before the proof [51].

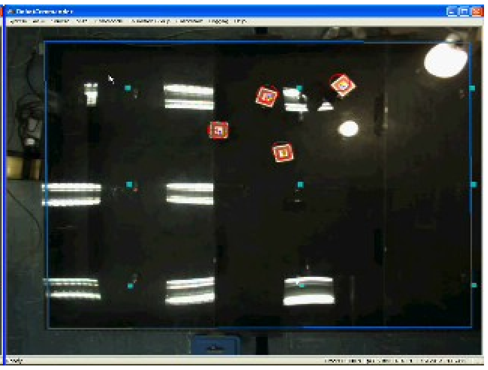
Lemma 4.4.1. *Let ρ be a positive and smooth density function defined on a smooth bounded domain Ω. Then*



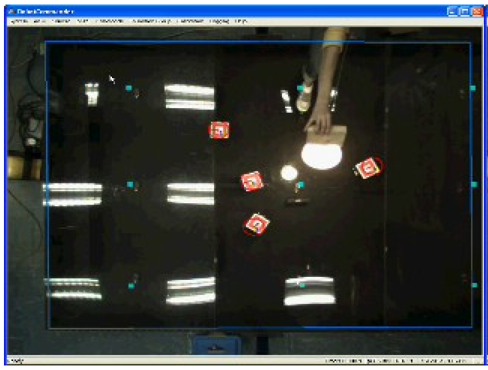
(a) Initial positions



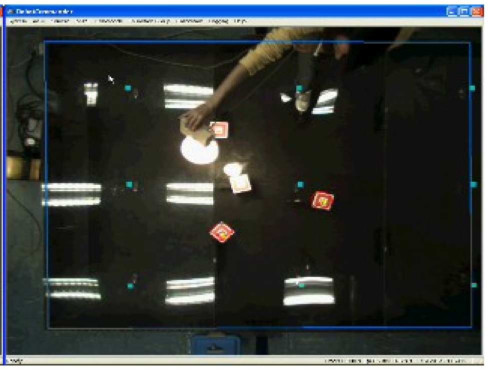
(b) $t = 14s$



(c) $t = 36s$



(d) $t = 54s$



(e) $t = 58s$

Fig. 4.5: Progression of MASnet robots for phototaxis.

1. *$\mathcal{H}$ is continuous and differentiable in $\bar{\Omega}^k \times \bar{\Omega}^k$;*

2. $\mathcal{H}(\mathbf{P}, \mathbf{T}(\mathbf{P})) = \min_{\mathbf{Y}} \mathcal{H}(\mathbf{P}, \mathbf{Q})$;

3. $\mathcal{H}(\mathbf{P}, \mathbf{P}) = \min_{\mathbf{Y}} \mathcal{H}(\mathbf{Q}, \mathbf{P})$.

This Lemma simply shows that CVT sets minimize the energy function, $\mathcal{H}$ as discussed in sec. 1.5.2. The next Lemma describes the relation between the set of mass centroids, $\mathbf{T}$, and Lloyd iterations.

Lemma 4.4.2. *Let $\{\mathbf{P}_n\}_1^\infty$ be the sequence of generating point sets from the Lloyd algorithm, where n is the number of iterations. Then*

1. $\mathbf{P}_n = \mathbf{T}(\mathbf{P}_{n-1})$;

2. $\mathcal{H}(\mathbf{P}_n, \mathbf{P}_n) \leq \mathcal{H}(\mathbf{P}_{n-1}, \mathbf{P}_{n-1})$.

This Lemma implies that the energy function monotonically decreases as Lloyd iterations continue. With these Lemmas in mind, convergence to a certain CVT can be obtained.

Theorem 4.4.3. *Any limit point $\mathbf{P}$ of the Lloyd algorithm is a fixed point, and thus, $(\mathbf{P}, \mathbf{P})$ is a critical point of $\mathcal{H}$ [51]. The set of limit points $\mathbf{P}$ comprise a distinct centroidal Voronoi tessellation.*

Proof. Because $\mathcal{H}(\mathbf{P}_n, \mathbf{P}_n)$ is monotonically decreasing (Lemma 4.4.2.2)

$$\mathcal{H}(\mathbf{P}, \mathbf{P}) = \lim \mathcal{H}(\mathbf{P}_{n_j}, \mathbf{P}_{n_j}) = \inf \mathcal{H}(\mathbf{P}_n, \mathbf{P}_n).$$

It is also known that because $\mathcal{H}(\mathbf{P}, \mathbf{P})$ is a minimum (Lemma 4.4.1.3)

$$\frac{\partial \mathcal{H}(\mathbf{U}, \mathbf{P}_n)}{\partial \mathbf{U}}|_{\mathbf{U}=\mathbf{P}_n} = 0.$$

Therefore, it is also true that

$$\frac{\partial \mathcal{H}(\mathbf{P}, \mathbf{P})}{\partial \mathbf{P}} = 0.$$

If $\frac{\partial \mathcal{H}(\mathbf{P},\mathbf{U})}{\partial \mathbf{U}}|_{\mathbf{U}=\mathbf{P}} = 0$, then $(\mathbf{P},\mathbf{P})$ is a critical point of $\mathcal{H}$ and we are finished. Otherwise, there exists some $\mathbf{Q}$ such that

$$\mathcal{H}(\mathbf{P},\mathbf{Q}) < \mathcal{H}(\mathbf{P},\mathbf{P}).$$

Thus, for small enough δ and large enough n_j

$$\begin{aligned}
\mathcal{H}(\mathbf{P}_{n_j},\mathbf{Q}) &< \mathcal{H}(\mathbf{P},\mathbf{Q}) + \delta \\
&< \mathcal{H}(\mathbf{P},\mathbf{P}) \\
&\leq \mathcal{H}(\mathbf{P}_{n_j+1},\mathbf{P}_{n_j+1}) \qquad \text{Lemma 4.4.2.2} \\
&\leq \mathcal{H}(\mathbf{P}_{n_j},\mathbf{P}_{n_j+1}) \qquad \text{Lemma 4.4.1.3,}
\end{aligned}$$

which contradicts Lemma 4.4.2.1 that claims

$$\mathcal{H}(\mathbf{P}_{n_j},\mathbf{P}_{n_j+1}) = \min_{\mathbf{Y}} \mathcal{H}(\mathbf{P}_{n_j},\mathbf{Q}).$$

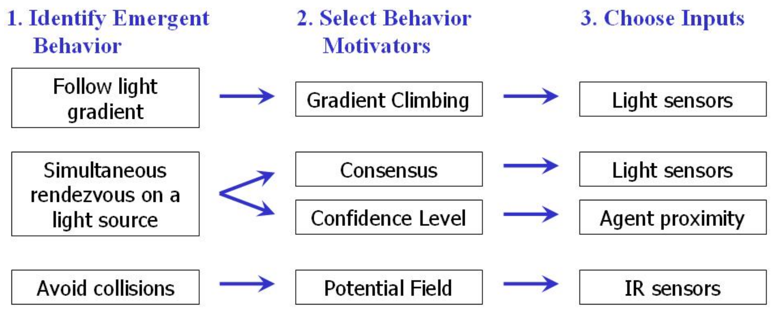
Because the set of limit points $\mathbf{P}$ is fixed and each point is a mass centroid ($\mathbf{P}_\infty = \mathbf{T}(\mathbf{P}_\infty)$ from Lemma 4.4.2.1), these points converge to a fixed and distinct CVT. $\square$

4.5 CVT Versus Behavioral Phototaxis

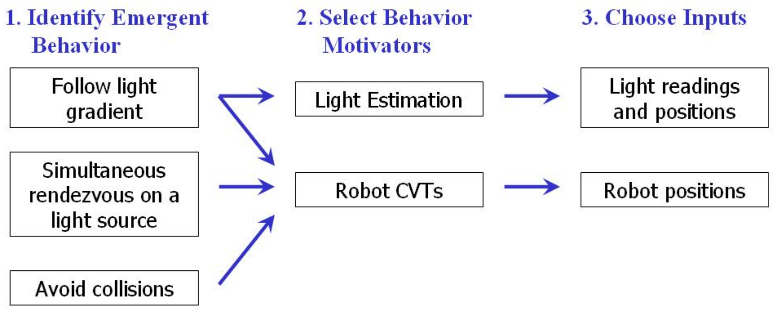
Dynamic phototaxis has been previously achieved using behavioral algorithms [47]. This section presents a brief summary of the behavioral phototaxis design on the MASnet platform and compares results with the new CVT phototaxis algorithm proposed in this thesis.

4.5.1 Behavioral Phototaxis Design

The general desired phototaxis behavior is explained in sec. 4.2.1. However, the behavior motivators and inputs needed are much different for a behavioral design. Figure 4.6 shows a diagram of the behavior motivators and inputs needed to perform both the behavioral and CVT algorithms. The behavioral algorithm requires two more behavior motivators



(a) Behavioral algorithm


1. Identify Emergent Behavior
2. Select Behavior Motivators
3. Choose Inputs
Follow light gradient
Light Estimation
Light readings and positions
Simultaneous rendezvous on a light source
Robot CVTs
Robot positions
Avoid collisions

(b) CVT algorithm

Fig. 4.6: Algorithm comparison for the first three steps of phototaxis.

and an additional input.

Individual algorithms are needed for each behavior motivator in this case. The gradient climbing algorithm switches between a curve left state and a curve right state. Fifteen previous light measurements are stored in an array for each robot. At constant intervals, each robot compares its current light reading to previous light readings. If the current reading is greater, the state switches; there is no change otherwise. These states cause a swimming motion that generally follows the light gradient but is much less smooth and direct than the CVT algorithm robot motion. See fig. 4.7 for an illustration. The dots indicate the path of the robot.

It is important to note that the light readings come from the light sensors on each robot, see fig. 2.5, whereas the light readings for CVTs come from the stationary sensor array. Admittedly, there are several advantages to having collocated sensors and actuators. In this experiment, sensors are dynamic and all functionality is performed on one microprocessor—no cross-communication or separate applications are needed. Future work on

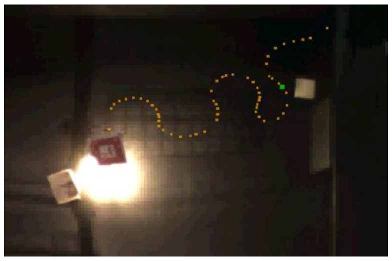


Fig. 4.7: Gradient climbing of one robot (courtesy of William Bourgeous).

collocated sensors and actuators for CVTs can be significant to research.

The second algorithm needed for behavioral phototaxis is consensus. Robot cooperation is required for all robots to simultaneously meet at the light source. The information state for the i^{th} robot, x_i, is the average of the front and back light sensors. A consensus is reached with the following equation

$$\dot{x}_i = -\sum_{j \in \Gamma_i(t)} \alpha_{ij} \left(x_i(t) - x_j(t) \right), \tag{4.5}$$

where $\dot{x}_i$ is the consensus variable, which is broadcast to all in-range robots; $\Gamma_i(t)$ is the set of robots that can communicate with robot i; and α is a positive scalar. The consensus variable is then scaled to modify the i^{th} robot's velocity as follows.

$$\dot{v}_i = v_b \left[1 + \frac{\dot{x}_i - x_i}{\dot{x}_i} \right], \tag{4.6}$$

where $\dot{v}_i$ is the robot's velocity and v_b is the base velocity of the individual robots.

The third behavior motivator for the behavioral approach, is the confidence level. Merely, if at least two robots have found a light source, the robots begin gradient climbing. Finally, collision avoidance was originally programmed for the robots, but the additional algorithm proved too much for the robots computational capabilities. The collision avoidance algorithm is simple, but robots cannot perform this on top of the gradient climb, consensus, and confidence level algorithms. The CVT algorithm, however, does not require stitching complicated behavioral algorithms together.

Also, consider that these behavioral algorithms only work for a stationary light source.

Once the robots reach the light, the gradient climbing state switches to a light follow state. This light follow algorithm divides light intensity readings into 10 zones according to the characterization of one specific incandescent light. A potential field keeps the robots within zone 6 of the light. Figure 4.8 is a state flow diagram for dynamic phototaxis demonstrating a potential field. In contrast, the CVT algorithm does not require switching between states and the light estimation allows any light source to be identified, which eliminates the need for characterizing a particular light source.

4.5.2 Behavioral Phototaxis Results

Two well-documented trials using behavioral phototaxis are explained here. One robot is intentionally disabled in each trial to show the robustness of the system. The incandescent light is placed under the platform on the left side. The first trial consists of three robots and converges after 58 seconds. The second trial converges after 38 seconds with four robots. Convergence rates are typically slowed by the robots' inability to locate the light source and the slow convergence of the consensus algorithm. On average, the CVT algorithm converges nearly four times faster than the behavioral algorithm.

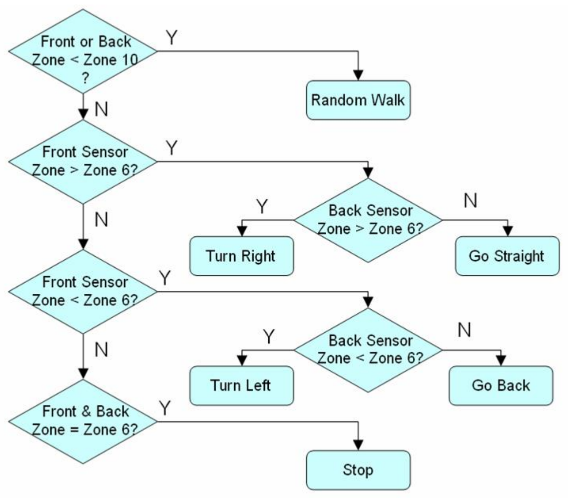

Front or Back Zone < Zone 10 ?
Y
Random Walk
N
Front Sensor Zone > Zone 6?
Y
Back Sensor Zone > Zone 6?
Y
Turn Right
N
Go Straight
N
Front Sensor Zone < Zone 6?
Y
Back Sensor Zone < Zone 6?
Y
Turn Left
N
Go Back
N
Front & Back Zone = Zone 6?
Y
Stop

Fig. 4.8: Light follow algorithm state flow diagram.

Chapter 5

Formation Control Using Centroidal Voronoi Tessellations

Advances in sensors, mesh networking protocols, microcontrollers, etc., provide opportunities to develop small and cheap autonomous robots. These simple robots working in a swarm can perform tasks much more efficiently and reliably than one large robot. Moreover, robots moving in a formation are useful for space, military, and search and rescue tasks such as fire mop-up crews and GPS satellite formations.

A handful of techniques have been developed for robotic formation control. Some of these techniques are listed as follows.

Artificial Potentials uses virtual leaders as moving reference points and artificial potentials to keep a desired spacing between robots [52].

Generalized Coordinates characterizes the desired formation by location, orientation, and shape coordinates [53].

Virtual Structure considers the entire formation group as one unit, which is typically used in spacecrafts or small satellites [54].

Behavioral Control keeps formations by behavioral algorithms inspired by swarms and flocks in nature [55].

Leader-Follower allows one robot to lead a group, or creates chains of robot leaders and followers [2,3].

This chapter introduces results for an emerging CVT formation control technique. CVT formation control research is scarce and has not yet been applied to actual robots to the authors knowledge [45,48]. This research presents the first physical implementation results of CVT formation control and compares these results with other formation control techniques. The simplicity of the algorithm presents its advantages over other formation control techniques.

This research also explores optimal starting positions for any CVT application. The objective is to quickly converge to the desired CVT, which can be used in many real-world applications. For example, suppose fire clean up robots are deployed near a fire. The robots should be able to locate the hot spots accurately and as quickly as possible. The impact of robot number and optimal control gains is also studied. Finally, this thesis discusses a relationship between CVTs and the basic consensus method, which explains fundamental behaviors for CVTs and consensus.

5.1 Previous Formation Control Work

Several experiments for formation control have been conducted on the MASnet platform. All previous formation work is decentralized. In other words, each robot contains its own controller or consensus algorithm to converge to a certain formation. Four separate algorithms and formations have been tested: the leader-follower approach [2, 3], discrete rendezvous, axial alignment, and dynamic formations [1,56]. The following sections discuss these algorithms.

5.1.1 Leader-Follower Method

The most developed formation control method on the MASnet platform is the leader-follower technique. There are many leader-follower scenarios, but all scenarios include at least one leader that follows commands by the operator and one follower that adapts to the movements of the leader(s). MASnet leader-follower experiments implement a distance-heading controller requiring at least one leader and one follower where each follower must communicate with its leader and each leader must communicate with all its followers. Essentially, the controller allows the designated follower(s) to track the leader's heading θ_f while keeping a constant distance r_f from the leader. See fig. 5.1 for an illustration.

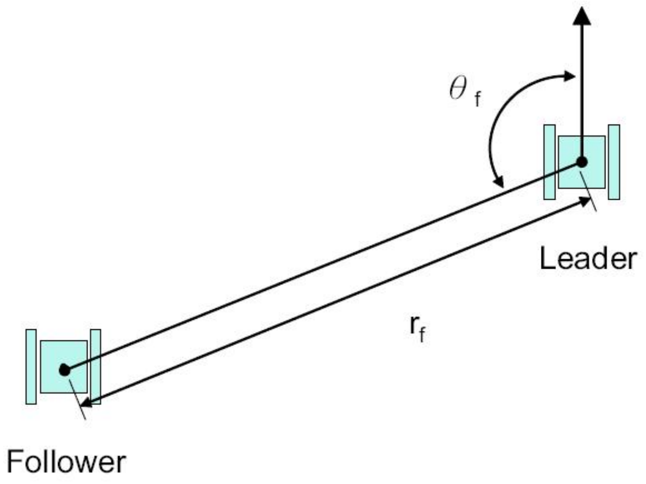


Fig. 5.1: Leader-follower controller parameters (courtesy of Pengyu Chen).

To keep the formation, the follower recalculates its desired position and heading by the following equation.

$$\begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} r_f \cos(\theta_f + \theta_l) \\ r_f \sin(\theta_f + \theta_l) \\ 0 \end{bmatrix} + \begin{bmatrix} x_l \\ y_l \\ \theta_l \end{bmatrix},$$

where (x_l, y_l, θ_l) indicate the position and heading of the leader. The follower updates its calculations every time pGPS information is received. Unfortunately, during experiments, it has been discovered that followers become lost when they stray more than 15 cm from their desired positions, which happens quite frequently [2]. To avoid breaking the formation the follower sends a WAIT_4_ME signal to the leader, which stops till the followers reach their desired position. This successfully keeps formation, but movement is slow and choppy. In one experiment, three followers formed an equilateral triangle around the stationary leader. The formation converged after 53 seconds.

5.1.2 Consensus Rendezvous

Unlike the leader-follower approach, the rendezvous consensus algorithm does not require a leader. With a static or dynamic communication topology, the robots eventually come to a common agreement. In this case, the objective of the rendezvous algorithm is for all robots to converge to the group's centroidal center. Let $r_i = (x_i, y_i)$ be the actual position of the i^{th} robot and let $r_i^* = (x_i^*, y_i^*)$ be the desired position of the i^{th} robot. Robot rendezvous can be achieved by the following equation [1].

$$r_i^* = \frac{\sum_{j=1}^{k} g_{ij} r_j}{\sum_{j=1}^{k} g_{ij}}, \tag{5.1}$$

where k is the number of robots in the group and g_{ij} is a positive weighting factor if robot i receives information from robot j, and 0 otherwise, according to the communication topology matrix.

Perhaps the most intriguing aspect of (5.1) is that it mirrors the mass centroid equation for CVTs in (1.4). This may be considered obvious since both equations are calculations of centroids. However, a new proposition follows considering this relationship.

Proposition 5.1.1. *Consider the mass centroid equation (1.4) for CVTs. The rendezvous consensus equation (5.1) solves the mass centroid equation for individual Voronoi cells. Therefore, the consensus equation, for each Voronoi cell, minimizes the energy function (1.6) and converges to a CVT. Furthermore, unique and finite communication topologies and weighting factors cause systems to converge to unique rendezvous centroids.*

Proof. Isolate one Voronoi cell V_i in the discrete mass centroid equation (1.4). Suppose that a group of robots ($j = 1, \ldots, N$) within V_i serve as the sample points of that cell. In other words, suppose $r_j = q_j$. Make this substitution in the mass centroid equation

$$p_i^* = \frac{\sum_{j=1}^{N} \rho_j r_j}{\sum_{j=1}^{N} \rho_j}.$$

Now, the weighting factor g_{ij} acts as a sample density for each sample point or robot ($g_{ij} = \rho_j$)

$$p_i^* = \frac{\sum_{j=1}^{N} g_{ij} r_j}{\sum_{j=1}^{N} g_{ij}}.$$

It is now easy to see $p_i^* = r_i^*$ from (5.1).

Because unique densities converge to unique CVTs (Proposition 4.3.1 and Theorem 4.4.3), specific communication topologies and weighting factors will converge to specific rendezvous centroids. □

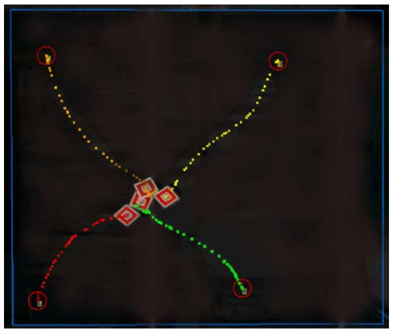
Several consensus rendezvous experiments have been conducted with varying communication topologies and switching topologies. Four robots were placed at the four corners of the platform. Experiments with a connected communication tree converged successfully. The fastest convergence occurred with a fully connected topology at 11.58 seconds. Other topologies converged between 15 and 20 seconds. Switching topologies converged between 24 and 45 seconds. See fig. 5.2 for results of selected experiments. Circles show robot starting positions and the dots show the path of each robot.

Notice in these results that the robots collide with one another to converge to one point. Unfortunately, the consensus algorithm does not include collision avoidance like CVTs. This consensus algorithm converges to one point while the CVT algorithm converges to a tessellation which avoids generator collisions [48, 49].

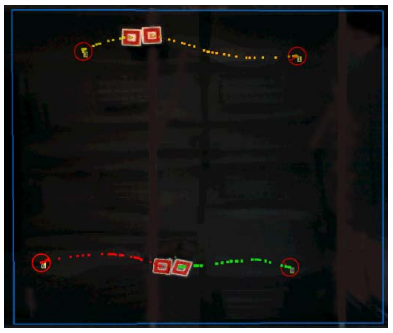
5.1.3 Consensus Static Formations

Equation (5.1) is simply the fundamental consensus algorithm. This algorithm can be expanded to converge to lines and shapes instead of one point. The axial alignment algorithm allows robots to rendezvous to a line while the robots are separated by a small distance δ_{ij}. The equation for the axial alignment formation is as follows [1]:

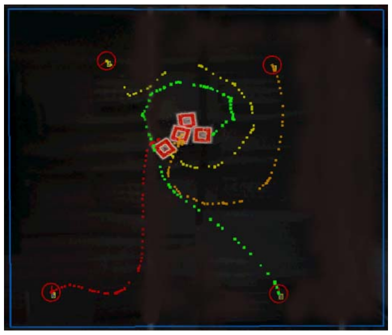
$$r_i^* = \frac{\sum_{j=1}^{k} g_{ij} h_{ij} (r_j + \delta_{ij})}{\sum_{j=1}^{k} g_{ij}}. \tag{5.2}$$



(a) Fully connected ($t_f = 11.58s$)



(b) Not connected ($t_f = 6.93s$)



(c) Connected ($t_f = 18.89s$)

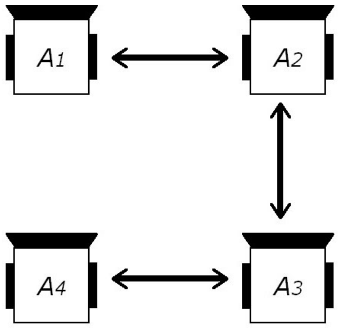
Fig. 5.2: Rendezvous consensus results (courtesy of William Bourgeous).

Figure 5.3 shows the communication topology and results of an axial alignment experiment using (5.2). The robots are intended to converge to a line separated by 24 cm and are initially placed, at random, on the platform. Observe, the robots eventually converge to their desired positions with some overshoot.

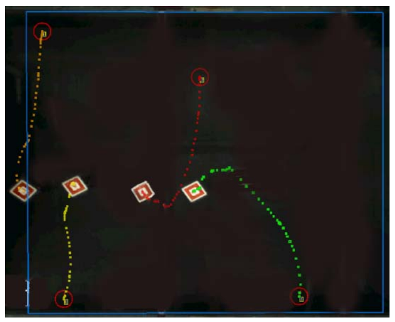
5.1.4 Consensus Dynamic Formations

Finally, by adding a leader to the robot group, the platform can combine the axial alignment algorithm and the leader-follower method to achieve dynamic formations [1]. Unfortunately, because of delays in the system, the robots cannot keep a straight line formation. Followers can only lag behind the leader causing a V-formation.

Figure 5.4 shows the communication topology and results of the dynamic formation. The head robot is the leader. Because of frequent divergence problems, the leader is programmed to move slow. Consequently, the dynamic formation holds well, provided the leader moves slow enough for each robot to converge to its desired position.

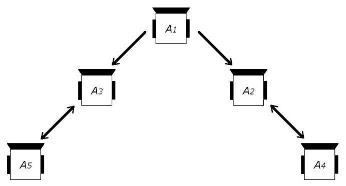


(a) Communication topology

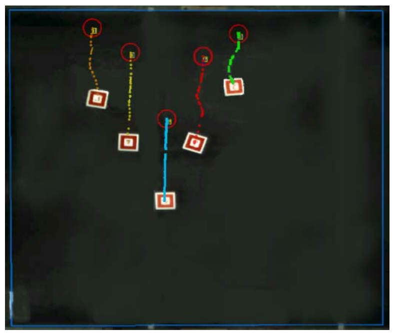


(b) Robot trajectories

Fig. 5.3: Axial alignment results (courtesy of William Bourgeous).



(a) Communication topology



(b) Robot trajectories

Fig. 5.4: V-formation results (courtesy of William Bourgeous).

5.2 CVT Formation Control

The rendezvous CVT formation has been demonstrated and analyzed in Chapter 4. Essentially, a concentrated Gaussian function is given as the density ρ, where the maximum is placed at the desired rendezvous point. Unlike the phototaxis experiment, the CVT formation algorithm defines the rendezvous point. Therefore, an array of sensors is not needed to find a rendezvous point.

Other formations can be created by density functions that form line segments, ellipses, or other arbitrary polygons and curves [45]. For instance, consider an equation for an ellipse, $a(x-x_c)^2 + b(y-y_c)^2 = r^2$, where a, b, and r are positive scalars and (x_c, y_c) is the center of the ellipse. The density function that creates an ellipse formation is as follows.

$$\rho(x,y) = \exp^{-\sigma\left[a(x-x_c)^2+b(y-y_c)^2-r^2\right]^2}, \tag{5.3}$$

where σ is a large positive gain. Compare (5.3) with the Gaussian density used for rendezvous (4.1). Essentially, the ellipse density is a Gaussian density extended to a circle instead of one point, much like (5.2) is a line extension of the consensus rendezvous equation. This “extension” can form other lines and shapes as well. Examples of other formations and their corresponding equations can be found in Table 5.1.

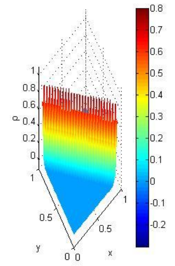
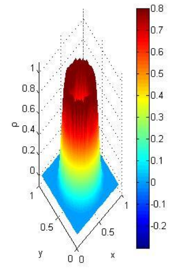
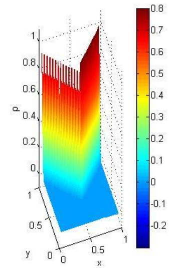
Finally, dynamic formations can easily be created by changing the center of the formation over time $(x_c(t), y_c(t))$. Furthermore, the shape of the formation can evolve by time-varying certain terms. For example, $r(t)$ can change the size of an ellipse formation while $a(t)$ and $b(t)$ can squeeze or widen the ellipse. These shape shifting formations are useful when traversing through tight passageways, corners, etc.

5.3 Simulation Results

The `MAS2D` platform is used to test the performance of these densities. The first set of experiments tested the effect of initial starting positions and number of robots on convergence rate. The control law for the robots is set as previously to $u_i = 3(p_i - p_i^*) - 3\dot{p}_i$.

For each formation (ellipse, line, and v-shape), a group of four, nine, and sixteen robots

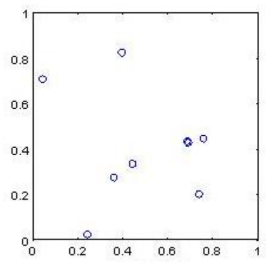
Table 5.1: CVT formation examples; $\rho(x,y)$ takes the form $e^{-\sigma k^2}$.

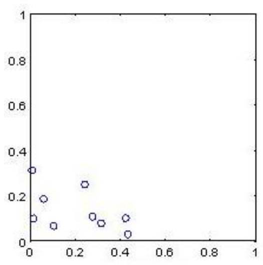
Formation	k	Image
line	$ax + by + c$	
ellipse	$a(x - x_c)^2 + b(y - y_c)^2 - r^2$	
v-shape	$a\,\lvert x - x_c\rvert + b(y - y_c) + c$	

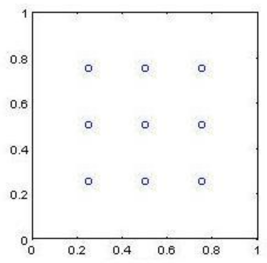
were placed in four different initial configurations. These configurations include random placement and even distribution over the entire region, or over one corner of that region. See fig. 5.5 for examples of these initial configurations and their corresponding numbers for future reference.

The simulation time step is set to 0.05 seconds. Robots compute desired positions every 0.2 seconds. The convergence time is determined by the energy function $\mathcal{H}_V(p)$, which is plotted at each time step. The tessellation converges as the energy function converges. Illustrations of final positions for each formation can be found in fig. 5.6. A CVT approach to formation control allows the robots even spacing without the need of rigid formations or reference points as in other methods.

Figure 5.7 shows convergence rate bar plots for each formation. Each formation follows different convergence trends. However, configurations 3 and 4, which are uniform distributions, typically converge faster. From these results, ideally, a CVT initial configuration would give the quickest convergence, but if the formation is not known, uniformly distributing robots is recommended.


(a) Starting position 1


(b) Starting position 2


(c) Starting position 3

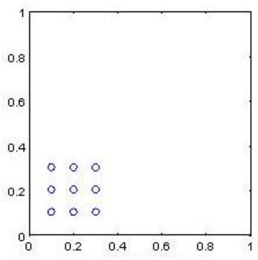
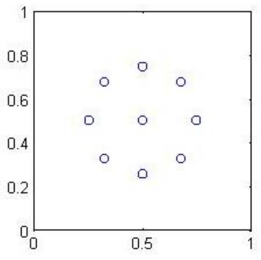
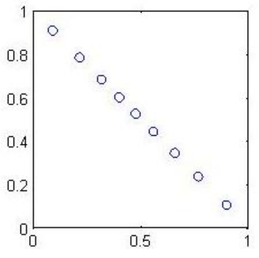

(d) Starting position 4

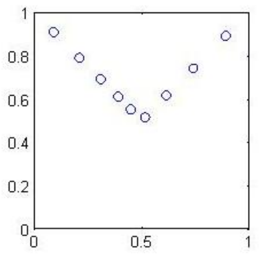
Fig. 5.5: Starting positions for simulation experiment.



(a) Ellipse



(b) Line



(c) V-Shape

Fig. 5.6: Final formation positions.

Also, notice the effect of robot number. An increase in robot number leads to an increase in convergence rate for the ellipse formation as expected. However, this is not the case for configurations 1 and 2 for the line and v-shape formations. This is caused by the nature of random distributions. Some distributions may randomly be closer to the desired CVT than others and, therefore, converge quicker.

It is important to note, here, that these four configurations are by no means representative of all possible initial robot configurations. These experiments attempt to provide a rough analysis on robot CVT performance. As mentioned earlier, CVT formation control is an emerging research topic, which is still in its infancy. Future research will include more detailed analysis on convergence trends and optimal performance.

A second experiment evaluates the control law and its effects on the convergence. Previous research presents a thorough assessment of the damping coefficient k_v [57]. A low k_v will cause overshoot, while a high k_v will cause slow convergence. The distance from the robots' desired position also changes the coefficient's performance. The definition of "too high" or "too low" is finally determined by the optimal damping coefficient, which is found to be 3.

In this thesis, the influence of the proportional gain k_p is presented. Using a group of nine robots with a starting configuration 3, several values of k_p are tested with each formation. See fig. 5.8 for a line plot of the results. Notice, convergence rates for all formations monotonically decrease as a function of k_p, although the decline occurs at different rates. This drop-off takes place because higher proportional gains allow robots to move faster

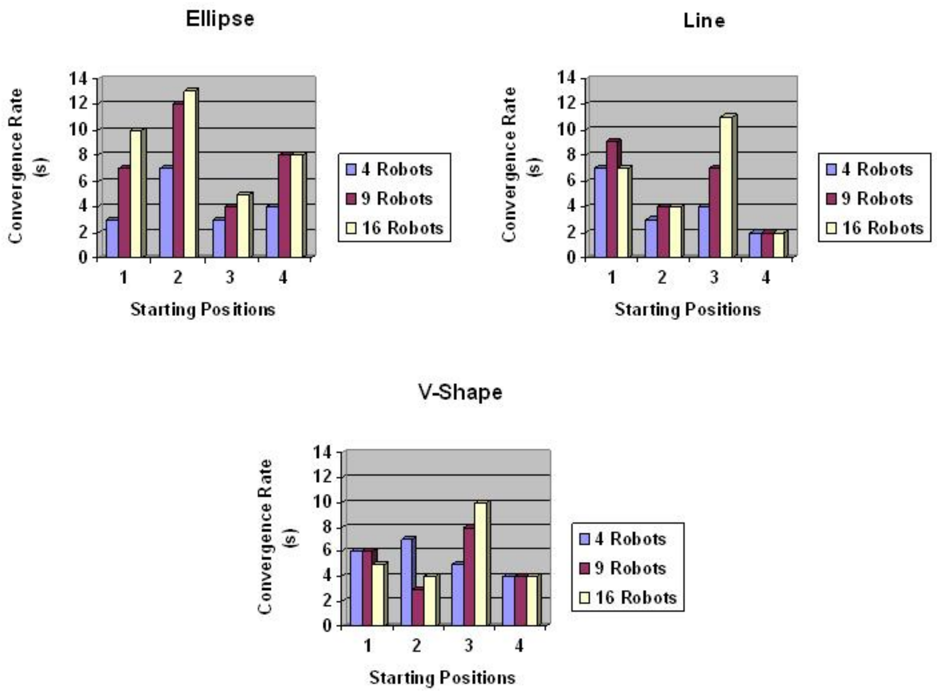


Fig. 5.7: Formation simulation results.

to their desired positions. Also, observe the concentrated convergence rates at $k_p = 10$ and $k_p = 20$. The line plot shows that convergence rates also approach a minimum as k_p increases. A gain of 5 allows a fast convergence rate for all three formations. Further mathematical analysis can find models for convergence trends, which can further explain the difference in drop-off rates and minimum convergence times.

Finally, several dynamic formations were also made. It has been discovered that formations can keep well if the speed of the formation is less than or equal to the the speed of the CVT algorithm. In simulation, this number is 0.05 units/s.

5.4 Robot Implementation Results

Experiments similar to the simulation tests have been conducted. Different densities are programmed in RobotCommander for line, v-shape, rendezvous, and ellipse formations. The base-station recalculates the CVT once every robot has reached its desired position. Initial positions of the robots are set up similar to configuration 3 used in simulation (robots are evenly distributed over the platform). Table 5.2 shows average convergence times for

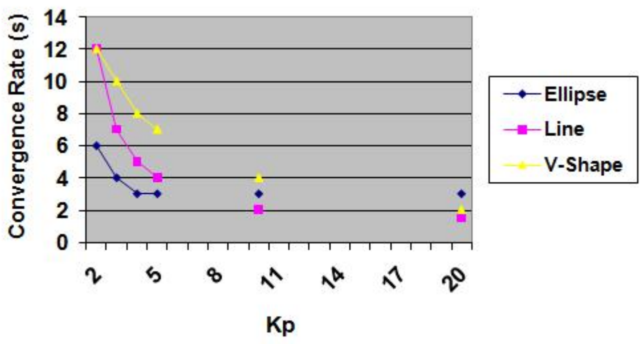


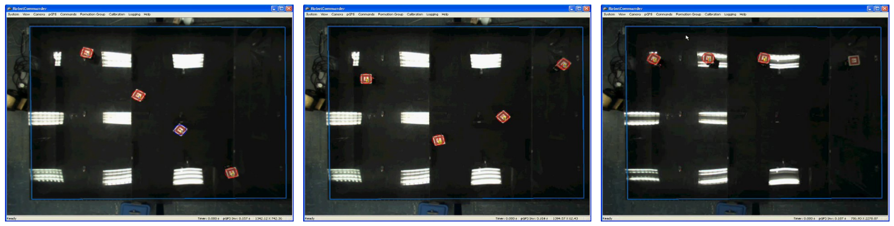
Fig. 5.8: k_p influence on convergence rate.

Table 5.2: Average convergence time for each formation.

Formation	Rendezvous	Ellipse	Line	V-Shape
Convergence Time	13.5 s	10 s	44 s	12.5 s

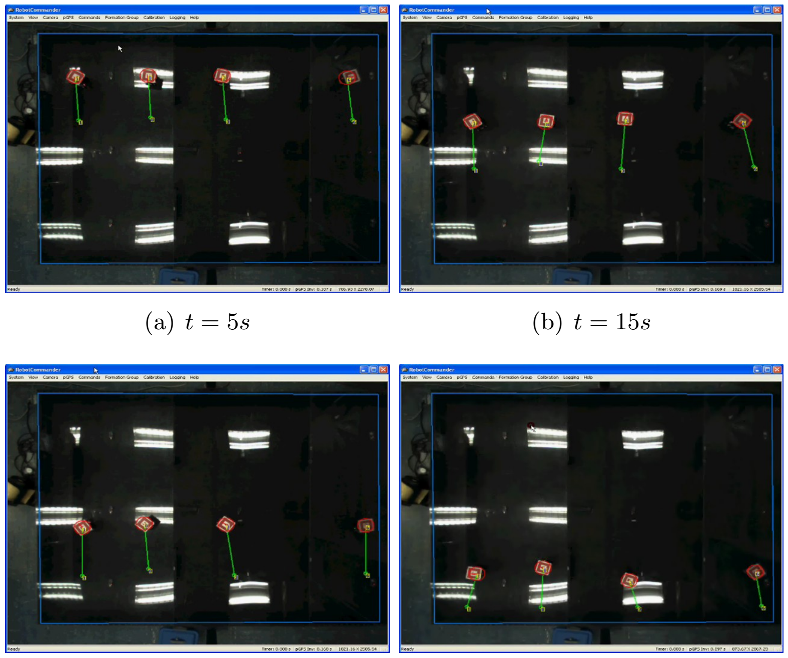
all tested formations. Figure 5.9 shows examples of robot formations.

Similar convergence occurs for dynamic formations. Interestingly, the speed of the formation does not need to be less than the speed of the algorithm. It is actually easier for robots on this platform to follow a faster formation because robots do not need to reorient themselves as often with farther desired positions. Figure 5.10 shows screen shots of a dynamic straight line formation. The red circles indicate robots and the green circles indicate desired positions for each robot. Unlike a dynamic leader-follower formation demonstrated in sec. 5.1.4, the CVT formation can keep a straight line where robots move side by side regardless of network delay. Visit the YouTube channel `http://www.youtube.com/user/MASnetPlatform` for videos of static and dynamic CVT-based formation control experiments on the MASnet platform. The nesC code for CVT-based



(a) Diagnal line (b) V-shape (c) Straight line

Fig. 5.9: Final robot formations.



(a) $t = 5s$ (b) $t = 15s$

(c) $t = 31s$ (d) $t = 49s$

Fig. 5.10: Straight line dynamic formation.

formation control can be found in Appendix A.

5.5 CVT Versus Other Formation Control Techniques

The features of swarm design for multi-robot systems [47] are discussed in the following section to demonstrate the advantages of a CVT-based formation control algorithm. Experiments are conducted to showcase each feature and to compare other formation control techniques previously explained in this chapter. The features of multi-robot swarm design include [47]:

Robustness - The group continues operation despite individual robot failures.

Scalability - Robots can be added and/or removed from the group without reprogramming.

Self-organization - A peacemaker or leader is not required for a robot swarm.

Flexibility - A robot group typically generates new solutions with each trial.

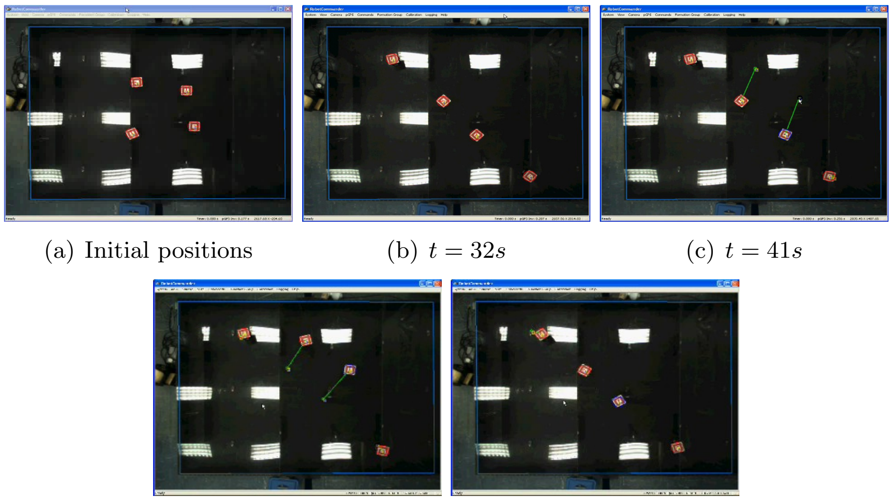
Longevity - A group of robots can outlive a single robot.

Low-cost - Less parts are required for a simple robot design.

5.5.1 Robustness

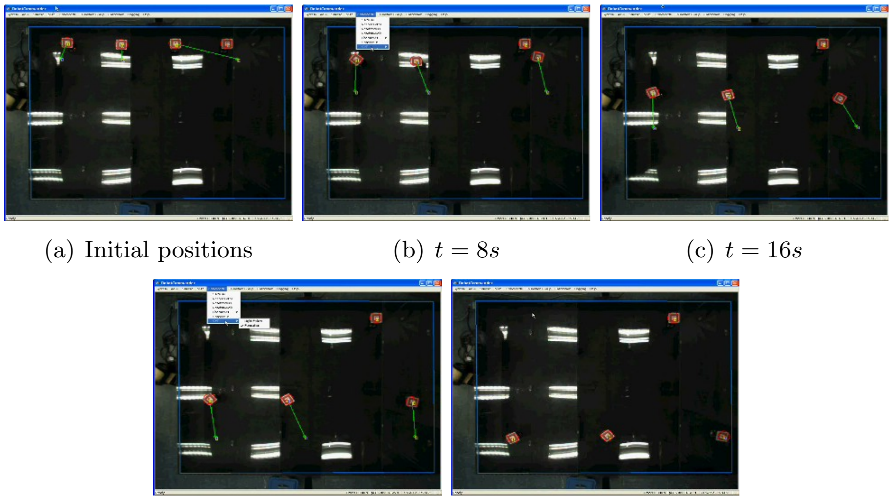
The system's robustness is demonstrated in two ways: pushing robots out of the formation and intentionally impairing selected robots. In the first case, a disturbance is introduced in the formation to observe its effects. Figure 5.11 shows screen shots of this test. After the robots create a line formation in 32 seconds, a command is given for two robots to break the formation in fig. 5.11(c). The robots, then, repair the formation within 9 seconds. Additional experiments produce similar results.

The second robustness test observes the swarm's ability to continue despite individual robot failure. In this case, one robot is intentionally impaired while the CVT formation continues. The disabled robot may be able to communicate, but not move, or vice-versa. One trial is shown in fig. 5.12 using a dynamic line formation. Notice that the three left robots compensate for the failed robot on the right. The swarm eventually distributes itself evenly over the dynamic line.


(a) Initial positions (b) $t = 32s$ (c) $t = 41s$

(d) $t = 56s$ (e) $t = 65s$

Fig. 5.11: Results for the first robustness test.



(a) Initial positions (b) $t = 8s$ (c) $t = 16s$

(d) $t = 33s$ (e) $t = 47s$

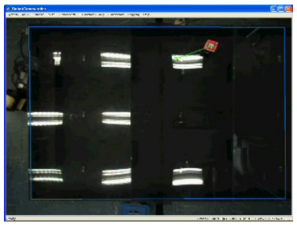
Fig. 5.12: Results for the second robustness test.

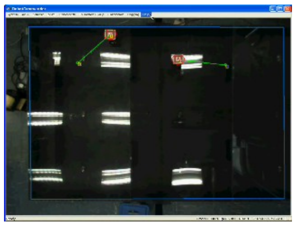
These tests successfully illustrate the robustness of the CVT-based formation control algorithm. Other formation control techniques can also perform disturbance rejection, but some can only tolerate a much smaller disturbance [1–3]. For example, the leader-follower method in secs. 5.1.1 and 5.1.4 diverges if the follower is more than 15 cm away from its desired positions [2].

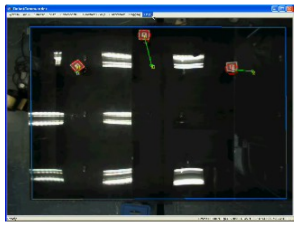
Unlike disturbance rejection, robot failure is often a problem in formation control [1–3]. Most methods require rigid patterns and positions. The failure of one robot in the formation can cause failure of the entire group. In the leader-follower method, each follower depends on its leader [2]. If one leader fails, its followers also fail. In consensus, the failure of one robots will permanently change the communication topology, which can cause divergence or an undesirable consensus [1, 56, 58].

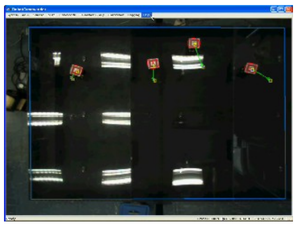
5.5.2 Scalability

The scalability of the system is tested by placing new robots arbitrarily on the platform, at random moments, during the formation task. Figure 5.13 shows screen shots of one trial using a dynamic line formation. Notice, as each new robot appears, the group reconfigures itself for an even line distribution across the platform. This behavior occurres


(a) Initial position


(b) $t = 13s$


(c) $t = 27s$


(d) $t = 43s$

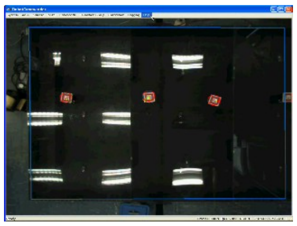

(e) $t = 81s$

Fig. 5.13: Results for the scalability test.

for all scalability tests with different static and dynamic formation scenarios, proving its strong reliability.

Considering other formation control methods, scalability is also often a problem [1–3]. It is difficult for new robots to enter or leave a rigid formation without group failure. The number of robots included in a consensus formation is limited to the size of the communication topology matrix [1, 56, 58]. The CVT formation control algorithm is much more scalable, allowing the formation to self-heal or reconfigure itself once robots are added or taken away from the group.

5.5.3 Self-Organization

A CVT is extremely robust and easily scalable because of its self-organization property. The swarm does not require a leader or peacemaker in the group. The CVT-based algorithm requires information from the collective group and individual robots respond to the collective information. In order to achieve this, robots rely on nearest neighbors, creating a web of dependance. If one robot fails, robots continue to rely on other neighbors in the group. A new robot can also be easily added to the web. Self-organization for CVT-based formation control is evident in all experiments.

5.5.4 Flexibility

Granted, a CVT-based swarm is much less flexible than a behavioral-based swarm. Too much flexibility leads to unpredictable results. CVTs are intended to take much of the guess work out of the algorithm's performance by introducing a mathematical model in place of a behavioral model. Theorems, propositions, and proofs presented in this thesis are the products of analyzing the predictable properties of CVTs.

However, Theorem 4.4.3 shows that unique solutions to CVTs can be achieved by changing the algorithm's density function. The imperfection of the robots also cause each robot to perform somewhat differently to commands. These two factors bring a limited flexibility, allowing diverse, yet, reasonably predictable behaviors and solutions.

5.5.5 Longevity and Low-Cost

Longevity is a positive aspect of any multi-robot application. All formation control techniques require multiple robots. However, a robot group can last longer if it remains stable after individual robot failures. Therefore, the more robust a robot group is, the longer it can last.

A simple, self-contained robot design helps individual robots to last longer. Any swarm-based system, such as this CVT-based formation control system, should use simple robots [47]. With several robots working together, a complicated robot design is unnecessary. Simple robot designs, with fewer hardware components, are much less expensive and easier to build.

5.5.6 Other Advantages and Observations

For many multi-robot systems, a separate algorithm for collision avoidance is needed [1, 14, 47]. Observe the consensus rendezvous results in sec. 5.1.2. The robots collide with each other and contend for the rendezvous point. CVTs, however, converge to a tessellation to avoid generator collisions [48, 49].

It should be mentioned that collisions are still possible with CVT applications, although is it much less frequent. Each robot/generator is represented by a point. The points do not collide, but the robots are much larger than one point. Future work can improve collision avoidance by representing each robot/genreator as a square and implementing techniques such as fractional potential fields [14].

Because of delay and divergence problems, dynamic formations previously implemented on the MASnet platform move slow and choppy. The leader-follower approach often needed the leader and followers to alternate between moving to keep the formation stable [2]. Because of system delay, a dynamic axial alignment formation could not hold [1]. The followers would lag behind the leader creating a v-shape. The dynamic v-shape could hold only if the formation moved slow enough for the robots to converge; see secs. 5.1.3 and 5.1.4.

The CVT-based formation, however, can keep a straight line formation and dynamic formations do not need to move slower than the algorithm converges for success. Faster moving formations are desirable because the robots do not need to reorient themselves as often when moving farther distances. The heading controller used for the robots should be improved to lessen this problem in the future.

Finally, with all CVT experiments, the robot group is able to evenly distribute themselves in the formation without the use of rigid structures or formations. This is an interesting aspect of CVTs that is useful for self-healing once a robot is added or taken away from the formation. However, for some formation control applications, rigid structures and exact positions are needed. In these cases, CVT-based formation control should not be used.

Considering all of these aspects, CVTs is an extremely robust and scalable method for formation control. Though it may take longer to converge, CVT-based formation control is self-healing and does not require rigid structures. CVTs are beneficial to formation control applications that do not demand precise positioning.

Chapter 6
Conclusion

6.1 Contributions

In this thesis, several topics have been discussed related to the CVT phototaxis and formation control algorithms. Some of these topics have been used or mentioned before. Below is a list of new contributions presented in this thesis, separated by chapter:

- MASnet Platform
  - New second generation MicaZ MASmote design
    - Bolted battery pack mount
    - Photo-diode Plexiglas® support
    - Bolted voltage divider mount
    - GWS servo motors
    - PCB layout
- Heterogeneous Swarm System
  - MicaZ and Tmote cross-communication
  - Efficient light sensing application
  - Tmote logging function
  - “Read Tmote” program for analyzing Tmote data
- Phototaxis
  - Light estimation algorithm
  - CVT phototaxis algorithm

  - Simulation results and proof of convergence
  - Robot implementation results and proof of convergence

- Formation Control
  - Propose and proof of relationship between CVTs and consensus
  - CVT formation control algorithm
  - Analysis on control law, robot number, and robot initial positions
  - Robot implementation results of static and dynamic formations.

6.2 Future Work

This research is simply the beginning of many possibilities. Suggestions for future work on the MASnet platform and CVT algorithms in general are listed here:

- MASnet Platform
  - Improve point-to-point robot control
  - Increase the number of markers detected by RobotCommander
  - Implement a more reliable communication protocol
  - Reduce the need for pGPS
    * Improve robot position estimation by Kalman filters, RF signal strength, additional encoders, etc.
  - Observe CVTs with fog diffusion
- CVT and Other Algorithms
  - Create a true decentralized design
    * Implement a mesh sensor network with a dynamic topology
    * Limit the range of each robot
  - Analyze and improve CVT performance in detail

    * Mathematically validate the effects of control law, initial starting positions, and robot number
    * Create models for convergence trends
    * Improve collision avoidance by potential fields or other methods
  - Experiment with collocated sensors
  - Experiment with electromagnetic-taxis (EM-taxis)

References

[1] W. K. Bourgeous, “Engineering swarms for mobile sensor networks,” Master’s thesis, Utah State University, Logan, UT, 2007.

[2] P. Chen, “Pattern formation in mobile wireless sensor networks,” Master’s thesis, Utah State University, Logan, UT, 2005.

[3] Z. Wang, “Distributed control of distributed parameter systems using mobile actuator and sensor networks,” Master’s thesis, Utah State University, Logan, UT, 2005.

[4] A. Ganguli, S. Susca, S. Martinez, F. Bullo, and J. Cortes, “On collective motion in sensor networks: sample problems and distributed algorithms,” *The Proceedings of the 44th IEEE Conference on Decision and Control and 2005 European Control Conference*, pp. 4239–4244, Dec. 2005.

[5] E. H. Callaway, *Wireless Sensor Networks: Architectures and Protocols.* Boca Raton, FL: CRC Press LLC, 2003.

[6] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu, *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*, 2nd ed., ch. 4, pp. 229–287. Hoboken, NJ: John Wiley & Sons, Ltd., 2002.

[7] Q. Du, M. Emelianenko, H.-C. Lee, and X. Wang, “Ideal point distributions, best mode selections and optimal spatial partitions via centroidal Voronoi tessellations,” in *The 2nd International Symposium on Voronoi Diagrams in Science and Engineering*, pp. 325–333, Seoul, Korea, Oct. 2005.

[8] H. Chao, Y. Chen, and W. Ren, “A study of grouping effect on mobile actuator sensor networks for distributed feedback control of diffusion process using central Voronoi tessellations,” *The Proceedings of the 2006 IEEE International Conference on Mechatronics and Automation*, pp. 769–774, June 2006.

[9] Q. Du, V. Faber, and M. Gunzburger, “Centroidal Voronoi tessellations: Applications and algorithms,” *SIAM Review*, vol. 41, no. 4, pp. 637–676, 1999.

[10] H.-C. Lee and S.-W. Lee, “Reduced-order modeling of burgers equations based on centroidal Voronoi tessellation,” in *The 2nd International Symposium on Voronoi Diagrams in Science and Engineering*, pp. 346–357, Seoul, Korea, Oct. 2005.

[11] J. B. MacQueen, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, L. M. L. Cam and J. Neyman, Eds., vol. 1, pp. 281–297. University of California Press, 1967.

[12] J. Liang and Y. Chen, *Diff-MAS2D (version 0.9) - User’s Manual*, 1st ed., `http://mechatronics.ece.usu.edu/reports/USU-CSOIS-TR-04-03.pdf`, Utah State University, Logan, UT, Oct. 2004.

[13] H. Chao, Y. Chen, and W. Ren, "Consensus of information in distributed control of a diffusion process using centroidal Voronoi tessellations," *The Proceedings of the 46th IEEE Conference on Decision and Control*, pp. 1441–1446, Dec. 2007.

[14] A. Howard, M. J. Mataric, and G. S. Sukhatme, "Mobile sensor network deployment using potential fields: A distributed, scalable solution to the area coverage problem," in *Proceedings of the 6th International Symposium on Distributed Autonomous Robotics Systems (DARS02)*, pp. 299–308, Fukuoka, Japan, June 2002.

[15] Y. Chen, K. L. Moore, and Z. Song, "Diffusion-based path planning in mobile actuator-sensor networks (MAS-Net): Some preliminary results," in *Proceedings of SPIE Conference on Intelligent Computing: Theory and Applications II, part of SPIE's Defense and Security*, Orlando, FL, Apr. 2004.

[16] Z. Wang, Z. Song, P. Chen, A. Arora, D. Stormont, and Y. Chen, "MASmote - a mobility node for MAS-net (mobile actuator sensor networks)," in *IEEE International Conference on Robotics and Biomimetics*, pp. 816–821, Shengyang, China, Aug. 2004.

[17] Crossbow Technology Inc., "MICAz, Wireless Measurement System," `http://www.xbow.com`.

[18] Futaba Digital Servos, "Digital Servo S9254," `http://futaba-rc.com/servos/digitalservos.html`.

[19] Nubotics, "WheelWatcher Encoder," `http://www.nubotics.com/products/ww01/index.html`.

[20] Burr-Brown Products from Texas Instruments, "Monolithic Photodiode and Single-Supply Transimpedance Amplifier," `http://focus.ti.com/lit/ds/symlink/opt101.pdf`, Jan. 1994.

[21] Sharp, "General Purpose Type Distance Measuring Sensors," `http://www.active-robots.com/products/sensors/sharp/gp2d120.pdf`.

[22] A. Arora, "Sensing a diffusion process using distributed robots," Master's thesis, Utah State University, Logan, UT, 2005.

[23] Acroname Robotics, "R157-black-oring-wheel," `http://www.acroname.com/robotics/parts/R157-BLACK-ORING-WHEEL.html`, May 2008.

[24] Grand Wing Servo-Tech Co., Ltd., "Servo products: Standard series," `http://www.gwsus.com/english/product/servo/standard.htm`, Aug. 2005.

[25] Mark III Robot Store, "Mark III chassis kit," `http://www.junun.org/MarkIII/Info.jsp?item=2`.

[26] A. M. J. Cortes and J. M. Worth, *Generation II MAS-Motes Construction Manual*, Center for Self-Organizing Intelligent Systems (CSOIS), Utah State University, Logan, UT, Aug. 2008.

[27] Crossbow Technology Inc., "Getting Started Guide for Wireless Sensor Networks," `http://www.xbow.com`.

[28] D. Gay, P. Levis, D. Culler, and E. Brewer, "nesC 1.2 Language Reference Manual," `http://www.tinyos.net`, Aug. 2005.

[29] University of California at Berkeley, "TinyOS - An open-source OS for sensor networks," `http://www.tinyos.net`.

[30] University of California at Berkeley, "TinyOS Tutorial," `http://www.tinyos.net/tinyos-1.x/doc/tutorial/index.html`, Sept. 2003.

[31] University of Washington, "ARToolKit," Human Interface Technology Laboratory, `http://www.hitl.washington.edu/artoolkit`, 2004.

[32] Moteiv, "Tmote Sky: Ultra low power IEEE 802.15.4 compliant wireless sensor module," `http://www.eecs.harvard.edu/\char126konrad/projects/shimmer/references/tmote-sky-datasheet.pdf`, Feb. 2006.

[33] L. Lee, "Monitoring of indoor relative humidity levels in residential dwellings: A sensor network application," Master's thesis, Utah State University, Logan, UT, 2008.

[34] Hamamatsu, "Si photodiode s1087/s1133 series: Ceramic package photodiode with low dark current," `http://www.datasheetcatalog.org/datasheet/hamamatsu/S1133.pdf`, 2001.

[35] J. Thorn, "Deciphering TinyOS serial packets," Octave Tech Brief 5-01, Mar. 2005.

[36] W. Ye and J. Heidemann, "Medium access control in wireless sensor networks," *USC/ISI Technical Report ISI-TR-580*, 2003.

[37] L. Kleinrock and F. Tobagi, "Packet switching in radio channels: Part I–carrier sense multiple-access modes and their throughput-delay characteristics," *IEEE Transactions on Communications [legacy, pre - 1988]*, vol. 23, no. 12, pp. 1400–1416, Dec. 1975.

[38] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient MAC protocol for wireless sensor networks," in *IEEE Global Telecommunications Conference*, vol. 3, p. 1567, St. Louis, MO, 2005.

[39] J. Afonso, L. Rocha, H. Silva, and J. Correia, "MAC protocol for low-power real-time wireless sensing and actuation," *The Proceedings of the 13th IEEE International Conference on Electronics, Circuits and Systems*, pp. 1248–1251, Dec. 2006.

[40] M. Salajegheh, H. Soroush, and A. Kalis, "Hymac: Hybrid TDMA/FDMA medium access control protocol for wireless sensor networks," *The Proceedings of the IEEE 18th International Symposium on Personal, Indoor and Mobile Radio Communications*, pp. 1–5, Sept. 2007.

[41] S. Krishnamurthy, "TinySIP: Providing seamless access to sensor-based services," *The Proceedings of the Third Annual International Conference on Mobile and Ubiquitous Systems: Networking & Services*, pp. 1–9, July 2006.

[42] D. Zarzhitsky, D. F. Spears, and W. M. Spears, “Swarms for chemical plume tracing,” *The Proceedings of the 2005 IEEE Swarm Intelligence Symposium (SIS2005)*, pp. 249–256, June 2005.

[43] Y. Q. Chen, Z. Wang, and J. Liang, “Automatic dynamic flocking in mobile actuator sensor networks by central Voronoi tessellations,” *The Proceedings of the 2005 IEEE International Conference on Mechatronics and Automation*, vol. 3, pp. 1630–1635, 2005.

[44] Y. Chen, K. Moore, and Z. Song, “Diffusion boundary determination and zone control via mobile actuator-sensor networks (MAS-net): challenges and opportunities,” *International Society for Optical Engineering Proceedings Series*, vol. 5421, pp. 102–113, 2004.

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

[46] S. Kazadi, “On the development of a swarm engineering methodology,” in *Proceedings IEEE Conference on Systems, Man, and Cybernetics*, pp. 1423–1428, Oct. 2005.

[47] W. Bourgeous, S. Rounds, and Y. Chen, “A swarm engineering approach to mobile sensor network design towards collaborative phototaxis with a slowly moving light source,” in *ASME/IEEE 3rd International Conference on Mechatronics and Embedded Systems Application*, Las Vegas, NV, Sept. 2007.

[48] M. Lindhe and K. H. Johansson, *Taming Heterogeneity and Complexity of Embedded Control*, ch. 24, pp. 419–434. Hoboken, NJ: John Wiley & Sons, Ltd., Feb. 2007.

[49] C.-Y. Mai and F.-L. Lian, “Analysis of formation control and networking pattern in multi-robot systems: a hexagonal formation example,” *International Journal of Systems, Control and Communications 2008*, vol. 1, no. 1, pp. 98–123, 2008.

[50] M. Hall, “Taking robots to the extreme,” NSF REU 2006 PowerPoint Presentation, `http://www.csois.usu.edu`, Aug. 2006.

[51] Q. Du, M. Emelianenko, and L. Ju, “Convergence of the Lloyd algorithm for computing centroidal Voronoi tessellations,” *SIAM Journal on Numerical Analysis*, vol. 44, no. 1, pp. 102–119, 2006.

[52] N. Leonard and E. Fiorelli, “Virtual leaders, artificial potentials and coordinated control of groups,” *The Proceedings of the 40th IEEE Conference on Decision and Control*, vol. 3, pp. 2968–2973, 2001.

[53] S. Spry and J. Hedrick, “Formation control using generalized coordinates,” *The Proceedings of the 43rd IEEE Conference on Decision and Control*, vol. 3, pp. 2441–2446, Dec. 2004.

[54] R. Beard, J. Lawton, and F. Hadaegh, “A coordination architecture for spacecraft formation control,” *IEEE Transactions on Control Systems Technology*, vol. 9, no. 6, pp. 777–790, Nov. 2001.

[55] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *SIGGRAPH International Conference on Computer Graphics and Interactive Techniques*, vol. 21, no. 4, pp. 25–34, 1987.

[56] W. Ren, H. Chao, W. Bourgeous, N. Sorensen, and Y. Chen, "Experimental validation of consensus algorithms for multivehicle cooperative control," *IEEE Transactions on Control Systems Technology*, vol. 16, no. 4, pp. 745–752, July 2008.

[57] S. Wolf, "An interpolation technique for reduction of sensor density in CVT-based mobile actuator sensor networks," NSF REU 2008 PowerPoint Presentation, `http://www.csois.usu.edu`, Aug. 2008.

[58] W. Ren and R. Beard, "Consensus of information under dynamically changing interaction topologies," *The Proceedings of the American Control Conference*, vol. 6, pp. 4939–4944, June 2004.

[59] Crossbow Technology Inc., "MoteWorks Getting Started Guide," `http://www.xbow.com`, Apr. 2007.

[60] Crossbow Technology Inc., "XServe Users Manual," `http://www.xbow.com`, Apr. 2007.

[61] Crossbow Technology Inc., "XMesh Users Manual," `http://www.xbow.com`, Apr. 2007.

Appendices

Appendix A

RobotCommander Code for CVT Algorithms

This appendix includes files and excerpts from the RobotCommander code for CVT phototaxis and CVT formation control. For all excerpts, code added by the author is commented with the name `Shelley`. For a list of all files used for RobotCommander, refer to Appendix C.

Note. *Some line numbers have been changed from the actual files for the reader's convenience. Also, the listed code does not include all changes made by the author. Details such as variable declarations are not included. To see all changes by the author, refer to the latest RobotCommander code.*

A.1 masnet_Messages.h

This section includes code that allows RobotCommander to understand incoming Tmote messages as described in sec. 3.4.2. These excerpts are from the file `masnet_Messages.h`.

Message type definitions

```
enum MASNET_MSG_TYPE {
    /* From mote to pc */
    // 0 is reserved for error
     AM_MOTE_SENSOR_MSG             = 1 /* sensor readings from motes*/
    ,AM_CMD_COMPLETE_MSG            = 2 /* sent from mote to pc once all
        current commands it has are completed */
    ,AM_EXCEPTION_MSG               = 3 /* error alert */
    ,AM_DEBUG_MSG                   = 4
    ,AM_COLLECT_MSG                 = 5

    /* From Tmote to PC (tmote sensor data)  added by Shelley 10132007*/
    ,AM_TMOTE_MSG       = 10    /*so PC can recognize Tmote messages*/
```

Message structure definition

```
/* The following is added 10132007 to recieve Tmote messages by Shelley */
enum {
    OSCOPE_BUFFER_SIZE = 10,    //define constants
};

typedef struct OscopeMsg_st {
  uint16_t sourceMoteID;
  uint16_t lastSampleNumber;
  uint16_t channel;
  uint16_t data[OSCOPE_BUFFER_SIZE];
} OscopeMsg;
/* ADDED by Shelley 10132007*/
```

A.2 VoronoiTessellation

The following files were added to RobotCommander to calculate CVTs and light estimation, as well as perform phototaxis and formation control.

VoronoiTessellation.h

```
/***************************************************************
Header file added to calculate centroidal Voronoi tessellations
***************************************************************/

#pragma once
#include "masnet_def.h"
#include "masnet_Messages.h"
#include <math.h>
#include <vector>
#include <matrix.h>

/*For matrix template*/
#ifndef _NO_NAMESPACE
using namespace std;
using namespace math;
#define STD std
#else
#define STD
#endif

#ifndef _NO_TEMPLATE
typedef matrix<double> Matrix;
#else
typedef matrix Matrix;
#endif
/*********************/

#define MASNET_MAX_TMOTES    9

typedef struct TmoteMsg_st {    //tmote (sensor) info
```

```
    int sourceMoteID;
    int lastSampleNumber;
    int channel;
    int PAR;
    float  X_POS;
    float  Y_POS;
    int voronoiRobot;   //robot assocciated with that tmote
} TmoteMsg;

typedef struct RobotMsg_st {    //robot (actuator) info
    int mote;
    int frontPhoto;
    int backPhoto;
    float  x;
    float  y;
    float  angle;
} RobotMsg;


class CVoronoiTessellation
{
protected:
    TmoteMsg tmote_array[MASNET_MAX_TMOTES+1]; // index is tmote id
    RobotMsg robot_array[MASNET_PGPS_MAX_ROBOTS+1];

    //Added for light estimation
    Matrix apos;
    Matrix sinfo;
    Matrix readings;
    vector<int> robotIDs;
    int timeCount;
    float xcen, ycen;

    // variables for comparing distances
    int mini;
    float min;
    float temp;
    float robotX;
    float robotY;
    float sensorX;
    float sensorY;
    bool m_cancel;          //flag to check if cancel has ever been pressed
    bool m_cvt_lightfollow; //flag to check if light follow mode is
        activated
    bool m_cvt_formation;   //flag to check if formation mode is activated

public:
    CVoronoiTessellation (void);
    ~CVoronoiTessellation (void);
    void updateCVT();
    void setTmote(OscopeMsg* msg, int sensor, float x, float y);
    void setRobot(MoteSensorMsg* msg);
    void clearCVT();
    void flushTmote();
    TmoteMsg* getTmote(int sensor){return &tmote_array[sensor];}
    RobotMsg* getRobot(int actuator){return &robot_array[actuator];}
    int isNotFull(void);
    bool robotReceived(int actuator){return !robot_array[actuator].mote==0;}
    bool initialize();

    bool cancel() {return m_cancel;}
    void setCVTLightFollow(bool flag) {m_cvt_lightfollow = flag;}
    void setCVTFormation(bool flag) {m_cvt_formation = flag;}
    bool isCVTLightFollow() {return m_cvt_lightfollow;}
    bool isCVTFormation() {return m_cvt_formation;}

    void setMin() {min = sqrt(pow(MASNET_PGPS_PLATFORM_LENGTH_X,2)+pow(
        MASNET_PGPS_PLATFORM_LENGTH_Y,2));}
    void setTime() {timeCount = 0;}
    bool noMovement();  //returns true if there are no commands to any
        active robots

    float getLightX() {return xcen;}    //returns x location of light
    float getLightY() {return ycen;}    //returns y location of light

    /*Functions to calculate light estimation*/
    void interpSinfo(Matrix sinfo, Matrix readings, Matrix* newSinfo, Matrix
        * newReadings);
    void lightLocation(Matrix sinfo, Matrix readings, float xmin, float xmax
        , float ymin, float ymax, int dlev);
    void lightLocationRec(Matrix sinfo, Matrix readings, float xmin, float
        xmax, float ymin, float ymax, int step);
    void lightmin(float x, float y, Matrix sinfo, Matrix lambda, float* res,
         Matrix* A);
    Matrix pseudoInverse(Matrix A);
    Matrix multiply(Matrix A, float scalar);
    float dot(Matrix A, Matrix M);
    /*****************************************/

};
```

VoronoiTessellation.cpp

```
#include "StdAfx.h"
#include "VoronoiTessellation.h"
#include "RobotCommanderCore.h"
#include "MainFrm.h"
#include <math.h>

extern CRobotCommanderCore RcCore;


CVoronoiTessellation::CVoronoiTessellation(void)
: m_cancel(false)
 ,m_cvt_lightfollow(false)
 ,m_cvt_formation(false)

{
    memset(tmote_array, 0, MASNET_MAX_TMOTES+1 * sizeof(TmoteMsg));//set
        tmote array to 0
    memset(robot_array, 0, MASNET_PGPS_MAX_ROBOTS+1 * sizeof(RobotMsg));//
        set robot array to 0
    setMin();
    setTime();
    mini = 0;
    apos.SetSize(MASNET_PGPS_NUM_OF_ROBOTS, 2);
    sinfo.SetSize(MASNET_MAX_TMOTES, 2);
    readings.SetSize(MASNET_MAX_TMOTES, 1);
    robotIDs.assign(MASNET_PGPS_MAX_ROBOTS,0);  //initialized to zeros
    xcen = 0, ycen = 0;
}

CVoronoiTessellation::~CVoronoiTessellation (void)
{
}

void CVoronoiTessellation::updateCVT()
{
    int numOfRobots = 0;
    int i, t;
    for(i=1; i<= MASNET_PGPS_NUM_OF_ROBOTS; i++){
        if(robotReceived(i)){
            apos(numOfRobots,0) = robot_array[i].x;
            apos(numOfRobots,1) = robot_array[i].y;
            robotIDs[numOfRobots] = robot_array[i].mote;
            numOfRobots++;
        }
    }
    apos.SetSize(numOfRobots,apos.ColNo()); //resize apos matrix
    robotIDs.resize(numOfRobots);           //resize robotIDs vector

    Matrix newSinfo, newReadings;   //holds all virtual sensor position and
        readings respectively
    interpSinfo(sinfo, readings, &newSinfo, &newReadings);

    int mini = 0;
    int virtualSensorNum = (int)newReadings.RowNo();
    vector<int> voronoiCell(virtualSensorNum,mini);

    for(i=0; i < virtualSensorNum;  i++){   //sensors
        setMin();
        for(t=0; t < numOfRobots;   t++){   //actuators
            float temp = (float)sqrt(pow(apos(t,0)-newSinfo(i,0),2)+pow(apos
                (t,1)-newSinfo(i,1),2));
            if(temp < min){
                min=temp;
                mini=t;
            }
        }
        voronoiCell[i] = mini;
    }
    for(t=0; t < numOfRobots;   t++){   //calculate mass centroid
        float sumx = 0;
        float sumy = 0;
        float sums = 0;

        for(i=0; i < virtualSensorNum; i++){
            if(voronoiCell[i] == t){    //only loops through the sensors in
                the robots voronoi cell
                sumx += (float)(newReadings(i,0)*newSinfo(i,0));
                sumy += (float)(newReadings(i,0)*newSinfo(i,1));
                sums += (float)newReadings(i,0);
            }
        }
        if(sums != 0)   //avoid division by zero
            RcCore.setRobotDestByID(sumx/sums, sumy/sums, MASNET_UNSET_ANGLE
                , robotIDs[t], true);
    }

    apos.SetSize(MASNET_PGPS_NUM_OF_ROBOTS, 2); //in case more robots are
        introduced (elliminates matrix error)
    robotIDs.resize(MASNET_PGPS_NUM_OF_ROBOTS);
}

void CVoronoiTessellation::setTmote (OscopeMsg* msg, int reading, float x,
    float y)
{
    int i = msg->sourceMoteID - 20;
    tmote_array[i].sourceMoteID = i;
    tmote_array[i].lastSampleNumber = msg->lastSampleNumber;
    tmote_array[i].channel = msg->channel;
    tmote_array[i].PAR = reading;
    tmote_array[i].X_POS = x;
    tmote_array[i].Y_POS = y;

    //set matricies
    sinfo(i-1,0) = x;
    sinfo(i-1,1) = y;
    readings(i-1,0) = reading;
}

void CVoronoiTessellation::setRobot (MoteSensorMsg* msg)
{
    int i = msg->mote;
    robot_array[i].mote = i;
    robot_array[i].frontPhoto = msg->frontPhoto;
    robot_array[i].backPhoto = msg->backPhoto;
    robot_array[i].x = (msg->x);
    robot_array[i].y = (msg->y);
    robot_array[i].angle = (msg->angle);
}

void CVoronoiTessellation::clearCVT(){
    flushTmote();
    memset(robot_array, 0, MASNET_PGPS_MAX_ROBOTS+1 * sizeof(RobotMsg));//
        set robot array to 0
    apos.Null();
    xcen = 0;
    ycen = 0;
}

void CVoronoiTessellation::flushTmote(){
    memset(tmote_array, 0, MASNET_MAX_TMOTES+1 * sizeof(TmoteMsg));//set tmote array to 0
    sinfo.Null();
    readings.Null();
}

int CVoronoiTessellation::isNotFull (void)
{
    for(int i=1; i <= MASNET_MAX_TMOTES; i++){
        if(tmote_array[i].sourceMoteID == 0)    return i;   //if any tmote msg has not been received
    }
    return 0;
}

bool CVoronoiTessellation::initialize()
{
    CWaitCursor wait;   //create an hourglass waiting cursor for sensor array to initialize
    m_cancel = false;

    //wait to gather sensor data
    for(int i=1; i <= 500000000; i++){
        if(!isNotFull()){
            MessageBox(NULL,"Sensor array is ready.", "Message", MB_OK);
            return true;
        }
    }

    //if it is not full after 5 seconds display an error message
    int j = isNotFull();
    CString id;
    id.Format   (_T("All 9 Tmotes must be on.  The following \nTmote has not yet been recieved\n\n  ID:     %d\n"),j+20);
    int message = MessageBox(NULL, id, "Error", MB_ICONSTOP|MB_RETRYCANCEL);

    switch (message){
        case IDCANCEL:  m_cancel = true;    return false;   break;
        case IDRETRY:   initialize();   break;
    }
    return true;
}

/*The following functions are for light estimation*/
bool CVoronoiTessellation::noMovement(){    //retruns true if no robots have a command
    for(int t=1; t <= MASNET_PGPS_NUM_OF_ROBOTS;    t++)    //actuators
        if(!RcCore.hasNoCmd(t)) return false;
return true;
}

void CVoronoiTessellation::interpSinfo(Matrix sinfo, Matrix readings, Matrix* newSinfo, Matrix* newReadings){
    int i,j;

    if(m_cvt_lightfollow){              //if Light Follow mode is on
```

```
        int step1 = 1000;           //stepsize to find location of light
        lightLocationRec(sinfo, readings, 0, MASNET_PGPS_PLATFORM_LENGTH_X,
            0, MASNET_PGPS_PLATFORM_LENGTH_Y, step1);
    }else{                          //if Formation mode is on
        xcen = 1500;
        ycen = 2000 - timeCount*500;
        //ycen = 1200;
    }

    int step2 = 50;         //stepsize to create virtual sensors
    int M = (int)floor(MASNET_PGPS_PLATFORM_LENGTH_X/step2);
    int N = (int)floor(MASNET_PGPS_PLATFORM_LENGTH_Y/step2);
    Matrix pos((M-1)*(N-1), 2);
    Matrix intensity((M-1)*(N-1), 1);

    int k = 0;
    for(i=0; i < (M-1); i++){
        for(j=0; j < (N-1); j++){
            pos(k,0) = (float)i*step2;
            pos(k,1) = (float)j*step2;
            if(m_cvt_lightfollow)
                intensity(k,0) = (float)(1024*exp(-.00001*(pow((i*step2-xcen
                    ),2)+pow((j*step2-ycen),2))));
            else{
                intensity(k,0) = (float)(1024*exp(-.00001*(pow((i*step2-xcen
                    ),2)+pow((j*step2-ycen),2))));  //rendezvous
                //intensity(k,0)  = (float)(1024*exp(-.00001*pow(((i*step2-
                    xcen)*(i*step2-xcen)+(j*step2-ycen)*(j*step2-ycen)
                    -500*500),2))); //ellipse
                //intensity(k,0)  = (float)(1024*exp(-.00001*pow(((i*step2-
                    xcen)+(j*step2-ycen)),2)));           //diagnal line
                //intensity(k,0)  = (float)(1024*exp(-.00001*pow((j*step2-
                    ycen),2)));           //line along x-axis
                //intensity(k,0)  = (float)(1024*exp(-.00001*pow((-abs(xcen-i
                    *step2)+(j*step2-ycen)),2)));       //v-shape
            }
            k++;
        }
    }
    *newSinfo = pos;
    *newReadings = intensity;
    timeCount++;
}

void CVoronoiTessellation::lightLocation(Matrix sinfo, Matrix readings,
    float xmin, float xmax, float ymin, float ymax, int step){
    Matrix lambda = readings;
    for(int i=0; i<lambda.RowNo(); i++)
        lambda(i,0) = log(lambda(i,0));

    float resmin = 0, res = 0, x, y;
    Matrix Amin(lambda.RowNo(),2), A(lambda.RowNo(),2);

    lightmin(0, 0, sinfo, lambda, &resmin, &Amin);

    for(x=0; x < MASNET_PGPS_PLATFORM_LENGTH_X; x+=step){
        for(y=0; y < MASNET_PGPS_PLATFORM_LENGTH_Y; y+=step){
            lightmin(x,y,sinfo,lambda,&res,&A);
            if(res < resmin){
                resmin = res;
                xcen = x;
                ycen = y;
                Amin = A;
            }
        }
    }
}

void CVoronoiTessellation::lightLocationRec(Matrix sinfo, Matrix readings,
    float xmin, float xmax, float ymin, float ymax, int step){
    Matrix lambda = readings;
    for(int i=0; i<lambda.RowNo(); i++)
        lambda(i,0) = log(lambda(i,0));

    float resmin = 0, res = 0, x, y;
    Matrix Amin(lambda.RowNo(),2), A(lambda.RowNo(),2);

    lightmin(0, 0, sinfo, lambda, &resmin, &Amin);

    for(x=xmin; x < xmax; x+=step){
        for(y=ymin; y < ymax; y+=step){
            lightmin(x,y,sinfo,lambda,&res,&A);
            if(res < resmin){
                resmin = res;
                xcen = x;
                ycen = y;
                Amin = A;
            }
        }
    }

    //calculate smaller step size and window for recursion
    int newStep = step/10;
    xmin = xcen - 5*newStep;
    xmax = xcen + 5*newStep;
    ymin = ycen - 5*newStep;
    ymax = ycen + 5*newStep;

    if (newStep > 1)    //call function again with a smaller step
        lightLocationRec(sinfo, readings, xmin, xmax, ymin, ymax, newStep);
}

void CVoronoiTessellation::lightmin(float x, float y, Matrix sinfo, Matrix
    lambda, float* res, Matrix* A){
    int i;

    Matrix temp(sinfo.RowNo(), 2);
    for(i=0; i<sinfo.RowNo(); i++){
        temp(i,0) = 1;
        temp(i,1) = (float)(-0.5*log(pow((sinfo(i,0)-x),2) + pow((sinfo(i,1)
            -y),2) + 1e-6));
    }

    /*****************************************************************
    The next two lines are two different ways to calculate the P matrix.
    The second line calculates P about twice as fast, but may lose data
    converting size_t to int. This is a bug in the matrix.h template.
    *****************************************************************/
    Matrix P(temp * pseudoInverse(temp));
    //Matrix P(temp * !(~temp * temp) * ~temp);

    Matrix eye(P.RowNo(),P.RowNo());        //set identity matrix
    eye.Null();
    for(i=0; i< P.RowNo(); i++)
        eye(i,i) = 1;

    Matrix Pperp = ~lambda * (eye-P) * lambda;
    *res = (float)Pperp(0,0);
    *A = temp;
}

Matrix CVoronoiTessellation::pseudoInverse(Matrix A){
    int i, j;
    int k=1;

    int numRow = (int)A.RowNo();
    int numCol = (int)A.ColNo();
    int NRow, NCol;

    Matrix Pinv(1,numCol), dk, ck, bk;
    Matrix ak(numRow,1);

    for(i=0; i<numRow; i++)
        ak(i,0) = A(i,0);

    if(!ak.IsNull())
        Pinv = multiply(~ak, (1/dot(ak,ak)));

    while(k < numCol){
        for(i=0; i<numRow; i++)
            ak(i,0) = A(i,k);

        dk = Pinv * ak;
        Matrix T(numRow,k);
        for(i=0; i<numRow; i++)
            for(j=0; j<k; j++)
                T(i,j) = A(i,j);

        ck = ak - (T*dk);

        if(!ck.IsNull())
            bk = multiply(~ck, (1/dot(ck,ck)));
        else
            bk = multiply(~dk, (1/(1+dot(dk, dk)))) * Pinv;

        Matrix N(Pinv - dk*bk);
        NRow = (int)N.RowNo();
        NCol = (int)N.ColNo();
        Pinv.SetSize(NRow+1,NCol);  //resize the rows of Pinv;

        for(i=0; i<NRow; i++)
            for(j=0; j<NCol; j++)
                Pinv(i,j) = N(i,j);
        for(j=0; j<NCol; j++)
            Pinv(NRow,j) = bk(0,j);
        k++;
    }
    return Pinv;
}

Matrix CVoronoiTessellation::multiply(Matrix A, float scalar){
    for(int i=0; i<A.RowNo(); i++)
        for(int j=0; j<A.ColNo(); j++)
            A(i,j) = A(i,j)*scalar;
    return A;
}

float CVoronoiTessellation::dot(Matrix A, Matrix M){
    if(A.RowNo() != M.RowNo() || A.ColNo() != M.ColNo()){
        cout << "Bad dimensions in dot()" << endl;
        exit(1);
    }
    float sum = 0;

    for(int i=0; i<A.RowNo(); i++)
        for(int j=0; j<A.ColNo(); j++)
            sum += (float)(A(i,j) * M(i,j));
    return sum;
}
```

A.3 ChildView.cpp

This code, from `ChildView.cpp`, plots the Tmote and estimated light positions in the pGPS GUI image.

Displays estimated light position and Tmote markers

```
        /**************************************************
            Draw tmote MARKER (added by Shelley 07072008)
        **************************************************/
    if(Voronoi.isCVTLightFollow() && !Voronoi.cancel()){
        if(!Voronoi.isNotFull()){
            Voronoi.updateCVT();    //recalculate CVTs
            Voronoi.flushTmote();   //clears tmote array for new calculation
        }

        int picX, picY;
        RcCore.ptTransPlatform2Pic(Voronoi.getLightX(), Voronoi.getLightY(),
             &picX, &picY);
        picX = (int) (((double) picX) * m_scaleX);
        picY = (int) (((double) picY) * m_scaleY);
        m_drawingDc.SelectObject(&m_LightLocationPen);
        m_drawingDc.Ellipse(picX−10, picY−10, picX+10, picY+10);//draw
            circle at light position

        TmoteMsg* tmote;
        for(int i=1; i <= MASNET_MAX_TMOTES;    i++){   //sensors
            tmote = Voronoi.getTmote(i);
            int picX, picY;
            RcCore.ptTransPlatform2Pic(tmote−>X_POS, tmote−>Y_POS, &picX, &
                picY);
            picX = (int) (((double) picX) * m_scaleX);
            picY = (int) (((double) picY) * m_scaleY);

            m_drawingDc.SelectObject(&m_TmotePenDefault);
            m_drawingDc.Rectangle(picX−3, picY−3, picX+3, picY+3);  //draw
                square at tmote position
        }
    }else if(Voronoi.isCVTFormation() && Voronoi.noMovement()){
        Voronoi.updateCVT();    //recalculate CVTs
    }
```

A.4 MainFrm.cpp

This section contains code from `MainFrm.cpp`. This file holds the function handlers for all messages and commands in the main frame of the RobotCommander GUI. The code shown here include handlers for messages received over the radio and commands selected from dropdown menus.

The beginning of `AM_MOTE_SENSOR_MSG` saves robot data in an array

```
/*****************************************************************
                    handler of AM messages
 *****************************************************************/

afx_msg LRESULT CMainFrame::OnAMReady(WPARAM, LPARAM){
    TOS_Msg tosmsg;
    int length;
    while(RcCore.getAM(&tosmsg, &length) == AMTRANS_OK){
        switch(tosmsg.type){
            case AM_MOTE_SENSOR_MSG:
                {
                MoteSensorMsg* msg =(MoteSensorMsg*)  tosmsg.data;
                Monitor.updateSensorData(msg−>mote, msg);
                Voronoi.setRobot(msg);  //puts robot data in array for CVTs
                    (added by Shelley 07012008)
```

Tmote data handling

```
                case AM_TMOTE_MSG:  // 10122007 added by Shelley for logging
                    TmoteMSG
                    {
                    OscopeMsg* msg =(OscopeMsg*)  tosmsg.data;


                    /* Format sensor data*/
                    CString TmoteID_str;
                    CString lastSampleNumber_str;
                    CString TmoteSensor_str;
                    CString PAR_str;
                    CString X_str;
                    CString Y_str;
                    CString Robot;

                    TmoteID_str.Format          (_T("ID:            %d\n"), msg
                        ->sourceMoteID);
                    lastSampleNumber_str.Format (_T("Last Sample:   %d\n"), msg
                        ->lastSampleNumber);
                    TmoteSensor_str.Format      (_T("Sensor:        %d\n"), msg
                        ->channel );

                    int tmoteCount;     //for loop counter
                    float avg = 0;      //initialise Tmote photo sensor reading
                    float X_POS;
                    float Y_POS;

                    if(msg->channel == 0){
                        for(tmoteCount=0; tmoteCount < OSCOPE_BUFFER_SIZE;
                            tmoteCount++){
                            avg = avg + (int16_t) msg -> data[tmoteCount];
                        }
                        avg = avg/OSCOPE_BUFFER_SIZE;   //caculates average for
                            all 10 photo signals
                                                        //avg = 0 if channel #
                                                            is not 3
                    }

                    switch(msg->sourceMoteID)   //assign x and y position data (
                        mm)
                    {
                        case 21:   X_POS = 590;     Y_POS = 2010;   break;
                        case 22:   X_POS = 590;     Y_POS = 1200;   break;
                        case 23:   X_POS = 590;     Y_POS = 350;    break;
                        case 24:   X_POS = 1800;    Y_POS = 2010;   break;
                        case 25:   X_POS = 1800;    Y_POS = 1200;   break;
                        case 26:   X_POS = 1800;    Y_POS = 350;    break;
                        case 27:   X_POS = 3020;    Y_POS = 2010;   break;
                        case 28:   X_POS = 3020;    Y_POS = 1200;   break;
                        case 29:   X_POS = 3020;    Y_POS = 350;    break;
                        default:   X_POS = 0;       Y_POS = 0;      break;
                    }

                    PAR_str.Format              (_T("Photo Data:    %f\n"), avg
                        );
                    X_str.Format                (_T("PosX:      %f\n"), X_POS );
                    Y_str.Format                (_T("PosY:      %f\n\n"), Y_POS
                        );

                    Voronoi.setTmote(msg, avg, X_POS, Y_POS);   //puts tmote
                        data in array for CVT

                    if(m_bLogTmoteMsg && m_fpLogTmoteMsg != NULL){  //write data
                         to tmote sensor log file

                        fwrite(TmoteID_str,1,TmoteID_str.GetLength(),
                            m_fpLogTmoteMsg);
                        fwrite(lastSampleNumber_str,1,lastSampleNumber_str.
                            GetLength(),m_fpLogTmoteMsg);
                        fwrite(TmoteSensor_str,1,TmoteSensor_str.GetLength(),
                            m_fpLogTmoteMsg);
                        fwrite(PAR_str,1, PAR_str.GetLength(),m_fpLogTmoteMsg);
                        fwrite(X_str,1, X_str.GetLength(),m_fpLogTmoteMsg);
                        fwrite(Y_str,1, Y_str.GetLength(),m_fpLogTmoteMsg);

                    }//end Tmote Sensor log

                    // Communication Window Displays
                    CString msg_disp = "TMOTE to PC: Sensor Array Message";
                    Comm_Out.show_comm(msg_disp);
                    Comm_Out.show_comm(TmoteID_str);
                    Comm_Out.show_comm(TmoteSensor_str);
                    Comm_Out.show_comm(PAR_str);
                    }
                    break;
```

Tmote logging command functionality

```
void CMainFrame::OnLoggingLogtmotemsg() // 10132007 added by Shelley for
    logging TmoteMSG
{

    CMenu* pMenu = GetMenu();
    if(m_bLogTmoteMsg){
        fclose(m_fpLogTmoteMsg);
        m_fpLogTmoteMsg = NULL;
        pMenu->CheckMenuItem(ID_LOGGING_LOGTMOTEMSG,MF_UNCHECKED |
            MF_BYCOMMAND);
        m_bLogTmoteMsg = false;
    }else{
        CFileDialog fdialog(false,
                            _T(".tlg"),
                            _T("TmoteMsg"),
                            OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
                            _T("TmoteMsg Log File (*.tlg)|*.tlg"),
                            this);
        CString fileName;
        if(fdialog.DoModal() == IDOK)
            fileName = fdialog.GetPathName ();
        else
```

```
            return;

        m_fpLogTmoteMsg = fopen((char*)(LPCTSTR) fileName, "wb");
        if(m_fpLogTmoteMsg==NULL){
            MessageBox(_T("Can not open the file"));
            return;
        }
        pMenu->CheckMenuItem(ID_LOGGING_LOGTMOTEMSG,MF_CHECKED |
            MF_BYCOMMAND);
        m_bLogTmoteMsg = true;
    }
}
```

CVT phototaxis and formation control command functionality

```
void CMainFrame::OnCvtLightfollow() //added by Shelley for CVT light follow
    command
{
    CMenu* pMenu = GetMenu();
    Voronoi.setCVTFormation(false);

    if(m_cvt_lightfollow_check){
        pMenu->CheckMenuItem(ID_CVT_LIGHTFOLLOW, MF_UNCHECKED | MF_BYCOMMAND
            );

        for(int t=0; t < MASNET_PGPS_NUM_OF_ROBOTS; t++){   //stops all
            robots
            RcCore.stopRobots(&t,1);
        }
        m_cvt_lightfollow_check = false;
        Voronoi.setCVTLightFollow(false);
    }else{
        Voronoi.setCVTLightFollow( Voronoi.initialize() );

        if(Voronoi.isCVTLightFollow() && !Voronoi.cancel()){
            pMenu->CheckMenuItem(ID_CVT_LIGHTFOLLOW, MF_CHECKED |
                MF_BYCOMMAND);
            m_cvt_lightfollow_check = true;
        }else{
            pMenu->CheckMenuItem(ID_CVT_LIGHTFOLLOW, MF_UNCHECKED |
                MF_BYCOMMAND);
            m_cvt_lightfollow_check = false;
            return;
        }
    }
}

void CMainFrame::OnCvtFormation() //added by Shelley for CVT formation
    control command
{
    CMenu* pMenu = GetMenu();
    Voronoi.setCVTLightFollow(false);

    if(m_cvt_formation_check){
        Voronoi.setTime();
        pMenu->CheckMenuItem(ID_CVT_FORMATION, MF_UNCHECKED | MF_BYCOMMAND);
        for(int t=0; t < MASNET_PGPS_NUM_OF_ROBOTS; t++){   //stops all
            robots
            RcCore.stopRobots(&t,1);
        }
        m_cvt_formation_check = false;
        Voronoi.setCVTFormation(false);
    }else{
        Voronoi.setCVTFormation(true);
        pMenu->CheckMenuItem(ID_CVT_FORMATION, MF_CHECKED | MF_BYCOMMAND);
        m_cvt_formation_check = true;
    }
}
```

Appendix B

MASnet User's Manual, Version 2.0 [1]

As with most experimental hardware, MASnet is complex, unproven and only lightly documented which makes the learning curve very steep for new users. This appendix is to aid those who will use the MASnet platform as illustrated in fig. B.1. For a quick start guide to cover everything required to make the robots move; see sec. B.2.

B.1 MASnet Overview

The MASnet project at Utah State University combines a wireless sensor network with a mobility platform. That is, a large number of robots can serve both as environmental sensors and actuators. The objective of MASnet is to develop systems that can collect information and respond to spatially distributed diffusion processes such as a chemical plume dispersement [44]. Each robot has limited sensing, computation, and communication abilities, but they can coordinate with each other to study challenging tasks like formation keeping, environment monitoring, consensus algorithms, and swarm intelligence.

The MASnet system is comprised of two main components and their associated tasks, as listed below:

- **Base-Station:** Image processing, serial to programming board communication, pGPS information broadcasting, and command issuing.
- **Robots:** Inter-robot and robot to base-station communication, data collecting, mobility, PWM signal generation, and encoder counting.

[1]The original version of this user's manual can be found in the Appendix of William Bourgeous' thesis [1].

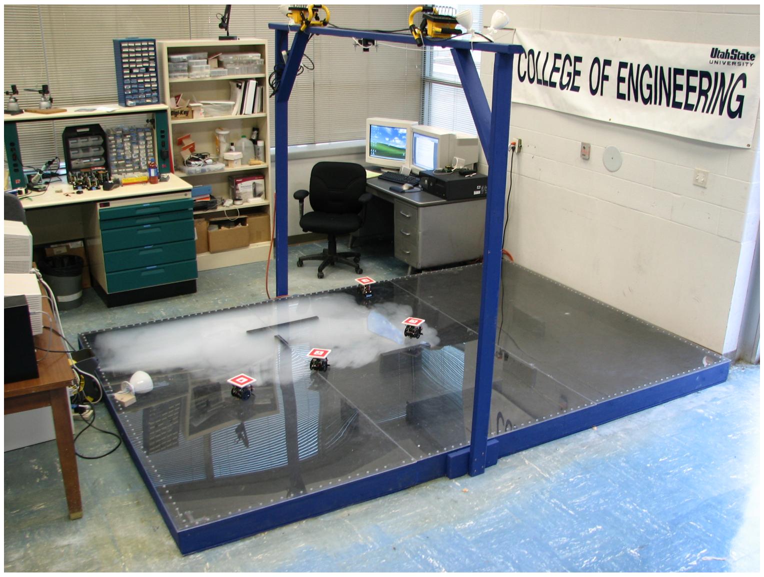


Fig. B.1: MASnet mobile robot experimental platform.

B.1.1 Base-Station

The base-station is designed to provide absolute position information to each robot and issue command and control communication. An overhead CCD camera is hung about 72 inches above the platform. The function of the camera is to capture an image from which each robot's global position and orientation can be extracted. Image processing and all functionality is performed by a C++ MFC application written explicitly for this purpose called RobotCommander; see fig. B.2.

RobotCommander localizes each robot on the platform by a marker on top of the robot. The marker features a rectangular frame and a special symbol pattern inside the frame. The RobotCommander first detects the frame and then tries to match the inside pattern with its pattern repository. The id, position, and orientation of each marker are recorded.

This localization subsystem is based on a modified ARToolKit [31]. The HSB color model is adopted for the color segmentation so the effect of illumination is reduced and any specified color can be used for markers.

The implementation of this system uses DirectX technology from Microsoft Corp. Therefore, the image processing can work in multi-threading and pipeline fashion. The process includes lens distortion compensation, marker detection, and screen rendering, while

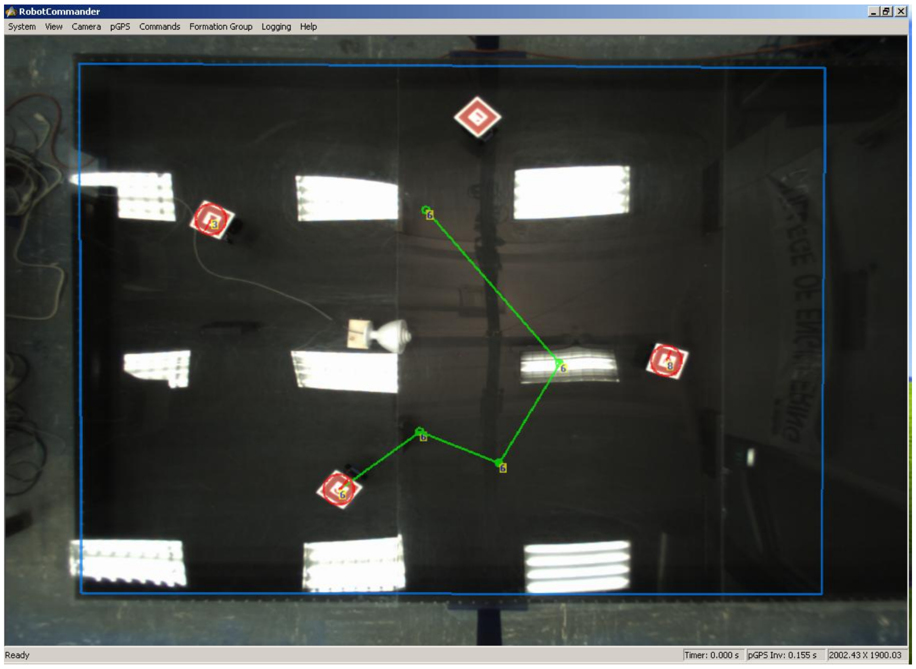


Fig. B.2: RobotCommander GUI screen capture.

the performance can be as high as 150 ms per frame. The transformation between the image coordinates and the world coordinates is found through an extrinsic camera calibration. The average errors of position and orientation are about 1.7 cm and 1.2, respectively. The result of the localization is broadcast to all robots. The complete structure of RobotCommander is illustrated in fig. B.3

B.1.2 MASnet Robots

The robots are two-wheel, servo driven, differentially steered robots developed by CSOIS specifically for MASnet; see fig. B.4(a). The robots are designed to be simple and low-cost. Two modified Futaba S2954 servo motors are used to drive each robot. Four first generation MicaZ robots use Futaba S2954 servo motors. The remaining six second generation MicaZ robots use GWS S03N 2BB servo motors. The robots have two high resolution commercial optical encoders, called WheelWatcher [19], with a resolution of 128 counts per revolution. The robots have a Plexiglas® fixture on top to mount an identification maker.

The robots use a commercially available MicaZ mote, from Crossbow [17], for all

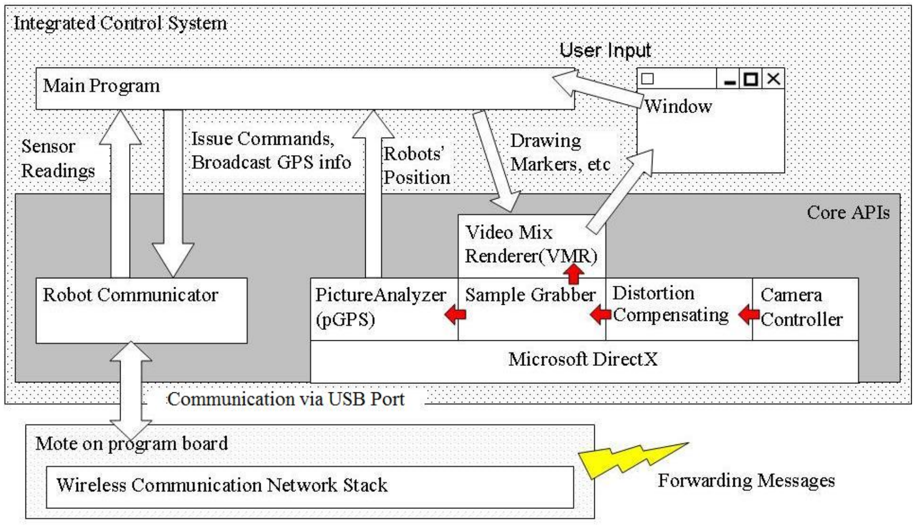


Fig. B.3: RobotCommander software diagram.

functions including computation, sensing, and communication; see fig. B.4(b). From this, the robots are also referred to as MASmotes [16]. The main CPU is an ATmega 128L(8 MHz) with 128KB programmable flash memory, 4KB EEPROM, and 512KB flash memory for measurements. It also has reconfigurable PWM outputs, eight 10-bit ADC channels, and multiple data interfaces including I2C, SPI and UART. Communications are handled through a 2.4 GHz, IEEE 802.15.4/ZigBee compliant RF transceiver IC, CC2420. The communication rate can be as high as 250 kbps. The robot operating system is TinyOS which was developed by UC Berkeley parallel to mote hardware technology. TinyOS is an event-driven operating system, which supports non-preemptive multi-tasking and is programmed in nesC; see fig. B.5. External sensors currently include IR object detection sensors for collision avoidance and light sensors to track a diffusion process.

B.2 Quick Start

Quick start steps include:

1. Setup and start RobotCommander:

   (a) Start the base-station computer, the username is **csois**, obtain the current password from an administrator.

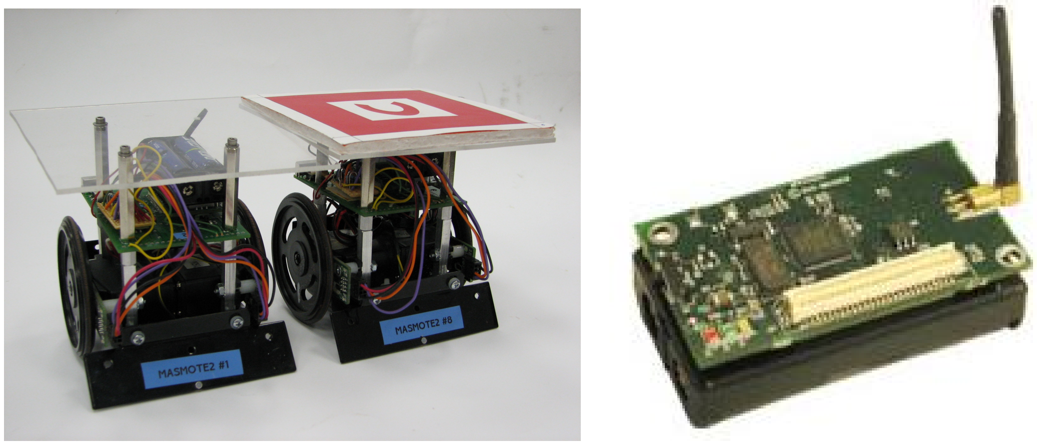

(a) (b)

Fig. B.4: MASnet robots (a) and MicaZ wireless sensor network mote from Crossbow (b).

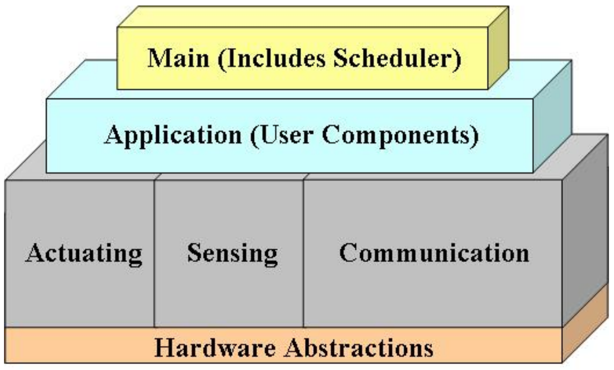


Fig. B.5: TinyOS program structure.

(b) Place the base-station mote, programmed with *TOSBase*, on the programming board located beside the base-station computer. Take care to properly seat the white 52-pin connector. A MicaZ mote that is not completely connected can cause damage to the mote and programming board as well as cause system malfunctions. Make sure the mote is switched off or has no batteries to avoid further damage to the mote. A green LED on the programming board will turn on when the mote is completely connected to the programming board.

(c) Launch RobotCommander from either the taskbar or the shortcut on the desktop.

(d) From the main menu, select **System→Setup→Dectection** tab. Verify that a non-zero number is present in the Broadcast text box. If a zero is present, no pGPS messages will be broadcast and the robot will not know its global position. A pGPS broadcast of every 250 ms is typical. Close the **Setup** box. If any value in the **Setup** window is changed, stop RobotCommander and restart it so that the changes will take effect.

(e) Select **System→Start** from the main menu to start the program. An image from the pGPS camera should now be visible. If an error message appears instead of the image, restart the computer. Occasionally, the base-station computer does not recognize the pGPS camera signal till it is restarted.

2. Command a robot to move:

   (a) Program a MicaZ mote with the desired application. See sec. B.4 for instructions on mote programming.

   (b) Select the MASmote robot corresponding to the MicaZ number and connect the mote to the sensor board. Turn on the mote at the front of the robot, then the motor board at the rear of the robot. If done in the reverse order, the encoders will give a bogus reading. The large rectangular red and green LEDs on the back of the robots should now be on.

(c) Identify the robot number on the front scoop and find the corresponding marker; numbers are hand written in the top left corner of each marker. Make sure the the robot number, mote number, and marker number all correspond to keep the MASnet platform organized. Place the robot with its marker on the platform at least 0.25 meters away from the edge.

(d) The marker should be identified by RobotCommander and a red circle with a line indicating orientation is drawn on the image at the location of the robot. The robot number is also printed in the red circle. If the robot is not recognized, see sec. B.3.1 to adjust the camera settings.

Note. *Currently, only markers 1, 3, 6, and 8 can be recognized by RobotCommander. Go to* **System→Make Pattern** *to add more markers to the RobotCommander registry.*

(e) Select the robot by left clicking it on the RobotCommander screen. The red identifying circle should now turn blue. Right click any place you would like the robot to move. A green line should extend from the robot to the desired position and the robot will begin to move using its PI controller to arrive at the selected target point. If the robot does not move, see sec. B.6 to pinpoint the problem.

B.3 RobotCommander

B.3.1 Functions and Features

The RobotCommander is written in C++ using MFCs to build the multi-window application. The core application is constantly processing images from the pGPS camera while all other functionalities are event driven. A standard windows type menu system is used to navigate and use RobotCommander's added functions and features. This section provides an explanation of each option available in RobotCommander's main menu.

- **System** includes all parameters and commands that change the RobotCommander system.
  - **Start** starts the RobotCommander program.
  - **Stop** stops the RobotCommander program.
  - **Setup** sets parameters before starting RobotCommander. See sec. B.3.2 for a detailed explanation of the **Setup** process.
  - **Find 4 Corners** allows RobotCommander to find the four corners of the platform for camera calibration. See sec. B.3.3 for camera calibration steps.
  - **Make Pattern** adds new marker patterns for RobotCommander to identify. Follow the Make Pattern steps to add more robots to the platform.
  - **Eval Error** evaluates the camera distortion error. This should not be done on the platform, but in a simulated environment described in PungYu Chen's thesis [2]. This function was only used for the initial setup of the MASnet platform.
  - **Save Image** saves a Bitmap file of the current pGPS image.
- **View** displays data in the current window or other windows available in RobotCommander.
  - **Control Panel** provides detailed data about the state of each robot in operation and can be used to issue commands.
  - **Show Labeled** creates a child window that shows the image after it has been filtered according to the parameters set in the **Major Tweak** window. This is useful when tuning the camera filter because it allows the user to see what the computer sees.
  - **Display Comm** displays incoming or outgoing messages in a child window.
  - **L-Matrix Editor** changes the communication topology for consensus experiments. This window has not been fully developed; see the source code for details.

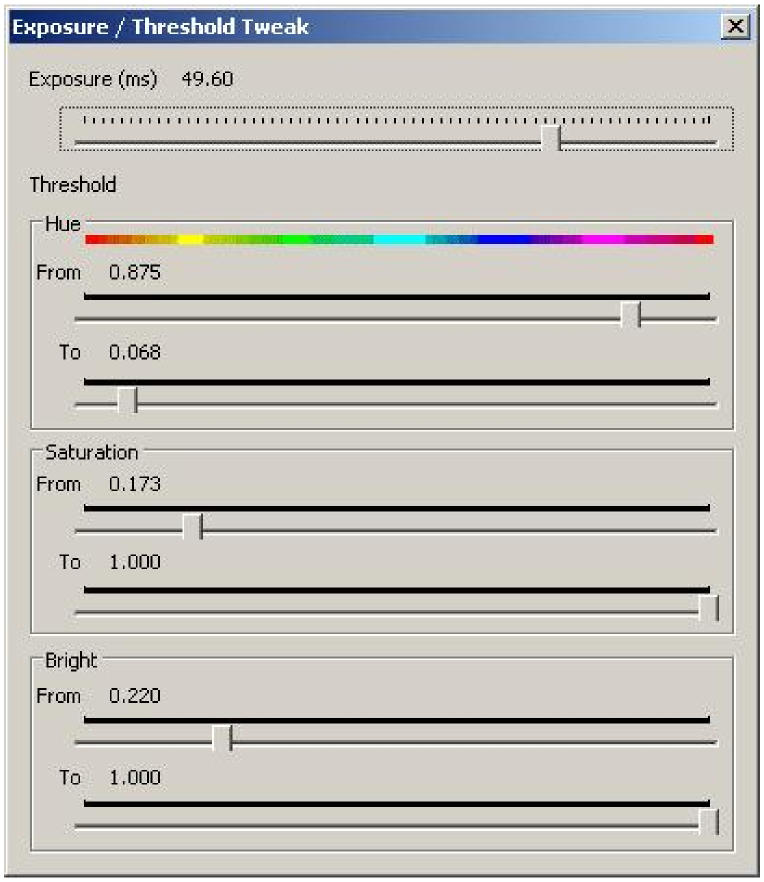
  - **Show pGPS Data** displays pGPS in a child window.
  - **Compare Position Data** compares the position data derived by RobotCommander on the left and the position data sent from a robot back to the basestation. This was primarily used to fix encoder problems.
  - **Robot Destination** decides whether **All**, **None**, or only **Selected** robots will show their destination. **All** is typically selected.
  - **Clear Fog Map** clears a fog map used for the TinyOS Tech Exchange demo. This option is obsolete, but can be reinstated with some source code manipulation.

- **Camera** changes image and detection performance for the pGPS camera.
  - **Camera Adjust 1 and 2** controls driver settings on the actual camera.
  - **Major Tweak** opens a window that is only activated by pressing the *alt* key. Parameters of the image filter can be adjusted in this window; see fig. B.6. The hue, saturation, and brightness parameters can be used to select a limited range of colors, red in our case. These parameters are rarely changed but the exposure time must be adjusted frequently throughout the day as the ambient light levels change. This is by far the most important adjustment on the MASnet vision system. Use the **Show Labeled** window to see how adjustments affect the filtered image.

Note. *For better robot detection, keep the* **Major Tweak** *settings the same as fig. B.6, except for the exposure time.*

  - **Enable Undistortion** turns on the undistortion feature which removes barrel distortion from the image.

- **pGPS**


Exposure / Threshold Tweak

Exposure (ms) 49.60

Threshold

Hue
From 0.875
To 0.068

Saturation
From 0.173
To 1.000

Bright
From 0.220
To 1.000

Fig. B.6: Major Tweak window used to adjust vision parameters.

  - **Detect Now** searches for robot markers in the field of view once.
  - **Broadcast Now** immediately broadcasts pGPS information for all robots identified.

- **Commands**

  - **STOP ALL** stops motion and resets some states of the robot. The button is considered the emergency stop of the system and is frequently used in experiments.
  - **Record Paths** turns on the **Record Paths** function to track robot paths. A small dot is placed at the center of the robot at each time step. To clear the screen click on record paths again.
  - **Rendezvous** broadcasts a **Rendezvous** packet once. Originally it was used to trigger the simultaneous start of the rendezvous experiment but has since been used to trigger the start of several different experiments, depending on the robot code.
  - **RandomWalk** broadcasts a **RandomWalk** packet once for robots to randomly traverse the platform. Only robots programmed with a random walk command can respond, which exclusively includes applications written by Florian Zwetti.

Note. *To see what commands or message types are included in each robot application, check under* MASNET_MSG_TYPE *in the* masnet_Messages.h *file.*

  - **Phototaxis** broadcasts command packets for robots to perform behavioral dynamic phototaxis. Only robots programmed with a random walk command can respond, which exclusively includes applications written by Florian Zwetti.
  - **Consensus** should broadcast a **Consensus** command for the robots to perform formation control. However, there is currently no code written for any robot to respond to this command.

  - **CVT** begins a **CVT Light Follow** or **CVT Formation** command to perform phototaxis or formation control using centroidal Voronoi tessellations.

- **Formation Group** encompasses tools used for formation maneuvers developed during early MASnet projects.

- **Calibration** tests the performance of the robot controller by a **Turn Around** or **Go Straight** command. These commands are only included in certain applications.

- **Logging** gathers and save information during experiments.

  - **Log pGPS** records the pGPS information of all visible robots at each interval in a text file.
  - **Log SensorMsg** records all sensor messages sent from the robots to the base-station. This file can then be read by one of the MATLAB scripts in the *MASnet Log Plotter* folder located in `C:\MASNET`.
  - **Log CollectMsg** records all "collect messages" sent from the robots to the base-station. This option is obsolete.
  - **Log DebugMsg** records all debug messages sent from the robots to the base-station.
  - **Log TmoteMsg** records all Tmote messages sent from the robots to the base-station. This file can then be read by one of the MATLAB scripts in the *MASnet Log Plotter* folder located in `C:\MASNET`.

B.3.2 Setup

In the RobotCommander Setup menu there are five tabs:

- **Camera** - Default camera resolution is 1280x1024 the result of changing this is unknown. The default capture rate is 15 frames per second. This parameter may be increased or decreased. However, this is a desired rate and is limited by the processor.

- **Calibration** - The **Distortion Parameters** should not be altered. The **Camera Matrix** block contains the intrinsic parameters of the cameras. The default matrix is

$$\begin{bmatrix} 713.484 & 0 & 691.651 \\ 0 & 712.373 & 496.98 \\ 0 & 0 & 1 \end{bmatrix}.$$

- **Detection** - The detection rate should be greater than or equal to the camera's capture rate. In the **Broadcast** text box, a zero means no pGPS will be sent where a -1 will cause the pGPS messages to be sent as frequently as they are updated. A specific time interval for the pGPS broadcast can be entered in milliseconds. If **extended format** is checked, pGPS information for only two robots can be packed into one transmission. If **compact format** is checked, one transmission can contain pGPS information for up to four robots. If pGPS information for all robots does not fit in one transmission, RobotCommander will make multiple transmissions at each update. RobotCommander can support up to 10 robots.

- **Platform** - The platform size is the actual dimensions of the blue rectangle painted on the screen and is entered by the user to perform camera calibration; see sec. B.3.3. All other parameters in this tab are calculated by the camera calibration routine.

- **Formation** - This tab is used to control formation control parameters used for early MASnet projects.

- **Test Markers** - This tab was used to configure camera calibration experiments.

B.3.3 Calibration

This section describes how to calibrate the cameras extrinsic parameters assuring an accurate coordinate transform from the camera's image to real world coordinates. This calibration should be executed monthly or anytime the camera is moved.

1. Define the boundaries of the platform:

    (a) Start RobotCommander.

    (b) Place a corner marker in each corner of the desired platform size. All four corner markers have an "L" symbol, a balsa wood block, and an orange base. Markers are usually placed at the extremities of the platform but may be placed to form any size rectangular area. Smaller work areas are more accurate that larger work areas.

    (c) Select **System→Setup→Platform** tab. Enter the x and y platform dimensions in millimeters measuring from the corner markers placement. Click **OK**.

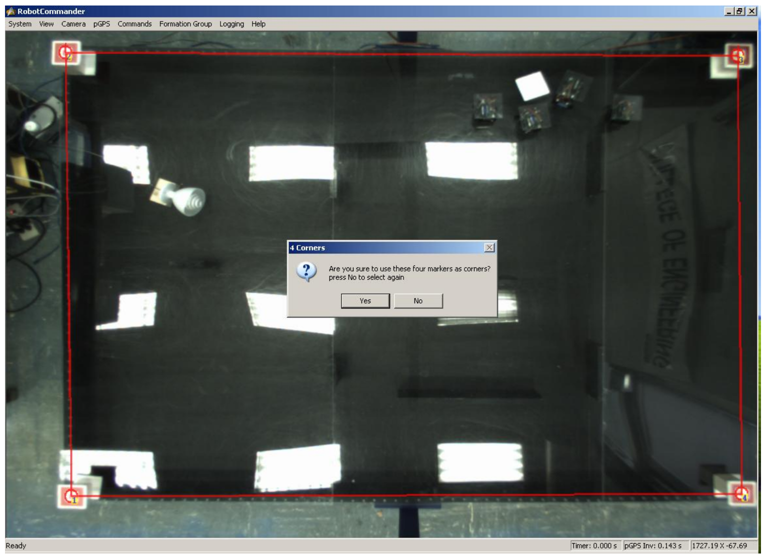
2. Calibrate the camera within the platform boundaries:

    (a) Select **System→Find 4 Corners** from the menu. RobotCommander will search for the corner markers in a predefined order. If all markers are not found, adjust the camera settings in **Major Tweak** or reduce the size of the calibration area.

Note. *It is difficult for all robot markers to be detected at one time. It would be easier to hold detected robot markers till all four have been found. To do this, the* detectRobots *function under the* CPictureAnalyzer *class, found in file* PictureAnalyzer.cpp*, must be altered.*

    (b) Once all four corners are found and a red rectangle is drawn on the screen connecting the corners, double click anywhere in the screen. A verification box will appear, select **Yes**; see fig. B.7. The calibration is now complete.

    (c) The calibration can be verified by moving the mouse to a position on the platform and comparing the calculated mouse coordinates in the bottom left corner of RobotCommander to actual measurements.



Are you sure to use these four markers as corners?
press No to select again

Timer: 0.000 s | pGPS Inv: 0.143 s | 1727.19 X -67.69

Fig. B.7: Camera calibration.

B.4 MicaZ Programming

The robots are programmed in the nesC 1.1 language which was developed for the TinyOS operating system. Since the summer of 2007 the old TinyOS software platform has been updated to the MoteWorks software platform on the base-station computer. Details of MoteWorks and a free download is available at the Crossbow website [17]. Because MoteWorks requires an updated version of Cygwin to operate, all old applications must be updated to the MoteWorks platform to work. All MoteWorks program files for MAS-net are stored on the base-station hard drive at `C:\Crossbow\cygwin\opt\MoteWorks\apps\general` and all old TinyOS files are located at `C:\ProgramFiles\UCB\cygwin\opt\tinyos-1.x\apps`. There should be shortcuts on the desktop for both folders (**MoteView apps** and **TinyOS apps**). As these are community files, please do not make changes to these folders or files. When you decide to make program changes, copy a known good program, rename it with your name first and then edit it. Programmers Notepad 2 is typically used to edit and program the motes. Vim is also available for editing and any other text editor can be used. Documentation of the nesC language can be found in*nesC 1.2 Language Reference Manual* [28].

The following is a list of programs from the MoteWorks platform included on the base-station computer [59].

Cygwin - A Linux-like environment for Windows. This can be used to compile and program motes.

Programmers Notepad - An Integrated Development Environment for code editing, compiling, programming, and debugging motes.

XSniffer - A Network Monitoring Tool that displays incoming packets in a user-friendly GUI. This program is useful to test if the gateway mote is receiving wireless packets.

MoteConfig - A GUI environment for Mote Programming and Over The Air Programming. MoteConfig can only program integrated applications and cannot program new robot application.

MoteView - A GUI environment for viewing mesh networks. MoteView is an upgrade from SergeView and is similar to Moteiv's Trawler application.

For more information on these applications beyond this user's manual, see *MoteWorks Getting Started Guide*, *XServe Users Manual*, and *XMesh Users Manual* [59–61].

There are two different ways to program a MicaZ mote with new robot applications included in the MoteWorks package. The programmer can either use Cygwin or Programmers Notepad. Programmers Notepad is typically used because the user can edit, compile, and program a mote easily. The following sections describe how to program in both cases.

B.4.1 Cygwin

To program a MicaZ with Cygwin:

1. Make sure the mote is switched off and RobotCommander is closed. Having the mote on while programming can damage the mote, programming board, and the computer. Also, programming while RobotCommander is running will cause the program to freeze.

2. Place the mote on the programming board making sure it is completely connected. A MicaZ mote that is not completely connected can also cause damage to the mote and programming board as well as cause system malfunctions. A green LED on the programming board will turn on when the mote is completely connected to the programming board.

3. If you would like to program an old application not yet in the **MoteView apps** directory, copy and paste the files from the **TinyOS apps** directory to the **MoteView apps** directory. Delete the current makefiles and copy and paste the makefiles from *VoronoiPYMicaZ*. If your application is already in the **MoteView apps** directory, skip to the next step.

4. Before programming a mote with a robot application, open the *config.h* file for the desired application in a text editor. Change the `ROBOT_NUMBER` on line 26 to the robot you are programming. Each robot has its own parameters which is included in *config.h*.

5. Start the Cygwin environment from the desktop. This is a Linux environment so all commands are entered as text. Table B.1 shows some useful commands in Cygwin.

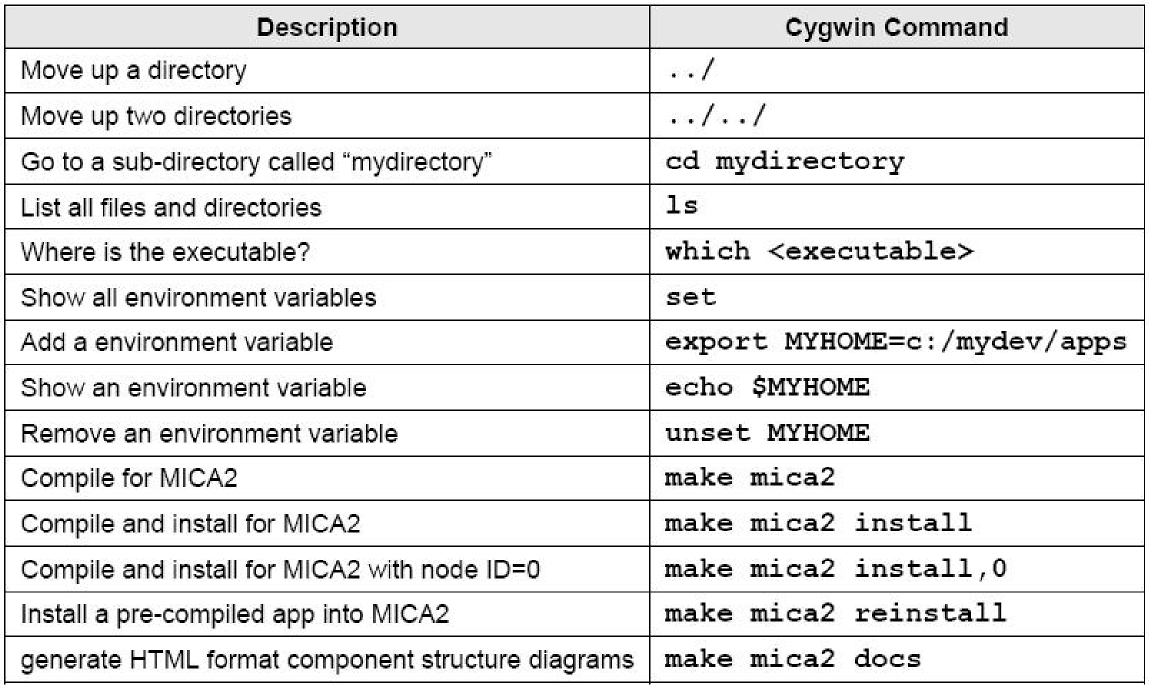
6. Use **cd** to change the path to `C:\Crossbow\cygwin\opt\MoteWorks\apps\general\[DesiredApplication]`.

7. Type `make micaz install.1` to compile and load the program into the mote's memory. The number at the end of the install command indicates the robot number and should be changed appropriately. See fig. B.8 for an illustration of Steps 5 through 7.

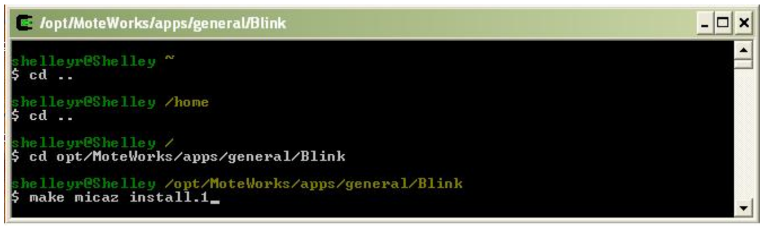
Note. *It is important to know other basic commands for compiling and installing programs for MicaZ motes. Here is a short list.*

make micaz install *compiles and installs the program into the mote.*

Table B.1: Some useful Cygwin commands (from MoteWorks Getting Started Guide).



Description	Cygwin Command
Move up a directory	`../`
Move up two directories	`../../`
Go to a sub-directory called “mydirectory”	`cd mydirectory`
List all files and directories	`ls`
Where is the executable?	`which <executable>`
Show all environment variables	`set`
Add a environment variable	`export MYHOME=c:/mydev/apps`
Show an environment variable	`echo $MYHOME`
Remove an environment variable	`unset MYHOME`
Compile for MICA2	`make mica2`
Compile and install for MICA2	`make mica2 install`
Compile and install for MICA2 with node ID=0	`make mica2 install,0`
Install a pre-compiled app into MICA2	`make mica2 reinstall`
generate HTML format component structure diagrams	`make mica2 docs`



```
shelleyr@Shelley ~
$ cd ..

shelleyr@Shelley /home
$ cd ..

shelleyr@Shelley /
$ cd opt/MoteWorks/apps/general/Blink

shelleyr@Shelley /opt/MoteWorks/apps/general/Blink
$ make micaz install.1_
```

Fig. B.8: Robot programming in the Cygwin environment.

make micaz *only compiles the program. This command is useful to check for syntax errors while writing new code.*

make micaz reinstall *only installs the program onto the mote. This command can speed up the process of installing the same code on several motes, but it will not work if the program has never been compiled before.*

B.4.2 Programmers Notepad

To program a MicaZ with Programmers Notepad:

1. Open Programmers Notepad on the desktop. This will be used to edit and install your programs to MicaZ motes.

2. Select a file from your desired application in the **Projects** window on the left. The file will appear on the right; see fig. B.9.

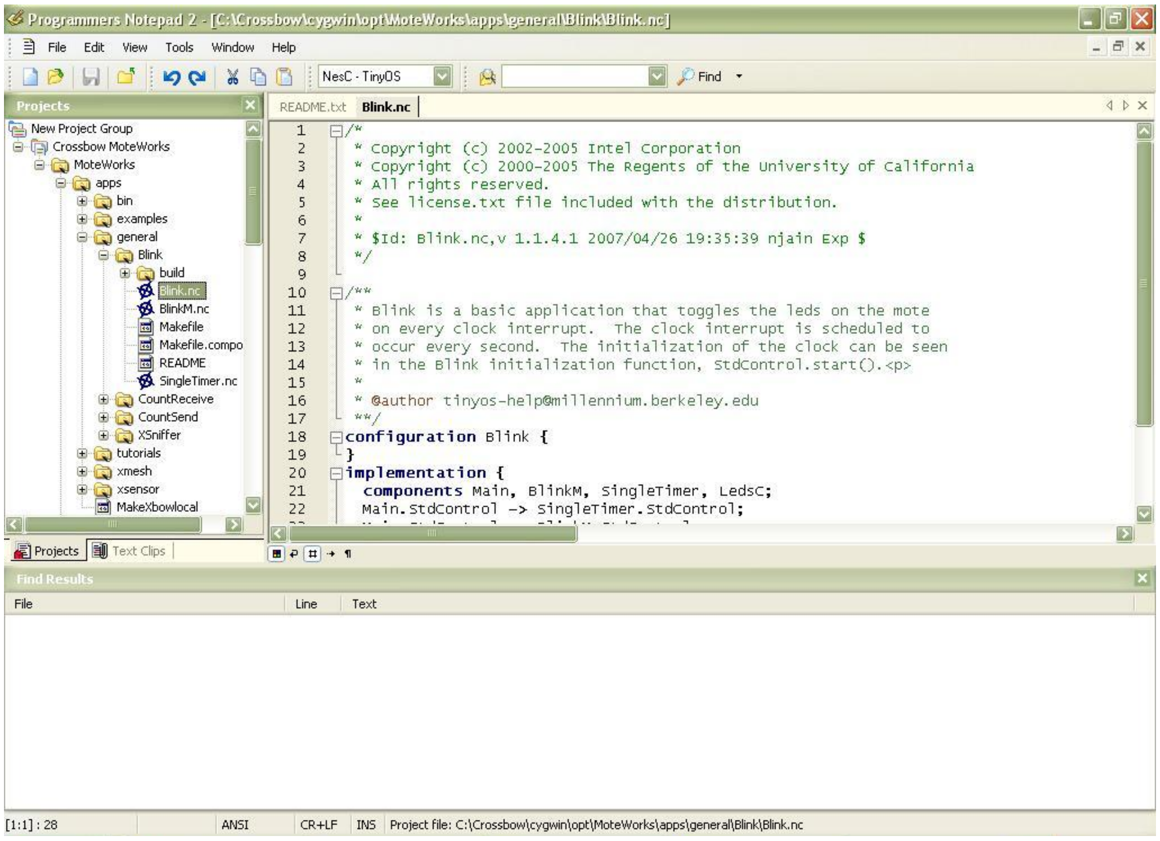
3. Follow Steps 1 through 4 from the previous section using Programmers Notepad as the text editor.

4. Make sure the curser is on a file that is part of your desired application. From the main menu, select **Tools→shell** (or type *F6*).

5. In the command shell, follow Step 7 from the previous section to install the program to the mote.

If errors are found in the program code they will be displayed. If the program loads but fails during the verification process, it is most likely a bad connection between the mote and the programming board. To repair this, secure the connection and load the program again.

Finally, the gateway mote is designated with the number 11. If RobotCommander is closed before stopping the gateway mote, it may need to be reprogrammed with *TOSBase*. To reinstall the gateway mote, type `make micaz reinstall.11`.



```
/*
 * Copyright (c) 2002-2005 Intel Corporation
 * Copyright (c) 2000-2005 The Regents of the University of California
 * All rights reserved.
 * See license.txt file included with the distribution.
 *
 * $Id: Blink.nc,v 1.1.4.1 2007/04/26 19:35:39 njain Exp $
 */

/**
 * Blink is a basic application that toggles the leds on the mote
 * on every clock interrupt.  The clock interrupt is scheduled to
 * occur every second.  The initialization of the clock can be seen
 * in the Blink initialization function, StdControl.start().<p>
 *
 * @author tinyos-help@millennium.berkeley.edu
 **/
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;
  Main.StdControl -> SingleTimer.StdControl;
```

Fig. B.9: Robot programming in Programmers Notepad.

For more documentation on MicaZ robot testing, refer to the Developer's Manual for MASmotes, by Zhongmin Wang, and other documents in the *MicaZ_Robot_Document* folder.

B.5 Tmote Programming

Currently, Tmotes are used solely for the CVT phototaxis experiment as the light sensor array programmed with the application *SenseLight*. Tmotes are programmed in the nesC 1.2 language. However, because Tmotes require a different instance of Cygwin, the TinyOS package for Tmotes cannot exsist on the same computer as the MoteView package for MicaZs. For the MASnet platform, the TinyOS for Tmotes is installed on the computer beside the base-station desk. The username is **CSIOS**. Obtain the current password from an administrator.

All TinyOS program files for Tmotes are in `C:\cygwin\opt\moteiv\apps`. There should be a shortcut on the desktop called **tinyOS apps**. Please do not make changes to these folders or files. When you decide to make program changes, copy a known good program, rename it with your name first and then edit it. Vim is typically used to edit the programs, but other text editor can be used.

To program a Tmote for MicaZ cross-communication:

1. Connect a Tmote to the computer. This computer has a USB extension for convenience.
2. Start the Cygwin environment from the desktop.
3. Change the path to `C:\cygwin\opt\moteiv\apps\[DesiredApplication]`.
4. Type `make telosb install.21` to compile and load the program into the mote's memory. The number at the end of the install command should correspond to the number on the mote. The mote DOES NOT need to be off during this process.

B.6 Troubleshooting

There are many things that can go wrong on the MASnet platform. This section helps pinpoint the problem when robots are not moving as they should.

Possible code error

First, check to see if it is not an error in the code by giving a position command in RobotCommander. If the robots move to that spot, there is a problem with the code. Remember, nesC runs concurrently and if two motor commands are given at once, the robot will not move.

MicaZ lock-up or connection problem

If your robot still will not move as it should after the first step, check the mote connected to the robot. Occasionally, the mote may be bumped out of place when colliding with other robots or running between Plexiglas® sheets. If the mote is not fully connected, it can cause odd robot behavior. The problem could also be caused by the MicaZ processor locking-up. The red LED on the MicaZ should toggle every time message is received and the green LED should toggle every time a message is sent. If these lights are not blinking, the processor has locked-up. Simply, turn of the motors and mote, then turn them back on again to stop the lock-up.

Low battery voltage

If this is not a problem, replace the four AA batteries under the robot. The brightness of the red LED on the back of the robot is a good indicator of how much voltage is applied by those four batteries. If it is too dim, the batteries should be replaced. The two AA batteries on the mote do not need to be replaced as often since two 1.5 V batteries should last up to one year. However, the green LED on the back of the robot indicates how much voltage these two batteries supply. If it is too dim, the mote batteries should be replaced.

Because the batteries on the bottom of the robot should be changed often, it is important to keep charged and uncharged batteries separate while always keeping a charger working. An organized system will keep the MASnet platform running efficiently. A box of separated charged and uncharged batteries is on the bookcase between the MASnet base-station desk, and the MASnet bench. A charger, along with a multi-meter to check battery voltage, should be on the MASnet bench.

Limited mote range

Finally, if the problem remains, the MicaZ may have limited range. If not handled properly, a voltage spike may occur during mote programming. This spike can limit the range of the mote. Fortunately, motes with limited ranges can repair themselves, but may take some weeks before they are restored to full range. These motes are called sick motes. Do not use these motes, with the exception of testing their range, till they are fully functional.

If motes have short ranges the green LED on the MicaZ should toggle while the red LED is static. In other words, the mote should send messages, but not receive them at longer ranges. Bring the robot next to the gateway mote while in operation. If the red LED begins to toggle at very close range, then the mote is sick and cannot be used.

To test the range of more motes, program transmitting motes with *CountSend* and program receiving motes with *CountReceive*. The application *CountSend* uses a binary counter, where the first three bits are displayed by the LEDs, and transmits the number through the radio. The application *CountReceive* receives that number and displays the first three bits on its LEDs. Basically, motes programmed with *CountReceive* should mimic the mote programmed with *CountSend*. Once the receiver stops mimicking the sender, both motes have exceeded their transceiver range.

B.7 Adding New Functionality to the Platform

If a new task has to be introduced to generate a new behavior for the MASmotes, modifications are most likely required on both the C++ RobotCommander program that runs the base-station and the nesC program that runs the robots.

B.7.1 Adding a New Message to the Robot nesC Program

Since telling the robot to perform a specific task consists of sending a message from the base-station mote to the moving mote on the platform, an interface for transferring the message has to be created. This can be done by defining an instance of the *ReceiveMsg* interface as *[Task]Msg* in file *robotMainM.nc*, where *[Task]* denotes the name of the task.

The newly created interface has to be connected in the configuration file of the application, which is *robotMain.nc*. Here, the interface is used by the module *robotMainM*, which must be connected to the *ReceiveMsg* interface of the communication module. Since this is a parameterized interface, appending data has to be included in brackets.

Now, program the robot behavior as a task in the definition file for the associated incoming message which is done in the file *incoming_msg.nc*. Tasks, which normally has no parameters to return, are defined. Timers can also be called, which can activate events when fired.

A new event of receiving a message should be defined to later perform the specific task. This has to be done, since the command from RobotCommander should activate this certain behavior, which is done by passing a message through the base-station mote to the motes on the platform. The event is of type *TOS_MsgPtr*.

An event is activated, when the interface *[Task]Msg* receives a message. The task is put into the task queue and the event closed again. When creating an event, attention should be paid to the fact that the event should be as short as possible. It should only pass the Task into the task queue and then close again.

Next, update the MASnet message list to identify the message for the new behavior in the file *masnet_Messages.h*. The message identification list has to be extended by adding the message with a specific number `x` under the `MASNET_MSG_TYPE`. Then a new variable type has to be created with the `typedef` syntax.

B.7.2 Adding a New Command to RobotCommander

After modifying the robot behavior, RobotCommander must also be modified to be able to activate the new behavior. In order to make sure that the current version of RobotCommander is not affected during modification, it is recommended to create a new folder and to copy all files in the existing RobotCommander directory to the newly created one. Currently, the latest instance of RobotCommander is called *Shelley_RobotCommanderLightInterp*.

Start Microsoft Visual Studio.Net and open the newly copied project file *RobotCommander.vcproj*. Click the **solution explorer** icon, which will display a treelike folder structure of the project with source, header and resource files. Now, click on the **resource view** structure and then the **Menu** folder; finally, select *IDR_MAINFRAME*. A new command can be implemented by opening the main frame display and choosing the **Commands** menu, then typing in a command name; see fig. B.10. After successfully assigning a name to the new task, the right mouse button has to be used on the new menu item to choose the option **Add Event Handler**; see fig. B.11. In the **Event Handler** window a function handler name is assigned automatically. This name normally does not have to be changed. The same holds for the Message type, which is assigned to `COMMAND`. After choosing the *CMainFrame* class in the class list, the actual code of the event can be inserted by clicking on the button **Add and Edit**.

After closing the event handler with **Add and Edit**, the file *MainFrm.cpp* is automatically opened and the code of the function can to be added to the *CMainFrame* class. Therefore, the following lines are created automatically and the code can easily be inserted inside. After finishing the function, a look at the top of *MainFrm.cpp* shows that an according `ON_COMMAND` instruction was added automatically to the message map. If the class definition of *CMainFrame* is opened by double-clicking on *MainFrm.h*, it can be seen that the new function was also added at the end of the public section of the class.

Since the new function in the *CMainFrame* class probably calls another function to broadcast the message for the new task, this new broadcast function must also be declared and defined in the *CRobotCommanderCore* class. This is done by modifying *RobotCommanderCore.h*. The broadcast function has to be defined by opening *RobotCommanderCore.cpp* and adding the required elements.

After successfully modifying all involved files, the whole project can be compiled by

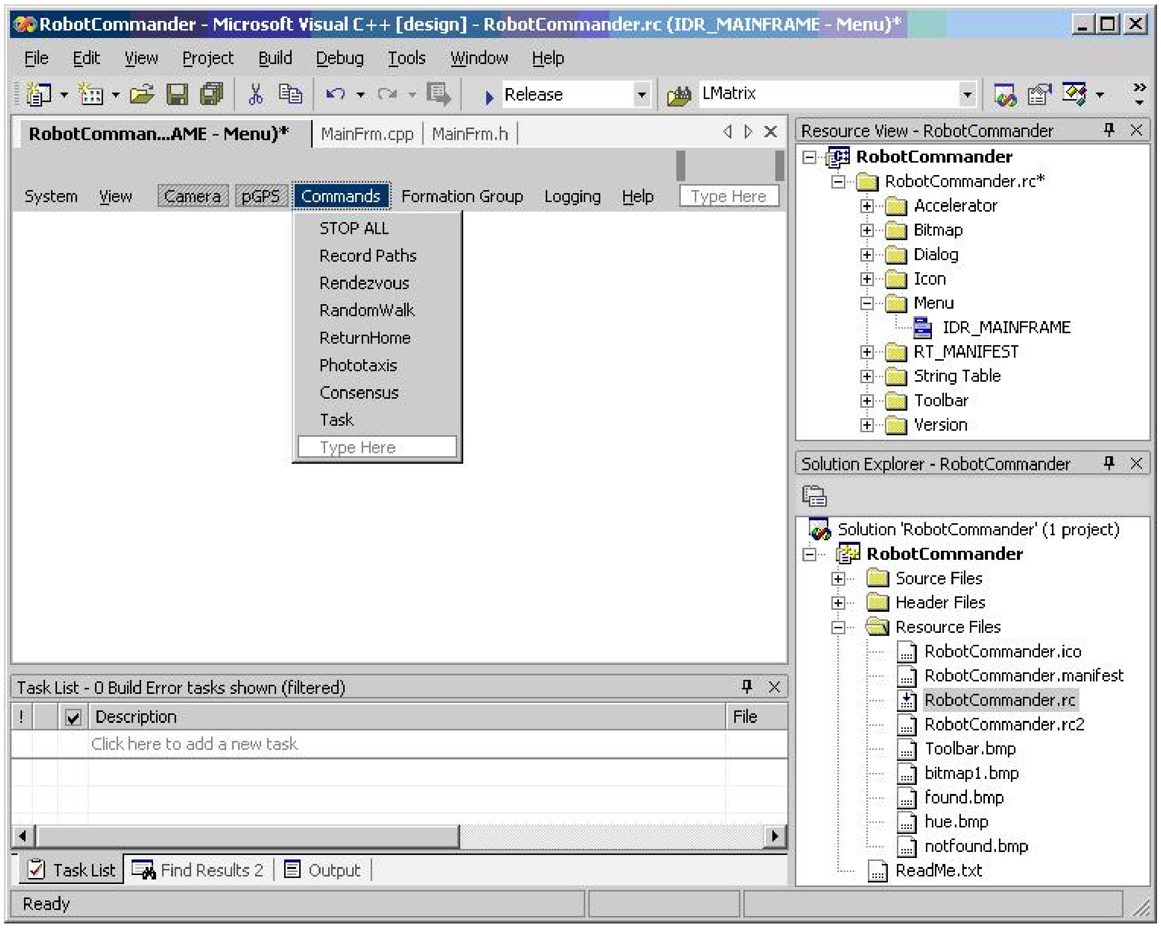
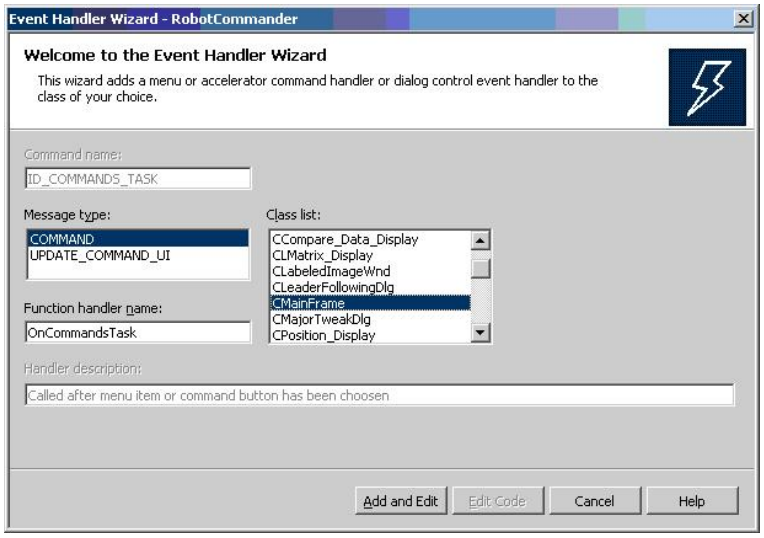


Fig. B.10: Adding a new menu icon in RobotCommander.rc of the RobotCommander project.


Event Handler Wizard - RobotCommander

Welcome to the Event Handler Wizard

This wizard adds a menu or accelerator command handler or dialog control event handler to the class of your choice.

Command name: ID_COMMANDS_TASK

Message type: COMMAND (selected); UPDATE_COMMAND_UI

Class list: CCompare_Data_Display, CLMatrix_Display, CLabeledImageWnd, CLeaderFollowingDlg, CMainFrame (selected), CMajorTweakDlg, CPosition_Display

Function handler name: OnCommandsTask

Handler description: Called after menu item or command button has been choosen

Fig. B.11: Adding an event for the newly created task in the Event Handler.

clicking **Build RobotCommander** in the **Build** menu. If no errors are detected during compilation, the file *RobotCommander.exe* is created in the directory `C:\MASNET\bin`. Now the new RobotCommander GUI can be opened by double-clicking the .exe file and the proper implementation of the behavior can be tested by choosing the new command in the command menu.

Appendix C

RobotCommander File Reference

With over 60 files that make up RobotCommander, adding code and functionality may seem an impossible task. This section provides a description of each file to help programmers tackle RobotCommander. Items without a file extension indicate a .h and .cpp file with the item name.

C.1 General Files

Table C.1: RobotCommander general file and class reference.

File Name	Class	Description
ChildView	*CChildView*	controls the pGPS image on the GUI
Comm_Display	*CComm_Display*	displays all platform communication in a dialog box
Compare_Data_Display	*CCompare_Data_Display*	compares pGPS and encoder position data for robots in a dialog box
LabeledImageWnd	*CLabeledImageWnd*	this class is obsolete and is not used in RobotCommander
LeaderFollowingDlg	*CLeaderFollowingDlg*	displays a dialog of Leader-Follower parameters after the **Follow Leader** command has been selected
LMatrix_Display	*CLMatrix_Display*	displays a dialog of a communication topology matrix for consensus experiments. This class is not yet fully functional
MainFrm	*CMainFrm*	handles all commands in RobotCommander
MajorTweakDlg	*CMajorTweakDlg*	controls the **Major Tweak** window for adjusting the camera filter
Position_Display	*CPosition_Display*	saves position data in the class
RobotCommander	*CRobotCommanderApp* *CAboutDlg*	creates the the main RobotCommander program and the executable file
RobotMonitor	*CRobotMonitor*	controls the **Control Panel** window
stdafx	*N/A*	includes standard system include files
VoronoiTessellation	*CVoronoiTessellation*	calculates centroidal Voronoi tessellations and light position estimation

C.2 Core Files

Below is a list of Core files and their descriptions. It is important to note that changes to any Core files will change the Core files to all instances of RobotCommander. Once any Core file is changed, older instances of RobotCommander will not work.

Table C.2: RobotCommander Core file and class reference.

File Name	Class	Description
AM.h	*N/A*	defines the TinyOS message structure (do NOT change)
AMTransceiver	*CMessageQueue* *CAMTransceiver*	reads and handels messages from the gateway mote
ar.h	*N/A*	includes the ARToolKit (do NOTchange)
CameraController	*CCameraController*	controls the pGPS camera
GroupManager	*CGroupManager*	calculates and performs formation commands under the **Formation Group** menu
lucamapi.h	*N/A*	includes inherant pGPS camera parameters (do NOT change)
masnet_def.h	*N/A*	defines constants for the MASnet platform
masnet_Messages.h	*N/A*	defines all incomming and outgoing wireless messages
PictureAnalyzer	*CPictureAnalyzer*	converts from image to real world coordinates and vice-versa
RobotCommanderCore	*CRobotCommanderCore*	handles all outgoing messages
RobotCommandList	*CRobotCommandList*	handles the list of commands for robots
RobotMask	*CRobotMask*	counts number of selected robots

C.3 PropertyPages Files

Below is a list of PropertyPages files and their descriptions. Note, here, that changes to any PropertyPages files will change the PropertyPages files to all instances of Robot-Commander and the computer registry.

Table C.3: RobotCommander PropertyPages file and class reference.

File Name	Class	Description
CalibrationPropertyPage	*CCalibrationPropertyPage*	saves and loads calibrates camera distortion properties in the registry
CameraPropertyPage	*CCameraPropertyPage*	saves and loads camera properties in the registry
DetectionPropertyPage	*CDetectionPropertyPage*	saves and loads robot detection properties in the registry
FormationPropertyPage	*CFormationPropertyPage*	saves and loads old robot formation command properties in the registry
PlatformPropertyPage	*CFormationPropertyPage*	saves and loads physical platform properties in the registry
TestMarkersPropertyPage	*CTestMarkersPropertyPage*	saves and loads the camera distortion evaluation error in the registry