AN EXPERIMENTAL TESTBED FOR SWARMING AND COOPERATIVE ROBOTIC NETWORKS

By

DANIEL ENRIQUE CRUZ

Bachelor of Science

Pontificia Universidad Javeriana

Bogota, Colombia, South America

2001

Submitted to the Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
July, 2006

AN EXPERIMENTAL TESTBED
FOR SWARMING AND COOPERATIVE
ROBOTIC NETWORKS

Thesis Approved:

Dr. Rafael Fierro - Thesis Advisor

Dr. Gary Yen

Dr. Sohum Sohoni

Dr. A. Gordon Emslie - Dean of the Graduate College

Acknowledgments

My special thanks to Chris Flesher, Brent Perteet and James McClintock for their work on the platform hardware and software. I would also like to thank Warren Lewis for his kind assistance manufacturing sensitive mechanical parts for the robots.

My most sincere gratitude to the financial sponsors of my studies, the MARHES laboratory and the U.S. Army Research Office, under grant DAAD19-03-1-0142 (through the University of Oklahoma).

I would also like to thank Dr. Rafael Fierro for his support, confidence, enthusiasm and continuous advice during my stay at OSU.

Table of Contents

1 Introduction **1**

1.1 Motivation 3

1.2 Contributions 4

1.3 Thesis Outline 5

2 Literature Review **6**

2.1 Introduction 6

2.2 Other Platforms 6

2.2.1 Caltech Multi-Vehicle Wireless Testbed. 6

2.2.2 University of Illinois HoTDeC 7

2.2.3 Cornell University's RoboFlag 8

2.2.4 MIT Multi-Vehicle Testbed 9

2.2.5 Brigham Young University Multi-Vehicle Testbed for Cooperative Robotics 10

2.2.6 University of Pennsylvania Autonomous Robots Testbed . . . 11

2.2.7 ICSL Cooperative Driverless Vehicles Project 11

3 Platform Description **13**

3.1 Introduction 13

3.2 Vehicle Description 13

3.2.1 Initial Platform Assembly 15

3.2.2 On-board Computing 19

3.2.3 Control Board - CANBot 19

3.2.4 Sensor Suite . . . 23

4 The Player/Stage Project **27**

4.1 Introduction . . . 27

4.2 Player, A Sensor Server. . . . 28

4.2.1 Drivers, Interfaces and Devices . . . 29

4.2.2 Configuration Files . . . 30

4.2.3 Player Utilities . . . 33

4.2.4 Client Side Operation . . . 34

4.3 Gazebo, 3D World Simulator . . . 36

4.3.1 World Files . . . 37

4.3.2 Working With Player . . . 39

5 Player - TXT CAN Driver **41**

5.1 Introduction . . . 41

5.2 Development Tools . . . 42

5.3 Hardware Support . . . 43

5.4 Third Party Library Support . . . 43

5.5 Driver Description . . . 44

5.5.1 Supported Devices . . . 45

5.5.2 Operation . . . 45

5.6 Plug and Play Operation . . . 49

5.7 Scalability . . . 49

5.8 Navigation Subsystem . . . 50

5.8.1 Odometry-derived Position Estimation . . . 51

5.8.2 Gyro-Enhanced Odometry . . . 52

5.8.3 GPS Corrected Odometry . . . 53

6 Swarming Behaviors **56**

6.1 Introduction . . . 56

6.2 Behavioral Building Blocks . . . 57

6.2.1 Vision Software Library - VSL 57
6.2.2 Potential Field Controller 60
6.2.3 Obstacle Avoidance 61
6.2.4 Goal-Seeking 63
6.3 Swarming Behaviors 65
6.3.1 Vision-based Formation Control 65
6.3.2 Communication-based Formation Control 65
6.3.3 Boundary Detection and Tracking 67
6.3.4 Flocking 68

7 Graphical User Interface 70

7.1 Introduction 70
7.2 The Qt Framework 71
7.2.1 Qt Desginer 71
7.2.2 Qt Assistant 71
7.3 MARHES Multi-Vehicle Graphical User Interface 72
7.3.1 GUI Features 73
7.3.2 Interface Description 75
7.3.3 Robot Display Description 79
7.3.4 Centralized and Decentralized Operation 80

8 Experimental Results 83

8.1 Introduction 83
8.2 Positioning System 83
8.2.1 Odometry-derived Position Estimation 84
8.2.2 Gyro-Enhanced Odometry 85
8.2.3 GPS Corrected Odometry 87
8.3 Target Assessment Implementation 89
8.3.1 Secure Convoy Path 90
8.3.2 Drive Convoy to Target 90

8.3.3 Spread Convoy Around Target 92

9 Conclusions and Future Work 94

List of Figures

1.1 Multi-vehicle cooperative control & networking testbed. 2
1.2 Natural cooperative behaviors. 3
1.3 Applications of cooperative behaviors. 4

2.1 Caltech multi-vehicle wireless testbed. 7
2.2 HotDec testbed, simulation and control environment. 8
2.3 Cornell's Roboflag testbed. 9
2.4 MIT multi-vehicle testbed. 10
2.5 Brigham Young University multi-vehicle testbed. 10
2.6 University of Pennsylvania's Multiple Autonomous Robots (MARS) testbed. 11
2.7 The Intelligent Control Systems Laboratory (ICSL) testbed. 12

3.1 In-vehicle CAN network. 14
3.2 E7MS Encoder placement in TXT wheel. 17
3.3 Robot assembly. 18
3.4 Suspension modification. 18
3.5 CANBot control unit architecture. 20
3.6 Vehicle reference frame. 21
3.7 Linear and angular speed profile tracking. 23

4.1 Playerv snapshot showing frontal IR range information. 34
4.2 Robots simulated in Gazebo. 37

5.1 Player client-server operation. 42

5.2 TXT Player driver block diagram. 44
5.3 Player driver operation [1]. 45
5.4 GPS corrected odometry operation. 55

6.1 Vehicle carrying a NameTag array. 58
6.2 Example image taken in Gazebo and after Canny edge detection . . . 59
6.3 NameTags identify each vehicle. 59
6.4 Pose estimate of a moving target. 60
6.5 Index of IR sensor range data. 62
6.6 Goal-seek experiment setup and trajectory. 64
6.7 Goal-seek experiment speed and states. 65
6.8 Video-based leader-following. 66
6.9 Communication-based formation control. 66
6.10 Boundary detection and tracking. 67
6.11 Boundary detection and tracking plots. 67
6.12 Flocking with five vehicles. 69

7.1 Qt Designer. 72
7.2 Qt Assistant. 73
7.3 MARHES multi-vehicle GUI description. 76
7.4 MARHES GUI map display box. 80
7.5 MARHES GUI modes of operation. 82

8.1 Odometry. Clockwise and counterclockwise turns. 85
8.2 Gyro-enhanced odometry. Clockwise and counterclockwise turns. . . . 86
8.3 Heading of the robot during different experiments. 86
8.4 Positioning return errors for Odometry and Gyro-enhanced odometry. 87
8.5 Square path experiment performed with a GPS equipped robot. . . . 88
8.6 Target assessment application site. 89
8.7 Scout one in operation. 91
8.8 Position, velocities, and state plot of scout one. 91

8.9 Convoy of robots traveling to the target. 92

8.10 Target assessment position plots and video feedback. 92

Chapter 1

Introduction

This thesis presents the development of a multi-vehicle testbed for cooperative robotics. The objective of this platform is to show the viability of newly researched cooperative control algorithms and techniques by offering a way of experimental validation.

The work introduces the MARHES multi-vehicle testbed, that consists of a team of ten wirelessly networked *TXT*-1 trucks (from Tamiya Inc.) suited with a control board, odometry, a variety of interchangeable sensors and an on-board embedded computer. The system is optimally sized to function well in indoor environments, and robust enough to operate outdoors as well. Each robot features a linear and angular speed controller. High-level control is performed by the on-board embedded computer that hosts the sensor server Player [1]. Player/Stage/Gazebo is a software suite that interfaces with the on-board sensors and provides a standard interface for the programmer as well as remote network access to sensor and control data, the basis for inter-vehicle communication. Player, in conjunction with the on-board wireless network interface, allows each robot and thus the team, to be accessed or controlled from the Internet a LAN or a wireless LAN.

As we will see later in this document, since each robot is, in fact, a network of sensors itself, the multi-vehicle platform becomes a collection of small dynamic networks. This is a unique feature that enables the multi-vehicle platform to serve as testbed for studying decentralized, dynamic, real-time sensor networks as well as networked control and cooperative control algorithms. Additionally, the platform offers a

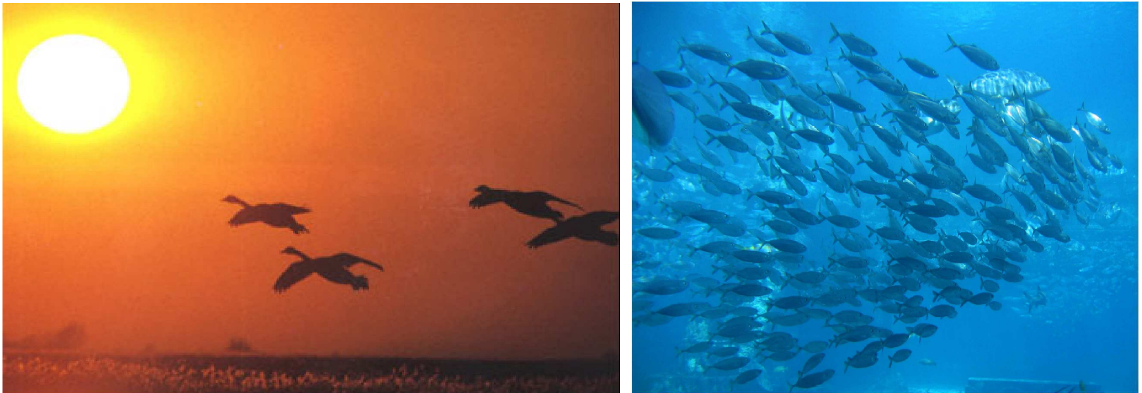
Figure 1.1: Multi-vehicle cooperative control & networking testbed.

unique educational opportunity by training students in several key technologies, such as embedded computing systems and CAN. These technologies are widely accepted as an industry standard. Students also learn classic, hybrid and cooperative control theory and human-machine interaction among others. In the making of each robot students learn microcontroller level as well as computer level programming. They also learn computer vision and become familiar with 3-D rapid prototyping tools such as SolidWorks and CNC machining.

Player is compatible with Gazebo [1], a tool of the Player/Stage/Gazebo suite capable of simulating multiple robots in indoor or outdoor environments. Gazebo is capable of simulating output data for a wide set of sensors and objects in a three dimensional world, with accurate simulation of rigid body physics. This enables the programmer to develop and test algorithms, and have a realistic feel before implementing on the actual multi-vehicle testbed trucks. Gazebo can simulate one or many robots, and uses Player to control each one of them. This allows the software developed for simulation to run with virtually no changes on the real robot, a feature that considerably reduces development time.

1.1 Motivation

In past years, the widely researched robotics topic has expanded from single robotic control to systems comprised of a collection of robotic entities that feature cooperative behaviors. The motivation for such a direction stems from the recognition that several tasks can be performed more efficiently and robustly by a group of individuals rather than only one. Additionally, the increased computing power and resource availability of a group of robots allows users to extend the range of applications to tasks such as multi-point surveillance, distributed localization and mapping, cooperative transportation [2]. Such tasks are impossible to accomplish by a single individual machine.



(a) Birds fly in formation

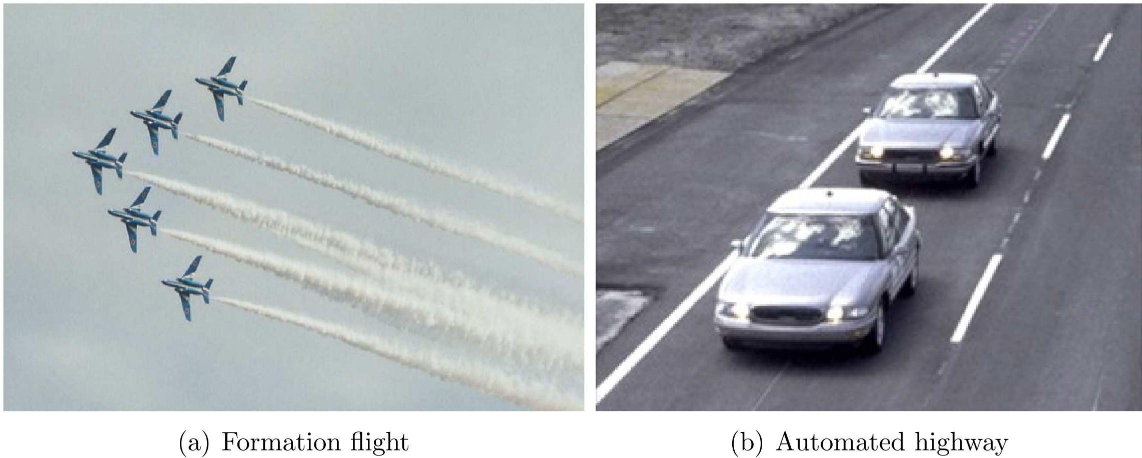
(b) School of fish

Figure 1.2: Natural cooperative behaviors.

However, achieving interaction and cooperation of many individuals may be quite a challenge. Team members must communicate and exchange sensorial information, be able to resolve conflicts, and converge towards a common goal. Additionally, the team is deployed to unknown environments, and robots must account for changing environmental conditions for the duration of the mission.

Transportation plays a vital role in our lives since it has become a primary human need. Among the many other applications, it is a field that would dramatically benefit from robotics and cooperative techniques. Current research on Intelligent Transportation Systems studies real-time control software for improved cruise control, computer vision algorithms for the monitoring of vehicles, individuals or crowds and coopera-

tive autonomous driving. Future transportation systems with these technologies will be safer, quicker, less expensive and ultimately more efficient.



(a) Formation flight

(b) Automated highway

Figure 1.3: Applications of cooperative behaviors.

The work presented here is intended to close the gap between theoretical developments in areas such as cooperative control and navigation, multi-vehicle formation control or vision based cooperative techniques, and experimental results. The testbed offers flexible and robust hardware, a standard programming environment, and a simulation tool. A simulated environment enables researchers to implement cooperative control and multi-vehicle robotics techniques in a fast paced environment.

1.2 Contributions

This work presents several contributions and advantages in respect to existing platforms for cooperative robotics. Each of the vehicles is built with flexibility in mind. Depending on the application, robots may be suited with several sensors. New sensors can be easily added to the sensor pool available for each of the robots. Each robot is capable of adapting to different applications *on-the-fly*.

The platform itself is scalable, so the number of team members can be increased without modifications to the existing system. The availability of a tightly coupled simulated environment helps researchers and developers preview their algorithms while minimizing testing and development time.

Additionally, the multi-vehicle testbed serves an educational purpose. It is not only a tool for the study of robotics, communications, cooperative control, embedded systems and sensor networks. In the preparation of each robot students also learn Linux OS, microcontroller programming, communication protocols and automated machining techniques.

1.3 Thesis Outline

This document is organized as follows: A literature review of the most relevant work is presented in Chapter 2, containing other multi-vehicle platforms developed in laboratories around the United States. The platform hardware specification is discussed in Chapter 3. Each of the vehicle modifications is described in detail, as well as the onboard sensing and low level control systems. Chapter 4 introduces Player, the open source robot sensor server and Gazebo, the robot simulator. Chapter 5 explains the development of the Player driver for the TXT robots. Chapter 6 presents the swarming behaviors and all the high level software developments part of the system, as well as some of the applications already implemented with the platform. A human interface to interact with the multi-vehicle testbed is presented in Chapter 7. Chapter 8 provides results of comprehensive experiments performed with the platform. Finally Chapter 9 gives concluding remarks of the thesis and ideas for future improvements of the testbed.

Chapter 2

Literature Review

2.1 Introduction

At present time there are several other multi-vehicle platforms, developed at universities and research labs. The objective of most groups, is to validate new theories in cooperative control and multi-vehicle control techniques.

The review presented here is not meant to be exhaustive, but introduces some of the most well known multi-vehicle platforms, developed in different labs in the United States.

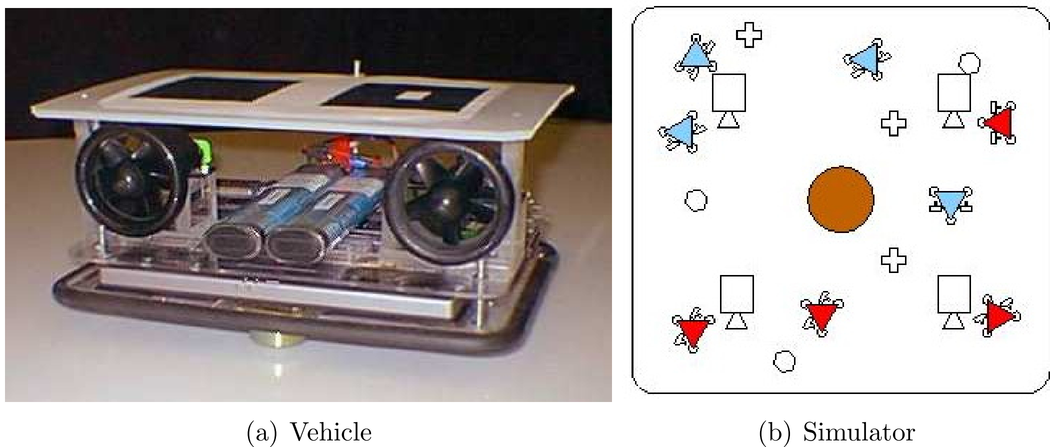
2.2 Other Platforms

2.2.1 Caltech Multi-Vehicle Wireless Testbed.

The Caltech multi-vehicle wireless testbed MVWT [3] shown in Figure 2.1 introduces small hovercraft vehicles, with embedded computers and communication capabilities. The most interesting characteristic of the MWVT is the vehicles themselves. Each of the vehicles consists of a PDA mounted over two high performance ducted fans, that allow the system to hoover a few millimeters off the floor. The vehicle is described by second order dynamics. Computing power is provided by a Sharp Zaurus PDA with a 206 MHz Strong ARM processor, the Zaurus interfaces with the fans in the

hovercraft via an Atmel microcontroller. Each hovercraft has a 2-axis accelerometer and a gyro that stabilize the vehicle in motion. Each vehicle of the testbed runs the QNX real time operating system, and uses the RHexLib C++ programming suite to control the robot. Inter-vehicle communication is accomplished via TCP/IP and 802.11b, wireless networking. Bluetooth networking is also available as an alternative communication method. Position of each of the vehicles is fedback to the system using an array of overhead cameras, that limits the system to work in indoor environments.

The platform is intended for validation of theoretical analysis in multi-vehicle coordination, cooperative control, networked control systems and high confidence distributed computation.



(a) Vehicle

(b) Simulator

Figure 2.1: Caltech multi-vehicle wireless testbed.

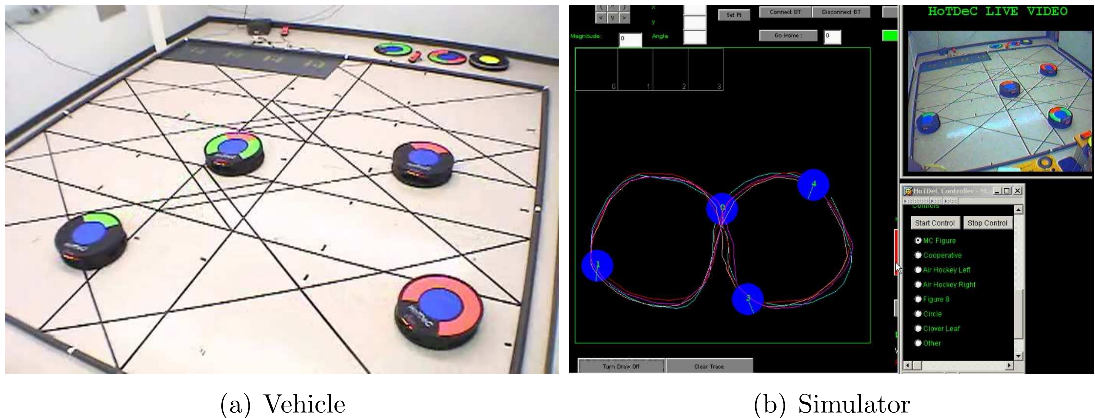
2.2.2 University of Illinois HoTDeC

HoTDeC [4] shown in Figure 2.2 developed at the University of Illinois, stands for Hovercratf Testbed for Decentralized Control. This system is similar to Caltech's MVWT, uses several hovercraft vehicles in an indoor environment, where the position estimation of each vehicle relies on an overhead multi-camera system. Each vehicle carries an in-house X86 Crusoe compatible processor system and a MCU with several PWM, digital and analog inputs and outputs, USB, PCI and Ethernet connectivity. The on-board system controls the local operation of the hoovercraft.

The overhead camera system processes the image from the robots playground,

and broadcasts the location of each vehicle using a bluetooth network. Positioning systems similar to this are used in international robotics competitions like *Robocup.*

Researchers at the University of Illinois are currently expanding the overhead positioning system, to be able to have more robots in a simultaneous run. Additionally they plan to introduce obstacles and communication dead zones to simulate loss of communications.



(a) Vehicle (b) Simulator

Figure 2.2: HotDec testbed, simulation and control environment.

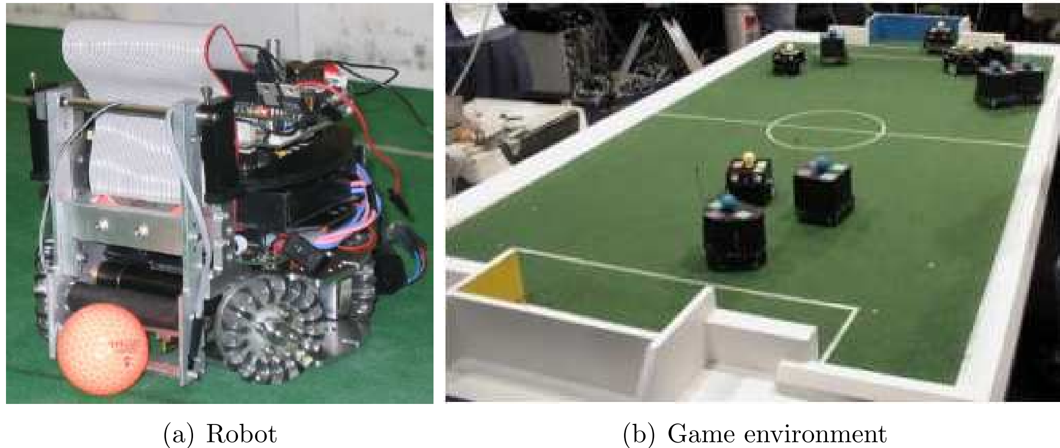
2.2.3 Cornell University's RoboFlag

Cornell University's RoboFlag [5] shown in Figure 2.3, features several small wheeled units with local control loops. The testbed was originally designed to be used in a game called 'RoboFlag', a two team game, where each of the teams attempt to infiltrate the other team's zone and steal its flag. By allowing students to use the platform and play the game, new cooperative behaviors emerge since winning requires a combination of offense and defense strategies.

Each of the vehicles carries a simple on-board controller or MCU, in charge of commanding the actuators in the robot. High level control is done off-board by a remote workstation.

The platform is intended to be used indoors, as position of each vehicle relies on an overhead camera positioning system, and the control of every robot is centralized. An external computer calculates the position of each team member and forwards the information to the high level control computer. Since the task of controlling the

vehicles is done in a single machine, the communication environment is simulated. Additionally, the system also offers a simulation environment where is possible to test algorithms without the real robots, in a computer simulated physics environment.



(a) Robot (b) Game environment

Figure 2.3: Cornell's Roboflag testbed.

2.2.4 MIT Multi-Vehicle Testbed

The Massachusetts Institute of Technology multi-vehicle testbed [6] shown in Figure 2.4 features air and ground vehicles. Eight Pioneer rovers in ground and four blimps and several Unmanned Autonomous Vehicles UAV's conform the fleet. The Pioneer robots are manufactured by MobileRobots, and feature a 4-wheel skid steer traction mechanism and an on-board microcontroller that executes the local control loops. Pioneer robots are extremely robust and easy to use, however their price is elevated. Each of the vehicles is equipped with on-board computing, wireless communications and suited with several sensors like GPS, sonars and video cameras.

The MIT testbed provides a combination of vehicles that enable the system to have unmatched flexibility and mobility. This testbed addresses distributed command and control algorithms, dynamically reconfigurable network topologies, resource allocation using hierarchical networks, vehicle autonomy and multi-vehicle operations with humans-in-the-loop.

The platform offers a simulation environment for testing of the algorithms. By using a hardware-in-the-loop system, most of the hardware is used as part of the

simulation run.

(a) Pioneer Robots

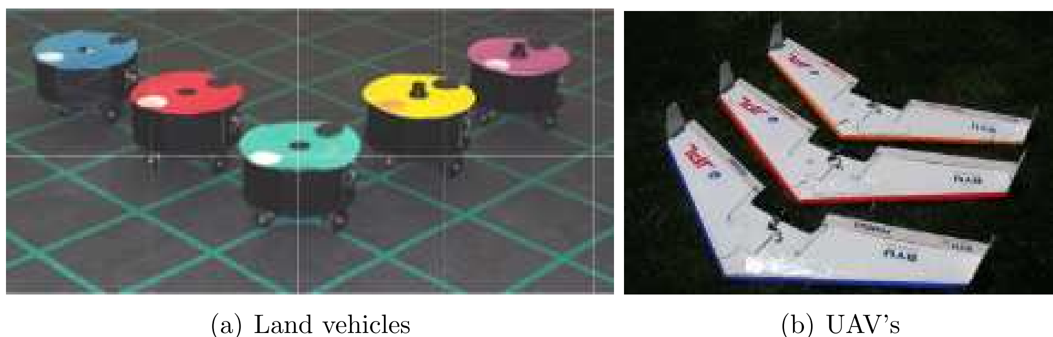
(b) UAV Fleet

Figure 2.4: MIT multi-vehicle testbed.

2.2.5 Brigham Young University Multi-Vehicle Testbed for Cooperative Robotics

Brigham Young University has been using a multi-vehicle testbed [7] shown in Figure 2.5 for cooperative robotics research for a number of years in the MAGICC lab. The initial system consisted of five nonholonomic robots, with an on-board pentium based computer, and wireless communications, the system has received upgrades since. Vehicle position is provided via an overhead vision system. The objective of the platform is to evaluate formation control schemes, and is intended to be just a step before evaluating formation control algorithms directly on UAV's. The platform has proved to be useful in evaluating cooperative control algorithms as well.

Their latest efforts focus on Unmanned Aerial Vehicles, but still small mobile units equipped with an on-board PC104 are used to simulate UAV coordination maneuvers.



(a) Land vehicles

(b) UAV's

Figure 2.5: Brigham Young University multi-vehicle testbed.

2.2.6 University of Pennsylvania Autonomous Robots Testbed

The University of Pennsylvania's Multiple Autonomous Robots (MARS) testbed [8] 2.6, features a team of inexpensive remote control trucks suited with on-board computing, and a suite of sensors. Additionally the team has a blimp as part of the multi-vehicle team. The radio controlled trucks have been extensively modified, suited with a Pentium III laptop to run control and sensing algorithms, a GPS, and a small 3 axis Inertial Measurement Unit IMU to provide means of position estimation.

Communications are achieved via wireless networking. The vehicles are intended to acquire and maintain knowledge of the terrain to maximize mission effectiveness, and integrate air-ground assets in support of continuous operation in varying terrain.

Recent work has focused on improving position estimation of the vehicles, integrating information from the blimp's on-board camera.

Figure 2.6: University of Pennsylvania's Multiple Autonomous Robots (MARS) testbed.

2.2.7 ICSL Cooperative Driverless Vehicles Project

The Intelligent Control Systems Laboratory (ICSL) shown in Figure 2.7 at Griffith University has been working on intelligent vehicle technologies for quite some time. The group has demonstrated cooperative autonomous driving with three computer

assisted experimental vehicles, developed by the French scientific organization INRIA and industry partner ROBOSOFT. Some of the concepts in research on Intelligent Transportation, fall under the category of *driver assistance*, where the system aid human beings in maneuvers such as parallel parking, or vision based autonomous driving. The ICSL research focuses on more advanced concepts to enable vehicles to drive independently, and cooperative autonomous driving, where vehicles that coexist in the road drive in cooperation with each other.

In order to implement and test experimentally the cooperative driving algorithms, the ICSL developed Cooperative Mobile Robot Platforms. Each platform consist of a number of microcontroller based modules for tasks such as decision and control, motor actuation, sensor interfacing, radio communications and power management. Each of the modules communicates with one another using the I^2C Bus. The ICSL platform has a sensory subsystem that includes ultrasonic distance sensors, infrared transmitter receiver pairs, infrared white line sensors, laser diodes and optical sensing arrays.

Techniques researched with the ICSL platform include road-lane following, cooperative overtaking, unsignalized intersection negotiation, distance and tracking control, and static and dynamic obstacle avoidance.

(a) Small scale robots (b) Intelligent vehicles

Figure 2.7: The Intelligent Control Systems Laboratory (ICSL) testbed.

Chapter 3

Platform Description

3.1 Introduction

This section describes the hardware of the cooperative multi-vehicle testbed. Each vehicle features an on-board computer for local control, and can be suited with a variety of hot-swappable sensors depending on the application. The entire system is integrated with Player, an open source sensor server also compatible with Gazebo, a 3D world simulator. Control algorithms can be evaluated in simulation mode in Gazebo, and then ported to the real robot with virtually no code change. The platform presented is an innovative and complete robotic system that serves the study of Robotics, Cooperative Control, Hybrid and Embedded Systems, Sensor Networks, Networked Control and many other techniques with an extensive range of applications.

3.2 Vehicle Description

The testbed consists of ten scaled R/C monster trucks that have undergone extensive modifications to become a highly adaptable team of networked robotic units. This platform is well suited to robotics research, and can be used both indoors and outdoors. It is small enough to be easy to work with in a laboratory environment, yet large enough to carry a substantial load including a laptop computer and multiple

environmental sensors, while negotiating rough terrain outdoors.

Each robot carries a local computer and an array of sensors to interact with the environment. The flexibility of the system allows the user to configure each vehicle individually with different types of sensors as the application may require. The sensors are networked into a small CAN network. Only the vision sensors are connected to the local computer via FireWire due to bandwidth requirements. Figure 3.1 illustrates the network of sensors in each individual robot. Each of the vehicle's front wheels have been modified and suited with miniature encoders from USDigital that register wheel rotation, and make possible to control the angular and linear speeds and to estimate the position (odometry) of the robot.

Once modified, each vehicle is 498 mm long, 375 mm wide, 240 mm tall, weighs 8.2 kg loaded with the on-board computer and sensors or 5.22 kg by itself. The robot can reach speeds of 1.6 m/s forward or in reverse, can be loaded with a maximum payload of 3.7 kg and has a minimum turning radius of 1300 mm. With double steering, the turning radius can be reduced by half. Double steering requires the addition of an extra servo to the robot.

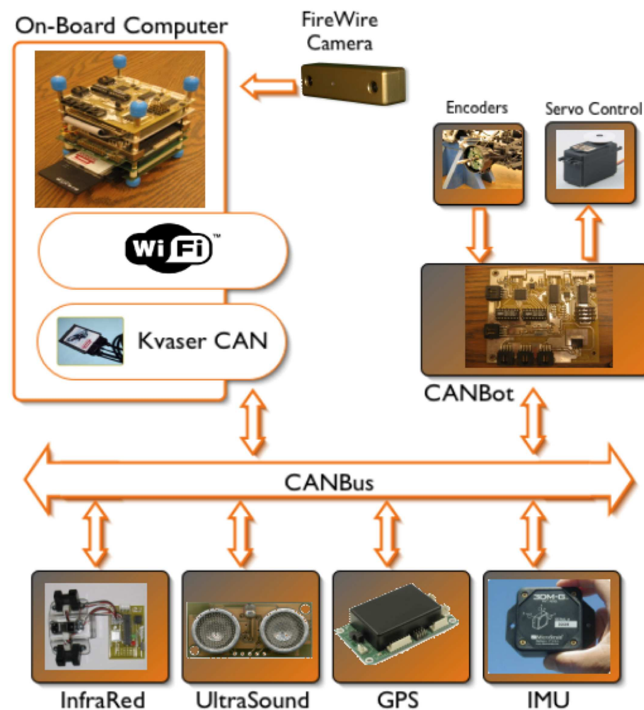


Figure 3.1: In-vehicle CAN network.

CAN was the protocol of choice because of its low latency times 1 ms and high speed rates up to 1 Mbps [9]. CAN is a message-based bus, therefore addresses are not defined, only messages. These messages are recognized by message identifiers. Messages have to be unique within the network and define not only the content, but also its priority. Each device or sensor attached to the robot, is accompanied by a small board that interfaces with the sensor, and broadcasts sensor related information to the CAN bus. Each of these boards use a PIC[1] 18F series micro-controller, with flash memory, wide I/O, ADC capabilities, and also CAN connectivity. The lower-level CAN Bus system allows for sensors and actuators to be added and removed from the system, with no reconfiguration required to the higher-level structure. Initially, message ID's are defined for each type of module. When a new module is added to the bus, the lower-level software configures the device to make information readily available by the higher-level software. This *plug-and-play* scheme allows for ease of transition from one application to the next, makes the mobile platform flexible to higher-level controllers, and allows for larger module networks to be built without the need to reconfigure the entire system. Moreover, it permits a higher level of abstraction, i.e., each sensor is a node within the robot network, and each robot is seen as a node in a network consisting of several robots.

3.2.1 Initial Platform Assembly

In general, the TXT-1 was constructed by simply following the instructions provided by Tamiya, the manufacturer. Ceramic grease was applied very liberally in all areas where it was called for, and the hardest damper oil was used inside the shocks in order to increase the stiffness of the suspension, and prevent it from leaning when heavily loaded. Also, three minor changes to the instructions needed to be made in order to facilitate the modifications. The plastic parts (F1, F5) which are used to attach the monster truck shell to the chassis were not used. Instead, a metal plate for carrying the computers and sensors was attached. The servo driven mechanical speed

[1]Manufactured by Microchip Inc.

controller system was not used. This includes the servo, speed controller, resistor, receiver, switch, and mounting plate. Instead, the motors were rewired for use with an electric speed controller. Three 0.01μF capacitors were soldered onto each of the robot's motors. One capacitor was attached between each terminal on the motor and the motor's casing. The third capacitor is attached between the two terminals.

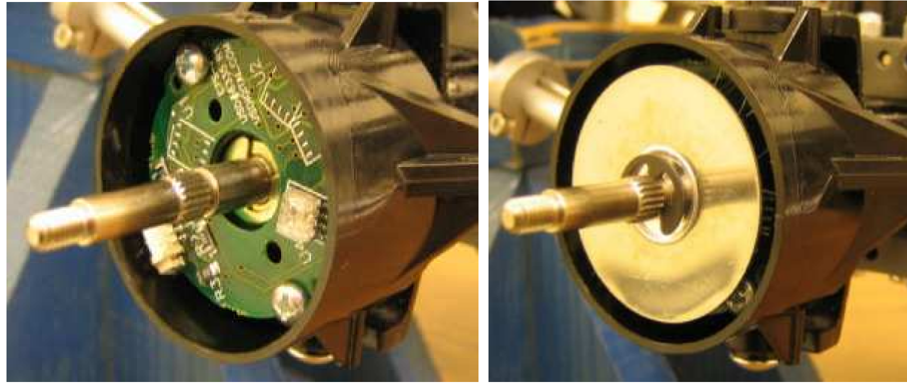
Optical Encoders

One of the most important modifications that has been made to this chassis, is the addition of optical encoders in each robot for odometry. These encoders make use of a small led, a photo detector, and a reflective code wheel that spins with the tire. Each encoder module supplies two digital outputs in quadrature, providing information about how fast the wheel is spinning and its direction of rotation. Both front wheels have been equipped with encoders, together, these encoders allow us to determine both the linear and the angular velocities of the robot.

The encoders are based in the E7MS, a commercially available system from US-Digitial. Installation required removing the wheel wells from the robot and making a few modifications. First, three holes were drilled at the back of the wheel well. Two of them were used for standoffs to secure the PCB board which contains the optical sensor. The third hole was used for a two line telephone cable which carries the output signal from the encoder to the CAN bus and power. Finally, a milling machine was used to reduce the length of the wheel wells by about 0.5 inches. Once this was done, the E7MS PCB was installed as shown in Figure 3.2a and the wheel wells were reattached to the robot. Next, the code wheel was slipped over the wheel axel. It sits flush with the surface of the wheel well as shown in Figure 3.2b.

Aluminum Plate for Sensor Support

Each vehicle has a precision milled Aluminum plate shown in Figure 3.3. This plate was designed to be light-weight but sturdy enough to support a variety of computers and sensors. It is 11 inches wide, 18 inches long, and weights approximately 1.3 lbs.



(a) PCB in wheel hub (b) Encoder wheel

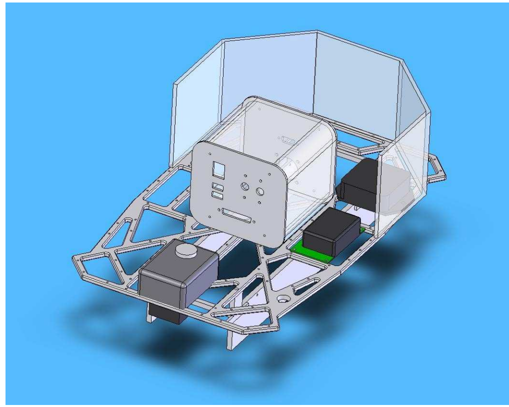
Figure 3.2: E7MS Encoder placement in TXT wheel.

Major features of this plate include a cantilever design structure which allows cabling to be easily run through the plate between components. It contains over one-hundred 4/40 screw holes for securing sensors and four 8/32 holes for attaching the plate to the robot chassis. Finally, there is one large 0.62 inch hole for mounting an emergency power switch.

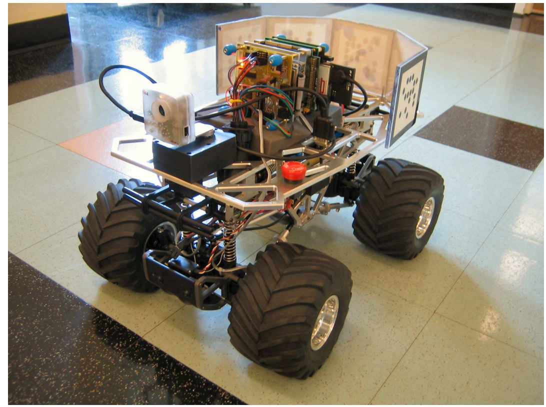
The plate was designed using SolidWorks 2004, and milled using a Haas CNC milling machine. The milling process required to first machine a mount to attach the aluminum block to the milling machine. The CNC machine was able to mill each plate in about 5 hours, with very little human interaction. Once the plates were finished, the last hurdle was attaching them to the robot chassis. This required to make four small aluminum mounting brackets that could attach to the robot in the same place where the plastic mounts for the monster truck shell would normally be attached. These mounting brackets were made of 0.15 inch think aluminum with standard milling machine.

Additional Shocks

The suspension system on the chassis has been improved through the addition of one shock absorber per wheel. This upgrade increases the maximum load that the robot can carry without leaning, and raises the chassis slightly higher off the ground. In order to complete this modification, we first purchased two aluminum damper sets (part #53492) from Tamiya. Installation was fairly easy because the shocks can



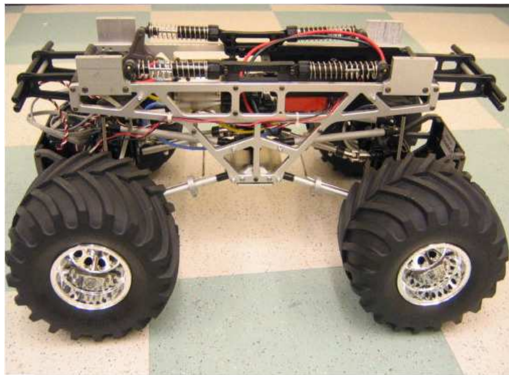
(a) 3D SolidWorks Model



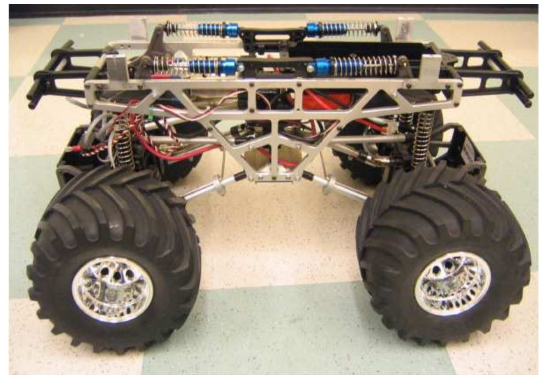
(b) Assembled robot

Figure 3.3: Robot assembly.

simply replace a set of push rods that are normally used on the chassis. After testing the newer aluminum dampers, it was found that they are stiffer than the original shocks which come with the robot. For this reason, we chose to replace the stock damper set with the new set and install the originals in place of the vertical push rods. A picture of the chassis both before and after this upgrade was made, is shown below in Figure 3.4.



(a) Original configuration



(b) Shock addition

Figure 3.4: Suspension modification.

Other Modifications

Additional modifications include the installation of an electric speed controller (Novak Super Rooster), foam inserts for the wheels to increase wheel stiffness, stabilizer bar clamps, and center skid plates that protect the gear box.

3.2.2 On-board Computing

A rugged PC-104 (manufactured by Advanced Digital Logic) miniature computer hosts the local control software on each vehicle. The PC-104 system has the same capabilities of a full blown desktop system, with a reduced size factor ideal for robotic and mobile sensor networks applications. The PC-104 has been equipped with a 40 GB hard drive, FireWire, Kvaser CAN cards, WiFi support and a Pentium M processor. The systems feature RedHat Fedora Core 2 and run the 2.6 Linux kernel. The on-board computer is in charge of running higher-lever control tasks and interfaces with the robot via the CANBot Control Unit. The higher-lever control sends linear and angular speed commands to the CAN bus CANBot board, which executes PID local control loops based on odometry readings. The CANBot Control Unit is the only CAN device that is mandatory for each of the vehicles and will be explained in detail later.

The PC-104 system executes Player, the sensor server. Once Player is running, the robot is controllable via the network. A control program may run on the local machine or in a remote computer on the network. As with any other communication systems, this has implicit systematic and non-systematic delays that affect the performance of the controllers. The lower-level controller has to deal with the CAN-Bus network delays and latency; nevertheless, the influence of these are assumed to be small enough to be disregarded. The higher-level control assumes information is readily available at each sampling interval. However, when the higher-level control operates using the Local Area Network (LAN), delays can be significant and visibly affect the performance of the system. The capability of the platform to operate in local or networked mode gives the opportunity to study the effects of such delays in networked embedded systems.

3.2.3 Control Board - CANBot

The control board CANBot shown in Figure 3.5 has three main features explained below. A section devoted to the PID controller is also included

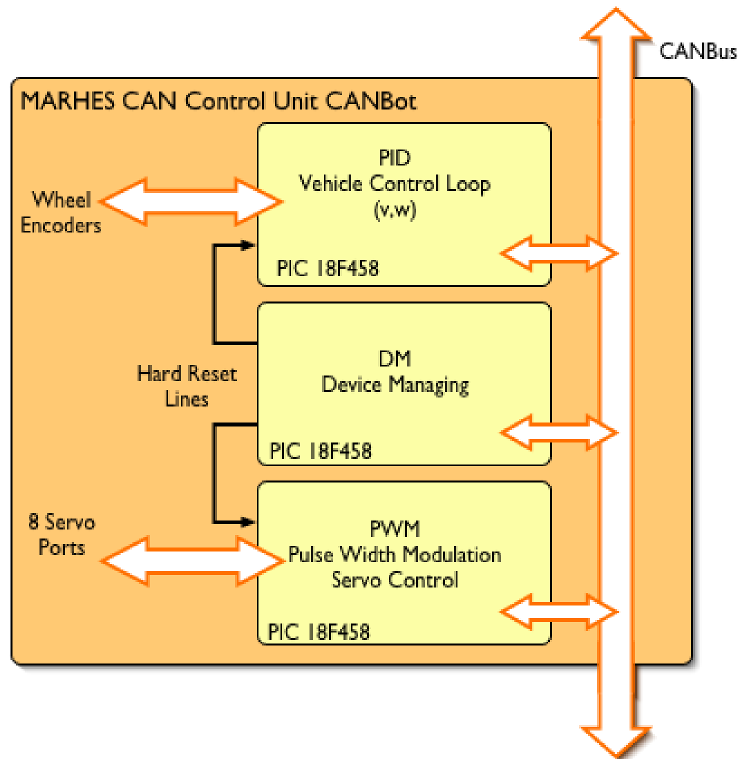


Figure 3.5: CANBot control unit architecture.

Device Managing

The CANBot board performs active bus monitoring to relieve the higher-lever control of the task of looking for newly connected sensors. The Device Manager (DM) broadcasts a data packet with information about which devices are connected and active at any moment in the CANBus. If a device stops transmitting information, the DM attempts a soft or hard-reset to try and bring the device back in operation. In case of no response, the DM will signal the higher-lever of sensor disconnection. Also most of the sensors have the capability of going into a standby mode that allows for minimum power drain.

Vehicle Control

Robot control is one of the most important tasks performed by the CANBot control unit. It is based on PID controllers that runs on a single microcontroller and use wheel encoders to close the loop. Each front wheel in the TXT robot has a 400 CPR quadrature optical encoder based on the USDigital E7MS optical encoder. CANBot

reads the encoder counter at a 50 Hz sampling rate and performs a tight PID control loop to ensure that the linear and angular speed of the vehicle reach the commanded values sent by the higher-level control. The board also makes the encoder counter information available through the CANBus to provide means of position estimation based on odometry.

Pulse Width Modulation Generation

Pulse width modulation (PWM) is an important task since it is responsible for executing all the servo control commands. All the motors in the TXT robots including speed control, steering angle and camera pan angle are controlled via servo commands sent over the CAN bus. The CANBot board can control eight different servos through its PWM generation chip. A typical servo connected to this unit will have a resolution of 0.043° per control bit.

Robot Kinematics and PID Tuning

The success of every higher-lever control algorithm is based upon the ability to control the vehicle. Thus the vehicle linear and angular speed controller are analyzed in this section. Specifically. the vehicle is considered to move in a plane world reference system given by (X_F, Y_F), where the vehicle's location coordinates are completely described by (x, y) within the world frame as shown in Figure 3.6. The (x, y) location of the vehicle is given relative to a point, chosen to be the center of mass of the robot.

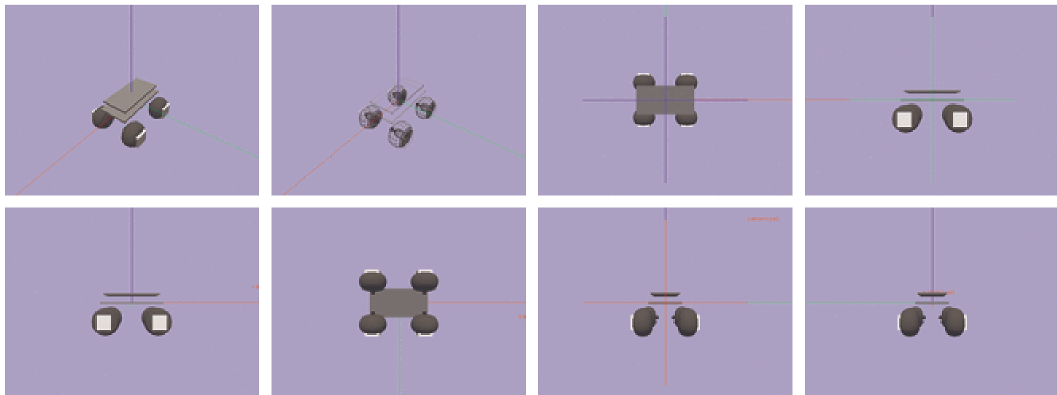


Figure 3.6: Vehicle reference frame.

The heading or orientation of the vehicle relative to the world frame, is given by θ and considers the relative orientation of a fixed frame in the vehicle (X_R, Y_R) with respect to the world frame (X_F, Y_F). The pose of the platform is then completely specified by a vector q_i where

$$q_i = \begin{bmatrix} x_i & y_i & \theta_i \end{bmatrix}^T \tag{3.1}$$

For modeling purposes, the vehicle can be represented as a unicycle type vehicle.

$$\begin{aligned} \dot{x} &= \upsilon \cos\theta \\ \dot{y} &= \upsilon \sin\theta \\ \dot{\theta} &= \omega \end{aligned} \tag{3.2}$$

The inputs to the system are the linear speed υ and the angular speed ω.

Even though the vehicle will not behave as the perfect kinematic model described in (3.2) as the linear and angular speeds will not be instantly achieved, the addition of a lower-lever controller will make the vehicle kinematically controllable. The lower-lever controller will guarantee that after a short period of time, a commanded linear υ and angular speed ω will be reached.

The vehicle's linear and angular speeds are calculated by the CANBot Control Unit using the wheel encoders and the following equations:

$$\upsilon = \frac{(d_R + d_L)\zeta}{2\Delta T} \tag{3.3}$$

$$\omega = \frac{(d_R - d_L)\zeta}{d_W \Delta T} \tag{3.4}$$

Where $\zeta = 0.3$ mm/tick is the distance traveled by a wheel in one encoder tick, d_W is the wheel separation or robot width. And $\Delta T = 20$ ms is the sampling time.

This values are hard coded in the CANBot unit firmware. d_R and d_L are the traversed distance measured by the right and left encoders in ticks, in one sampling interval.

The lower-lever PID controller parameters were adjusted to reach the commanded speed in the minimum time, with the smallest possible overshoot. Once the performance of the controller was satisfactory, they were set in the firmware according to the following values for the linear speed controller:

$$K_i = 0.02, \quad K_d = 1, \quad K_p = 0.8 \tag{3.5}$$

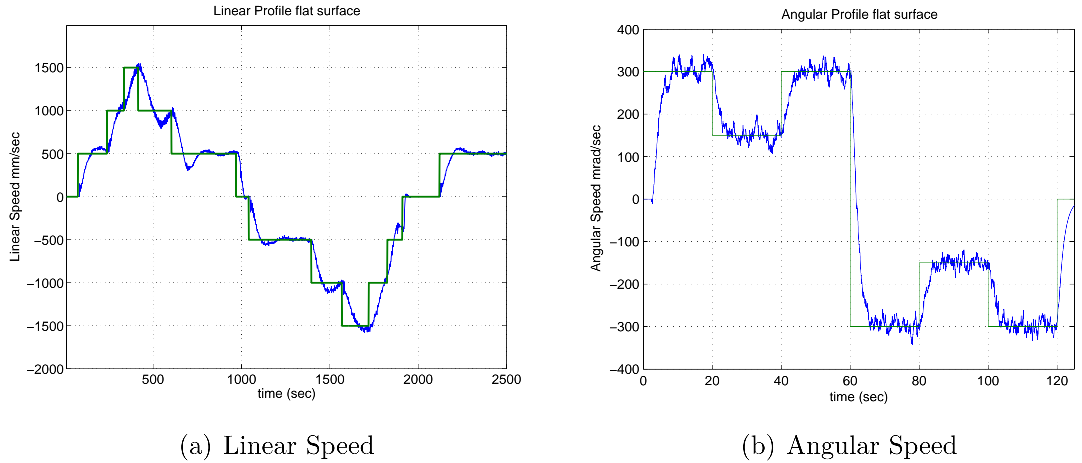
And for the angular speed controller, we have:

$$K_i = 0.1, \quad K_d = 0.01, \quad K_p = 0.4 \tag{3.6}$$

The controller output is given by:

$$u(n) = K_p e(n) + K_i \Delta T \sum_{k=0}^{n} e(k) + \frac{K_d}{\Delta T}(e(n) - e(n-1)) \tag{3.7}$$

Figure 3.7 shows the controller performance on even terrain.



(a) Linear Speed

(b) Angular Speed

Figure 3.7: Linear and angular speed profile tracking.

3.2.4 Sensor Suite

This section describes the suite of sensors that is available for each of the vehicles. Each sensor features CAN connectivity based on the PIC18F258/248 microcontrollers.

InfraRed

The InfraRed (IR) devices are based on the Sharp 2Y0A02 InfraRed long distance measuring sensor. This device outputs an analog voltage proportional to the object distance. The range of the Sharp long distance measuring sensor is $0.2 - 1.5$ m. The testbed IR device is composed of three Sharp IR long distance sensors in a 45 ° offset array. This allows the system to identify obstacles straight in front of the vehicle, in diagonal and on the side. A single microcontroller chip reads the distance sensors at a 25 Hz sampling rate using the built-in Analog to Digital Converters (ADC) and once the range information is read, it is transmitted through the CANBus via the built in CAN controller.

Global Positioning System - GPS

A variety of GPS receivers may be connected to the vehicle. Currently the DS-GPM GPS receiver from Designer Systems and the Garmin GPS 18 5 Hz are supported, additionally the interface board is capable of supporting any GPS receiver that outputs standard NMEA 0183 sentences. These devices may be accessed via the slave I^2C interface or RS-232 and provide data which includes longitude, latitude, altitude, heading, speed, satellites detected and dilution of precision. Time, date and analog inputs, are available but not used at this time. Both GPS receivers have an adjustable update rate of up to 1Hz and are capable of differential correction with the support of extra hardware. As with other GPS receivers in this class, these receivers are accurate to approximately 15 m without differential correction. Data from the GPS receivers, regardless of the particular interface or protocol used, is parsed and packaged into two separate CAN bus messages that are forwarded to bus.

Inertial Measurement Unit - IMU

At present time, the IMU used by the testbed is the Microstrain 3DM-G. The IMU provides temperature compensated triaxial orientation information in pitch (ϕ), roll (ψ) and yaw (θ) by integrating data from various internal sensors and measured

against a local coordinate system. Specifically, the IMU incorporates three accelerometers for measuring gravity, three magnetometers for measuring magnetic fields and three rate gyroscopes for measuring the IMU's rate of rotation about each of the three axes. An addition of the IMU to the vehicle enables the system to provide 2D Gyro-Enhanced odometry, where the orientation angle is provided by the IMU. Gyro-Enhanced odometry is far more accurate than solely encoder based odometry, particularly in determining the heading of the robot, since this angle is provided by a gyro instead of differences in wheel rotation. While 2D Gyro-Enhanced odometry is automatically provided when the IMU is connected to the system, the user may alternatively request 3D position that makes comprehensive use of the sensor. The MCU interface board, which samples data from the IMU via the RS-232 data port, has the same design and layout than the GPS interface board. The only difference between the two is the firmware loaded into the microcontrollers flash. The PCB was specifically designed to be populated with only those components required by the particular application, in order to reduce the number of different PCBs on the vehicle. The IMU is connected to the interface by one of the RS-232 ports and is set to communicate at 38400 bps, in order to accommodate data sampling at 50 Hz (20 ms/sample). A high baud rate is required due to the size of the messages returned by the IMU. Currently, only the vectors and compensated angular rates for each axis are reported to the main computer via the CAN bus, although the IMU is capable of providing much more data, including both instantaneous and gyro-stabilized data for the quantities of Euler angles, quaternions, orientation matrices, and angular acceleration. All of this data is temperature compensated over a wide range of temperatures (approximately $-30°C$ to $+60°C$).

Vision

Vision gives a great source of information about the environment. Computer vision is not only widely used in robotics, but also in several other fields like medical imaging, military intelligence, and industrial inspection. Most of the interaction between hu-

man beings and the world is based on visual impressions. Vision is one more sensor available for the TXT robots. The video cameras are the only sensors in the sensor suite that are out of the CAN network. We use FireWire cameras connected directly to the on-board computer. Currently the Vision Software Library (VSL) works with a BumbleBee stereo camera manufactured by Point Grey or a Fire-i digital camera from Unibrain. The VSL will be explained in detail in the software section.

Chapter 4

The Player/Stage Project

4.1 Introduction

The Player/Stage project [1] started on the University of Southern California and was originally developed by Brian Gerkey and Kasper Støy. The project has been open source from its beginnings and is protected by the GNU General Public License, at present time most of the robotics laboratories around the world use it and continue developing new code for it.

This chapter discusses Player's main features, and can be used as a quick guide to work with it. The chapter is mostly based on Player's documentation available on the web, and in the MARHES Lab server. Readers are encouraged to read Player's own documentation and use this chapter as an introduction and reference only.

The Player/Stage project comprises three main components that can be used individually or all together. Player, a sensor server, Stage a 2D robot simulator, and Gazebo, a 3D world simulator. The MARHES multi-vehicle platform uses Player in each of the robots, and Gazebo to simulate behaviors and test algorithms before using the TXT trucks.

Player receives its name in reference to William Shakespeare from *As You Like It* Act II, Scene 7: *All the world's a stage, And all the men and women merely players: They have their exits and their entrances; And one man in his time plays many parts*

Player constitutes the central piece of software used in the multi-vehicle platform.

The architecture of the system revolves around player, the sensor server. Additionally the 3D simulator Gazebo, has been widely used to test all of the behavior implementations.

4.2 Player, A Sensor Server.

The Player/Stage project is an ongoing open source software development effort, that offers a suite of tools for research in robotics. The project was initially started, and is now maintained by researchers in the University of Southern California Robotics Research lab. The source code is available for free download in *sourceforge.net* and protected by the GNU General Public License. Player/Stage is one of the most popular and widely used robotic interfaces in the world. Since the source code is available for download and open to modifications, researchers from all over the world keep adding support for new sensors, robots and interfaces. Player/Stage runs on several platforms like Solaris, Linux and Mac OSX.

Player provides a network interface to each sensor and actuator in the robot. Player, is a robot server that enables users to login as clients to the robot's infrastructure. It allows several clients to access the same device, which make it optimally suited for multi-vehicle control and coordination applications. Player supports a variety of hardware including laser range finders, video cameras, GPS and most of MobileRobots (previously ActivMedia) robots, a popular robot manufacturer. Player runs in the on-board computer and interacts with the CAN sensor network on-board the robot. Making use of the wireless network interface, the Player sensor server offers wireless remote access to the sensors and actuators in the robot. The Player sensor server and the wireless interface transform each TXT truck in a versatile robotic platform.

Player interacts with the on-board CAN network via a CAN Player Driver, explained in detail in chapter 5. Once Player is running, it allows the user, connected through a client-side application, to log in and access all the information available from sensors in the robot. Client side applications connect via a standard TCP socket.

Communication is accomplished by exchanging a small set of simple messages. As with all open source software, every piece of code can be changed to suit particular needs. The TXT trucks use the off the shelve version 1.6.2 of Player.

4.2.1 Drivers, Interfaces and Devices

Player makes a particular difference between devices, interfaces and drivers. **Drivers**, support specific pieces of hardware, like the TXT CAN driver, that supports the TXT trucks, laser range finders such as the SICK LMS200 and many other equipment. An **interface** is a generic mechanism to get data in and out of a piece of hardware, a GPS **interface** will specify how to get latitude and longitude from the hardware, and might work with different **drivers** for both a GARMIN or Magellan, which support different hardware. Finally, a **device** describes a particular **Driver/Interface** pairing. Devices are given an index, so Player allows the system to have more than one sensor of the same kind.

Player supports a variety of different hardware, whole robots, like rflex, activmedia's pioneers and segway RMP's. Cameras, such as the Cannon VCC4, Sony EVID30 and most FireWire cameras, and a variety of Sick laser range finders. Since the TXT sensors all run on a CAN network, an specific CAN driver was written to interface with our robots, all the sensors but the firewire cameras use this driver that is explained in the next chapter.

Player supports an extensive number of interfaces, almost any sensor you can think of in robotics is supported. Currently our system uses the **position** interface, that provides (X, Y, θ) location of the robot and accepts linear and angular velocity commands. The position interface is the basic control interface for the robot. The **IR** interface provides infra-red range data and accepts no commands. The **GPS** interface reads latitude and longitude from the on-board GPS. The **SONAR** interface is also supported for a future addition of a sonar array to the sensor pool. The Inertial Measurement Unit (IMU) provides yaw, pitch and roll information, that combined make it possible to use the **position3D** interface that provides 3D localization data.

The **PTZ** or pan-tilt-zoom interface is also supported and allows the user to move the camera located in front of the robot. As mentioned before, all of this interfaces, supported by Player, are implemented in the single TXT Driver developed for the TXT trucks since they all run off the CAN sensor network.

The TXT Trucks are suited with a firewire camera. Player supports FireWire video and provides a driver called **camera1394**. The driver is used in conjunction with a **cameracompress** driver that reads the raw camera data and provides a compressed JPG image.

4.2.2 Configuration Files

In order to tell Player what **devices** (or **Driver/Interface** pairs) will be used, a user must provide a configuration file. Such file is a plain text file that the server reads at startup to load the desired interfaces, and match them with the selected drivers. The configuration file is provided as the last argument at prompt when executing Player. The purpose of the configuration file is to map physical or even simulated devices to logical Player devices.

The following example of a configuration file maps a SICK laser range finder connected to the serial port 0 to the Player device $laser : 0$

```
driver
(
  name "sicklms200"
  provides ["laser:0"]
  port "/dev/ttyS0"
)
```

An example configuration file for a TXT robot will look like this:

```
# Config File MARHES TXT-1
# Daniel Cruz 2006


driver
```

```
(
  name "clodbuster"
  provides ["position:0" "ptz:0" "gps:0" "sonar:0" "ir:0"]
)
```

The driver is named *clodbuster* as it is the truck model from tamiya, and it provides all the interfaces described before.

As for FireWire cameras, Player provides a driver itself. The configuration file would include the following lines in case a FireWire camera is used in the robot.

```
driver
(
  name "camera1394"
  provides ["camera:0"]
  mode "640x480_rgb"
)
driver
(
  name "cameracompress"
  provides ["camera:1"]
  requires ["camera:0"]
)
```

The MARHES multi-vehicle platform also supports BumbleBee stereo cameras, manufactured by Point Grey technologies. These cameras used a special model, that sends Bayer tiled images to the host that must be specified in the configuration file. To use Player in a system equipped with a BumbleBee, the following directive should be used in the Player configuration file:

```
driver
(
  name "camera1394"
  provides ["camera:0"]
```

```
  mode "640x480_mono"
  bayer "RGGB"
)
driver
(
  name "cameracompress"
  provides ["camera:1"]
  requires ["camera:0"]
)
```

Player also allows interfacing with popular wireless joystick available in the market. Logitech's line of *wingman* USB wireless controllers is supported in Linux, and Player has a driver available for use with it. A joystick may be connected to the system and used to control the robot, but it must be previously specified in the configuration file. Note that when a joystick device is added, the robot will continuously read the joystick commands overriding other commands sent from other Player clients. In order to use a joystick the following section must be present in the configuration file.

```
driver
(
  name "linuxjoystick"
  provides ["joystick:0"]
  requires ["odometry::position:0"]
  max_yawspeed 50
  max_xspeed 0.5
  axes [3 4]
  axis_minima [5000 5000]
  port "/dev/js0"
  alwayson 1
)
```

Communications are a key element in the platform. They are based on wireless

ethernet connectivity. The platform can work with an access point (AP) or in peer-to-peer mode, where no AP is required. In any configuration mode, is useful for the user to know the connectivity status of each of the robots, Player includes a **linuxwifi** driver that interacts with the wireless interface to provide such data. By using the **linuxwifi** driver, users can monitor the *wifi* interface to access link quality and signal strength information. The configuration file must include the directive below to make use of such functionality.

```
driver
(
  name "linuxwifi"
  provides ["wifi:0"]
)
```

4.2.3 Player Utilities

Several utilities described bellow are included in the Player distribution. These tools help user setup the system and verify that all subsystems function correctly.

playerjoy

Allows teleoperation via a joystick or keyboard. Supports the position and position3D interfaces and allows the user to drive the robot via a text based console. Sensorial information is printed back to the screen.

playerprint

Data logger. This tool prints sensor data in the console, useful mainly to verify operation of the devices.

playerv

Playerv is a small Graphical User Interface (GUI) that visualizes sensorial data from the Player server. It also allows the user to perform teleoperation. Playerv requires

the use of an additional library called *librtk* that must be installed in the system previous to the Player installation. When Playerv starts a small window pops up, *Devices* is a menu to show subscriptions to devices, *View* allows to change the look of the GUI, and *File* allows to save screenshots and making MPEG movies.

Playerv was widely used in the development of my thesis for testing purposes, is a simple yet powerful interface to the robot. A screenshot is shown in Figure 4.1 that shows IR range data, position control and camera pan unit control.

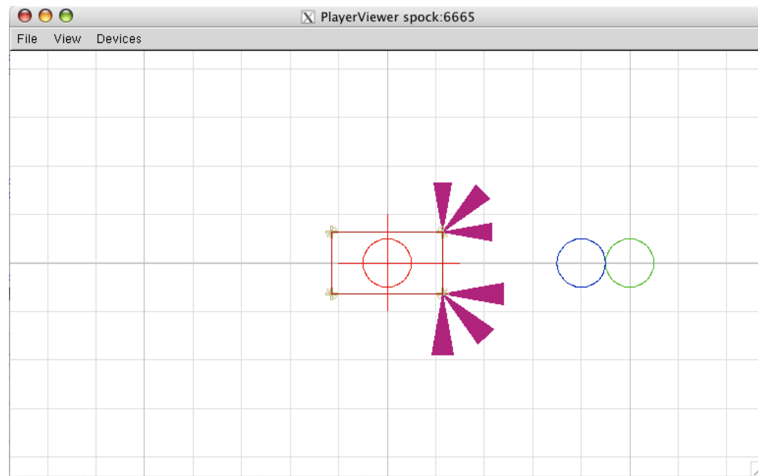


Figure 4.1: Playerv snapshot showing frontal IR range information.

4.2.4 Client Side Operation

Writing a Player client is a simple task. Player offers interface libraries ready to connect and access sensorial data read by the low level driver and provided by the different interfaces.

libplayerc is a client library for the Player robot device server. Was written in C to maximize portability. The library uses a *proxy* model to access the sensor server, so that devices maintain a local proxy for each of the devices in the remote server. There is also an special *client* proxy, to control the Player server itself.

A client program that interacts with the Player sensor server can be broken down into the following steps:

- Create and connect a client Proxy

```
PlayerClient robot("spock");
```

This directive, creates a new client proxy and return a pointer to be used in future function calls. *spock* is the network name of the robot or it's IP address. The function notifies Player that a new client is ready to send commands and receive data.

- Create and subscribe a device proxy

```
PositionProxy* position;
position = new PositionProxy(&robot,0,'a');
```

A pointer is defined to the position interface proxy. The *new PositionProxy* function notifies the Player sensor server that the client is using the position (position: 0) device, and that the client expects to both send commands and recieve data.

- Send commands, get data

```
robot.Read();
position->SetSpeed(v,w);
```

A single *Read*() will take care of reading data from all subscribed proxies. Additionally, the position interface accepts the command *SetSpeed*() which will command the robot to move at an specific linear and angular rate.

- Disconnect from Player

```
robot.Disconnect();
```

This directive tells the server that the client is shutting down.

- Compiling the Program

```
g++ -o goto client.cpp -lplayerclient
```

> The program may be compiled using GCC linking the libplayer client side library.

This section is meant to describe the basic operation of the sensor server, for a more detailed and complete description of the operation of Player, visit their website [1].

4.3 Gazebo, 3D World Simulator

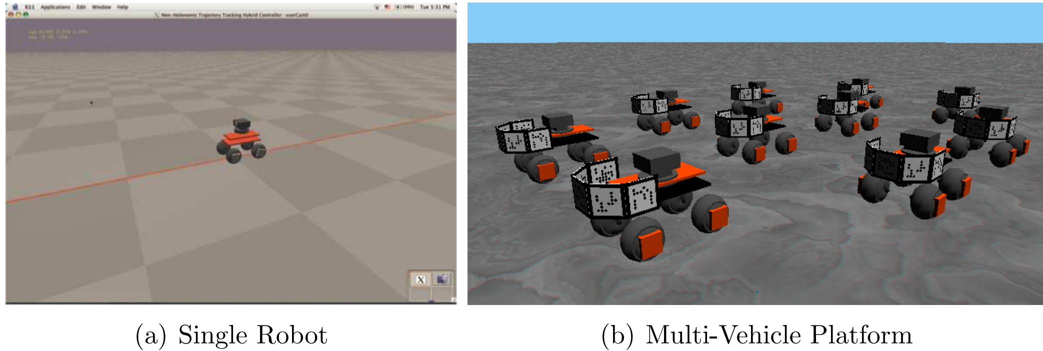
Gazebo is a 3D multi-robot simulator capable of simulating populations of robots, sensors, objects and environments. It also simulates sensor feedback and interactions between robots and objects. Gazebo has a standard Player interface, so that software written for simulation, works on the real robot and vice-versa.

As Player, Gazebo is open source. It was primarily developed for X86 Linux platforms using GCC and GNU, but can be easily ported to X11 systems and OpenGL extensions. It runs on Apple OSX too.

Gazebo makes extensive use of third party libraries like the Open Dynamics Engine (ODE) and the Geospatial Data Abstraction Library (GDAL) which make Gazebo a complex package to configure and install. However once installed on the system, Gazebo is simple to understand and use.

Gazebo can simulate one or many robots in different indoor and outdoor environments, a variety of sensors are also supported. Video cameras, laser range finders and sonars for example, are simulated too, and output data as it would its real counterpart. Gazebo makes use of the Open Dynamics Engine (ODE) to simulate rigid body dynamics which make the 3D simulated world accurate in collision response. The *clodbuster* model of the TXT truck is already part of the Gazebo distribution, so there was no need of any modification and the simulator is used of the shelve. Figure 4.2 shows a single TXT truck with an onboard camera and the complete multi-vehicle platform.

[1] http://playerstage.sourceforge.net/



(a) Single Robot (b) Multi-Vehicle Platform

Figure 4.2: Robots simulated in Gazebo.

4.3.1 World Files

The world file contains a description of the world that will be simulated by Gazebo. It includes the features of the environment, light sources, the solid shapes it might contain and the robots to be simulated, along with a layout of the sensors it may use.

Gazebo world files are text files written in XML or eXtensible Markup Language, a markup language designed to describe data, similar to HTML.

Examples of world files can be found with the Gazebo distribution in the worlds directory. A world file starts with a block of XML meta-data as can be seen next.

```
<?xml version="1.0"?>
  <gz:world
    xmlns:gz='http://playerstage.sourceforge.net/gazebo/xmlschema/#gz'
    ...
```

Global parameters of the environment are described:

```
<params:GlobalParams>
      <gravity>0.0 0.0 -9.8</gravity>
    </params:GlobalParams>
```

The world file is then required to describe the GUI components such as the location and orientation of an observer camera used to visualize the environment.

```
<model:ObserverCam>
      <id>userCam0</id>
      <xyz>0.504 -0.735 0.548</xyz>
```

```
      <rpy>-0 19 119</rpy>
      <window>
        <title>Observer</title>
        <size>640 480</size>
        <pos>0 0</pos>
      </window>
    </model:ObserverCam>
```

Light sources, planes and terrains are basic building blocks of Gazebo worlds, those are described next.

```
<model:LightSource>
      <id>light0</id>
      <xyz>0.0 0.0 10.0</xyz>
    </model:LightSource>
 <model:GroundPlane>
      <id>ground1</id>
    </model:GroundPlane>
```

Robots and objects as parts of the environment will be described next. Each of these objects is identified with a Gazebo id and is set in a particular (X,Y,Z) location and a set of angles identified by (r,p,y) in the file that specify roll, pitch and yaw. The following snippet locates a TXT truck robot in the environment.

```
  <model:ClodBuster>
    <id>robot1_position</id>
    <xyz>0 0 0.40</xyz>
    <rpy>0 0 45</rpy>
  </model:ClodBuster>
```

The world block is then closed with the following statement.

```
 </gz:world>
```

4.3.2 Working With Player

The Player sensor server treats Gazebo exactly the same as it would treat a real robot. Thus, the algorithms developed and tested in Gazebo can run with virtually no changes in the real robot. This feature speeds development as testing time is reduced considerably.

In order to have Player control the Gazebo simulated robot, simulated drivers have to be used, the Player configuration file will inform the server that the controlled robot is being simulated in Gazebo for it to take data properly.

The basic steps to setup a simulated robot in Gazebo and control it with the sensor Player are as follows:

Write Gazebo world file

World file were introduced before. Following that format, one or multiple robots can be simulated in Gazebo and suited with an array of sensors. In this particular example a Clodbuster truck suited with a camera will be shown.

```
<model:ClodBuster>
  <id>robot1_position</id>
  <xyz>0 0 0.40</xyz>
  <rpy>0 0 45</rpy>
  <model:SonyVID30>
    <id>robot1_camera</id>
    <xyz>0.15 0 0.20</xyz>
    <rpy>0 0 0</rpy>
  </model:SonyVID30>
</model:ClodBuster>
```

Start Gazebo

Once Gazebo is provided with the appropiate world file, it may be started. Note that Gazebo has to be running before attempting to run Player.

Write Player configuration file

The next step is to write a configuration file for Player. Since the sensor server will run on top of Gazebo, the drivers used are Gazebo drivers identified by the *gz* prefix.

```
position:0
(
  driver "gz_position"
  gz_id "robot1_position"
)

laser:1
(
  driver "gz_camera"
  gz_id "robot1_camera"
)
```

Start Player

Player must be then provided with the configuration file and started. However the command line varies when using Gazebo and an additional flag must be used. The following command will start Player and command it to connect to the simulated robot.

```
player -g default <config file>
```

Start Client programs

At this point, Player is already running and accepting commands in port 6665 as usual. Clients may then connect to the *localhost* interface if running in the same machine.

Chapter 5

Player - TXT CAN Driver

5.1 Introduction

The TXT Player driver is a key element in the system. It functions as a bridge between Player, the sensor server and the on-board CAN sensor network. The Player driver is in charge of forwarding commands from the Player server to the CAN network of actuators in the robot, and data from the sensors to the sensor server. The driver itself took an important share of developing time, as it required familiarity with the operation of Player and the CAN network. As more devices are added to the robot, the driver will have to be modified to support them. The driver was written with scalability in mind, and it is fully documented to ease the addition of more devices. This driver makes the MARHES multi-vehicle platform compatible with the Player/Stage/Gazebo distribution, it can be used and distributed as open source.

The Player sensor server interacts with a client side application, receiving commands and submitting sensor data as shown in Figure 5.1. Once the client has connected, the TXT Player driver performs a loop that continuously reads the CAN sensor network and have the most recent sensor data available for the client side application.

The TXT Player driver contains routines to deal with the navigation sensors, process the data and compute accurate odometry. It can also use Global Positioning System (GPS), or Inertial Measurement Unit (IMU) information data, if available

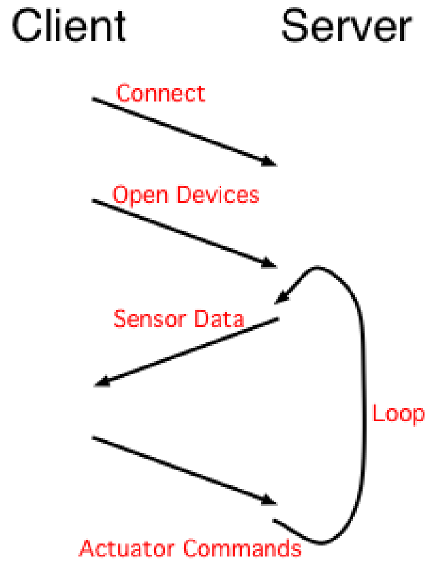


Figure 5.1: Player client-server operation.

and merge it with encoder information to enhance the robots position estimate. This software routines are explained in this chapter.

5.2 Development Tools

The driver was developed using Apple's XTools development environment, however this tool was only used as a typing editor, and any other could be used as well. Player drivers are compiled with the C++ compiler, embedded with most linux distributions. The compilation process however, needs to link many Player specific files to inherit the functionality of the sensor server. To minimize the amount of modifications to the standard Player distribution in order to set the driver up, the existing ClodBuster driver folder was modified. The Player stage distribution includes drivers for most commercially available robots such as the Segway RMP and Mobile Robotics Pioneers. It also provides a ClodBuster driver developed at University of Pennsylvania to interface with their robots. The folder containing this ClodBuster driver was modified to hold the MARHES TXT driver sources, and the Makefile was slightly altered to include the CANLIB functionality and link supporting libraries.

The driver installation is part of the Player installation. Installation scripts are

provided to setup the Linux machines, copy the Player distribution and set the driver files in the right place before compiling the sensor server. Once the driver sources are copied into the Player software suite files, compiling and installing Player will also compile and install the TXT driver.

5.3 Hardware Support

The TXT Player driver is a bridge between the Player sensor server, and the CAN sensor network. The driver itself is a chunk of code inside player and to interface with the CAN sensor network requires hardware and software elements. A PCMCIA-CAN card or USB-CAN connector is used as the interfacing hardware. Also the KVASER CANLib library of functions is used as the software element. Interfacing with the cameras is done with Player's own driver, but it requires additional hardware and software just like the TXT driver. The hardware used for interfacing with the cameras is a firewire PCI card, and the software, the Firewire Linux libraries.

Figure 5.2 shows a block diagram of how the TXT Player driver interfaces with the existing software and hardware. Elements in blue are software parts of Player, in orange are third party software libraries, and in grey, hardware. As mentioned before the driver interfaces with the CAN network on the robot via a PCMCIA CAN card or a USB-CAN interface, both manufactured by KVASER. Player interfaces with the FireWire cameras using it's own driver, called *camera1394*. The driver interfaces with the firewire camera using a PCI firewire card.

5.4 Third Party Library Support

KVASER additionally provides a software library to interface with the card called **CANLIB**, which provides read and write, as well as configuration functions to send and read messages from the CANbus.

The firewire cameras are supported by the ieee1394 support built into the Linux kernel and the open source software libraries libraw1394 and libdc1394. These two

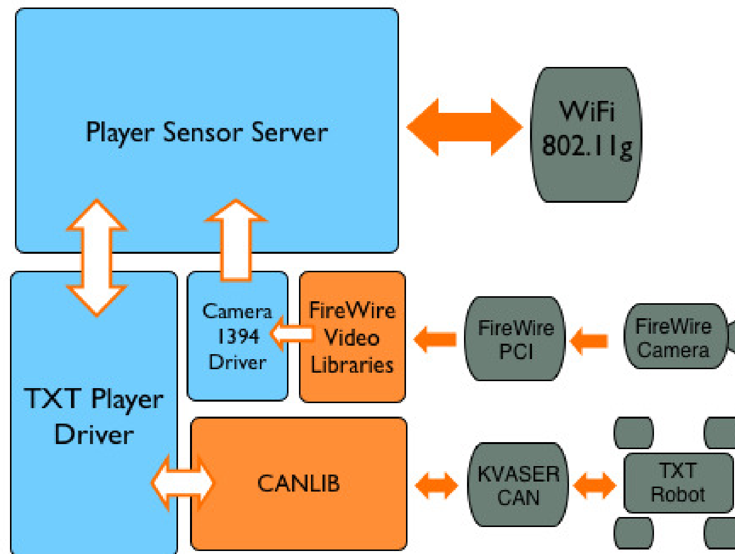


Figure 5.2: TXT Player driver block diagram.

libraries provide video specific functions to get images from the camera, and support the Player sensor server camera drivers.

Player also interfaces with the wireless adapter in the computer via the Linux wifi subsystem. By using the *wifi* interface, users can monitor the status of the network connection, which is critical for the connectivity of the robot, and success of communications.

5.5 Driver Description

Figure5.3 shows how the overall system architecture of Player. Each device has a single read and write thread that loops continuously getting device data, and sending commands. The whole CAN network is treated as a single device and it is serviced by a single thread. Other devices such as cameras, have a separate service thread. For more detailed operation of the Player sensor server, please refer to the Player/Stage software suite documentation in the project website [1].

[1] http://playerstage.sourceforge.net/

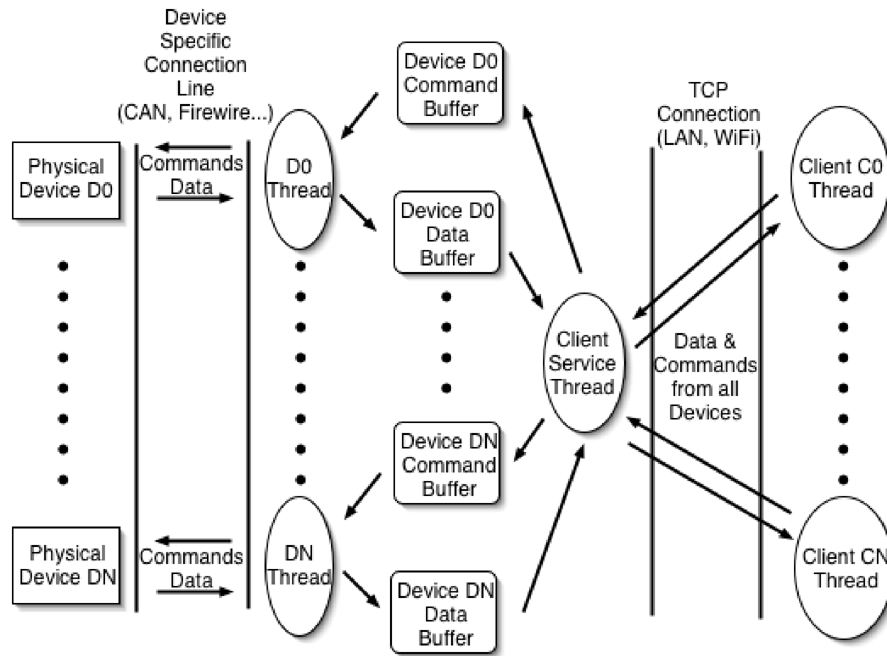


Figure 5.3: Player driver operation [1].

5.5.1 Supported Devices

At the time of writing this document, the supported CAN enabled sensors are, Garmin GPS, CANBot control unit, MicroStrain IMU and IR distance sensors.

5.5.2 Operation

The Player TXT driver interacts with the CAN network of sensors, via the KVASER CAN card and the CANLIB library. The library provides the commands to complete the information exchange between the computer and the CAN network. The driver is divided in two C++ files. The first file *packet.cc* contains code that enables the driver to exchange information with the robot. This file uses the CANLIB library to provide the higher level with data transfer functions. Packet, provides five (5) functions to interface with the low level hardware as follows:

- **UpCANBUS** This functions enables the CANBUS for operation. It configures the bus and opens a communication channel. Must be called to initiate the CAN message transfer.

- **DownCANBUS** This function closes the CAN channel. To be called when disconnecting of the CAN network.

- **ReadCANBUS** A single read takes care of assigning data from each sensor to the appropriate data structure. The ReadCANBUS function reads information from the bus, based on the CAN message identifier decodes the information and fills each device data structure. This function is called at a 1ms rate to update sensor information.

- **SetSpeed** This function interacts with the CAN network to send commands to the low level speed controller. The SetSpeed function passes the linear and angular speed parameters as set points for vehicle control.

- **SetServo** The SetServo function commands the PWM chip in the CANBot controller to any desired position. It is mainly used to control the pan and tilt of cameras in the TXT platform, by using Players' *ptz* interface.

The second file in the TXT Player driver is *clodbuster.cc*. It comprises the configuration required for Player to setup data transmission, supported devices and a few odometry functions.

The TXT Player driver has the following entries:

- **Setup** A setup entry called at initialization to instantiate all the supported devices.

- **Shutdown** Called at program termination to deallocate memory and call cleanup services.

- **Main** A main block that continuously reads the CANBUS, updates the device data structures and sends commands to the actuators based on the client commands. The main loop has a execution rate of 1ms. The main loop operation is explained more in detail using pseudo code at the end of this section with algorithm [1].

- **Calculate2DOdometry** This function integrates encoder counts and provides 2D position using Player's *position* interface. If an IMU is detected in the robot, this function automatically introduces the yaw reading to provide what is called **GyroEnhanced** odometry. This improved version of odometry has reduced error in the heading calculation since it provides from the vehicles' on-board yaw gyroscope, and improves greatly the accuracy of the estimated vehicle position. The CANBot unit provides encoder counts as a CAN output. The unit transmits the difference between two consecutive readings spaced 20 ms in time. This speed measure can be integrated over the time to have an estimate of the distance the robot has traveled. To be able to get a position estimation of the robot, however, it is also necessary to have an estimation of the heading or bearing of the robot's body with respect to a reference frame. The reference frame is usually assumed to be at the vehicle original location, where all values of X position, Y position and θ heading are reset to zero. The values may be set to non-zero to localize the vehicle at a certain initial location. Heading of the vehicle may be provided in two different ways: (i) Integrating the difference of counts on opposing side wheels, which yields angular velocity; (ii) or by using an Inertial Measurement Unit IMU or gyroscope to measure directly the turning rate. The CANBot unit encoder output is used to get an estimate of the angular velocity of the vehicle subtracting encoder counts; however, due to the low angular speeds of the robot, the relatively low linear speeds and the relatively low encoder resolution, this angular speed computation will suffer from low resolution. Since heading is the result of angular speed integration, coarse resolution affects the accuracy of the heading output. An IMU offers far better heading estimates. IMU's may derive this output from fiber-optic gyroscopes, electromechanical (MEMS), or magnetic sensors. Gyroscopes, fiber-optic or coriolis, suffer from bias in the output that affect the heading reading even when the vehicle is stationary. Magnetic sensors are not accurate close to buildings, and useless in indoor environments. The TXT driver offers automatic

adaptation to merge odometric and IMU information if available to generate the most accurate position estimate.

- **GPSCorrection** This function uses the GPS reading if available, to bound the always growing odometric error. The encoder derived position output has far better resolution, since odometry and GyroEnhanced odometry can provide millimetric resolution output for the position of the robot. However, this measure has a growing unbound error that cannot be compensated using dead-reckoning or internal state sensors alone. A very good dead-reckoning system will have about 1% of traveled distance of error. That means that on a 1 km run, the system might be 10 m off the target, and still think is right on spot. GPS units offer a position output with low resolution of about 1 m to 3 m with accuracies close to 5 10 m. The TXT driver uses odometry all the time but uses GPS information when available to reset the integral dead-reckoning error. This results in an overall system where the error is bound to the maximum error of the GPS unit and the resolution is as good as the odometric sensors provide.

- **HandleConfig** is a configuration handler. The sensor server replies to a client that wants to use a interface with this command, with interface specific information. A configuration request reply for an interface such as the **IR's**, will contain the location of the **IR** sensors in the robot. Configuration requests for all supported devices are acknowledged using the data in the configuration handler. HandleConfig is a Player specific function and must be written in accordance. More information on particular interfaces configuration handlers, can be found in Player's documentation.

Once the driver is in operation in the robot, it executes a loop of operations described by the following algorithm [1]:

Algorithm 1 PseudoCode of TXT Player driver.

```
1:  Acknowledge Client Interfaces
2:  Start CANBUS
3:  Start Query Thread
4:  while Driver Up do
5:     Get Commands from Server
6:     Send Servo Commands to PanCamera
7:     Send Servo Commands to Motors
8:     Read CANBUS
9:     Update IR Range Data
10:    Update GPS Data
11:    if New Encoder Data then
12:       Compute 2D/3D/GyroEnhanced GPS Corrected Odometry
13:    end if
14: end while
```

5.6 Plug and Play Operation

As it was mentioned on the hardware description section, the sensor network on board the robots is capable of reconfiguring sensors *on-the-fly*. When a new sensor is added and connected to the sensor network, it is detected and activated to broadcast the device information. Such capability is exploited by the high level software. All devices connected to the robot can be connected in run time and the system reconfigures automatically to start using them. The Player driver monitors devices at a low frequency of 1 Hz, and raises a disconnection flag in case the device is not present in the network anymore.

5.7 Scalability

Scalability is important for the TXT driver since sensors might be added to the CAN network, and users need to interact with them via Player. When adding a new device the first thing to define is which interface will support it. The IR interface supports IR devices, the PTZ interface supports the camera panning mechanism and so on. Once the interface is identified, an entry in the configuration handle must be added that will inform Player that the driver supports a new interface. Then, a read from the server and write to the device, and/or a read from the device and write to the

server, depending on the characteristics of the device itself, has to be added to the main loop in the driver. Addition of devices to an existing driver is fairly simple. The development of a new Player interface for an unsupported sensor goes beyond the scope of this document and should be studied in the Player/Stage documentation.

5.8 Navigation Subsystem

Most of the algorithms developed for the testbed rely on a *go-to* function or a potential field controller, that drives the vehicle from it's current location to the target's location. For this operation to be successful, the vehicle needs to know with some degree of accuracy, its position relevant to the targets. The subsystem in charge of evaluating the robots' position is called positioning system. The positioning system is based on internal state sensors (i.e., encoders) and external state sensors (i.e., GPS). As it was explained in previous sections, the system configures the available devices automatically to produce the best possible output. The positioning system is part of the Player TXT driver. Navigation, on the other hand, uses positioning system information to drive or guide the robot, and its usually written as a Player client application.

If the vehicle is not suited with any other navigation sensor, positioning will be based solely on encoder information. This type of positioning is called *odometry.* If the robot has an on-board Inertial Measurement Unit (IMU), then the positioning system output will merge gyroscopic information to produce *gyro-enhanced odometry.* If the robot is equipped with a Global Positioning System (GPS) sensor, then the output of the positioning system is *GPS-corrected odometry.* In its basic configuration, each robot produces only an *odometry* derived positioning output that can be improved with additional navigation sensors. The experimental section in chapter 8 presents a thorough analysis of each positioning technique.

5.8.1 Odometry-derived Position Estimation

Each vehicle is equipped with front-wheel encoders by default. By using wheel rotation information solely, it is possible to estimate the distance the robot has traveled and its heading. This information together can be used to estimate the vehicle's position. The position of the robot is defined by a vector$[x, y, \theta]$ where the duple x, y is the position on a cartesian plane and θ is the heading. The position of the robot is referenced to its original location where the system is started up. It is usually regarded as the origin $(0, 0, 0)$, but can be easily changed by the user if necessary, via a Player client command.

In order to compute the vehicle's position, the CANBot board forward the traversed distance in encoder ticks at each sampling interval. The onboard computer integrates that information to derive linear and angular speed, and compute the position estimate.

Once the encoder ticks are received, the angular and linear velocities are computed using the following equations:

$$\upsilon = \frac{(d_R + d_L)\zeta}{2\Delta T} \tag{5.1}$$

$$\omega = \frac{(d_R - d_L)\zeta}{d_W \Delta T} \tag{5.2}$$

Where $\zeta = 0.3$ mm/tick is the distance traveled by a wheel in one encoder tick, d_W is the wheel separation or robot width. And $\Delta T = 20$ ms is the sampling time.

This values are hard coded in the CANBot board firmware. d_R and d_L refer to the traversed distance measured by the right and left wheels in ticks, during one sampling interval.

Once the linear and angular velocities are estimated, the heading is calculated by integrating the angular speed at each sampling interval:

$$\theta^+ = \theta + \omega\Delta T \tag{5.3}$$

With the heading of the vehicle calculated, the system uses the following equations

to calculate the position estimate. Note that each coordinate is calculated based on the previous estimated location:

$$x^+ = x + \upsilon \cos\theta \Delta T \tag{5.4}$$

$$y^+ = y + \upsilon \sin\theta \Delta T \tag{5.5}$$

At system start-up, the output of the positioning system is the origin, that is $x = 0$, $y = 0$ and $\theta = 0$. However, using Player's position interface, the original location may be shifted to any other location. This feature is useful when using multiple robots, in the same reference frame, as not every vehicle can physically start at the same location.

5.8.2 Gyro-Enhanced Odometry

Gyro-enhanced odometry presents an improvement over odometry, by adding a gyroscope or another inertial measurement sensor. Inertial units can combine sensors such as gyroscopes and magnetic compasses to measure turning rates and robot attitude. The typical problem of inertial measurement units (IMU's) is drift over time. The traditional way to deal with this inconvenience, is to bound the run time or use some sort of landmark or external reference.

If the robot is equipped with an IMU, the system automatically reconfigures itself to use gyro-enhanced odometry. With this technique the traveled distance is still measured by encoders as in plain odometry, but the turning rate of the vehicle is measured by the inertial sensor. If the robot is equipped with gyro-enhanced odometry, the heading is calculated using equation (5.6) instead of (5.3) as follows.

$$\theta = \varphi IMU_{CF} \tag{5.6}$$

The constant IMU_{CF} is a conversion constant specified by the IMU's manufacturer. And φ is the raw yaw angle output by the IMU.

In plain odometry, vehicle heading is calculated by integrating angular speed over time. Angular speed is calculated by taking the difference of the encoder counts on opposing side wheels on each sampling interval. Since each sampling interval has a brief duration of 20 ms, the resolution of this measure is relatively low. An IMU measures turning rate and heading with increased resolution, alleviating this problem.

5.8.3 GPS Corrected Odometry

The simplest position estimation that can be done is odometry. It is cheap and easy to implement in real-time. The only disadvantage is the unbound accumulation of errors. The addition of an IMU allows gyro-enhanced odometry to have less heading accumulated error, but to bound system error, an external reference must be used. This reference can be a set of landmarks spread in the environment, such as visual tags or reflective marks. In outdoor environments, it is possible to use the Global Positioning System (GPS) maintained by the United States government. If available, the testbed will use a GPS receiver to localize the vehicle. Since the GPS sensor outputs globally referenced coordinates, the positioning system error is always bound to the error of the GPS receiver, and does not depend on how much distance the robot travels.

The GPS consists of a constellation of 24 satellites that orbit the earth in 12 hours. There are often more satellites available, since new ones are launched to replace older ones. The satellites are evenly spread over the earth, so that a user in any point of earth's surface can have between five and eight visible satellites. The GPS receiver outputs its location over the earth surface in terms of latitude and longitude. Each GPS receiver requires a transient time of about five seconds to acquire satellite signal and compute its location.

GPS longitude and latitude is available from the *GPS* Player interface. However, in order to integrate the GPS and on-board position information, the GPS output must be translated into cartesian coordinates. This is accomplished by using equation

(5.7) and (5.8).

$$x_{GPS} = (Long - Long_O)Long_{Dist} \tag{5.7}$$

$$y_{GPS} = (Lat - Lat_O)Lat_{Dist} \tag{5.8}$$

Where $Long_O$ is the longitude at the origin, Lat_O is the latitude at the origin, and:

$$Long_{Dist} = 2000\pi 6378.137 \frac{\sqrt{1 - \left[\frac{z}{6356.7523}\right]^2}}{360} \tag{5.9}$$

$$Lat_{Dist} = -0.003{Lat_O}^3 + 0.5{Lat_O}^2 - 1.2Lat_O + 110570 \tag{5.10}$$

$$z = 6378.137(1 - 0.00669438)\frac{\sin Lat_O\pi/180}{\sqrt{1 - 0.00669438\left[\sin Lat_O\pi/180\right]^2}} \tag{5.11}$$

GPS generates low resolution and low frequency position fixes. On-board odometry has high resolution and high frequency. In order to merge information for both systems, a simple algorithm uses the high frequency odometry information continuously, and uses every GPS position fix when available.

When a GPS position fix arrives, the algorithm calculates the absolute difference between the current odometry position and the received GPS position fix. If the GPS location is assumed as a global reference with no error, the maximum difference between the position readings should be no greater the resolution of the GPS.

An absolute difference greater than the resolution of the GPS system, indicates that the error of the position estimator has grown beyond an acceptable bound. In this case, the position is reset to the position fix of the GPS system. In this way, odometry is used to keep a high resolution position measure, while GPS bounds the positioning error, to the maximum error of the GPS device used. This routine presents a simple, yet effective approach to merge GPS and odometry data.

Figure 5.4 illustrates the operation of the algorithm on a straight path. As the vehicle travels forward, because of systematic and non-systematic errors, odometry degrades and the position output error grows. When the difference between GPS data and odometry reaches the limit allowed, the system position is reset to the GPS loca-

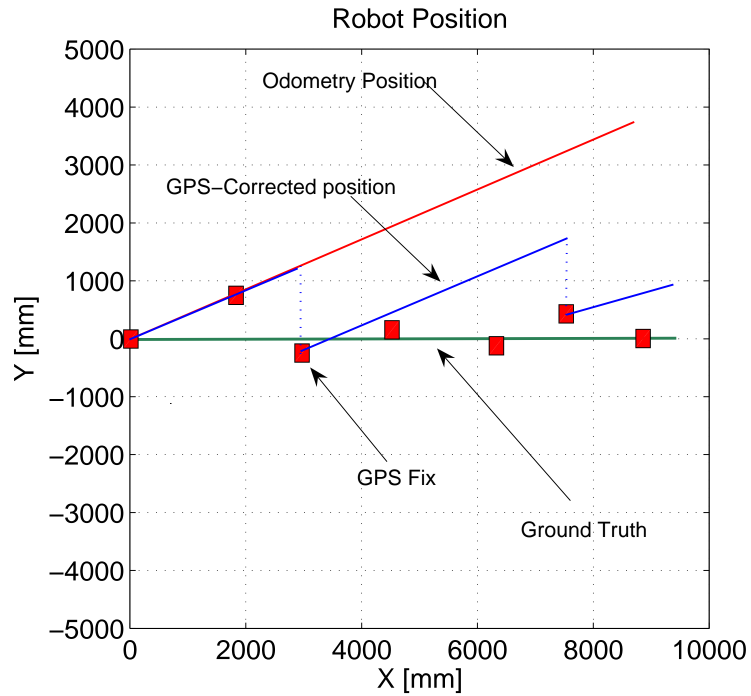


Figure 5.4: GPS corrected odometry operation.

tion. The GPS information is assumed as the ultimate ground truth, thus the overall error will be bound to the error of the GPS device. When triggered, this technique generates a discontinuous position path, which affects navigation algorithms. The algorithms developed for the platform are robust enough to deal with it, however, a better data fusion technique using Kalman filters is being developed.

Chapter 6

Swarming Behaviors

6.1 Introduction

A swarm is a large number of organisms in motion. Swarming can be defined as a number of robots moving in mass, with the objective or reaching a goal, survey an area, or travel safely in group. Swarming is a tactical operation proven to be successful in the battlefield. It is based on the principle of converging highly distributed forces at a single point, to leverage the principle of mass. Swarming can guarantee mission success, if enough powerful forces are concentrated at a particular point of conflict.

The concept of swarming can be extended to different applications, such as exploration, surveillance, and localization and recovery operations [10]. These tasks can be performed far more efficiently with the use of swarm intelligence. The multi-vehicle testbed thay features modularity, communications and distributed processing power, is perfectly suited to research on swarm intelligence. The purpose of this chapter, is to introduce higher-level software routines that use the communication capability of the testbed to create a swarm. A swarm that can be used in a number of different applications.

6.2 Behavioral Building Blocks

This section introduces a set of small but powerful functions that can be used together to compose applications similar to the swarming behaviors described later in this chapter. The functions have been developed in the C programming language using the *playerclient* libraries to interface each of the robots.

6.2.1 Vision Software Library - VSL

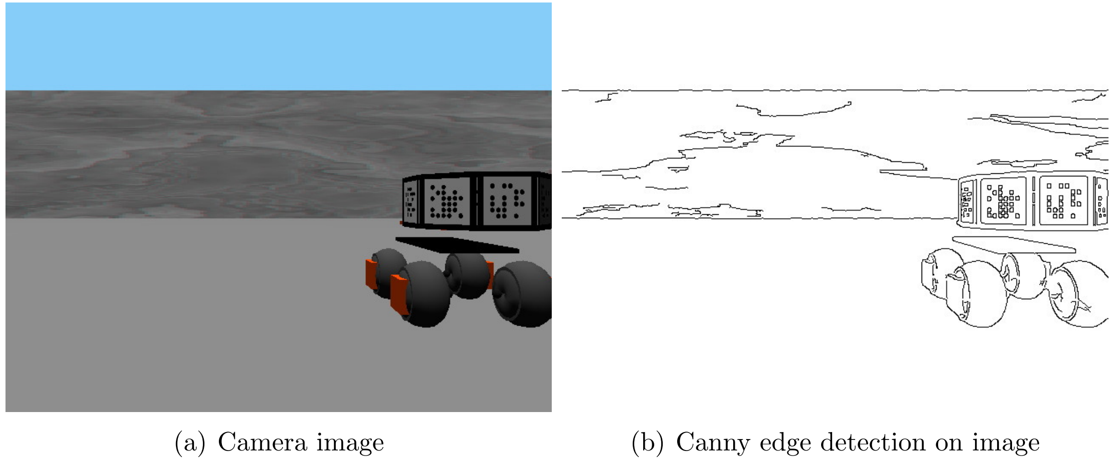
If a group of robots is attempting to perform a coordinated behavior such as flocking, leader-following, or some other multi-vehicle maneuver, it is important for each robot in the group to have a rough estimate of the position and orientation of its neighboring robots. If this information is available, users can implement a variety of advanced control algorithms to coordinate the motion of the group. To accomplish robot identification, shape and color tracking are used. Shape tracking has several advantages over color tracking. Gradient filters are suitable for outdoor use because they are insensitive to large variations in light intensity. Shape tracking can also extract relative distance and orientation information. However, pure shape tracking might not work well in practice because of the probability of false positive readings and data association errors.

The approach presented here attempts to find the relative position and orientation (also known as pose) between robots, adapting the POSIT algorithm [11], [12]. This algorithm is designed to measure the relative 6 degree of freedom (6 DOF) transformation, between the observation camera and the target being tracked. The algorithm reduces the probability of false positive measurements, determines the name of neighboring robots and keeps the system scalable so that it works just as well for ten robots as it would for one thousand. To identify each robot, the algorithm decodes machine readable code to determine the robot's name. Each robot carries a *NameTag* that consists of a 6×6 grid of dots surrounded by a thick black square perimeter. Each NameTag is based on the Siemens SCR matrix code [13]. To reduce the probability of false positive readings, we try to keep NameTags as different as possible. The

separation distance between two NameTags is defined as the number of pixels on which each NameTag differs from another. Using MATLAB sets of 100 NameTags were generated until a maximum distance of 10 was found between them. Each robot carries five NameTags mounted on the back (Figure 6.1) to ease identification by a neighboring robot. In order to extract information of every NameTag, the algorithm needs to perform the following steps:

Figure 6.1: Vehicle carrying a NameTag array.

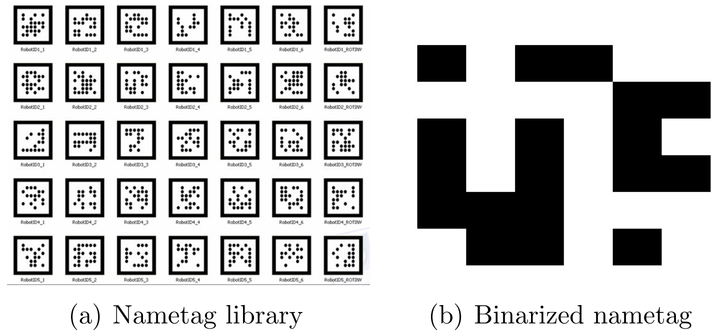
- The original color image is first converted to grayscale and the Canny edge detector is applied to reveal the contours of the image as shown in Figure 6.2.
- The next step is to filter the set of all contours to the set of square contours. All contours are filtered by area and shape constant of the contour.
- Next the algorithm moves along each point of every contour to find areas where the greatest change in angle of the contours occur. These points are called corners and if a given contour has four corners then it is accepted by the algorithm as a square.
- The algorithm then extracts a NameTag from every square shape in the image.
- The NameTag ID is retrieved. The extracted image is converted into a 64-bit number. The algorithm compares the separation distance between the observed



(a) Camera image (b) Canny edge detection on image

Figure 6.2: Example image taken in Gazebo and after Canny edge detection

NameTag, and every NameTag in the Library (Figure 6.3) to determine if the observed NameTag is valid or not.



(a) Nametag library (b) Binarized nametag

Figure 6.3: NameTags identify each vehicle.

- The algorithm now determines point to point correspondence between the model points and image points. To accomplish this, the algorithm has to determine which 2-D image points correspond to which 3-D structural points.

- In the final step the POSIT algorithm is used to get the 6 DOF transformation from the camera origin to the target origin. Given the object model and the object image, a least-squares pose approximation is calculated via the object model pseudoinverse.

Figure 6.4 shows the estimated and true pose of a moving target using the VSL tracking system. During this experiment the robot remains idle, and the on-board camera attempts to calculate the position of another moving robot. Both, the estimated

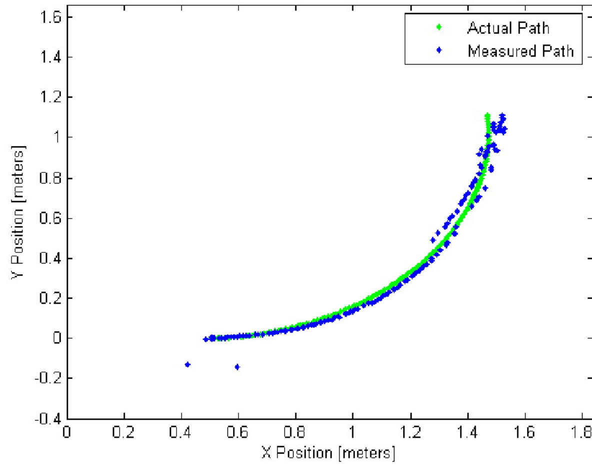


Figure 6.4: Pose estimate of a moving target.

and true path of the moving robot are shown. The accuracy of the system degrades gracefully, as the target moves away from the camera.

6.2.2 Potential Field Controller

The Potential Field Controller (PFC) was one of the first algorithms developed to serve as a *go_to_goal* function in a perimeter detection and tracking application. This algorithm calculates forces in the (X, Y) axes that attract the robot to a desired location in the space, by aligning the robot with the target location [14]. The PFC is based on vehicle odometry and target location that must be given in (X, Y, θ) coordinates relative to the vehicle's positioning world frame. Potential fields have been used by a number of groups for controlling a swarm [15, 10].

The Potential Field Controller (PFC) uses an attractive potential allowing the robots to quickly move to a specific location. PFC may be used in feature tracking, leader following, waypoint tracking or any other application where the robots needs to travel to a designated goal. The attractive potential, $\mathbf{P_a}(x_i, y_i)$ is [16]:

$$\mathbf{P_a}(x_i, y_i) = \frac{1}{2}\epsilon[(x_i - x_g)^2 + (y_i - y_g)^2], \tag{6.1}$$

where (x_i, y_i) is the position of robot i, ϵ is a positive constant and (x_g, y_g) is the position of the attractive point (goal). The attractive force, $\mathbf{F_a}(x_i, y_i)$, is derived

below.

$$\mathbf{F_a}(x_i, y_i) = - \bigtriangledown \mathbf{P_a}(x_i, y_i) = - \begin{bmatrix} \frac{\partial \mathbf{P_a}}{\partial x_i} \\ \frac{\partial \mathbf{P_a}}{\partial y_i} \end{bmatrix}$$

$$\mathbf{F_a}(x_i, y_i) = \epsilon \begin{bmatrix} x_g - x_i \\ y_g - y_i \end{bmatrix} = \begin{bmatrix} F_{a,x_i} \\ F_{a,y_i} \end{bmatrix} \tag{6.2}$$

Equation (6.2) is used to get the desired orientation angle, $\theta_{i,d}$, of robot i

$$\theta_{i,d} = \arctan 2(F_{a,y_i}, F_{a,x_i}) \tag{6.3}$$

After the orientation relative to the goal has been calculated, a proportional controller is used to align the robot with the desired heading.

$$\omega_i = \pm k \ (\theta_{i,d} - \theta_i) \tag{6.4}$$

where $k = \frac{\omega_{max}}{2\pi}$ and $\omega_{max} = 0.4$ rad/s. The linear speed of the robot while executing the PFC may vary. In some applications as feature detection it may be held constant, while in others as *go_to_goal* the speed of the robot depends on the distance to the goal. In leader-follower applications, the speed of the leader may be used as a feed-forward term to increase system robustness.

6.2.3 Obstacle Avoidance

Obstacle Avoidance (OA) is a basic building block of autonomous robotic systems [17]. It ensures safe operation by preventing collisions with the environment or nearby robots. Our basic *Obstacle Avoidance* algorithm uses sonar or IR devices to detect and avoid obstacles. The obstacle avoidance function overrides any other higher-level control to steer the vehicle away from obstacles in the path. It generates a repulsive force that, depending on the distance to the obstacle, could even command the robot to back up [18]. The TXT IR and sonar sensors yield distance to obstacle

information straight ahead, at a 45° angle and at a 90° angle. The 90° sensor gives distance to obstacles in the side of the robot. Making use of these three sensors, our OA algorithm avoids obstacles directly ahead and tracks the shape of an obstacle's perimeter. Each CAN IR device uses three IR sensors. The robot uses two CAN IR devices located in the front so that a total of six range readings from IR sensors are available. These readings are stored in an array where the element in position 0 corresponds to the range reading of the sensor farthest to the right, and the element in position 5 corresponds to the range reading of the sensor farthest to the left. The array of sensors is shown in Figure 6.5.

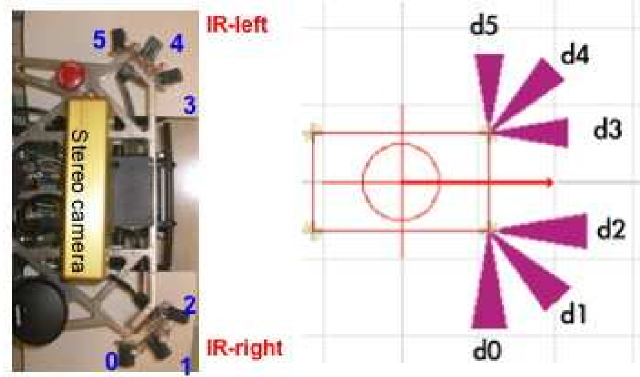


Figure 6.5: Index of IR sensor range data.

The repulsive force in the right and left side of the robot is given by

$$F_r = \sum_{n=1}^{2} \frac{\gamma_n}{d_n}, \qquad F_l = \sum_{n=3}^{4} \frac{\gamma_n}{d_n} \tag{6.5}$$

where d_n is the distance to the obstacle reported by the n_{th} sensor and $\gamma_n = [0.7, 1]$. The use of γ weights the repulsive force in favor of obstacles straight ahead of the robot. The OA controller is a hybrid controller. It switches to track the perimeter of the obstacle if the obstacle is in the way of the designed goal, but not directly in front of the vehicle. If an obstacle has to be negotiated, the following controller is

applied

$$v_k = v_{OA} - k_{vp} \sum_{n=1}^{5} \frac{\gamma_n}{d_n} \qquad \omega_k = k_{\omega p}[F_l - F_r]. \tag{6.6}$$

If the obstacle is negotiated and out of the way of the robot, the higher-level control (i.e., PFC) may continue operation. However, if the obstacle is still in the path of the designed goal location the robot is going to, the perimeter of the obstacle is tracked until the obstacle is completely out of the way. The switching condition is given by $d_{0,5} \geq \rho$ where $\rho = 0.25$ m. In order to track the perimeter of the obstacle, OA uses one of the following proportional controllers depending on the obstacle location:

$$\omega = k_p(d_5 - D_{SAFE}) \qquad \omega = k_p(D_{SAFE} - d_0) \tag{6.7}$$

Where D_{SAFE} is the constant distance at which the perimeter of the obstacle will be tracked. D_{SAFE} was set to 0.2 m for our experiments. The perimeter tracking speed was set constant at 0.6 m/sec.

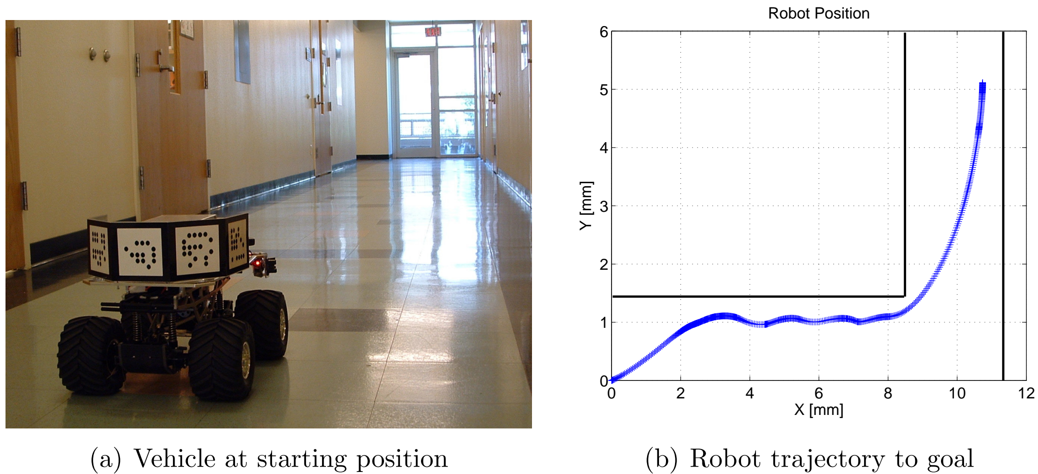
6.2.4 Goal-Seeking

Goal-seeking is a behavior that emerges from the composition of two basic functions: obstacle avoidance and potential field control. The objective of this behavior is to drive the robot to a designated waypoint, without colliding with obstacles that might be in the way. In order to detect obstacles, the robot must be equipped with **IR** or **sonar** sensors. This behavior is useful as it enables the robot to travel autonomously. Most of the applications require such a function and it is included as a fundamental block of the Graphical User Interface (GUI) presented in the next chapter.

A potential field controller drives the vehicle towards the goal following a straight line. However, if an obstacle is detected in the way, the system transitions to obstacle avoidance. The obstacle avoidance controller, as explained before, backs the vehicle if necessary to avoid a collision or tracks the perimeter of the obstacle to negotiate it. Once the obstacle is surpassed, the system returns control back to the potential

field controller to continue towards the goal.

In order to explain in further detail the operation of the algorithm, a simple goal-seeking operation was implemented. Figure 6.6 shows the robot in a starting location $(0, 0)$, the goal where the robot is commanded to travel to, is located at (11 m,5.5 m), around the corner in the hallway. The hallway wall stands in the way between the robot and the only one goal the robot must reach.



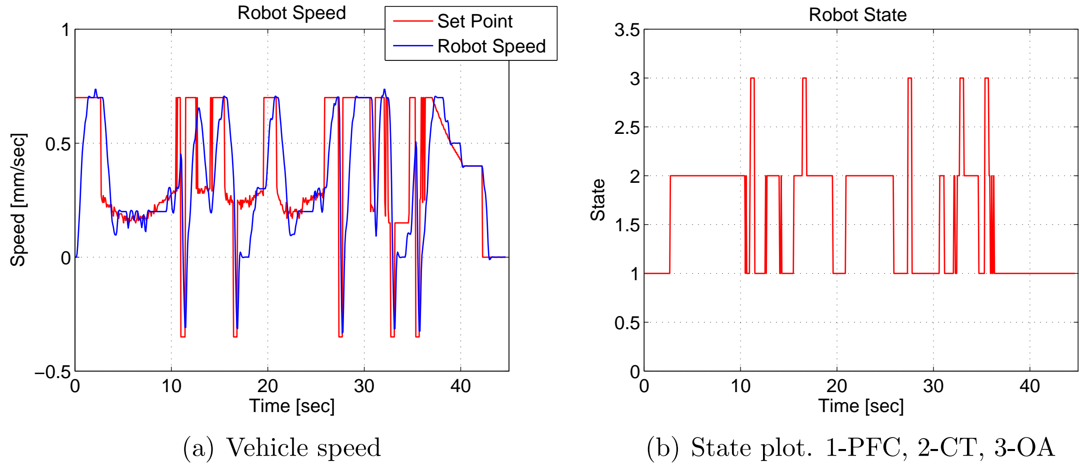
(a) Vehicle at starting position

(b) Robot trajectory to goal

Figure 6.6: Goal-seek experiment setup and trajectory.

As soon as the robot is started, the potential field controller quickly aligns it towards the goal. When the wall is detected, the robot backs up, and starts following the wall. When the wall is not detected anymore, the robot moves again towards the goal. Figure 6.7 shows the transition of the controller between the potential fields controller (PFC), obstacle avoidance (OA), and obstacle contour tracking (CT) depending on the distance to the obstacle.

The *goal-seek* algorithm is not a path planner, thus cannot guarantee that the goal is in fact reached. The algorithm will transition between obstacle avoidance and potential field control continuously, while avoiding obstacles. However, while avoiding obstacles, the vehicle is not aware of the location of the goal and the shape of it might drive it away of the target. The algorithm is intended to be a basic building block function, that can be a part of a library of more complex path planning algorithms. In this particular experiment the algorithm successfully drives the vehicle to the goal with no collisions.



(a) Vehicle speed (b) State plot. 1-PFC, 2-CT, 3-OA

Figure 6.7: Goal-seek experiment speed and states.

6.3 Swarming Behaviors

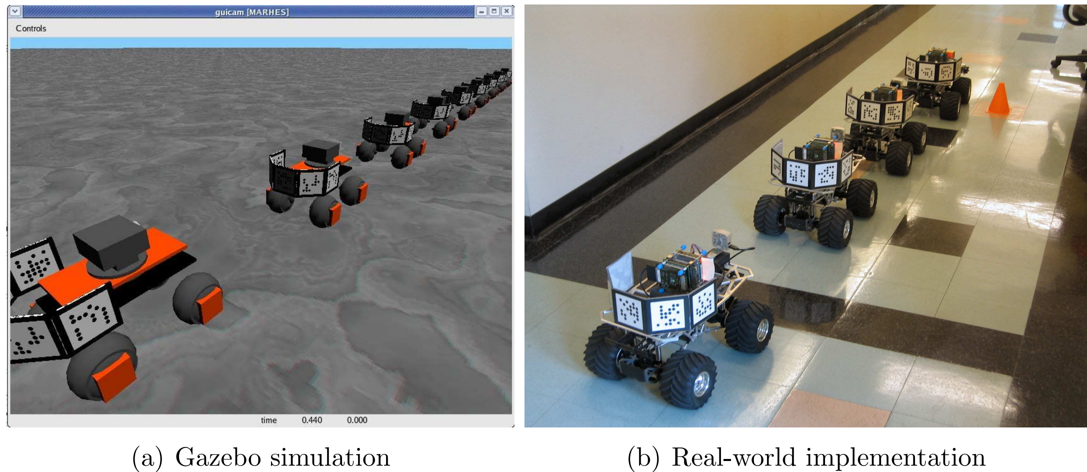
This section describes some of the behaviors developed and implemented with the platform.

6.3.1 Vision-based Formation Control

Formation control is the focus of extensive research in the robotic community [19, 20, 21, 22]. Autonomous vehicle following has a great deal of interest, because of its multiple applications like automated highways, UAV formation flight, and military convoys. Our vision-based formation control uses the on-board camera and VSL, the Vision Software Libary. Robots can be programmed to follow a particular robot at a specific distance and orientation angle. Once the video sensor and the VSL extract the relative position and orientation of the leader, the robot uses the PFC to command the follower continuously to a location behind the leader's path [23]. Figure 6.8 shows the experimental testbed traveling in formation. Each vehicle follows the robot directly in front while the leader is tele-operated.

6.3.2 Communication-based Formation Control

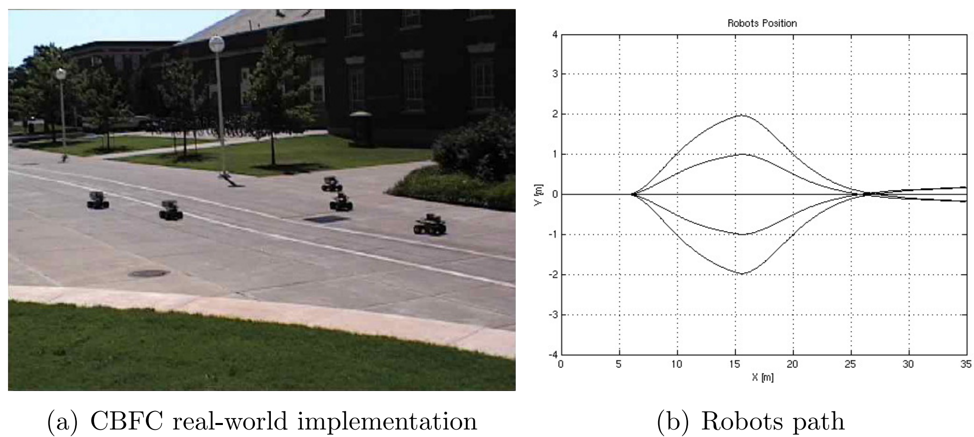
In communication-based formation control, vehicles are able to follow one another based on their position relative to a common world frame. This behavior requires



(a) Gazebo simulation (b) Real-world implementation

Figure 6.8: Video-based leader-following.

each vehicle to communicate its actual position. The followers are programmed to follow the location communicated by its pre-selected leader. Communication-based following is useful in environments (e.g. urban environments) where the leader may not be seen all the time. The success of the technique depends on the accuracy of the positioning estimation system. Additionally, the robots have to share an absolute position reference frame. Figure 6.9 depicts a picture where a group of TXT vehicles



(a) CBFC real-world implementation (b) Robots path

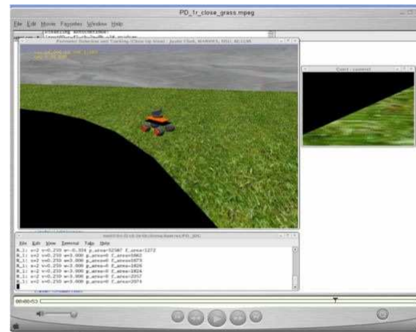
Figure 6.9: Communication-based formation control.

changes from a convoy formation to a V type formation and returns to a convoy formation again. The maneuver is based on odometry and inter-vehicle communications. The cameras do not operate in this experiment. Odometry position information is also shown in Figure 6.9.

6.3.3 Boundary Detection and Tracking

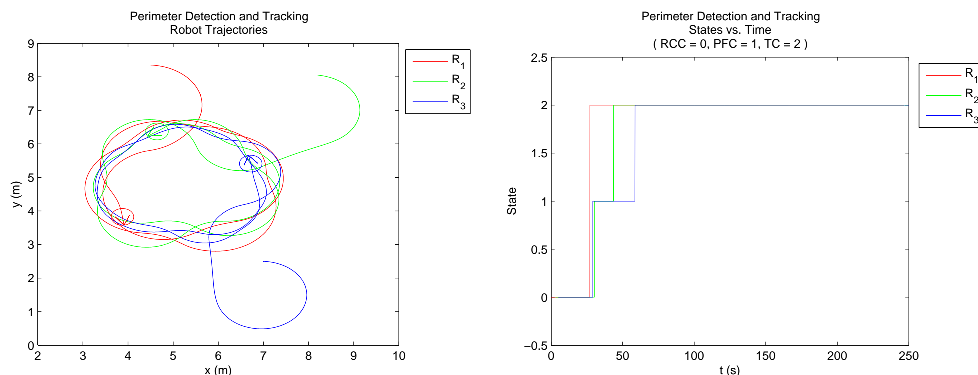
A decentralized coordination algorithm that allows a robotic swarm to find and track a dynamic perimeter was developed and tested [24]. A boundary can be a chemical substance spill, a building, or landmark. Robots must be equipped with an adequate sensor to detect the substance. The algorithm is composed of simple but effective controllers: (i) a Random Coverage Controller (RCC) to start the search for the boundary, (ii) a Potential Field Controller (PFC discussed above) to attract robots to the boundary once it has been located, and, (iii) a Tracking Controller (TC) to track the boundary. Additionally, OA is active in the background throughout the execution of the algorithm preventing collisions. The algorithm was tested with three robots but it can be easily scalable to many more.

(a) PD real-world implementation



(b) Gazebo simulation

Figure 6.10: Boundary detection and tracking.



(a) Robots path

(b) Robots state

Figure 6.11: Boundary detection and tracking plots.

Figure 6.10 shows the algorithm in operation. The RCC commands the robot in

a spiral search path looking for the boundary to be tracked. A blobfinder, part of the vision libraryVSL, uses the on-board camera to detect and track the boundary. Once the substance is found by a robot, it broadcasts its position as the best known boundary location. Each robot that receives the communication, starts the PFC to the boundary location and transmits it again serving as a communication relay. Once a robot finds the boundary, the blobfinder-based TC tracks it, and a cyclic behavior emerges.

6.3.4 Flocking

Flocking [25, 26], has been subject of extensive study in recent years. Biology, computer, math and control communities have focused attention on how crowds of people, flocks of birds, or schools of fish move together in decentralized and coordinated manner. In [27], Vicsek et al., propose a simple model where a number of agents move with the same speed but different heading. Each agent aligns its heading based on the average of the heading of its surrounding neighbors. Despite the absence of a centralized coordination algorithm, the group is shown to align itself in the same direction and a flocking behavior emerges. Jadbabaie et al., [28] describe a theoretical explanation for the observed behavior.

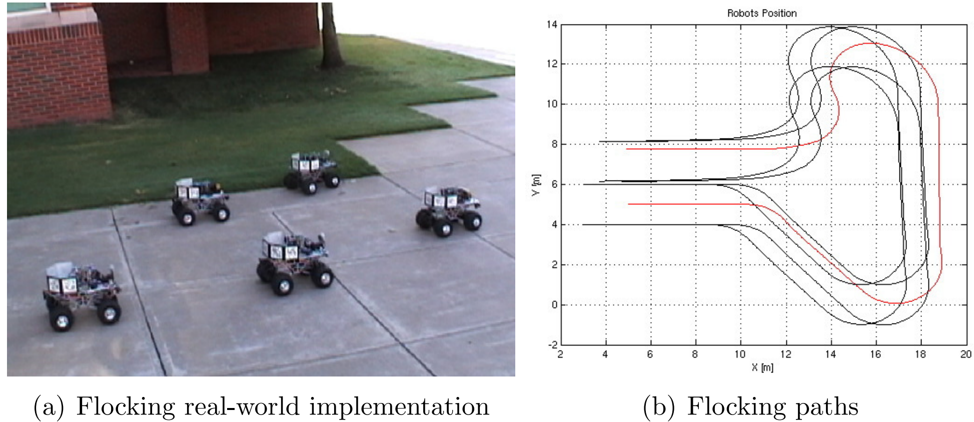
This algorithm has been implemented this algorithm in the platform. Using communications, each robot is aware of the heading of its neighboring peers. A local PI controller enforces the neighboring rule that aligns the heading of the robot to the average of its current heading, and the heading of the neighboring robots. All the group is commanded to follow the same linear speed. OA is active but the implementation was tested in an obstacle free environment.

The desired heading of the $i-th$ robot in the group is given by (6.8) where n represents the number of neighboring robots. A PI controller oversees that the heading in (6.8) is reached.

$$\theta_i = \frac{1}{n}\sum_{k=1}^{n}\theta_k \tag{6.8}$$

Even though the algorithm is in essence leaderless, our implementation had a

leading robot. The leader is commanded to follow a set of way-points using the PFC. As a result, the entire group flocks across the path defined by waypoints. The linear velocity of the robots is set to be equal to the leader's speed.



(a) Flocking real-world implementation (b) Flocking paths

Figure 6.12: Flocking with five vehicles.

Figure 6.12 shows a five-robot team where the leader robot is commanded to follow a series of waypoints.

The swarming behaviors described before are examples of applications already implemented with the platform. They can also be combined to create new applications. Chapter 7 introduces the development of a Graphical User Interface (GUI) which allows users to control the team of robots, and compose missions using the swarming behaviors described here.

Chapter 7

Graphical User Interface

7.1 Introduction

In order to work with a system users need to be able to control and assess its state. The success of the system is often dependent on its ease of use. This relies on how much effort a particular user has to invest in giving input to the system and interpreting the output. Graphical User Interfaces (GUI's) have proved to be more successful than text-only interfaces, since they are easier to understand and work with. Nowadays systems use touch screens with almost exclusive graphical interaction, text interfacing is reserved for system managers for system setup. GUI's take into account the human psychology and physiology to make the process of interacting with the system more effective, efficient and satisfying.

This section describes a Graphical User Interface (GUI) developed to interact with the multi-vehicle platform. The GUI is intended to enable the user to monitor and control a team of robots in an easy to use graphical environment. The chapter introduces the *Qt framework* used to develop the GUI, and explains the use and functionality of the interface.

7.2 The Qt Framework

Trolltech's Qt was selected as the C++ framework to develop the interface. Qt has an extensive array of features, capabilities, and tools to enable programmers to develop high performance, and cross-platform applications. The Qt class library has over 400 C++ classes for application development. It also provides an elegant Application Program Interface (API) with a rich set of object models, and an extensive collection of classes for GUI programming, database programming, and networking.

Qt enables developers to learn a single API and develop applications for multiple platforms. Currently Qt is supported in Windows, MacOSX, Linux, Solaris and many other Unix variants. The *designer* provided by Qt, is a powerful GUI layout designer that reduces development time. It also shares the same look and functionality across all supported platforms.

Qt is easy to use, intuitive and simple, making the programming experience successful and enjoyable. Lastly Qt is also open source.

7.2.1 Qt Desginer

Qt Desginer is a GUI layout and form builder. It enables users to develop rapidly high performance interfaces. The look of the interfaces can be easily altered, or kept standard when used in different platforms. The Qt Designer visual interface, facilitates to design, preview and adjust advanced user interfaces.

The Qt Designer was crucial in the development of the MARHES GUI. A visual development tool minimizes development time and helps produce high quality results.

7.2.2 Qt Assistant

The Qt assistant is a documentation browser embedded with the Qt distribution. It works in a similar way to a web browser giving the user access to a collection of hyperlinked pages that contain all of Qt's documentation.

The Qt assistant also provides fast keyword lookup to access all function and

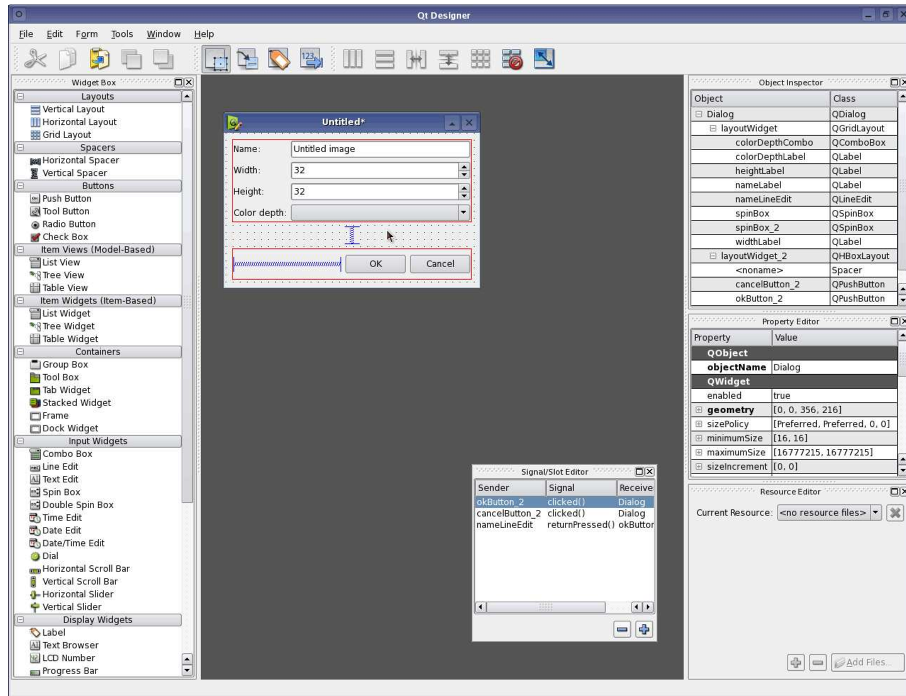


Figure 7.1: Qt Designer.

classes, and full text search capability to access all documentation. The assistant uses Qt's support for rich text and HTML to display the help pages .

7.3 MARHES Multi-Vehicle Graphical User Interface

The MARHES GUI depicted in Figure 7.3 allows the user to monitor and control each individual robot of the testbed as well as the group itself. It has built in swarming behaviors and basic building block functions to enable fail-safe operation of the platform with minimal user interaction, while maximizing ease of use and functionality.

The MARHES GUI uses Player, the sensor network server, to command and monitor the robots. This feature enables the interface to run in any computer or handheld station connected to the network. Since the GUI uses Player commands, the *playerlib* library of functions must be linked at compilation time. The computer where the GUI is compiled must have this libraries installed.

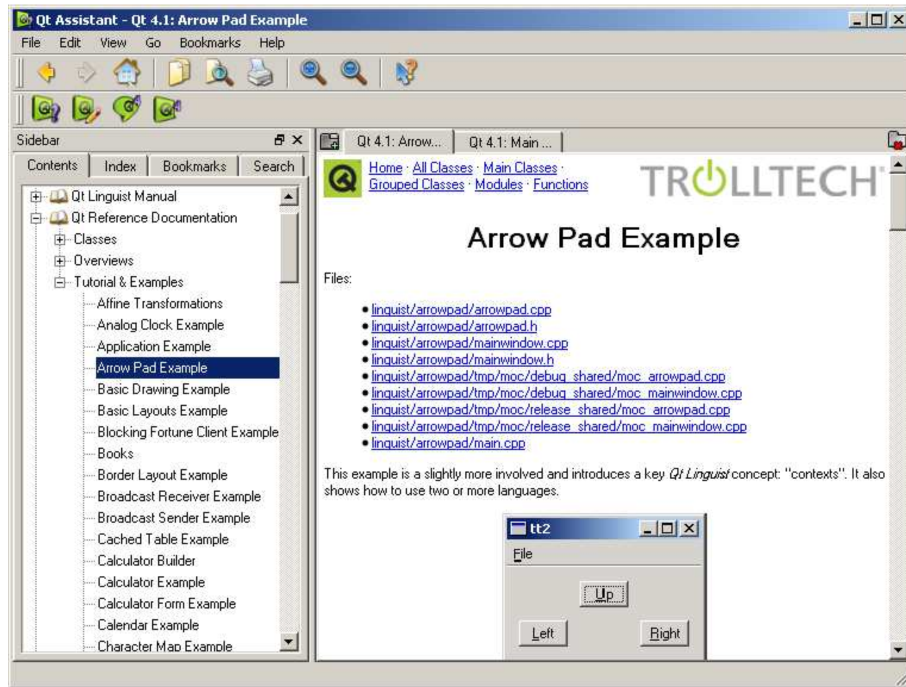


Figure 7.2: Qt Assistant.

7.3.1 GUI Features

The MARHES GUI gives the user control over each robot in the platform, as well as access to the sensorial information. The GUI interacts with each robot and allows the user to conform teams for cooperative behaviors.

The MARHES GUI offers the following features:

Robot Display

The GUI main window displays a map of the environment previously loaded by the user. The interface shows in real time the location of each of the robots on a map. Each robot is depicted as a white circle with a red mark aligned with the robots' heading. Obstacle proximity is also shown as a green bubble surrounding the robot whose size is correlated to IR range data.

The location of a robot representation in the map is linked to the robot positioning system. As the robot moves in the real world, the interface will show a moving icon in the map.

Local Control

The GUI can command each of the robots separately. It enables the user to command the linear and angular speeds of the robot using Player's control commands. The GUI has visible emergency stop buttons, in case the user needs to halt the robot immediately.

Single Robot or Team Operation

Each robot can be commanded to go to a particular location. This operation is based on the potential field controller explained in previous chapters. The GUI sends locations (coordinates) of targets to the robot. Robots use the potential field controller (PFC) to reach it. Many targets can be fed to the GUI to define a route. In this case each target point is called a waypoint of the robot's trajectory.

Each time a robot is commanded to go to a location, obstacle avoidance functions in the background in order to avoid obstacles in the path between the robot and the waypoint.

The user can also create a team of connected robots and command a mission for them. Teams can move in a leader-following or flocking behavior to a desired location or through a route. In team operation, the first robot of the list is always leader. If the user selects leader following operation, robots will move in a convoy like formation behind the leader, while the leader follows the waypoints. If the user commands a flocking behavior, the leader will move across the waypoints while all the other robots flock around it. Their linear speed will be the same as the leader's, and their heading is commanded as the average heading of the flock. The leader follower and flocking techniques were described in some detail in chapter 6.

Robot Info

The GUI displays information of each individual robot through a query button. The main screen shows each robot location as well as an obstacle free bubble around the robot, generated based on IR or sonar information. In the main window, the GUI

has separate numeric fields showing the actual position, orientation and velocities of the selected robots.

Remote Video

The GUI offers live video from any of the robots equipped with a camera. Additionally, users can pan the camera to direct focus to features not directly in front of the robots. The video transfer makes heavy use of the network and users may choose to stop the video feed at any time using the stop video button. The camera panning mechanism relies on Players' *ptz* interface.

Route Following

The user can command robots to follow a series of waypoints on the map. Robots can be commanded to follow waypoints individually or in teams. When a team of robots is commanded, the user can select two different group behaviors: (i) flocking or (ii) leader-follower. Flocking commands robots to flock together while the leader executes route following on the selected waypoints. Leader-following commands the robots to execute a convoy formation behind a leader, who follows the route.

Obstacle avoidance is always working in each robot to ensure collision free operation.

7.3.2 Interface Description

Figure 7.3 shows a snapshot of the GUI in operation. Each of the buttons and display boxes are numbered and will be explained in detail in this section. The numbers in parentheses refer to the numbers in Figure 7.3:

1. Selection List

This drop down menu contains the name of every robot in the platform. Users may select from this list the robot that they wish to exert action upon. If a user wants to

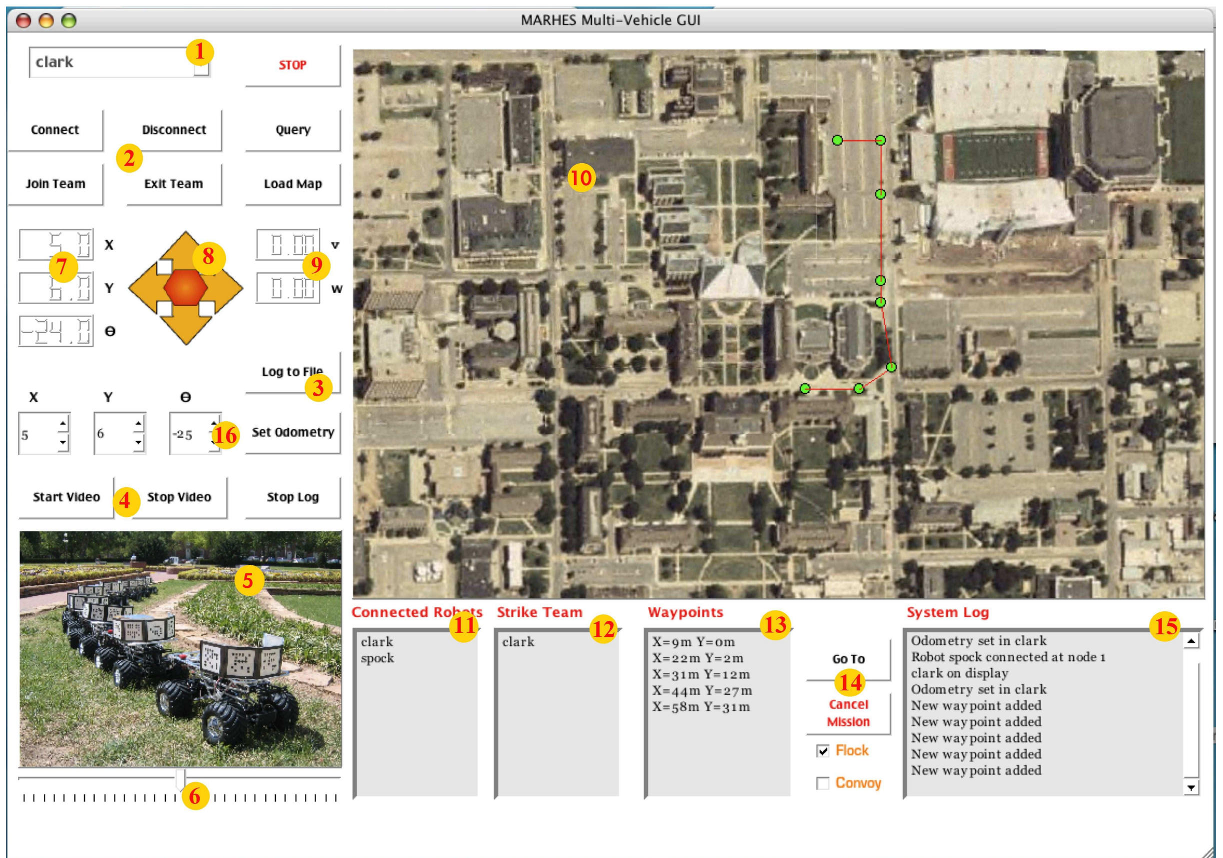


Figure 7.3: MARHES multi-vehicle GUI description.

display video, for example, before pressing the **start video** button, the name of the robot where video will be requested from must be selected in this box.

2. Action Buttons

These buttons operate on the robot selected in (1). **Connect**, connects to a Player server. **Disconnect** logs off the Player server. **Query** will show robot position and speeds in the numeric displays (7) and (9). **Join team** will insert the robot in the strike team list. **Exit team** will remove it from this list. **Load map**, displays in the map display box an image previously stored as map.jpg.

3. Log Actions

These buttons enable data logging to a file. Position, heading and vehicle speed data will be written to the **data.log** file at a every 100 ms. Data from all robots connected to the GUI will be logged. **Stop log** will stop the data storing and close the file.

4. Video Actions

These buttons operate on the robot selected in (1). **Start video** will show live video in the video box (5). Users can choose to stop the video feed at any moment by pressing the **stop video** button.

5 Video Box

The video box, displays video from a camera in the robot selected in (1).

6. Pan Slider

The position of this slider is linked trough the *ptz* interface to the camera panning mechanism in the robot. By moving this slider, users can pan the camera in the robot selected in (1).

7. Position Display

These three boxes display the position (x, y, θ) or the robot selected in (1). The display units for x and y are meters while degrees for the heading θ.

8. Command Module

Users can command linear and angular speeds to the robot selected in (1) by using these arrows. A click on the **up/down** arrows will increase or decrease the linear speed, while a click in the **left/right** arrows will increase or decrease the angular speed. A click on the center of the module stops the robot.

9. Speed Display

Users can monitor the linear and angular speeds of the robot selected in (1). This information is provided by the *position* interface and shows the actual angular and linear speeds of the robot.

10. Map Box

This is the main window of the GUI. The background corresponds to a user selectable JPG file that contains a map of the area where the robots are operating. Robots in the map are shown according to their location in the real world. Each robot in the map box is represented as a white circle. The front of the robot is shown as a red dashed area. Each IR equipped robot will show a green bubble around its body that symbolizes an obstacle free zone. The radius of the bubble is proportional to range data from the *ir* interface. A larger bubble around a robot signifies no obstacles surround it.

Users can click on the map to select a waypoint for robots to follow. If more than one point is selected, the GUI will show a route connecting the waypoints. The GUI only accepts 10 waypoints at this time, but this number can be expanded easily.

11. Connected Robots List

This list shows the robots that are connected to the GUI.

12. Strike Team List

This list shows the robots in the team that can be commanded on waypoint following missions.

13. Waypoint List

This list shows the series of waypoints accepted by the GUI.

14. Mission Control Buttons

Users can start a *go-to* operation, that will send the robots to the previously defined waypoints. The *go-to* button commands the robots on the **strike team** list only. The first robot in the list will be the leader. Users can cancel the operation at any time by pressing **cancel**. Canceling the operation will also erase the waypoints. Also, multiple

robot missions can be carried out in two modes: (i) flocking, and (ii) leader-following. Other algorithms can be implemented as well.

15. System Log

All the actions a user performs on the GUI are logged to this text box.

16. Robot Set Position

Each robot's odometry starts at zero when the robot is first turned on. That is, no matter what the real location of the robot is, the system will report it at $(0, 0, 0)$ for x, y and θ. Since the display on the map depends on the position reported by the robot, each robot's odometry must be reset to a value relative to the same reference frame, the *maps' reference frame.*

Once the robot is located in the real world, the user can use the *odometry* text boxes to localize the robot in any location on the map reference frame. The units for (x, y) and θ are meters and degrees, respectively.

7.3.3 Robot Display Description

The MARHES GUI contains a map box that displays a previously loaded image of a map or area where the robots work. This display box allows the user to know the position of each of the robots. The display box uses Player's *position* interface to update in real time the location of the robots and show an interactive map. Each pixel on the map box display is equivalent to 10 cm of travel of the robot.

Figure 7.4 shows a section of a map in the MARHES GUI. The map is a JPG image as shown in (1). Each robot is represented as a white circle as shown in (3). It's location on the screen is relative to the origin shown by the yellow cross-hairs in 7.4. To help the user identify the robot's heading, a red shaded pie within the body represents the robots front end. All the robots used by the platform are non-holonomic (i.e., they cannot move sidewise)

If the robot is equipped with range sensors such as Infra Red or Ultrasound devices,

Figure 7.4: MARHES GUI map display box.

the interface will show a green bubble around the robot (2) that represents an obstacle free area. The radius of the obstacle free bubble is calculated as an average of the range data of all range sensors in the robot. As the robot gets closer to obstacles the radius of the bubble decreases.

Waypoints in the map box display are represented as green dots interconnected with red lines (4). A set of two or more waypoints defines a route. When a *go-to* is called, users can see the progress of the operation in real time while the interface updates the position of the robots. As the mission progresses robots move across the waypoints, assuming an obstacle free environment. In case obstacles are detected, robots will navigate as close to obstacles as the environment allows.

7.3.4 Centralized and Decentralized Operation

The GUI can be used in both a centralized or decentralized control fashion. In centralized control, all the control algorithms for each robot such as obstacle avoidance, potential fields, or waypoint navigation, run in the machine where the GUI runs. This requires a powerful machine running the GUI since an instance of the navigation algorithms will be needed for each different robot. Centralized operation uses the Linux multithreading environment that exploits multiple processor systems.

The GUI, however, was designed to monitor the state of the robots and coordinate

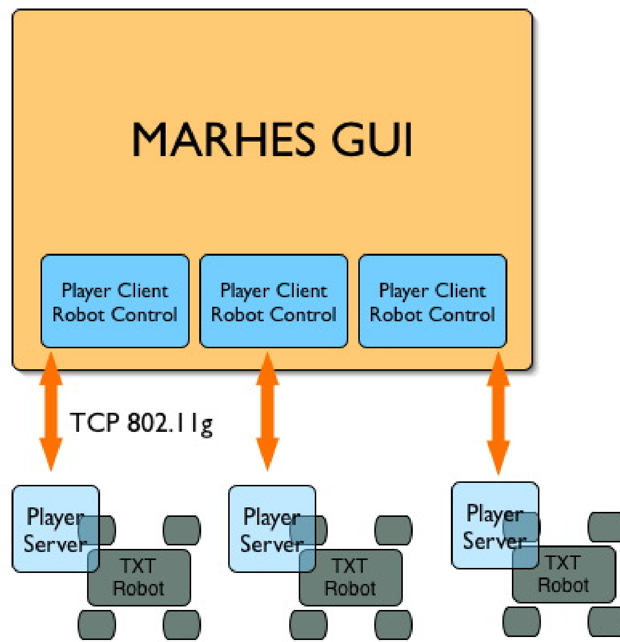
them. In decentralized operation, the GUI is used as a display and control terminal. Robot execute a small monitor program that receives GUI commands and controls the robot locally. Decentralized operation allows the GUI to run on small handheld machines or laptops, increasing system portability.

In decentralized control, the GUI uses a light network packet to transfer waypoints to the robot. Once waypoints are sent the robot is commanded via the network to start waypoint following. However, navigation and obstacle avoidance are executed locally in the robot's on-board computer. This approach reduces network traffic and computational overhead in the machine running the GUI. The Player sensor server is always running in the robot, hence, remote clients such as the GUI can connect to the robot and query sensor status.

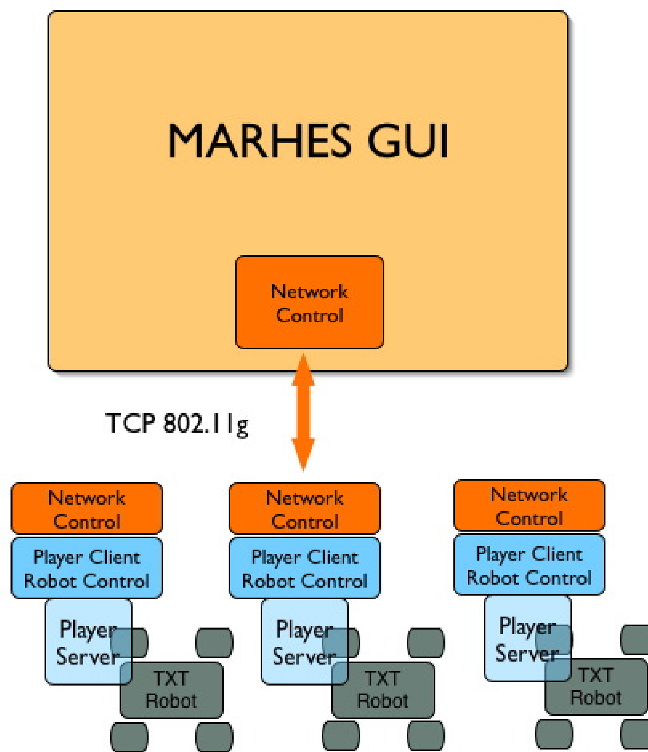
Figure 7.5 shows a diagram of both operation modes. In centralized operation the GUI uses individual Player client threads to control each robot.

In decentralized operation, each robot runs a local Player client. A monitor program, controls the robot's operation sending small commands via the network. Network commands are used to communicate data such as waypoints and mode of operation to the robot. This monitor program runs on a separate thread on the robot, and is in charge of starting and stopping the local Player client. The Player client is in charge of running control algorithms such as obstacle avoidance and potential field controllers.

The GUI makes using the multi-vehicle platform simpler. It also enables users to rapidly implement applications. Chapter 8 presents experimental results, and an example of an application implemented with the GUI. The experimental results presented validate the software and hardware architecture developed in this work.



(a) Centralized



(b) Decentralized

Figure 7.5: MARHES GUI modes of operation.

Chapter 8

Experimental Results

8.1 Introduction

This section presents comprehensive experimental results performed with the platform. The chapter is divided into two sections. In the first section, the testbeds' position estimation subsystem is evaluated. Results for encoder-derived, gyro-enhanced and GPS corrected navigation are shown. The second section presents an implementation of an outdoor surveillance and target assessment application.

8.2 Positioning System

The techniques described in the previous sections rely, on the most part, on positioning estimation information. Consequently, position accuracy becomes of utmost importance.

This section shows experimental results for the three positioning techniques developed for the MARHES platform: (i) *Odometry* which uses encoder information exclusively to produce a position estimate; (ii) *Gyro-enhanced odometry*, that uses an on-board Inertial Measurement Unit (IMU) to improve the heading estimate and the positioning system output accuracy overall; and (iii) *GPS corrected odometry*, that utilizes a Global Positioning System (GPS) receiver to bound the errors associated with the previously described dead reckoning techniques.

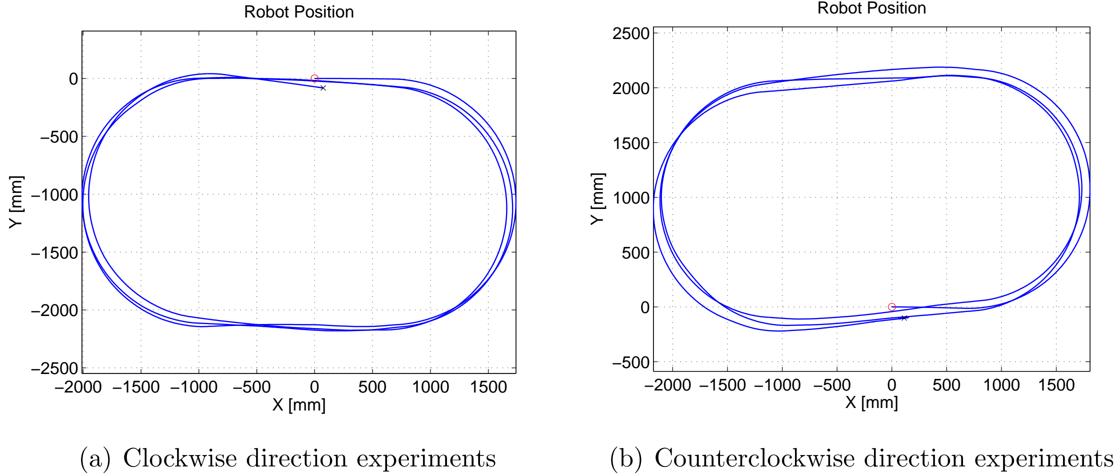
8.2.1 Odometry-derived Position Estimation

Odometry is the most basic and poor performing position estimation technique. After installing encoders in the wheels, it was necessary to calibrate the system to know what was the distance traveled per encoder tick. Using the resolution of the encoders (400 counts per revolution) and the wheel diameter it is possible to determine such constant. However, due to the fact that wheels are soft and compressible, this parameter may vary. Factors such as load weight and terrain roughness that also affect this constant were disregarded. Experiments were performed with the robot to get a good estimate of the constant's value. Several runs were performed where the robot traveled a fixed distance of 10 m. After the results were averaged, encoders were characterized with a distance factor of 0.3 mm/tick.

In order to quantify the goodness of the positioning system using encoders only or *odometry*, the robot was driven on an oval path performing three consecutive turns. At the end of the experiment, the robot was returned precisely to its original position $[0, 0, 0]$. The discrepancy of this measure and what is estimated by the positioning system is known as the *positioning system return error.* A total of 10 experiments (see Figure8.1) were performed in the clockwise and counterclockwise direction and the average results are presented. Performing experiments in both clockwise and counterclockwise direction is extremely important for position estimation systems, if experiments are performed in only one direction, dominant sources of systematic errors may cancel one another and great accuracy is seemingly achieved. Running experiments in both directions guarantees that such sources of errors will not cancel [29, 30].

Figure 8.4 shows the errors. The *percentile average of absolute errors* for $n = 10$ experiments is 9.6% in the clockwise direction and 10.2% in the counterclockwise direction. The *average absolute positioning error* X_e and Y_e is calculated using:

$$X_e = \frac{1}{n}\sum_{i=1}^{n} |x_i| \qquad Y_e = \frac{1}{n}\sum_{i=1}^{n} |y_i| \tag{8.1}$$



(a) Clockwise direction experiments (b) Counterclockwise direction experiments

Figure 8.1: Odometry. Clockwise and counterclockwise turns.

Where $n = 10$ is the number of experiments per run, x_i is the return error in the X axis, and y_i is the return error in the Y axis. The *absolute error* is calculated using:

$$E = \sqrt{{X_e}^2 + {Y_e}^2} \tag{8.2}$$

And expressed as a percentile of the traveled distance d using:

$$E[\%] = 100\frac{E}{d} \tag{8.3}$$

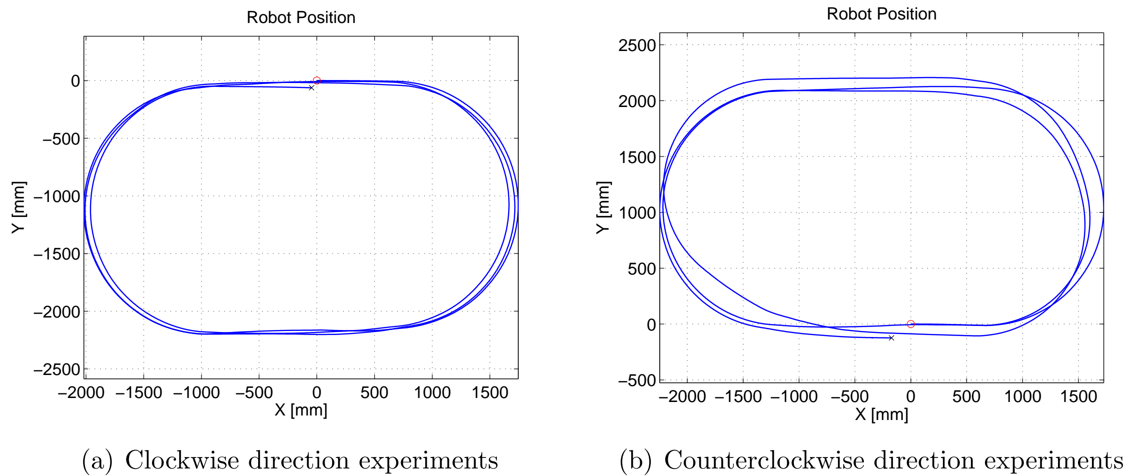
8.2.2 Gyro-Enhanced Odometry

Encoder derived odometry is a cheap and easy way to get position estimation. However, the results may not be very accurate. Part of the problem comes from the calculation of the vehicle's heading. An error in heading will have a great impact on the accuracy of the position estimation system. When using encoders as the sole source for position estimation, robot's heading is calculated by integrating the angular speed. Angular speed is not particularly accurate in an encoder-derived odometry system.

In *odometry* only, angular speed is calculated as the difference in encoder ticks of opposing side wheels. Encoders information is affected by slippage, thus if only one of the wheels slips, the system will generate a false reading. Additionally, since

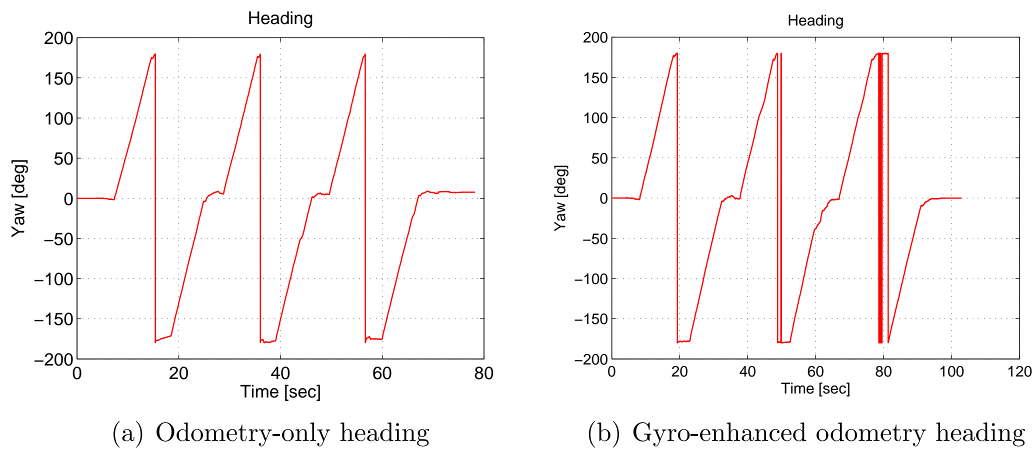
each encoder tick subtraction must be done in a brief sampling interval, the resolution of the angular speed calculated is usually low even when using high resolution encoders. *Gyro-enhanced odometry* alleviates heading related problems by using an inertial sensor to measure turning rates.

To characterize the quality of *gyro-enhanced odometry*, a new set of experiments having an on-board Inertial Measurement Unit (IMU) was performed. Figure 8.2 shows two experiments in clockwise and counterclockwise direction, with a robot equipped with an IMU. Figure 8.3 shows a comparison of the robot heading in an experiment performed using *odometry* only, and another using *gyro-enhanced odometry*. Notice how the final heading error is reduced by the inertial sensor addition.



(a) Clockwise direction experiments (b) Counterclockwise direction experiments

Figure 8.2: Gyro-enhanced odometry. Clockwise and counterclockwise turns.



(a) Odometry-only heading (b) Gyro-enhanced odometry heading

Figure 8.3: Heading of the robot during different experiments.

Figure 8.4 shows *Odometry* and *Gyro-Enhanced Odometry* positioning return er-

rors in the same chart for comparison purposes. The *percentile average return error* for *Gyro-Enhanced Odometry* in the clockwise direction was calculated to be 6.2% and 5.8% in the counterclockwise direction.

Gyro-Enhanced Odometry reduces by almost a factor of two, the errors of the position estimation system. Note how in Figure 8.4 the addition of the inertial sensor concentrates and reduces the return position errors in both the clockwise and counterclockwise direction.

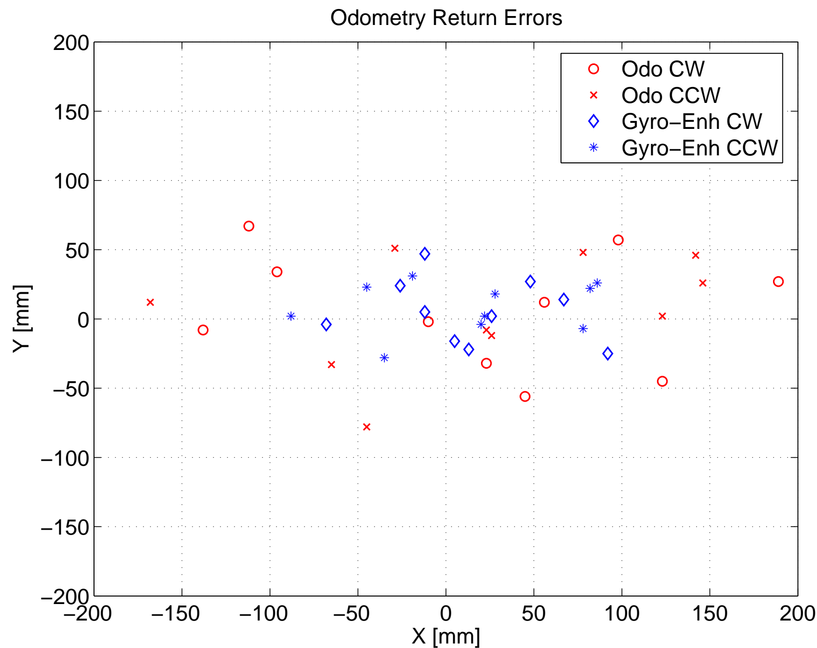


Figure 8.4: Positioning return errors for Odometry and Gyro-enhanced odometry.

Table 8.1 presents the absolute error associated with each mode of position estimation.

Table 8.1: Positioning System Error

Odometry	**CW** $\leq 9.6\%$	**CCW** $\leq 10.2\%$
Gyro-Enhanced Odometry	**CW** $\leq 6.2\%$	**CCW** $\leq 5.8\%$

8.2.3 GPS Corrected Odometry

When available, the system will use Global Positioning System (GPS) information to correct the position of the vehicle. The platform uses a 5 Hz Garmin GPS receiver

to acquire satellite information, and output global position fixes. The system uses a simple algorithm to merge GPS and odometry information. GPS is accepted as ground truth, thus the maximum difference between GPS positioning and odometry data is limited to the resolution of the GPS receiver. Each GPS reading is compared to the current odometry output. The positioning system output will be reset to the GPS fix, if the difference between odometry and the GPS fix is greater than the GPS resolution.

The addition of a GPS receiver to the system bounds the error of the positioning system to the error of the GPS system. The receiver used with the platform specifies a position error of less than 5 m which becomes the upper bound error of the platform's position estimation system. This simple filter combines the high resolution output of the odometry system and the error bound of the GPS.

Figure 8.5 shows the trajectory of a robot following a series of waypoints in an square path. The robot was equipped with an on-board GPS and global fixes are shown too.

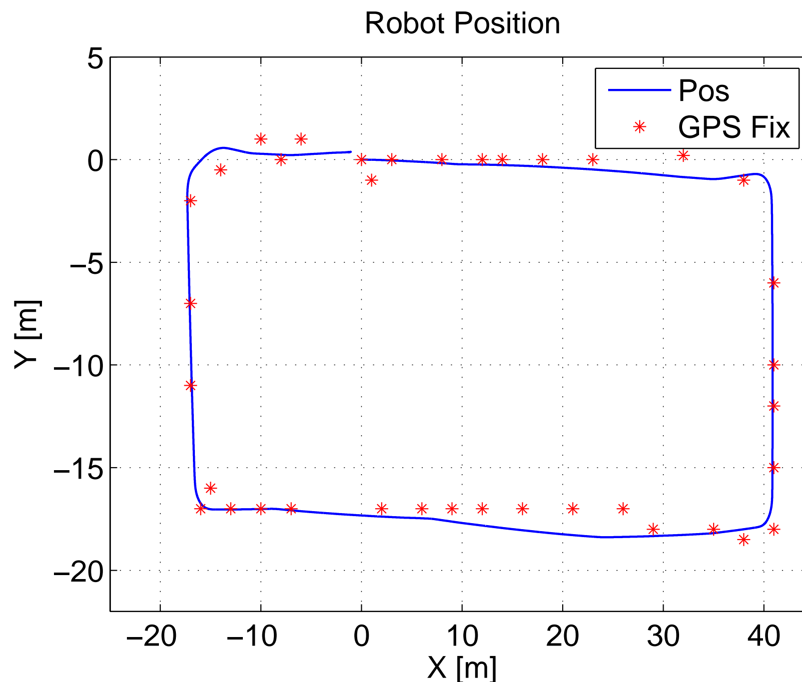


Figure 8.5: Square path experiment performed with a GPS equipped robot.

8.3 Target Assessment Implementation

To demonstrate the usability of the platform in a real world environment, a surveillance and target assessment application was implemented. This application makes use of the features described in the Graphical User Interface (GUI) section shown in previous chapters.

Figure 8.6 shows a picture describing the site where the application was implemented. The objective of the target assessment implementation is to safely drive a convoy of vehicles from a remote location to a known target. The robots will be equipped with an on-board camera that will allow the user to assess the condition of the target.

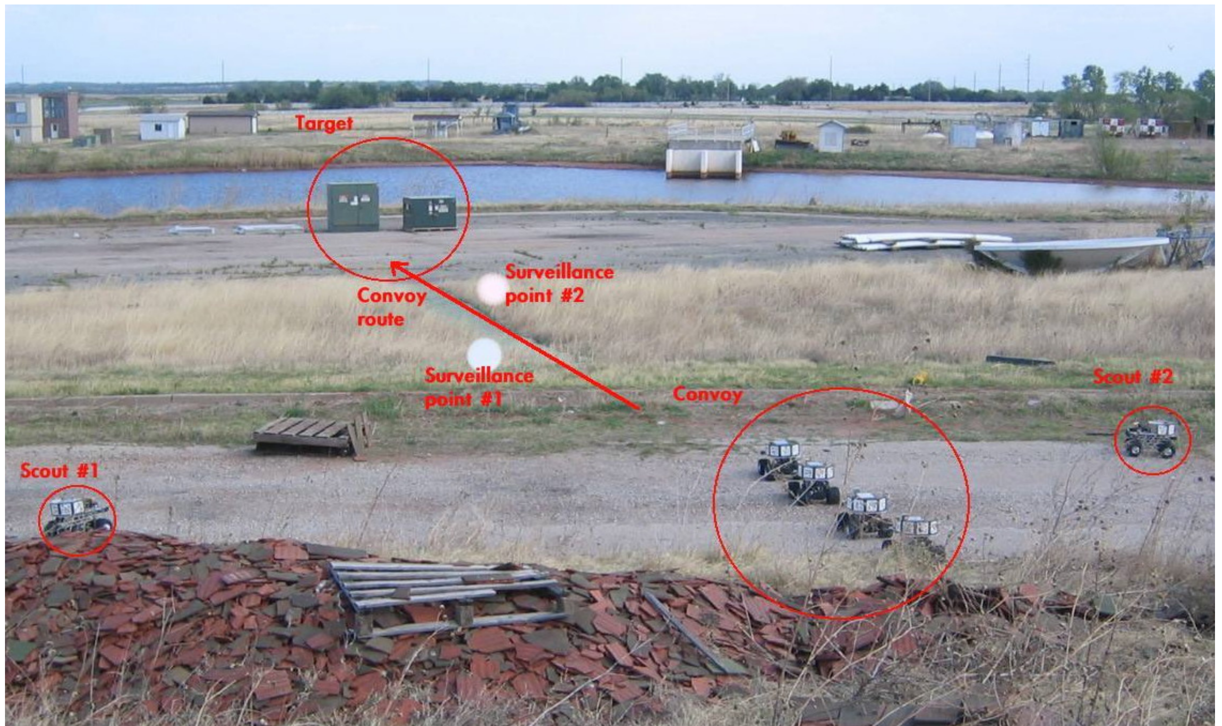


Figure 8.6: Target assessment application site.

Three phases comprise the mission:

- Secure Convoy Path,
- Drive Convoy to Target,
- Spread Convoy Around Target.

The reminder of this chapter gives a detailed description of each phase.

8.3.1 Secure Convoy Path

Before the convoy of robots is sent to the target, two other robots will explore the area to guarantee that is safe for other robots to travel to the target. Two strategic points were selected from where it is possible to examine the condition of the preselected path to the target. Such points are denominated *surveillance points*. The robots sent to the *surveillance points* are called *scouts*. The robots are equipped with on-board video cameras to allow the user to assess the condition of the environment before other robots are deployed. Figure 8.6 shows the location of the *surveillance points* and the *scouts* at their starting position.

Each of the *scout* robots is commanded to the surveillance point using a series of waypoints. Once the surveillance point is reached, the *scout* robot pans the camera to provide a panoramic view to the user via the GUI. The user in turn can verify if the path is safe. The Marhes GUI is used to control the surveillance operation, as depicted in Figure 8.7.

Figure 8.8 shows the trajectory of the first *scout* robot and its associated waypoints. The first vehicle encounters an obstacle in the path, avoids it successfully, and positions itself at the first surveillance point. The figure also shows the commanded and actual linear and angular speeds. A state diagram that shows when the system switches from goal seeking to obstacle avoidance is also given in Figure 8.7(b).

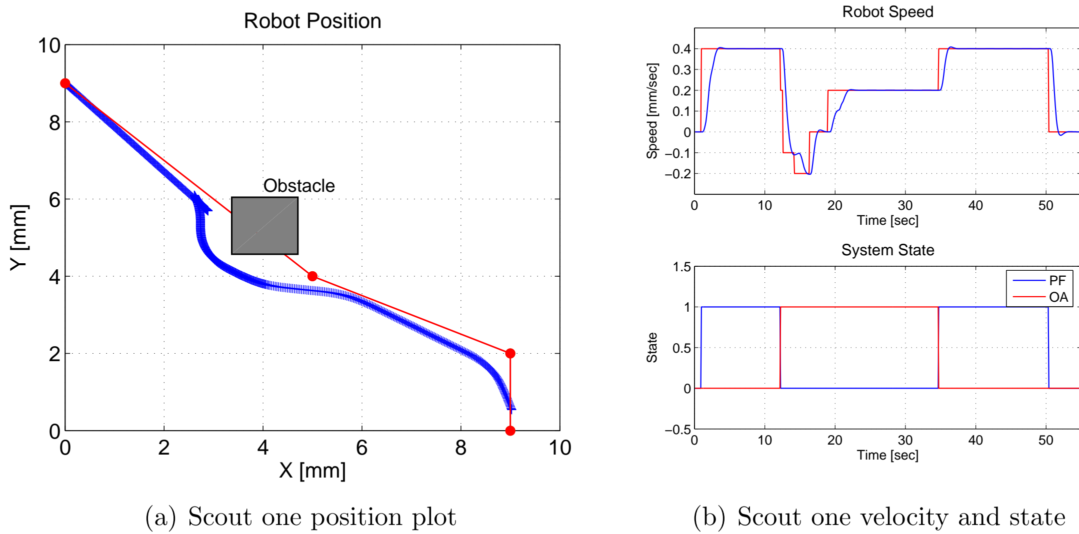
The second *scout* vehicle drives from a distant location farther into the preselected convoy path. As the first scout, its function is to monitor the area and provide surveillance video feedback. Once the robot reaches the surveillance point, it pans the camera around providing video feedback through the GUI.

8.3.2 Drive Convoy to Target

Once the path is considered to be safe, the convoy is commanded to the target location. The convoy is composed of one leader robot and three followers. The first robot

(a) Scout one (b) GUI control

Figure 8.7: Scout one in operation.



(a) Scout one position plot (b) Scout one velocity and state

Figure 8.8: Position, velocities, and state plot of scout one.

in the convoy is driven to the target through a series of waypoints. The other robots follow the leader executing a communication-based leader-follower operation. While the leader travels autonomously through the preassigned waypoints, it communicates its location to the followers. Followers attempt to maintain a safe distance from the leader. This distance is set to be 0.8 m.

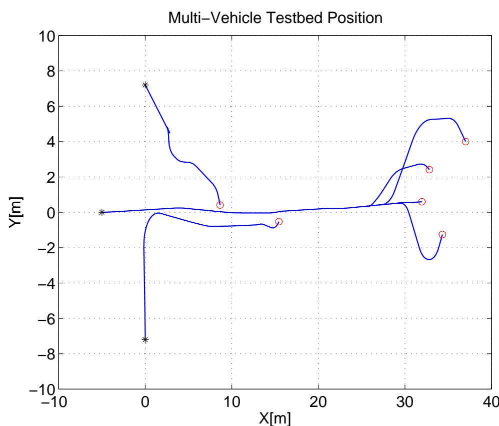
Figure 8.9 shows the convoy of robots traversing the path in formation.

The success of the communication-based leader-follower operation depends on the accuracy of each robot's position estimation system. Future algorithms will include a video based navigation aid that will use external references to improve the technique.

Figure 8.9: Convoy of robots traveling to the target.

8.3.3 Spread Convoy Around Target

Once the convoy has reached the vicinity of the target each of the robots is sent to a different location closer to the target. The operation is based on a series of preselected waypoints that guide each vehicle to a particular location near the target. When each of the robot reaches a pre-established waypoint near the target, the system switches from leader-following to waypoint tracking to travel closer to the target. The objective of this rearrangement is to provide the user with a diverse view of the target. The user can now, via remote video, assess the target condition.



(a) Trajectory of team of robots

(b) Target assessment pictures

Figure 8.10: Target assessment position plots and video feedback.

Figure 8.10(a) shows the final position of the team close to the target. It also

shows images from the on-board camera on each robot. The user target assessment is based on those images.

The multi-vehicle platform has proven to be a reliable and flexible tool. With this example is successfully demonstrated how the GUI can be used to develop cooperative robotics applications. As more sensors are added to the CAN network

Chapter 9

Conclusions and Future Work

This work presents the development of a comprehensive robotics system for the study of cooperative mobile control systems and sensor networks. The lower level features increased flexibility to new configurations with a pool of hot-swappable sensors that can be added or removed *on-the-fly*. A modular framework for integrating various sensing and control capabilities has been implemented and tested.

The platform is compatible with Player and Gazebo which makes it remotely controllable and accessible, and offers the benefit of a simulation environment with full sensor support. The software library gives the developer a basic set of building blocks for more complex algorithms.

The testbed described here, presents a low-cost, highly versatile, flexible, and expandable alternative to high price robotic systems commercially available. Many other research laboratories around the world chose to develop their own systems as well, but most of them function only indoors, lack scalability, or a simulation environment. The cooperative robotics testbed has become a tool of utmost importance in the MARHES laboratory. New controllers and algorithms are implemented and tested in a realistic environment using only a fraction of the time after debugging in the simulation environment provided.

One of the most important features of the platform is its applicability to different technology fields. At present time the platform is used to develop vision based formation control algorithms. Current plans envision the addition of environmental

sensors for the study of wireless sensor networks. The testbed can adapt to changing research trends, making it a never outdated tool.

The experimental testbed is also a comprehensive educational tool. While working with the testbed, students learn hardware and software design for embedded computer systems and micro-controlled systems. They become familiar with cutting edge computer controlled design and manufacturing technology. Additionally students and researchers have the opportunity to study computer vision, hybrid, classic and networked control and cooperative sensor nets applications.

The testbed's scalability makes it ideal to train new students in the lab. They develop hardware and software skills, while upgrading the platform. Currently, a newly arrived student develops a battery monitoring subsystem for the robots. With the testbed, students have the unique opportunity to see their work evolve from a simulated environment into reality, which makes the experimental testbed an enjoyable tool.

Future plans include the addition of wireless environment sensors (MOTES) [31] for the study of sensor networks, battery monitoring, and laser range finders for navigation. The platform software may also be run in different operating systems such as Apple's OSX. New robots may use Apple's mac mini which is ideally sized for embedded applications.

Currently the MARHES lab owns a small blimp that has already performed autonomous indoor flights. In the future, the testbed will have land and air vehicles working together. The addition of an aerial robot will greatly benefit the system as it would provide overhead images that could be used for target assignment or air-ground localization.

Lastly, a behavior composer would help researchers develop new applications based on previously tested behaviors. Such software packages are already being used in the MARHES lab for proyects such as *robotic games*, and are considered as a future feature for the testbed.

Bibliography

[1] B. Gerkey, R. T. Vaughn, and A. Howard, "The Player/Stage project: Tools for multi-robot and distributed sensor systems," in *Proc. of the 11th Int. Conf. on Advanced Robotics*, Coimbra, Portugal, June 2003, pp. 317–323.

[2] S. S. Ge and F. L. Lewis, *Autonomous Mobile Robots.* Boca Raton, Florida USA: Taylor and Francis Group, 2006.

[3] Z. Jin, S. Waydo, E. B. Wildanger, M. Lammers, H. Scholze, P. Foley, D. Held, and R. M. Murray, "Mvwt-ii: The second generation caltech multi-vehicle wireless testbed," in *American Control Conference (ACC)*, vol. 6, Boston, MA, June 2004, pp. 5321–5326.

[4] V. Vladimerou, A. Stubbs, J. Rubel, A. Fulford, and G. Dullerud, "A hovercraft testbed for decentralized and cooperative control," in *American Control Conference (ACC)*, vol. 6, Boston, MA, June 2004, pp. 5333–5337.

[5] R. DAndrea and M. Babish, "The roboflag testbed," in *American Control Conference (ACC)*, vol. 1, Boston, MA, June 2004, pp. 656–660.

[6] E. King, Y. Kuwata, M. Alighanbari, L. Bertuccelli, and J. How, "Coordination and control experiments on a multi-vehicle testbed," in *American Control Conference (ACC)*, vol. 6, Boston, MA, June 2004, pp. 5315–5320.

[7] T. W. McLain and R. W. Beard, "Unmanned air vehicle testbed for cooperative control experiments," in *American Control Conference (ACC)*, vol. 6, Boston, MA, June 2004, pp. 5327–5331.

[8] L. Chaimowicz, B. Grocholsky, J. F. Keller, V. Kumar, and C. J. Taylor, "Experiments in multirobot air-ground coordination," in *Proc. IEEE Int. Conf. Robot. Automat.*, vol. 4, New Orleans, LA, April 2004, pp. 4053–4058.

[9] K. Etschberger, *Controller Area Network: Basics, Protocols, Chips and Applications.* Weingarten, Germany: IXXAT Press, 2001.

[10] D. J. Bruemmer, D. D. Dudenhoeffer, M. D. McKay, and M. O. Anderson, "A robotic swarm for spill finding and perimeter formation," in *Spectrum 2002*, Reno, Nevada USA, August 2002.

[11] Intel Corporation, "POSIT, open source computer vision library reference manual," December 2000, pp. 6.9 – 6.15.

[12] D. DeMenthon and L. Davis, "Model-based object pose in 25 lines of code," in *International Journal of Computer Vision*, vol. 15, june 1995, pp. 123–141.

[13] M. Fiala, "Vision guided control of multiple robots," in *Canadian Conference on Computer and Robot Vision*, vol. 4, May 2004, pp. 241–246.

[14] B. Bayazit, "Potential field methods," Washington University in St. Louis, St. Louis, Missouri USA, Tech. Rep., 2003, cs 522A course notes.

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

[16] J. C. Latombe, *Robot Motion Planning.* Boston, Massachusetts USA: Kluwer Academic Publishers, 1991.

[17] D. E. Chang, S. C. Shadden, J. E. Marsden, and R. Olfati-Saber, "Collision avoidance for multiple agent systems," in *Proc. IEEE Conf. on Decision and Control*, Maui, Hawaii USA, December 2003, pp. 539–543.

[18] J. Borenstein and U. Raschke, "Real-time obstacle avoidance for non-point mobile robots," in *SME Transactions on Robotics Research*, vol. 2, September 1992, pp. 2.1–2.10.

[19] R. Fierro, P. Song, A. Das, and V. Kumar, "Cooperative control of robot formations," in *Cooperative Control and Optimization*, R. Murphey and P. Pardolos, Eds. Kluwer Academic Press, 2002, vol. 66, ch. 5, pp. 73–93.

[20] A. K. Das, R. Fierro, V. Kumar, J. P. Ostrowski, J. Spletzer, and C. J. Taylor, "A vision-based formation control framework," *IEEE Trans. on Robotics and Automation*, vol. 18, no. 5, pp. 813–825, October 2002.

[21] R. Fierro and A. K. Das, "A modular architecture for formation control," in *IEEE 3rd Int. Workshop on Robot Motion and Control*, Bukowy Dworek, Poland, November 9-11 2002, pp. 285–290.

[22] R. Fierro, A. Das, V. Kumar, and J. P. Ostrowski, "Hybrid control of formations of robots," in *Proc. IEEE Int. Conf. Robot. Automat.*, Seoul, Korea, May 2001, pp. 157–162.

[23] A. Das, R. Fierro, V. Kumar, J. Southall, J. Spletzer, and C. Taylor, "Real-time vision-based control of a nonholonomic mobile robot," in *Proc. IEEE Int. Conf. Robot. Automat.*, Seoul, Korea, May 2001, pp. 1714–1719.

[24] J. Clark and R. Fierro, "Cooperative hybrid control of robotic sensors for perimeter detection and tracking," in *Proc. American Control Conf.*, Portland, Oregon USA, June 8-10 2005, (To appear).

[25] A. J. Herbert Tanner and G. J. Pappas, "Stable flocking of mobile agents, part ii: Dynamic topology," in *42th IEEE Conference on Decision and Control*, December 2003, pp. 2016–2021.

[26] A. Regmi, R. Sandoval, R. Byrne, H. Tanner, and C. Abdallah, "Experimental implementation of flocking algorithms in wheeled mobile robots," in *American Control Conference*, June 2005, pp. 4917 – 4922.

[27] T. Vicsek, A. Czirok, E. B. Jacob, I. Cohen, and O. Schochet, "Novel type of phase transitions in a system of self-driven particles," in *Physical Review Letters*, vol. 75, 1995, pp. 1226–1229.

[28] J. L. A. Jadbabaie and A. S. Morse, "Coordination of groups of mobile autonomous agents using nearest neighbor rules," in *IEEE Transactions on Automatic Control*, 2003, pp. 988–1001.

[29] J. Borenstein and L. Feng, "Umbmark: A benchmark test for measuring odometry errors in mobile robots," in *SPIE Conference on Mobile Robotics*, October 1995.

[30] L. Ojeda, D. Cruz, G. Reina, and J. Borenstein, "Current-based slippage detection and odometry correction for mobile robots and planetary rovers," in *IEEE Trans. on Robotics and Automation*, vol. 22, April 2006, pp. 366–378.

[31] Y. Choi, M. Gouda, M. Kim, and A. Arora, "The mote connectivity protocol," in *Proc. Int. Conf. on Computer Comms. and Networks*, October 2003, pp. 533–538.

Name: Daniel Cruz Date of Degree: degreedate

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: AN EXPERIMENTAL TESTBED FOR SWARMING AND COOPERATIVE ROBOTIC NETWORKS

Pages in Study: 99 Candidate for the Degree of Master of Science

Major Field: Electrical Engineering

This document describes an innovative cooperative robotics multi-vehicle testbed, featuring a flexible architecture that enables the system to be rapidly adapted to different applications. It also offers tools to reduce development and implementation time. The testbed consists of ten non-holonomic car-like robots networked together to share sensor information. Each vehicle features an on-board computer for local control, and a network of devices that can be suited with a variety of hot-swappable sensors depending on the application. The entire system is integrated with Player, an open source sensor server compatible with Gazebo, a 3D world simulator. Control algorithms can be evaluated in simulation mode and then ported to the real vehicle with virtually no code change. This thesis presents a flexible and complete system that serves the study of Cooperative Control, Hybrid and Embedded Systems, Sensor Networks, Networked Control and that can be used in an extensive range of applications.

ADVISOR'S APPROVAL: Dr. Rafael Fierro

VITA

Daniel Cruz

Candidate for the Degree of

Master of Science

Thesis: AN EXPERIMENTAL TESTBED FOR SWARMING AND COOPERATIVE ROBOTIC NETWORKS

Personal Data: Born in Bogota, Colombia, on January 20th, 1978, son of Graciela and Guillermo Cruz.

Education: Graduated from Colegio de Boyaca, Tunja, Colombia, in December 1993; received an Electronics Engineering degree from Pontificia Universidad Javeriana , Bogota, Colombia in November 2001. Completed the requirements for the Master of Science degree with a major in Electrical Engineering at Oklahoma State University in July, 2006.

Experience: Researcher at the Mobile Robotics Lab, University of Michigan, Ann Arbor, 2002 to 2004. Employed by Oklahoma State University, Department of Electrical Engineering as a graduate research assistant, 2004 to 2006.

Professional Memberships: Phi Kappa Phi Honor Society.