Control of Self-Organizing and Geometric Formations

Navigation and Formation Control Methods using the Velocity Potential Flow and the Reconfiguring Leader-Follower Controller

Elisha Pruner

A thesis presented for the degree of
Master of Applied Science in Engineering

uOttawa

Department of Graduate Studies in Mechanical Engineering
University of Ottawa
Canada

Abstract

Multi-vehicle systems offer many advantages in engineering applications such as increased efficiency and robustness. However, the disadvantage of multi-vehicle systems is that they require a high level of organization and coordination in order to successfully complete a task. Formation control is a field of engineering that addresses this issue, and provides coordination schemes to successfully implement multi-vehicle systems. Two approaches to group coordination were proposed in this work: geometric and self-organizing formations. A geometric reconfiguring formation was developed using the leader-follower method, and the self-organizing formation was developed using the velocity potential equations from fluid flow theory. Both formation controllers were first tested in simulation in MATLAB, and then implemented on the X80 mobile robot units. Various experiments were conducted to test the formations under difficult obstacle scenarios. The robots successfully navigated through the obstacles as a coordinated as a team using the self-organizing and geometric formation control approaches.

Dedication

This work is dedicated to my family and friends, without whom this work would not have been possible. I would like to specifically thank my parents for their ongoing support, and to Shaz, for his love and encouragement through this journey of mine.

Acknowledgements

I would like to express my gratitude and thanks to my supervisor Dr. Dan Necsulescu. We have worked together for many years now, and you have given me the time, support, patience, and guidance to let me flourish under your supervision. I would also like to thank Dr. Bumsoo Kim who facilitated the research work with Defense Research and Development Canada (DRDC), and provided the X80 mobile robots for our formation experiments. Finally, I would like to thank my colleagues in the research lab for their ideas, encouragement, and advice. I will always remember the long nights working on robot experiments, thank you Sharareh, Ehsan, and Mostafa.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

There is currently a strong motivation to implement autonomous robot systems in military, civilian, and commercial applications. Over the past decade, unmanned vehicle systems have become increasingly prevalent in the military settings; this is primarily due to their reconnaissance, and surveillance abilities. Enhanced intelligence through robotic systems allow military leaders to make better, more informed decisions, which in turn has saved countless lives on the battlefield. Furthermore, civilian applications are also greatly enhanced by autonomous robotic systems. Organizations such as police, emergency response, coast guard, fire response, search and rescue, border security, and arctic surveillance all implement unmanned robotic systems, and these unmanned vehicles allow first responders to stay a safe distance away from dangerous situations. Finally, autonomous robots will play a large role in emerging commercial applications in natural resources sectors such as mining, oil and gas, and agriculture; moreover, they will continue to increase efficiencies in manufacturing, warehousing, and distribution.

At present, there is a great interest in deploying multi-robot systems in military, civilian, and commercial applications. Multi-robot systems have many benefits over their single robot counterparts, since they can perform tasks with greater efficiency, less cost, and they present a more robust solution. Autonomy and group coordination are key traits for the successful navigation of robot teams in dynamic environments. Autonomous motion in multi-robot systems is a very complex problem, and scientists and engineers from different fields are working together to come up with solutions.

Most navigation solutions for autonomous mobile robots are designed for single robot systems. Multi-robot systems are more complex than their single robot counterparts, and require extra elements such as networked communication and group coordination strategies, to perform their tasks effectively. Teams of mobile robots must decide as a group what is the best path of motion when difficult obstacles configurations appear in the environment. There is still no perfect solution to the multi-robot coordination problem, and researchers are currently publishing new and innovative solutions. Solving the navigation and coordination problems for multi-robot systems could change the way many robot systems are used in military, civilian, and commercial applications. Multi-agent robotic systems are a billion dollar industry, and researchers and corporations are racing to develop the best technologies.

1.2 Research Objective

In order to increase the productivity of multi-robot systems, and to avoid inter-robot collisions during their mission, it is very important to develop a coordination strategy among members of the team. Coordination strategies for multi-robot systems define how a group of homogeneous robots travel through a terrain without collid-

ing into obstacles or other robots along their paths. In our research we will look at two strategies for coordination: self-organizing and geometric formations. We will also develop a navigation strategy to move the formation through the terrain to a goal position without colliding with static obstacles in the environment. We will be developing controllers to allow for the decentralized autonomous movement of the team. Controllers will use sensor information to update feedback control systems for each robot. A central controller will be responsible for overall group formation decisions such as formation shape and separation distance between the robots, while individual robots will make reactive decentralized decisions on the best movement that goes along with the team formation objectives. Controllers will first be tested on a group of simulated robots, and then the controllers will be applied in experimentation with a group of differential drive lab robots.

1.3 Method of Approach

Two types of formations will be examined in this work. Firstly, a geometric formation will be developed using a reconfiguring Leader-Follower controller. In this formation, the leader will navigate autonomously through the terrain, while the followers will orient themselves at predefined positions behind the leader. The leader-follower geometric formation will add order and organization to the group as they travel through the environment. Secondly, a self-organizing formation will be developed using the velocity potential flow method. In this method, the velocity potential model will regulate local interaction between the vehicles, and the vehicles will organize themselves in a self-organized formation, in order to avoid inter-robot collisions as the team travels towards the goal position.

1.4 Thesis Outline

This work consists of eleven chapters that go through the process of designing and executing formation controllers for a team of mobile robots.

Chapter 2 provides background information about autonomous mobile robots, motion planning, navigation, and formation control for multi-vehicle systems.

Chapter 3 provides a literature review that examines formation control strategies in literature, and an analysis of each method is carried out.

Chapter 4 describes the proposed navigation and formation control approaches, and derives the equations of motion for the differential drive vehicles.

Chapter 5 details the hardware components on the X80 mobile robot that is used in our experiments. This chapter examines the sensors, actuators, and processing that allows the X80 robot to travel autonomously through the terrain.

Chapter 6 describes the hardware-software interaction, and how high-level navigation controllers are sent to the X80 robot through a wireless network.

Chapter 7 describes the simulation and experimental design in MATLAB and Player/Stage. This chapter details the steps needed to execute formation controllers on simulated robots and on the X80 robots.

Chapter 8 is the first results chapter, and explains the simulation and experimental work for single robot navigation.

Chapter 9 is the second results chapter, and describes the simulation and experimental work in developing geometric formation controllers.

Chapter 10 is the final results chapter, and describes the simulation and experimental work in developing self-organizing formation controllers.

Finally, Chapter 11 provides a conclusion to this work, explains our research contributions, and discusses future work in this project.

Chapter 2

Background Information

2.1 Mobile Robots

Mobile robots are currently at the forefront of research and innovation. They are designed to search and explore large areas, to perform repetitive, unpleasant, or dangerous tasks, and are engineered to work in unstructured and dynamic environments. Mobile robots are used in tasks such as agriculture, container handling, intelligent transportation, inspection, scientific exploration, mining, waste management, search and rescue, fire-fighting, and military applications[4]. There is a great interest in the research and deployment of mobile robots, since they can reduce the cost of operations, and perform tasks with greater speed and efficiency than humans[17].

There are many types of mobile robots in use today, and their classification is based on their working environments. Mobile robots working on land are called Unmanned Ground Vehicles (UGV), mobile robots working underwater are called Autonomous Underwater Vehicles (AUV), and mobile robots working in the air are called Unmanned Aerial Vehicles (UAV). Within the three sub-genres, robots can be further classified based on their mechanisms for locomotion. For example,

wheels or legs generally power unmanned ground vehicles, and the design choices are made based on the needs of the operating environment. Legged mobile robots have the advantage of being more maneuverable and adaptable in rough terrain, and generally mimic legged configurations of animals in nature. Mobile robots with legs incur high degrees of freedom, they tend to be mechanically complex, and are expensive to build and manufacture. For the vast majority of applications, wheeled mobile robots are preferred since they are mechanically simpler, they can travel at high velocities, and suspension systems can be added to handle many types of terrain[53]. Furthermore, autonomous underwater vehicles can be further classified into submarines or biologically inspired vehicles that mimic the movement of sea creatures. Finally, unmanned aerial vehicles are further classified by their wing design and are either fixed wing or rotary wing. Fixed wing UAVs are used for applications that require traveling long distances, and rotary winged aircraft, like the quad-rotor aircraft, are implemented in applications that require a high level of maneuverability and dexterity.

Our research in mobile robotics will focus on unmanned ground vehicles. Specifically we will be working with wheeled mobile robots that travel on flat, two-dimensional terrain. Our robots will be equipped with two wheels using a differential drive steering model. Differential drive mobile robots offer a large range of mobility, due to their wheel setup. Both wheels are controlled independently from the other, and by coordinating the two different speeds, the robot can perform various maneuvers such as: spin in place, move in a straight line, move in a circular path, and follow prescribed trajectories. The mobile robot will also be characterized by nonholonomic kinematic constraints that prevent the wheeled robots from moving sideways, perpendicular to the wheel movement, during its motion. Nonholonomic constraints limit the mobility of mobile robots in their workspace, and since sideway motion is prohibited, robots must move around obstacles in a smooth path with

an appropriate velocity profile. The two-wheel differential drive mobile robots fare better than their four or six wheel counterparts in working with their nonholonomic motion constraints, since the differential drive robot can slow down and even stop and spin on place when difficult motions are required.

2.2 Autonomous Mobile Robot Systems

When designing mobile robot systems, engineers must decide on the level of autonomy for the vehicle, and how it will interact with the human operator. The decision on the level of autonomy depends on the scope of the work and the budget of the project; fully autonomous systems are more complex and expensive solutions, and require more time to develop. In general there are four levels of autonomy for unmanned vehicle systems[17].

The first level is remote control and teleoperation. In this level a human operator controls the robots from a distance, the human performs all the cognitive processes, onboard sensors enable the operator to visualize the location and movement of the platform within its environment, and actuators enable the human to act on the information.

The second level of autonomy is the semi-autonomous mobile robot. In this case, robots are programmed with advanced navigation, obstacle avoidance, and data fusion capabilities that minimize the need for operator interaction.

The third level of autonomy is platform-centric autonomous. Platform-centric autonomous vehicles are fully autonomous unmanned vehicle systems that can undertake complex missions, and can respond to additional commands from a controller without the need for further guidance from an operator.

Finally, the fourth level of autonomy is called network-centric autonomous. Vehicles in this level must be able to receive information from the communications

network and incorporate it in the mission execution, they must respond to appropriate information requests and action commands received from the network. They must also resolve conflicts when commands interfere with each other or interfere with the original mission assigned to the UGV.

2.3 Motion Planning

Some of the most significant challenges confronting autonomous mobile robots lie in the domain of automatic motion planning. The motion planner must find a path for the robot to travel from one configuration to another, while avoiding obstacles. Motion planning algorithms are derived from elements of mechanics, control theory, computational and differential geometry, and computer science. There is generally no single solution to solve motion-planning problems, and controllers are often combinations of various methods from different fields of engineering.

The Piano Movers Problem is a classic path-planning problem. Given a three-dimensional rigid body and a known set of obstacles, the problem is to find a collision-free path for the omnidirectional free-flying piano to move from a start configuration to a goal configuration[9]. The obstacles are assumed to be perfectly known, and execution of the planned path is exact. This method is called offline planning because the planning is finished in advance of the execution. The configuration space is the space of all configurations the robot can achieve. The dimension of the configuration space is equal to the degrees of freedom (DOF) of the system. A piano has six DOF x, y, z, roll, pitch, and yaw. The problem is to find a curve in the configuration space that connects the start and goal points and avoids all configuration space obstacles that arise due to obstacles in the space.

Motion planning consists of four main components: navigation, coverage, localization, and mapping[9]. Firstly, navigation is the problem of finding a collision-free

motion for the robot to travel from one configuration to another. Secondly, coverage is the problem of passing a sensor or tool over all points in the space, determining if all points are available to the robot. Thirdly, localization is the problem of determining the current configuration of the robot in the environment. Finally, mapping is the problem of exploring and sensing an unknown environment to construct a map. Recently it was found that localization and mapping can be combined into a single SLAM (Simultaneous Localization and Mapping) algorithm[53].

An effective motion planner depends heavily on the properties of the robot solving the task. For example, the robot and the environment determine the number of degrees of freedom of the system and the shape of the configuration space. Once we understand the robots configuration space, we can ask if the robot is free to move instantaneously in any direction in its configuration space. Omnidirectional mobile robots can move in any direction in the configuration space, however they are impractical in most robot applications, and are applied more in a simulation context to show the ideal movement of robots. Wheeled mobile robots with nonholonomic constraints are more prevalent in mobile robotics research, and since wheels cannot translate sideways, it is constrained in its movement in the configuration space. Wheeled mobile robots can be modeled using kinematic equations with velocities as controls, or using dynamic equations of motion with force as controls.

The basic strategy in developing motion planner to regulate the motion of mobile robots is: Sense $\rightarrow$ Map $\rightarrow$ Plan $\rightarrow$ Act[4]. Firstly, sensors gain information about the robot pose and the local environment. Secondly, mapping algorithms take in sensor information and develop a two-dimensional map of the terrain to determine the location of obstacles and walls. Thirdly, path-planning algorithms determine the best path, and velocity profile along the path, based on the map, sensor data, and the vehicles nonholonomic constraints. Finally the robot converts velocity data into torque values at the motors of the differential drive mobile robot.

2.3.1 Navigation

Navigation encompasses the ability of the robot to act, based on its knowledge and sensor values, so as to reach its goal position as efficiently and reliably as possible without collision. Two competences are required for mobile robot navigation: path planning and obstacle avoidance[14]. Given a map and a goal location, path planning involves identifying a trajectory that will cause the robot to reach the goal location when executed. Path planning is a strategic problem solving competence, as the robot must decide what to do over the long term to achieve its goals[59]. In comparison, the second competence obstacle avoidance is more of a reactive problem, and requires real-time sensors reading and modulating the trajectory of the robot in order to avoid collisions. Both planning and reacting work together when developing motion planners for mobile robots. Planners and reactors complement each other and are critical to the others success. The navigation challenge for the robot involves executing a course of action to reach its goal position. During the execution the robot must react to unforeseen events in such a way as to still reach the goal.

The goal of path planning is to find a path in the physical space from the initial position to the goal position, while avoiding all collisions with obstacles. Nonholonomic constraints limit the robots mobility[48]. Due to Brocketts theorem[6], it is well known that nonholonomic systems with restricted mobility cannot be stabilized to a desired configuration via differentiable, continuous, pure-state feedback. Different approaches have been proposed, which include discontinuous, hybrid, and time varying control laws. Moreover, researchers sometimes assume omnidirectional motion to simplify the control problem. The control strategies for nonholonomic systems can be divided into two main groups: open-loop strategies and closed-loop feedback strategies. In open-loop strategies the control signal is calculated off-line starting from the knowledge of the initial and final configurations of the system. By

their own nature, these strategies are not able to compensate for disturbances and model errors, therefore in practice the reached configuration may differ significantly from the desired final configuration. In closed-loop strategies the control signal is computed online, based on the knowledge of the actual configuration of the system and the final one. Closed-loop control strategies have a better chance of reaching the goal pose, as long as the nonholonomic constraints allow the motion[20].

2.4 Multi-Robot Systems

There is increasing interest in the development and deployment of groups of mobile robots to perform tasks that may be difficult or impossible for a single robot. Firstly, multiple robots can complete tasks more rapidly and efficiently than single robots. This concept is exploited in applications such as searching for unexploded landmines, where a team of mobile robots can distribute the task of searching the terrain among its members, to increase the overall efficiency of the search. Furthermore, multi-robot systems can perform tasks that are beyond the limits of a single robot. For example a team of unmanned ground vehicles could work together to cooperatively lift or push large objects in the workspace; an operation that is impossible for a single robot. Moreover complex tasks are often less expensive to execute by implementing a group of specialized simpler vehicles, rather than with a single expensive all-purpose platform. As tasks become more complex, having a team of mobile robots, each with their own specialization, is the best method to complete the mission. Finally, when exploring hazardous environments, where failure of one robot should not lead to the failure of the entire mission, a team of mobile robots is the best approach[39]. Multi-robot teams add redundancy, increase the fault tolerance of the group, and are a more robust as a team than single robots.

Multi-robot systems have many benefits over their single robot counterparts,

since they can perform tasks with greater efficiency, less cost, and they present a more robust solution. Teams of mobile robots must coordinate with each other in order to successfully complete tasks. Robots need to allocate jobs, define their workspace, work together, and cooperate with each other to perform their mission. Multi-robot teams exchange information over a wireless network, so all of the robots know where their team members are at all time, what they are working on, and they can direct each other in their tasks[45]. Multi-robot systems are more complex to engineer than single robot systems, and as the number of mobile robots working in an environment increases, the greater the likelihood of inter-robot collisions. A coordination strategy between the robots is instrumental in the successful deployment of multi-robot teams. Collision avoidance strategies alone are not enough, and large groups of mobile robots must have some sense of how to move together as a group[13].

2.5 Formation Control of a Team of Mobile Robots

For large multi-robot systems, navigation strategies for path planning and collision avoidance are insufficient for all of the robots to successfully reach the goal position while avoiding obstacles, and a mechanism of cooperation is required in order for the robots to move effectively and cohesively in the environment. Two popular approaches to group coordination are proposed: self-organizing systems and geometric formations. Firstly, self-organizing systems create a global order out of the local interactions between components of an initially disordered system[41]. Self-organizing systems are spontaneous, they are not directed or controlled by any agent or subsystem, and are regulated by individual behaviors of the agents. The resulting organization is wholly decentralized and distributed over all components of the system. Groups of self-organizing robots look like swarms as they travel

across the environment, where each robot takes a place among the pack and stay a safe distance away from neighboring robots. The swarms also change their shape to accommodate changes in the terrain, to fit into narrow passages and to travel around difficult obstacles. Geometric formations on the other hand take on a more centralized approach to group coordination. They adopt military style formation strategies, where all robots move in a defined manner dictated by the desired shape of the formation, and follow a leader to reach the goal position. Formation shapes must accommodate obstacles in the environment in order for the group to successfully reach the goal position; for example, a group of robots in a V-formation must morph its shape into a platoon formation in order to cross a narrow bridge.

Chapter 3

Literature Review

In this chapter, seven formation controller approaches from literature are examined. Section 3.1 looks at behavior based formations in multi-agent systems. Section 3.2 describes the work contributed to the artificial potential field formation controller. Section 3.3 examines graph theoretic solutions to the multi-robot coordination problem. Section 3.4 looks at formations that move together in a virtual structure. Section 3.5 looks at the leader-follower formation controller and its variations. Section 3.6 describes biologically inspired formations and self-organizing formations. Finally, section 3.7 looks at navigation and formation approaches using fluid mechanics principles.

3.1 Behavior Based Formations

In the behavioral approach, a number of basic behaviors are prescribed, like obstacle avoidance, formation keeping, and goal seeking. The overall control action is a weighted average of the control actions for each basic behavior.

In [1], the author describes how formations can be derived as a result of the culmination of reactive behaviors. In the paper, new formations in multi-robot teams are presented and evaluated based on the behavioral approach. The formation

behaviors are integrated with other navigational behaviors to enable a robotic team to reach navigational goals, avoid hazards and simultaneously remain in formation.

In [12], the paper describes a flexible architecture for modeling thousands of autonomous agents simultaneously. The agents behavior is based on a subsumption architecture in which individual behaviors are prioritized with respect to all others. The primary behavior explored in this work is a group formation behavior based on social potential fields, and this paper extends the social potential field model by introducing a neutral zone within which other behaviors may exhibit themselves. The paper evaluates the effect of social potential fields in the presence of agent death (failure) and imperfect sensory input.

In [25], the authors address the challenge in deploying teams of robots in real-world applications, and develops a system to automate the control of teamwork, such that the designer can focus on the task-work. Existing teamwork architectures seeking to address this challenge are monolithic, in that they commit to interaction protocols at the architectural level, and do not allow the designer to mix and match protocols for a given task. A BITE architecture is presented, a behavior-based teamwork architecture that automates collaboration in physical robots in a distributed fashion. BITE separates task behaviors that control a robot's interaction with its task, from interaction behaviors that control a robot's interaction with its teammates. This distinction provides for flexibility to collaborate effectively. It also allows BITE to synthesize and significantly extend existing teamwork architectures.

In [2], the author addresses a wide range of multi-robot coordinated movement tasks to derive a formation strategy that offers: scalability (the approach should easily scale to any number of agents), locality (the behaviors should depend only on the local sensors of each agent), and flexibility (the behaviors should be flexible so as to support many formation shapes). To provide these features a new behavior-based approach is introduce. The new strategy is based loosely on the way molecules

form crystals. From the point of view of each robot in the group, every other robot has several local attachment sites other robots may be attracted to. This type of attachment site geometry roughly corresponds to molecular covalent bonding. Just as different crystal shapes result from different covalent bond geometries, robot formation shapes are influenced by the attachment site geometries employed.

In [55], a component is constructed through the use of a heterogeneous multi-robot system. The task requires component acquisition, cooperative transport, and cooperative precision manipulation. A behavior-based architecture provides adaptability, and the approach minimizes computation, power, communication, and sensing for applicability to space-related construction efforts.

In [61], the authors examines hybrid systems with behavioral and model-based strategies. They claims that a strict behavior-based approach can scale to higher levels of complexity than many robotics researchers assume, and that the resulting systems are in many cases more efficient and robust than those that rely on classical AI deliberative approaches. The system focuses on cooperative autonomous robots in dynamic environments. Three design principles are introduced for complex cooperative behaviorminimalism, statelessness and tolerance

In [40], the paper describes the use of behaviors as the underlying control. The paper focuses on the role and power of behaviors as a representational substrate in learning policies and models in multi-robot systems, as well as learning from other agents (by demonstration and imitation).

3.2 Artificial Potential Field Formations

In the navigation case, the artificial potential field method creates a field across the robot's map that directs the robot to the goal position. The artificial potential field approach directs a robot as if it were a particle moving in a gradient vector

field. Positively charged particles are attracted to the negatively charged goal, and repulsed by positively charged obstacles. In the multi-robot case, artificial potential fields can be used to bind robots together in a self-organizing formation as the group travels to the goal.

In [3], an artificial potential field swarming method is derived using artificial potential fields generated from normal and sigmoid functions. These functions construct the surface that swarm members travel on, controlling the overall swarm geometry and the individual member spacing. Limiting functions are defined to provide tighter swarm control by modifying and adjusting a set of control variables forcing the swarm to behave according to set constraints, formation shape, and member spacing. The swarm function and limiting functions are combined to control swarm formation, orientation, and swarm movement as a whole. Parameters are chosen based on desired formation as well as user-defined constraints. This approach compared to others, is simple, computationally efficient, scales well to different swarm sizes, to heterogeneous systems, and to both centralized and decentralized swarm models.

In [38], a behavior-based reactive control is developed for the formation control of multiple robots. The combination of potential-based avoidance and following-wall behavior ensures the robots are able to avoid the obstacles smoothly. To get rid of the following-wall behavior when it is not needed, a step of judging its necessity is added. Finally the experiment on the Amigo-Bot robots showed that the robots can return the formation after passing the obstacles.

In [24], a new control algorithm based on a potential function and behavior rules is developed to effectively control the formation of a multiple autonomous underwater vehicle (AUV). Proper potential functions concerning with object, obstacle and the structure of the formation is chosen to design a new distributed control algorithm to make the formation of the multiple AUV system track the target effectively.

In [15] an efficient, simple, and practical real time path planning method for multiple mobile robots in dynamic environments is introduced. Harmonic potential functions are utilized along with the panel method known in fluid mechanics. First, a complement to the traditional panel method is introduced to generate a more effective harmonic potential field for obstacle avoidance in dynamically changing environments. Second, a group of mobile robots working in an environment containing stationary and moving obstacles is considered. Each robot is assigned to move from its current position to a goal position. The group is not forced to maintain a formation during the motion. Every robot considers the other robots of the group as moving obstacles and hence the physical dimensions of the robots are also taken into account. The path of each robot is planned based on the changing position of the other robots and the position of stationary and moving obstacles.

In [30], a control method based on a set of artificial potential functions is described. The type of artificial potential used for the experiments is dependent on the experiment objectives: avoiding collisions between robots and keeping them in the ordered formation, executing task by the formation (e.g. moving formation into desired position), building formation and avoiding collisions with obstacles. In this paper a framework is proposed for the attraction potential function, which allows the system to build formation and repulsion potential function to avoid collisions with obstacles. The main advantage of presented approach is that it is easy scalable because control is based on general rule that determines behavior of all robots.

In [69], collision between robots and the global minimum point, which exist in the traditional artificial potential field TAPF, seriously affect the formation of multi-robot system. The paper constructs a dynamic artificial potential field based on the local information to solve the problem of the TAPF. Collisions between robots are avoided by introducing the repulsion between robots. Adjusting the robot movement direction within the escaping area solves the problem of global minimum point.

In [34], a framework is presented for coordinated and distributed control of multiple autonomous vehicles using artificial potentials and virtual leaders. Artificial potentials define interaction control forces between neighboring vehicles and are designed to enforce a desired inter-vehicle spacing. A virtual leader is a moving reference point that influences vehicles in its neighborhood by means of additional artificial potentials. Virtual leaders can be used to manipulate group geometry and direct the motion of the group. The approach provides a construction for a Lyapunov function to prove closed-loop stability using the system kinetic energy and the artificial potential energy. Dissipative control terms are included to achieve asymptotic stability. The framework allows for a homogeneous group with no ordering of vehicles; this adds robustness of the group to a single vehicle failure.

In [21], a novel approach for representing formation structures in terms of queues and formation vertices, rather than with nodes. Furthermore, the new concept of artificial potential trenches is introduced to effectively control the formation of a group of robots. The scheme improves the scalability and flexibility of robot formations when the team size changes, and at the same time, allows formations to adapt to obstacles. Furthermore, for multi-robot teams to operate successfully in real and unstructured environments, the instant goal method is used to effectively solve the local minima problem.

In [19], a control strategy of multi-agent swarms is derived based on artificial potential functions and the sliding-mode control technique. In that model, the individual interactions are based on artificial potential functions, and the motion of the individuals is along the negative gradient of the combined potential. A general model for vehicle dynamics of each agent (swarm member) is applied using sliding-mode control theory to force their motion to obey the dynamics of the kinematic model. The presented control scheme is robust with respect to disturbances and system uncertainties.

In [42], a framework is proposed for formation stabilization of multiple autonomous vehicles in a distributed fashion. Each vehicle is assumed to have simple dynamics, with a directed (or an undirected) information flow over the formation graph of the vehicles. The goal is to find a distributed control law for each vehicle that makes use of limited information regarding the state of other vehicles. The key idea in formation stabilization is the use of natural potential functions obtained from structural constraints of a desired formation in a way that leads to a collision-free, distributed, and bounded state feedback law for each vehicle.

3.3 Graph Theoretic Formation

Graph theory provides mathematical relations that connect robot agents in a multi-agent network.

In [11] the paper addresses a new approach for modeling and control of multiple teams of mobile robots navigating in a terrain with obstacles, while maintaining a desired formation and changing formations when required. A set of shape variables, r, describe the relative positions of robots, and a control graph, H, describes the behaviors of the robots in the formation. It is assumed that all the robots are equipped with the appropriate sensors to detect and avoid other robots and obstacles in the environment. The framework enables the representation and enumeration of possible control graphs, and the coordination of transitions between any two control graphs. Further, an algorithm is described that allows each team of robots to move between any two formations, while avoiding obstacles. As the number of robots increases, the number of possible control graphs increases. However, because the control computations are decentralized, the algorithms scale with the number of robots.

In [47], the author introduces a class of triangulated graphs for algebraic represen-

tation of formations that specifies a mission cost for a group of vehicles. This representation plus the navigational information allows the system to formally specify and solve tracking problems for groups of vehicles in formations using an optimization-based approach. The approach is illustrated using a collection of six under-actuated vehicles that track a desired trajectory in formation.

In [31] the paper investigates a method for decentralized stabilization of vehicle formations using techniques from algebraic graph theory. The vehicles exchange information according to a pre-specified communication digraph, G. A feedback control is designed using relative information between a vehicle and its neighbors in G. A direct relationship between the rate of convergence to formation and the eigenvalues of the (directed) Laplacian of G are calculated. Various special situations are discussed, including symmetric communication graphs and formations with leaders.

In [16], cooperation among a collection of vehicles, performing a shared task and using inter-vehicle communication to coordinate their actions, is developed. Tools from algebraic graph theory prove useful in modeling the communication network and relating its topology to formation stability. The authors prove a Nyquist criterion that uses the eigenvalues of the graph Laplacian matrix to determine the effect of the communication topology on formation stability. A method is also proposed for decentralized information exchange between vehicles. The information flow can thus be rendered highly robust to changes in the graph, enabling tight formation control despite limitations in inter-vehicle communication capability.

3.4 Virtual Structures

In [67] the formation control problem of multiple mobile robots is investigated. A formation control scheme based on a hierarchical virtual structure is proposed. The mobile robots are divided into clusters according to their distributions in space

and then the motion trajectory of every cluster (virtual structure) is defined, so the motion of virtual structure is transformed to desired trajectory of every robot. Furthermore, a finite-time tracking control algorithm with variable structure is proposed.

In [7], the formation control problem for unicycle mobile robots is studied. A virtual structure control strategy with mutual coupling between the robots is proposed. The rationale behind the introduction of the coupling terms is the fact that these introduce additional robustness with respect to perturbations as compared to typical leader-follower approaches. The applicability of the proposed approach is shown in experiments with a group of mobile robots controlled over a wireless communication network.

In [46], Formation control ideas for multiple spacecraft using the virtual structure approach are presented. If there is no formation feedback from the spacecraft to the virtual structure, the spacecraft will get out of formation when the virtual structure moves too fast for the spacecraft to track. The spacecraft may also get out of formation when the system is affected by internal or external disturbances. To overcome these drawbacks, an idea of introducing formation feedback from the spacecraft to the virtual structure is illustrated in detail. An application of these ideas to multiple spacecraft interferometers in deep space is given.

In [32], an approach to 3D formation tracking control based on virtual structures. In this approach, the virtual structure specifies the formation position of each vehicle, and virtual leaders are constructed ahead of each position for the vehicles to follow. The velocity of the virtual structure is assumed to be piecewise constant to provide zero steady-state error. In order to prevent transients from leading to collisions, deconfliction techniques are applied. Analysis of stability and performance is given, and results are demonstrated in simulation.

In [35], the maintenance of a geometric configuration during cooperative move-

ment is analyzed. To address this problem, the concept of a Virtual Structure is introduced. Using this idea, a general control strategy is developed to force an ensemble of robots to behave as if they were particles embedded in a rigid structure. The method was instantiated and tested using both simulation and experimentation with a set of 3 differential drive mobile robots. Results are presented that demonstrate that this approach is capable of achieving high precision movement that is fault tolerant and exhibits graceful degradation of performance. In addition, this algorithm does not require leader selection as in other cooperative robotic strategies. Finally, the method is inherently highly flexible in the kinds of geometric formations that can be maintained.

3.5 Leader-Following Formations

In [68] a leader-follower pattern is applied to multiple UAVs to maintain a fixed geometrical formation during navigation. In this approach, the leader is commanded to fly on some predefined trajectories, and each follower is controlled to maintain its position in formation using the measurement of its inertial position and the information of the leader position and velocity, obtained through a wireless modem. In order to avoid possible collisions of UAV helicopters in the actual formation flight test, a collision avoidance scheme based on some predefined alert zones and protected zones is employed.

In [51], a framework for controlling groups of autonomous mobile robots is developed to achieve predetermined formations based on leader-following approach. A three-level hybrid control architecture is proposed to implement both centralized and decentralized cooperative control. Under such architecture, the global-level formation control problem of N robots is decomposing into decentralized control problems between N followers and their designated leader. In the leader-follower control level,

two basic controllers are proposed to make the following robot keep relative position with respect to the leader and avoid collisions in the presence of obstacles. Then, graph theory is introduced to formalize specified formation patterns in a simple but effective way, and two types of switching between these formations are also proposed.

In [10], the author formalizes the leader-follower problem in a geometric framework, and a proposed a controller is offered as an alternative to those existing in the literature. It is shown that the geometry of the formation imposes a bound on the maximum admissible curvature of leader trajectory. A peculiar characteristic of the proposed strategy is that the position of the followers is not fixed with respect to the leader reference frame but varies in suitable cones. The formation geometry adapts to the followers dynamics and this allows lower control effort with respect to other approaches based on rigid formations.

In [54], the design of a linear robust dynamic output feedback control scheme for output reference trajectory tracking tasks in a leader-follower non-holonomic car formation. A simplification is proposed on the followers exact open loop position tracking error dynamics, obtained by flatness considerations. Linear Luenberger observers may be readily designed, which include the, self-updating, internal model of the unknown disturbance input vector components as generic time-polynomial models. The proposed Generalized Proportional Integral (GPI) observers, which are the dual counterpart of GPI controllers, achieve a, simultaneous, disturbance estimation and tracking error phase variables estimation. This, on-line, gathered information is used to advantage on the followers linear output feedback controller thus allowing for a simple, yet efficient, disturbance and control input gain cancelation effort. The results are applied to control the fixed time delayed trajectory tracking of the leader path on the part of the follower.

In [58], the paper investigates the stability properties of mobile agent formations which are based on leader following. nonlinear gain estimates are derived that

capture how leader behavior affects the interconnection errors observed in the formation. Leader-to-formation stability (LFS) gains quantify error amplification, relate interconnection topology to stability and performance, and offer safety bounds for different formation topologies. Analysis based on the LFS gains provides insight to error propagation and suggests ways to improve the safety, robustness, and performance characteristics of a formation.

In [36], the paper investigates the leader-following based formation control of nonholonomic mobile robots. A new kinematic model is presented for the leader-follower system using Cartesian coordinates. Based on this new model and the direct Lyapunov method, a globally stable controller is derived for the whole system. Simulation results are included to verify the feasibility of the proposed model and controller.

In [8], a review on the current control issues and strategies on a group of unmanned autonomous vehicles/robots formation is presented. Formation control has broad applications and becomes an active research topic in the recent years. In this paper, a review of key issues in formation control in including the leader-follower method. This paper contributes with a new and interesting consideration on formation control and its application in distributed parameter systems.

3.6 Biologically Inspired and Self-Organizing Formations

Biologically inspired formations mimic successful groupings of animals in nature, such as flocking birds, foraging insects, schooling fish, and swarming bees. Biologically inspired formations often apply self-organizing principles in their motion. Self-organizing systems create organization and structure over time without a centralized controller. It is a process of generating order out of randomness, and can

be though of as reverse entropy. In a self-organization algorithm, the specifications of components are left undefined, and these components are expected to organize themselves until they form a system that possesses the desired functionality.

In [65], the paper deals with formation control strategies based on Virtual Structure (VS) for multi-vehicle systems. Several control laws are proposed for networked multi-nonholonomic vehicle systems in order to achieve VS consensus, VS Flocking and VS Flocking with collision- avoidance. First, Virtual Vehicle for the feedback linearization is considered, and we propose VS consensus and Flocking control laws based on a virtual structure and consensus algorithms. Then, VS Flocking control law considering collision avoidance is proposed and its asymptotical stability is proven.

In [56], the paper revisits the problem of multi-agent flocking. The authors prove that the Olfati-Saber flocking algorithm still enables all the informed agents to move with the desired constant velocity, and an uninformed agent to also move with the same desired velocity if the informed agents can influence it from time to time during the evolution. Numerical simulation demonstrates that a very small group of the informed agents can cause most of the agents to move with the desired velocity and the larger the informed group is the bigger portion of agents will move with the desired velocity. In the situation where the virtual leader travels with a varying velocity, a proposed modification to the Olfati-Saber algorithm shows that the resulting algorithm enables the asymptotic tracking of the virtual leader. That is, the position and velocity of the center of mass of all agents will converge exponentially to those of the virtual leader. The convergent rate is also given.

In [57], the control law is derived that ensures collision avoidance and cohesion of the group exhibiting flocking motion. flocking motion is established, as long as connectivity in the neighboring graph is maintained.

In [33], the problem of multiple robots system formation using a distributed

control method is studied. The paper uses a swarm flocking control algorithm to implement the biods model of Reynolds among multi-robots. With the help of graph theory, a provably-stable flocking control law, which ensures that the internal group formation is stabilized to a desired shape, while all the robots velocities and directions converge to the same. Player/Stage simulation results show that the proposed method can be efficiently applied to multiple robots formation control. With the characteristic of Player/Stage, the algorithm in this paper can be implemented on the real robots with so few or no changes.

3.7 Navigation and Formation Control Based on Fluid Principles

In [64], the paper presents a general framework for coordinated motion control of autonomous swarms in the presence of obstacles. The proposed framework judiciously combines concepts and techniques from potential flows, artificial potentials and dynamic connectivity to realize complex swarm behaviors. To begin with, existing concepts from potential flows in fluid mechanics are used to solve the single-agent navigation problem. As an extension, an analytical solution to the stagnation point problem is provided. The potential flow based framework is then modified significantly to facilitate the coordinated control of swarms navigating through multiple obstacles. Artificial potentials are employed for swarming as well as enhanced obstacle avoidance. A novel concept of dynamic connectivity is utilized to improve the performance of obstacle avoidance (Line of Sight Connectivity) and to organize diverse swarm behaviors (Random Connectivity). Simulation results with a set of developed algorithms are included to illustrate the viability the proposed framework.

In [50], the authors present a novel approach to obstacle avoidance for a group of robots moving in formation. This idea is originally inspired by a phenomenon in

hydrodynamics. In this approach, a virtual robot is introduced as a reference point (beacon) to determine a collision-free trajectory. Taking the virtual point as a basic point, the robot group establishes a rigid body-like formation. Since the collision-freeness of the virtual robot does not guarantee all other robots avoiding obstacles, we employ a flexible formation control scheme in the framework of rigid body-like formation. Two kinds of transformation are introduced to help every robot negotiate obstacles. This approach gives a new principle to deal with obstacle avoidance in formation control for multi-robot system. Simulation results are presented to verify its effectiveness.

In [22], the paper presents a path planning method, based on fluid mechanics, able to deal with unstructured terrain models. The algorithm uses the finite element method to compute a velocity potential function free from local minima. Then, several streamlines are computed as a road map and the optimal path is selected among the candidate paths. The approach is implemented on the Canadian Space Agency (CSA) Mars Robotics Testbed (MRT) rover and tested at the CSA Mars Emulation Terrain (MET). To confirm the feasibility of the method, the path planner has been tested on 284 LIDAR scans collected in a realistic outdoor challenging terrain.

In [27], a computational metaphor for path generation is generated from fluid dynamics. It is powerful because it can find optimal paths an a maze of arbitrary complexity, and it is flexible because it readily adapts to any change in the topology of the maze. With the selection of appropriate boundary conditions, the fluid dynamics metaphor does not suffer from local minima. Finally the topological properties of the path generated by the fluid dynamics metaphor are discussed using tools of topology.

In [37], the paper presents a numerical potential function for point-robot path planning in configuration space based on the theory of fluid mechanics. Ideal fluid

is first simulated using Poissons equation and heuristic path planning algorithms are established by comparisons of the velocity potentials. Several computational techniques are experimented and compared. A bitmap collision detection technique is proposed for non-point robots. This fluid model creates an environment, which is not only free of local minima but also beneficial for navigation control.

In [60], the authors borrow from concepts of hydrodynamic analysis. The paper presents stream functions, which satisfy Laplace's equations as local-minima free methods for producing potential-field based navigation functions in two dimensions. These functions generate smoother paths (i.e. more suited to aircraft-like vehicles) and a method is developed for constructing analytic Stream functions. The effects of introducing multiple obstacles are discussed and current work in this direction is detailed.

Chapter 4

Proposed Navigation and Formation Control Approaches

In this chapter, the proposed navigation and formation controllers will be discussed. Section 4.1 will discuss the proposed navigation controller, modeled after the fluid flow principles of streamlines and the velocity potential. Moreover, section 4.2 and section 4.3 will look at the proposed formation control approaches for a team of homogenous mobile robots. In our research, two formation control approaches will be explained for their overall coordination effectiveness. Section 4.2 will look at the derivation of the reconfiguring Leader-Follower formation controller, that adapts its formation shape in order to navigate the team around and through difficult obstacles. Section 4.3 will expand the velocity potential flow method to a self-organizing formation, that will be able to organize the movement of the team in a decentralized fashion.

4.1 Navigation Using the Velocity Potential Method

For our simulation and experimental work, we needed to develop a navigation controller to bring the team of mobile robots from a start position to a goal position without colliding with obstacles in the environment. For our research work, we decided to look at reactive navigation controllers for the team motion, since they were more adaptable to changes in the environment.

This section will look at the work done to create a reactive navigation controller using the velocity potential method. The velocity potential method is inspired by fluid flow principles, and derives its equations from fluid mechanics. First, section 4.1.1 will look at the strengths and weaknesses of the artificial potential field method, a popular reactive navigation controller used by roboticists. Section 4.1.2 will look at fluid mechanics principles that describe fluid motion. Finally, section 4.1.3 will describe the attractive and repulsive flow equations that regulate the velocity potential navigation controller.

4.1.1 The Artificial Potential Field Method: Strengths and Weaknesses

The artificial potential field method is one of the most popular reactive navigation strategies in mobile robotics. The algorithm was first introduced by [29], and since then many variations on the artificial potential field method have been introduced to improve robot navigation in complex dynamic terrains.

The artificial potential field (APF) method creates a field across the robot's map that directs the robot to the goal position. The APF approach directs a robot as if it were a particle moving in a gradient vector field. Positively charged particles are attracted to the negatively charged goal, and repulsed by positively charged

obstacles. Figure 4.1 [9] illustrates the navigation concept, and shows the attractive and repulsive pull on the robot. The combination of repulsive and attractive forces directs the robot from the start location to the goal location while avoiding obstacles.

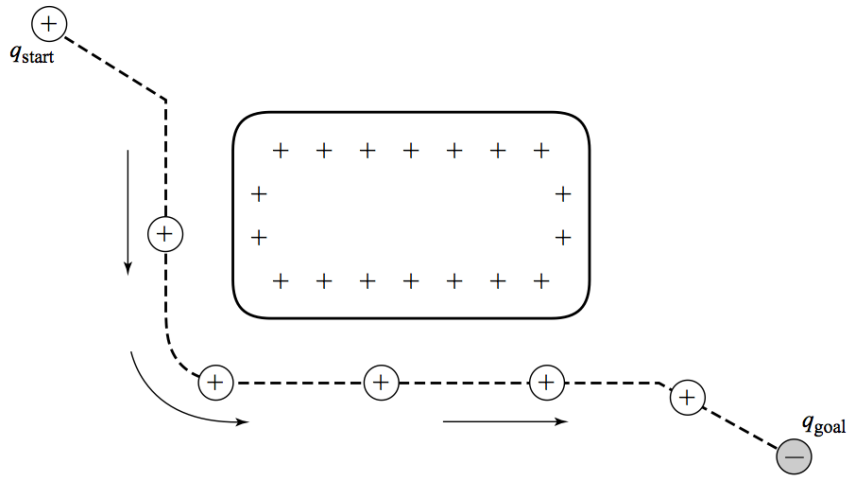


Figure 4.1: The negative charge attracts the robot and the positive charge repels it, resulting in a path, denoted by the dashed line, around the obstacle and to the goal

The robot follows a path "downhill" by following the negative gradient of the potential function U, termed the gradient descent. The robot terminates its motion when it reaches a point where the gradient vanishes $\nabla(q) = 0$. The point where the gradient vanished is termed the critical point, and it is classified as either: a maximum, a minimum, or a saddle point. Figure 4.2 [9]. Figure 4.3a[28] and figure 4.3b[28] illustrate the force field created by the artificial potential field method that directs the robot towards the goal position.

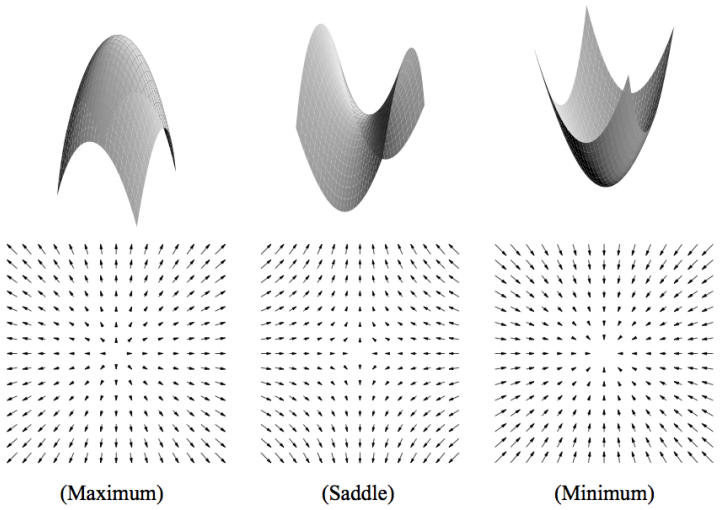
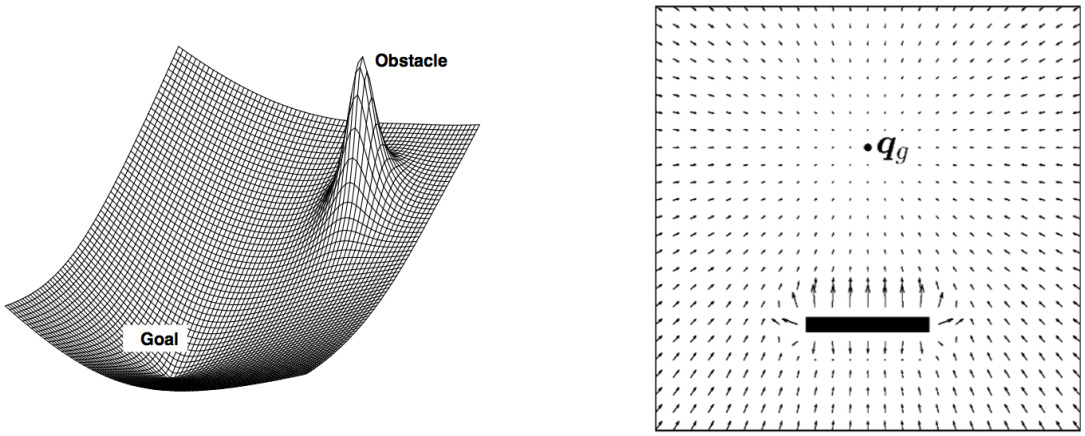


Figure 4.2: Different types of critical points: (top) graphs of functions, (bottom) gradients of functions



(a) (b)

Figure 4.3: Potential distribution example for an obstacle and a goal

In the artificial potential field method, a net force $F(q)$ is calculated for a robot at coordinates $q = (x, y)$, by determining the gradient of the attractive and repulsive artificial potential field $U(q)$.

$$F(q) = -\nabla U(q) \tag{4.1}$$

$$U(q) = U_{att}(q) + U_{rep}(q) \tag{4.2}$$

The attractive potential field is modeled after a parabolic function. A common attractive field equation is displayed below.

$$U_{att}(q) = \frac{1}{2}\zeta d^2(q, q_{goal}) \tag{4.3}$$

$$\nabla U_{att}(q) = \zeta(q, q_{goal}) \tag{4.4}$$

The repulsive potential keeps the robot away from an obstacle, and its strength is relative to the robot's proximity to the obstacle. As the robot comes closer to the obstacle, the repulsive force acting on the robot increases, to push it far away from a potential collision. The strength of the repulsive force is dependent on the distance $D(q)$, which is the distance to the closest obstacle. In the equation, $Q^* \in \mathbb{R}$ is a factor that allows the robot to ignore obstacles sufficiently far away from it, and ζ and η are attractive and repulsive gains respectively

$$U_{rep}(q) = \begin{cases} \frac{1}{2}\eta\left(\frac{1}{D(q)} - \frac{1}{Q^*}\right)^2, & D(q) \leq Q^* \\ 0, & D(q) > Q^* \end{cases} \tag{4.5}$$

$$\nabla U_{rep}(q) = \begin{cases} \eta\left(\frac{1}{Q^*} - \frac{1}{D(q)}\right)\frac{1}{D^2(q)\nabla D(q)}, & D(q) \leq Q^* \\ 0, & D(q) > Q^* \end{cases} \tag{4.6}$$

4.1.1.1 The Local Minima Problem

The problem that plagues all gradient descent algorithms is the possible existence of local minima in the potential field. There is no guarantee that the gradient descent will find a path the goal position, and the robot may be stranded in the terrain and never reach the goal position. Figure 4.4 [9] shows a diagram of the local minima problem, where the attractive and repulsive forces on the robot have equal and

opposite magnitudes, which stops all robot movement.

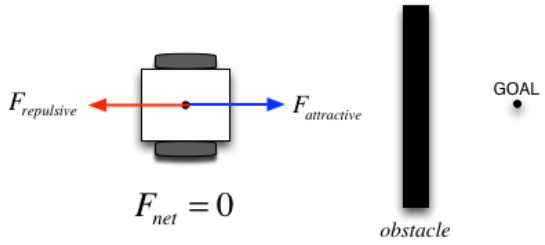
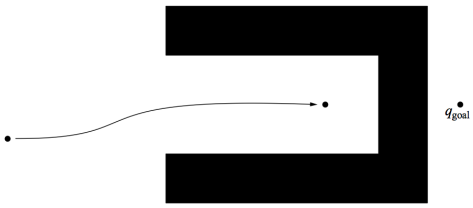
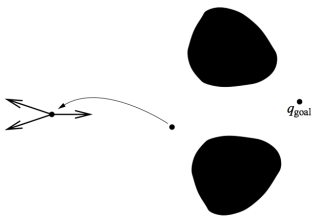


Figure 4.4: The local minima problem

The local minimum problem occurs most often when the robot encounters two types of obstacle geometries: concave and narrow passage. Firstly, figure 4.5a shows a robot entering a concave obstacle. The robot is initially attracted to the goal as it approaches the horseshoe shaped obstacle, and as the robot moves inside the obstacle, the bottom arm of the obstacle deflects the robot upward until the top arm of the horseshoe begins to influence the robot. At this point, the effect of the top and bottom arms keeps the robot halfway between them and the goal continues to attract the robot; the robot reaches a point where the effect of the obstacle's geometry counteracts the attraction of the goal, $\nabla(q) = 0$, and the robot never reaches the goal position. A similar result occurs with the narrow passage, figure 4.5b [9], where the effect of the two adjacent obstacles create a net zero force which stops the robot in place.



(a) Local minimum inside the concave obstacle



(b) Local minimum approaching a narrow passage

Figure 4.5: Local minima cases

4.1.2 Fluid Flow and the Velocity Potential Method

In our research, we were attracted to the simplicity and efficiency of the artificial potential field reactive navigation controller, however we wanted to improve on its results for complex obstacle geometries such as the concave obstacle and narrow passage. We were inspired by fluid flow around obstacles; to create smooth paths that facilitate the nonholonomic constraints of unmanned ground vehicles. Figure 4.6[18] shows the streamlines of fluid flowing over an automobile in a wind tunnel.

Figure 4.6: Streamlines over an automobile in a wind tunnel

The objective was to design a navigation controller that would lead the robots on streamlines, flowing around obstacles and towards a goal position. For incompressible flow, the velocity vector of the fluid always points tangent to the streamline. Figure 4.7[44] shows the motion of fluid streamlines over an airfoil shape. When the velocity is zero there is no unique direction for the streamline, and the streamline can branch out into different directions. This point is called the stagnation point.

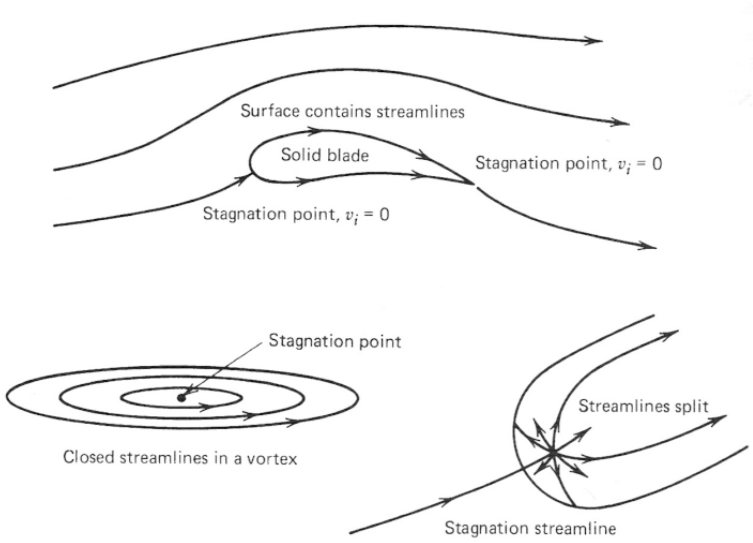


Figure 4.7: Streamline patterns with stagnation points

Streamlines are everywhere tangent to the velocity vectors, and their equations in the two-dimensional plane are represented by the stream function $\psi = \psi(x, y)$ [49]. The stream function is very useful in describing fluid flows, and all properties of the flow including velocities and pressure could be related to it; the streamline is a single scalar unknown that can yield a complete description of the flow. By knowing the stream function of the fluid, velocity parameters u and v, velocity in the x and y directions respectively, can be calculated according to the equation below.

$$u = \frac{\delta\psi}{\delta y} \tag{4.7}$$

$$v = -\frac{\delta\psi}{\delta x} \tag{4.8}$$

Moreover, the fluid velocity can be calculated in cylindrical coordinates based on the stream function ψ.

$$V_r = -\frac{1}{r}\frac{\delta\psi}{\delta\theta} \tag{4.9}$$

$$V_\theta = -\frac{\delta\psi}{\delta r} \tag{4.10}$$

The velocity potential function ϕ is a companion function to the stream function ψ, to describe the flow of fluid. The velocity potential function is also a single scalar variable $\phi = \phi(x, y, t)$ that can be used to calculate flow velocities for irrotational flow according to the equation below[52].

$$u = -\frac{\delta\phi}{\delta x} \tag{4.11}$$

$$v = -\frac{\delta\phi}{\delta y} \tag{4.12}$$

The cylindrical velocity values can be calculated from the equations:

$$V_r = \frac{1}{r}\frac{\delta\phi}{\delta r} \tag{4.13}$$

$$V_\theta = -\frac{1}{r}\frac{\delta\phi}{\delta\theta} \tag{4.14}$$

Since, in an irrotational flow, the velocity field may be defined by the potential function ϕ, the theory is often referred to as the potential flow theory[66].

4.1.2.1 Elementary Plane Flows

Many methods exist for determining the stream function *psi* and the velocity potential function *phi* for a fluid motion[23]. A popular method utilizes the stream function and velocity potential function for five elementary plane flows, and develops the ψ and ϕ functions by combining the elementary flow models. In this section,

the five elementary plane flows are explained and their stream function and velocity potential function are listed.

Firstly, the uniform flow model in figure 4.8[18] is used for determining ψ and ϕ for fluid flowing in a straight path in the positive x-direction at a constant linear velocity.

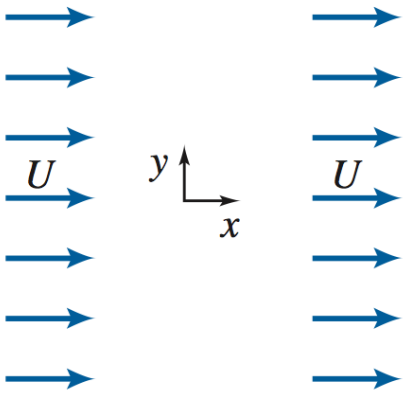
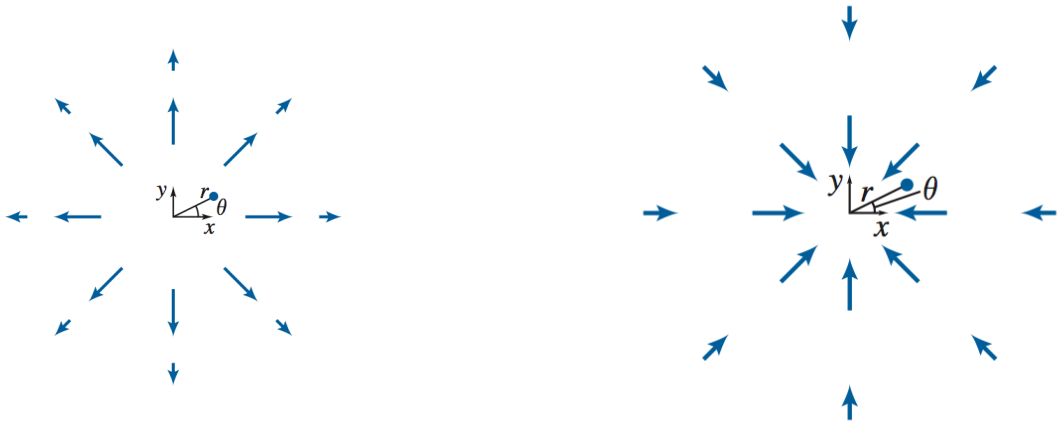


Figure 4.8: Uniform flow

$$\psi = Uy \tag{4.15}$$

$$\phi = -Ux \tag{4.16}$$

Furthermore, the source and sink flows, figure 4.9a[18] and figure 4.9b[18], shows the flow of fluid travelling away from the orgin and the flow of fluid moving toward the origin respectively. Their stream function and velocity potential functions are listed below.



(a) Source flow

(b) Sink flow

Figure 4.9: Source and Sink Flow

Source flow (from origin):

$$\psi = \frac{q}{2\pi}\theta \tag{4.17}$$

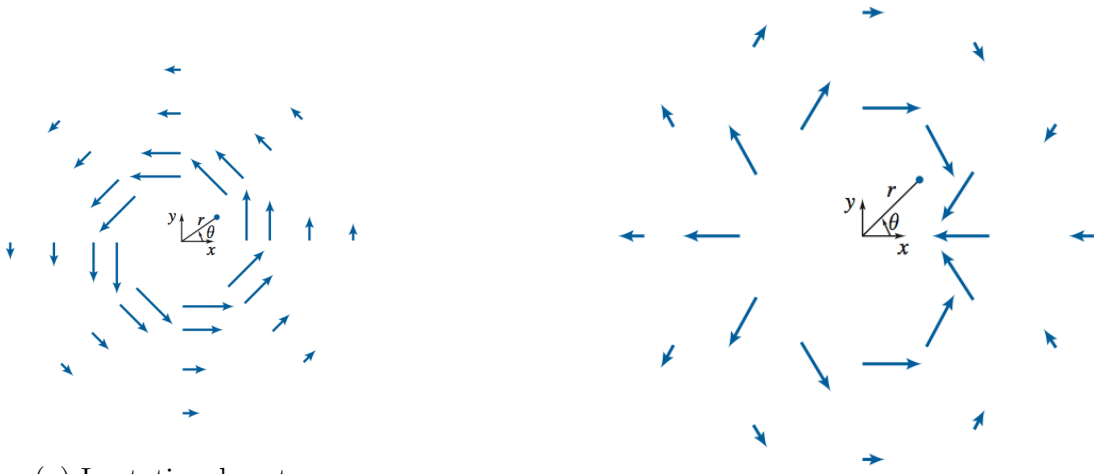
$$\phi = -\frac{q}{2\pi}\ln r \tag{4.18}$$

Sink flow (toward origin):

$$\psi = -\frac{q}{2\pi}\theta \tag{4.19}$$

$$\phi = \frac{q}{2\pi}\ln r \tag{4.20}$$

Moreover, figure 4.10a[18] and figure 4.10b[18] show the fluid flow of an irrotational vortex and doublet. Their equations for ψ and ϕ are calculated as follows:



(a) Irrotational vortex

(b) Doublet

Figure 4.10: Irrotational and Vortex Flow

Irrotational vortex (counterclockwise, center at origin):

$$\psi = -\frac{K}{2\pi}\ln r \tag{4.21}$$

$$\phi = -\frac{K}{2\pi}\theta \tag{4.22}$$

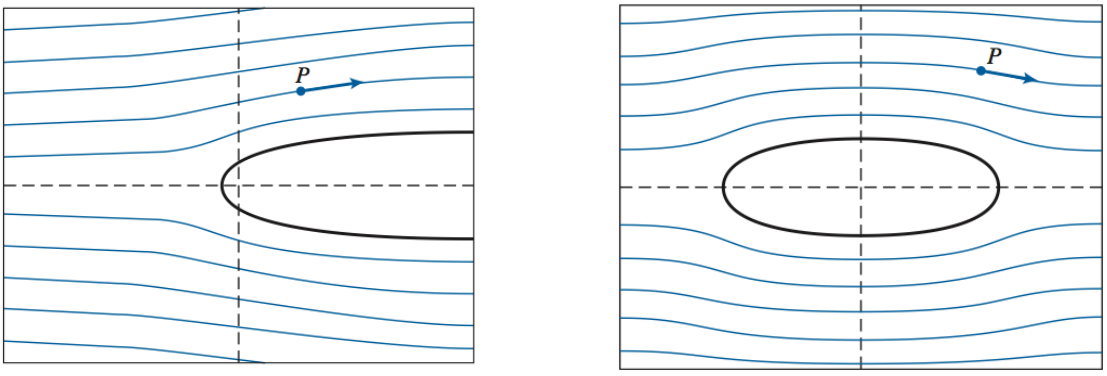
Doublet (center at origin):

$$\psi = -\frac{\Lambda \sin\theta}{r} \tag{4.23}$$

$$\phi = -\frac{\Lambda \cos\theta}{r} \tag{4.24}$$

4.1.2.2 Superposition of Elementary Plane Flows

Using the elementary plane flow model, complex flows can be approximated by combining various elementary flows. Figure 4.11a[18] and figure 4.11b[18] show the flow around an obstacle when combining the (1) source and uniform flow, and (2) source, sink, and uniform flow, respectively.



(a) Source and uniform flow

(b) Source, sink, and uniform flow

Figure 4.11: Superposition Flow 1

Source and uniform flow (flow past a half-body)

$$\psi = \frac{q}{2\pi}\theta + Ur\sin\theta \tag{4.25}$$

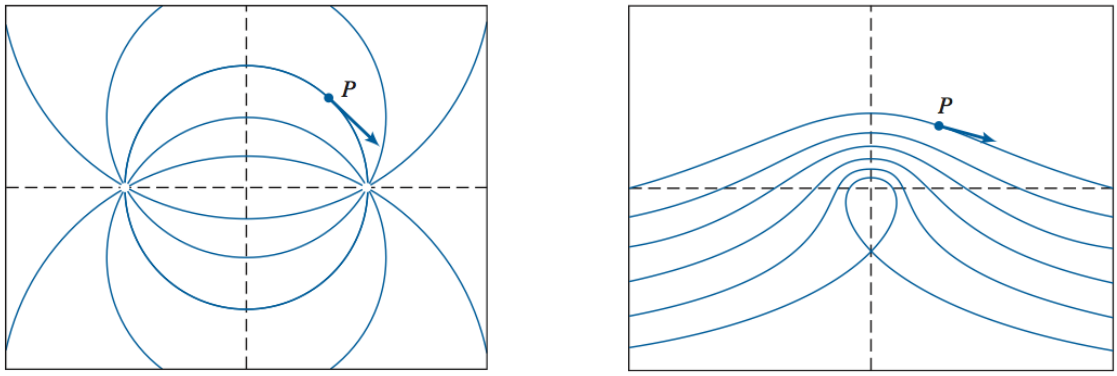
$$\phi = -\frac{q}{2\pi}\ln r - Ur\cos\theta \tag{4.26}$$

Source, sink, and uniform flow (flow past a Rankine body)

$$\psi = \frac{q}{2\pi}(\theta_1 - \theta_2) + Ur\sin\theta \tag{4.27}$$

$$\phi = \frac{q}{2\pi}\ln r\frac{r_2}{r_1} - Ur\cos\theta \tag{4.28}$$

Moreover, figure 4.12a[18] and figure 4.12b[18] show the flow of fluid when combining the (1) source and sink, and (2) vortex and uniform flow, elements respectively.



(a) Source and sink (b) Vortex and uniform flow

Figure 4.12: Superposition Flow 2

Source and sink (equal strength, separation distance on x-axis = 2a)

$$\psi = \frac{q}{2\pi}(\theta_1 - \theta_2) \tag{4.29}$$

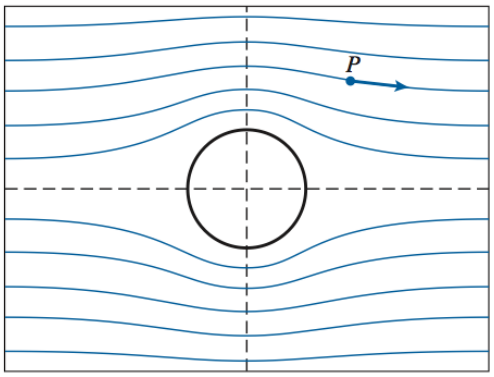
$$\phi = \frac{q}{2\pi} \ln r\frac{r_2}{r_1} \tag{4.30}$$

Vortex (clockwise) and uniform flow

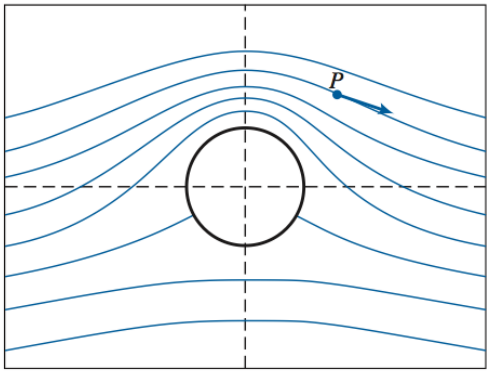
$$\psi = \frac{K}{2\pi} \ln r + ur \sin\theta \tag{4.31}$$

$$\phi = \frac{K}{2\pi}\theta - Ur\cos\theta \tag{4.32}$$

Furthermore, figure 4.13a[18] and figure 4.13b[18] show the flow characteristics when combining (1) the doublet and uniform flow, and (2) the doublet, vortex, and uniform flow, respectively.



(a) Doublet and uniform flow



(b) Doublet, vortex, and uniform flow

Figure 4.13: Superposition Flow 3

Doublet and uniform flow (flow past a cylinder)

$$\psi = U\left(r - \frac{\Lambda}{Ur}\right)\sin\theta \tag{4.33}$$

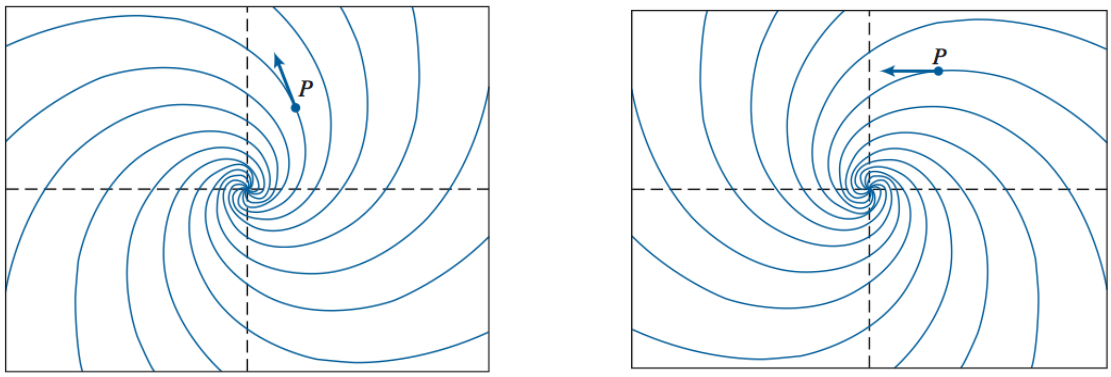
$$\phi = -\frac{\Lambda\cos\theta}{r} - Ur\cos\theta \tag{4.34}$$

Doublet, vortex (clockwise), and uniform flow (flow past a cylinder with circulation)

$$\psi = -\frac{\Lambda\sin\theta}{r} + \frac{K}{2\pi}\ln r + Ur\sin\theta \tag{4.35}$$

$$\phi = -\frac{\Lambda\cos\theta}{r} + \frac{K}{2\pi}\theta - Ur\cos\theta \tag{4.36}$$

In addition, figure 4.14a[18] and figure 4.14b[18] show the flow characteristics and equations when combining (1) a source and vortex, and (2) a sink and vortex, respectively.



(a) Source and vortex

(b) Sink and vortex

Figure 4.14: Superposition Flow 4

Source and vortex (spiral vortex)

$$\psi = \frac{q}{2\pi}\theta - \frac{K}{2\pi}\ln r \tag{4.37}$$

$$\phi = -\frac{q}{2\pi}\ln r - \frac{K}{2\pi}\theta \tag{4.38}$$

Sink and vortex

$$\psi = -\frac{q}{2\pi}\theta - \frac{K}{2\pi}\ln r \tag{4.39}$$

$$\phi = \frac{q}{2\pi}\ln r - \frac{K}{2\pi}\theta \tag{4.40}$$

Finally, figure 4.15 shows the vortex pair that is created when combining two vortices of equal strength and opposite rotation at a given separation distace on the x-axis.

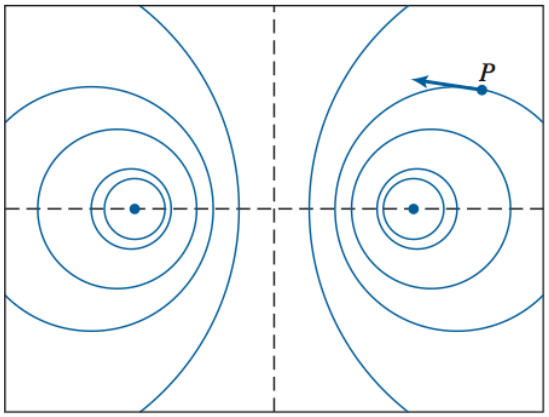


Figure 4.15: Vortex pair

Vortex pair (equal strength, opposite rotation, separation distance on x-axis = 2a)

$$\psi = \frac{K}{2\pi} \ln \frac{r_2}{r_1} \tag{4.41}$$

$$\phi = \frac{K}{2\pi}(\theta_2 - \theta_1) \tag{4.42}$$

4.1.3 Proposed Navigation Controller using the Velocity Potential Flow Theory

For our reactive navigation controller, we implemented the velocity potential flow functions for two desired movements. Firstly, we incorporated the uniform flow equation to describe the flow of the vehicle towards the goal position. The attractive flow is set to travel at the maximum linear velocity set by the user. Figure 4.16 shows a differential drive mobile robot moving with the attractive flow towards the goal position.

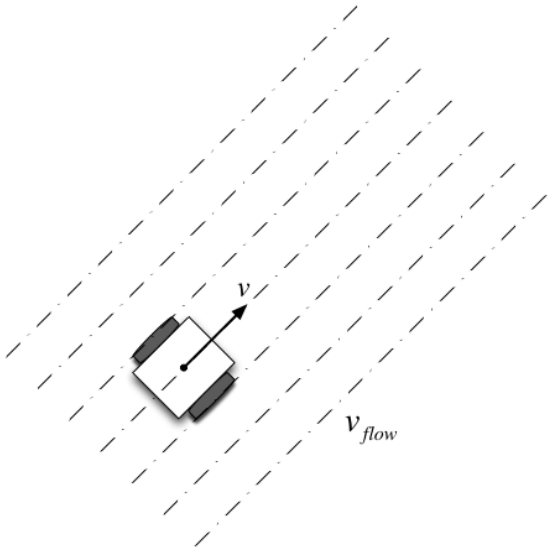


Figure 4.16: Attractive velocity flow

Secondly, we used the spiral vortex (source and vortex) potential flow function to push the robot away and around the obstacles. Figure 4.17 shows the path of the robot as it approaches the obstacle, and changes course to avoid collision, along the flowlines of the spiral vortex.

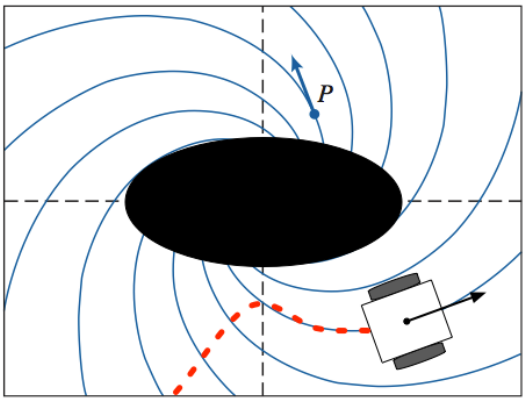


Figure 4.17: Repulsive spiral vortex

The normal and tangential velocities of the robot as it travels along the sprial vortex can be calculated from the cylindrical velocity equations. The velocity potential function for the spiral vortex was found to be:

$$\phi = -\frac{q}{2\pi}\ln r - \frac{K}{2\pi}\theta \tag{4.43}$$

By applying the cylindrical velocity equations from section 4.1.2, the normal and tangential velocities were calculated to be:

$$V_r = \frac{1}{r}\frac{\delta\phi}{\delta r} \tag{4.44}$$

$$V_r = -\frac{q}{2\pi}\left(\frac{1}{r}\right) \tag{4.45}$$

$$V_r = -\frac{A}{r} \tag{4.46}$$

$$V_\theta = -\frac{1}{r}\frac{\delta\phi}{\delta\theta} \tag{4.47}$$

$$V_\theta = \frac{1}{r}\left(-\frac{K}{2\pi}\right) \tag{4.48}$$

$$V_\theta = -\frac{B}{r} \tag{4.49}$$

Where A and B are constants.

4.1.4 Initial Parameters

When designing the algorithm for the reactive navigation controller with the velocity potential method, it was decided that the vehicle would flow towards the goal using a uniform flow field. The vehicle would travel on a straight line path from the initial (x, y) coordinate to the goal (x, y) coordinate. If the vehicle was not initially pointing at goal, it would continue to travel at the same linear velocity, but would also rotate slowly until it pointed in the direction of the goal. When the vehicle entered the region of interest around the goal (dashed circle), it was designed to

automatically slow down and then stop at the goal position. FIgure 4.18 shows the mobile robot velocity and angle parameters, the distance to the goal ρ_G, and the region of interest around the goal position. The robot parameters are listed below:

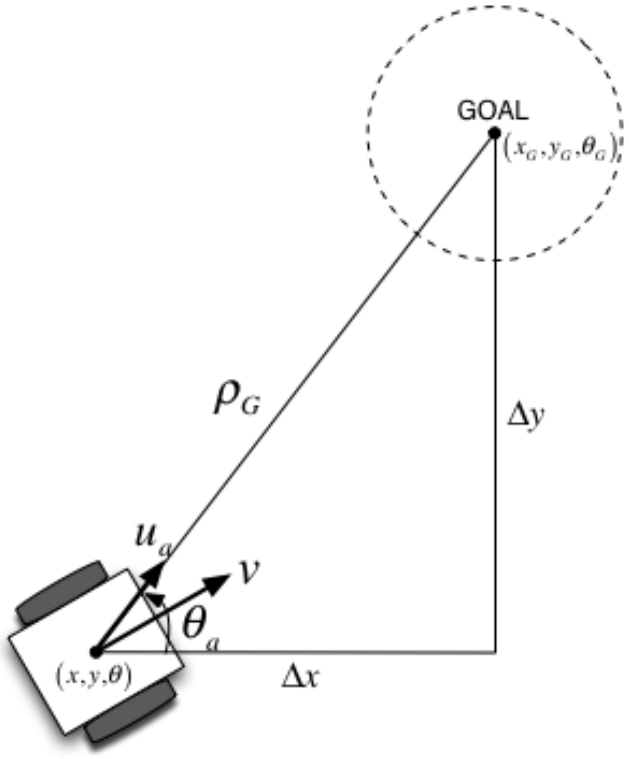


Figure 4.18: Attractive velocity variables

Vehicle current position

$$(x, y, \theta) \tag{4.50}$$

Goal position

$$(x_G, y_G, \theta_G) \tag{4.51}$$

Maximum linear and angular velocity (chosen by the user)

$$v_{max}, \omega_{max} \tag{4.52}$$

The first step in the procedure is to calculate the distance and angle variables.

Distance from the current position to the goal position

$$\rho_G = \sqrt{(x_G - x)^2 + (y_G - y)^2} \tag{4.53}$$

Desired vehicle angle to goal

$$\theta_a = \tan^{-1}\left[\frac{(y_G - y)}{(x_G - x)}\right] \tag{4.54}$$

Let $\Delta\theta$ be the difference between the desired angle and the current vehicle angle

$$\Delta\theta_a = \theta_a - \theta \tag{4.55}$$

4.1.4.1 Calculating the attractive velocity

The goal radius, the critical radius around the goal position ρ_{GR}, is a distance set by the user. It is used in the relation $f(\rho_G, \rho_{GR})$ to slow down and eventually stop the vehicle at the goal position when it enters the goal radius.

$$f(\rho_G, \rho_{GR}) = e^{-\rho_G/\rho_{GR}} \tag{4.56}$$

We initally set the magnitude of the attractive velocity to be equal to the maximum speed of the robot.

$$|u_a| = v_{max} \tag{4.57}$$

The attractive component of velocity is then calculated as follows:

$$u_a = |u_a|(\cos\theta_a + j\sin\theta_a) \tag{4.58}$$

If there are no obstacles in view, the final velocity command is listed below.

$$v_a = k_a(1 - f(\rho_G, \rho_{GR}))u_a \tag{4.59}$$

The angular velocity is then found, at a check is made to make sure the the attractive angular velocity lies within the safe driving bounds for the robot unit.

$$\omega_a = k_a(1 - f(\rho_G, \rho_{GR}))\frac{\Delta\theta_a}{\Delta t} \tag{4.60}$$

4.1.4.2 Calculating repulsive velocity

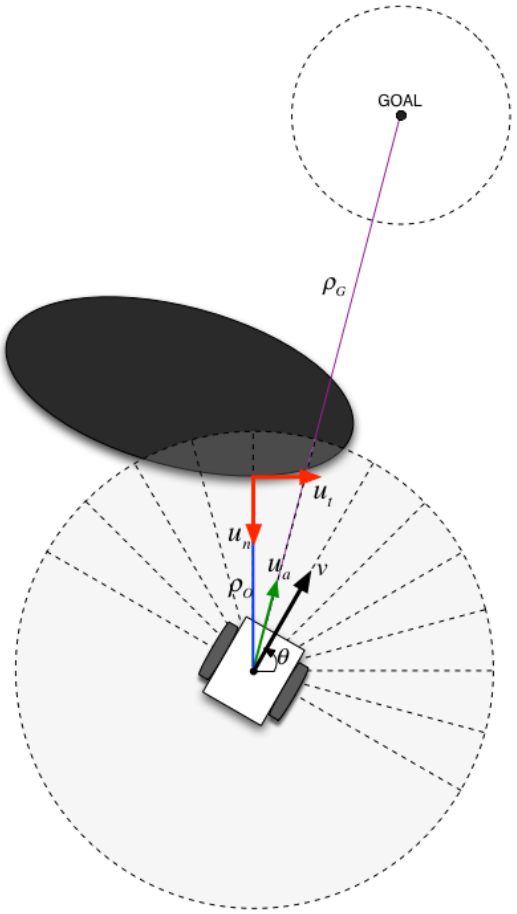


Figure 4.19: Attractive and repulsive velocity variables

The magnitude of the normal and tangent velocity vectors is determined from

equation 4.46 and equation 4.49.

$$|u_n| = |u_t| = \frac{1}{\rho_O} \tag{4.61}$$

As the robot comes closer to the obstacle, the following relation would slow the vehicle down, so that it would have time to react and navigate around the obstacle.

$$f(\rho_O, \rho_{OR}) = e^{-\rho_O/\rho_{OR}} \tag{4.62}$$

Normal vector angle

$$\theta_n = \theta_O - \pi \tag{4.63}$$

Tangent vector angle

$$\theta_t = \theta_n \pm \frac{\pi}{2} \tag{4.64}$$

The normal velocity command

$$u_n = |u_n|(\cos\theta_n + j\sin\theta_n) \tag{4.65}$$

The tangent velocity command

$$u_t = |u_t|(\cos\theta_t + j\sin\theta_t) \tag{4.66}$$

The resultant vector R is found by summing the normal vector with the tangent vector. The resultant vector has x-component R_x, and y-component R_y. The angle of the resultant vector is the new desired angle of the robot

$$\theta_{new} = \tan^{-1}\left(\frac{R_y}{R_x}\right) \tag{4.67}$$

The change in the obstacle angle is calculated

$$\Delta\theta_O = \theta_{new} - \theta \tag{4.68}$$

Velocity command around the obstacle

$$v_O = k_n u_n f(\rho_O, \rho_{OR}) + k_t u_t f(\rho_O, \rho_{OR}) \tag{4.69}$$

Final velocity command including the attractive and repulsive velocities.

$$v_R = k_a u_a (1 - f(\rho_G, \rho_{GR})) + v_O = k_n u_n f(\rho_O, \rho_{OR}) + k_t u_t f(\rho_O, \rho_{OR}) \tag{4.70}$$

$$\omega_R = k_o f(\rho_G, \rho_{GR}) \frac{\Delta\theta_O}{\Delta t} \tag{4.71}$$

4.1.5 Robot Navigation Around Complex Obstacles using the Velocity Potential Method

In our experiments, we will be testing the velocity potential flow algorithm on complex obstacle geometries such as the concave obstacle and the narrow passage obstacle. Figure 4.20 and figure 4.21 show the concave obstacle and narrow passage geometries, and diplay how the robot will tackle these two obstacles in the velocity potential approach.

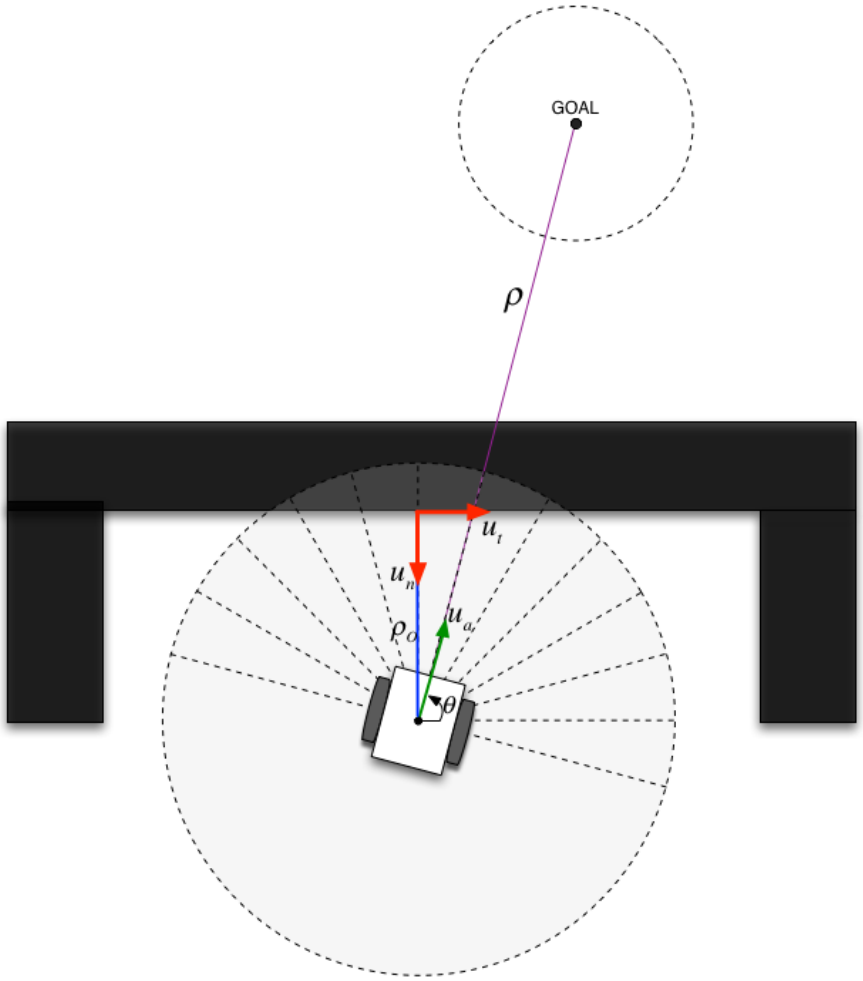


Figure 4.20: Robot approaching a concave obstacle

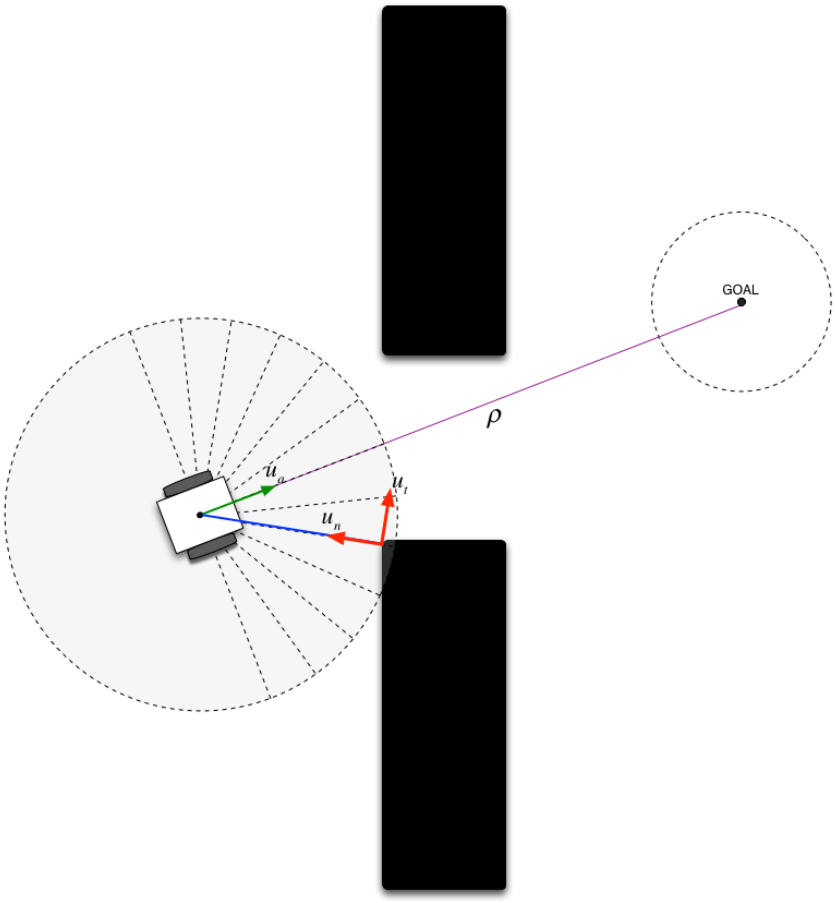


Figure 4.21: Robot approaching a narrow passage

4.2 Reconfiguring Geometric Formations using the Leader Follower Method

In Chapter 3: Literature Review, several formation control methodologies were examined, including: self-organizing systems, swarm systems, formations modeled after biological systems, virtual structures, and leader-following formations. In our analysis, we were attracted to the Leader-Follower formation control method. The leader-follower methodology, is a geometric formation controller that create formations in specific group shapes that are defined by the user.

The leader-follower formation will use geometry principles between the leader and follower robot, to get a follower vehicle to continuously track the leader. Figure 4.22 shows the initial pose of the leader and follower robots, and their variables ρ, α, and ψ, that describes the position and orientation of the follower with respect to the leader.

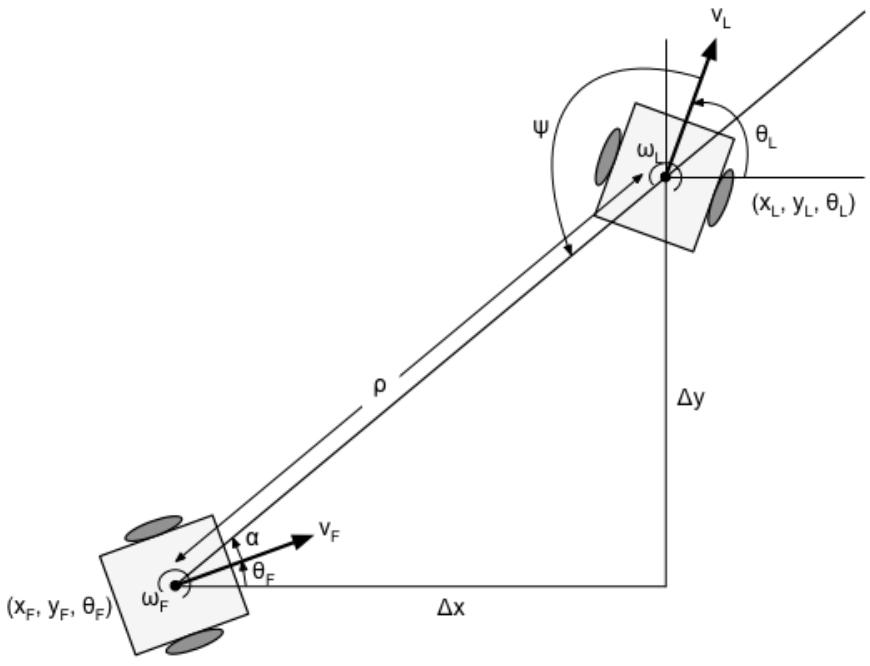


Figure 4.22: Leader-Follower geometry

Most of the calculations for the leader-follower formation are carried out in polar coordinates. The first step in the methodology is to convert Cartesian (x, y, θ) coordinates to polar (ρ, α, ψ) coordinates.

$$\rho = \sqrt{(x_L - x_F)^2 + (y_L - y_F)^2} \tag{4.72}$$

$$\alpha = \tan^{-1} \frac{(y_L - y_F)}{(x_L - x_F)} - \theta_F \tag{4.73}$$

$$\psi = \alpha + \theta_F - \theta_L + \pi \tag{4.74}$$

We then find the derivative of the distance variable $\dot{\rho}$

$$\rho^2 = (x_L - x_F)^2 + (y_L - y_F)^2 \tag{4.75}$$

$$2\rho\dot{\rho} = 2(x_L - x_F)(\dot{x_L} - \dot{x_F}) + 2(y_L - y_F)(\dot{y_L} - \dot{y_F}) \tag{4.76}$$

Let,

$$(x_L - x_F) = \rho\cos(\alpha + \theta_F) \tag{4.77}$$

$$(y_L - y_F) = \rho\sin(\alpha + \theta_F) \tag{4.78}$$

Sub into equation 4.76

$$\rho\dot{\rho} = (\dot{x_L} - \dot{x_F})\rho\cos(\alpha + \theta_F) + (\dot{y_L} - \dot{y_F})\rho\sin(\alpha + \theta_F) \tag{4.79}$$

$$\dot{\rho} = (\dot{x_L} - \dot{x_F})\cos(\alpha + \theta_F) + (\dot{y_L} - \dot{y_F})\sin(\alpha + \theta_F) \tag{4.80}$$

Let,

$$(\dot{x_L} - \dot{x_F}) = v_L \cos\theta_L - v_F \cos\theta_F \tag{4.81}$$

$$(\dot{y_L} - \dot{y_F}) = v_L \sin\theta_L - v_F \sin\theta_F \tag{4.82}$$

Sub into equation 4.80

$$\begin{aligned}\dot{\rho} = v_L \cos\theta_L \cos(\alpha + \theta_F) - v_F \cos\theta_F \cos(\alpha + \theta_F) \\ + v_L \sin\theta_L \sin(\alpha + \theta_F - v_F \sin\theta_F + v_L \sin\theta_L \sin(\alpha + \theta_F)\end{aligned} \tag{4.83}$$

Expand and simplify

$$\dot{\rho} = v_L \cos(\theta_L - \alpha - \theta_F) - v_F \cos(\theta_F - \alpha - \theta_F) \tag{4.84}$$

$$\dot{\rho} = v_L \cos(\theta_L - \alpha - \theta_F) - v_F \cos(-\alpha) \tag{4.85}$$

According to the geometry,

$$\theta_L - \alpha - \theta_F = \pi - \psi \tag{4.86}$$

So,

$$\dot{\rho} = -v_L \cos\psi - v_F \cos\alpha \tag{4.87}$$

Find $\dot{\alpha}$

$$\tan(\alpha + \theta_F) = \frac{y_L - y_F}{x_L - x_F} \tag{4.88}$$

$$\sin(\alpha + \theta_F)(x_L - x_F) = \cos(\alpha + \theta_F)(y_L - y_F) \tag{4.89}$$

Take the derivative

$$\begin{aligned}(\dot{\alpha} + \dot{\theta_F})\cos(\alpha + \theta_F)(x_L - x_F) + (\dot{x_L} - \dot{x_F})\sin(\alpha + \theta_F) \\ = -(\dot{\alpha} + \dot{\theta_F})\sin(\alpha + \theta_F)(y_L - y_F) + (\dot{y_L} - \dot{y_F})cos(\alpha + \theta_F)\end{aligned} \tag{4.90}$$

Let,

$$(\dot{x_L} - \dot{x_F}) = v_L \cos\theta_L - v_F \cos\theta_F \tag{4.91}$$

$$(\dot{y_L} - \dot{y_F}) = v_L \sin\theta_L - v_F \sin\theta_F \tag{4.92}$$

$$(x_L - x_F) = \rho\cos(\alpha + \theta_F) \tag{4.93}$$

$$(y_L - y_F) = \rho\sin(\alpha + \theta_F) \tag{4.94}$$

Expand and simplify

$$\rho(\dot{\alpha} + \dot{\theta_F}) = v_L \sin(\theta_L - \alpha - \theta_F) - v_F \sin(-\alpha) \tag{4.95}$$

According to the geometry

$$\theta_L - \alpha - \theta_F = \pi - \psi \tag{4.96}$$

Sub in

$$\rho(\dot{\alpha} + \dot{\theta_F}) = v_L \sin\psi + v_F \sin\alpha \tag{4.97}$$

Then,

$$\dot{\alpha} = \frac{v_F \sin\alpha}{\rho} + \frac{v_L sin\psi}{\rho} - \dot{\theta_F} \tag{4.98}$$

Find $\dot{\psi}$

$$\psi = \alpha + \theta_F - \theta_L + \pi \tag{4.99}$$

Take the derivative

$$\dot{\psi} = \dot{\alpha} + \dot{\theta_F} - \dot{\theta_L} \tag{4.100}$$

$$\dot{\psi} = \frac{v_F \sin\alpha}{\rho} + \frac{v_L sin\psi}{\rho} - \dot{\theta_L} \tag{4.101}$$

In matrix form

Set

$$\omega_L = \dot{\theta_L} \tag{4.102}$$

$$\omega_F = \dot{\theta_F} \tag{4.103}$$

$$\begin{bmatrix} \dot{\rho} \\ \dot{\alpha} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} -\cos\alpha & 0 \\ \frac{sin\alpha}{\rho} & -1 \\ \frac{sin\alpha}{\rho} & 0 \end{bmatrix} \begin{bmatrix} v_F \\ \omega_F \end{bmatrix} + \begin{bmatrix} -\cos\psi & 0 \\ \frac{sin\psi}{\rho} & 0 \\ \frac{sin\psi}{\rho} & -1 \end{bmatrix} \begin{bmatrix} v_L \\ \omega_L \end{bmatrix} \tag{4.104}$$

4.2.1 Controller

Let ρ_d be the desired distance between the two vehicles, and let α_d be the desired angle between the follower's direction of motion an ρ

$$\dot{\rho} = -k(\rho - \rho_d) \tag{4.105}$$

$$\dot{\alpha} = -k(\alpha - \alpha_d) \tag{4.106}$$

Where $k > 0$

In matrix form

$$\begin{bmatrix} \dot{\rho} \\ \dot{\alpha} \end{bmatrix} = \begin{bmatrix} k & 0 \\ 0 & k \end{bmatrix} \begin{bmatrix} \rho - \rho_d \\ \alpha - \alpha_d \end{bmatrix} \tag{4.107}$$

Find the follower's velocity and angular velocity

$$\begin{bmatrix} \dot{\rho} \\ \dot{\alpha} \end{bmatrix} = \begin{bmatrix} -\cos\alpha & 0 \\ \frac{sin\alpha}{\rho} & -1 \end{bmatrix} \begin{bmatrix} v_F \\ \omega_F \end{bmatrix} + \begin{bmatrix} -\cos\psi & 0 \\ \frac{sin\psi}{\rho} & 0 \end{bmatrix} \begin{bmatrix} v_L \\ \omega_L \end{bmatrix} \tag{4.108}$$

$$\begin{bmatrix} -\cos\alpha & 0 \\ \frac{sin\alpha}{\rho} & -1 \end{bmatrix} \begin{bmatrix} v_F \\ \omega_F \end{bmatrix} = \begin{bmatrix} \dot{\rho} \\ \dot{\alpha} \end{bmatrix} - \begin{bmatrix} -\cos\psi & 0 \\ \frac{sin\psi}{\rho} & 0 \end{bmatrix} \begin{bmatrix} v_L \\ \omega_L \end{bmatrix} \tag{4.109}$$

$$\begin{bmatrix} v_F \\ \omega_F \end{bmatrix} = \begin{bmatrix} -\cos\alpha & 0 \\ \frac{sin\alpha}{\rho} & -1 \end{bmatrix}^{-1} \left(\begin{bmatrix} \dot{\rho} \\ \dot{\alpha} \end{bmatrix} - \begin{bmatrix} -\cos\psi & 0 \\ \frac{sin\psi}{\rho} & 0 \end{bmatrix} \begin{bmatrix} v_L \\ \omega_L \end{bmatrix} \right) \tag{4.110}$$

$$\begin{bmatrix} v_F \\ \omega_F \end{bmatrix} = \begin{bmatrix} -\frac{1}{\cos\alpha} & 0 \\ -\frac{\tan\alpha}{\rho} & -1 \end{bmatrix} \left(\begin{bmatrix} \dot{\rho} \\ \dot{\alpha} \end{bmatrix} - \begin{bmatrix} -\cos\psi & 0 \\ \frac{sin\psi}{\rho} & 0 \end{bmatrix} \begin{bmatrix} v_L \\ \omega_L \end{bmatrix} \right) \tag{4.111}$$

Thus, the follower velocity can be calculated based on the leader velocity and angular velocity, and distance and angle variables ρ and α

$$\begin{bmatrix} v_F \\ \omega_F \end{bmatrix} = \begin{bmatrix} \frac{k}{\cos\alpha} & 0 \\ \frac{k\tan\alpha}{\rho} & k \end{bmatrix} \begin{bmatrix} \rho - \rho_d \\ \alpha - \alpha_d \end{bmatrix} + \begin{bmatrix} -\frac{\cos\psi}{\cos\alpha} & 0 \\ -\frac{\tan\alpha\cos\psi}{\rho} + \frac{\sin\psi}{\rho} & 0 \end{bmatrix} \begin{bmatrix} v_L \\ \omega_L \end{bmatrix} \tag{4.112}$$

4.3 Self-Organizing Group Formations using the Velocity Potential Method

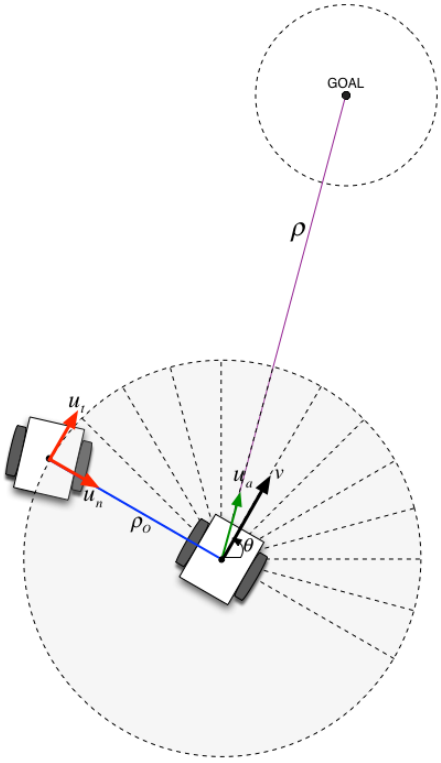


Figure 4.23: Self-organizing collision avoidance

In the self-organizing formation, we will be expanding the velocity potential flow method to handle multiple vehicles moving as a group. Figure 4.23 shows the inter-robot interaction when a robot enters into the sensor radius of another robot. The robot treats the neighbor robot as an obstacle and determines appropriate normal and tangential velocities to navigate around the neighbor robot.

The self-organizing velocity potential algorithm will be designed to be scalable to accommodate any number of robots, from small to large groups.

Chapter 5

Hardware Architecture

The WiRobot X80 platform from Dr. Robot Inc. is a differential drive mobile robot that is designed to provide a user friendly programming environment for hobbyists, students, and researchers. The X80 platform consists of actuators, sensors, and processors to facilitate autonomous robot motion. The X80 system allows developers to quickly test motion algorithms for multi-agent networked robots

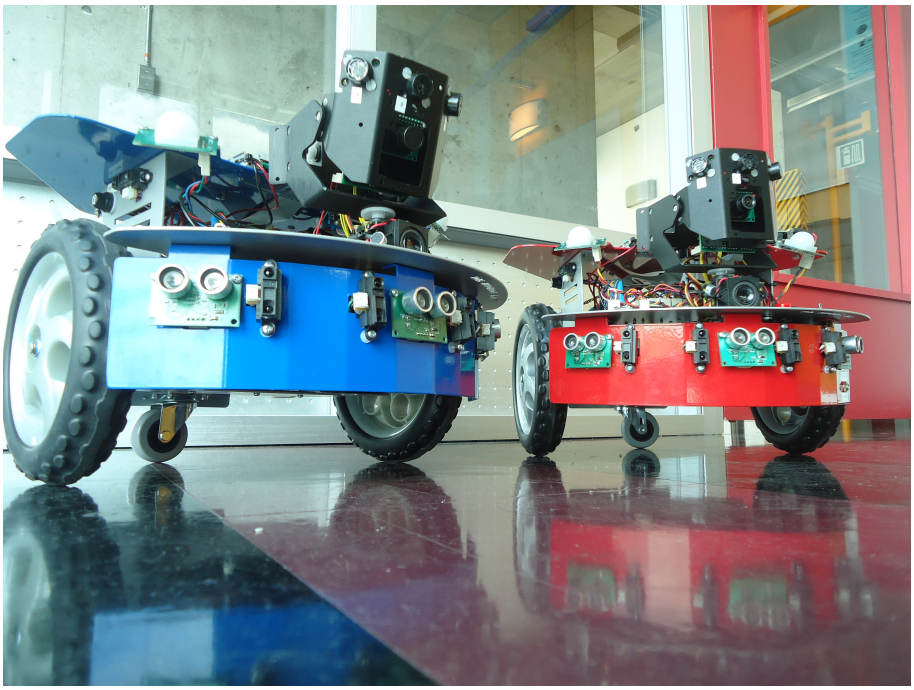


Figure 5.1: X80 mobile robots

This chapter describes all of the hardware components on the X80 mobile robot platform. Section 5.1 gives an overview of the X80 platform, its geometry, and design. Section 5.2 describes the various sensors on the X80 unit. Section 5.3 discusses the actuators for the robot, two 12V DC motors. Section 5.4 outlines the power supply for the robot. Section 5.5 describes the networking setup that enables a personal computer to control all robots via a local WiFi connection. Finally, section 5.6 describes the onboard microcontrollers that dictate low-level control for the mobile robot. Figure 5.1 shows a picture of two X80 mobile robot units from Dr. Robot Inc.

5.1 The X80 Mobile Robot Platform

The X80 platform is a differential drive mobile robot, equipped with sensing and intelligence to achieve autonomous motion. The X80 robots are designed to work in single or multi-robot applications, and operate best in flat terrain environments. The robots are fast, strong, lightweight, and nimble.

Figure 5.2[63] shows a CAD drawing of the front and back view of the X80 platform. The figure outlines the various components of the robot including: range sensors (sonar and IR), human sensor, camera, WiFi antenna, and battery location on the robot unit.

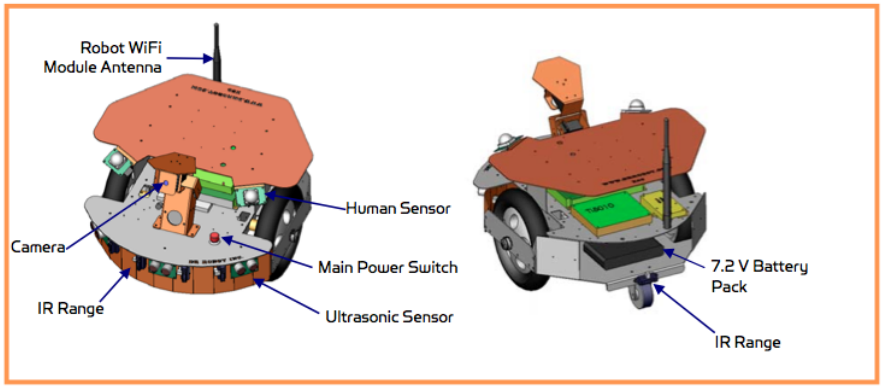


Figure 5.2: The X80 mobile robot and its components

The X80 robot work very well in experimentation in multi-agent systems, since they are designed to operate in a networked setup. Figure 5.3[63] show how three X80 robots all connect to a personal computer (PC) through a wireless network.

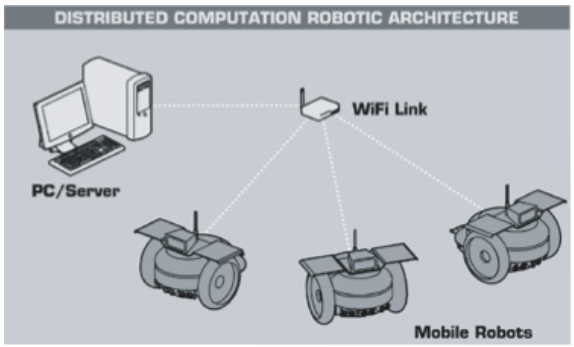


Figure 5.3: Distributed computation and robotic architecture

The WiRobot X80 is designed with a Distributed Computation Robotic Architecture and System (DIRAS) technology, which offloads most of the computation and storage intensive tasks to a personal computer. The PC and X80 robot communicate via a wireless network connection supporting a 100 kbps data communication rate. High-level schemes such as teleoperation, navigation, reasoning, learning, recognition, and image processing routines are programmed and executed on the PC remotely. The block diagram in figure 5.4[63] shows the link between the control program written on the PC and the microcontrollers that executes the program on the robot.

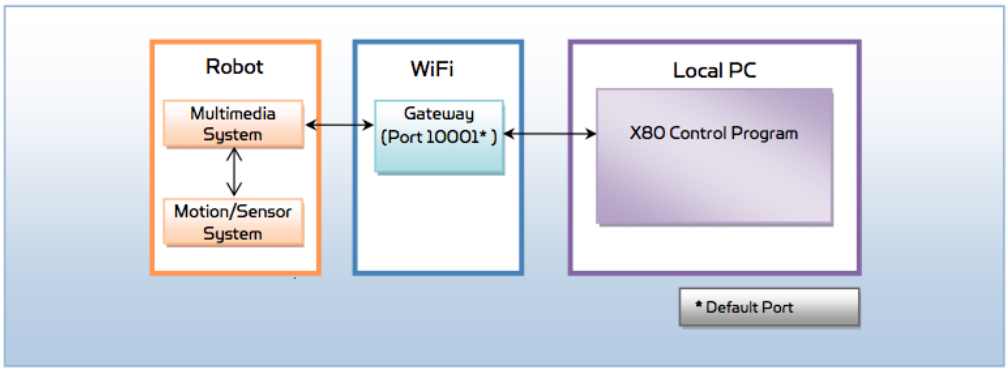


Figure 5.4: Block diagram - robot and local PC network connection

The X80 robot has a top speed of 1 m/s. It is powered by two 12 Volt DC motors that supply 300 oz-inches of torque to the 7 inch wheels. The vehicle is able to move forwards and backwards; when the wheels rotate at equal speeds, in opposite directions, the robot rotates on the spot. Along with the two main wheels, a third castor wheel is added at the rear of the robot to provide added stability. Two high-resolution quadrature encoders, mounted on each wheel, provide high-precision measurement and control of wheel movement. The vehicle is powered by two 3700 mAh nickelmetal hydride (NiMH) batteries.

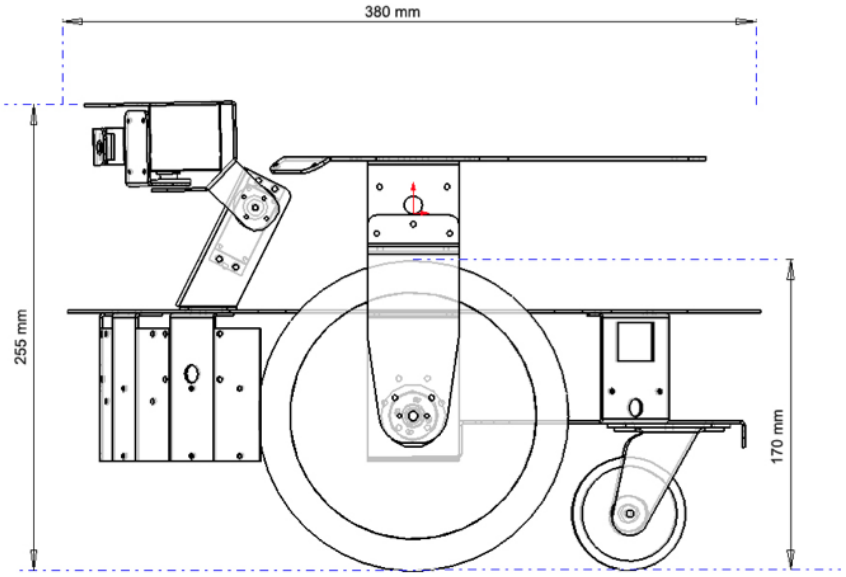


Figure 5.5: Side view of X80 robot with dimensions

The X80 platform measures 38 cm in diameter and 25 cm in height. The platform weighs a total of 3.5 kg, and can support an additional payload of 10 kg. Designers of the X80 made two levels for hardware placement: many of the functional elements are placed on the lower deck, leaving space on the upper deck to carry additional devices such as a laptop. Figure 5.5[63] shows a side view of the X80 mobile robot platform along with its dimensions.

5.2 Sensors

Sensors are the eyes and ears of the mobile robot. They are crucial to the operation of autonomous robots in unknown and dynamic environments, where complete a priori information is impossible to generate before the robot mission. This is especially the case in multi-robot experiments, where the robots must navigate around obstace and simultaneously avoid collisions with each other.

The X80 robot has a series of exteroceptive sensors that acquire information from the robot's surrounding environment. These sensors can be characterized as either passive or active. Passive sensors measure ambient environmental energy entering the sensor. While active sensors emit energy into the environment, and then measure the environmental reaction. The passive sensors on the X80 platform include: human sensors, camera, microphone, and a temperature sensor. Furthermore, the active sensors on the X80 include: ultrasonic range sensors, and infrared range sensors. The X80 robot also has proprioceptive sensors that measure internal values of the robot unit. The X80's proprioceptive sensors include: a tilt sensor to measure robot inclination, and quadrature encoders to measure the vehicle's speed and position.

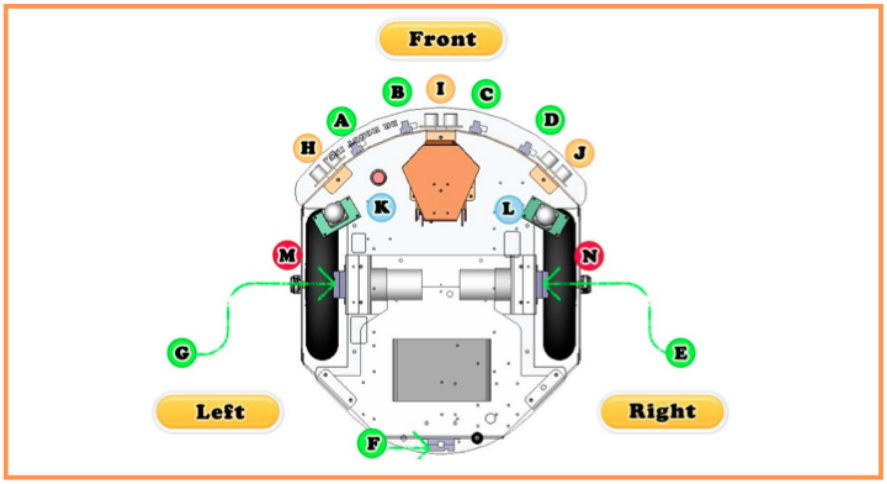


Figure 5.6: X80 top view sensor locations

Figure 5.6[63] shows a top view of the X80 unit, along with the positions of the ultrasonic range sensors, infrared range sensors, human sensors, and quadrature encoders. For our platform, three extra ultrasonic range sensors were installed at the rear of the X80 unit, at the left rear, middle rear, and right rear positions. Table 5.1[63] list the sensor module associated with the letters marked in figure 5.6.

Sensor Module	**Location**
Ultrasonic 1	H - Left front
Ultrasonic 2	I - Middle front
Ultrasonic 3	J - Right front
Human Sensor 1	K - Left front
Human Sensor 2	L - Right front
Infrared Range Sensor 1	A - Front left
Infrared Range Sensor 2	B - Front middle
Infrared Range Sensor 3	C - Front middle
Infrared Range Sensor 4	D - Front right
Infrared Range Sensor 5	E - Right
Infrared Range Sensor 6	F - Rear
Infrared Range Sensor 7	G - Left
Quadrature Encoder 1	M - Left, use channel 1
Quadrature Encoder 2	N - Right, use channel 2

Table 5.1: Sensor Modules

This section will examine all of the sensors on the X80 robot. It will begin with a look at the exteroceptive active sensors on the platform, which are the dominant sensors used in our experimentation. The section will then discuss exteroceptive passive sensors and proprioceptive sensors.

5.2.1 Exteroceptive Sensors - Active Sensors

5.2.1.1 Ultrasonic Range Sensors

Ultrasonic range sensors are used in mobile robotics for map building, obstacle detection, collision avoidance, and general-purpose distance detection. The ultrasonic range sensor module on the X80 transmits packets of ultrasonic pressure waves, and

measures the time it takes for wave packets to reflect and return to the receiver. By detecting the propagation time of the sonic wave between the sensor and the object (if any) in the path of the wave, the controller is able to calculate the distance[53].

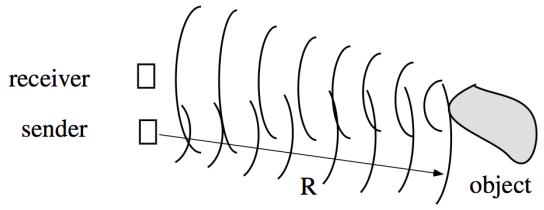


Figure 5.7: Ultrasonic pressure wave reflection

The sonar sensor module transmits packets at a frequency of 40 kHz. The sound wave propagation speed v can be calculated as a function of air temperature T (degrees Celsius) by the following formula:

$$v = 331.5 + 0.6 \cdot T[m/sec] \tag{5.1}$$

The distance to object (in meter) D can be obtained as function of the reflection time t and the velocity v:

$$D = \frac{t \cdot v}{2}[m] \tag{5.2}$$

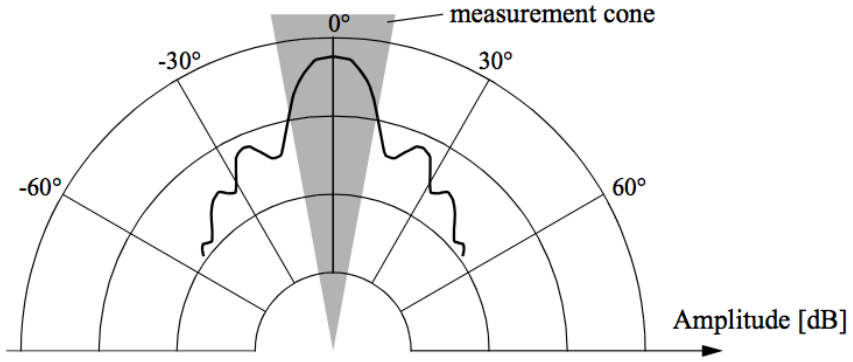


Figure 5.8: Sonar sensor intensity distribution

The ultrasonic range sensor module comes plug-and-play in the WiRobot system, allowing for quick and easy setup and replacement if needed. The sensor can detect ranges from 8 cm to 255 cm, and has a field of view of 10 degrees. Figure 5.8 shows the sensor intensity distribution over the field of view. The ultrasonic range sensor module is shown in figure 5.9, and a view of the sensor installed on the X80 platform is shown in 5.10.

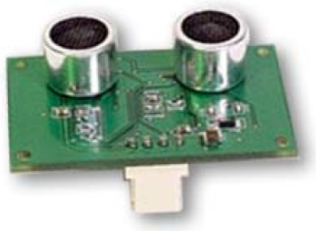


Figure 5.9: Ultrasonic Range Sensor Module

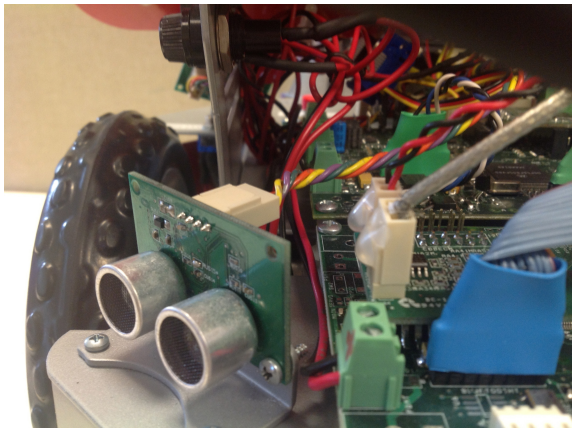
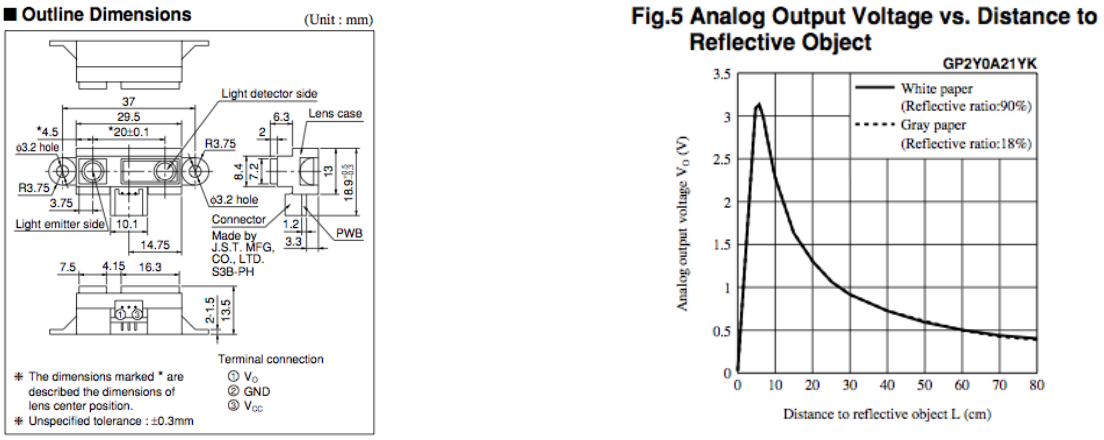


Figure 5.10: Sonar module on the X80 platform

5.2.1.2 Infrared Range Sensors

In mobile robotics, infrared range sensors are generally used as proximity sensors, that detect nearby obstacles. On the X80 platform, the infrared sensor developed by Sharp has a range of 10 cm to 80 cm, and a field of view of 5 degrees. Distance measurements to a reflective object are calculated as a function of the analog output

voltage from the sensor, as seen in figure 5.11b. FIgure 5.12 shows a close-up view of an ultrasonic and two infrared sensor on the front of the X80 unit[63].



(a) (b)

Figure 5.11: (a) Infrared sensor dimensions, and (b) calculating distance as a function of voltage

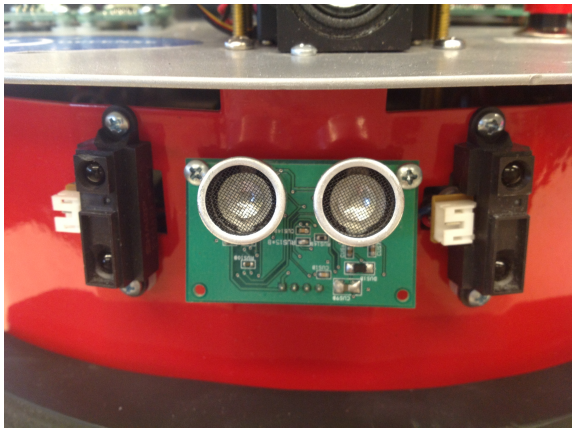


Figure 5.12: Ultrasonic sensor (center) and two infrared sensors (left and right)

5.2.2 Exteroceptive Sensors - Passive Sensors

5.2.2.1 Human Sensor

The human motion sensor module incorporates pyroelectric infrared sensors, to detect infrared energy radiating from a human body. Objects that generate heat also

generate infrared radiation; the infrared radiation generated by a human is strongest at a wavelength of 9.4 micrometers. This sensor is able to detect a human presence in the range of 5 meters. Furthermore, a moving human presence can also be detected, and the direction of movement can be detected in the range of 1.5 meters. Figure 5.13 shows the human sensor module on the X80 robot. Figure 5.14 shows the operation overview of the human sensor and its geometry[63].

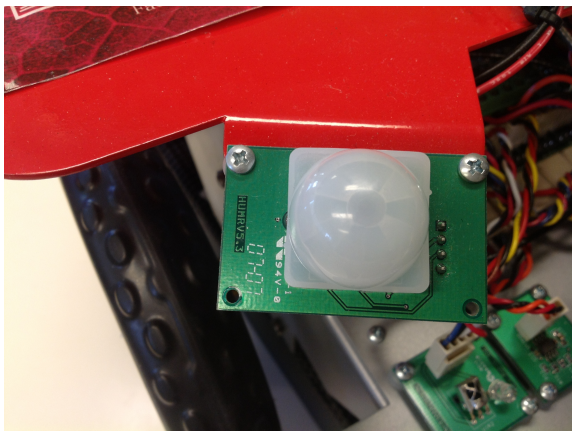
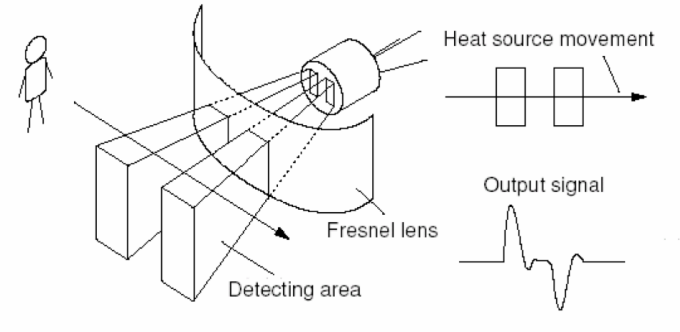
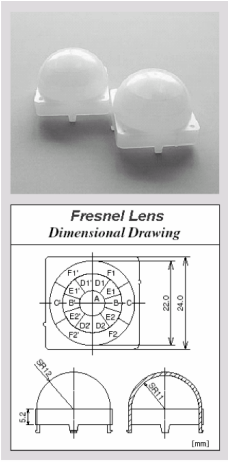


Figure 5.13: Human sensor on the X80



(a)



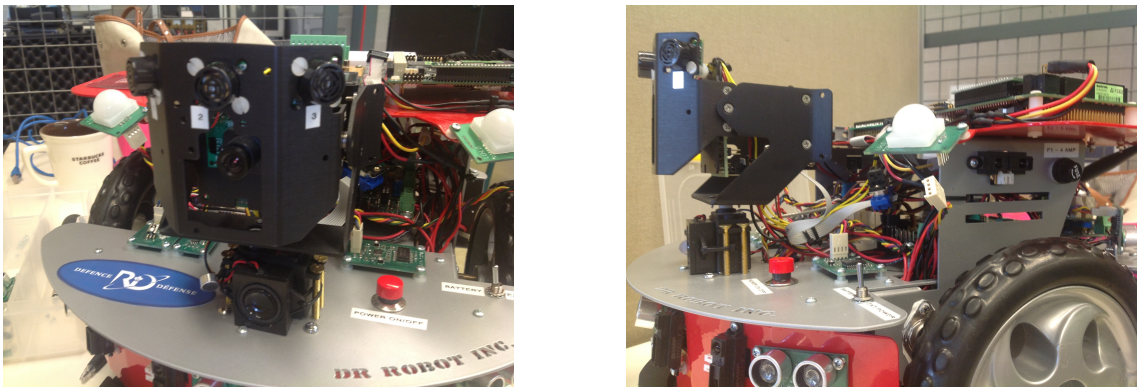
(b)

Figure 5.14: (a) Human sensor operation overview, and (b) human sensor fresnel lens

5.2.2.2 Camera

The X80 mobile robot platform is equipped with a CMUcam3 CMOS (complementary metal-oxide semiconductor) camera. The camera comes complete with an embedded computer vision system; custom C code, for computer vision algorithms, can be developed straight onto the CMUcam3 for optimal processing time. The camera mounting is designed to pan and tilt independently for optimal movement.

The CMUcam3 is a hardware platform coupled with an open source development environment. It is targeted toward users that are already familiar with basic image processing, and who are comfortable working with microcontroller programming. Figure 5.15 shows the front and side view of the CMUcam3 on the X80 platform.



(a) (b)

Figure 5.15: (a) Camera front angle (b) camera side angle

5.2.2.3 Microphone

The Audio Codec and Audio Power Amplifier Module is used as an audio input/output interface in the WiRobot system. The on-board codec provides high resolution signal conversion from digital-to-analog (D/A) and from analog-to-digital (A/D), and with the on-board audio output power amplifier and the microphone preamp in the codec, the external speaker and microphone can be directly connected to the board. Audio Codec and Audio Power Amplifier Module can be used in applications

such as: audio input/output for robots systems, voice and speech recognition, and voice and audio playback. Figure 5.16 shows the microphone location on the X80 robot. Furthermore, figures 5.17[63] and 5.18 display the Audio Codec and Audio Power Amplifier Module and its placement on the X80 robot.

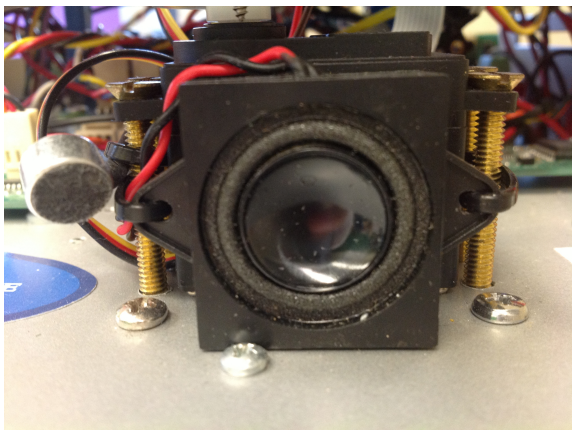


Figure 5.16: Microphone and speaker location on the X80

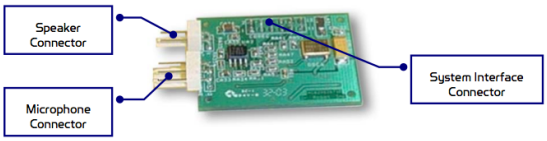


Figure 5.17: Microphone board with labels

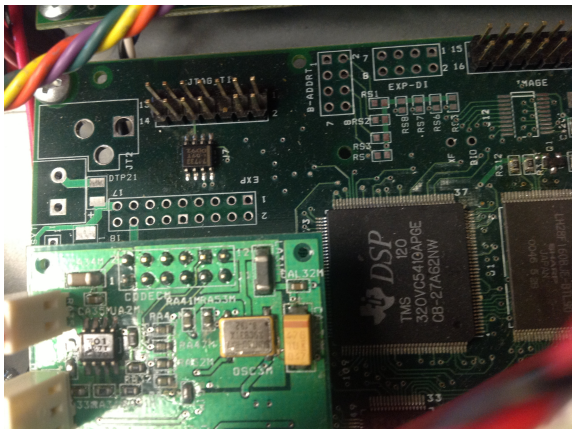


Figure 5.18: Microphone module on the X80 unit

5.2.2.4 Temperature Sensors

The Ambient Temperature Sensor Module is a CMOS temperature sensor that generates a linear voltage signal according to the ambient air temperature. The temperature sensor can be used for high-precision thermal control. Figure 5.19[63] shows the curve used to determine temperature values from voltage inputs. Moreover, figure 5.20 shows the position of the Ambient Temperature Sensor Module on the X80 robot.

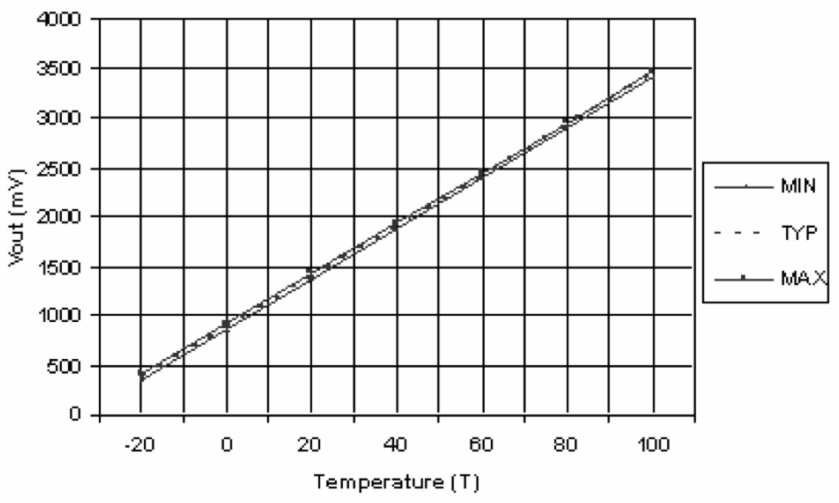


Figure 5.19: Calculating temperature as a function of voltage

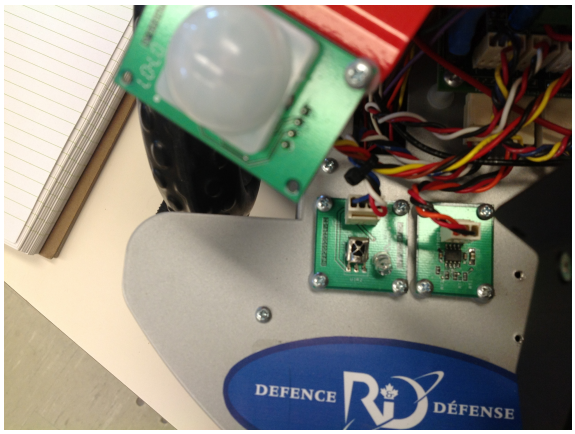


Figure 5.20: IR receiver (left), and temperature sensor (right)

5.2.3 Proprioceptive Sensors

5.2.3.1 Tilt Sensor

The tilt sensor is an inclinometer that measures the incline of the robot while it is moving in the terrain. It measures whether the vehicle is moving up or down a steep incline, or if the vehicle is shifting side to side. Inclinometers are helpful in mobile robotics because alert the user if the robot is inclined too far and could topple over. Figure 5.21 shows the tilt sensor chip on the X80.

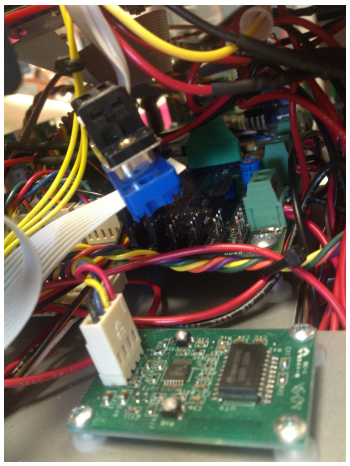


Figure 5.21: Tilt sensor module on the X80

5.2.3.2 Quadrature Encoders

Two high resolution 1200 cycles per revolution (CPR) quadrature encoders are mounted on each wheel to provide measurement and control of wheel movement. Quadrature wheel encoders measure distance by counting the number of ticks produced on a grid passing a barrier. Each quadrature encoder consists of two illumination and detection pairs placed at a 90 degree angle from each other. The addition of the second illumination and detection pair for the quadrature encoder provides significantly more information and improves the resolution of the system.

5.3 Actuators

The X80 is equipped with two wheels, using a differential drive steering model. Differential drive mobile robots offer a large range of mobility, due to their wheel setup. Both wheels are controlled independently from the other, and by coordinating the two different speeds, the robot can perform various maneuvers such as: spin in place, move in a straight line, move in a circular path, and follow prescribed trajectories.

The mobile robot will also be characterized by nonholonomic kinematic constraints that prevent the wheeled robots from moving sideways, perpendicular to the wheel movement, during its motion. Nonholonomic constraints limit the mobility of mobile robots in their workspace, and since sideway motion is prohibited, robots must move around obstacles in a smooth path with an appropriate velocity profile. The two-wheel differential drive mobile robots fare better than their four or six wheel counterparts in working with their nonholonomic motion constraints, since the differential drive robot can slow down and even stop and spin on place when difficult motions are required.

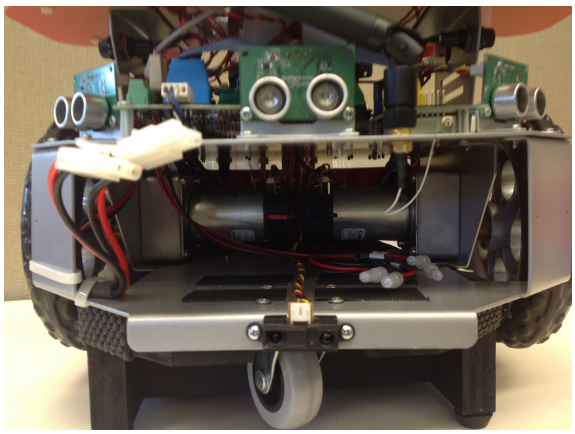
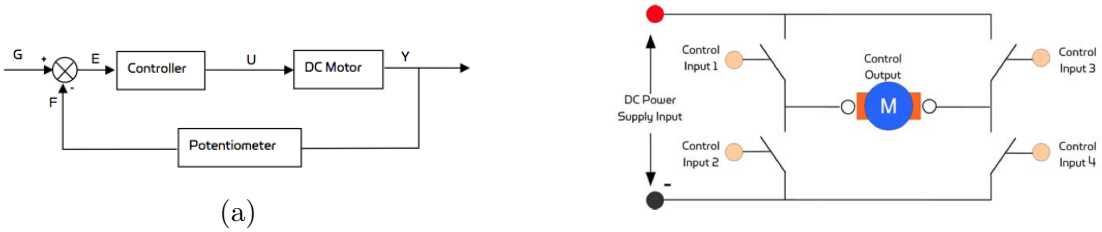


Figure 5.22: Back view of the X80 and its two DC motors

Each wheel is powered by a 12 V DC motor. Figure 5.22 shows the back view of the X80 and a look at the two motors. The DC motors supply 300 oz.inches of

torque to the 7 inch wheels, and yield a top speed 1 m/s.

The DC Motor Driver Module is a low-level controller designed to regulate motor commands in real-time. The controller module functionality is based on the feedback loop of figure 5.23a, and uses a three-channel H-bridge switching power amplifier, figure 5.23b to regulate control commands[63].



(a) (b)

Figure 5.23: (a) Block diagram of the motor control process (b) H-bridge switching device

The DC Motor Driver Module is located on top of and is directly connected to the Sense and Motion Controller board (see section 5.6). Desired velocity outputs from the user are sent from the Sense and Motion Controller board to the DC Motor Driver Module, in order to convert desired motor output to torque values. Furthermore, feedback loops within the module ensure that the torque outputs to the motor stay within a safe working range. Figure 5.24 shows the DC Motor Driver Module, and figure 5.25 shows the controller board location on the X80 robot.

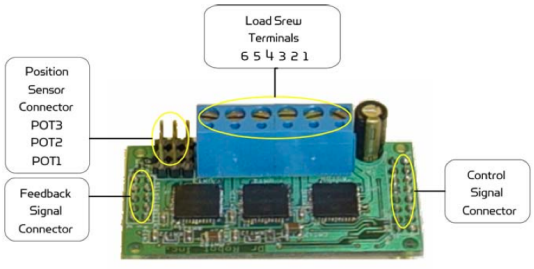


Figure 5.24: DC motor driver module board

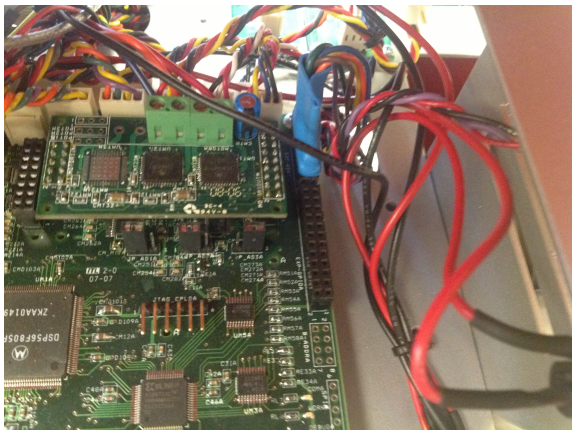
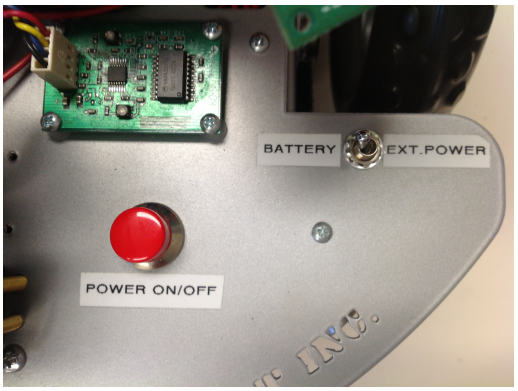


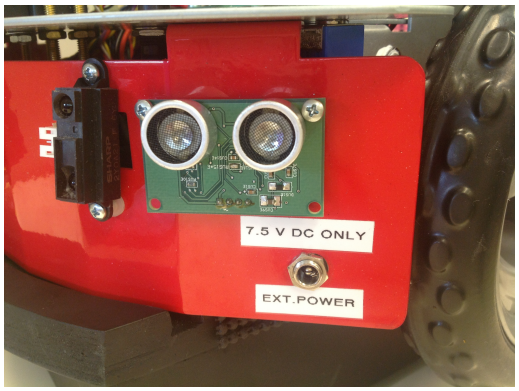
Figure 5.25: DC motor driver module board on the X80

5.4 Power

The X80 mobile robot requires electricity to power its many components. The unit can be powered by plugging it into an electric outlet with a special cable, or it can be powered by two nickel-metal-hydride (NiMH) batteries. Since mobility was a key aspect in our research, batteries were used to power the robot. The NiMH batteries supplied 7.2 V and 3700 mAh to the robot unit. With a full charge, the batteries last approximately 45 minutes.



(a)



(b)

Figure 5.26: (a) Power switch on the X80 (b) external power connection on the X80

Figures 5.26a and 5.26b show the power options and the DC external power port on the X80. Furthermore, figure 5.27 illustrates how the NiMH batteries connect the X80 system, and their location at the rear of the unit.

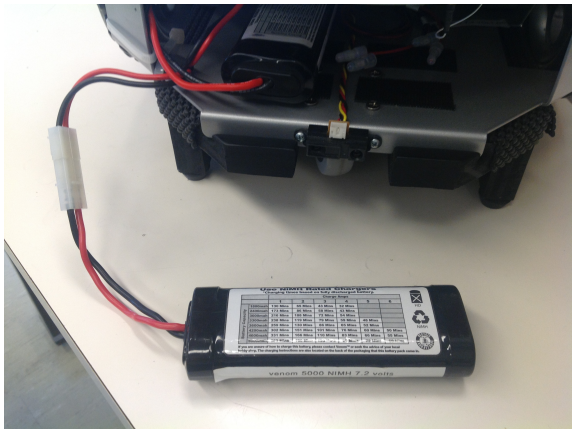


Figure 5.27: Battery connection on the X80

Figure 5.28: Wireless communication between the PC and X80 unit via WiFi

5.5 Networking

Information between the PC and X80 robots travel through a wireless connection. The connection runs at a data communication rate of 100 kbps, and allows for real-time updates to the robot's navigation and formation control parameters. The WiFi system on the X80 has a bandwidth of 11 Mbps and conforms with the IEEE 802.11 wireless standard. The system can upload all sensor data and encoder readings to a PC or server at rates in excess of 10 Hz. Furthermore, the WiFi 802.11 system with

dual serial communication channels, has a max rate of 912.4 Kbps per channel, and supports both UDP and TCP/IP protocol. Streaming audio transmits at a rate of 8 Hz, and streaming video at up to 4 frames per second (fps).

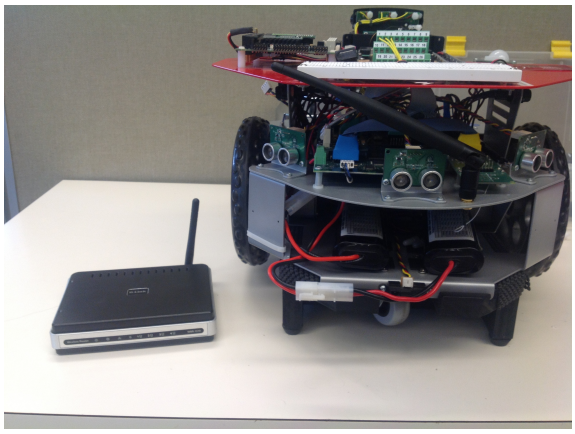


Figure 5.29: X80 robot and router

The networked system provides real-time control and access to the X80 unit without having to program high-level controllers onto its microcontroller. Figure 5.28 illustrates the wireless communication between the PC and X80 unit via WiFi, and figure 5.29 shows a picture of the X80 mobile robot and its router. Finally, figure 5.30 shows the wireless board located on the X80 unit that handles all of the networking commands.

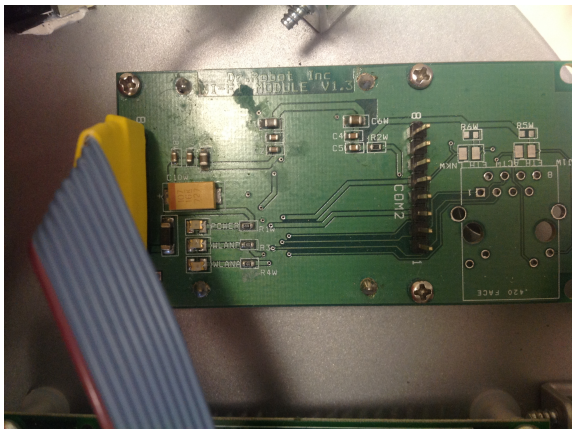


Figure 5.30: Wireless board

5.6 Processors

On the X80, two boards control all of the low-level functionality of the mobile robot: (1) the Sense and Motion Controller, and (2) the Multimedia Controller. The Sense and Motion Controller is the brain of the X80 robot, and as the name suggests, it is responsible for all of the sensing and motion tasks. The Multimedia controller handles all audio and video communications for the robot unit. High-level control for the robot can be communicated via a wireless network, see section 5.5, or it can be programmed onto a microcontroller. On the X80, a Kontron MOPSlcdLX microcontroller is installed for this use.

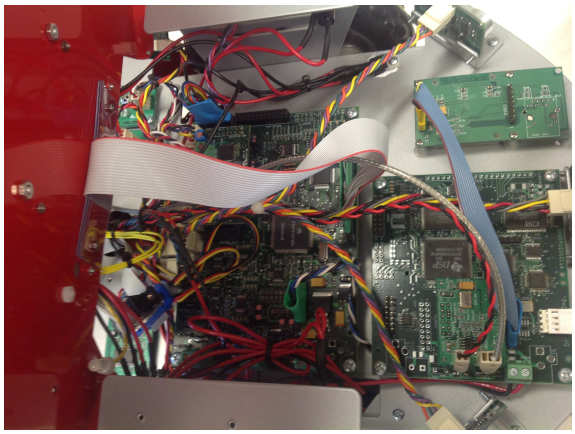


Figure 5.31: Sensing and motion controller board (left), multimedia board (right), and WiFi board (top)

This section will discuss the three boards that handle intelligence for the X80 robot. Figure 5.31 shows the three boards on the lower deck of the X80 unit. Moreover, figure 5.32[63] is a schematic of how the low-level controllers connect to the sensors and other components of the X80 robot.

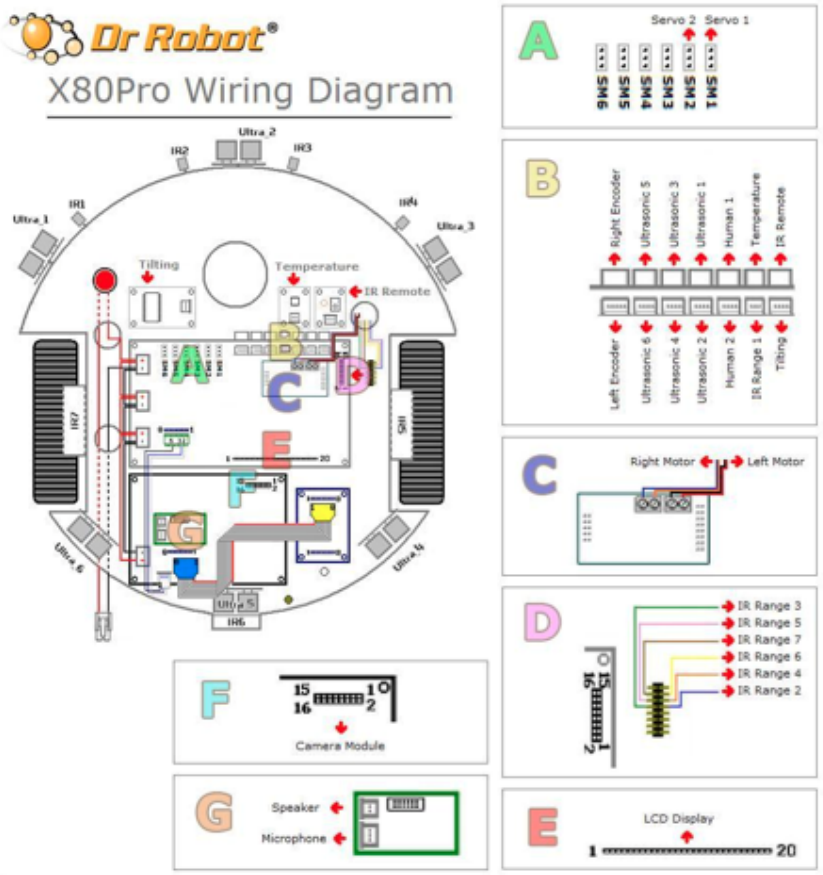


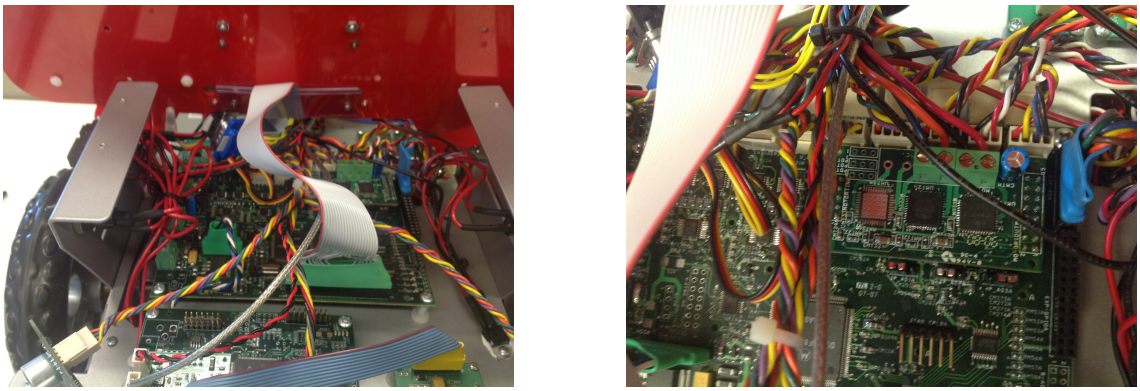
Figure 5.32: X80 wiring diagram

5.6.1 Sensing and Motion Controller

The sensing and motion controller board handles all sensing and actuator commands. It is the hub that collects all information about the robot's state and its local environment. Figure 5.33 shows the location of the sensing and motion controller board on the X80, and a close-up view of how the sensors connect to the board.

The purpose of this unit is gather information about the robot's state to send to the PC to make decisions, then when the PC returns desired velocity values, the sensing and motion controller board converts this information into data that the robot can process and execute. The sensing and motion controller unit is the most important component to the X80 system and is responsible for all of the movement

tasks for the robot. Figure 5.34 shows the sensing and motion controller board and labels all of its key components. Furthermore, figure 5.35 is a block diagram that demonstrates the functionality of the sensing and motion controller unit[63].



(a) (b)

Figure 5.33: (a) Sensing and motion controller board on the X80 platform, and (b) a close-up view of how the sensors connect to the sensing and motion controller board

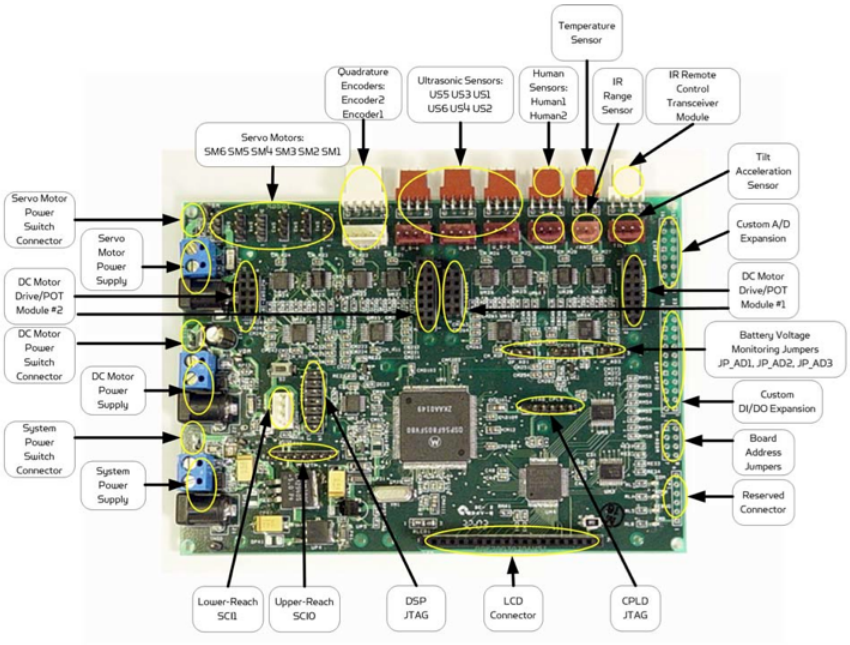


Figure 5.34: Sensing and motion controller board with labels

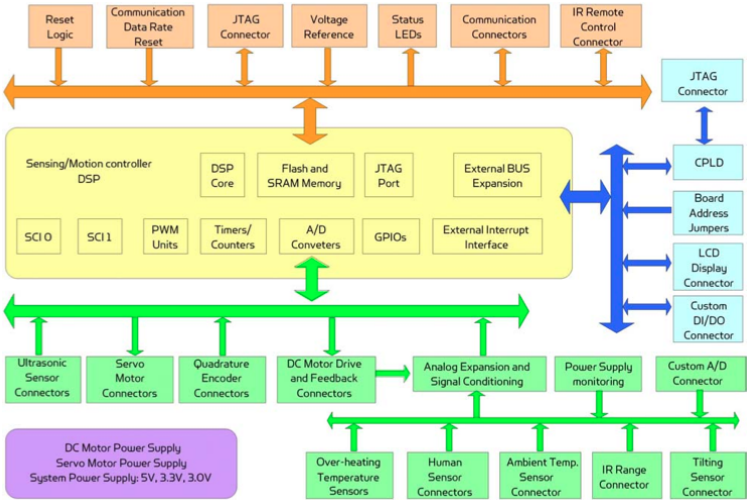


Figure 5.35: Sensing and motion controller block diagram

5.6.2 Multimedia Controller

The multimedia controller handles all of the media elements for the X80. Having a dedicated media controller is especially important for work with camera vision systems, where the processing time can be extensive. Figure 5.36 shows a the position of the multimedia controller on the X80 robot. Figure 5.37 labels the important components of the multimedia controller, and figure 5.38 displays of block diagram of the boards functional elements[63].

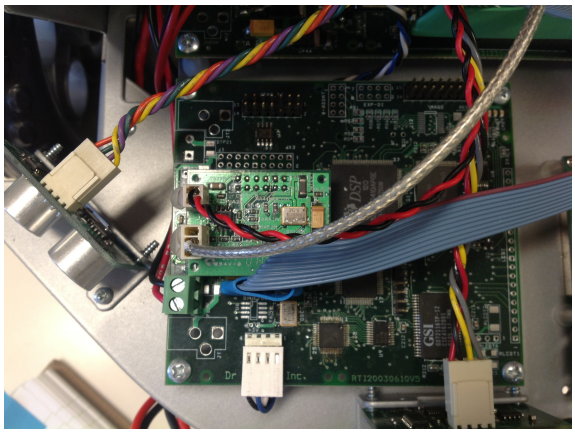


Figure 5.36: Multimedia controller chip on the X80

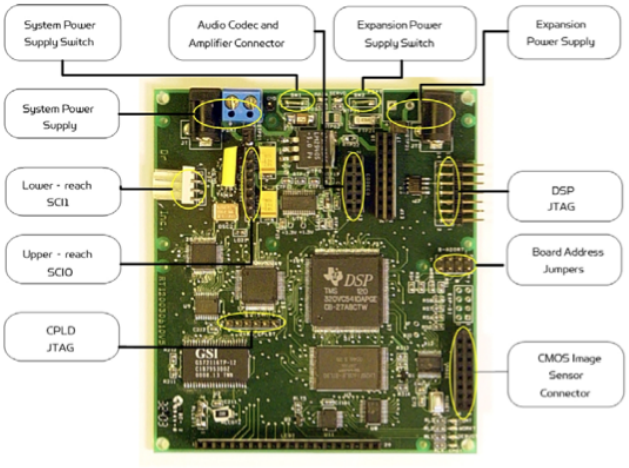


Figure 5.37: Multimedia controller board with labels

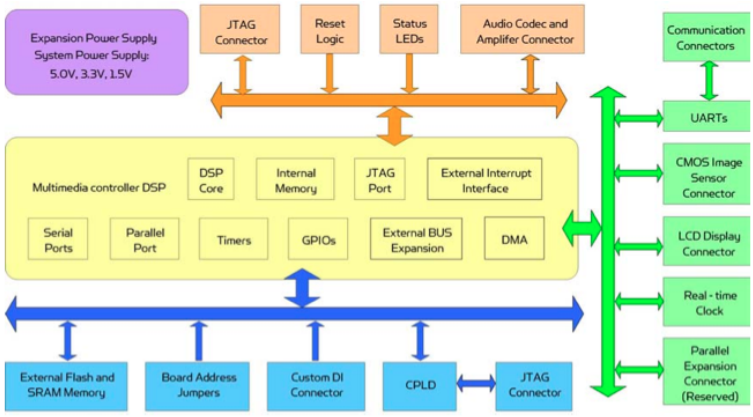


Figure 5.38: Multimedia controller block diagram

5.6.3 Programmable Microcontroller and Additional Inputs

The top deck of the X80 platform is equipped with addition inputs and computation for customized implementations. The Kontron programmable microcontroller lets users add high-level control schemes to dictate robot movement directly on the robot. This type of implementation is desirable after the testing phase, when controllers are proven to be effective, and faster processing time is required. Programs are written in the C language and are uploaded onto the board. Figure 5.39 shows the microcontroller on the X80 platform, and figure 5.40 shows a block diagram of its

northbridge and southbridge design[63]. Figure 5.41 displays additional inputs on the X80 including a breadboard and DB inputs. An LED screen can also be programmed by the user to output information while conducting experiments. Finally, figure 5.42 shows the joystick that was used in our experiments for teleoperation.

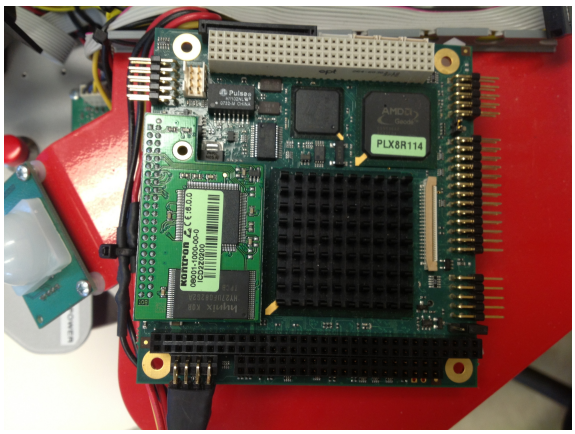


Figure 5.39: Kontron programmable microcontroller

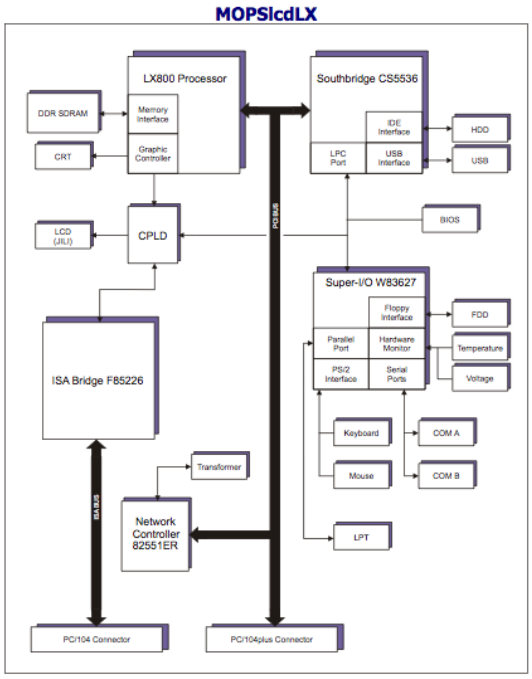


Figure 5.40: Kontron programmable microcontroller block diagram

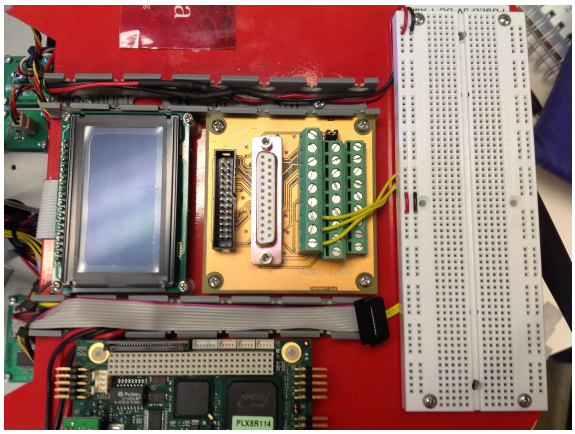


Figure 5.41: LED output screen (left), port inputs (middle), and breadboard input (right)

Figure 5.42: Joystick used for teleoperation with the X80

Chapter 6

Software Architecture

The X80 mobile robot is equipped with sensors, actuators, and processors to achieve autonomous motion. Its overall autonomy in dynamic environments is dependent on the successful implementation of high-level navigation controllers, that dictate the motion of the robots in real-time.

This chapter covers the software development required to build formation controllers for the X80 Mobile Robot Platform. Section 6.1 describes the design of the X80 platform, and the software-hardware interface. Section 6.2 goes through the benefits of using a prepackaged software framework for robot development and two popular frameworks are described. Finally, section 6.3 compares the two frameworks, and a framework is selected for the research.

6.1 The Software-Hardware Interface

On the X80 platform, software and hardware work together to steer the robots away from collisions and to keep desired formations throughout their movement. Software controllers can be applied to the X80 platform in one of two ways: (1) controllers are programmed in C and are uploaded directly onto its microcontroller, or (2) controllers are programmed in any modern programming language (C, C++,

C#, Java, Pyton, etc.) and messages are sent via a WiFi connection between your personal computer and the X80 robots. The networked setup of method 2 is an excellent architecture for research and development in robotics. This is the case since it allows for quick and easy testing of controller algorithms, without having to upload the new controller code onto its microcontroller each time a change is made.

In networked setup, robots use their sensors to inspect the environment, and sensor data is sent via WiFi to the personal computer where the navigation algorithm is stored. The computer calculates the linear and angular velocities required for the desired motion, and the message is sent back to the robot via WiFi. The robot then inputs velocity commands into its on-board lower-level controllers and calculates applicable torque values. Finally these torque commands are sent the differential drive motors. This process is repeated over every time step.

The networked setup for robot control is intended for prototyping purposes only, to quickly test if algorithms used in robot simulations have similar results when using the robot platform. For our experiments, the networked setup produced excellent results, with no perceivable time delay. However, for an actual implementation in industry, the first method of directly applying the controller algorithm onto the microcontroller is a more computationally efficient process.

6.2 Software Frameworks

Developing software from scratch to control a system of networked robots is a complex, time consuming, and error-prone task. It is far more beneficial for researchers to take advantage of existing software frameworks, designed for work in robotics, as a starting point to develop navigation controllers. A software framework reduces the time and energy required to program controllers onto robots by simplifying hardware access through the use of a hardware abstraction layer. The hardware abstraction

layer hides details about the robot's hardware by defining generic concepts such as sonar sensor, laser sensor, camera, etc., with each having its own standard interface. The navigation controller then does not need to know the details about any particular hardware component, as long as the component complies with its appropriate interface[5]. The details of making a particular component support the standard interface are handled by the driver, which can be programmed by either the hardware manufacturer or the researcher.

Software frameworks are an excellent tool for robot developers since they are scalable, reusable, efficient, fault-tolerant, and as projects gets larger the code can grow with it. Complex robotic applications are all based on frameworks, and these frameworks contain common functionality regarding robotic software and support many of the issues programmers would otherwise have to tackle themselves. A software framework improves the development process and lets developers focus their work on high-level programming to implement movement algorithms.

There are many software frameworks on the market today, and each framework focuses on different aspects such as robustness, efficiency, or ease of use. Currently there is no solution that is clearly superior to all others, so the selection of the software framework depends on the needs of the research laboratory, their budget, and software compliance with specific hardware components. The two most popular frameworks for robot development are Microsoft Robotics Developer Studio and the Player Project.

6.2.1 Microsoft Robotics Developer Studio

Microsoft Robotics Developer Studio (MRDS) is a robot development platform that runs within the Microsoft Visual Studios Integrated Development Environment. MRDS was released in 2008 with the goal of making robot developing easier and more efficient than it had been in the past. Programming robots in MRDS can be

done in one of two ways: (1) using the Microsoft Visual Programming Language, or (2) using the C# programming language.

The Visual Programming Language is a graphical user interface that creates robotic controllers in a visual and tactile way. Controllers are created by connecting blocks that represent hardware components with blocks that represent the logic or movement procedures, in a block diagram type of format[26]. The Visual Programming Language is geared towards hobbyists and students who are learning the robot programming process, to create simple robotic movements.

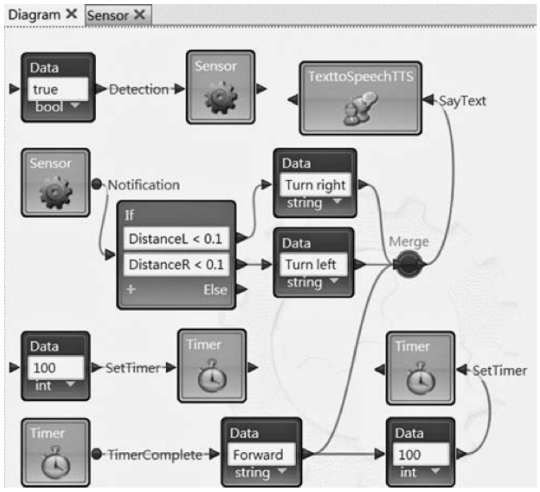


Figure 6.1: Visual programming language screen shot

For our navigation and control algorithms with the X80 robot, the Visual Programming Language did not meet our needs, and the second method, programming in C#, was approached. The developers at Dr. Robot Inc. created several methods and classes in C# that worked directly with the X80 mobile robot in the MRDS environment. Robots applications in MRDS consist of multiple services, which work together to achieve a common task of operating the robot. Services communicate by passing messages back and forth through ports, and manifest files written in XML store the values of hardware components. Concurrency and coordination runtime push services to work in parallel in an asynchronous manner, and allow for

multi-threading type programming[26].

Creating applications with Microsoft Robotics Developer Studio requires understanding of service-oriented architecture, decentralized software services, as well as XML. MRDS programming with the C# language is best suited for industrial large-scale robotics applications.

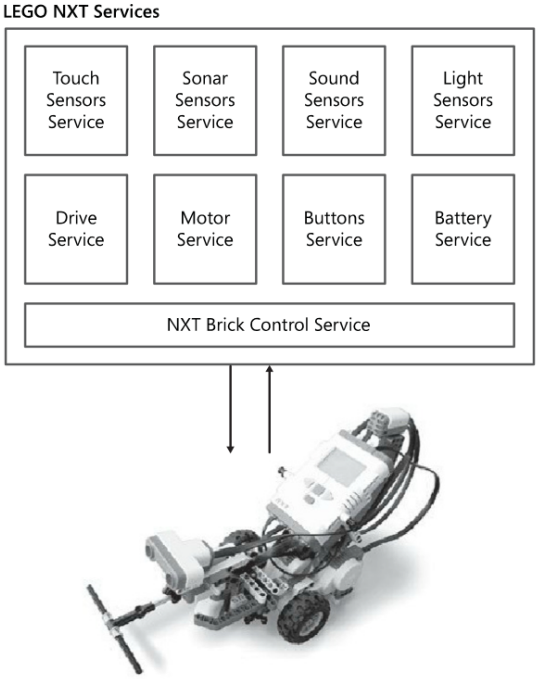


Figure 6.2: LEGO NXT services [26]

6.2.2 The Player Project

The Player Project is an open source robot development environment designed to work on the UNIX platform. Player is most commonly implemented in a Linux based operating system such as Ubuntu, and it has become the de facto standard in the open source robotics community. The software framework was developed in 1999 at the Robotics Research Lab of the University of Southern California to address an internal need for interfacing and simulation for multi-robot systems. In 2001 the project was released under the GNU public license and since then it has

been downloaded more than 100,000 times and is used in many labs and educational institutions worldwide[5].

The Player Projects consists of three major parts: (1) Player, which is the hardware abstraction layer that allows your code to talk to robot hardware components, (2) Stage, a plugin to Player, that creates a two-dimensional simulation of the robots, and (3) Gazebo for generating three-dimensional robot simulations. In general, the Stage simulator is designed to simulate large populations of robots with reasonable accuracy, while the Gazebos 3D simulator is designed with high accuracy and is suited for simulations with a small number of robots[43].

6.2.2.1 Player

Player is a robot device interface that provides a hardware abstraction layer for robotic devices. Player defines a set of standard interfaces, each of which is a specification of the ways that you can interact with some class of devices[62]. For example the position2d interface covers ground-based mobile robots, allowing them to accept commands to make them move and report their state. Many drivers support the position2d interface, and the job of the driver is to make the robot support the standard interface.

Player also provides transport mechanisms that allow data to be exchanged among drivers and control programs that are executing on different machines. The most common transport is the client/server TCP socket-based transport. In this setup, the Player server is executed with a configuration file that defines which drivers to instantiate and how to bind them to hardware. The drivers run inside the player server, often on multiple threads, and the users control program runs as a client to that server.

The Player server provides a convenient API (Application Programming Interface) to a broad range of commercial robots and robotic hardware. Its central

design goals were minimalism and simplicity concerning server and message protocol. Robot controls are implemented as clients of the Player server, and are sent over a network via WiFi. Client programs talk to Player over a TCP socket, reading data from sensors, writing commands to actuators, and configuring devices on the fly.

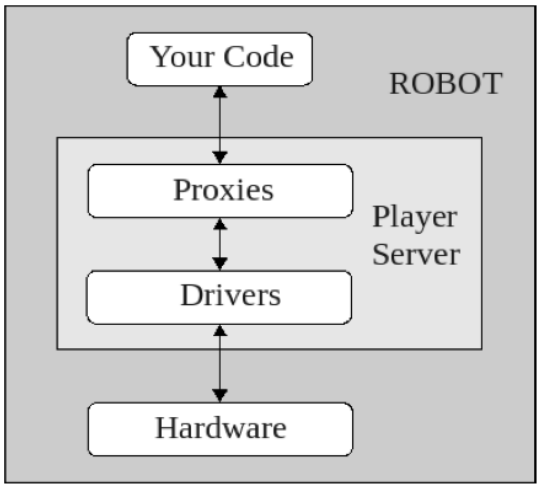


Figure 6.3: Player server [43]

Player is designed to be language and platform independent, and client programs can be run on any machine that has a network connection to the robot, and it can be written in any language that supports TCP sockets. Furthermore, Player makes no assumptions about how you want to structure your robot control programs, and the developer is free to program any way that they like. If you want your client to be a highly concurrent multi-threaded program write it like that, if you like simple read-think-act loop then do that. Client-side libraries are used that hide the internals of the Player message protocol from the developer when programming controllers. The libraries are available in C, C++, Python, Java, LISP, Ruby, and Ada; however C and C++ libraries are the most commonly used.

6.2.2.2 Stage

Stage is a 2D multi-robot simulator from the Player Project. Stage simulates a population of mobile robots, sensors, and objects in a two-dimensional bitmapped environment. Stage is designed to support research into multi-agent autonomous systems, so it provides fairly simple, computationally cheap models of lots of devices, rather than attempting to emulate any device with great fidelity.

There are three ways to implement Stage: (1) a standalone robot simulation program that loads your robot control program from a library that you provide, (2) the Stage plugin for Player provides a population of virtual robots for the Player networked robot interface system, or (3) for writing your own custom simulator inside your own programs. For our use the second implementation, Stage is as a plugin module to Player, is used; this implementation is often called Player/Stage in the robotics community. Users write robot controllers and sensor algorithms as clients to the Player server.

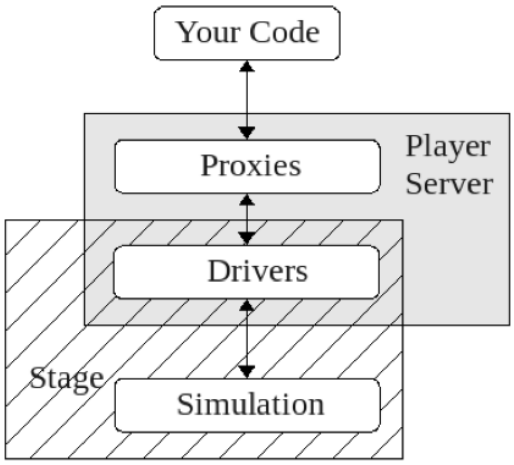


Figure 6.4: Player/Stage [43]

Stage allows for rapid prototyping of controllers designed for real robots. Player clients developed using Stage will work with little or no modification with the real robots, and typically clients cannot tell the difference between the real robot devices

and their simulated equivalents. Since we are working in multi-agent robotic systems, we will only be simulating in Stage and will not work with the 3D simulator Gazebo.

When developing robot controllers in Player/Stage, the methodology is to first start with Stage to simulate how the robot would move in the terrain, and then once the simulation is working as expected, you execute the robot movement with Player. The advantage of using the Player/Stage robot development environment is that as a programmer you can quickly communicate with sensors and actuators without having to worry about how those hardware communications work. Furthermore, the extensive library of methods and classes in C/C++ cover many tasks in robot programming, reducing programming and development time for developers.

6.3 Analysis and Decision: Robot Development Environment

Both Microsoft Robotics Development Studio and Player/Stage were tested with the X80 robot to decide which development environment best suited the research needs. Table 6.1[5] below shows a comparison of the MRDS and Player Project robot frameworks.

Framework	**Microsoft Robotics Developer Studio**	**The Player Project**
Developer	Microsoft Corp.	Robotics Research Lab University of Southern California (USC)
Homepage	www.microsoft.com/robotics	playerstage.sourceforge.net
Platform		
Operating System	Windows	Linux, Mac OSX
General		
Architecture	modular, service-oriented, similar to web services	simple, modular, server/client
Structural Elements	Decentralized System Services (DSS), DSS Nodes	"player servers", "clients", "(virtual) devices"
Runtime Model	event-driven	clents receive data periodically
Development		
Implementation Language	C#	C++
Build Tool	.NET IDE	make
Communication		
Standard Protocol	DSSP, based on SOAP HTTP for web browsers	custom, simple, based on TCP
Data Encoding	XML	XDR
Tools		
Tools Support	Visual Programming Language IDE, Advanced Simulation (3D), Testing	Debugging/Diagnosis, Simulation (2D and 3D)

Table 6.1: Comparison between Microsoft Robotics Developer Studio and Player/Stage

After a thorough analysis it was determined that Player/Stage was a lighter robot development environment, with a significantly reduced development time than the Microsoft alternative. Writing formation controllers in Player/Stage was faster, easier, and simpler than in MRDS. The Player/Stage robot development environment fulfilled all of our research needs, it was an easier platform to work with, and it successfully implemented robot commands from a computer to the X80 robot through a Wi-Fi connection.

Chapter 7

Simulation and Experimental Design

The purpose of the research was to develop formation controllers for a team of mobile robots. Formations add organization and order to the team as they travel through a terrain, and provide a mechanism to avoid collisions. Geometric and self-organizing formation approaches were developed in Chapter 4.

This chapter covers the work done to create simulations and experiments for single and multi-robot systems. Section 7.1 introduces the MATLAB programming language and the steps taken to develop animations of moving robots in a two-dimensional terrain. Section 7.2 explains the Player/Stage relationship, and the steps needed to create animations of multi-robot systems in Stage. Finally, section 7.3 explains the transition from simulations in Player/Stage to robot experiments with the X80.

7.1 MATLAB Simulation Design

MATLAB is a high-level programming language used by scientists and engineers to quickly analyze data, develop algorithms, and create models and applications. MATLAB is an excellent prototyping tool, since the language, toolboxes, and built-in math functions enable users to reach solutions faster than with traditional programming languages such as C/C++ or Java. MATLAB is advantageous for robotic simulations since it provides native support for vector and matrix algebra, enabling fast development and execution of robotic controllers. For our experimental procedure, MATLAB was used as a first step to analyze the feasibility of new navigation algorithms for single and multi-robot cases.

Simulations in MATLAB were created in the MATLAB Editor using the MATLAB programming language. Simulations were executed in the form of animations; animations were used since they could display robot movement and collision avoidance characteristics of the navigation algorithms, and could quickly validate their use. When the MATLAB code was executed, the animation was plotted in the MATLAB Figure window.

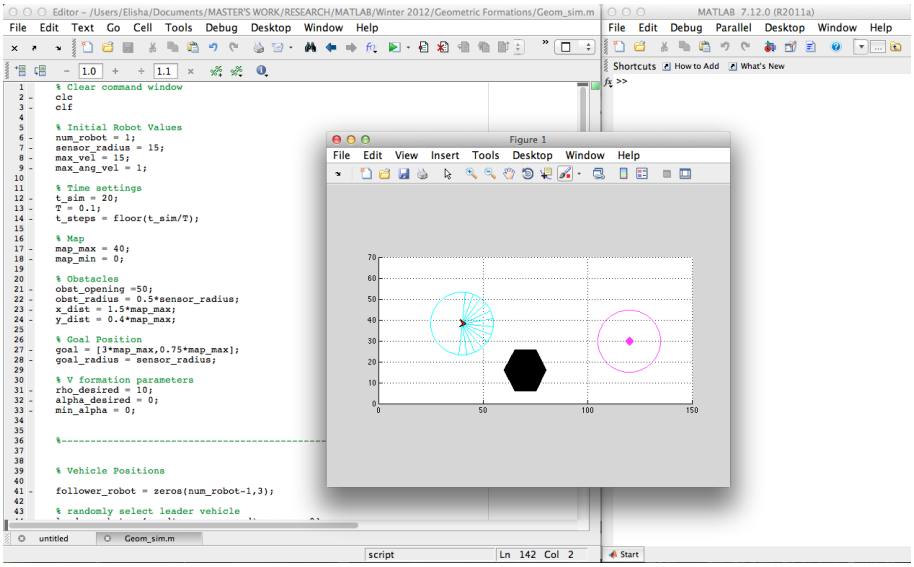


Figure 7.1: Working in the MATLAB Editor to create simulations

7.1.1 MATLAB Simulation Elements

Since simulations in MATLAB were intended to be a first test to see the feasibility of new navigation algorithms, animations were kept as simple as possible. An arrow was used to depict the vehicle's position and orientation in the 2D plane. The robots' x and y position was plotted at each time step in the Cartesian plane, and its yaw angle was depicted by the direction of the arrowhead at each time step. Using the arrow, non-holonomic movement of the mobile robots could easily be visualized.

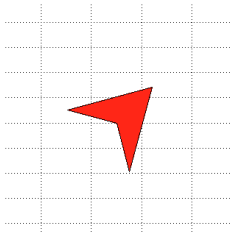
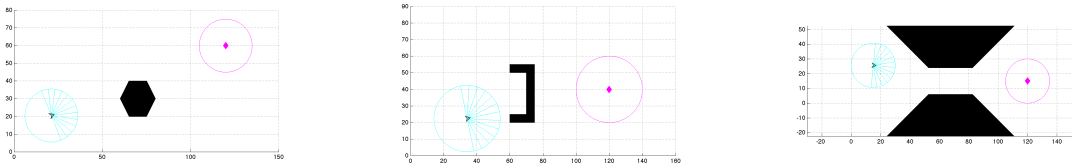


Figure 7.2: Arrow representing the mobile robot's pose in the terrain

Furthermore, obstacles in the terrain were designed with geometric shapes. Since complex shapes were difficult to draw in MATLAB, all obstacles were represented as polygons and were drawn using tools from MATLABs Graphics toolbox. The obstacles were colored black in all of our simulations



(a) (b) (c)

Figure 7.3: (a) Hexagon shaped obstacle, (b) C-shaped obstacle, c) narrow passage obstacle

Mobile robots use their sensors to analyze the terrain and find obstacles. For the simulations, the sensors were plotted as cyan beams extending from the center of the robot. Each beam was separated by an angle PI/12, and all beams had a

constant beam radius that was defined by the user. A circle joining the outer ends of the beams defined the viewing area of the mobile robot, the area where the robot could see and interact obstacles in the terrain.

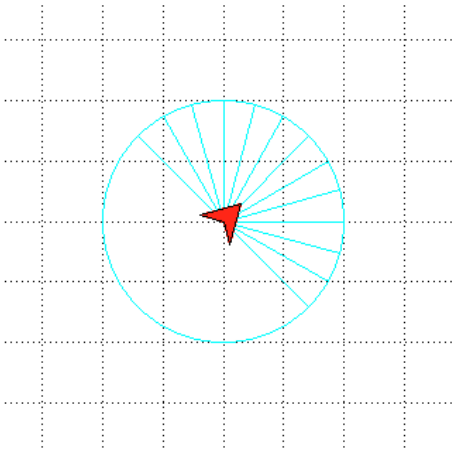


Figure 7.4: Sensor beams around the robot

When an obstacle entered the viewing area of the mobile robot, the intersection of the beam and the obstacle polygon was plotted using commands from MATLABs Mapping Toolbox. In the simulations, the intersection point was plotted with a small red circle, and the beam with the closest intersection point was implemented in the collision avoidance algorithm.

In the terrain, a goal position was plotted using magenta diamond. The simulations were designed to work for a single robot case, and also could easily expand to a multi-robot case. The multi-robot case was designed to incorporate any number of robots set by the user. At the beginning of all simulations, the robots were randomly positioned in the terrain, at random x, y, and theta values. Randomly generated initial positions were important to better test the validity navigation algorithms with difficult robot orientations.

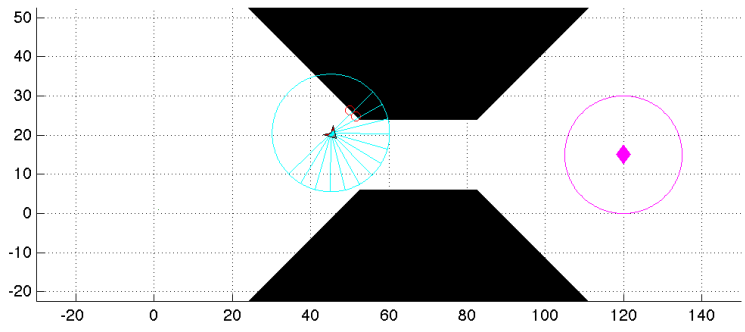


Figure 7.5: Beam intersection with an obstacle

Finally, the user had control over several constant constraints in the simulation. The user could modify the size of the terrain, simulation time and time steps, number of robots, sensor beam radius, the maximum velocity and angular velocity of the robots, the position and shape of the obstacles in the terrain, and the goal position in the terrain Simulations are executed by pressing the 'Run' command in the Editor window.

7.2 Stage Simulation Design

The purpose of the MATLAB simulations at the beginning of the research was to test the validity of our navigation and formation algorithms. Once we were satisfied with the MATLAB simulation results for the single and multi-robot cases, we moved forward to create simulations in Player/Stage. Stage simulations take into account the exact geometry of the X80 mobile robot, along with sensor locations, sensor limits, and actuator variables. Stage simulations allowed for more interesting and

complex terrains. This was the case since drawings made in a paint program could be added to the world file, making far more realistic terrains for the simulated robots to move in. Stage simulations were an intermediary step between MATLAB simulations and the X80 experiments, to see how the robots would behave with the navigation and formation control algorithm. Simulations in Stage gave a more accurate picture of the robot movement with its actual hardware constraints, and showed how the robots interacted with each other in complex terrains with many obstacles.

7.2.1 Player/Stage Implementation

Player and Stage work together to operate mobile robots from a Unix based operating system such as Linux or Mac. Player is a hardware abstraction layer that provides access to the robots sensors and actuators for real robot testing. Stage is a 2-dimensional simulation software. Within stage you set up the geometry of the robot, what sensors are onboard, where the sensors are located on the robot, and any obstacles situated in its environment.

The largest benefit of using the Player and Stage software is the quick and easy transition from simulation to testing. Control of the robot movement is achieved with a C++ file that access values from sensor data and apply actuator control. The controller runs in parallel to Player, and is implemented in the same manner in either simulation or testing mode.

In Player/Stage there are three main types of files that you work with. Firstly, the World file describes the simulated world that the robot moves in, and any items within the world like obstacles. Moreover, the Configuration file gives player information about the robot, and tells Player what drivers it needs to access in order to interact with the sensors and actuators on the robot. If you are creating a simulation, the configuration file points to the Stage driver. Finally, in order to move the

robot and apply any type of intelligence, a controller file is programmed in either C or C++. The controller file includes the Player C/C++ library, it uses player commands to access input sensor data, and the file can also output commands to actuators. The C/C++ code is compiled and built, and the executable file works in conjunction with the configuration file to move the robot.

7.2.2 The World File

The first step in developing a Stage simulation is to define the world that the robot lives in. Firstly, the world file described the details about the robot geometry, its exact size, and parameters. Table 7.1 describes the center of gravity, mass, and volume parameters of the X80 robot. Furthermore, table 7.2 details how the robots were drawn in the world file. The X80 was split up into various sections: the back, the middle, the front, and the left and right wheels. Each point on on the geometry was plotted to create a polygon, and the interior of the polygon was colored. Finally, figure 7.6 shows the final drawing of the X80 robot in the world based on the geometric parameters in table 7.2.

General Robot Parameters	**Values**
Center of Gravity [x, y, z] in meters	[0.0, 0.025, 0.0]
Mass in kg	3.5
Volume [x, y, z] in meters	[0.33, 0.36, 0.26]

Table 7.1: Robot parameters in the World

Robot Section	**Number of Points**	**Coordinates in the x-y Plane in meters** $[x, y]$	**Coordinates in the z-plane** in meters $[z_{min}, z_{max}]$	**Color**
Red back	6	point[0] [0.0, 0.21] point[1] [0.0, 0.12] point[2] [0.08, 0.05] point[3] [0.25, 0.05] point[4] [0.33, 0.12] point[5] [0.33, 0.21]	[0.17, 0.26]	Red
Middle gray rectangle	4	point[0] [0.06, 0.21] point[1] [0.27, 0.21] point[2] [0.27, 0.26] point[3] [0.06, 0.26]	[0.085, 0.165]	Gray
Front gray arc	11	point[0] [0.0, 0.29] point[1] [0.0, 0.26] point[2] [0.33, 0.26] point[3] [0.33, 0.29] point[4] [0.28, 0.33] point[5] [0.25, 0.34] point[6] [0.22, 0.35] point[7] [0.165, 0.36] point[8] [0.11, 0.35] point[9] [0.08, 0.34] point[10] [0.05, 0.33]	[0.085, 0.165]	Gray
Left wheel	4	point[0] [0.0, 0.09] point[0] [0.03, 0.09] point[2] [0.03, 0.26] point[3] [0.0, 0.26]	[0, 0.17]	Black
Right wheel	4	point[0] [0.3, 0.09] point[1] [0.33, 0.09] point[2] [0.33, 0.26] point[3] [0.3, 0.26]	[0, 0.17]	Black

Table 7.2: Drawing an exact copy of the X80 robot geometry in Stage

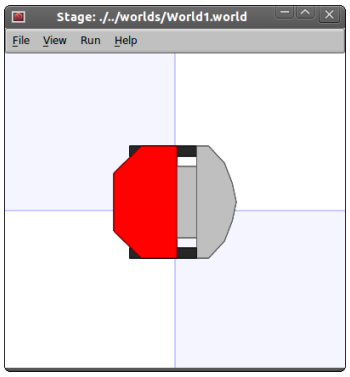


Figure 7.6: X80 simulated robot in Stage

The next step in the design of the world file is to tell Stage where the sensors are located on the robot unit. Tables 7.3, 7.4, 7.5, and 7.6, show the properties and positions of the sonar and infrared range sensors on the X80 robot. Sensor positions were included in the world file, and figure 7.7 shows a final picture of the X80 with its range sensors in the world file.

Sonar Sensor Properties	**Values**
Number of sonar sensors	6
Field of view of each transducer [range min, range max, view angle] in meters and degrees	$[0.08, 2.55, 10°]$
Size of each transducer $[x_{size}, y_{size}]$ in meters	$[0.05, 0.02]$

Table 7.3: Sonar parameters for the X80 robot

Sonar Position	**Coordinates [x, y, yaw angle] in meters and degrees**
Front left	$[0.14, 0.14, 45°]$
Front center	$[0.19, 0.0, 0°]$
Front right	$[0.14, -0.14, -45°]$
Back right	$[-0.12, -0.125, -135°]$
Back center	$[-0.18, 0.0, 180°]$
Back left	$[-0.12, 0.125, 135°]$

Table 7.4: IR position coordinates on the X80

IR Sensor Properties	Values
Number of infrared sensors	7
Field of view of each transducer [range min, range max, view angle] in meters and degrees	$[0.1, 0.8, 5°]$
Size of each transducer $[x_{size}, y_{size}]$ in meters	$[0.01, 0.03]$

Table 7.5: Infrared parameters for the X80 robot

IR Position	Coordinates [x, y, yaw angle] in meters and degrees
Front left	$[0.16, 0.10, 40°]$
Front left center	$[0.17, 0.04, 10°]$
Front right center	$[0.17, -0.04, -10°]$
Front right	$[0.16, -0.10, -40°]$
Right	$[0.0, -0.12, -90°]$
Rear	$[-0.18, 0.0, 180°]$
Left	$[0.0, 0.12, 90°]$

Table 7.6: Infrared position coordinates on the X80

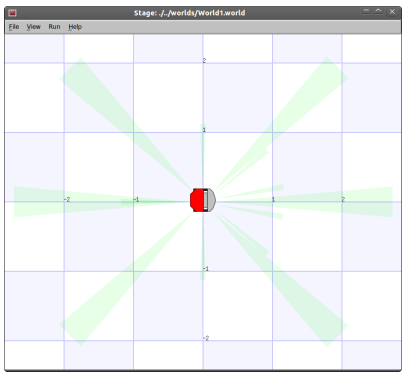


Figure 7.7: X80 with sonar and IR beams simulated robot in Stage

Finally the last step in creating the world file is to provide information about obstacles in the environment. Terrains could be generated in two ways: (1) simple geometric shapes, like rectangles and polygons could be drawn using drawing commands in the world file, or (2) terrains could be uploaded from a drawing file with a Portable Network Graphic (.png) or a Bitmap Image File (.bmp). For our simulations, image files of the terrain were saved in the .png file format. Figure 7.8 shows a complex environment that can be used to test the X80 controllers.

Figure 7.8: A possible simulation environment in Stage

7.2.3 The Configuration File

Once the robot parameters and the terrain environment were defined in the world file, a configuration file was created to begin the simulation. The configuration file lists the simulation plugin "stageplugin", the world file associated with the simulation, and the port (6665) for the position and sensor proxies. The code below is used to create a very simple simulation for a single robot. See the appendix section on developing a simple simulation in Stage for further details.

```
# STAGE DRIVER ------

driver
(
        name "stage"
        provides ["simulation:0"]
        plugin "stageplugin"
        worldfile "AutoWorldFiles/robot1.world"
)

# X80 DRIVERS ------

driver
(
        name "stage"
        provides ["6665:position2d:0" "6665:sonar:0" "6665:sonar:1"]
        model "robot1"
)
```

7.2.4 Developing Robot Controllers in C++

The configuration and world files in Player/Stage define the static parameters of the robot: its geometry, sensor parameters, and initial position in the world. When the configuration file is executed, the robot is displayed at its initial position, but it cannot move. In order to move the robot around its terrain to view its navigation and collision avoidance algorithms, a controller file must first be made to tell the robot how to move. Controllers in Player/Stage are written in C/C++. The first step in the process was to find an Integrated Development Environment (IDE) to program the C++ code in the Ubuntu Operating System. Two popular open source IDEs were investigated: Eclipse and Code::Blocks. After testing both IDEs with the PlayerC++ tools, it was found that Eclipse integrated best with the Player/Stage libraries, and was the easiest IDE to work with for our experiments.

The next step in the process was to take the MATLAB navigation and formation controllers and to write them in C++ in the Eclipse IDE. The C++ code was linked to the PlayerC++ library in order to utilize the Player commands. The following section outlines the methods and attributes of the Player classes used in our simulations.

The PlayerC++ library is a massive library of popular robot commands used by in academia. The library consists of classes that command manipulator and mobile robot actuators and sensors. In this section, only the PlayerC++ proxies that were used in the experiments with the X80 mobile robot will be described.

The PlayerClient is used for communicating with the Player server. Each robot is assigned its own PlayerClient at the beginning of the program to connect to the Player server, and each PlayerClient has a port number associated with it. In Player/Stage the port numbers start at port 6665 for the first robot, then increments by one for each additional robot (for example, the second robot will occupy port 6666, the third robot will occupy port 6667, etc.) Contained within the PlayerClient

object are methods for changing the connection parameters and obtaining access to devices.

The Position2dProxy class is used to control mobile robots in a 2D plane. Using the methods within this class you can get the current pose of the robot, and set the speed of the vehicle. In our experiments, velocity and angular velocity values were calculated from our algorithms. These velocity commands were executed in our Player/Stage simulation using the SetSpeed() method in the Position2dProxy class.

The SonarProxy is used to control sonar devices. In our experiments, both the sonar and infrared sensors were accessed using SonarProxy commands. The sonar interface provides access to a collection of fixed range sensors, and outputs the values as an array. Range measurements of the sensors allow the mobile robots to see and understand its environment in order to avoid collisions.

All of the functions associated with the PlayerClient, Position2dProxy, and SonarProxy are listed in Appendix C

7.2.5 Running a Stage Simulation

To run a simulation in Player/Stage you use the terminal window in Ubuntu. If the configuration file for the experiment was saved as sim.cfg, find the location of the file in the terminal window and once the folder is accessed use the command

player sim.cfg

The Player command will open a viewer with the terrain and robot described from the sim.cfg configuration and world files. Next, open a second terminal window and search for the C/C++ controller file that was built in Eclipse. If the C/C++ file was save as controller.cpp, Eclipse will debug and build the application. To run the controller, find the location of the executable file, and write the following command in the terminal window.

./controller

After the controller is executed, the robot will move in the environment until you tell it to stop. For our experiments we included the csignal library; methods from this library allow for keyboard inputs to stop the robot prematurely, if necessary, from the terminal window.

7.3 Transitioning from Simulations to Robot Implementation with Player/Stage

To run the navigation and formation control algorithm on the X80 in Player/Stage, it is a simple matter of changing the configuration file. Instead of calling the simulation proxy, player references a plugin specifically designed for the X80 robot called libx80. This plugin is a binary file created through the source code x80.cc that can be found in appendix F. In the configuration file you choose a connection type, in our case a UDP network, and write the IP address of the robot in use: 192.168.0.205. The port and initial pose of the robot are then listed, and sonar and IR filter parameters can be applied. The hardware configuration file for a single robot is shown below.

```
# X80 DRIVER ------------------------------------------------------

driver
(
        name "x80"
        plugin "../player_drivers/X80PlayerPlugin/lib/libx80"
        provides ["6665:position2d:0" "6665:sonar:0" "6665:sonar:1"]
        connection      "udp"
        address "192.168.0.205"
        port            10001                   # for udp only
        pose [0 0 0 0]

        sonar_filter    1.0     # low pass filter smoothing factor, <=
            1.0
        ir_filter       1.0     # low pass filter smoothing factor, <=
            1.0
)
```

Executing the X80 with the control algorithm follows the same procedure as executing the simulated robot. If the configuration file for the experiment was saved as hardware.cfg, find the location of the file in the terminal window and once the folder is accessed use the command

player hardware.cfg

To run the controller, find the location of the executable file, and write the following command in the terminal window.

./controller

After the controller is executed, the X80 robot will move in the room until you manually tell it to stop, or the robot reaches its destination.

Chapter 8

Results for Single Robot Experiments

The purpose of the research was to develop formation controllers for a team of autonomous mobile robots. Collision avoidance was a main requirement in our research objectives; the team was expected to travel together through a terrain, while avoiding collisions with obstacles in the environment. The first stage of our research was to develop a navigation strategy for the robots to avoid static obstacles in the terrain. The following chapter describes the results for navigation with a single robot using the velocity potential method detailed in Chapter 4.

Section 8.1 gives a quick overview of the flowchart symbols used to describe the methodology for our programming code. Sections 8.2 and 8.3 describe the simulation algorithm and simulation results for the single robot experiments. Finally, sections 8.4 and 8.5 describe the experimental algorithm and results for our work with the X80 mobile robot.

8.1 Flowchart Symbols

In our work, high-level controllers for collision avoidance and formation control were programmed in the MATLAB and C++ programming languages. Due to the length of the code for each experiment, it was not practical to insert the programming work in the results sections for chapters 8, 9, and 10. Instead, flowcharts are used to visually display the methodology, and the programming work for each simulation and experiment is listed in the Appendix. Figure 8.1 displays the flowchart symbols that were used in describing our simulation and experimental work.

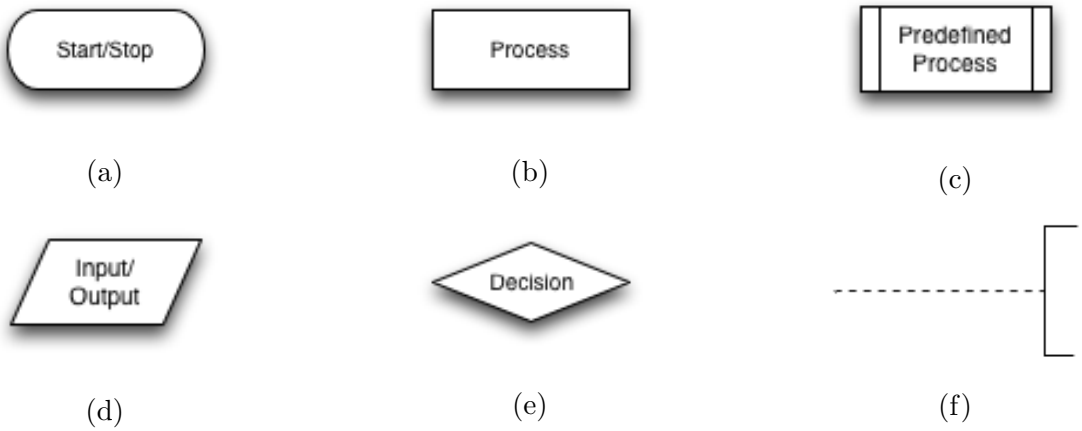


Figure 8.1: Flowchart symbols

Firstly, the Start/Stop terminal symbol 8.1a represents the beginning or end of a procedure or program. Terminals beginning a procedure will contain the name of the procedure, and terminals ending a procedure contain the word EXIT. Secondly, the Process symbol 8.1b represents a processing task within the program. Any data manipulation, computation, or movement of data within the program is illustrated using a process symbol. Thirdly, a Predefined Process 8.1c indicates that the instructions to be executed are located elsewhere in the program. In our case, we used the predefined process symbol to keep the code simple, by incorporating functions that separate portions of the code in a separate flow charts. Forthly, the Input/Output process 8.1d is used to indicate that some input or output operation

is being performed. Input operations include reading files and interacting with the user. Output operations include writing files and displaying information. Fifthly, the Decision symbol 8.1e represents a point in the program where the logic flow will follow one of the two paths depending on a given situation. Finally, the Annotation symbol 8.1f is used to comment on a portion of the program in an effort to explain why the program is performing some task. In our work, the annotation symbol is used as a reminder of actions we need to take in initializing our program for execution.

8.2 Simulation Algorithm

Simulations for navigation using the velocity potential method were first carried out in MATLAB using the methodology described in Chapter 7. In this simulation, a single robot navigates through the terrain towards the goal, following an attractive flow field. If the robot senses an obstacle, a spiral vortex is created in the obstacle position, and the flow pushes the robot away from the obstacle. Figure 8.2a shows a flowchart of the main function for the simulation. The main function initializes parameters such: as maximum velocity and angular velocity, sensor radius and goal radius, constants, and map dimensions. Furthermore, it gathers inputs from the user such as the initial robot pose and goal position in the map. The majority of the calculations occur in a loop, figure 8.2b, where robot checks for nearby obstacle, calculates its new pose, and plots the goal, obstacles, and robot pose in the MATLAB figure window.

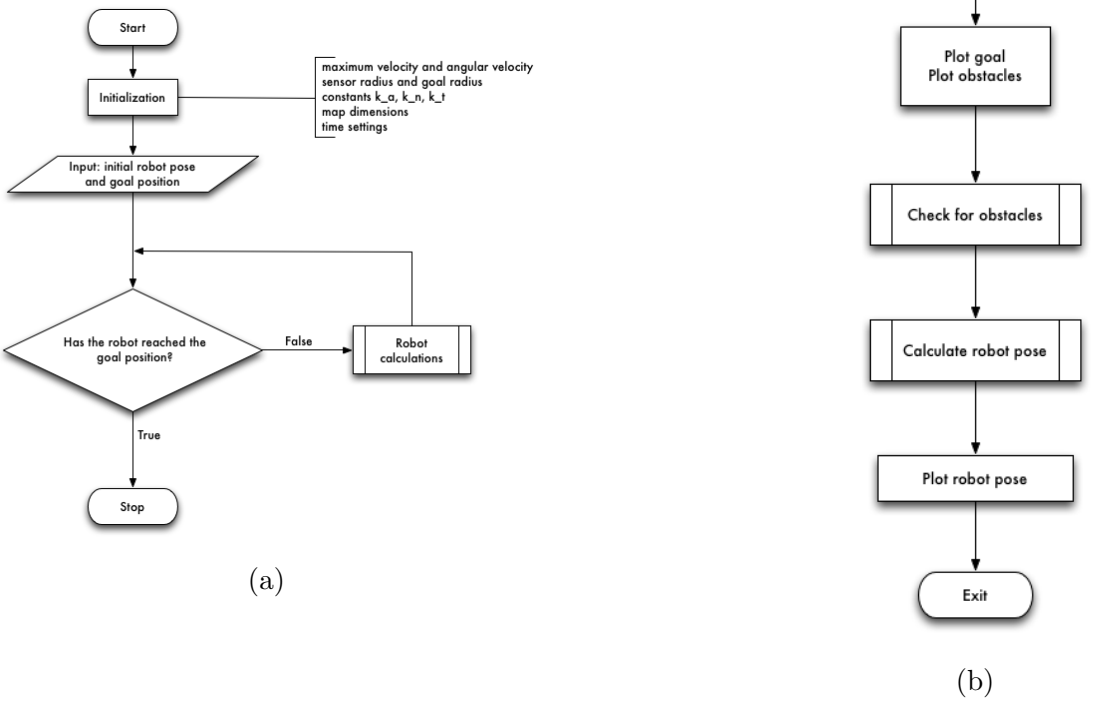


Figure 8.2: (left) main function, (right) robot calculations

The details of the *Check for obstacles* function and *Calculate robot pose* function are explained in figures 8.3a and 8.3b. The *Check for obstacles* function scans the sonar and infrared sensors to see if an obstacle is in range. If an obstacle is in view, the function stores the range and angle measurement for the sensor with the shortest distance to the obstacle. The *Calculate robot pose* function implements the velocity potential method calculations from Chapter 4, and calculates the attractive, repulsive, and resultant velocity and angular velocity of the robot. The full MATLAB program for the single robot simulation can be found in Appendix G.

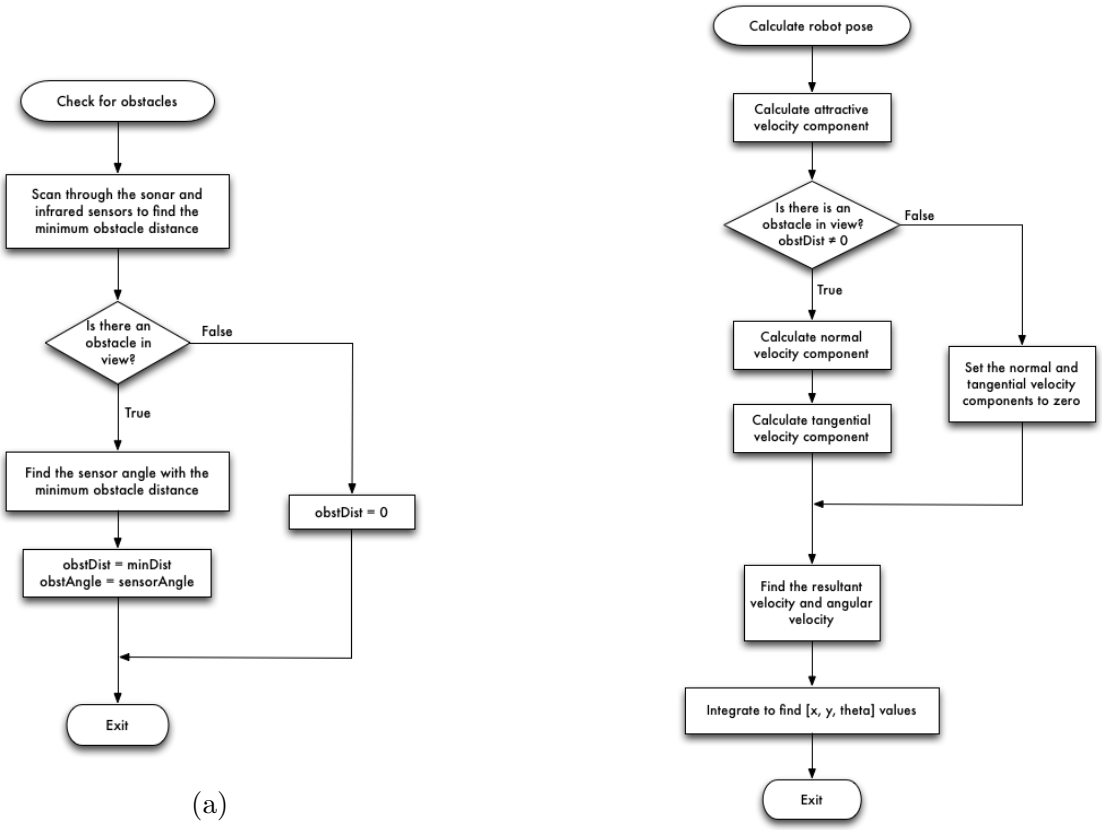


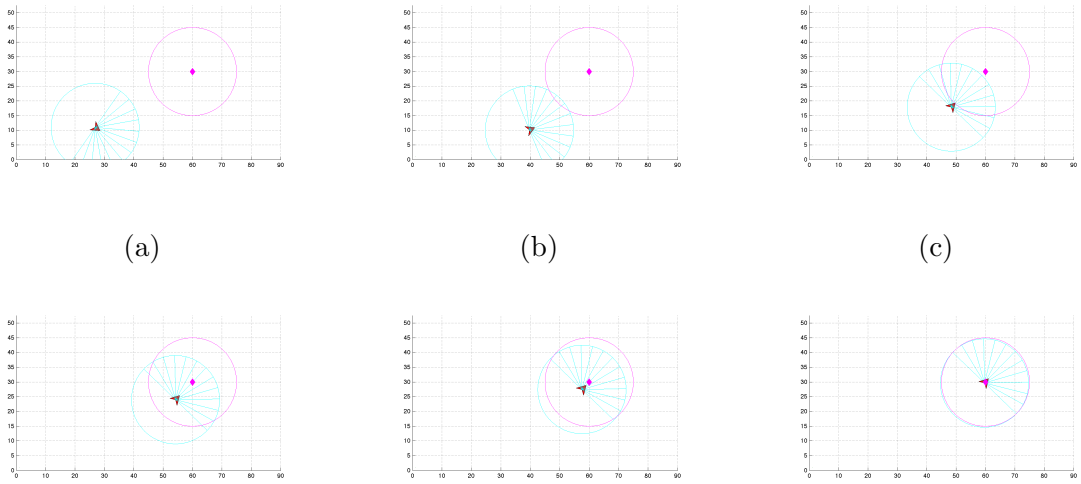
Figure 8.3: (left) check for obstacles, (right) calculate robot pose

8.3 Simulation Results

8.3.1 Single Robot - No Obstacles

For the single robot simulations, we first wanted to test how the robot was able to navigated through a terrain with no obstacles. Figure 8.4 shows snapshots of the robot as it travels to the goal. The robot was initially placed at an angle of -45 degrees, and using the attractive velocity potential only, it took the shortest and most direct route towards the goal position. As the robot approached the goal and entered the region of interest around the goal (the magenta circle), the robot slowed down until it came to a halt at the goal position.

Figure 8.4 shows snapshots of the simulation animation, and the movement of the robot between the initial and goal position. The simulation successfully proved that the velocity potential approach could be used to navigate a robot to a goal position with no obstacles in the terrain.



(a) (b) (c)

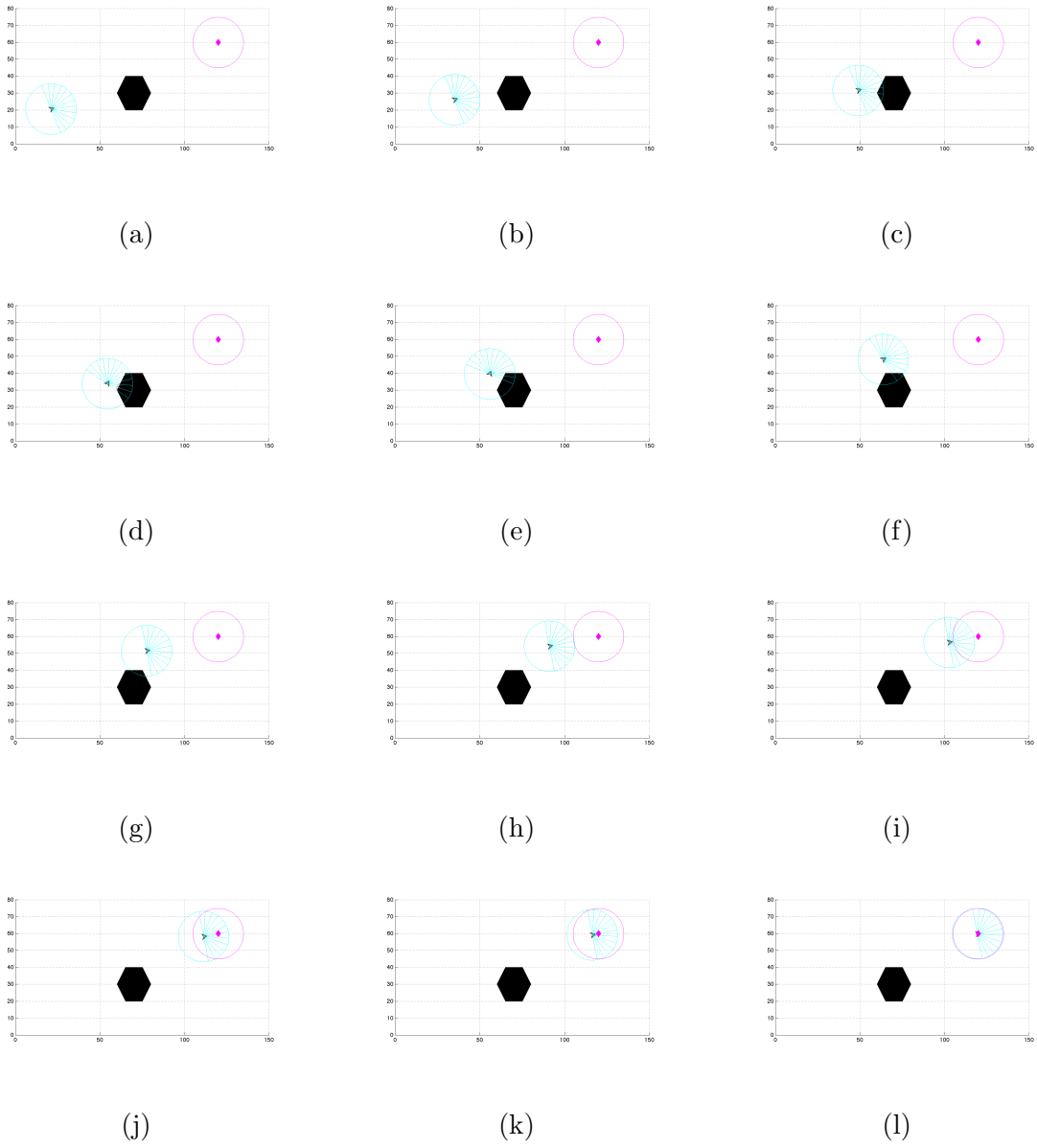
(d) (e) (f)

Figure 8.4: Simulation: no obstacles

8.3.2 Single Robot - With Obstacles

The next simulation placed the robot in a terrain with a simple obstacle. The obstacle was plotted with a hexagon shape, and was placed in the direct path between the robot's initial pose and the goal position. When the robot's sensors sensed the obstacle in the terrain, the shortest sensor range was found and a red circle was plotted at the interface between the shortest sensor beam and the obstacle. The robot then applied the range and angle measurement into its velocity calculation function to navigate the robot around the obstacle and back into the path towards the goal position. Finally, as the robot entered the region of interest around the goal, the robot slowed down and stopped at the goal position. Figure 8.5 displays

snapshots of the simulation for a single robot navigating around an obstacle.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

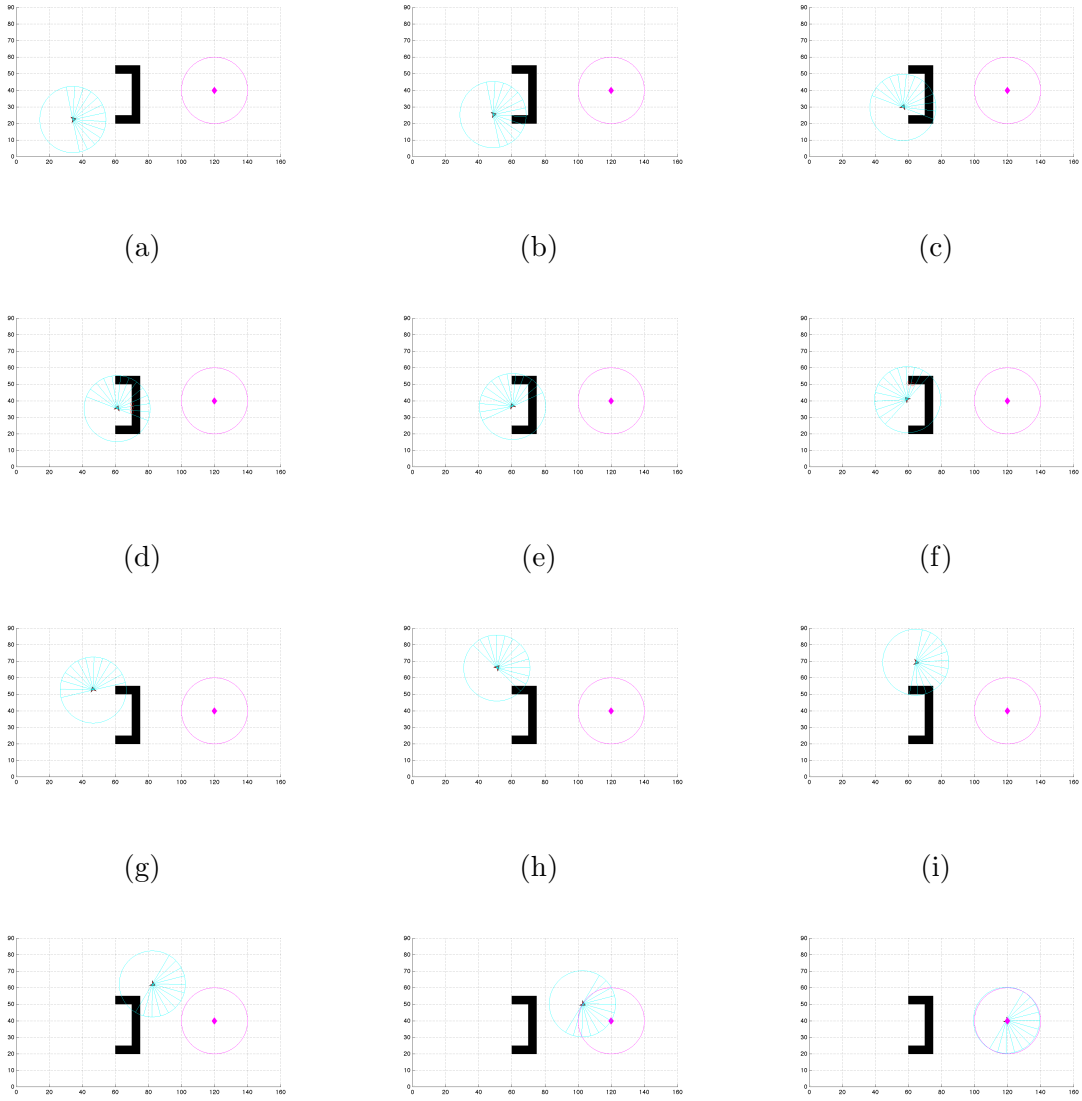
Figure 8.5: Robot travelling around an obstacle

8.3.3 Single Robot Travelling Around a Concave Obstacle

In Chapter 3 and Chapter 4, the Artificial Potential Field (APF) method was discussed. The APF method is a popular reactive navigation controller in mobile robotics, however the algorithm suffers from a local minima problem, which often limits the robot's mobility. The local minima problem causes the robot gets trapped around certain obstacles, and the robot cannot reach its final goal position. Many additions the the APF approach were introduced to address the local minima problem including the addition of Harmonic Potential Fields[20]. The next set of simulations looked to see if our Velocity Potential (VP) method could tackle obstacle shapes that are difficult for the APF method. Two obstacle geometries were considered: (1) the concave obstacle and (2) the narrow passage.

Firstly, the concave obstacle is a very difficult obstacle for the APF method since, when the robot enters the concave obstacle, the robot's attractive and repulsive vectors have equal and opposite magnitudes. This yields a net zero force and stops the vehicle inside the concave obstacle. In order to tackle the concave obstacle with the APF method, additional controllers are required to push the robot away from the concave obstacle.

In our simulations, we wanted to test how the robot behaved when approaching a concave obstacle using the Velocity Potential fluid flow method. The concave obstacle was plotted as a C-shaped polygon in MATLAB, and was placed in the direct path between the robot's initial pose and the goal position. In the simulation, the robot approached the concave obstacle, and then using its normal and tangential velocities, it was able to go around the obstacle until it had a free path towards the goal position. The robot then followed a uniform attractive flow field towards the goal and stopped at the goal position. Figure 8.6 displays the movement of the robot around a concave obstacle, and proves the versatility of the Velocity Potential method when confronting difficult obstacle geometries.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

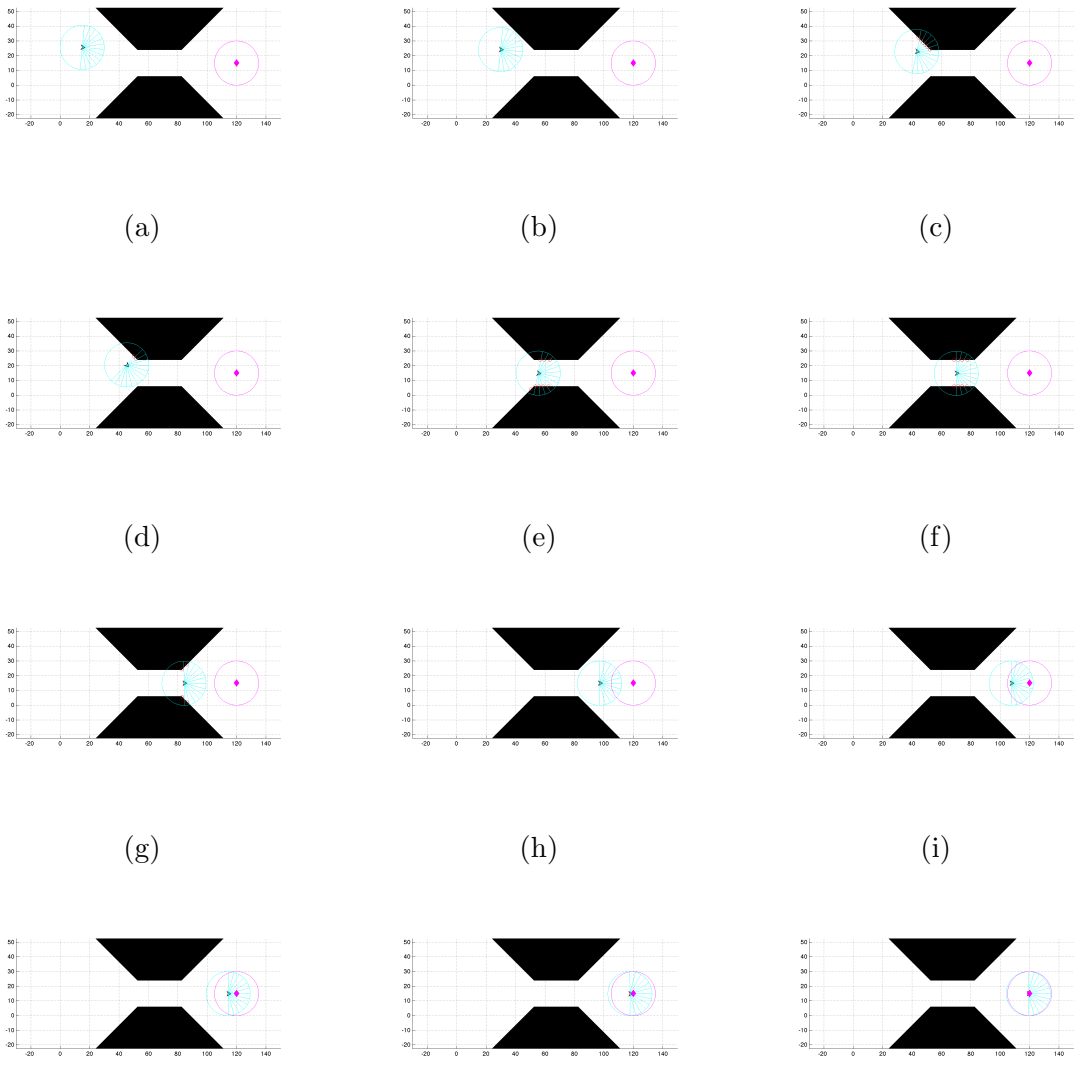
(j) (k) (l)

Figure 8.6: Robot travelling around a concave obstacle

8.3.4 Single Robot Travelling Through a Narrow Passage

Finally, the Velocity Potential method was tested in a narrow passage obstacle. The narrow passage is a difficult obstacle for the APF navigation controller, since it generates a local minimum. For our simulations, the narrow passage was plotted in MATLAB with two trapezoid shapes. Trapezoid shapes were used instead of

rectangle shapes to guide the robots into the narrow opening. In our simulations, the robot was able to successfully navigate through the narrow passage to reach the goal position.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 8.7: Robot travelling through a narrow passage

8.4 Experimental Algorithm

Single robot experiments using the Velocity Potential navigation controller were implemented on the X80 mobile robot. The controller was programmed in C++ using the methods described in Chapter 7. Figure 8.8 shows the main function methodology in the C++ code. The main function was responsible for initializing parameters such as the maximum velocity and angular velocity, the region of interest around the obstacle and goal radius, and constants for control. Furthermore Player Proxies were set in the main function that connected the Personal Computer (PC) to the robot through the wireless network. Finally, the majority of the calculations for robot navigation were carried out in a loop that calculated the velocity based on the Velocity Potential calculations from Chapter 4. The loop continued until the robot reached the goal position, then all robot velocity commands were set to zero, and the experiment was closed.

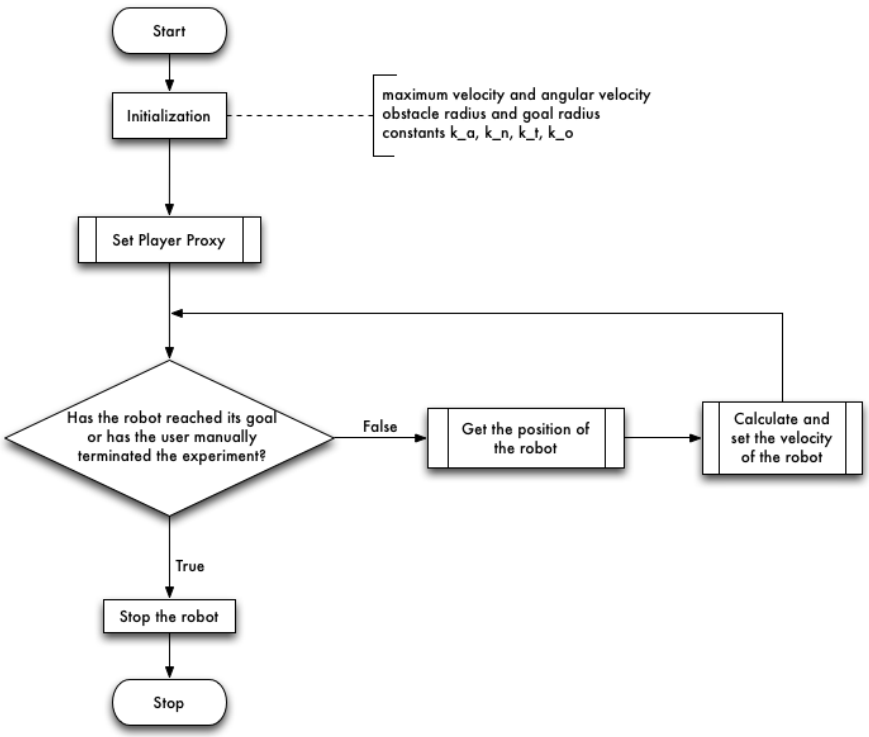


Figure 8.8: main function

The inner workings of the main function are displayed in figures 8.9a, 8.9b, and 8.9c. Firstly, figure 8.9a displays the steps to set the Player Proxy for the client robots. Furthermore, figure 8.9b shows how the controller can access the (x, y, θ) position variables using the GetXPos(), GetYPos, and GetYaw() methods from the Position2dProxy.

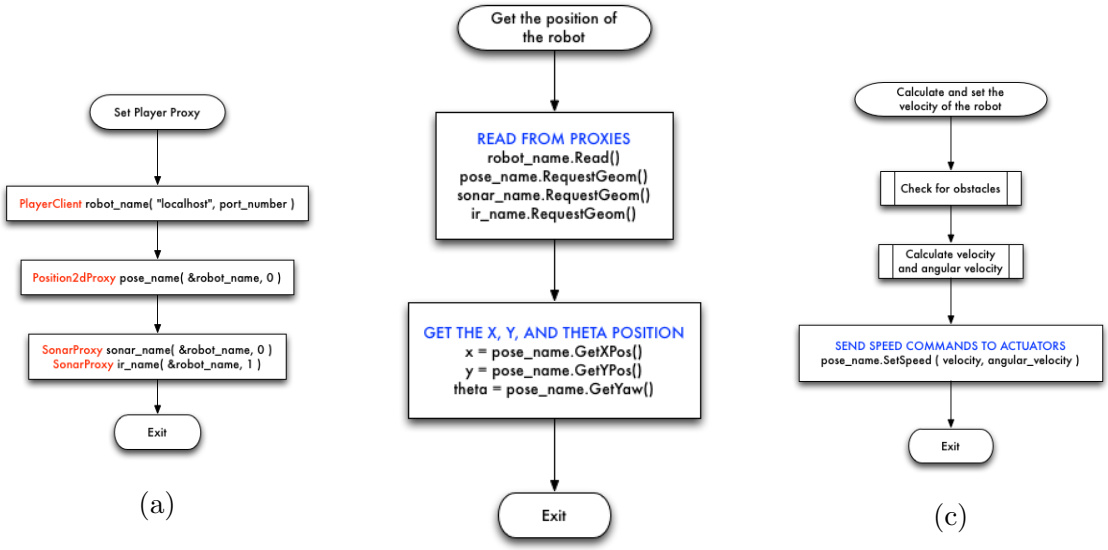


Figure 8.9: (left) set player proxy, (middle) get the robot pose, (right) calculate teh robot velocity

Finally, figure 8.9c displays the steps needed to calculate the velocity of the robot. The robot first checks for obstacles within its viewing range. Figure 8.10a details the steps in the *Check for obstacles* predefined process. After checking for obstacles, the robot calculates its velocity based on the Velocity Potential method from Chapter 4. The details of the implementation of this method in C++ are shown in figure 8.10b. Finally, the robots uses its SetSpeed() method to set the new velocity and angular velocity values onto the X80 robot.

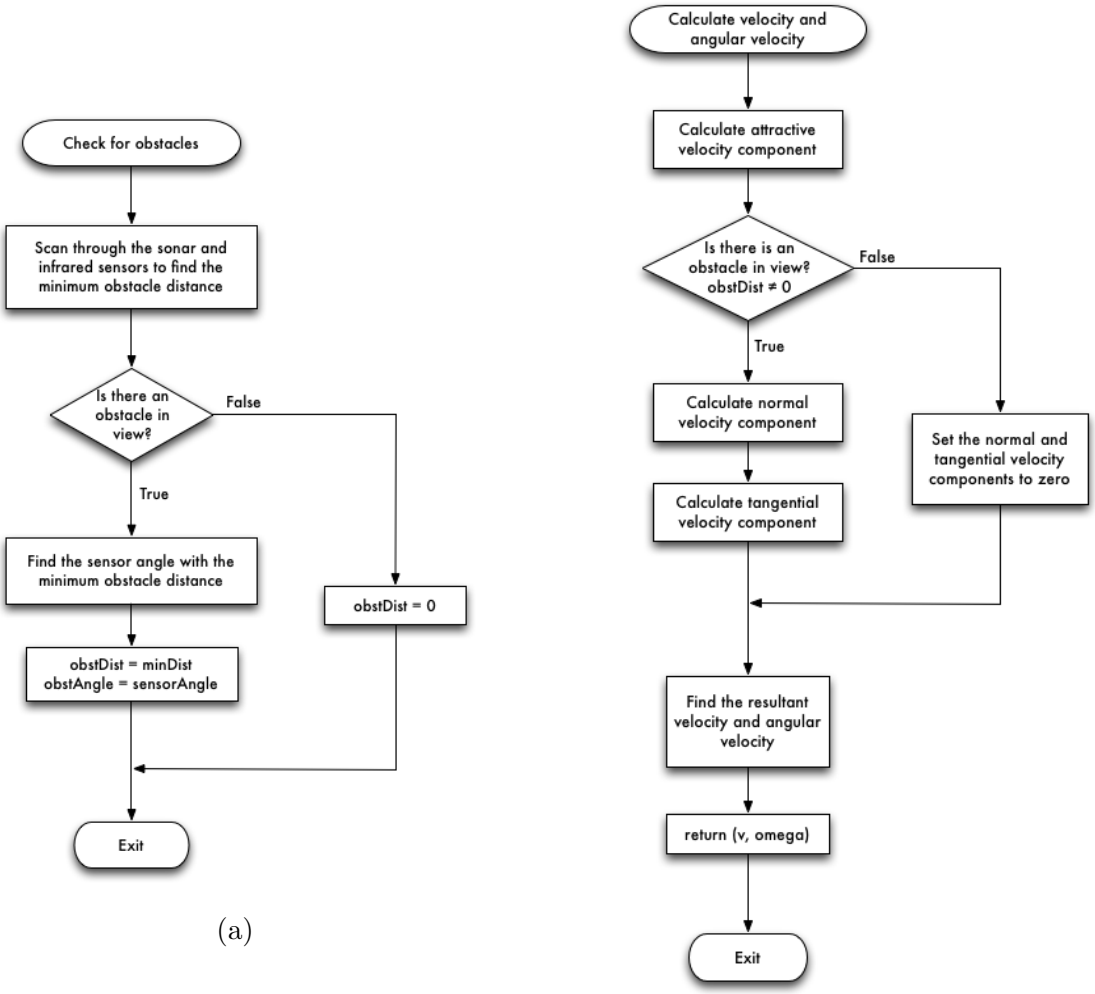


Figure 8.10: (left) check for obstacles, (right) calculate velocity and angular velocity

8.5 Experimental Results

8.5.1 Single Robot Travelling Through a Terrain with Obstacles

Three navigation experiments were tested on the X80 robot using the Velocity Potential navigation controller. The first experiments examined how the X80 mobile robot was able to navigate around simple obstacles in the terrain. In this experiment, potted plants were used as obstacles, and the plants were placed in the direct path between the robot's initial position and the goal position.

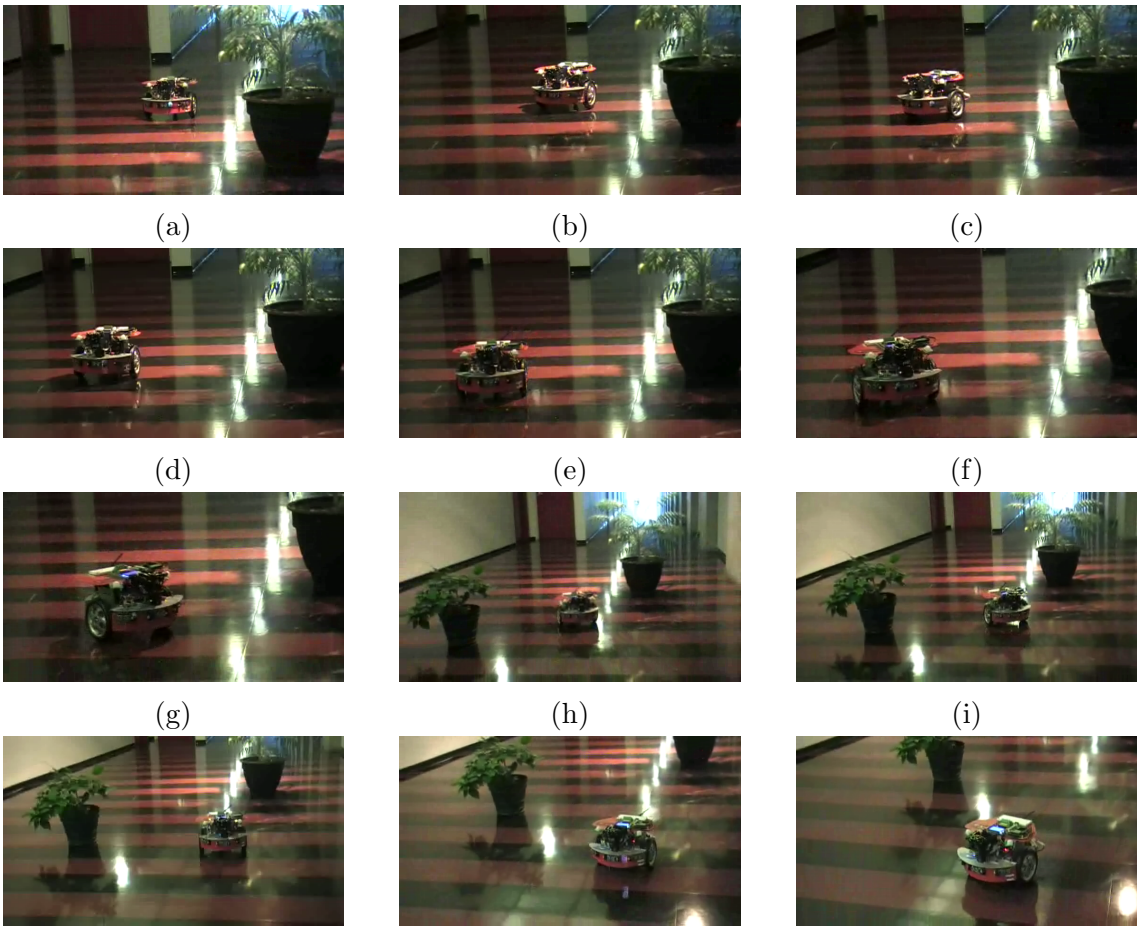


Figure 8.11: X80 navigating aroud an obstacle

Figure 8.11 shows snapshots, at different time intervals, of the robot navigating

around the plants. The robot was able to successfully navigate around the two obstacles and reach the goal position in front of the camera.

8.5.2 Single Robot Travelling Around a Concave Obstacle

In the second experiment, the X80 robot faced the concave obstacle that was built using three boxes. The boxes were paced in a C-shape, and various configurations on the shape of the concave were tested with the X80 robot.

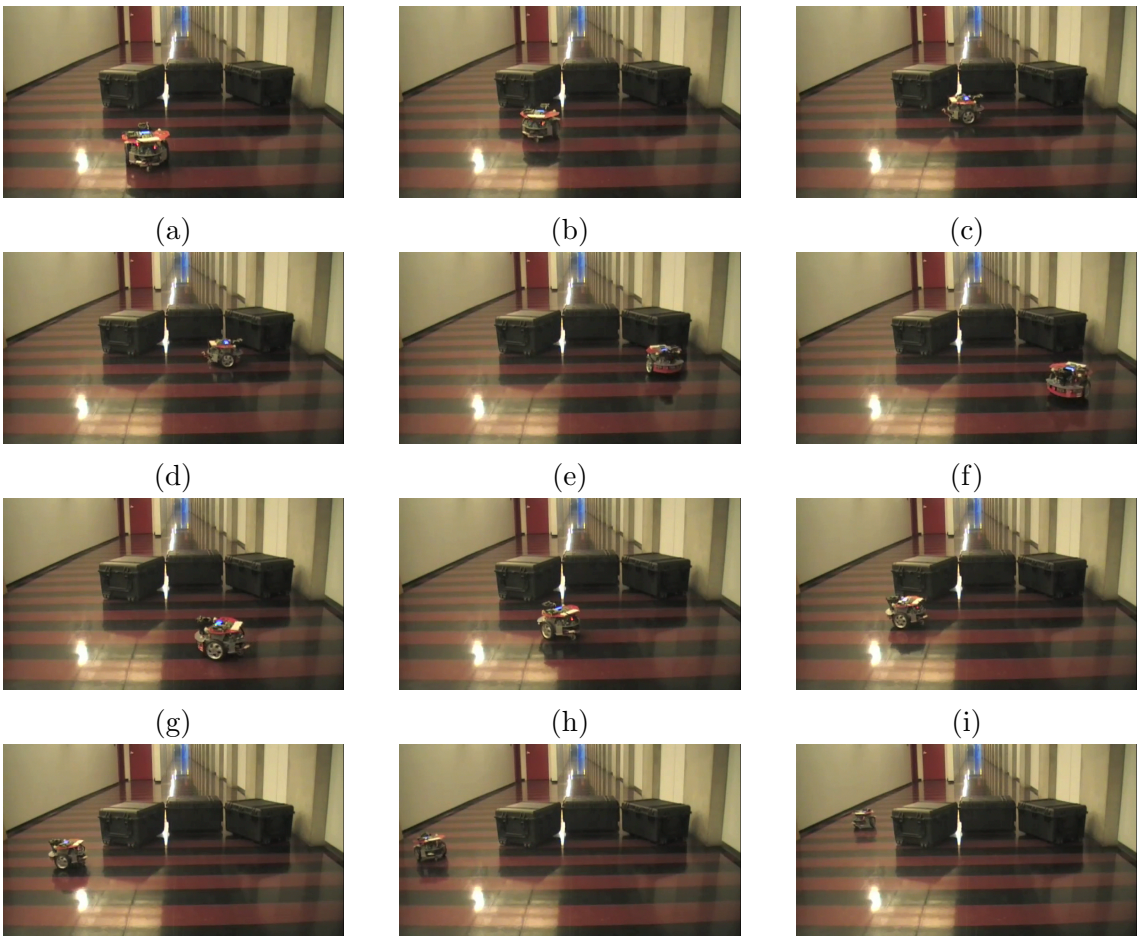
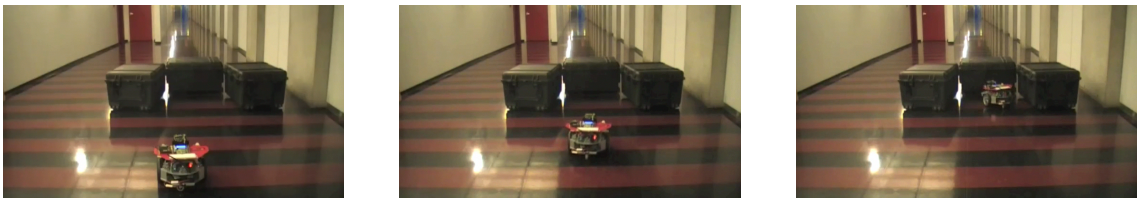


Figure 8.12: X80 travelling around a concave obstacle

It was found that for most shapes, the robot was able to successfully navigate around the concave obstacle to reach the goal position behind the obstacle. The only case when the robot became stuck was when the C-shape was too narrow, figure 8.13

where the robot had no space to turn around. This was an extreme case for the concave obstacle. All-in-all, the experiments proved that the X80 could successfully navigate around a concave obstacle with the Velocity Potential navigation controller.

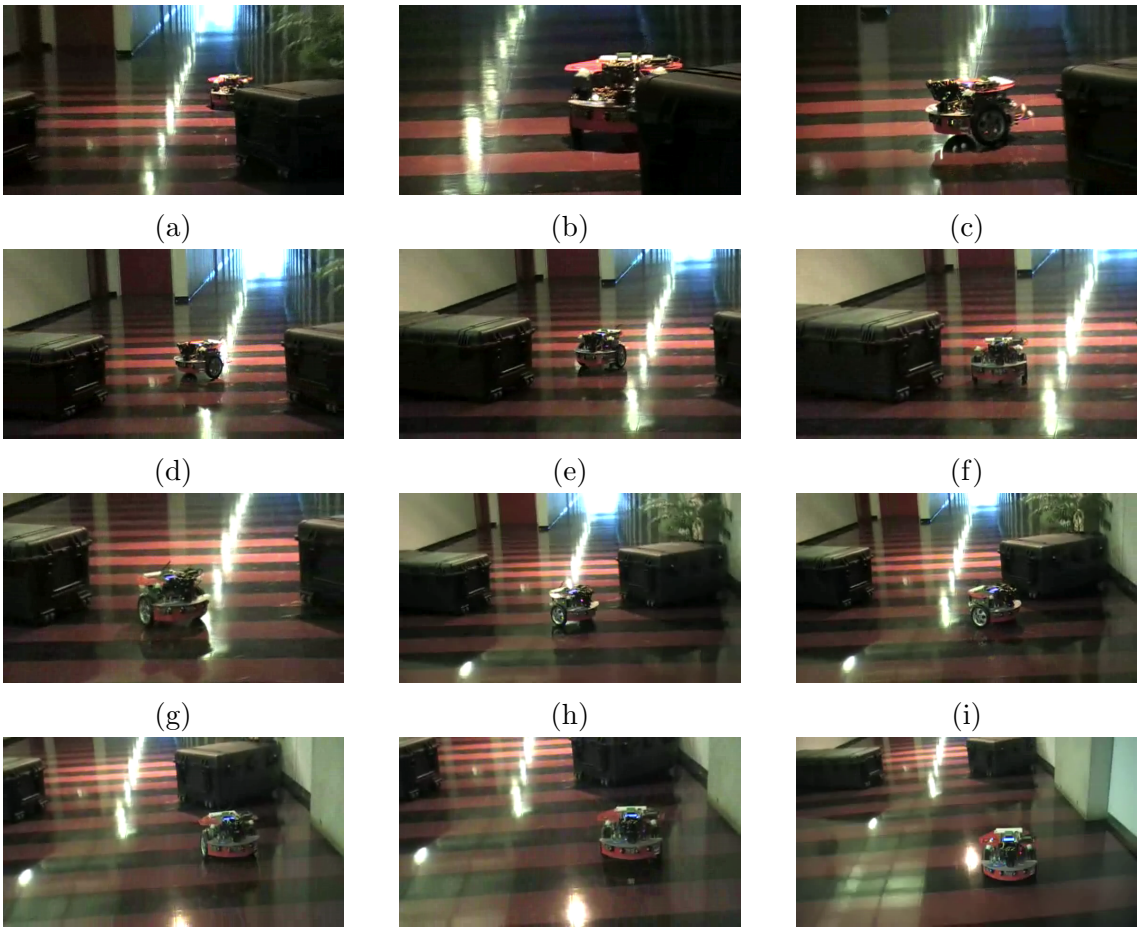


(a) (b) (c)

Figure 8.13: Concave obstacle geometry too small for the robot to escape

8.5.3 Single Robot Travelling Through a Narrow Passage

Finally, the last experiment tested how the X80 handled the narrow passage obstacle. A narrow passage was constructed using boxes, and various passage widths were tested. Figure 8.14 shows snapshots of the X80 travelling through the narrow passage to the goal position in front of the camera.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 8.14: X80 travelling through a narrow passage

Chapter 9

Results for Geometric Formations

This chapter will look at geometric formation control using the Leader-Follower formation control algorithm developed in Chapter 4. The chapter is organized in two parts: section 9.1 discusses the simulation and experimental results for the leader follower formation travelling through a terrain with no obstacles, and then section 9.2 will discuss the simulation and experimental results for reconfiguring geometric formations navigating through a terrain with obstacles. In this case, the leader navigates to the goal using the Velocity Potential navigation algorithm as seen in Chapter 8, and the followers will organize themselves in a specified geometric shape behind the leader.

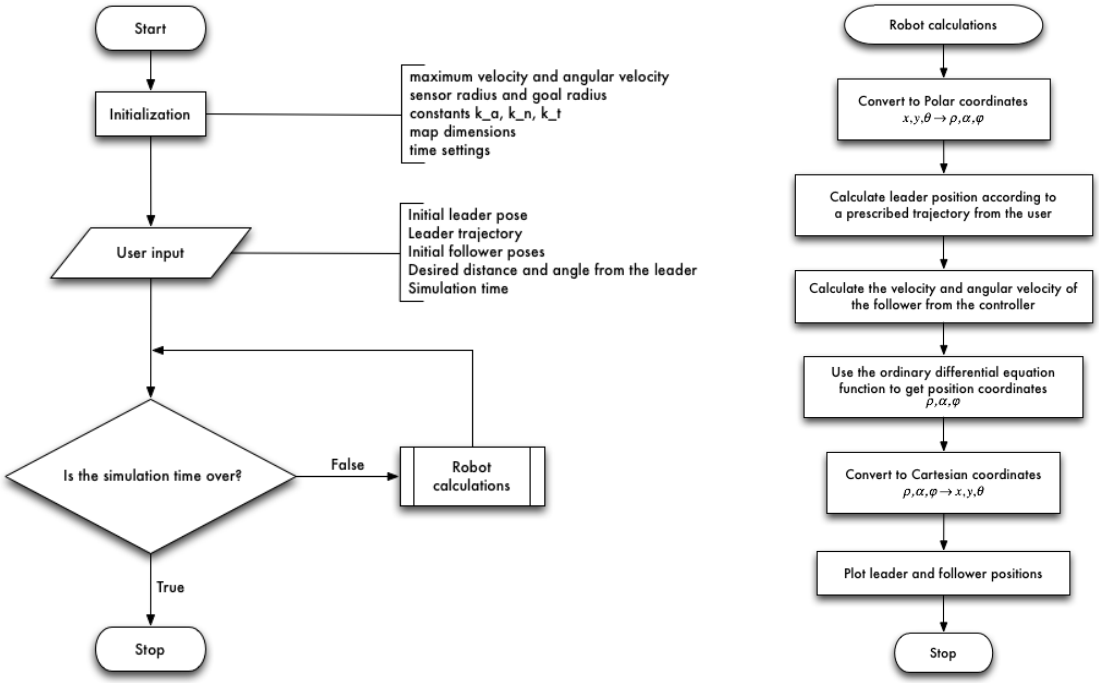
9.1 Leader-Follower Formations

9.1.1 Simulation Algorithm

Simulations for the Leader-Follower geometric formation were first carried out in MATLAB using the methodology described in Chapter 7. In this simulation, two robots are plotted in the MATLAB figure window, a red leader robot and a blue follower robot. The leader is given a prescribed trajectory, and the follower tracks the leader a specified distance and angle to follow the leader.

The main function of the LF simulation is shown in figure 9.1a. In the main function constants like the maximum linear and angular velocity, sensor and goal region of interest, controller constants, and map dimensions are set. Further the user can input robot variables such as the leader initial pose and velocity profile, the follower initial pose, and the desired distance and angle for the follower behind the leader. The main calculations occur in a loop that continues until the simulation time runs out.

The calculations in the loop are shown in figure 9.1b. The simulation first converts cartesian coordinates into polar coordinates. The leader position is then calculated based on the prescribed velocity inputs. Once the leader position is known, the follower calculates its velocity parameters based on the equations from Chapter 4. Polar coordinates are obtained using the ordinary differential equation function in polar coordinates, and then coordinates are converted back into cartesian coordinates. The leader and follower positions are plotted in the MATLAB figure window.



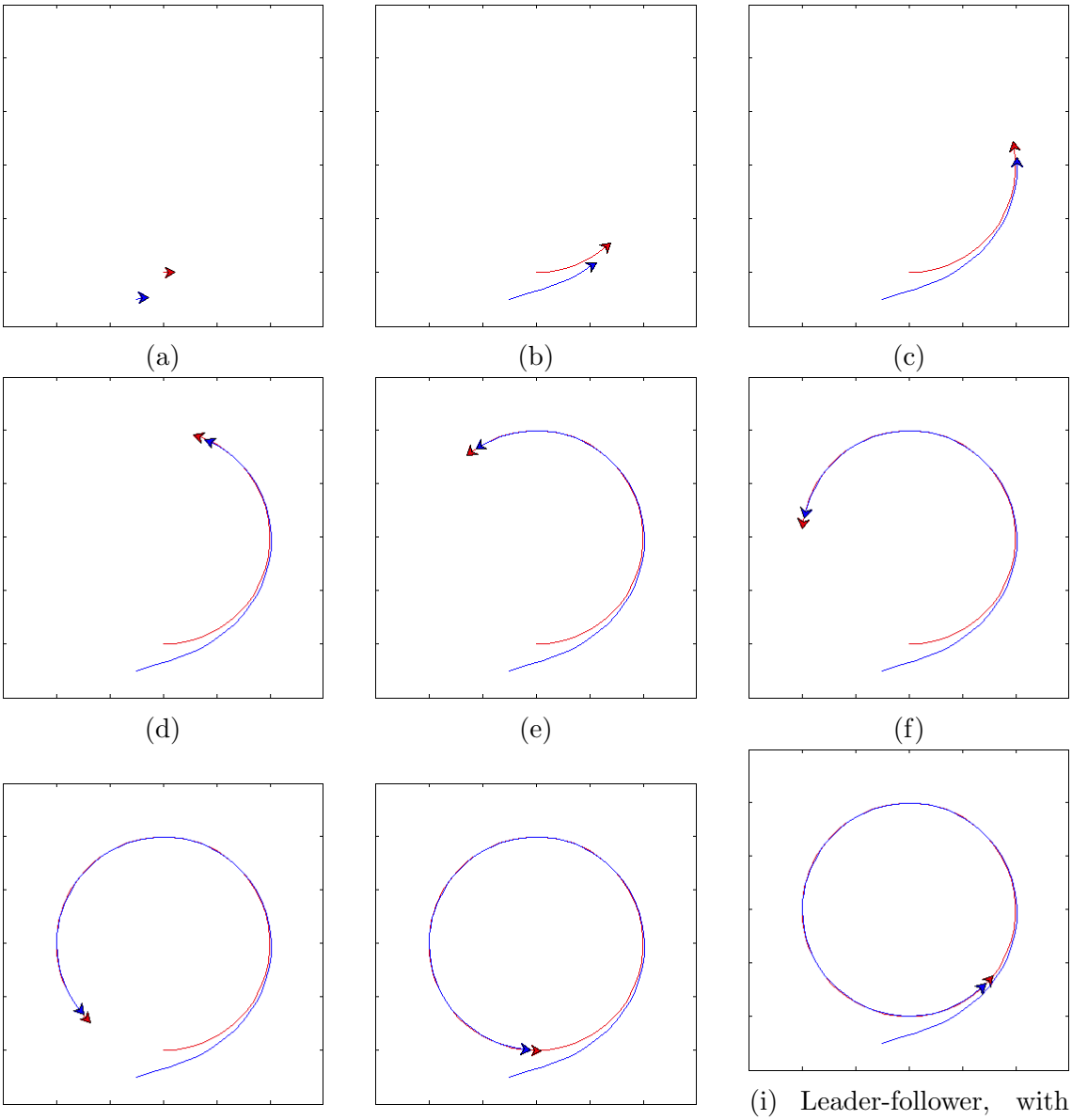
(a) (b)

Figure 9.1: (left) main function, (right) robot calculations

9.1.2 Simulation Results

9.1.2.1 Leader-Follower - 0 Degrees

The kinematic model and controller equations for the Leader-Follower setup were programmed in MATLAB to assess their validity. In the simulation we have two robots, one Leader (red) and a Follower (blue) at a random distance behind the Leader. In the first case, we set the Leader and Follower to be initially oriented in the same direction ($\theta = 0$). We set the desired distance between the Leader and Follower to be 1 m, and the angle to be zero so that the Follower would lie directly behind the Leader. Once the Leader vehicle began to move, the Follower adjusted its velocity and angular velocity to catch up with the Leader, and in figure 9.2 we saw that the two robots were moving in-sync with the Follower at a constant distance behind the Leader.



(a) (b) (c) (d) (e) (f) (g) (h)

(i) Leader-follower, with leader at a 0 degree angle with the leader

Figure 9.2: Leader-follower, with the follower at a 0 degree angle from the leader

9.1.2.2 Leader-Follower - 180 Degrees

Several other initial orientations were tested to see if the Leader-Follower model was a success. The most interesting results occurred when the Follower vehicle was initially pointing in the opposite direction of the Leader vehicle. The results pointed out the validity of Brocketts theorem[6], nonholonomic systems cannot be

asymptotically stabilized around a fixed point under any smooth time-independent state feedback control law. A continuous solution for asymptotic stability does not exist for nonholonomic vehicles, and instead, discontinuous or time-varying inputs should be applied.

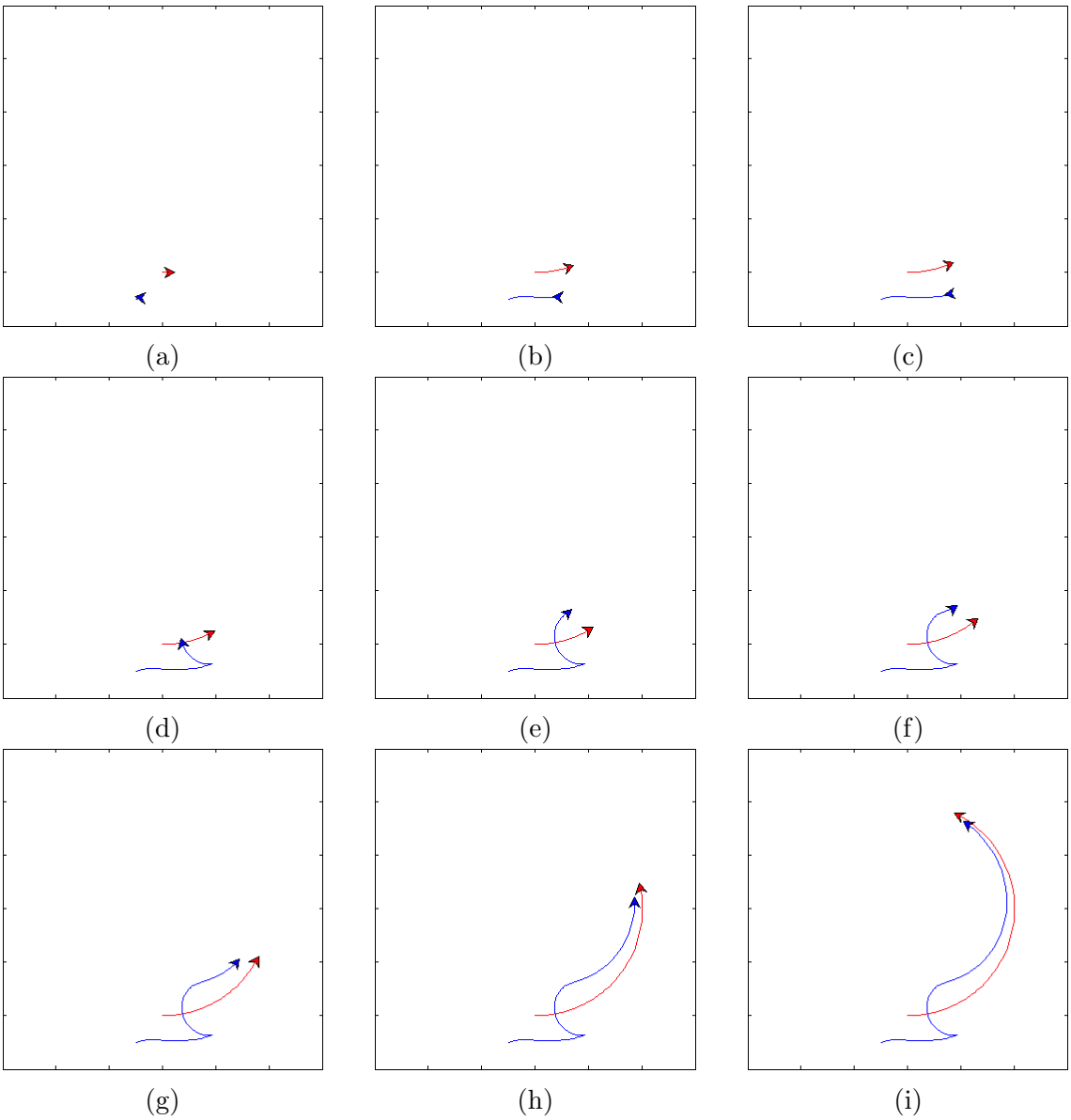


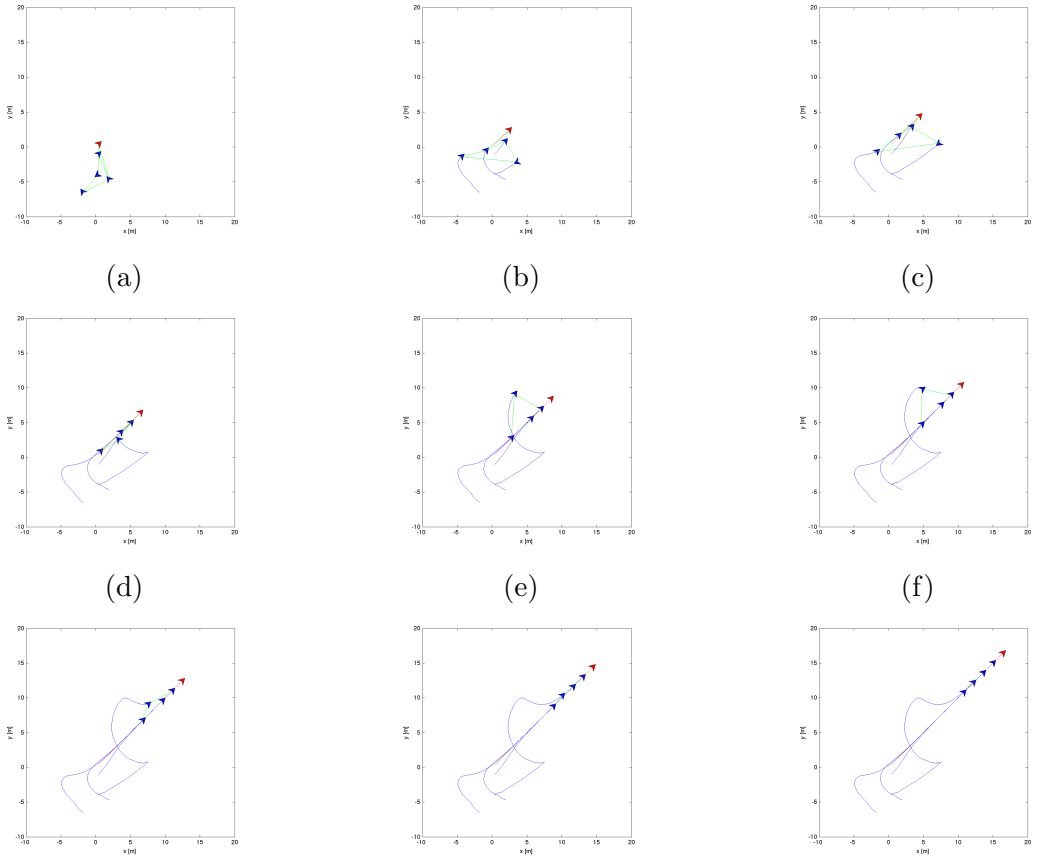
Figure 9.3: Leader-follower, with the follower at a 180 degree angle from the leader

In our simulations, when the Follower vehicle was positioned in the opposite direction to the Leader vehicle, the Follower would need to take a series of discontinuous movements in order to approach the Leading vehicle. For the case when the

Follower was positioned at 135 degrees from the Leader, the Follower was required to stop and let the Leader go ahead before the Follower began moving again behind the Leader. In the case where the Follower was positioned at 180 degrees from the Leader, the Follower initially moved backwards, stopped, then turned and moved in a forward direction to steer itself behind the Leader. Figure 9.3 shows the movement of the follower robot when it is initially positioned 180 degrees from the leader robot.

9.1.2.3 Platoon Formation

A platoon is a formation where a group of vehicles lines up one behind the other. It is a very simple and effective formation that is used when a group of vehicles must travel on a road, or through a tight space. The platoon is a direct application of the Leader-Follower formation, with one Leader that can be placed either in the front, middle, or end of the chain, and multiple Followers that follow the Leaders motion. In this simulation five nonholonomic vehicles are used, where four Followers (blue) are randomly placed behind a Leader (red) that moves in a straight line. By implementing the Leader-Follower controller, each Follower vehicle moves in its own specific way behind the Leader with the goal of orienting themselves at their position behind the Leader. The key to achieving this formation is to implement controller variables $rho_{desired} = 1$ m and $alpha_{desired} = 0$. In this simulation, the first, second, and fourth following vehicles were able to steer themselves quite easily behind the Leader due to their initial orientations. However, due to its initial pose, the third vehicle was required to move backwards for a short while, stop, then turn around and move forwards again to pursue the Leader, demonstrating several discontinuous motions.



(a) (b) (c)

(d) (e) (f)

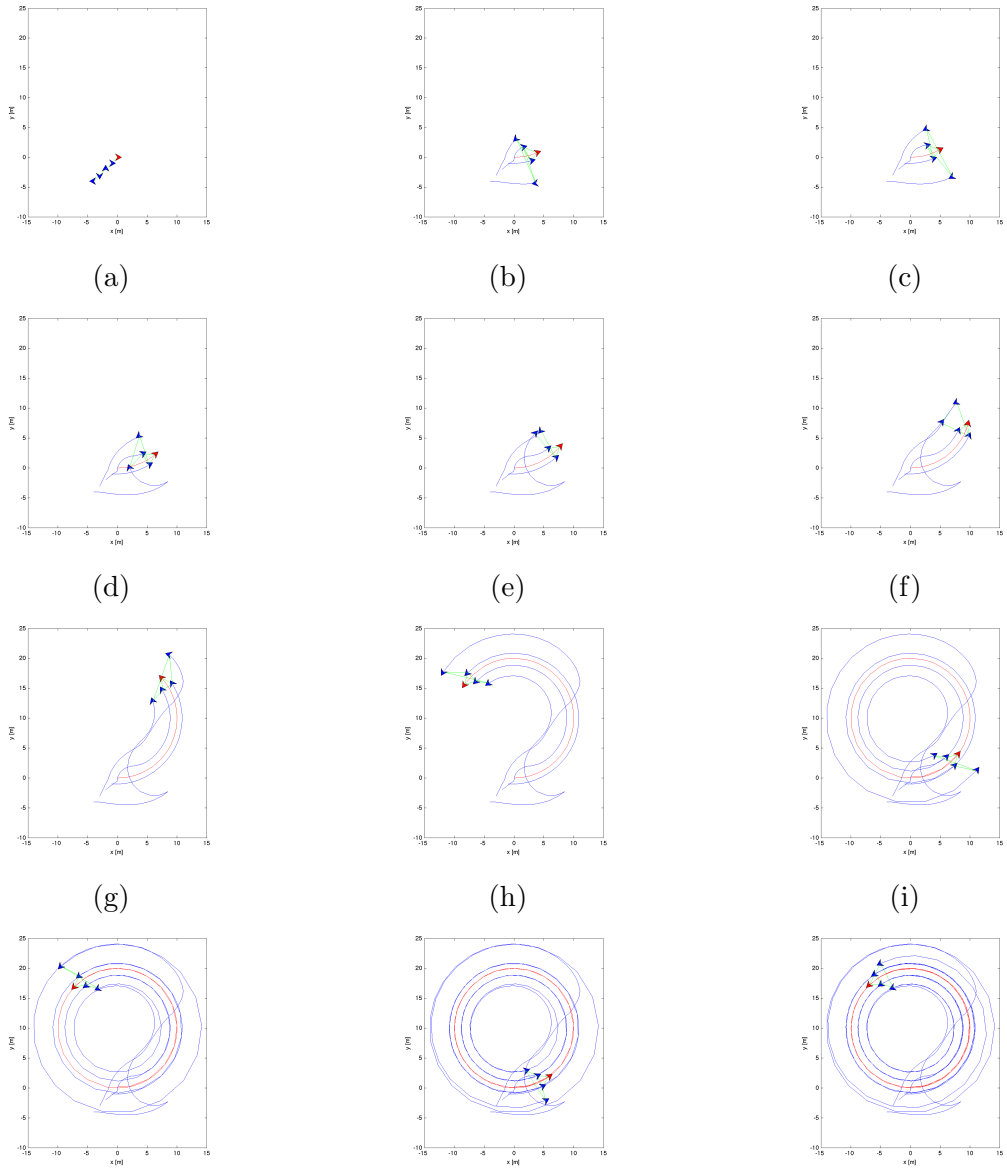
(g) (h) (i)

Figure 9.4: Platoon formation

9.1.2.4 V-Formation

The V-Formation is a geometry taken from nature that resembles Canada Geese flying south. The V-Formation is often used during the transition time, when string of vehicles moving in a platoon wish to spread out into a horizontal line similar to soldiers marching. The formation has one Leader at the front of the pack (red) and four Followers positioned at random behind the Leader. In this simulation, the Leader is moving in a circle at a constant velocity, and the Followers try to orient themselves behind the Leader while creating a V-shape. The key to achieving this formation is to implement the Leader-Follower controller using variables $rho_{desired} = 1$ and $alpha_{desired} = pi/6$ rad. Similar to the platoon, two of the Followers are

able to orient themselves quite easily behind the Leader by moving in a forward direction, while the two other vehicles are required to take a series of discontinuous movements, moving backwards and turning around, before they can get to the right position behind the Leader.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 9.5: V-formation

9.1.3 Experimental Algorithm

Geometric formation experiments using the Leader-Follower controller were implemented on the X80 mobile robot. In the simulations we used five robots to show the over geometric shape and movement of the robots through the terrain. For experiments we had three differential drive X80 robots, so all experiments consisted of one leader and two follower robots. The formation controller was programmed in C++ using the methods described in Chapter 7. Figure 9.6 shows the main function in the geometric formation controller. The main function initializes parameters and asks the user for inputs such as the goal position and the number of follower robots. The function then calls the Player Proxies to set up the network connection between the robots and Personal Computer (PC). Finally, the majority of the calculations for robot navigation were carried out in a loop that calculated the leader and follower velocities. The loop continued until the robot reached the goal position, then all robot velocity commands were set to zero, and the experiment was closed.

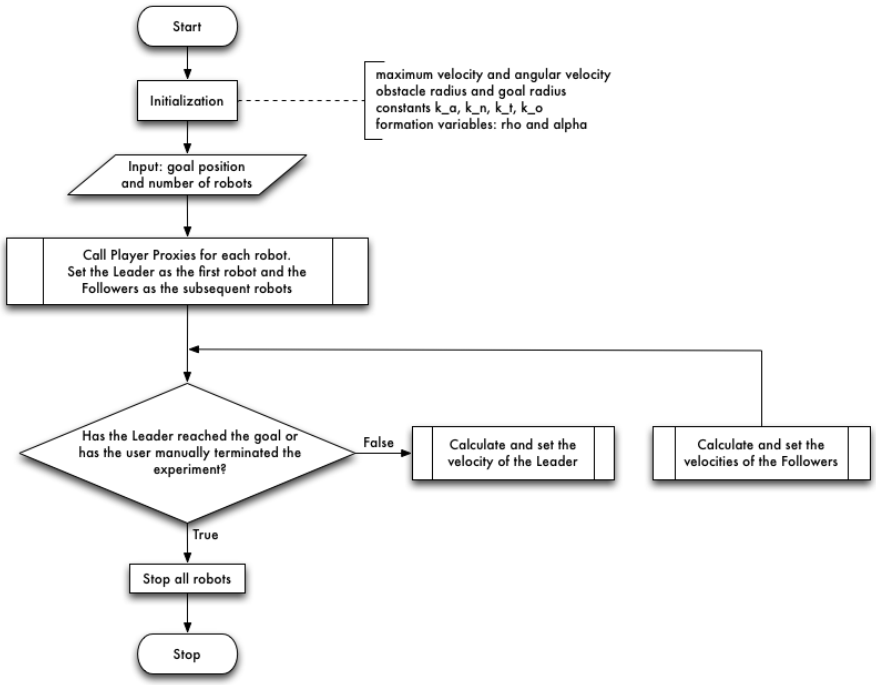


Figure 9.6: main function

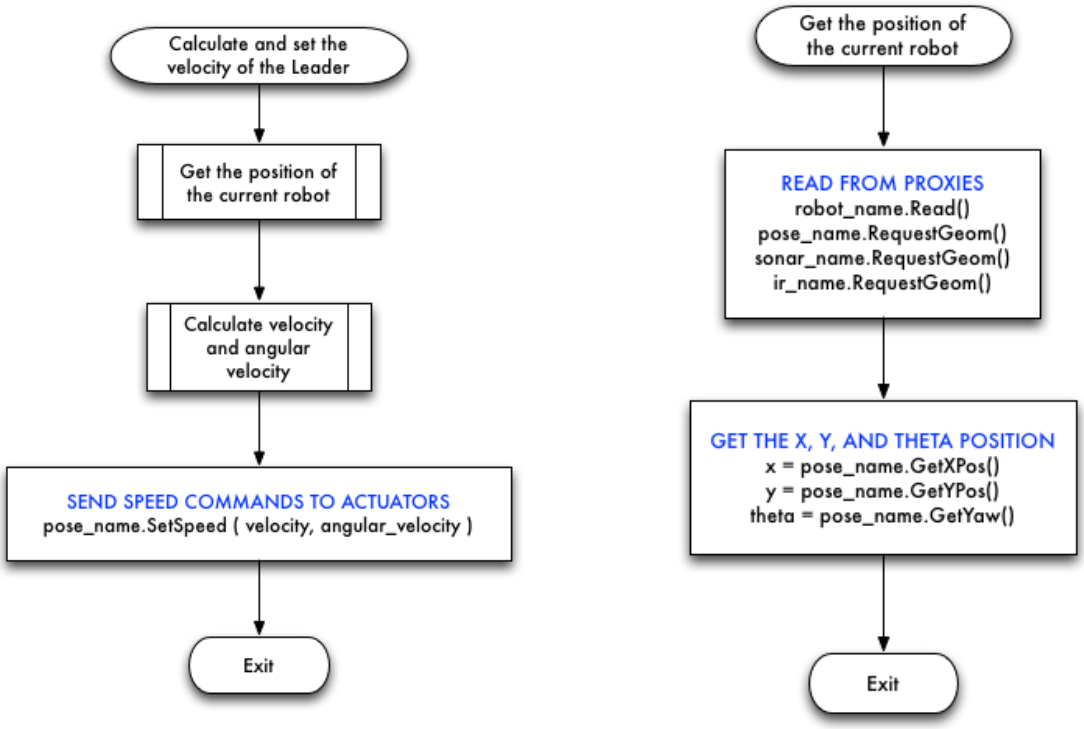


Figure 9.7: (left) calculate leader, (right) get leader pose

In the main loop, the controller calculates and sets the velocity of the leader and follower robots. The steps to carry out these tasks are shown in figure 9.7a and figure 9.9. In figure 9.7a, the leader velocity is calculated by first obstaining the current position of the leader robot. The current position can be found using the methodology of figure 9.7b, where the controller can access the (x, y, θ) position variables using the GetXPos(), GetYPos, and GetYaw() methods from the Position2dProxy. Once the position of the leader robot is found, the controller calculates the velocity and angular velocity based on the algorithm in figure 9.8. The velocity of the follower robots is calculated by first accessing its position variables, and then applying the follower velocity controller from Chapter 4. The leader and follower velocities are then implemented using the SetSpeed() method in the main controller function.

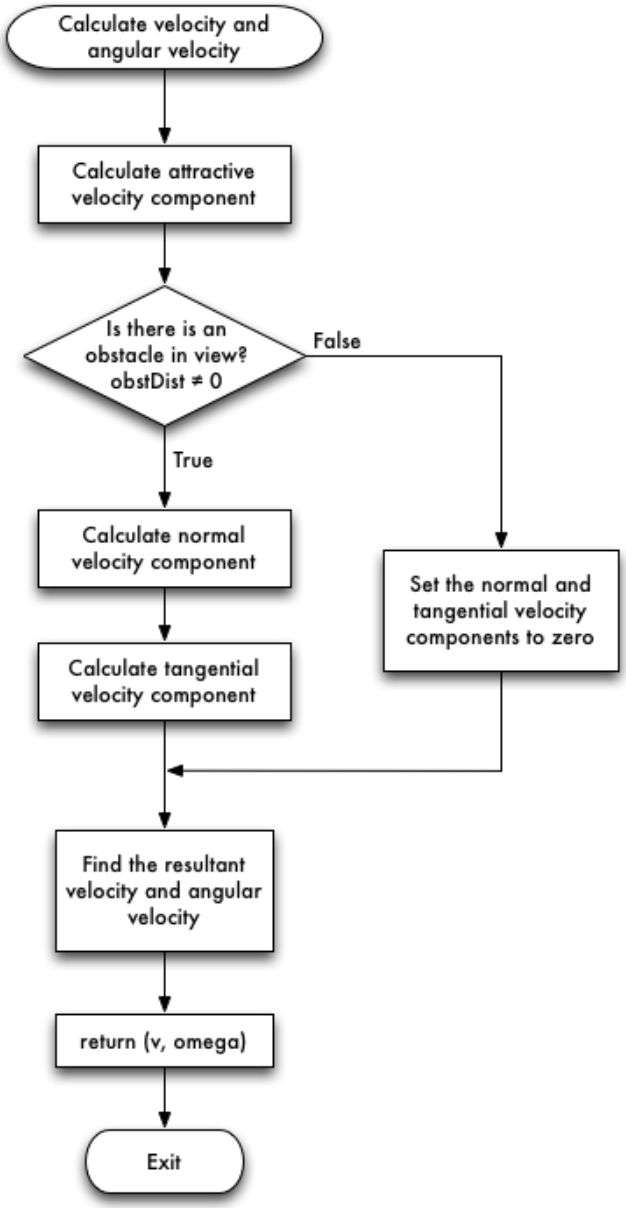


Figure 9.8: Calculate velocity and angular velocity

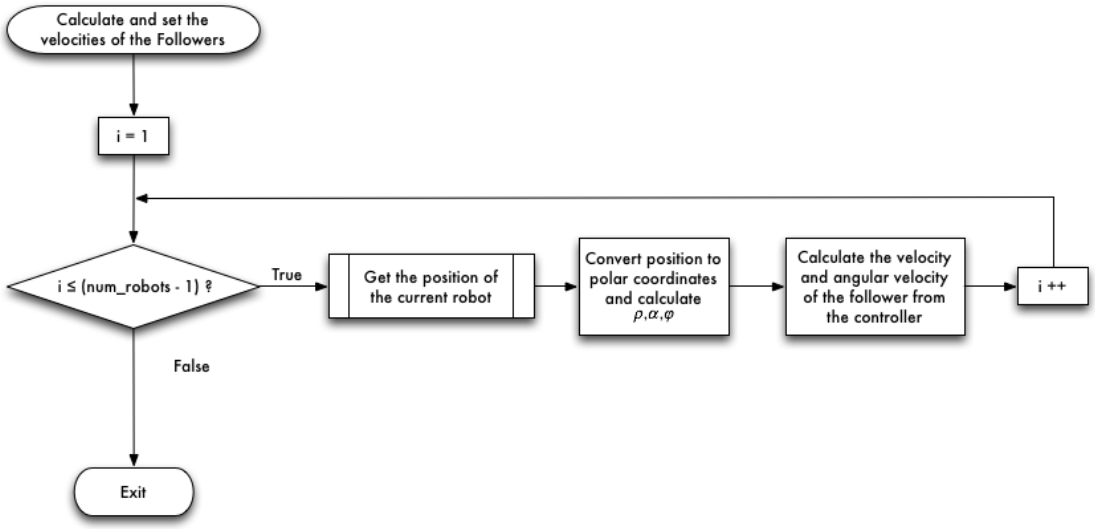
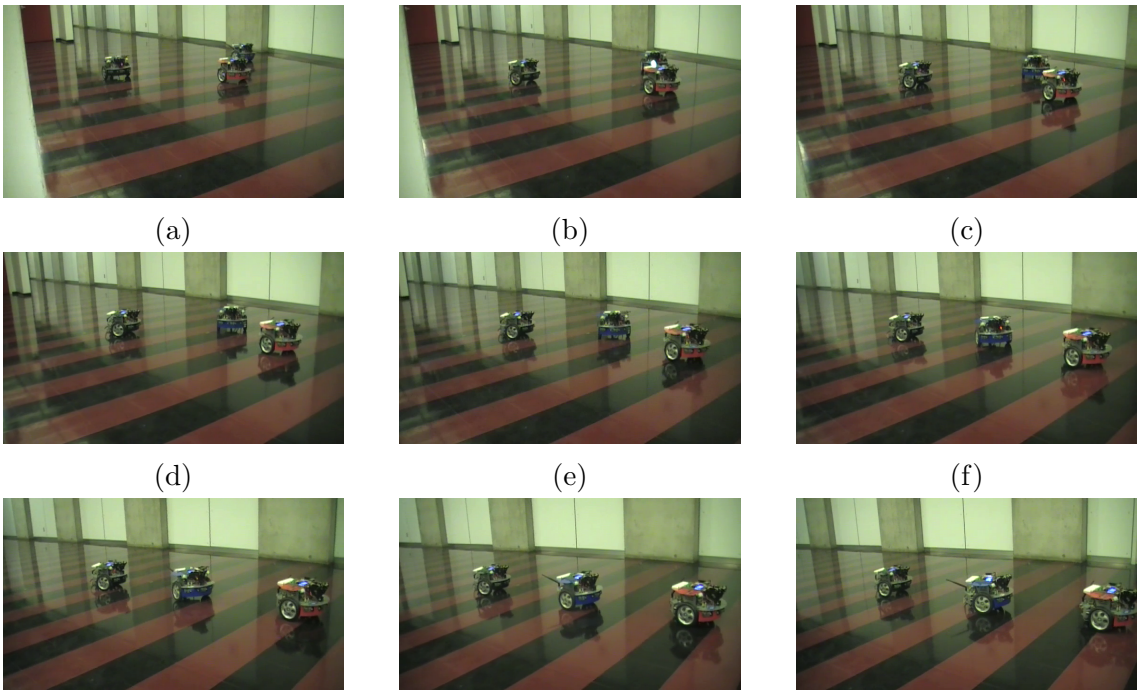


Figure 9.9: Calculate the velocity of the follower

9.1.4 Experimental Results

9.1.4.1 Platoon

In the first geometric formation control experiment with the X80 we wanted to test the platoon formation controller. In this experiment, the robots were initially placed in a V-formation with the red robot in the leader position at the front, and the blue and black follower robots were positioned behind the leader. The leader robot was set to go in a straight path with a constant velocity. When the leader began moving, the blue robot navigated itself behind the red leader. The black robot waited until the blue robot was ahead and then followed the formation behind the blue robot. Figure 9.10 shows the movement of the three robots to position themselves in a straight line platoon formation behind the red leader robot.



(a) (b) (c)

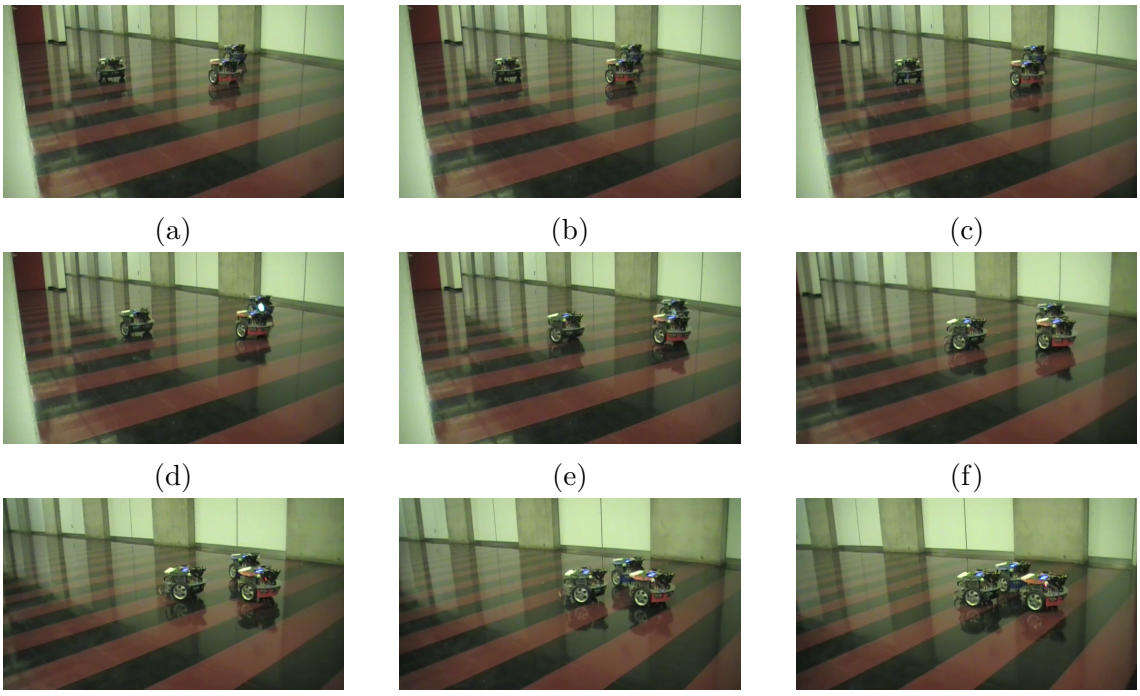
(d) (e) (f)

(g) (h) (i)

Figure 9.10: Platoon

9.1.4.2 V-Formation

In the next experiment, the V-formation was tested with the same initial setup as the Platoon formation. In this experiment, the leader robot travelled on a straight path with a constant velocity. The follower vehicles positioned themselves at a 45 degree angle from the leader position and with a 1 meter distance from the center of gravity of each robot unit. The experiment showed that the V-formation could be achieved with the X80 mobile robots using the Leader-Follower controller.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Figure 9.11: V-formation

9.2 Geometric Formation and Collision Avoidance

9.2.1 Simulation Algorithm

The next stage in the simulations was to incorporate collision avoidance for the group as they travelled together through the terrain. The velocity potential method was used to navigate the leader of the group to the goal while avoiding obstacles. The follower robots tracked the leader at a designated distance and angle, based on the desired formation shape. Figure 9.12 shows a flowchart of the main function of the geometric formation and collision avoidance algorithm.

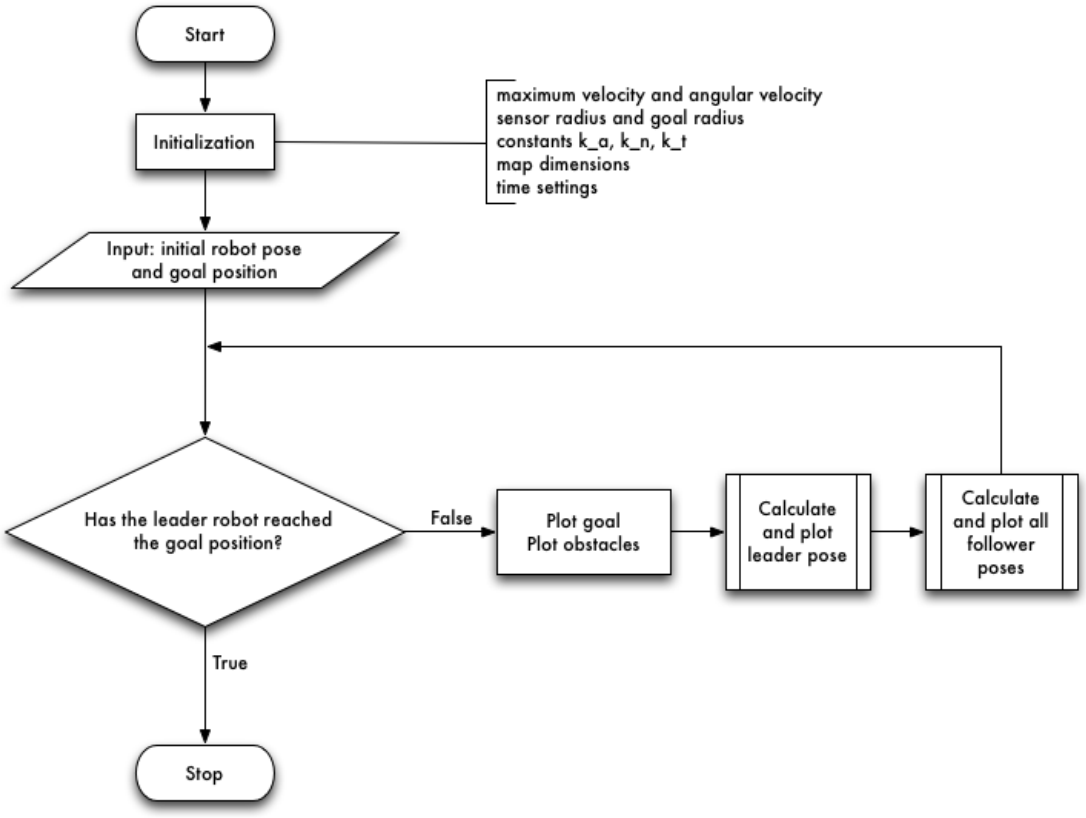
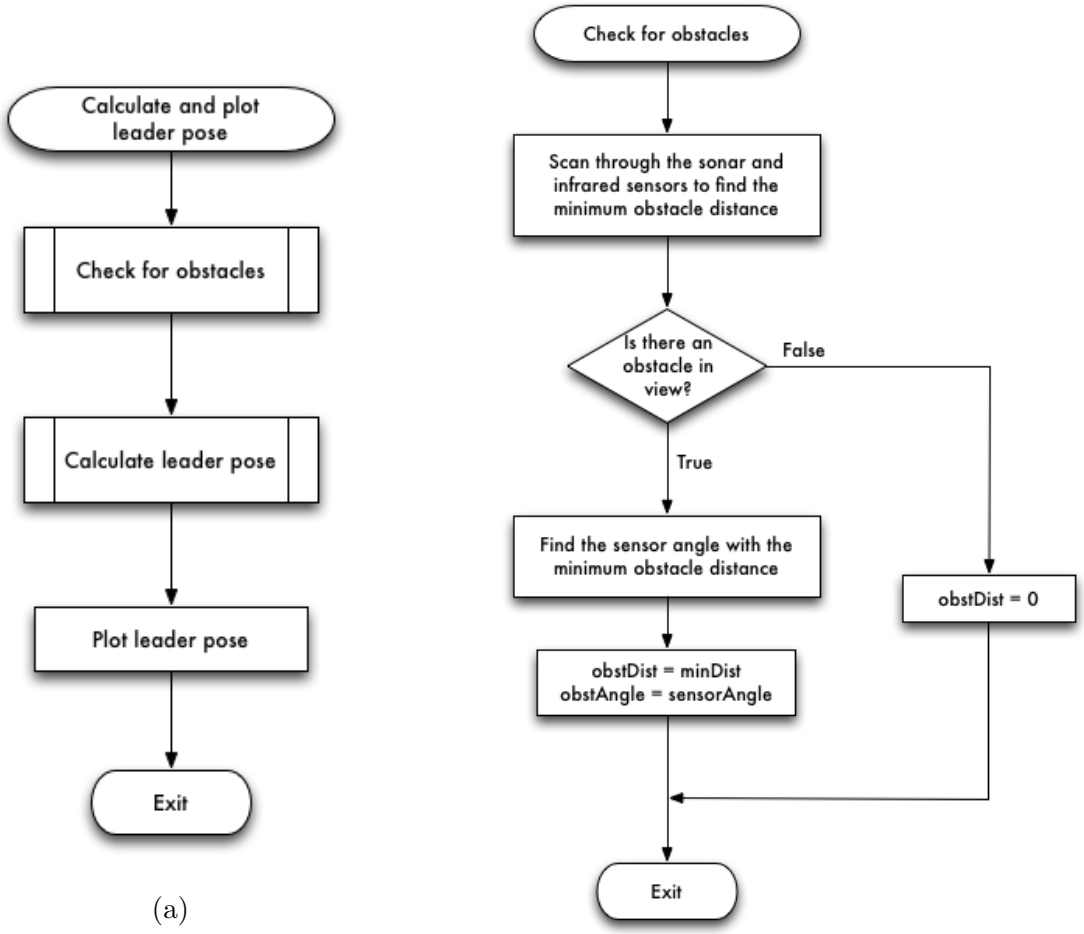


Figure 9.12: main function

The leader robot implements the reactive navigation strategy of the velocity potential method (figure 9.13a). Firstly the leader checks for obstacles (figure 9.13b) and uses the attractive and repulsive flow fields to determine its next pose (figure 9.14).



(a)

(b)

Figure 9.13: (left) calculate and plot leader pose, (right) check for obstacles

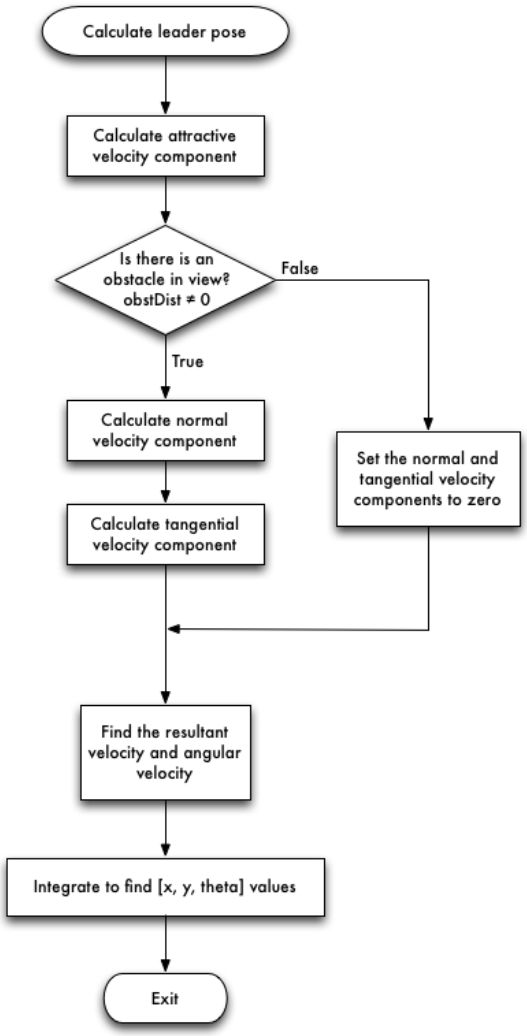


Figure 9.14: Calculate leader pose

Finally, the new follower pose is calculated based on the leader-follower controller shown in figure 9.15

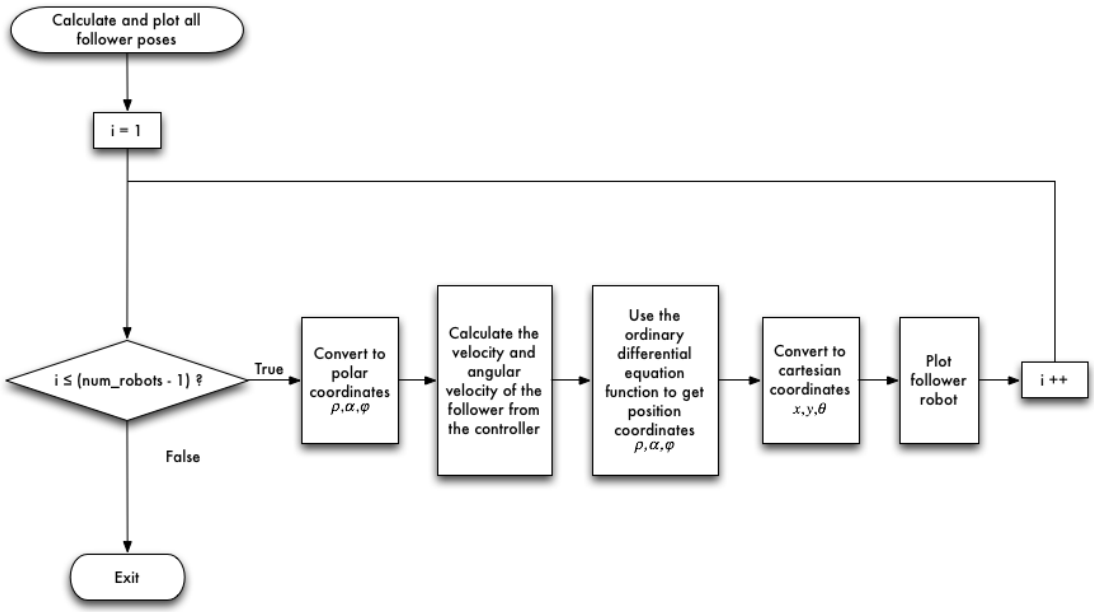


Figure 9.15: calculate follower pose

9.2.2 Simulation Results

9.2.2.1 Platoon

In the first simulation, a leader robot (red) navigates around a hexagon shaped obstacle to reach the goal position. The follower robots (blue) navigate around the obstacle by creating a platoon formation behind the leader. It was found that the platoon formation worked only for small groups of robots, and the longer the platoon chain, the more susceptible the formation was to collisions with obstacles. For a simulation with five robots, the platoon formation was able to successfully navigate around the obstacle and reach the goal position.

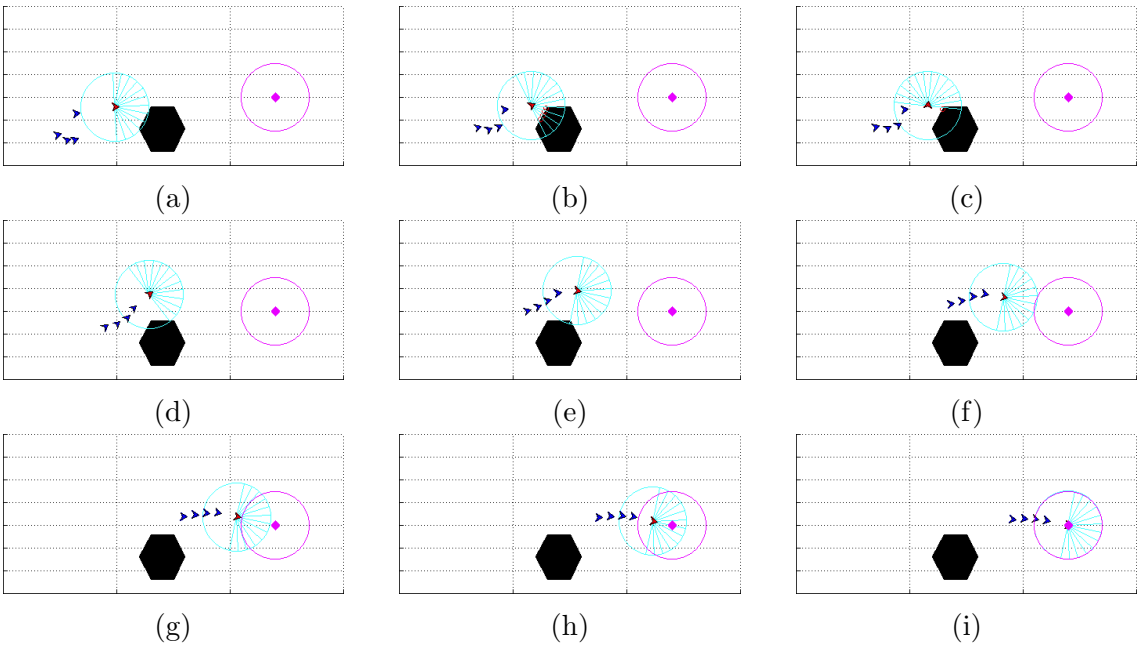
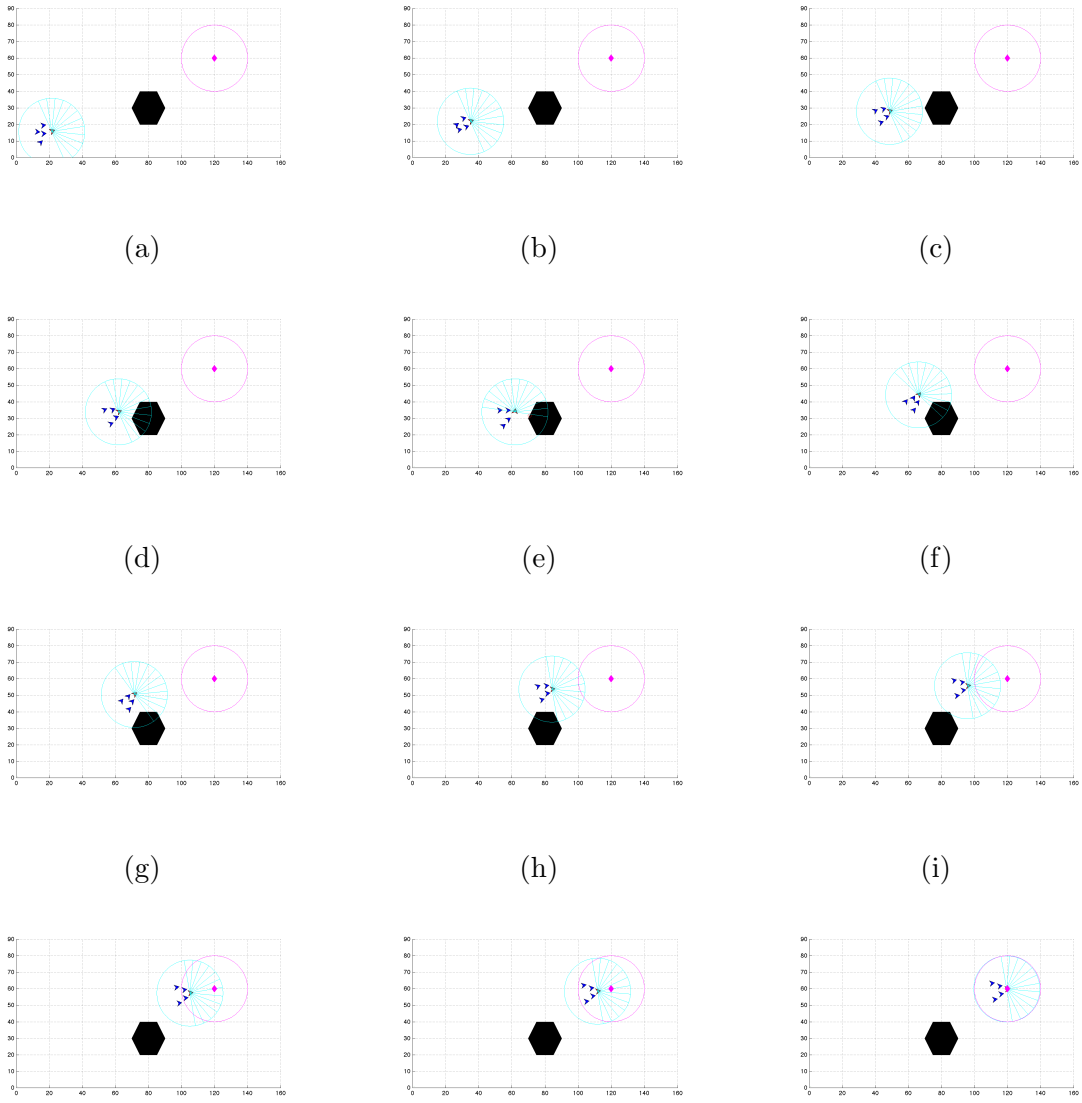


Figure 9.16: Platoon around an obstacle

9.2.2.2 V-formation

The next simulation tested how a V-formation could navigate around a hexagon shaped obstacle. Reconfiguring formations was a very important concept when designing the LF formation controller, and when designing the algorithm we set the angle of the V to change when an obstacle is in view. In the simulation, when

the group approached an obstacle, the angle of the formation shrank to more easily accommodate the group and allow them to navigate around the obstacle without collisions. Figure 9.17 shows snapshots of the LF simulation for a V-formation around an obstacle. The simulation successfully demonstrated that a V-formation could easily navigate around the obstacle as a group using the Velocity Potential navigation controller.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 9.17: V-formation around an obstacle

9.2.2.3 V-Formation Around a Concave Obstacle

The third simulation incorporated a concave obstacle for a team of mobile robots. Figure 9.18 shows snapshots of the LF movement around a concave obstacle. In this simulation the leader enters into the concave obstacle, and using the velocity potential navigation controller it finds a way to escape from the obstacle and to travel to the goal position. In the process, the follower robots back up to accommodate the leader, and then orient themselves in a V-formation behind the leader to reach the goal position.

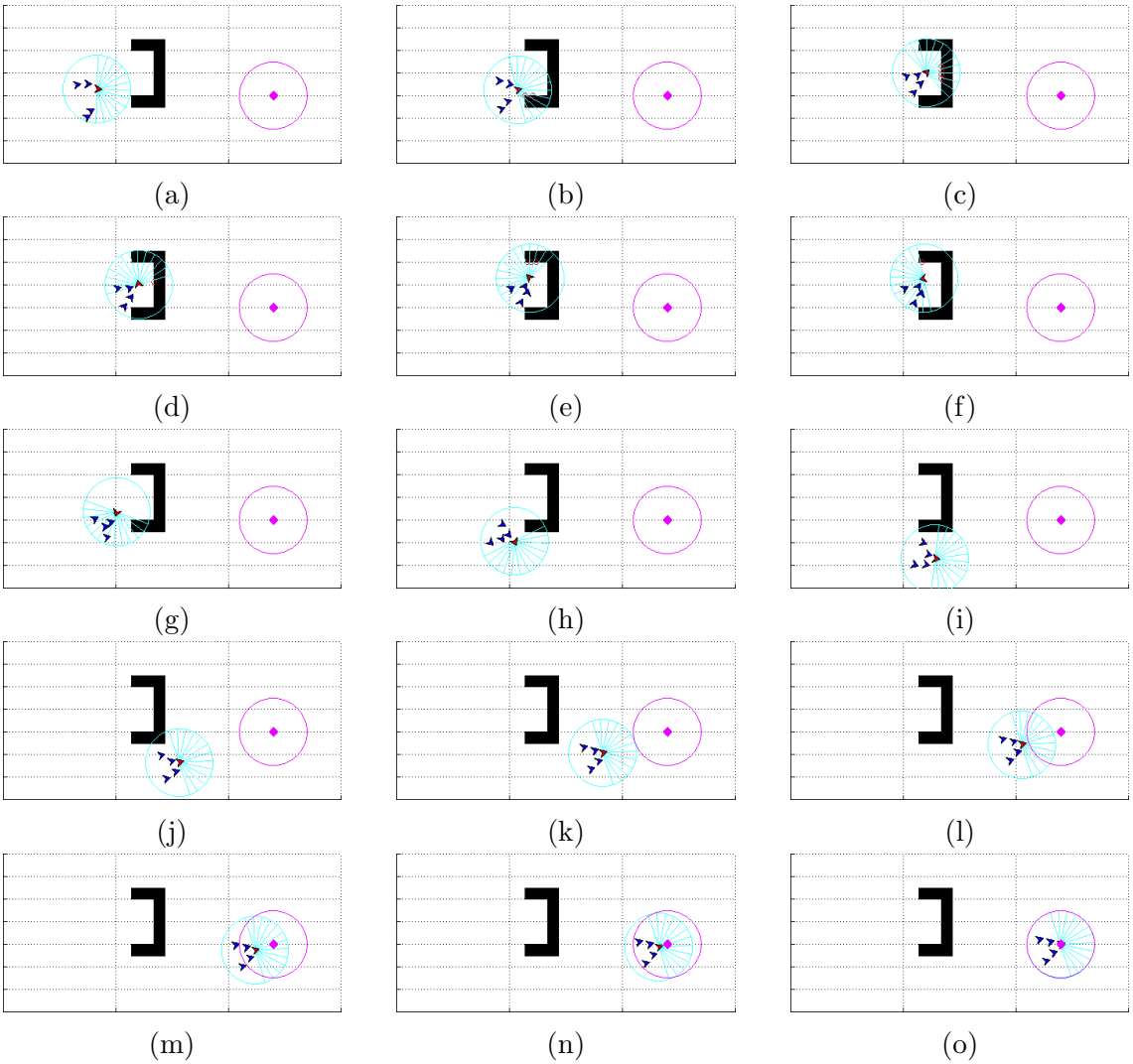
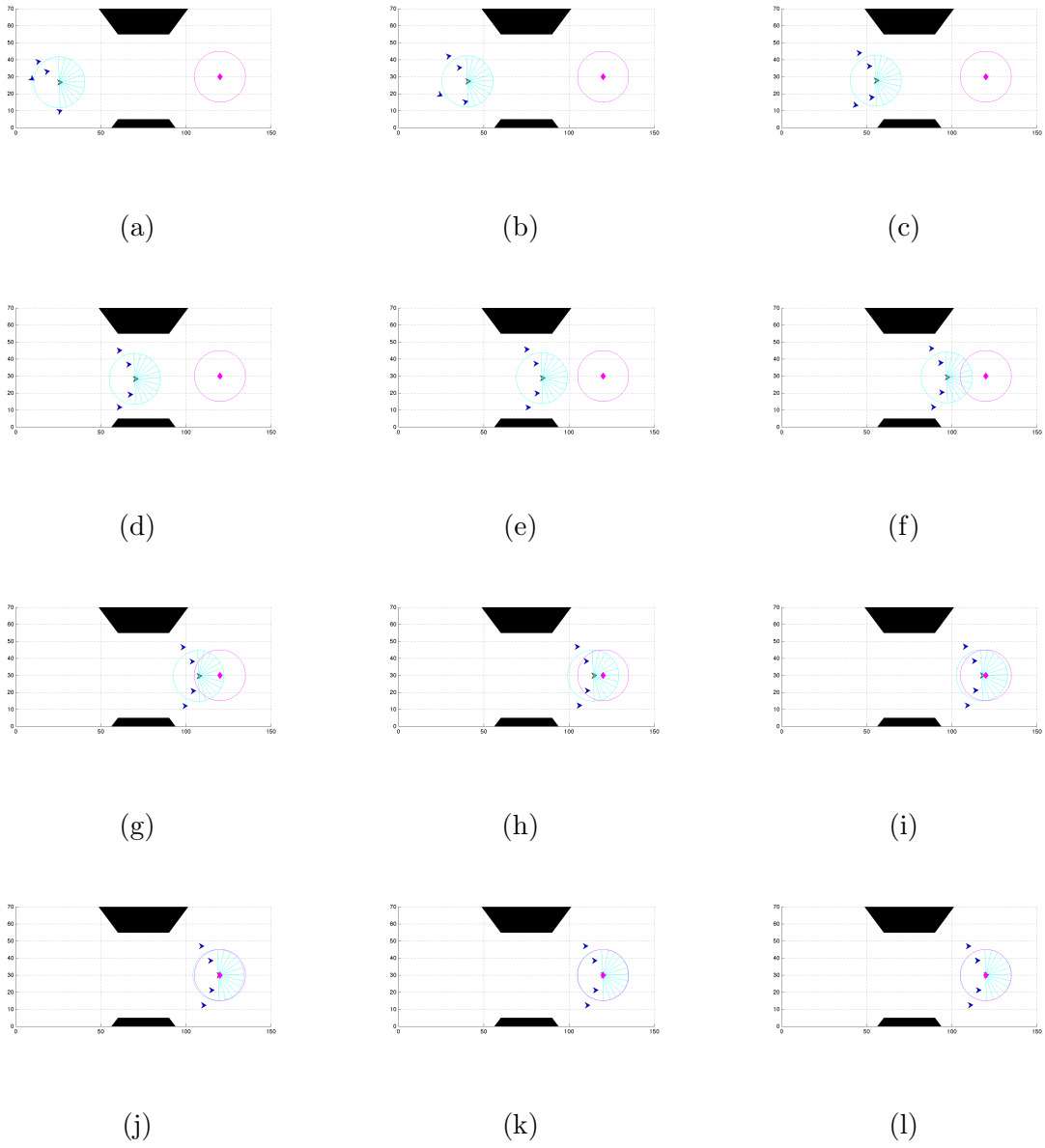


Figure 9.18: V-formation around a concave obstacle

9.2.2.4 V-Formation Through a Wide Passage

In the fourth simulation, a V-formation navigated through a wide passage. The angle of the V-formation was set to change according to the obstacle opening distance, determined by the leader robot. For the wide passage in figure 9.19, the opening was large enough that all of the robots could pass without having to change the angle of the V.



(a) (b) (c)

(d) (e) (f)

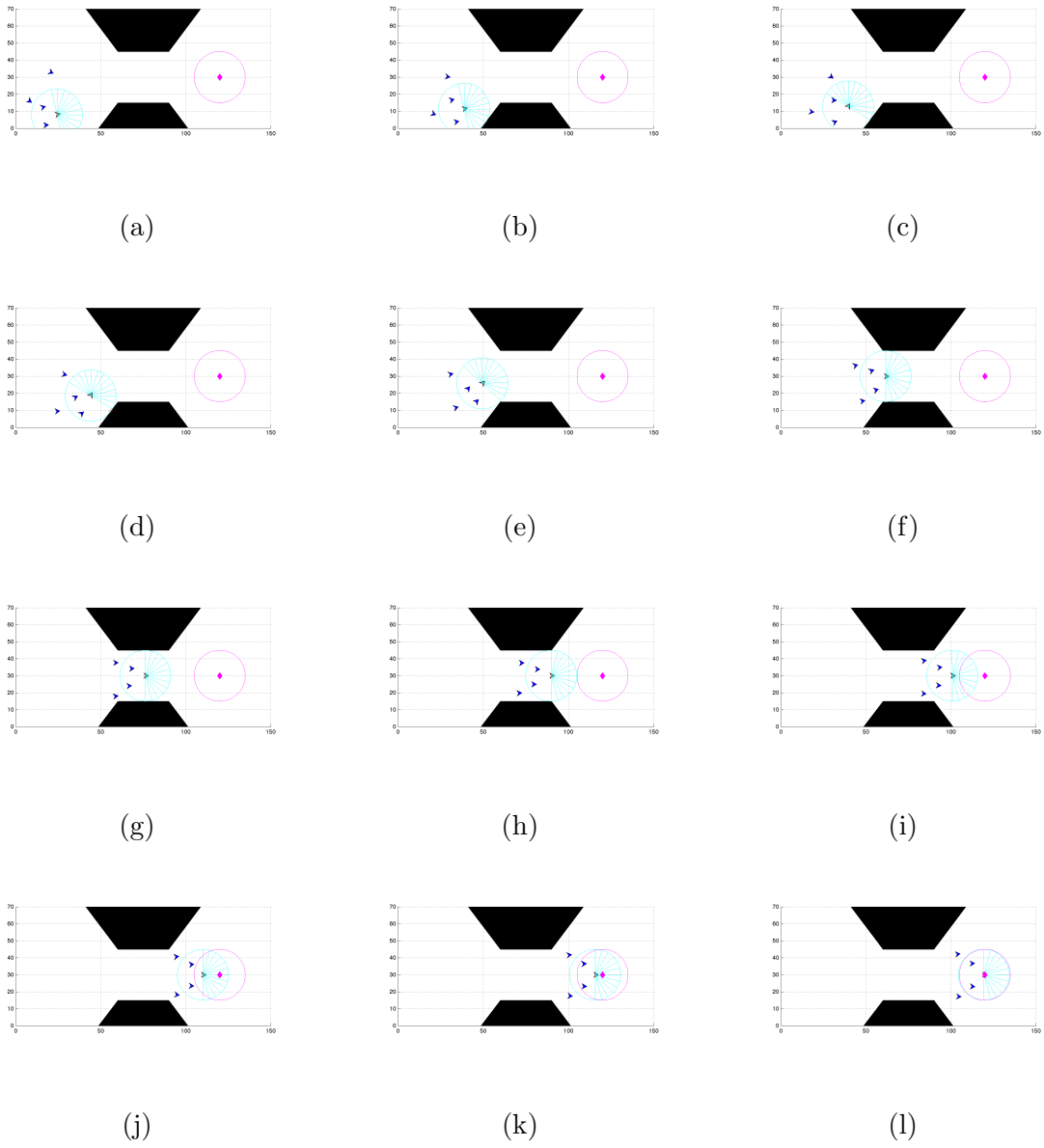
(g) (h) (i)

(j) (k) (l)

Figure 9.19: V-formation through a wide passage

9.2.2.5 V-Formation Through a Narrow Passage

For the fifth simulation, the V-formation was tested with a narrow passage. This narrow passage was small enough that the team of robots could not pass through with their original formation angle.



(a) (b) (c)

(d) (e) (f)

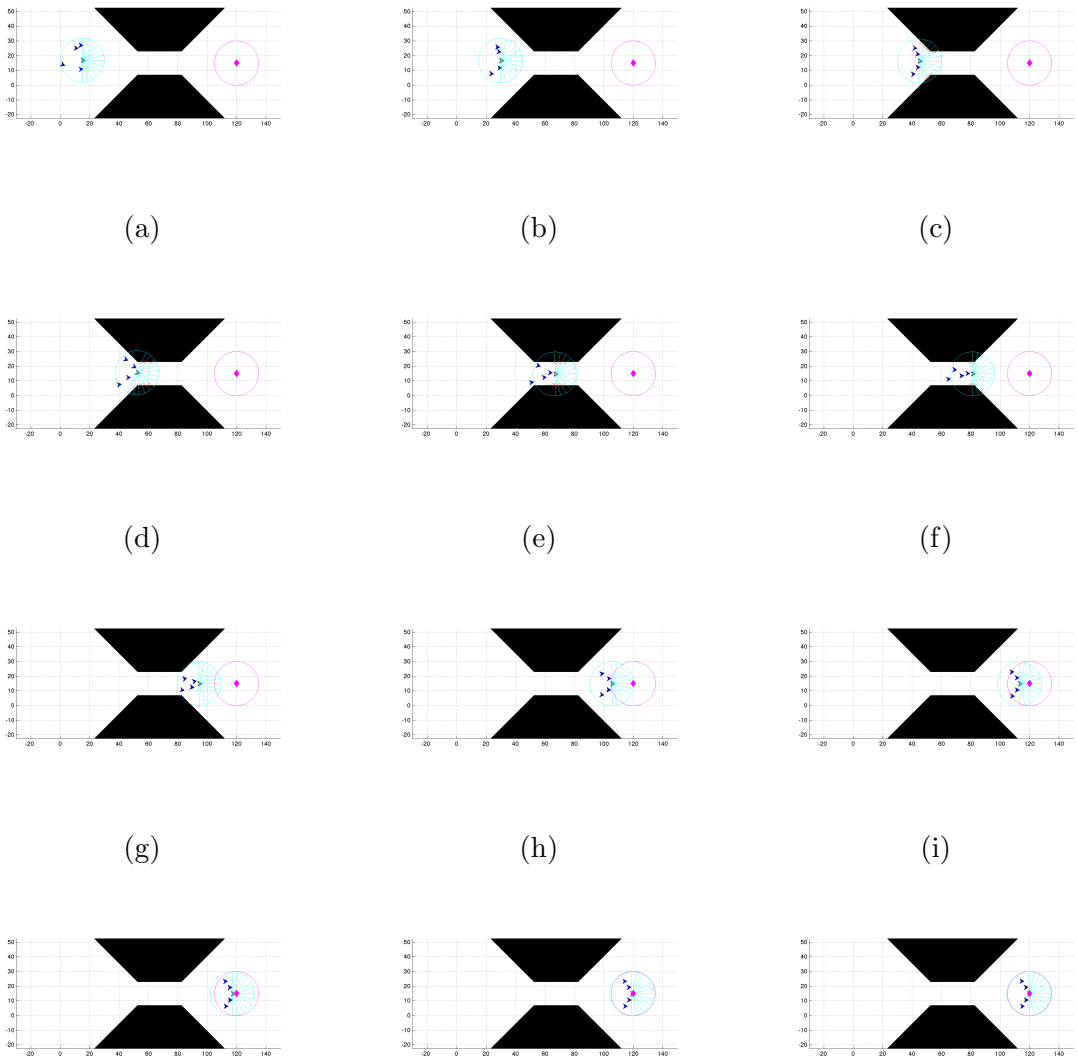
(g) (h) (i)

(j) (k) (l)

Figure 9.20: V-formation through a narrow passage

The leader robot determined the distance for the narrow passing using its sensor readings, and a desired V-angle was calculated for the group. The simulation

in figure 9.20 shows how the formation reconfigures to travel through the narrow passage, and then expands again when the obstacle is no longer in view.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 9.21: V-formation through the narrowest passage

9.2.2.6 V-Formation Through a Very Narrow Passage

Finally the last simulation examined a V-formation through a very narrow passage. In this simulation (figure 9.21), the passage opening is only large enough for a single

robot to pass through at a time. In order to navigate the group through this difficult obstacle, the robots position themselves into a platoon to go through the passage, and then once the obstacle is out of view, the formation reconfigures to its initial V-shape.

9.2.3 Experimental Algorithm

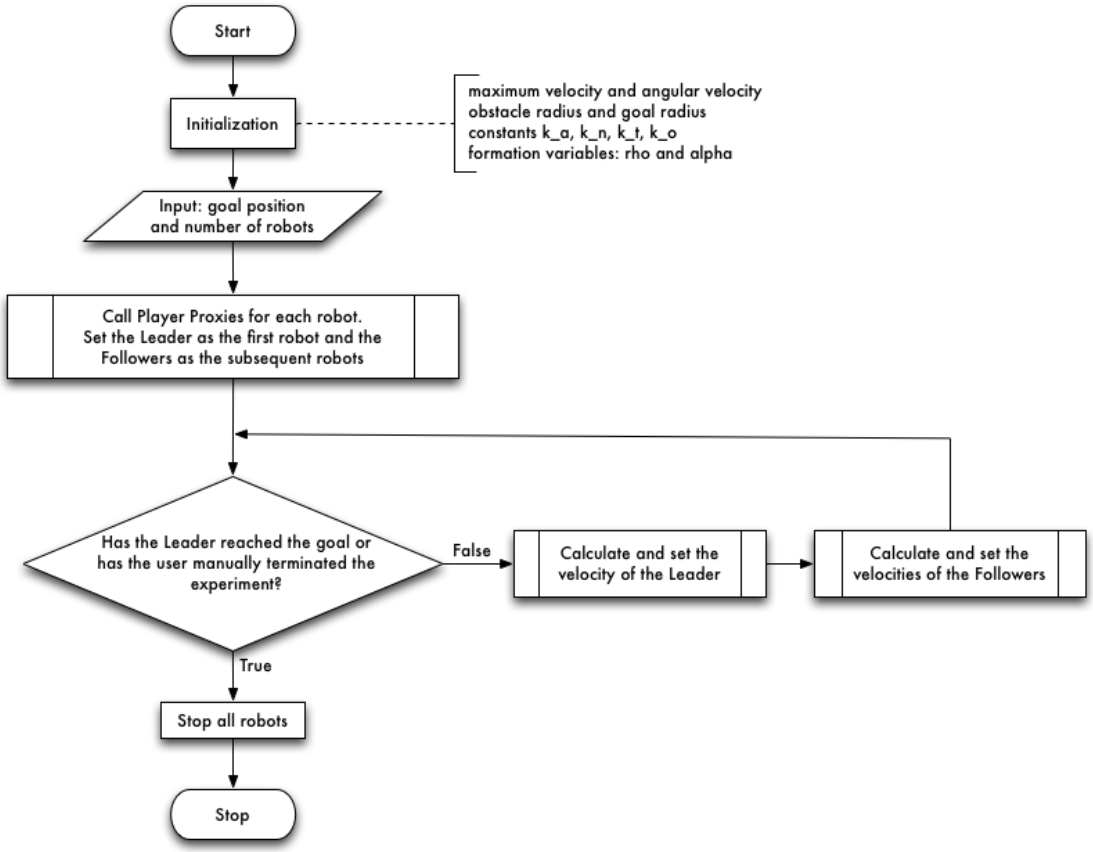
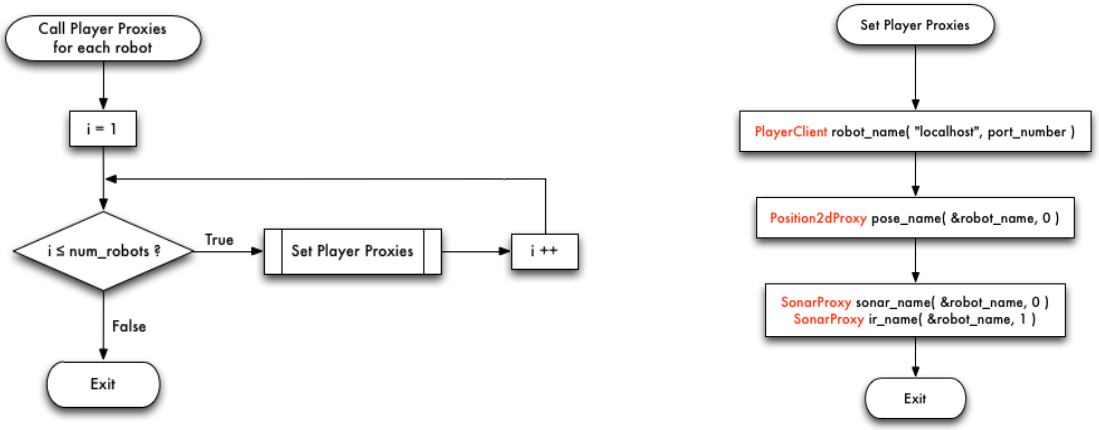
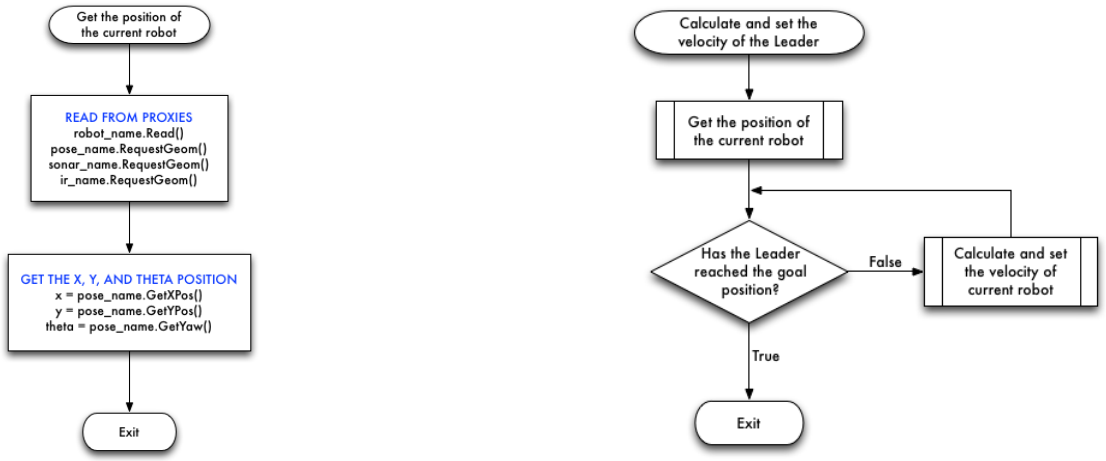


Figure 9.22: main function



(a) (b)

Figure 9.23: (left) call player proxies, (right) set player proxies



(a) (b)

Figure 9.24: (left) get the pose of the current robot, (right) calculate the velocity of the leader

The experimental algorithm for the Leader-Follower formation with collision avoidance follows a similar procedure to the previous experiments. In the main function (figure 9.22), parameters are initialized and the Player Proxies are called for the leader and follower robots. The velocities of the leader and follower robots are calculated in the loop, and the loop continues until the leader has reached the goal position.

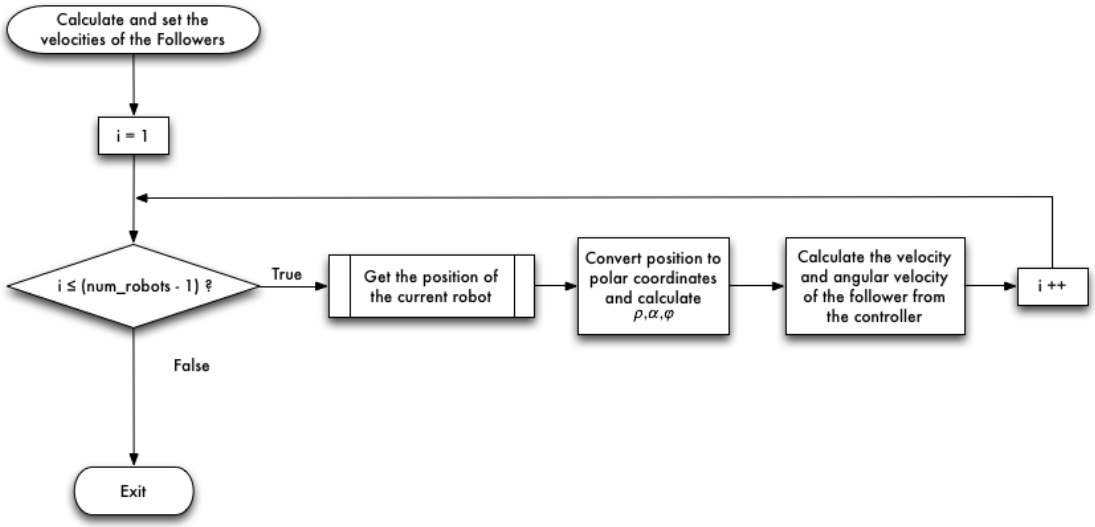


Figure 9.25: Calculate the velocities of the followers

The calculations for the leader and follower robots are displayed in figure 9.26a and figure 9.25. The leader employs a strategy where it checks for obstacles in the terrain and then calculates a navigation strategy through the obstacles. The follower robots rely on the following tracking controller to position the follower vehicles behind the leader at a desired distance and angle.

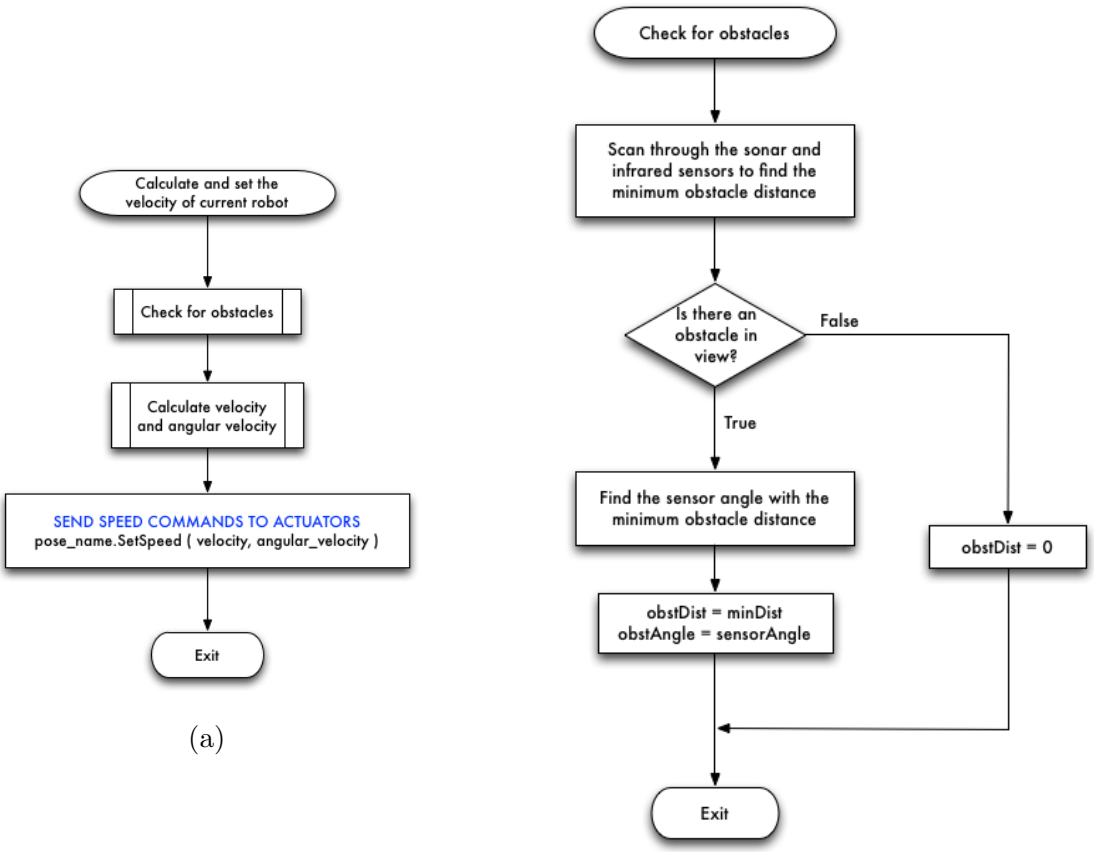


Figure 9.26: (left) calculate the velocity of the current robot, (right) check for obstacles

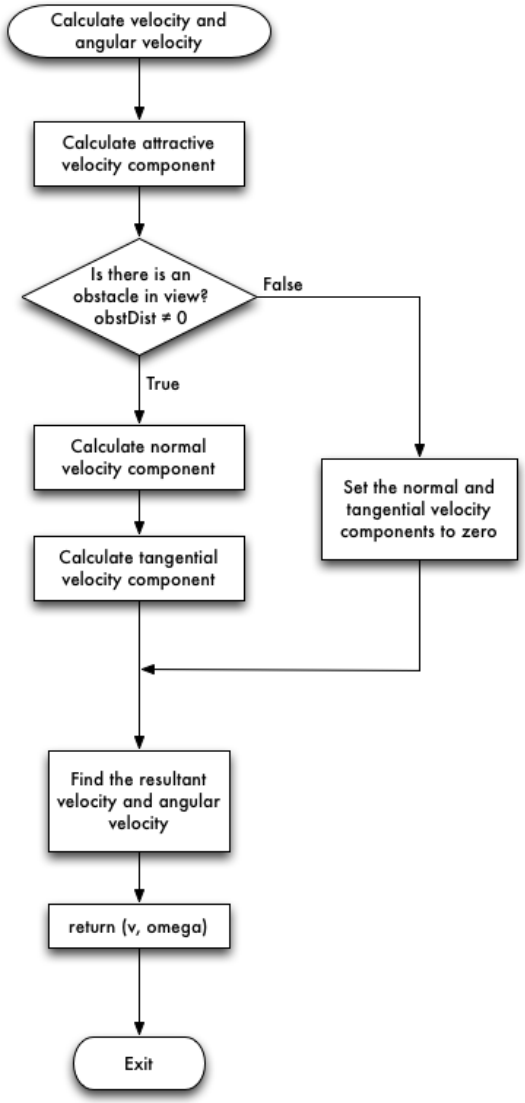


Figure 9.27: Calculate velocity and angular velocity

9.2.4 Experimental Results

9.2.4.1 Platoon Formation with Collision Avoidance

For the experiments with the X80 mobile robots, we first wanted to test a geometric formation navigating around an obstacle. A potted plant was used as the obstacle, and a platoon formation was chosen for the group. Using the LF formation with collision avoidance, the formation was able to successfully navigate around the obstacle and reach the goal position in front of the camera. The results for the formation controller with the X80 robots are shown in figure 9.28

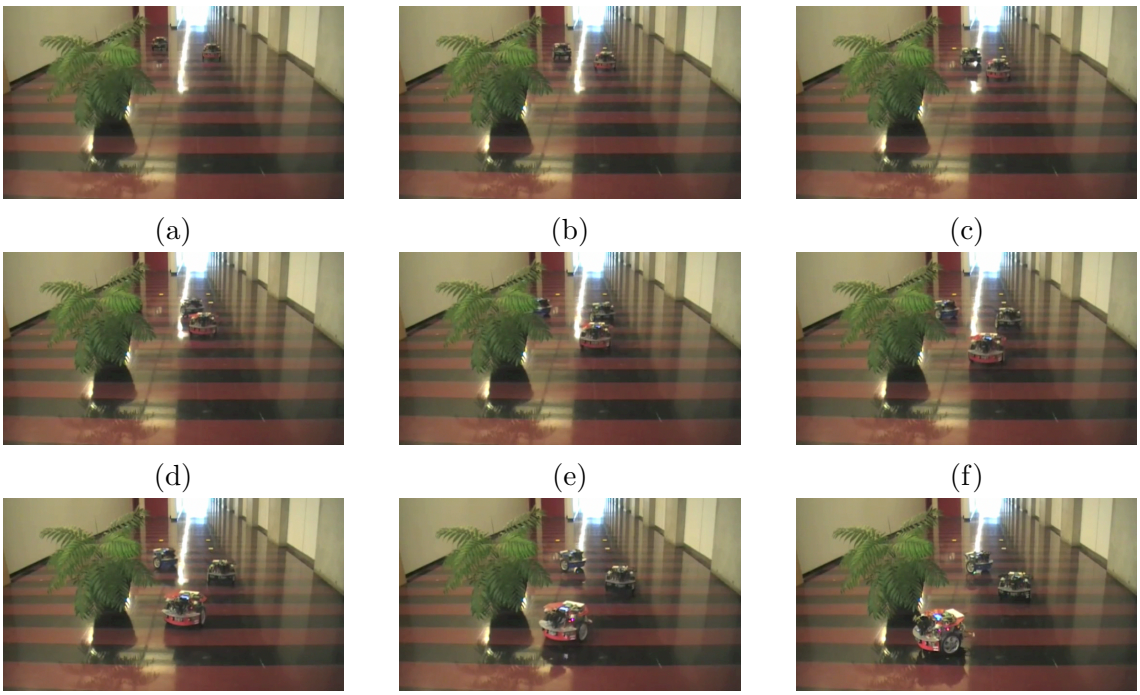


Figure 9.28: Platoon with an obstacle

9.2.4.2 V-Formation through a Narrow Passage

The next experiment tested the V-formation through a narrow passage. In this experiment, the passage distance was too small for all of the robots to travel through with their initial V-angles. Thus, as the leader passed through the narrow passage

it calculated the passage distance based on its sonar and infrared readings, and adjusted the V-angle to accommodate the group. Figure 9.29 shows the reconfiguring V-formation through a narrow passage.

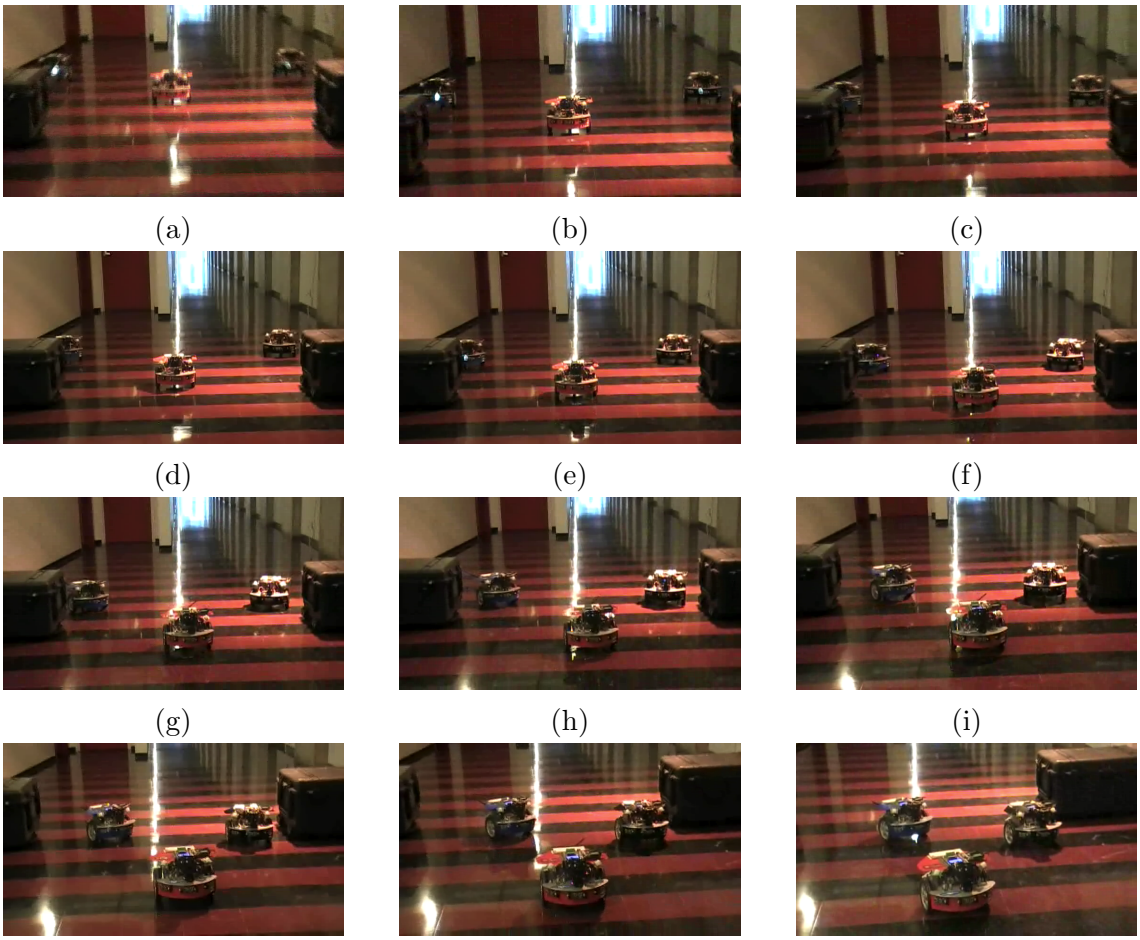
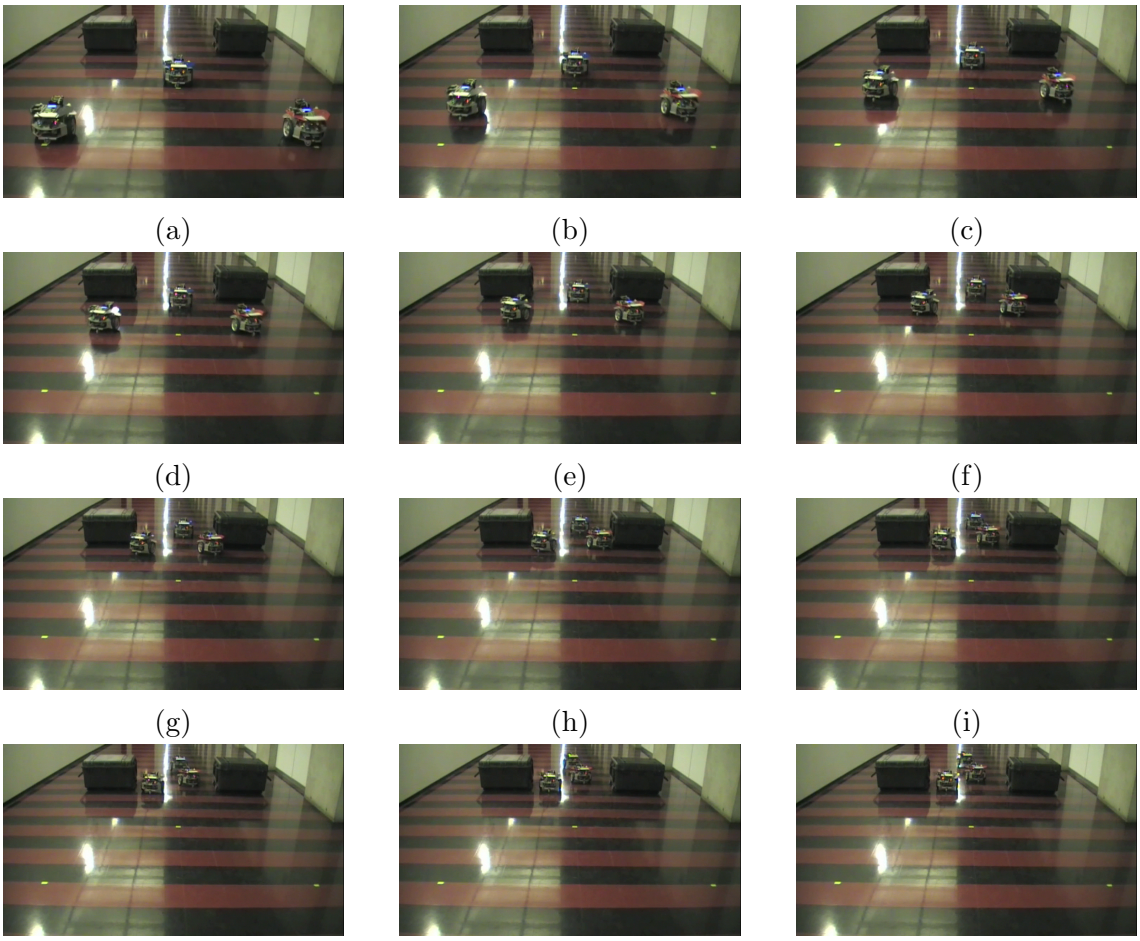


Figure 9.29: V-formation through a narrow passage

9.2.4.3 V-Formation through a Very Narrow Passage

Finally, the last experiment tested the V-formation through a very narrow passage. In this case, the passage distance is large enough for one robot to pass through at a time. Figure 9.30 shows how the angle of the V-formation decreases based on the calculated passage distance, and how the robots created platoon formation to navigate the group through the complex obstacle.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 9.30: V-formation through a very narrow passage

Chapter 10

Results for Self-Organizing Formations

In this final results chapter, we expanded the Velocity Potential (VP) method to a multi-robot scenario, to test its feasibility as a formation controller. Simulations were first conducted in MATLAB, however the multi-robot VP method was too computationally heavy for the MATLAB development environment, and the simulation experienced significant delays. For all self-organizing formation testing using the VP method, the Player/Stage simulation environment was used, since it supports multi-threading and is designed for multi-robot applications.

The following chapter describes the simulation and experimental work done to create self-organizing multi-robot formations using the Velocity Potential method. Section 10.1 explains the work done to develop a C++ file that automatically creates configuration and world files for large groups of robots. Section 10.2 explains the algorithm for the C++ self-organizing controller code. Finally, section 10.3 and section 10.4 describe the simulation and experimental work with the X80 robots for the self-organizing case.

10.1 Creating Configuration and World Files for Large Groups of Robots

The goal of our simulation work with the self-organizing formation was to test the Velocity Potential controller on a large group of homogeneous robots to see if they could navigate to a goal and avoids inter-robot collisions along its way. It was found that writing configuration and world files by hand for large teams of robots was a time consuming and error prone task. In order to make the self-organizing simulations scalable from a small group of robots to a very large group of robots, it was beneficial to create a C++ file to automatically generate the configuration and world files for any number of robots inputted by the user.

Appendix N contains the C++ code to automatically generate the configuration and world files. In the code, the user can update any of the parameters for the simulation, and the change is then applied globally to all of the configuration and world files. Creating this program improved the efficiency of the simulation development phase, and allowed us to test the VP self-organizing formation on any size of group.

10.2 Simulation Algorithm

The self-organizing simulation was programmed using object-oriented C++ programming concepts, and was designed to scale to any number of robots set by the user. The full code can be found in Appendix O and consists of seven files that all connect to the SelfOrganizing.cpp main function. The details of the main function are explained in the figure 10.1, and just like the simulations from Chapter 8 and Chapter 9, the self-organizing simulation initializes variables and acquires inputs form the user. It then calls on the Player Proxies to connect the robots to the PC over a wireless network. The simulation then calculates and sets the velocities of all

of the team members in a loop until one of the robots reaches the goal position. Figure 10.2a and figure 10.2b show how the controller sets the proxies for any number of robot.

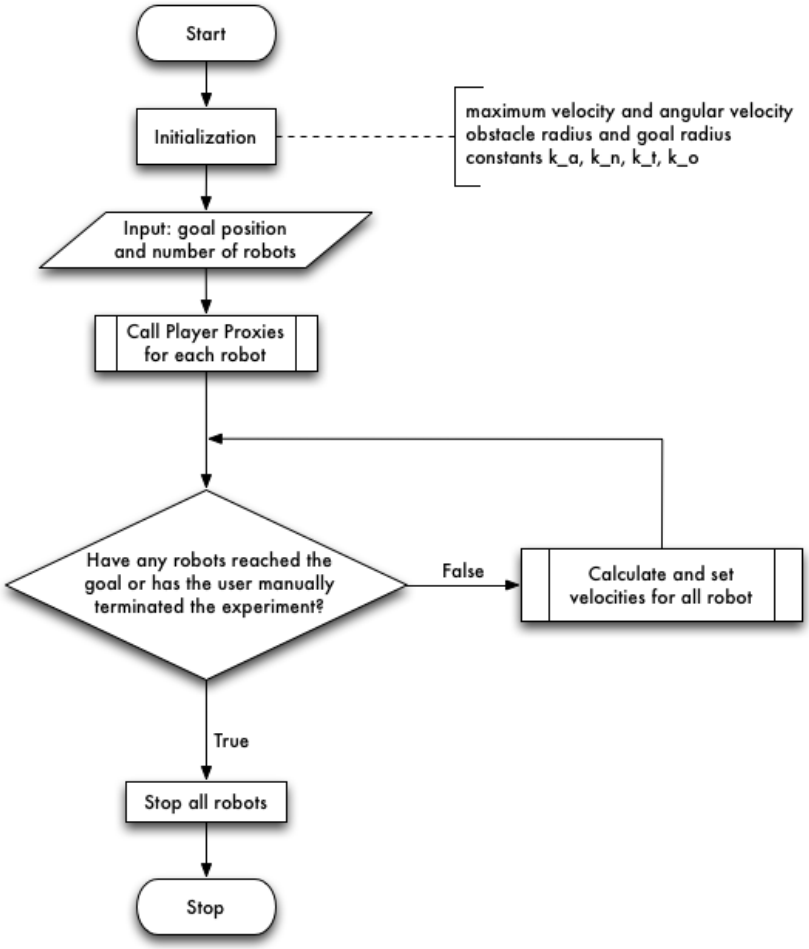


Figure 10.1: Main function

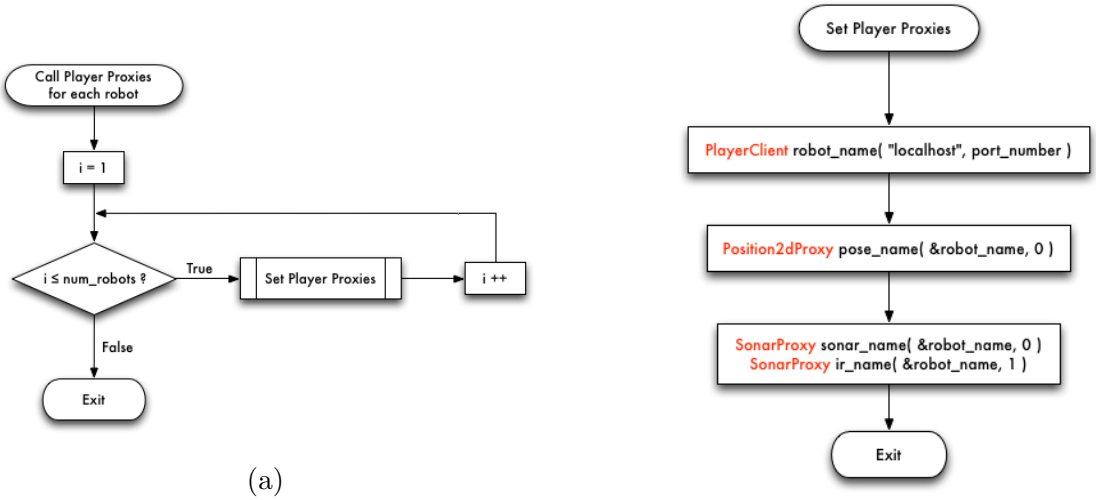


Figure 10.2: Set player proxies

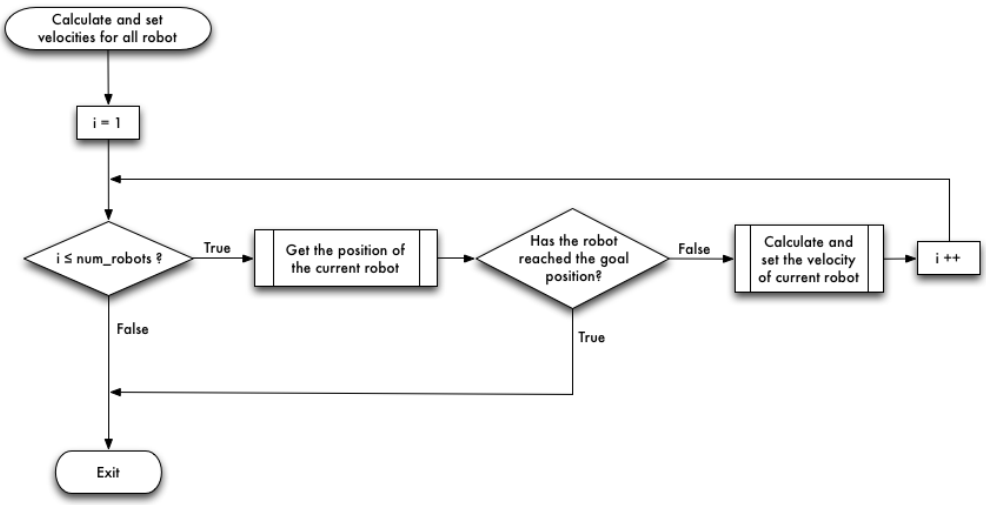


Figure 10.3: Calculate velocity

Figure 10.3 shows the steps required to calculate the linear and angular velocity for each robot. First the current pose of the robot is determined based on the methodolgy of figure 10.4a. If the robot has not reached the goal position, the velocity of the current robot is calculated based on the methodology of figure 10.4b, that checks for obstacles (figure 10.5a) and calculates linear and angular velocity (figure 10.5b) based on the equations from the Velocity Potential method.

The velocity calculations of figure 10.3 are looped until each robot calculation is carried out. Velocity commands are set using the SetSpeed() method, and when the a robot has reached the goal position all of the vehicles are stopped and the simulation ends.

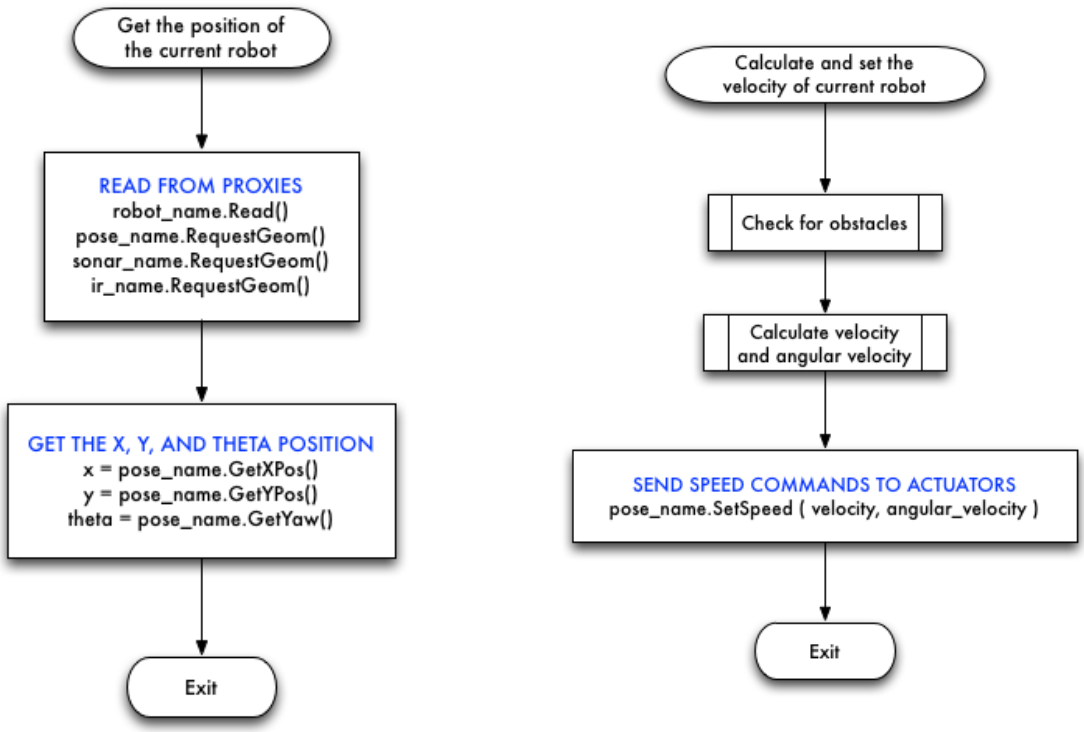


Figure 10.4: (left) get current pose, (right) set velocity

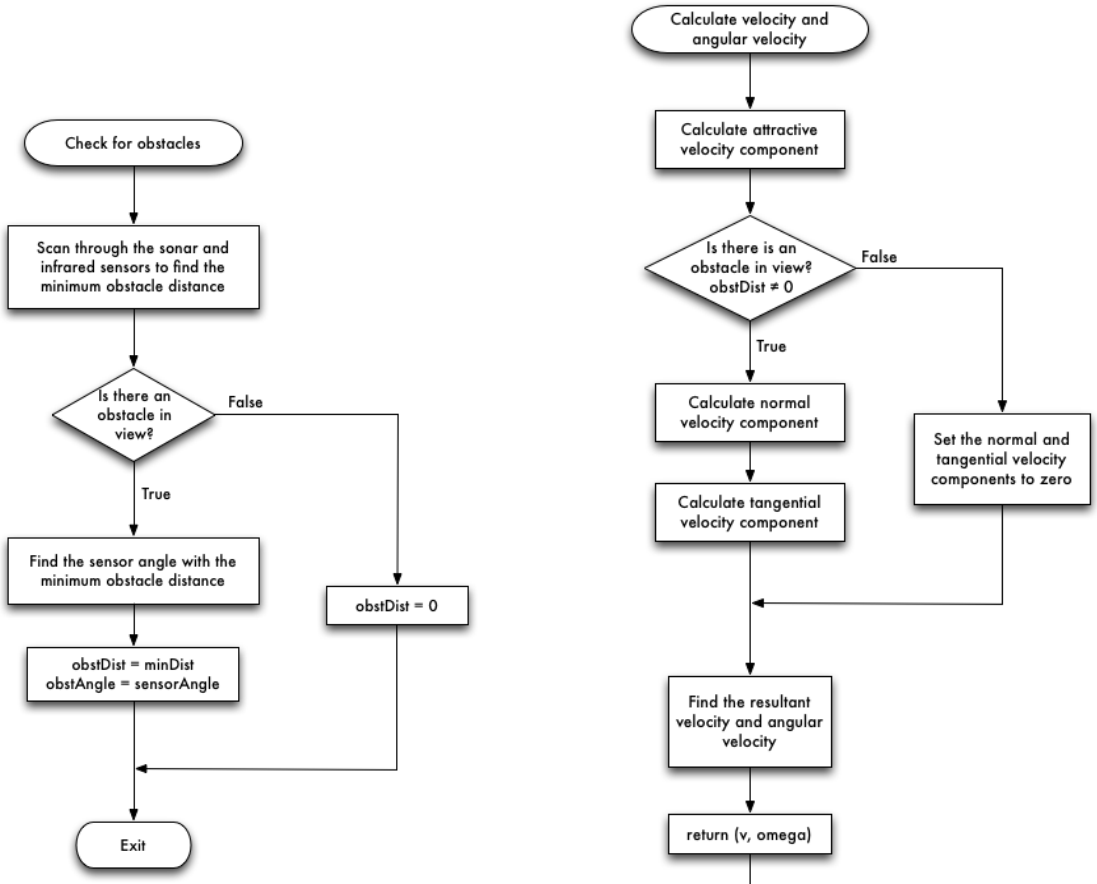


Figure 10.5: (left) check for obstacles, (right) calculate velocity and angular velocity

10.3 Simulation Results

10.3.1 Three Robots

The first simulation tested the Velocity Potential controller on a group of three robots. All robots were able to successfully navigate towards the goal without any collisions. By incorporating the velocity potential method, each robot created a safety zone around its body that kept the other robots safely away. Figure 10.6 shows snapshots of the self-organizing simulation with three robots.

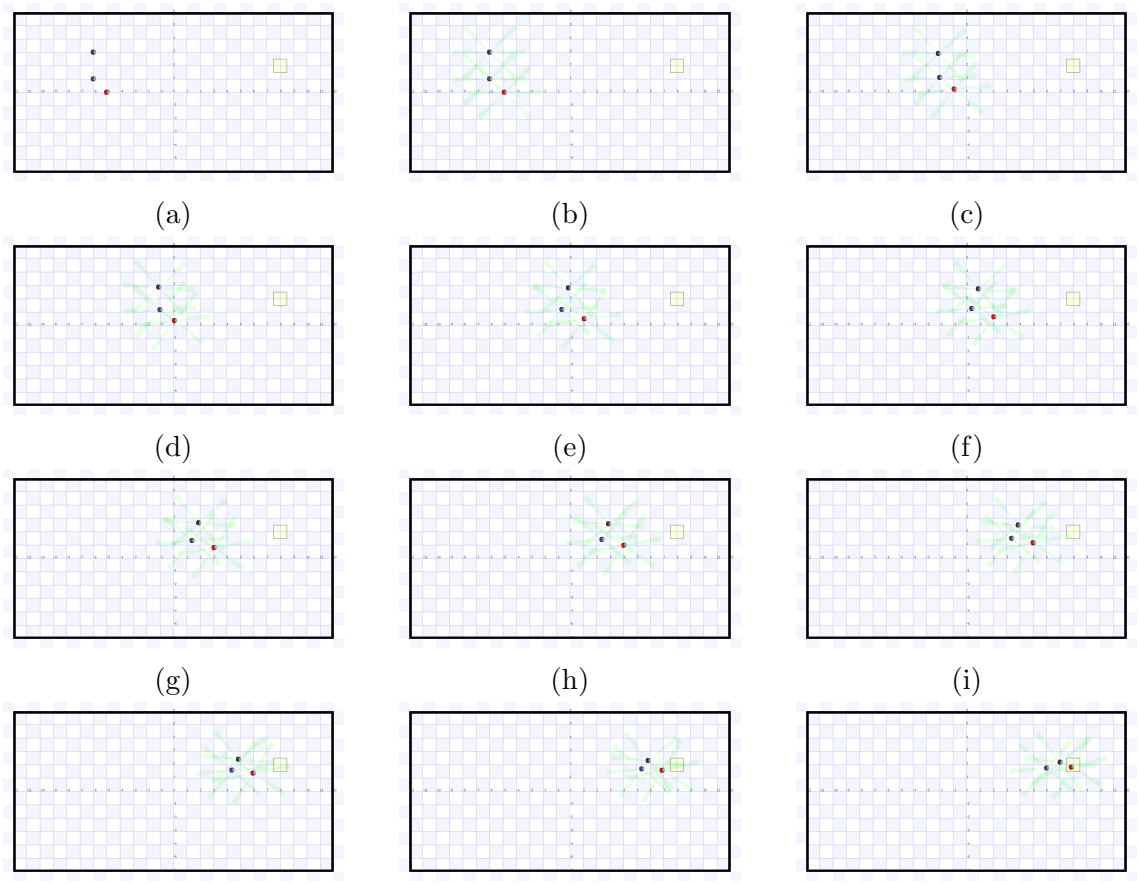
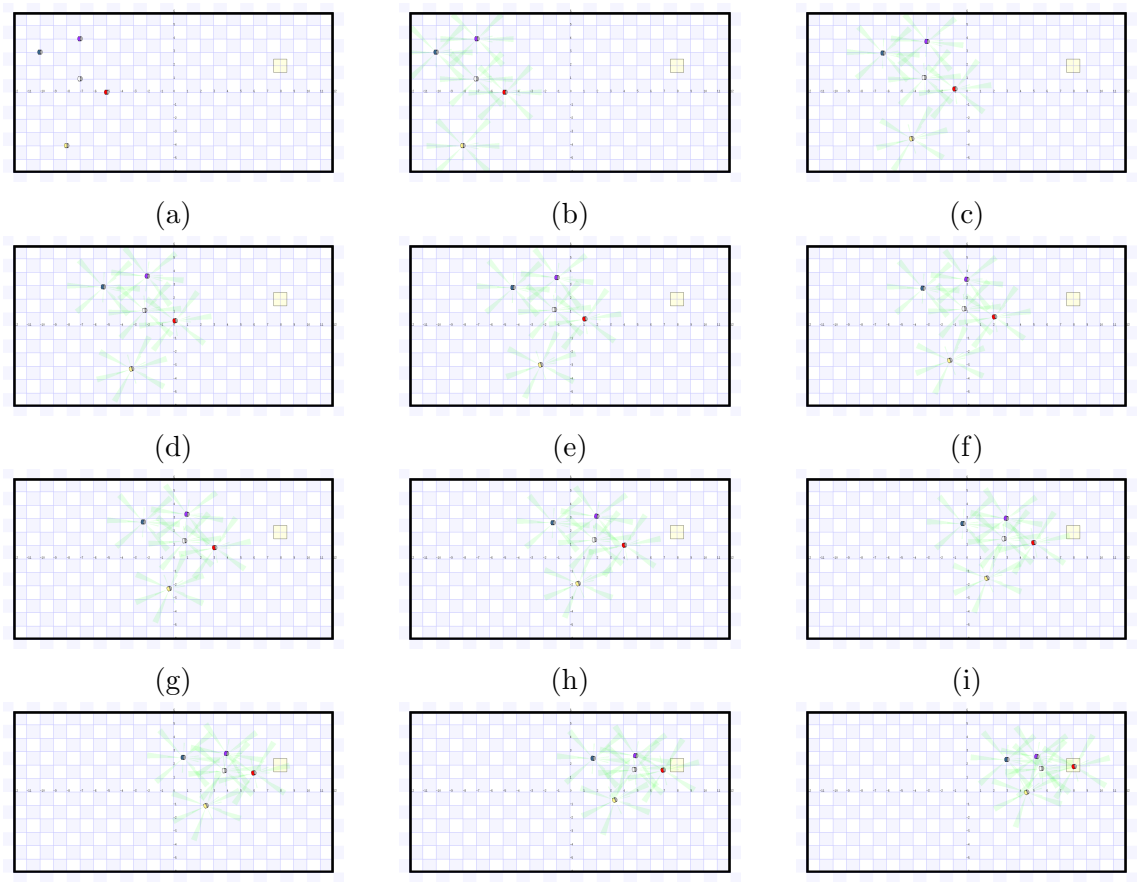


Figure 10.6: 3 Robots

10.3.2 Five Robots

In the next simulation we increased the number of robots in the terrain to five robots. All of the robots are attracted to the same goal position, and the simulation ended when one of the robots reached the goal position. Figure 10.7 shows snapshots of the simulation with five robots.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 10.7: 5 Robots

10.3.3 Eight Robots

The third simulation expanded the self-organizing case to eight robots. With eight robots, the group is more crowded than the three or five robot simulation, but the Velocity Potential method holds up, and the group is able to successfully navigate towards the goal position. Figure 10.9 shows snapshots of the simulation with eight robots.

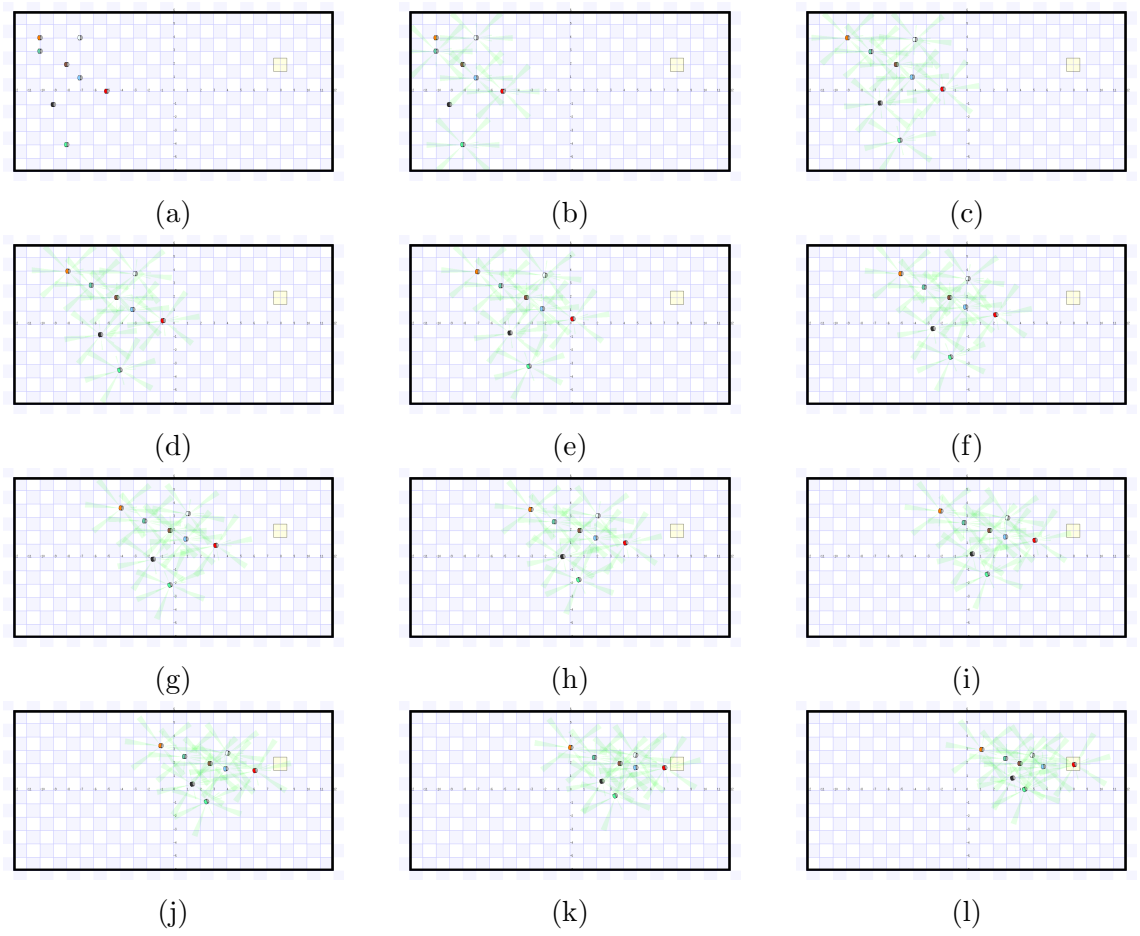


Figure 10.8: 8 Robots

10.3.4 Twelve Robots

In the fourth simulation, we expanded the group to twelve robots. With twelve robots the self-organizing VP group looks quite a bit like a swarm formation seen in the literature. As the robots travel towards the goal formation elongates since all of the robots are trying to reach the same goal position. Figure 10.9 shows snapshots of the self-organizing simulation with twelve robots.

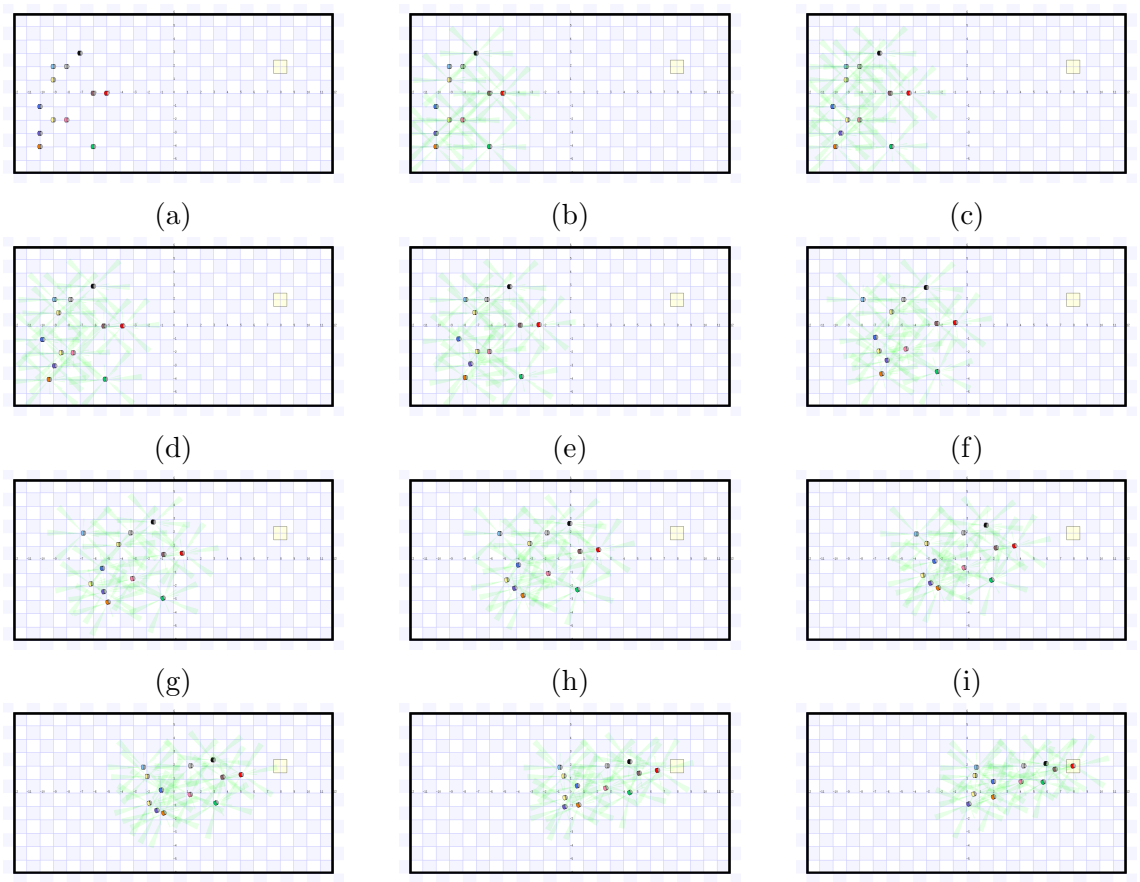


Figure 10.9: 12 Robots

10.3.5 Twenty Robots

Finally, the last simulation looks at a large team consisting of 20 mobile robots. As in the twelve robot case, the formation elongated as the robots navigated towards

the goal position. Since the twenty robot simulation had many team members, the formation created a triangle shape towards the rear of the formation, since the back team members needed to slow down in order to avoid colliding with the robots at the head of the formation. Figure 10.10 shows snapshots of the self-organizing formation for a team of twenty mobile robots.

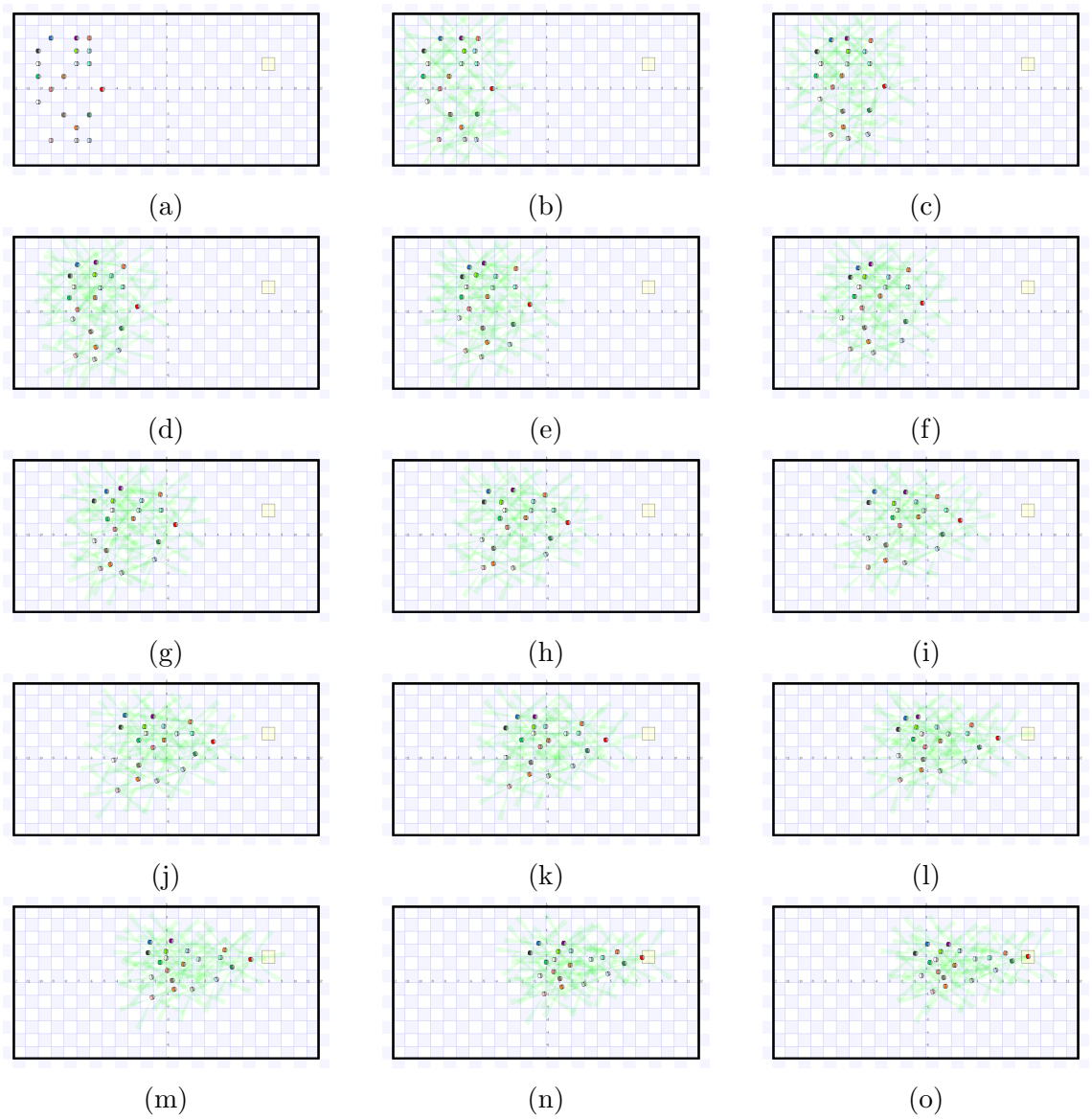


Figure 10.10: 20 Robots

10.4 Experimental Results

In the experiments with the X80, we wanted to test the Velocity Potential self-organizing formation on three robots. In the first experiment, the three robots we placed in a terrain without obstacles. Using the self-organizing formation approach, the robots navigated towards the goal position in front of the camera, while keeping a safe distance from each other. Figure 10.11 shows snapshots of the experiments with no obstacles in the terrain.

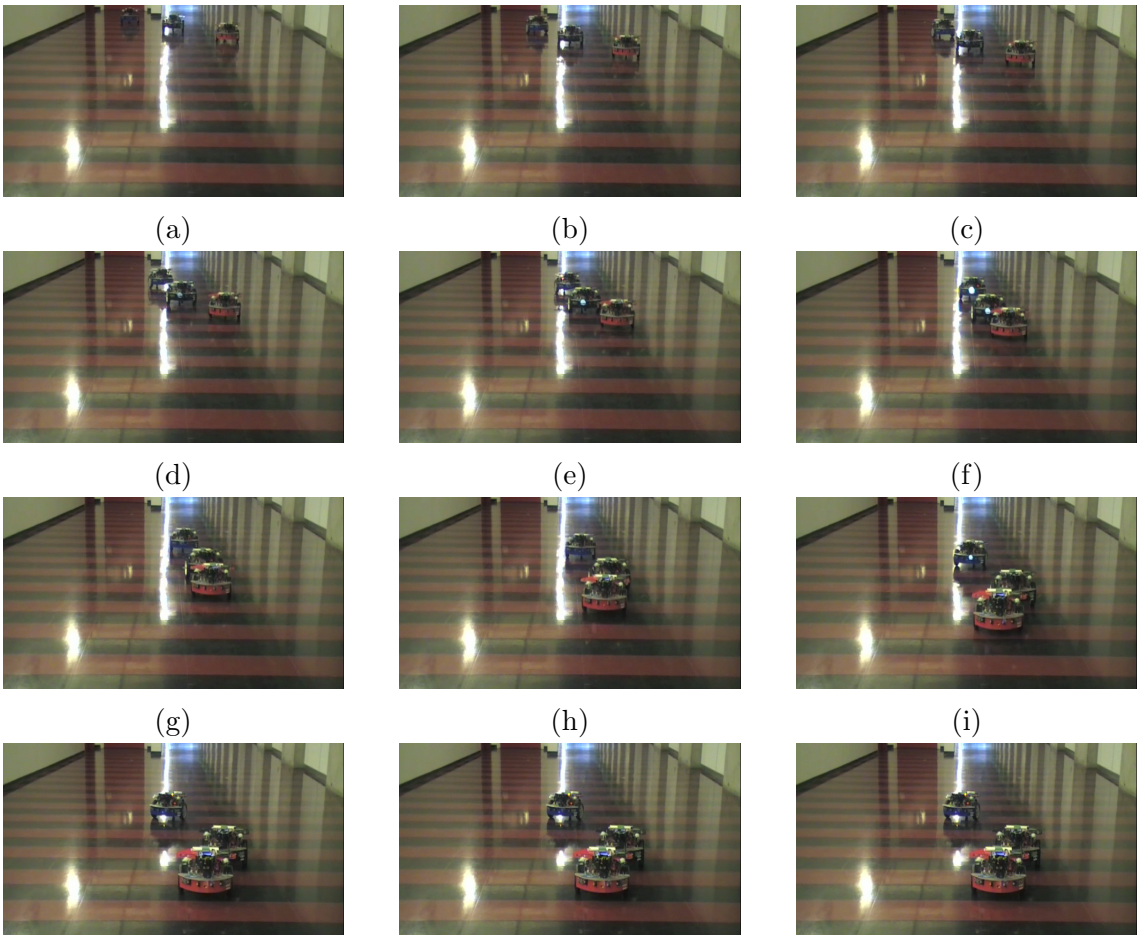


Figure 10.11: Self-organizing no obstacle

Finally, the last experiment with the X80 mobile robots added an obstacle in the terrain. The obstacle was a potted plant, and it was placed in the path between the initial pose of one of the robots and the goal position. Using the self-organizing velocity potential approach, the robots were able to avoid a collision with the obstacle in the terrain. Moreover, the other two robots adjusted their movement, and all three robots navigated around the obstacle as a unit, while keeping a safe distance from each other. Figure 10.12 shows the self-organizing formation around an obstacle in the environment.

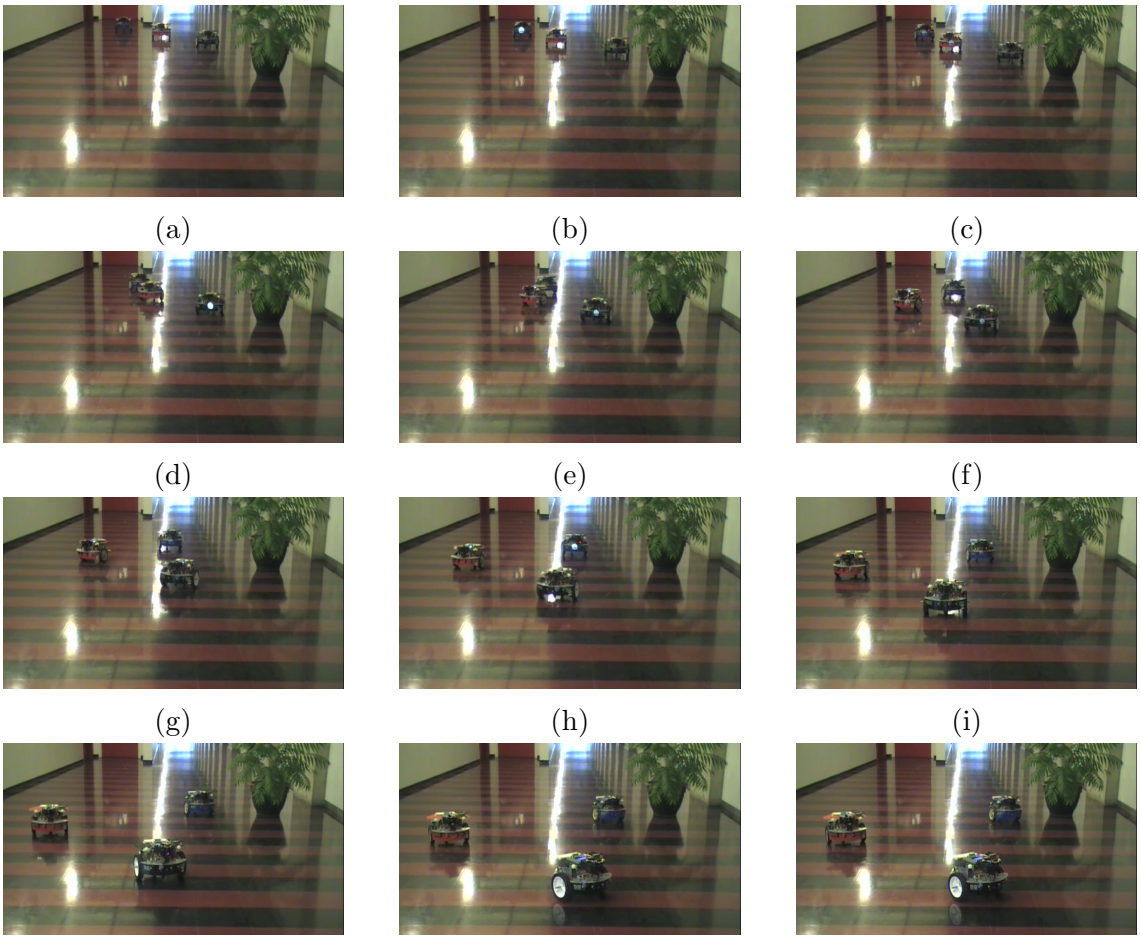


Figure 10.12: Self-organizing with an obstacle

Chapter 11

Conclusion

The purpose of our research work was to develop a navigation and formation control strategy to move a team of robots through a two-dimensional terrain to a goal position without colliding with obstacles in the environment. In order to solve the motion planning problem, we developed a Velocity Potential navigation controller. The navigation controller was inspired by the velocity potential function in fluid mechanics, and modeled the flow to the goal as a uniform flow field, and the flow around the obstacle as a spiral vortex. Simulations and experiments using the velocity potential controller were carried out in MATLAB and Player/Stage, and the experiments tested the algorithm in environments with simple and complex obstacles. The velocity potential navigation controller successfully navigated a single robot around all obstacle geometries.

After developing a navigation strategy for the team of robots, formation controllers were developed to coordinate the members of the group. A leader-follower formation controller was chosen, where the leader would navigate through the terrain using the navigation controller, while the followers would orient themselves at predefined positions behind the leader. Two different leader-follower geometries were created: (1) platoon formation, and (2) V-formation. In our experiments we

first tested how the robot formations would develop in a terrain with no obstacles. We then placed the formations in a terrain with obstacles, and applied a reconfiguration strategy, so that the platoon or the V-formation could adapt its formation shape to the obstacles in the terrain. The reconfiguration property was essential in the design of the leader-follower controller, so that the group could navigate around difficult obstacles, without any collisions with obstacles in the terrain. Simulations for the reconfiguring leader-follower formations were carried out in MATLAB, and the experiments with Player/Stage tested the formation on 3 X80 robot units.

Finally, in our last simulations and experiments we expanded the velocity potential method, and allowed the robot movement to be defined in a distributed decentralized manner by the velocity potential controller. The multi-robot velocity potential controller created a self-organizing formation, where robots on the team would orient themselves in order to reach the goal, but also not collide with each other along the path. Simulations for the multi-robot case were carried out on 3, 5, 8, 12, and 20 robots, and experiments with the X80 robot were carried out on 3 robot units.

11.1 Research Contributions

The proposed methods improved upon the artificial potential field navigation controller for self-organizing systems. The artificial potential field methods struggled from the local minimum problem, and had difficulties navigating around complex obstacles, such as the concave obstacle and the narrow passage obstacle. Used as a navigation controller, the velocity potential flow controller was able to easily navigate a robot around these complex obstacles. Furthermore, when the velocity potential flow controller was scaled to large multi-robot systems, the team was able to move through the environment like a swarm formation. Our research contributed to the

domain of swarm systems, and provided a new mechanism of self-organization to a group of distributed decentralized robots. Finally, the reconfiguring leader-follower controller, along with the velocity potential navigation controller, showed that a team of mobile robots could travel through the terrain in a desired geometric formation, and its formation shape could morph and adapt to changes in the environment, so that the formation could handle complex obstacles in dynamic environments.

11.2 Future Work

The next phase in the project is to increase the level of coordination and inter-robot collision avoidance. In our geometric formation control experiments with the leader follower method, the follower robots were assigned positions at an adequate distance and angle from the leader and each other in order to avoid inter-robot collisions. Although this method did provide good results for group coordination, it is not a global solution to the overall formation control problem. The next step in this research is to provide a mechanism for inter-robot collision avoidance for the follower robots while they travel in the leader-follower formation.

Furthermore, there are still work that needs to be addressed in the self-organizing formation case using the velocity potential method. The self-organizing formation was never tested on complex obstacles like the concave obstacle and narrow passage. The added confined space requires modification to the parameters of the self-organizing controller. The long-term goal for the velocity potential method and leader-follower formation controller is to develop a global solution to the navigation and formation control problem, where the overall movement of the team is adaptable to any environment that the team will face.

Bibliography

[1] Tucker Balch and Ronald C Arkin. "Behavior-based formation control for multirobot teams". In: *Robotics and Automation, IEEE Transactions on* 14.6 (1998), pp. 926–939.

[2] Tucker Balch and Maria Hybinette. "Behavior-based coordination of large-scale robot formations". In: *MultiAgent Systems, 2000. Proceedings. Fourth International Conference on*. IEEE. 2000, pp. 363–364.

[3] Laura Barnes, MaryAnne Fields, and Kimon Valavanis. "Unmanned ground vehicle swarm formation control using potential fields". In: *Control & Automation, 2007. MED'07. Mediterranean Conference on*. IEEE. 2007, pp. 1–8.

[4] George A Bekey. *Autonomous robots: from biological inspiration to implementation and control*. The MIT Press, 2005.

[5] Karsten Berns and Ewald von Puttkamer. *Autonomous Land Vehicles*. Springer, 2009.

[6] Roger W Brockett et al. "Asymptotic stability and feedback stabilization". In: (1983).

[7] Thijs HA van den Broek, Nathan van de Wouw, and Henk Nijmeijer. "Formation control of unicycle mobile robots: a virtual structure approach". In: *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control*

Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on. IEEE. 2009, pp. 8328–8333.

[8] Yang Quan Chen and Zhongmin Wang. "Formation control: a review and a new consideration". In: *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*. IEEE. 2005, pp. 3181–3186.

[9] Howie M Choset. *Principles of robot motion: theory, algorithms, and implementations*. MIT press, 2005.

[10] Luca Consolini et al. "On the control of a leader-follower formation of nonholonomic mobile robots". In: *Decision and Control, 2006 45th IEEE Conference on*. IEEE. 2006, pp. 5992–5997.

[11] Jaydev P Desai. "A graph theoretic approach for modeling mobile robot team formations". In: *Journal of Robotic Systems* 19.11 (2002), pp. 511–525.

[12] Donald D Dudenhoeffer and Michael P Jones. "A formation behavior for large-scale micro-robot force deployment". In: *Simulation Conference, 2000. Proceedings. Winter*. Vol. 1. IEEE. 2000, pp. 972–982.

[13] Barbara Dunin-Keplicz and Rineke Verbrugge. *Teamwork in multi-agent systems: A formal approach*. Vol. 21. John Wiley & Sons, 2011.

[14] Farbod Fahimi. *Autonomous robots: modeling, path planning, and control*. Springer New York, 2009.

[15] Farbod Fahimi, Chandrasekhar Nataraj, and Hashem Ashrafiuon. "Real-time obstacle avoidance for multiple mobile robots". In: *Robotica* 27.2 (2009), pp. 189–198.

[16] J Alexander Fax and Richard M Murray. "Information flow and cooperative control of vehicle formations". In: *Automatic Control, IEEE Transactions on* 49.9 (2004), pp. 1465–1476.

[17] Anthony Finn and Steve Scheding. *Developments and challenges for autonomous unmanned vehicles: a compendium*. Vol. 3. Springer, 2010.

[18] Robert W Fox, Alan T McDonald, and Philip J Pritchard. *Introduction to fluid mechanics*. Vol. 2. John Wiley & Sons New York, 2011.

[19] Veysel Gazi. "Swarm aggregations using artificial potentials and sliding-mode control". In: *Robotics, IEEE Transactions on* 21.6 (2005), pp. 1208–1214.

[20] Shuzhi Sam Ge. *Autonomous mobile robots: sensing, control, decision making and applications*. CRC press, 2010.

[21] Shuzhi Sam Ge and C-H Fua. "Queues and artificial potential trenches for multirobot formations". In: *Robotics, IEEE Transactions on* 21.4 (2005), pp. 646–656.

[22] David Gingras et al. "Path planning based on fluid mechanics for mobile robots using unstructured terrain models". In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE. 2010, pp. 1978–1984.

[23] William S Janna. *Introduction to fluid mechanics*. CRC Press, 2010.

[24] Qiuling Jia and Guangwen Li. "Formation control and obstacle avoidance algorithm of multiple autonomous underwater vehicles (AUVs) based on potential function and behavior rules". In: *Automation and Logistics, 2007 IEEE International Conference on*. IEEE. 2007, pp. 569–573.

[25] Gal A Kaminka and Inna Frenkel. "Flexible teamwork in behavior-based robots". In: *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*. Vol. 20. 1. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999. 2005, p. 108.

[26] Shih-Chung Kang et al. *Robot development using microsoft robotics developer studio*. Taylor & Francis US, 2011.

[27] Didier Keymeulen and Jo Decuyper. "The fluid dynamics applied to mobile robot motion: the stream field method". In: *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*. IEEE. 1994, pp. 378–385.

[28] Maher Khatib. *Sensor-based motion control for mobile robots*. Citeseer, 1996.

[29] Oussama Khatib. "Real-time obstacle avoidance for manipulators and mobile robots". In: *The international journal of robotics research* 5.1 (1986), pp. 90–98.

[30] Wojciech Kowalczyk and Krzysztof Kozlowski. "Artificial potential based control for a large scale formation of mobile robots". In: *Robot Motion and Control, 2004. RoMoCo'04. Proceedings of the Fourth International Workshop on*. IEEE. 2004, pp. 285–291.

[31] Gerardo Lafferriere et al. "Decentralized control of vehicle formations". In: *Systems & Control Letters* 54.9 (2005), pp. 899–910.

[32] Emmett Lalish, Kristi A Morgansen, and Takashi Tsukamaki. "Formation tracking control using virtual structures and deconfliction". In: *Decision and Control, 2006 45th IEEE Conference on*. IEEE. 2006, pp. 5699–5705.

[33] Bin Lei, Wenfeng Li, and Fan Zhang. "Stable flocking algorithm for multi-robot systems formation control". In: *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*. IEEE. 2008, pp. 1544–1549.

[34] Naomi Ehrich Leonard and Edward Fiorelli. "Virtual leaders, artificial potentials and coordinated control of groups". In: *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*. Vol. 3. IEEE. 2001, pp. 2968–2973.

[35] M Anthony Lewis and Kar-Han Tan. "High precision formation control of mobile robots using virtual structures". In: *Autonomous Robots* 4.4 (1997), pp. 387–403.

[36] Xiaohai Li and Jizhong Xiao. "Robot formation control in leader-follower motion using direct Lyapunov method". In: *International Journal of Control System* 10 (2005), pp. 244–50.

[37] ZX Li and TD Bui. "Robot path planning using fluid model". In: *Journal of Intelligent and Robotic Systems* 21.1 (1998), pp. 29–50.

[38] Bailong Liu, Rubo Zhang, and Changting Shi. "Formation control of multiple behavior-based robots". In: *Computational Intelligence and Security, 2006 International Conference on*. Vol. 1. IEEE. 2006, pp. 544–547.

[39] Jiming Liu and Jianbing Wu. *Multiagent Robotic Systems*. CRC press, 2010.

[40] Maja J Matarić. "Learning in behavior-based multi-robot systems: Policies, models, and other agents". In: *Cognitive Systems Research* 2.1 (2001), pp. 81–93.

[41] Satoshi Murata and Haruhisa Kurokawa. *Self-organizing robots*. Springer, 2012.

[42] Reza Olfati-Saber and Richard M Murray. "Distributed cooperative control of multiple vehicle formations using structural potential functions". In: *IFAC World Congress*. 2002, pp. 346–352.

[43] Jennifer Owen. *How to Use Player/Stage*. 2009.

[44] Ronald L Panton. *Incompressible flow*. John Wiley & Sons, 2006.

[45] Zhihua Qu. *Cooperative control of dynamical systems: applications to autonomous vehicles*. Springer, 2009.

[46] Wei Ren and Randal W Beard. "Formation feedback control for multiple spacecraft via virtual structures". In: *Control Theory and Applications, IEE Proceedings-*. Vol. 151. 3. IET. 2004, pp. 357–368.

[47] R Olfati Saber, William B Dunbar, and Richard M Murray. "Cooperative control of multi-vehicle systems using cost graphs and optimization". In: *American Control Conference, 2003. Proceedings of the 2003*. Vol. 3. IEEE. 2003, pp. 2217–2222.

[48] Lorenzo Sciavicco. *Robotics: modelling, planning and control*. Springer, 2009.

[49] William Rees Sears and Demetri P Telionis. *Introduction to theoretical aerodynamics and hydrodynamics*. American Institute of Aeronautics and Astronautics, 2011.

[50] Jinyan Shao, Long Wang, and Guangming Xie. "Flexible formation control for obstacle avoidance based on numerical flow field". In: *Decision and Control, 2006 45th IEEE Conference on*. IEEE. 2006, pp. 5986–5991.

[51] Jinyan Shao et al. "Leader-following formation control of multiple mobile robots". In: *Intelligent Control, 2005. Proceedings of the 2005 IEEE International Symposium on, Mediterrean Conference on Control and Automation*. IEEE. 2005, pp. 808–813.

[52] Edward J Shaughnessy, Ira M Katz, and James P Schaffer. *Introduction to fluid mechanics*. Oxford University Press New York, 2005.

[53] Roland Siegwart and Illah Reza Nourbakhsh. *Introduction to Autonomous Mobile Robotos*. Second Edition. The MIT press, 2011.

[54] Hebertt Sira-Ramírez and Rafael Castro-Linares. "Trajectory tracking for nonholonomic cars: A linear approach to controlled leader-follower formation". In: *Decision and Control (CDC), 2010 49th IEEE Conference on*. IEEE. 2010, pp. 546–551.

[55] Ashley Stroupe et al. "Behavior-based multi-robot collaboration for autonomous construction tasks". In: *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*. IEEE. 2005, pp. 1495–1500.

[56] Housheng Su, Xiaofan Wang, and Zongli Lin. "Flocking of multi-agents with a virtual leader". In: *Automatic Control, IEEE Transactions on* 54.2 (2009), pp. 293–307.

[57] Herbert G Tanner, Ali Jadbabaie, and George J Pappas. "Stable flocking of mobile agents part i: dynamic topology". In: *Decision and Control, 2003. Proceedings. 42nd IEEE Conference on*. Vol. 2. IEEE. 2003, pp. 2016–2021.

[58] Herbert G Tanner, George J Pappas, and Vijay Kumar. "Leader-to-formation stability". In: *Robotics and Automation, IEEE Transactions on* 20.3 (2004), pp. 443–455.

[59] Antonios Tsourdos, Brian White, and Madhavan Shanmugavel. *Cooperative path planning of unmanned aerial vehicles*. Vol. 32. Wiley, 2010.

[60] Stephen Waydo and Richard M Murray. "Vehicle motion planning using stream functions". In: *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*. Vol. 2. IEEE. 2003, pp. 2484–2491.

[61] Barry Brian Werger. "Cooperation without deliberation: A minimal behavior-based approach to multi-robot teams". In: *Artificial Intelligence* 110.2 (1999), pp. 293–320.

[62] Amanda Whitbrook. *Programming mobile robots with Aria and Player: a guide to C++ object-oriented control*. Springer, 2010.

[63] *WiRobot X80 User Manual*. Dr. Robot Inc.

[64] Guohua Ye, Hua O Wang, and Kazuo Tanaka. "Coordinated motion control of swarms with dynamic connectivity in potential flows". In: *Proceedings of the 16th International Federation of Automatic Control World Congress*. 2005.

[65] Chika Yoshioka and Toru Namerikawa. "Formation control of nonholonomic multi-vehicle systems based on virtual structure". In: *Proceedings of the 17th IFAC World Congress*. Korea Seoul. 2008, pp. 5149–5154.

[66] Donald F Young et al. *A brief introduction to fluid mechanics*. Wiley. com, 2010.

[67] Jian Yuan and Gong-You Tang. "Formation control for mobile multiple robots based on hierarchical virtual structures". In: *Control and Automation (ICCA), 2010 8th IEEE International Conference on*. IEEE. 2010, pp. 393–398.

[68] Ben Yun et al. "A leader-follower formation flight control scheme for UAV helicopters". In: *Automation and Logistics, 2008. ICAL 2008. IEEE International Conference on*. IEEE. 2008, pp. 39–44.

[69] Min Zhang et al. "Dynamic artificial potential field based multi-robot formation control". In: *Instrumentation and Measurement Technology Conference (I2MTC), 2010 IEEE*. IEEE. 2010, pp. 1530–1534.

Appendix A

How to Install Player/Stage

Player and Stage work together to operate mobile robots from a Unix based operating system such as Linux or Mac. Player is a hardware abstraction layer that provides access to the robots sensors and actuators for real robot testing. Stage is a 2-dimensional simulation software. Within stage you set up the geometry of the robot, what sensors are onboard, where the sensors are located on the robot, and any obstacles situated in its environment.

The largest benefit of using the Player and Stage software is the quick and easy transition from simulation to testing. Control of the robot movement is achieved with a C++ file that access values from sensor data and apply actuator control. The controller runs in parallel to Player, and is implemented in the same manner in either simulation or testing mode.

The goal of this appendix is to go through all of the steps needed to run a simulations and tests with multiple robots. The appendix will start with installation instructions for Player, Stage, and Eclipse. After installation is complete, the manual will then outline the steps of creating controllers in C/C++ in the Eclipse IDE, and how to run the code with the robots using the Terminal.

A.1 Operating System

Player and Stage can be installed on any Unix based operating system. The most popular operating system is Ubuntu from Linux. Player is a difficult program to install, expect a full afternoon to make the installation work perfectly. The safest bet is to install Player in Ubuntu since there is a big online community available to give advice if the install is unsuccessful. It is also possible to install Player on the Mac OSX, however a successful installation is more difficult than in Ubuntu.

Create a Linux partition on your computer and install the newest version of Ubuntu from

http://ubuntu.com/download

A.2 Installing Player and Stage

The robot drivers were programmed to work with Player version 2 and Stage version 3. Although there are newer versions of Player and Stage is available, changes must be done to the drivers for the robots to connect with these newer versions of the Player software. Hence for a quick transition it is suggested to install Player 2.x and Stage 3.x

You can get the .tar files for the installation from:

http://playerstage.sourceforge.net

Install Player version 2 first. Follow directions from the online community on how to complete the install. The success of the installation depends on your version of Ubuntu. If the installation is not successful on your first try, it is likely that the installation is missing components. Do an online search for common installation dependencies, install these dependencies, and then try the Player install again.

Repeat the procedure until Player is successfully installed onto your computer. After Player is working, install Stage. If there are problems while installing stage, check for installation dependencies, and try installing Stage again.

A.3 Testing Player and Stage

To test the Player and Stage installation we will use the Terminal window. Go to:

Applications → *Accessories* → *Terminal*

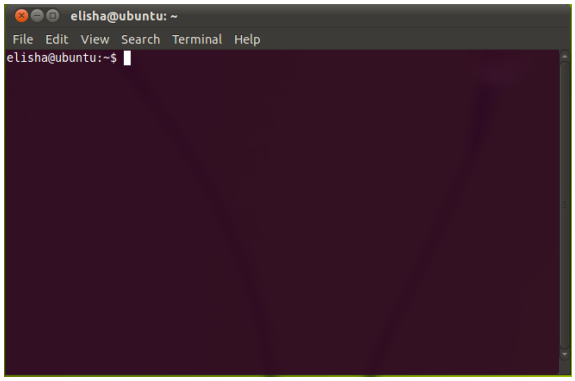


Figure A.1: Ubuntu terminal window

Use the ls command in the terminal to list options. Use the cd command to open folders. Find the location of the file simple.cfg.

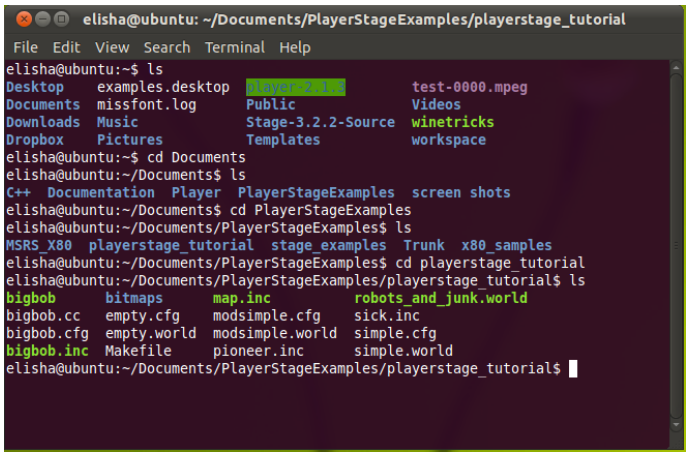


Figure A.2: Accessing Simple.cfg in the terminal

To start the simulation, use command to start the test simulation

player simple.cfg

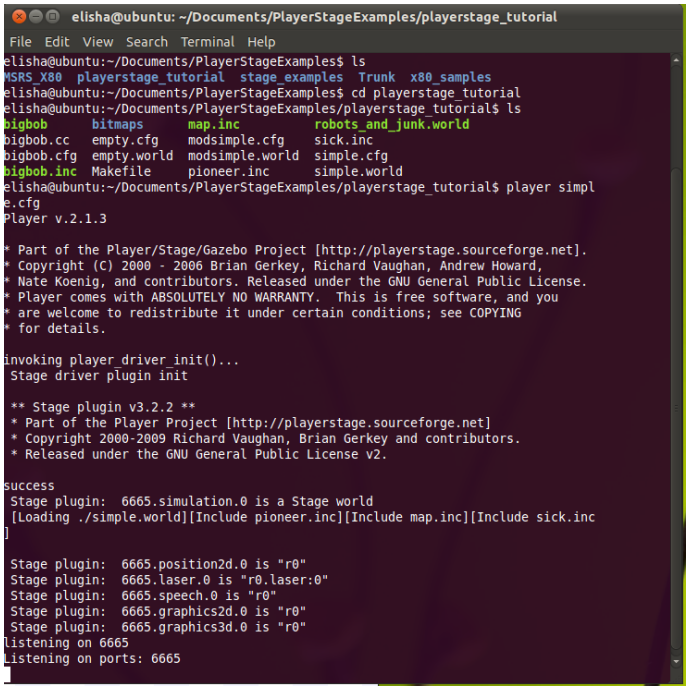


Figure A.3: Starting the simulation

If the simulation simple.cfg is running, then the installation was successful. A picture of the simulation window is displayed below.

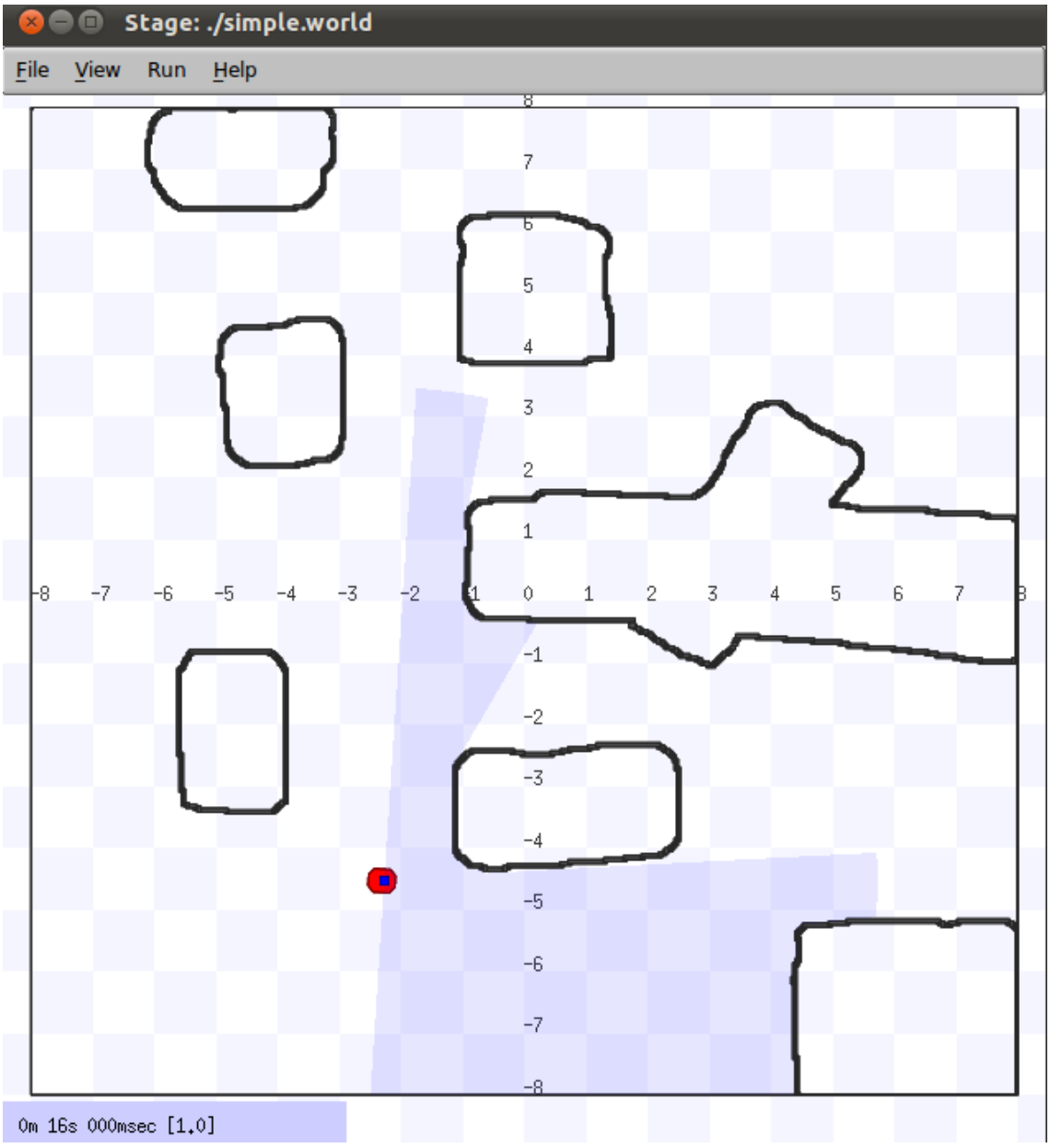


Figure A.4: A running simulation

Appendix B

How to Configure Eclipse to Access Player Libraries

B.1 Installing Eclipse CDT

Eclipse is the most popular development environment in Ubuntu. Programming your robot controllers in Eclipse will save you time and energy. Although there are many other development platforms in Ubuntu, Eclipse has the most support for Player and Stage.

Player controllers can be programmed in C/C++ or Python. Eclipse is generally used by Java programmers, and the general install has no C/C++ functionality. Hence a special version of Eclipse must be installed that has C/C++ functionality, this version is called Eclipse-CDT. To install Eclipse-CDT go to:

$$Applications \rightarrow UbuntuSoftwareCenter$$

In the search textbox write *eclipse-cdt*, double click on *C/C++ Development Tools for Eclipse* and follow installation directions.

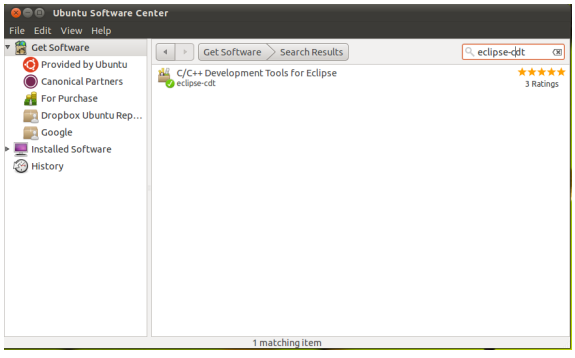


Figure B.1: Installing Eclipse from Ubuntu Software Center

B.2 Linking Eclipse to Player C/C++ Libraries

Every time you start a new project you need to link Eclipse to the Player C/C++ Libraries. The Player libraries include methods and attributes that are important to robot movement such as sensor and motor commands. Eclipse needs to know where this library is located on your hard drive so that it can access it when build the controller. Below is a step-by-step procedure on how to link Eclipse with Player.

- *file* → *newproject*
- call the project *example_setup*
- if you use *Hello World C++ Project*, Eclipse adds some basic start-up elements to the file

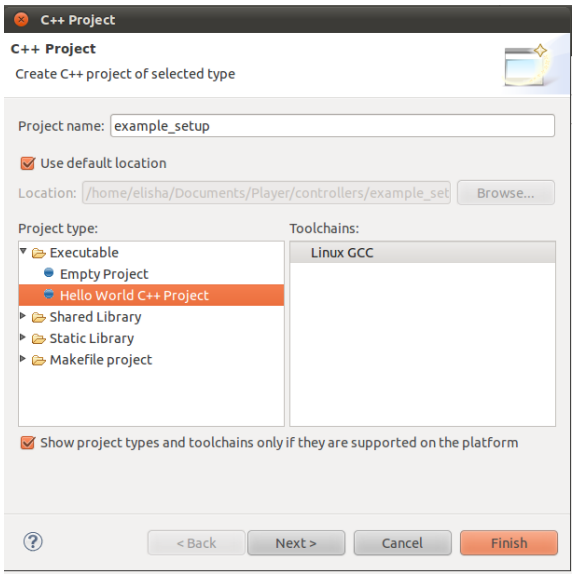


Figure B.2: Creating a new C++ project

- Now in the *Project Explorer* window on the left-hand-side of the screen, the *example_setup* project is displayed, and a cpp file with the same name, example_setup.cpp, is located in the source, src, folder within the project.
- For continuity, all c++ files for the robot control will be in the source folder

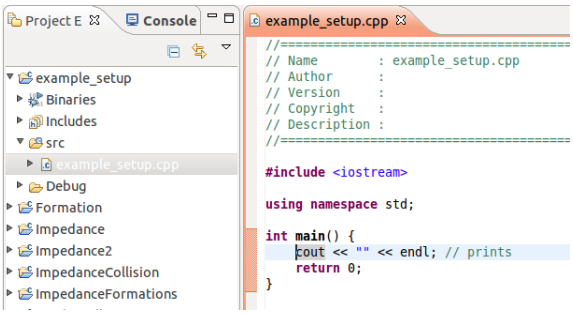


Figure B.3: Initial C++ file layout

- If you are using c++, do # include libplayerc++/player++.h
- If you are using c, do # include libplayerc/playerc.h
- Notice the yellow underline, Eclipse does not know what this library is or where it is stored. You need to add this library manually

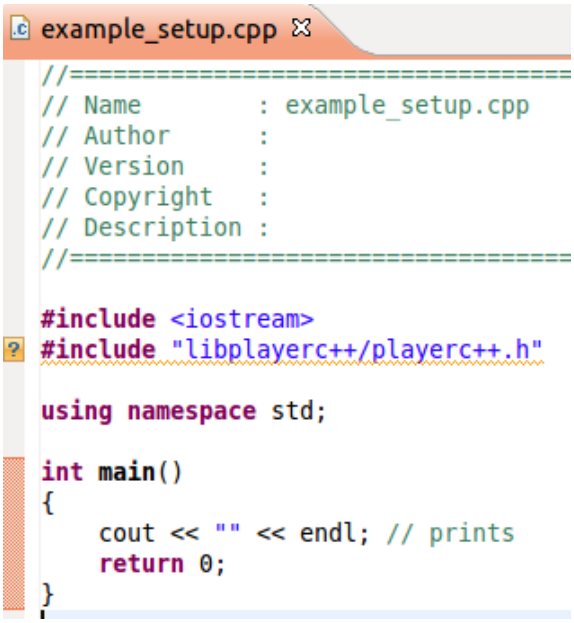


Figure B.4: Include the libplayerc++ library

- Go to *Project* → *Properties*

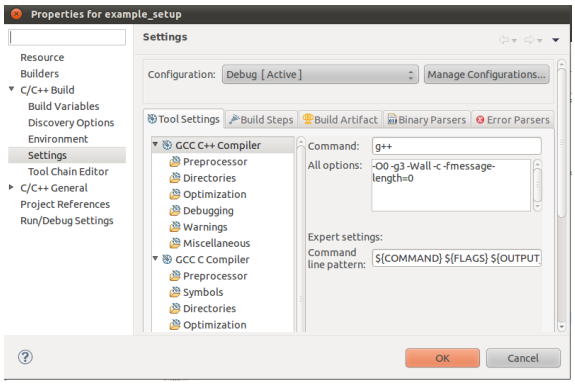


Figure B.5: Project Settings 1

- Under *C/C++ Build*, in *Settings* adjust *Tool Settings*

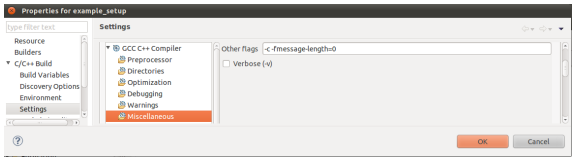


Figure B.6: Project Settings 2

- Under *GCC C++ Compiler*, click on *Miscellaneous*
- Add in the following line, exactly as written, in the *Other flags* text box

‘pkg-config –cflags playerc++‘

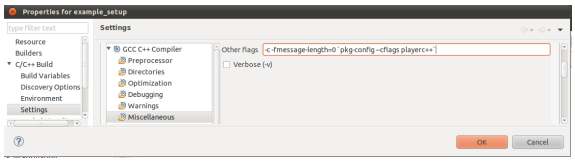


Figure B.7: Project Settings 3

- Press OK
- Scroll down to *GCC C Compiler*, and go to *Miscellaneous*
- Add in the following line, exactly as written, in the *Other flags* text box

‘pkg-config –cflags playerc‘

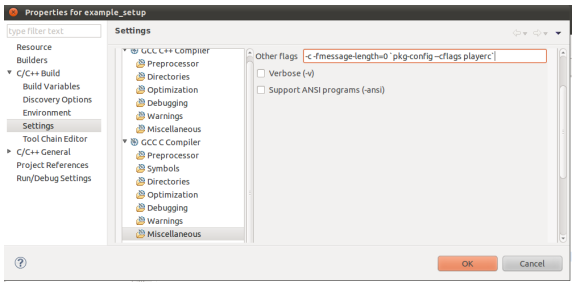


Figure B.8: Project Settings 4

- Under *GCC C++ Linker*, add in the following exactly as written in the *Linker flags* text box

‘pkg-config –libs playerc++‘

- After completing all of the above steps, press OK

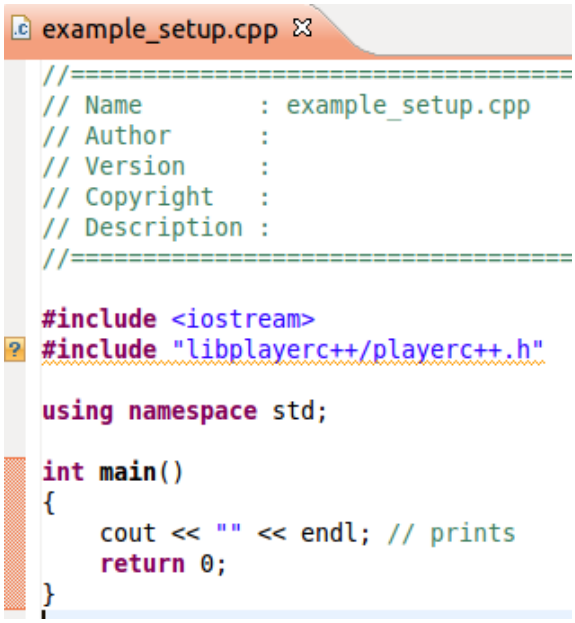


Figure B.9: Controller window

If there is still a yellow underline under # include libplayerc++, then the compiler does not know the location of the libraries. To fix this go back to Project-Properties

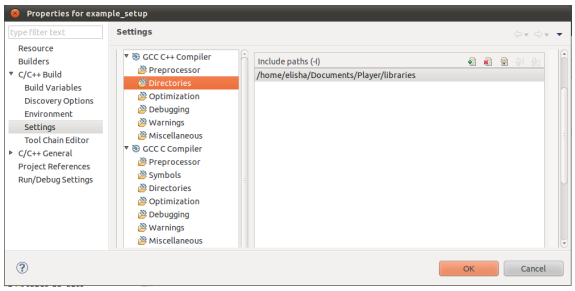


Figure B.10: Finding the location of the Player library on your system

- In the *Directories* folder for both *GCC C++ Compiler* and *GCC C Compiler* press the green plus sign and say where the player library is located on your computer
- For example, I have a copy of the library located in *Documents* inside a folder that I called *libraries*
- Your player library can be somewhere else

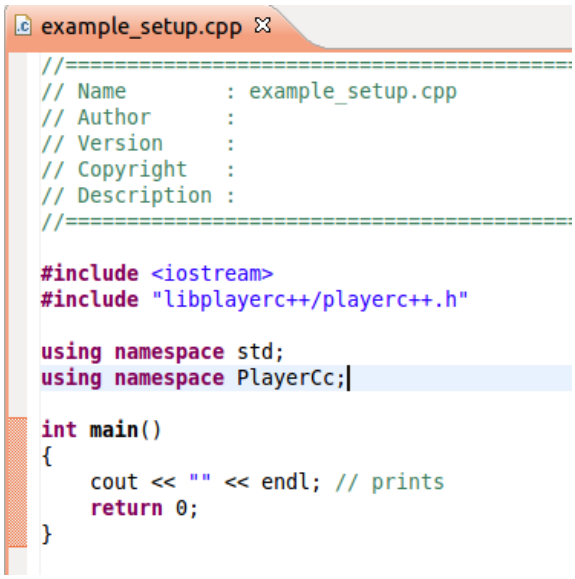


Figure B.11: Final controller window

- If there is no yellow underline after adding the line using namespace PlayerCc, then Eclipse is now set up to work with player and stage, and you can use all Player tools within its library

Appendix C

Player Library

C.1 PlayerClient Class - Functions

	Make a client and connect it as indicated PlayerClient(const std::string aHostname=PLAYER_HOSTNAME, uint aPort=PLAYER_PORTNUM, int transport=PLAYERC_TRANSPORT_TCP
	Destructor ~ PlayerClient ()
void	**Start the run thread** StartThread ()
void	**Stop the run thread** StopThread ()
void	**This starts a blocking loop on Read()** Run ()
void	**Stops the Run() loop** Stop ()
bool	**Check whether there is data waiting on the connection, blocking for up to timeout milliseconds (set to 0 to not block)** Peek (uint timeout=0)
void	**Set connection retry limit, which is the number of times that we'll try to reconnect to the server after a socket error** SetRetryLimit (int limit)
int	**Get connection retry limit, which is the number of times that we'll try to reconnect to the server after a socket error** GetRetryLimit (int limit)

void	**Set connection retry time, which is number of seconds to wait between reconnection attempts** SetRetryTime (double time)
double	**Get connection retry time, which is number of seconds to wait between reconnection attempts** GetRetryTime (double time)
void	**A blocking Read** Read ()
void	**A nonblocking Read** ReadIfWaiting ()
void	**Set whether the client operates in Push/Pull modes** SetDataMode (uint aMode)
void	**Set a replace rule for the clients queue on the server** SetReplaceRule (bool aReplace, int aType=-1, int aSubtype=-1, int aInterf=-1)
void	**Get the list of available device ids.** RequestDeviceList ()
std::string	**Returns the hostname** GetHostname () const
int	**Returns the port** GetPort () const
int	**Get the interface code for a given name** LookupCode (std::string aName) const
std::string	**Get the name for a given interface code** LookupName (int aCode) const

C.2 Position2dProxy Class - Functions

	Constructor Position2dProxy (PlayerClient *aPc, uint aIndex=0)
	Destructor ~ Position2dProxy ()
void	**Send a motor command for velocity control mode** SetSpeed (double aXSpeed, double aYSpeed, double aYawSpeed)
void	**Same as the previous SetSpeed(), but doesn't take the yspeed speed (so use this one for non-holonomic robots).** SetSpeed (double aXSpeed, double aYawSpeed)
void	**Send a motor command for velocity/heading control mode** SetVelHead (double aXSpeed, double aYSpeed, double aYawHead)
void	**Same as the previous SetVelHead(), but doesn't take the yspeed speed (so use this one for non-holonomic robots)** SetVelHead (double aXSpeed, double aYawHead)
void	**Send a motor command for position control mode** GoTo (pose, vel)
void	**Same as the previous GoTo(), but doesn't take speed** GoTo (pose)
void	**Same as the previous GoTo(), but only takes position arguments, no motion speed setting** GoTo (double aX, double aY, double aYaw)
void	**Sets command for carlike robot** GoTo SetCarlike (double aXSpeed, double aDriveAngle)

void	**Get the device's geometry; it is read into the relevant class attributes** RequestGeom ()
void	**Accessor for the pose (fill it in by calling RequestGeom)** GetPose ()
void	**Enable/disable the motors** SetMotorEnable (bool enable)
void	**Reset odometry to (0,0,0)** ResetOdometry ()
void	**Sets the odometry to the pose (x, y, yaw)** SetOdometry (double aX, double aY, double aYaw)
double	**Accessor method** GetXPos () const
double	**Accessor method** GetYPos () const
double	**Accessor method** GetYawPos () const
double	**Accessor method** GetXSpeed () const
double	**Accessor method** GetYSpeed () const
double	**Accessor method** GetYawSpeed () const
bool	**Accessor method** GetStall () const

C.3 SonarProxy Class

	Constructor SonarProxy (PlayerClient *aPc, uint aIndex=0)
	Destructor ~ SonarProxy ()
int	**Return the sensor count** GetCount () const
double	**Return a particular scan value** GetScan (uint aIndex) const
double	**This operator provides an alternate way to access the scan data** operator[] (uint aIndex) const
int	**Number of valid sonar poses.** GetPoseCount () const
double	**Sonar poses (m,m,radians)** GetPose (uint aIndex) const
void	**Request the sonar geometry** RequestGeom ()

Appendix D

How to Build a Simple Simulation in MATLAB

D.1 Plot Animation

```
% Clear all windows
clc
clf
clear

% Time settings
t_sim = 5;
T = 0.1;
t_steps = floor(t_sim/T);

% Map
map_max = 50;
map_min = -30;

% Robot with a circular path
R = 20;
gamma = 0.1;

% Create an initial pose vector
% pose = [0 0 0];
pose = zeros(t_steps,3);


% Movie
j = 0;
for t = 1:t_steps

    % Clear figure
    clf

    % Axis properties
    xmax = map_max;
    xmin = map_min;
```

```
34      ymax = map_max;
35      ymin= map_min;
36      axis([xmin, xmax, ymin, ymax]);
37      axis equal;     axis manual;     grid on;
38
39      % x, y, theta values
40      pose(t,1) = R*sin(gamma*t);
41      pose(t,2) = R*(1-cos(gamma*t));
42      pose(t,3) = gamma*t;
43
44      % Plot the animation
45      hold on;
46      plot(pose(1:t,1), pose(1:t,2), 'b');
47      hold off;
48
49      % Plot labels
50      xlabel('x [m]');
51      ylabel('y [m]');
52
53      % Capture the movie
54      j = j + 1;
55      M(j) = getframe(gcf);
56
57  end
58
59  % Create a movie and save it as an AVI file
60  movie2avi(M, 'plot_animation.avi');
```

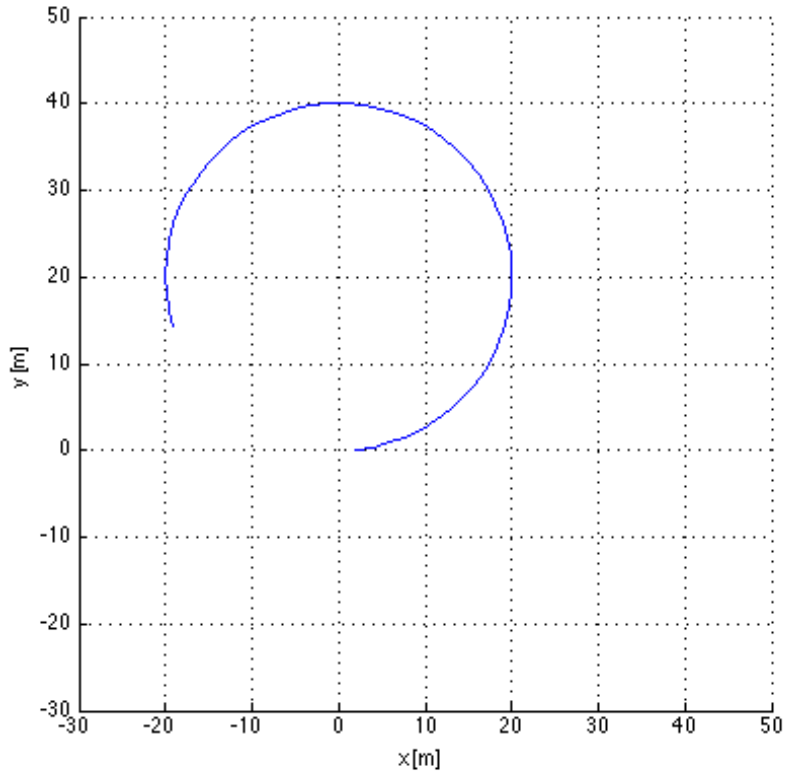


Figure D.1: MATLAB figure for plot animation simulation

D.2 Arrow Geometry Function

```
function [X,Y] = moving_arrows(Q)
x     = Q(1);
y     = Q(2);
theta = Q(3);

l = 1;
X = [x,x+l*cos(theta-2*pi/3),x+l*cos(theta),...
     x+l*cos(theta+2*pi/3),x];
Y = [y,y+l*sin(theta-2*pi/3),y+l*sin(theta),...
     y+l*sin(theta+2*pi/3),y];
```

D.3 Arrow Animation

```
% Clear command window
clc
clf
clear

% Time settings
t_sim = 10;
T = 0.1;
t_steps = floor(t_sim/T);

% Map
map_max = 50;
map_min = -30;

% Leader with a circular path
R = 20;
gamma = 0.1;

% Initialize pose
pose = zeros(t_steps,3);


% Movie
j = 0;
for t = 1:t_steps

    % Clear figure
    clf

    % Axis properties
    xmax = map_max;
    xmin = map_min;
    ymax = map_max;
    ymin= map_min;
    axis([xmin, xmax, ymin, ymax]);
    axis equal;     axis manual;     grid on;

    % x, y, theta values
    pose(t,1) = R*sin(gamma*t);
    pose(t,2) = R*(1-cos(gamma*t));
    pose(t,3) = gamma*t;

    % Get the arrow shape coordinates - use fill function to draw the
        shape
    [X,Y] = moving_arrows(pose(t,:));

    % Plot the animation
    hold on;
    fill(X,Y,'b');
    hold off;

    % Plot labels
```

```
52      xlabel('x [m]');
53      ylabel('y [m]');
54
55      % Capture the movie
56      j = j + 1;
57      M(j) = getframe(gcf);
58
59  end
60
61  % Create a movie and save it as an AVI file
62  movie2avi(M, 'arrow_animation.avi');
```

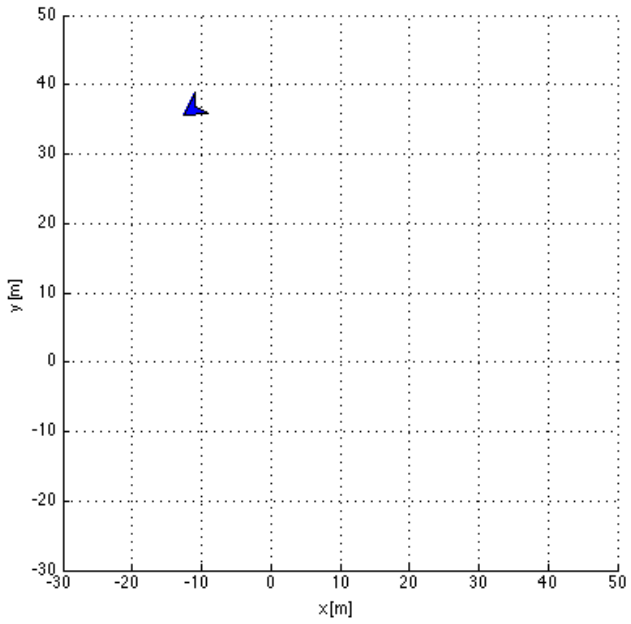


Figure D.2: MATLAB figure for arrow animation simulation

D.4 Arrow and Plot Animation

```
% Clear command window
clc
clf
clear

% Time settings
t_sim = 10;
T = 0.1;
t_steps = floor(t_sim/T);

% Map
map_max = 50;
map_min = -30;

% Leader with a circular path
R = 20;
gamma = 0.1;

% Initialize pose
pose = zeros(t_steps,3);


% Movie
j = 0;
for t = 1:t_steps

    % Clear figure
    clf

    % Axis properties
    xmax = map_max;
    xmin = map_min;
    ymax = map_max;
    ymin= map_min;
    axis([xmin, xmax, ymin, ymax]);
    axis equal;     axis manual;     grid on;

    % x, y, theta values
    pose(t,1) = R*sin(gamma*t);
    pose(t,2) = R*(1-cos(gamma*t));
    pose(t,3) = gamma*t;

    % Get the arrow shape coordinates - use fill function to draw the
        shape
    [X,Y] = moving_arrows(pose(t,:));

    % Plot the animation
    hold on;
    fill(X,Y,'b');
    plot(pose(1:t,1), pose(1:t,2), 'b');
    hold off;

```

```
52      % Plot labels
53      xlabel('x [m]');
54      ylabel('y [m]');
55
56      % Capture the movie
57      j = j + 1;
58      M(j) = getframe(gcf);
59
60  end
61
62  % Create a movie and save it as an AVI file
63  movie2avi(M, 'arrow_animation.avi');
```

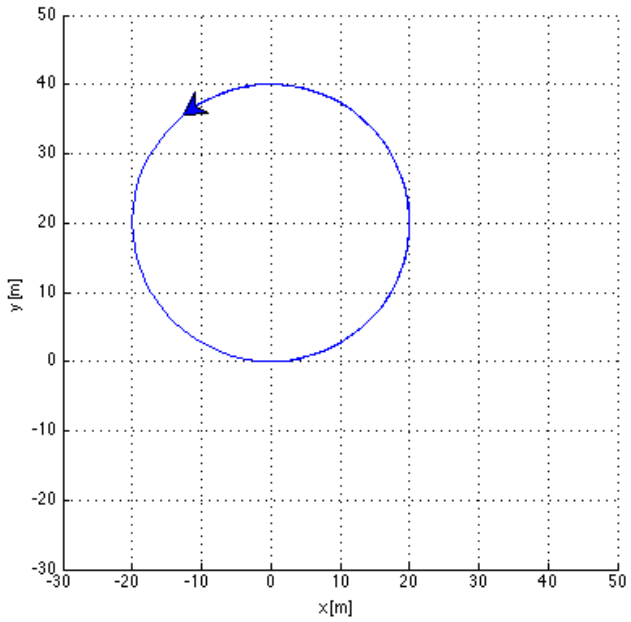


Figure D.3: MATLAB figure for arrow plot animation simulation

Appendix E

How to Build a Simple Simulation in Stage

E.1 Robot Controller

```
1 //========================================================
2 // Name        : SimpleSim.cpp
3 // Author      : Elisha Pruner
4
5 // Description : A simple simulation where a robot travels in a
      straight path.
6 //          If the robot sees an obstacle, it turns left until it can
7 //          continue on a straight path again.
8 //========================================================
9
10 #include <iostream>
11 #include <cmath>
12 #include <csignal>
13 #include "libplayerc++/playerc++.h"
14
15 using namespace std;
16 using namespace PlayerCc;
17
18 // ----------------------------------------------------------
19
20 // FUNCTION PROTOTYPES
21
22 void moveRobot(float&, float&, SonarProxy&, SonarProxy&);
23 volatile sig_atomic_t sigShutdown;
24 void UnexpectedShutdown (void);
25 void HandleShutdownSignal(int);
26
27 // ----------------------------------------------------------
28
29 // GLOBAL VARIABLES
30
31 float vmax = 0.4;
32 float omega_max = 0.2;
```

```
33 float max_dist = 0.4;
34
35 // ------------------------------------------------------------
36
37 // MAIN FUNCTION
38
39 int main()
40 {
41
42   try
43   {
44     // Initial velocity
45     float v = 0;
46     float omega = 0;
47
48     // Connect to the Player Proxies
49     PlayerClient robot("localhost", 6665);
50     Position2dProxy pose(&robot, 0);
51     SonarProxy  sonar(&robot, 0);
52     SonarProxy  ir(&robot, 1);
53
54     // Handle unexpected shutdown
55     UnexpectedShutdown();
56
57     // Loop will continue until user manually turns it off
58     while (sigShutdown == 0)
59     {
60       // read from the proxies
61       robot.Read();
62       pose.RequestGeom();
63       sonar.RequestGeom();
64       ir.RequestGeom();
65
66       // find the velocity and angular velocity to move the robot
67       moveRobot(v, omega, sonar, ir);
68
69       // send velocity command to the motors
70       pose.SetSpeed(v, omega);
71     }
72
73     // stop the vehicle by using a keyboard stroke
74     pose.SetSpeed(0, 0);
75     cout << "robot stopped" <<endl;
76     sleep(100000);
77
78     return 0;
79   }
80   catch (PlayerCc::PlayerError e)
81   {
82     std::cerr << e << std::endl;
83     return -1;
84   }
85 }
86
```

```
87 // ---------------------------------------------------------------
88
89 // FUNCTIONS
90
91 void moveRobot(float &v, float &omega, SonarProxy &sonar, SonarProxy &
       ir)
92 {
93   int too_close = 0;
94
95   // forward looking sonars are 0, 1 2
96   for(int i=0; i<=2; i++)
97   {
98     if(sonar[i] < max_dist)
99     {
100      too_close = 1;
101      break;
102    }
103  }
104
105  // forward looking ir sensors are 0,1,2,3
106  if( too_close == 0 )
107  {
108    for(int i=0; i<=3; i++)
109    {
110      if(ir[i] < max_dist)
111      {
112        too_close = 1;
113        break;
114      }
115    }
116  }
117
118  // update velocity values
119  if(too_close == 1)
120  {
121    v = 0.0;
122    omega = omega_max;
123  }
124  else
125  {
126    v = vmax;
127    omega = 0.0;
128  }
129 }
130
131 void UnexpectedShutdown (void)
132 {
133   sigShutdown = 0;
134   sigset_t blockmask;
135   struct sigaction sa;
136   sigfillset(&blockmask);
137   sa.sa_handler = HandleShutdownSignal;
138   sa.sa_mask = blockmask;
139   sa.sa_flags = 0;
```

```
140   sigaction(SIGINT, &sa, NULL);
141   sigaction(SIGQUIT, &sa, NULL);
142 }
143
144 void HandleShutdownSignal (int sig __attribute__ ((unused)))
145 {
146    sigShutdown = 1;
147 }
```

E.2 Robot Simulation Configuration File

```
# STAGE DRIVER ------

driver
(
  name "stage"
  provides ["simulation:0"]
  plugin "stageplugin"
  worldfile "AutoWorldFiles/robot1.world"
)

# X80 DRIVERS ------

driver
(
  name "stage"
  provides ["6665:position2d:0" "6665:sonar:0" "6665:sonar:1"]
  model "robot1"
)
```

E.3 Robot World File

```
include "IncludeFiles/map.inc"
include "IncludeFiles/X80.inc"

# OPEN ENVIRONMENT FILE ------
map
(
  bitmap "IncludeFiles/bitmap/warehouse.png"
  size [24 12 1]
  origin [0 0 0 0]
)

# ADD GOAL POSITION IN THE TERRAIN ------
rect_goal(name "goal" pose [8 2 0 0])


# ADD ROBOTS TO THE MAP ------
X80
(
  name "robot1"
  pose [-5 0 0 0]
  color "red"
)


# CONFIGURE GUI WINDOW ------
window
(
  size [1400 700]
  scale 50
  center [0 0]
  rotate [0 0]

  show_data 1
  show_flags 0
  show_blocks 1
  show_clock 0
  show_footprints 0
  show_grid 1
  show_trailarrows 0
  show_trailrise 0
  show_trailfast 0
  show_occupancy 0
  pcam_on 0
  screenshots 0
```

E.4 World File Map

```
1
2 define map model
3 (
4   # sombre, sensible, artistic
5   color "black"
6
7   # most maps will need a bounding box
8   boundary 1
9   gui_nose 1
10  gui_grid 1
11  gui_outline 0
12  fiducial_return 0
13  gripper_return 0
14  obstacle_return 1
15  ranger_return 1
16
17 )
18
19 define rect_goal model
20 (
21 polygons 1
22 polygon[0].points 4
23 polygon[0].point[0] [0 0]
24 polygon[0].point[1] [0 1]
25 polygon[0].point[2] [1 1]
26 polygon[0].point[3] [1 0]
27 size [1 1 0]
28 color "LightYellow"
29 obstacle_return 0
30 ranger_return 0
31 boundary 0
32 )
33
34 define rect_obst model
35 (
36 polygons 1
37 polygon[0].points 4
38 polygon[0].point[0] [0 0]
39 polygon[0].point[1] [0 1]
40 polygon[0].point[2] [1 1]
41 polygon[0].point[3] [1 0]
42 size [2 1 1]
43 color "black"
44 obstacle_return 1
45 ranger_return 1
46 boundary 1
47 )
48
49 define rect_obst2 model
50 (
51 polygons 1
52 polygon[0].points 4
```

```
53 polygon[0].point[0] [0 0]
54 polygon[0].point[1] [0 1]
55 polygon[0].point[2] [1 1]
56 polygon[0].point[3] [1 0]
57 size [2 1 1]
58 color "black"
59 obstacle_return 0
60 ranger_return 0
61 boundary 0
62 )
63
64 define blob_obst model
65 (
66 bitmap "bitmap/blob5.png"
67 size [0.15 0.15 0.15]
68 color "black"
69 obstacle_return 1
70 ranger_return 1
71 boundary 1
72 )
```

E.5 World File X80

```
1
2 define X80_base position
3 (
4   drive "diff"      # Differential steering model.
5   gui_nose 1                    # Draw a nose on the robot so we can
        see which way it points
6   obstacle_return 1             # Can hit things.
7   laser_return 1                  # reflects laser beams
8   ranger_return 1               # reflects sonar beams
9   blob_return 1                 # Seen by blobfinders
10  fiducial_return 1             # Seen as "1" fiducial finders
11
12  localization_origin [0 0 0 0]
13    localization "gps"
14
15
16  # X80 Sensors
17  X80_sonars()
18  X80_infra_reds()
19 )
20
21
22 define X80 X80_base
23 (
24  origin [0.0 0.025 0.0 -90.0]  # center of rotation offset is between
        the wheels
25  mass 3.5
26  size [0.33 0.36 0.26]
27
28  # top
29        block
30  (
31    points 6
32    point[0] [ 0.0 0.21 ]
33    point[1] [  0.0 0.12 ]
34    point[2] [  0.08  0.05  ]
35    point[3] [  0.25   0.05  ]
36    point[4] [  0.33  0.12 ]
37    point[5] [ 0.33  0.21 ]
38    z [ 0.17 0.26 ]
39    #color "LawnGreen"
40  )
41
42  # Grey Middle 1
43  block
44  (
45    points 4
46    point[0] [ 0.06  0.21]
47    point[1] [ 0.27  0.21 ]
48    point[2] [ 0.27  0.26 ]
49    point[3] [ 0.06 0.26 ]
50    z [ 0.085 0.165 ]
```

```
51
52      color "gray"
53    )
54
55    # Grey Middle Arc
56          block
57    (
58      points 11
59      point[0] [ 0.0   0.29]
60      point[1] [ 0.0  0.26 ]
61      point[2] [ 0.33  0.26 ]
62      point[3] [ 0.33  0.29 ]
63      point[4] [ 0.28   0.33 ]
64      point[5] [ 0.25   0.34 ]
65      point[6] [ 0.22   0.35 ]
66      point[7] [ 0.165  0.36 ]
67      point[8] [ 0.11   0.35 ]
68      point[9] [ 0.08   0.34 ]
69      point[10] [ 0.05   0.33 ]
70
71      z [ 0.085 0.165 ]
72      color "gray"
73    )
74
75    # left wheel
76    block
77    (
78      points 4
79      point[0] [ 0.0   0.09 ]
80      point[1] [ 0.03   0.09 ]
81      point[2] [ 0.03   0.26 ]
82      point[3] [ 0.0   0.26 ]
83      z [0 0.17 ]
84      color "gray15"
85    )
86
87    # right wheel
88    block
89    (
90      points 4
91      point[0] [ 0.3   0.09 ]
92      point[1] [ 0.33   0.09 ]
93      point[2] [ 0.33   0.26 ]
94      point[3] [ 0.3   0.26 ]
95      z [0 0.17 ]
96      color "gray15"
97    )
98  )
99
100 define X80_sonars ranger
101 (
102         # number of sonars
103     scount 6
104
```

```
105     # define the pose of each transducer [xpos ypos heading]
106   # relative to the origin
107     spose[0] [ 0.14  0.14 45 ]  # frt left
108     spose[1] [ 0.19 0.0 0 ]   # frt center
109     spose[2] [ 0.14  -0.14 -45]   # frt right
110    spose[3] [ -0.12 -0.125 -135]   # back right
111    spose[4] [ -0.18 0.0 -180]  # back center
112     spose[5] [ -0.12 0.125 135]   # back left
113
114   # define the field of view of each transducer
115   # [range_min range_max view_angle]
116    sview [0.08 2.55 10]
117
118    ssize [0.05 0.02]
119 )
120
121 define X80_infra_reds ranger
122 (
123         # number of IR sensors
124     scount 7
125
126     # define the pose of each transducer [xpos ypos heading]
127   # relative to the origin
128    spose[0] [ 0.16  0.10 40 ]  # frt left
129    spose[1] [ 0.17 0.04 10 ]   # frt-left center
130    spose[2] [ 0.17  -0.04 -10 ]  # frt-right center
131    spose[3] [ 0.16  -0.10 -40]   # frt right
132    spose[4] [ 0.0 -0.12 -90]   # right
133    spose[5] [ -0.18 0.0 -180]  # back
134     spose[6] [ 0.0 0.12 90]   # left
135
136   # adding a few sensors to simulation
137   #spose[7] [ 0.13  -0.12 -60]  # top right
138   #spose[8] [ -0.035  -0.12 -120]   # bottom right
139   #spose[9] [ 0.13  0.12 60]  # top left
140   #spose[10] [ -0.035  0.12 120] # bottom left
141
142   # define the field of view of each transducer
143   # [range_min range_max view_angle]
144    sview [0.1 1.0 5]
145
146    ssize [0.01 0.03]
147 )
```

Appendix F

How to Build a Simple Experiment with the X80 Mobile Robot Using Player

F.1 Robot Experiment Configuration File

```
1 # X80 DRIVER ---------------------------------------------------------
2
3 driver
4 (
5   name "x80"
6   plugin "../player_drivers/X80PlayerPlugin/lib/libx80"
7   provides ["6665:position2d:0" "6665:sonar:0" "6665:sonar:1"]
8   connection  "udp"
9   address "192.168.0.205"
10  port    10001     # for udp only
11  pose [0 0 0 0]
12
13  sonar_filter  1.0 # low pass filter smoothing factor, <= 1.0
14  ir_filter 1.0 # low pass filter smoothing factor, <= 1.0
15 )
```

F.2 Source Code for the X80 Player Plugin

```
1  /*
2   *  Player - One Hell of a Robot Server
3   *  Copyright (C) 2000
4   *     Brian Gerkey, Kasper Stoy, Richard Vaughan, & Andrew Howard
5   *
6   *
7   *  This program is free software; you can redistribute it and/or
        modify
8   *  it under the terms of the GNU General Public License as published
        by
9   *  the Free Software Foundation; either version 2 of the License, or
10  *  (at your option) any later version.
11  *
12  *  This program is distributed in the hope that it will be useful,
13  *  but WITHOUT ANY WARRANTY; without even the implied warranty of
14  *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
15  *  GNU General Public License for more details.
16  *
17  *  You should have received a copy of the GNU General Public License
18  *  along with this program; if not, write to the Free Software
19  *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
        02111-1307  USA
20  *
21  */
22
23 /*
24  * Dr. Robot X80 Driver for Player/Stage
25  */
26
27
28 /** @ingroup drivers */
29 /** @{ */
30 /** @defgroup driver_x80 x80
31  * @brief Dr Robot X80 platform
32
33 The X80 platform used at DRDC Ottawa uses a PMS5005 Robot Sensing/
      Motion Controller
34 and the PMB5010 Robot Multimedia Controller to drive the X80 robot and
      communicate
35 with its onboard sensors.  Implemented in this driver will be the
      motion control,
36 and feedback from the 7 IR sensors and 6 Sonar sensors mounted around
      the vehicle.
37 Sensor data is passed through a low pass filter when it arrives --
      setting the
38 smoothing factor (\alpha) to 1 effectively removes the filter
39
40 @par Compile-time dependencies
41
42 - none
43
44 @par Provides
```

```
45
46 The x80 driver provides the following device interfaces, some of
47 them named:
48
49 - @ref interface_position2d
50   - This interface returns position and velocity data, and accepts
       velocity commands,
51     and commands to set the velocity PID gains.
52   - PLAYER_POSITION2D_DATA_STATE
53   - PLAYER_POSITION2D_DATA_GEOM
54
55 - @ref interface_sonar:0
56   - This interface returns the Sonar range data.
57     - PLAYER_SONAR_DATA_RANGES
58     - PLAYER_SONAR_DATA_GEOM
59
60 - @ref interface_sonar:1
61   - This interface returns the IR range data -- the sonar interface is
       used for compatibility with Stage
62     - PLAYER_SONAR_DATA_RANGES
63     - PLAYER_SONAR_DATA_GEOM
64
65 @par Supported configuration requests
66
67 - @ref interface_position2d :
68   - PLAYER_POSITION2D_REQ_SET_ODOM
69   - PLAYER_POSITION2D_REQ_RESET_ODOM
70   - PLAYER_POSITION2D_REQ_GET_GEOM
71   - PLAYER_POSITION2D_REQ_MOTOR_POWER
72
73 - @ref interface_sonar : 0-1
74   - PLAYER_SONAR_REQ_GET_GEOM
75
76 @par Supported commands
77
78 - @ref interface_position2d :
79   - PLAYER_POSITION2D_CMD_VEL
80   - PLAYER_POSITION2D_REQ_SPEED_PID
81
82
83 @par Configuration file options
84
85 - address (string) -- serial port for serial connection or ip address
     for UDP connection
86   - Default: "/dev/ttyS0"
87
88 - connection (string) -- "serial" or "udp"
89   - Default: "serial"
90
91 - port (int) -- socket to communicate with the x80, only valid with udp
       connection
92   - Default: 10001
93
94 - pose (tupleFloat) -- initial odometry pose for the vehicle
```

```
95   - Default (0,0,0,0)
96
97 - sonar_filter(float) -- smoothing factor for sonar data
98   - Default 1.0
99
100 - ir_filter(float) -- smoothing factor for ir data
101   - Default 1.0
102
103 @par Example
104
105 @verbatim
106 driver
107 (
108   name "x80"
109   plugin "libx80r"
110   provides ["position2d:0" "sonar:0" "sonar:1"]
111
112   connection  "serial"    # or udp
113   address     "/dev/ttyS0"  # ip address for udp
114   port       10001     # for udp only
115   pose [0 0 0 0]
116   sonar_filter  0.5   # low pass filter smoothing factor, <= 1.0
117   ir_filter    0.5   # low pass filter smoothing factor, <= 1.0
118 )
119
120 @endverbatim
121
122 @author Luc Brunet, LB Robotics
123 */
124 /** @} */
125
126 #include <unistd.h>
127 #include <string.h>
128 #include <math.h>
129 #include <time.h>
130
131 #include "x80.h"
132
133 // A factory creation function, declared outside of the class so that
       it
134 // can be invoked without any object context (alternatively, you can
135 // declare it static in the class).  In this function, we create and
       return
136 // (as a generic Driver*) a pointer to a new instance of this driver.
137 Driver*
138 x80_Init(ConfigFile* cf, int section)
139 {
140   // Create and return a new instance of this driver
141   return((Driver*)(new x80(cf, section)));
142 }
143
144 // A driver registration function, again declared outside of the class
       so
```

```
145 // that it can be invoked without object context.  In this function, we
        add
146 // the driver into the given driver table, indicating which interface
        the
147 // driver can support and how to create a driver instance.
148 void
149 x80_Register(DriverTable* table)
150 {
151   table->AddDriver("x80", x80_Init);
152 }
153
154 double
155 irValue2Distance(uint16_t value)
156 {
157   /*
158    * Data plots give the following values
159    *
160    *  dist     voltage
161    *  0.1      2.3
162    *  0.12     2
163    *  0.15     1.7
164    *  0.2      1.36
165    *  0.3      0.9
166    *  0.35     0.81
167    *  0.4      0.7
168    *  0.6      0.5
169    *  0.8      0.4
170    *
171    *  Assume a linear scale between points
172    */
173
174   double voltage = (double)value * X80_AD_MULTIPLIER;
175
176   if( voltage < 0.4 )
177   {
178     return 1.0;
179   }
180   else if( voltage < 0.5 )
181   {
182     return (voltage - 0.5) * (0.8 - 0.6) / (0.4 - 0.5) + 0.6;
183   }
184   else if( voltage < 0.7 )
185   {
186     return (voltage - 0.7) * (0.6 - 0.4) / (0.5 - 0.7) + 0.4;
187   }
188   else if( voltage < 0.8 )
189   {
190     return (voltage - 0.81) * (0.4 - 0.35) / (0.7 - 0.81) + 0.35;
191   }
192   else if( voltage < 0.9 )
193   {
194     return (voltage - 0.9) * (0.35 - 0.3) / (0.81 - 0.9) + 0.3;
195   }
196   else if( voltage < 1.36 )
```

```
197   {
198     return (voltage - 1.36) * (0.3 - 0.2) / (0.9 - 1.36) + 0.2;
199   }
200   else if( voltage < 1.7 )
201   {
202     return (voltage - 1.7) * (0.2 - 0.15) / (1.36 - 1.7) + 0.15;
203   }
204   else if( voltage < 2.0 )
205   {
206     return (voltage - 2.0) * (0.15 - 0.12) / (1.7 - 2.0) + 0.12;
207   }
208   else if( voltage < 2.3 )
209   {
210     return (voltage - 2.3) * (0.12 - 0.1) / (2.0 - 2.3) + 0.1;
211   }
212   else
213   {
214     return 0.05;
215   }
216 }
217
218 ////////////////////////////////////
219 // Driver Callbacks
220
221 // ACK messages from the platform
222 void
223 X80_ACK_cb(X80_Message *msg, void *userData)
224 {
225   x80_Data *data = (x80_Data*)userData;
226
227   if(data == NULL)
228     return;
229
230   pthread_mutex_lock(&data->lock);
231
232   // TODO: process ACK
233
234   pthread_mutex_unlock(&data->lock);
235 }
236
237 // Sensor Information messages from the platform
238 void
239 X80_SensorData_cb(X80_Message *msg, void *userData)
240 {
241   x80_Data *uData = (x80_Data*)userData;
242
243   if(uData == NULL)
244     return;
245
246   X80_Sensor_Data *mData = (X80_Sensor_Data *)msg->data;
247
248   /*
249    * Implemented here is a low pass filter on the output
250    * y[i] = \alpha*x[i] + (1 - \alpha)*y[i-1]
```

```
251     * where \alpha is read in from the config file
252     *
253     * the first values are not filtered
254     *
255     * sonar information is received in cm
256     */
257
258   pthread_mutex_lock(&uData->lock);
259
260
261   if(uData->sonar.ranges[0] < 0.0)
262   {
263     uData->sonar.ranges[0] = (float)mData->sonar1 *
          X80_SONAR_MULTIPLIER;
264     uData->sonar.ranges[1] = (float)mData->sonar2 *
          X80_SONAR_MULTIPLIER;
265     uData->sonar.ranges[2] = (float)mData->sonar3 *
          X80_SONAR_MULTIPLIER;
266     uData->sonar.ranges[3] = (float)mData->sonar4 *
          X80_SONAR_MULTIPLIER;
267     uData->sonar.ranges[4] = (float)mData->sonar5 *
          X80_SONAR_MULTIPLIER;
268     uData->sonar.ranges[5] = (float)mData->sonar6 *
          X80_SONAR_MULTIPLIER;
269
270     uData->ir.ranges[0] = irValue2Distance(mData->infraRedValue);
271   }
272   else
273   {
274     uData->sonar.ranges[0] = uData->sonar_alpha * (float)mData->sonar1
          * X80_SONAR_MULTIPLIER
275         + (1.0 - uData->sonar_alpha) * uData->sonar.ranges[0];
276     uData->sonar.ranges[1] = uData->sonar_alpha * (float)mData->sonar2
          * X80_SONAR_MULTIPLIER
277         + (1.0 - uData->sonar_alpha) * uData->sonar.ranges[1];
278     uData->sonar.ranges[2] = uData->sonar_alpha * (float)mData->sonar3
          * X80_SONAR_MULTIPLIER
279         + (1.0 - uData->sonar_alpha) * uData->sonar.ranges[2];
280     uData->sonar.ranges[3] = uData->sonar_alpha * (float)mData->sonar4
          * X80_SONAR_MULTIPLIER
281         + (1.0 - uData->sonar_alpha) * uData->sonar.ranges[3];
282     uData->sonar.ranges[4] = uData->sonar_alpha * (float)mData->sonar5
          * X80_SONAR_MULTIPLIER
283         + (1.0 - uData->sonar_alpha) * uData->sonar.ranges[4];
284     uData->sonar.ranges[5] = uData->sonar_alpha * (float)mData->sonar6
          * X80_SONAR_MULTIPLIER
285         + (1.0 - uData->sonar_alpha) * uData->sonar.ranges[5];
286
287     uData->ir.ranges[0] = uData->ir_alpha * irValue2Distance(mData->
          infraRedValue)
288         + (1.0 - uData->ir_alpha) * uData->ir.ranges[0];
289   }
290
291   uData->new_sonar = 1;
```

```
292
293   pthread_mutex_unlock(&uData->lock);
294
295 }
296
297 // Motor Information messages from the platform
298 void
299 X80_MotorData_cb(X80_Message *msg, void *userData)
300 {
301   double vLeft = 0.0, vRight = 0.0;
302   double xLeft = 0.0, xRight = 0.0;
303   double deltaHdg = 0.0, deltaD = 0.0;
304
305   x80_Data *uData = (x80_Data*)userData;
306
307   if(uData == NULL)
308     return;
309
310   X80_Motor_Control_Data *mData = (X80_Motor_Control_Data *)msg->data;
311
312   pthread_mutex_lock(&uData->lock);
313
314   vLeft = PI * X80_WHEEL_DIAMETER * ((float)mData->encoder1Speed / (
        float)X80_WHEEL_TICKS_PER_REV);
315   vRight = PI * X80_WHEEL_DIAMETER * ((float)mData->encoder2Speed / (
        float)X80_WHEEL_TICKS_PER_REV);
316
317   uData->position.vel.px = (vLeft + vRight) / 2.0; // forward velocity
        is the average of the 2 wheels
318   uData->position.vel.py = 0.0; // no side slip
319   uData->position.vel.pa = (vRight - vLeft) / (X80_MIN_TURN_RADIUS *
        2.0);
320
321
322   // cannot do any position calculations if it is the first point
323   if((uData->encoder1PulseLast < 0) || (uData->encoder2PulseLast < 0))
324   {
325     uData->encoder1PulseLast = mData->encoder1Pulse;
326     uData->encoder2PulseLast = mData->encoder2Pulse;
327     pthread_mutex_unlock(&uData->lock);
328     return;
329   }
330
331   /*
332    * this will handle the case where it goes over the max encoder
        values
333    * and restarts at 0 .... both directions
334    * it assume that if the change is greater than one revolution,
335    * then the encoder has gone past the max value
336    *
337    * Remember that the right encoder always goes in the reverse
        direction
338    */
339   xLeft = mData->encoder1Pulse - uData->encoder1PulseLast;
```

```
340   if(abs(xLeft) > X80_WHEEL_TICKS_PER_REV)
341   {
342     xLeft = (uData->encoder1PulseLast > mData->encoder1Pulse) ?
343         X80_MAX_ENCODER_TICKS : -X80_MAX_ENCODER_TICKS;
344     xLeft += - uData->encoder1PulseLast + mData->encoder1Pulse;
345   }
346
347   xRight = -(mData->encoder2Pulse - uData->encoder2PulseLast);
348   if(abs(xRight) > X80_WHEEL_TICKS_PER_REV)
349   {
350     xRight = (uData->encoder2PulseLast > mData->encoder2Pulse) ?
351         X80_MAX_ENCODER_TICKS : -X80_MAX_ENCODER_TICKS;
352     xRight += - uData->encoder2PulseLast + mData->encoder2Pulse;
353   }
354
355   xLeft = PI * X80_WHEEL_DIAMETER * (xLeft / (double)
        X80_WHEEL_TICKS_PER_REV);
356   xRight = PI * X80_WHEEL_DIAMETER * (xRight / (double)
        X80_WHEEL_TICKS_PER_REV);
357
358   // calculate the distance travelled and the amount of rotation since
        the last update
359   deltaHdg = (xRight - xLeft) / (2.0 * X80_MIN_TURN_RADIUS);
360   deltaD = (xRight + xLeft) / 2.0;
361
362   uData->position.pos.px += deltaD * cos(uData->position.pos.pa +
        deltaHdg/2.0);
363   uData->position.pos.py += deltaD * sin(uData->position.pos.pa +
        deltaHdg/2.0);
364   uData->position.pos.pa += deltaHdg;
365
366   // normalize the heading
367   if(uData->position.pos.pa >  PI)
368   {
369     uData->position.pos.pa -= 2 * PI;
370   }
371   else if (uData->position.pos.pa <  -PI)
372   {
373     uData->position.pos.pa += 2.0 * PI;
374   }
375
376   /*printf("x: %.2f y: %.2f hdg: %.2f xRight: %.2f xLeft: %.2f\n",
377       uData->position.pos.px, uData->position.pos.py,
378       uData->position.pos.pa, xRight, xLeft);*/
379
380   uData->encoder1PulseLast = mData->encoder1Pulse;
381   uData->encoder2PulseLast = mData->encoder2Pulse;
382
383   uData->new_pos = 1;
384
385   pthread_mutex_unlock(&uData->lock);
386 }
387
388
```

```
389 // Custom Data (IR sensors) Information messages from the platform
390 void
391 X80_CustomData_cb(X80_Message *msg, void *userData)
392 {
393   x80_Data *uData = (x80_Data*)userData;
394
395   if(uData == NULL)
396     return;
397
398   X80_Custom_Data *mData = (X80_Custom_Data *)msg->data;
399
400   /*
401    * Implemented here is a low pass filter on the output
402    * y[i] = \alpha*x[i] + (1 - \alpha)*y[i-1]
403    * where \alpha is read in from the config file
404    *
405    * the first values are not filtered
406    */
407
408   pthread_mutex_lock(&uData->lock);
409
410   if(uData->ir.ranges[1] < 0.0)
411   {
412     uData->ir.ranges[1] = irValue2Distance(mData->adChannel3);
413     uData->ir.ranges[2] = irValue2Distance(mData->adChannel4);
414     uData->ir.ranges[3] = irValue2Distance(mData->adChannel5);
415     uData->ir.ranges[4] = irValue2Distance(mData->adChannel6);
416     uData->ir.ranges[5] = irValue2Distance(mData->adChannel7);
417     uData->ir.ranges[6] = irValue2Distance(mData->adChannel8);
418   }
419   else
420   {
421     uData->ir.ranges[1] = uData->ir_alpha * irValue2Distance(mData->
          adChannel3)
422         + (1.0 - uData->ir_alpha) * uData->ir.ranges[1];
423     uData->ir.ranges[2] = uData->ir_alpha * irValue2Distance(mData->
          adChannel4)
424         + (1.0 - uData->ir_alpha) * uData->ir.ranges[2];
425     uData->ir.ranges[3] = uData->ir_alpha * irValue2Distance(mData->
          adChannel5)
426         + (1.0 - uData->ir_alpha) * uData->ir.ranges[3];
427     uData->ir.ranges[4] = uData->ir_alpha * irValue2Distance(mData->
          adChannel6)
428         + (1.0 - uData->ir_alpha) * uData->ir.ranges[4];
429     uData->ir.ranges[5] = uData->ir_alpha * irValue2Distance(mData->
          adChannel7)
430         + (1.0 - uData->ir_alpha) * uData->ir.ranges[5];
431     uData->ir.ranges[6] = uData->ir_alpha * irValue2Distance(mData->
          adChannel8)
432         + (1.0 - uData->ir_alpha) * uData->ir.ranges[6];
433   }
434
435   uData->new_ir = 1;
436
```

```
437   pthread_mutex_unlock(&uData->lock);
438 }
439
440 ////////////////////////////////////////
441 // Constructor.  Retrieve options from the configuration file and do
      any
442 // pre-Setup() setup.
443 x80::x80(ConfigFile* cf, int section)
444     : Driver(cf, section, false, PLAYER_MSGQUEUE_DEFAULT_MAXLEN)
445 {
446   const char* c;
447
448   _driver = NULL;
449
450   this->_address = cf->ReadString(section,"address",DEFAULT_ADDRESS);
451
452   c = cf->ReadString(section,"connection",DEFAULT_CONNECTION);
453   _connection = (strcmp(c, "udp") == 0) ? X80_CONNECT_UDP :
        X80_CONNECT_SERIAL;
454
455   this->_port = cf->ReadInt(section, "port", DEFAULT_PORT);
456
457   memset(&_data.position, 0, sizeof(player_position2d_data_t));
458   _data.position.pos.px = cf->ReadTupleFloat(section, "pose", 0, 0.0);
459   _data.position.pos.py = cf->ReadTupleFloat(section, "pose", 1, 0.0);
460   _data.position.pos.pa = cf->ReadTupleFloat(section, "pose", 3, 0.0);
461
462   if(cf->ReadDeviceAddr(&(this->_position2d_id), section, "provides",
463       PLAYER_POSITION2D_CODE, -1, NULL) == 0)
464   {
465     if(this->AddInterface(this->_position2d_id) != 0)
466     {
467       this->SetError(-1);
468       return;
469     }
470   }
471
472   if(cf->ReadDeviceAddr(&(this->_sonar_id), section, "provides",
473       PLAYER_SONAR_CODE, 1, NULL) == 0)
474   {
475     if(this->AddInterface(this->_sonar_id) != 0)
476     {
477       this->SetError(-1);
478       return;
479     }
480   }
481
482   if(cf->ReadDeviceAddr(&(this->_ir_id), section, "provides",
483       PLAYER_SONAR_CODE, 2, NULL) == 0)
484   {
485     if(this->AddInterface(this->_ir_id) != 0)
486     {
487       this->SetError(-1);
488       return;
```

```
489       }
490     }
491
492     _data.sonar_alpha = cf->ReadFloat(section, "sonar_filter",
          DEFAULT_SONAR_ALPHA);
493     _data.ir_alpha = cf->ReadFloat(section, "ir_filter", DEFAULT_IR_ALPHA
          );
494
495     this->SetGeometry();
496
497     return;
498 }
499
500 x80::~x80()
501 {
502     this->Shutdown();
503
504     delete [] _sonar_geom.poses;
505     delete [] _ir_geom.poses;
506     delete [] _data.ir.ranges;
507     delete [] _data.sonar.ranges;
508
509     if(_driver)
510       delete _driver;
511 }
512
513 ////////////////////////////////////////
514 // Set up the device.  Return 0 if things go well, and -1 otherwise.
515 int
516 x80::Setup()
517 {
518
519     puts("X80 Driver initialising");
520     _driver = new X80_Driver();
521
522     /* Connect to robot */
523     if(_connection == X80_CONNECT_UDP)
524     {
525       puts("Connecting to UDP\n");
526       if(_driver->ConnectUDP(this->_address, this->_port) < 0)
527       {
528         delete _driver;
529         return -1;
530       }
531     }
532     else
533     {
534       puts("Connecting to Serial\n");
535       if(_driver->ConnectSerial(this->_address) < 0)
536       {
537         delete _driver;
538         return -1;
539       }
540     }
```

```
541
542   /* Register Message Callbacks from the Driver */
543   _driver->RegisterCallback(X80_GET_SENSOR_FEEDBACK, &X80_SensorData_cb
          , &_data);
544   _driver->RegisterCallback(X80_GET_MOTOR_CONTROL_SIGNAL, &
          X80_MotorData_cb, &_data);
545   _driver->RegisterCallback(X80_GET_CUSTOM, &X80_CustomData_cb, &_data)
          ;
546   _driver->RegisterCallback(X80_SET_SYSTEM_COMMS, &X80_ACK_cb, &_data);
547
548   /* Setup Quadrature Encoder */
549   _driver->SetControlSensor(X80_WHEEL_LEFT, X80_QUAD_ENCODER);
550   _driver->SetControlSensor(X80_WHEEL_RIGHT, X80_QUAD_ENCODER);
551
552   /* Setup Velocity Control */
553   _driver->SetControlMethod(X80_WHEEL_LEFT, X80_CONTROL_VELOCITY);
554   _driver->SetControlMethod(X80_WHEEL_RIGHT, X80_CONTROL_VELOCITY);
555
556   /* Set initial wheel directions */
557   _driver->SetWheelPolarity(X80_WHEEL_LEFT,X80_WHEEL_POSITIVE);
558   _driver->SetWheelPolarity(X80_WHEEL_RIGHT,X80_WHEEL_POSITIVE);
559
560   /* Set Initial PID Gains */
561   _driver->SetVelocityPIDGains(X80_WHEEL_LEFT, 30, 10, 0);
562   _driver->SetVelocityPIDGains(X80_WHEEL_RIGHT, 30, 10, 0);
563
564   /* Ensure Platform is stopped for safety*/
565   _driver->StopPlatform();
566
567   /* Begin receiving platform data */
568   _driver->RequestAllSensorDataBegin();
569   _driver->Run(1000);
570
571   puts("X80 Driver ready");
572
573   // Start the device thread; spawns a new thread and executes
574   // x80::Main(), which contains the main loop for the driver.
575   StartThread();
576
577   return(0);
578 }
579
580
581 ////////////////////////////////////////////
582 // Shutdown the device
583 int
584 x80::Shutdown()
585 {
586   puts("Shutting X80 Driver down");
587
588   // Stop and join the driver thread
589   StopThread();
590
591   _driver->StopPlatform();
```

```
592
593     _driver->RequestAllSensorDataStop();
594
595     _driver->Stop();
596
597     _driver->Disconnect();
598
599     puts("X80 Driver has been shutdown");
600
601     return(0);
602 }
603
604 int
605 x80::ProcessMessage(QueuePointer & resp_queue,
606                                       player_msghdr * hdr,
607                                       void * data)
608 {
609     /* Process:
610      * PLAYER_POSITION2D_REQ_SET_ODOM
611      * PLAYER_POSITION2D_REQ_RESET_ODOM
612      * PLAYER_POSITION2D_REQ_GET_GEOM
613      * PLAYER_SONAR_REQ_GET_GEOM
614      * PLAYER_POSITION2D_CMD_VEL
615      * PLAYER_POSITION2D_REQ_SPEED_PID
616      * PLAYER_POSITION2D_REQ_MOTOR_POWER
617      */
618     if(Message::MatchMessage(hdr,PLAYER_MSGTYPE_REQ,
619             PLAYER_POSITION2D_REQ_SET_ODOM, this->_position2d_id))
620     {
621       if(hdr->size != 0)
622       {
623         PLAYER_WARN("Arg to set odometry is wrong size; ignoring");
624         return(-1);
625       }
626       player_position2d_set_odom_req_t* pose = (
             player_position2d_set_odom_req_t*)data;
627
628       pthread_mutex_lock(&_data.lock);
629       _data.position.pos = pose->pose;
630       pthread_mutex_unlock(&_data.lock);
631
632       this->Publish(this->_position2d_id, resp_queue,
633               PLAYER_MSGTYPE_RESP_ACK, PLAYER_POSITION2D_REQ_SET_ODOM);
634       return 0;
635     }
636     else if(Message::MatchMessage(hdr,PLAYER_MSGTYPE_REQ,
637         PLAYER_POSITION2D_REQ_RESET_ODOM, this->_position2d_id))
638     {
639       if(hdr->size != 0)
640       {
641         PLAYER_WARN("Arg to reset odometry is wrong size; ignoring");
642           return(-1);
643       }
644
```

```
645     pthread_mutex_lock(&_data.lock);
646     _data.position.pos.px = 0.0;
647     _data.position.pos.py = 0.0;
648     _data.position.pos.pa = 0.0;
649     pthread_mutex_unlock(&_data.lock);
650
651     this->Publish(this->_position2d_id, resp_queue,
652             PLAYER_MSGTYPE_RESP_ACK, PLAYER_POSITION2D_REQ_RESET_ODOM);
653     return 0;
654   }
655   else if(Message::MatchMessage(hdr,PLAYER_MSGTYPE_REQ,
656       PLAYER_POSITION2D_REQ_GET_GEOM, this->_position2d_id))
657   {
658
659     if(hdr->size != 0)
660     {
661       PLAYER_WARN("Arg to get robot geometry is wrong size; ignoring");
662       return(-1);
663     }
664     this->Publish(this->_position2d_id, resp_queue,
665           PLAYER_MSGTYPE_RESP_ACK, PLAYER_POSITION2D_REQ_GET_GEOM,
666           (void*)&_robot2d_geom, sizeof(_robot2d_geom), NULL);
667
668     return 0;
669   }
670   else if(Message::MatchMessage(hdr,PLAYER_MSGTYPE_REQ,
671         PLAYER_POSITION2D_REQ_MOTOR_POWER, this->_position2d_id))
672   {
673     if(hdr->size != sizeof(player_position2d_power_config_t))
674     {
675       PLAYER_WARN("Arg to motor state change request wrong size;
              ignoring");
676       return -1;
677     }
678
679     /*
680      * This is just a place-holder for now -- used to provide same
           functionality
681      * as the stage model for SetMotorEnable
682      */
683
684     this->Publish(this->_position2d_id, resp_queue,
685         PLAYER_MSGTYPE_RESP_ACK, PLAYER_POSITION2D_REQ_MOTOR_POWER);
686   }
687   else if(Message::MatchMessage(hdr,PLAYER_MSGTYPE_CMD,
688       PLAYER_POSITION2D_CMD_VEL, this->_position2d_id))
689   {
690
691     if(hdr->size != sizeof(player_position2d_cmd_vel_t))
692     {
693       PLAYER_WARN("Arg to command robot velocity is wrong size;
              ignoring");
694       return(-1);
695     }
```

```
696
697     this->SetWheelVelocities((player_position2d_cmd_vel_t *)data);
698
699     return 0;
700   }
701   else if(Message::MatchMessage(hdr,PLAYER_MSGTYPE_REQ,
702       PLAYER_POSITION2D_REQ_SPEED_PID, this->_position2d_id))
703   {
704
705     if(hdr->size != 0)
706     {
707       PLAYER_WARN("Arg to command robot velocity is wrong size;
              ignoring");
708       return(-1);
709     }
710
711     // TODO: set velocity PID
712
713     this->Publish(this->_position2d_id, resp_queue,
714         PLAYER_MSGTYPE_RESP_ACK, PLAYER_POSITION2D_REQ_SPEED_PID);
715
716     return 0;
717   }
718   else if(Message::MatchMessage(hdr,PLAYER_MSGTYPE_REQ,
719       PLAYER_SONAR_REQ_GET_GEOM, this->_sonar_id))
720   {
721
722     if(hdr->size != 0)
723     {
724       PLAYER_WARN("Arg to get sonar geometry is wrong size; ignoring");
725       return(-1);
726     }
727     this->Publish(this->_sonar_id, resp_queue,
728         PLAYER_MSGTYPE_RESP_ACK, PLAYER_SONAR_REQ_GET_GEOM,
729         (void*)&_sonar_geom, sizeof(_sonar_geom), NULL);
730
731     return 0;
732   }
733   else if(Message::MatchMessage(hdr,PLAYER_MSGTYPE_REQ,
734       PLAYER_SONAR_REQ_GET_GEOM, this->_ir_id))
735   {
736
737     if(hdr->size != 0)
738     {
739       PLAYER_WARN("Arg to get IR geometry is wrong size; ignoring");
740       return(-1);
741     }
742     this->Publish(this->_ir_id, resp_queue,
743         PLAYER_MSGTYPE_RESP_ACK, PLAYER_SONAR_REQ_GET_GEOM,
744         (void*)&_ir_geom, sizeof(_ir_geom), NULL);
745
746     return 0;
747   }
748
```

```
749   return 0;
750 }
751
752 void
753 x80::PublishData()
754 {
755   pthread_mutex_lock(&_data.lock);
756
757   /*
758    * TODO: If there is data in the queue we should overwrite it so that
759    * it does not overflow
760    */
761
762
763   // publish position data
764   if(_data.new_pos == 1)
765   {
766     Driver::Publish(this->_position2d_id, PLAYER_MSGTYPE_DATA,
          PLAYER_POSITION2D_DATA_STATE,
767         (void*)&(this->_data.position), sizeof(player_position2d_data_t
              ), NULL);
768     _data.new_pos = 0;
769   }
770
771   // publish sonar data
772   if(_data.new_sonar == 1)
773   {
774     Driver::Publish(this->_sonar_id, PLAYER_MSGTYPE_DATA,
          PLAYER_SONAR_DATA_RANGES,
775         (void*)&(this->_data.sonar), sizeof(this->_data.sonar), NULL);
776
777     _data.new_sonar = 0;
778   }
779
780   // publish ir data
781   if(_data.new_ir == 1)
782   {
783     Driver::Publish(this->_ir_id, PLAYER_MSGTYPE_DATA,
          PLAYER_SONAR_DATA_RANGES,
784         (void*)&(this->_data.ir), sizeof(this->_data.ir), NULL);
785
786     _data.new_ir = 0;
787   }
788
789   pthread_mutex_unlock(&_data.lock);
790 }
791
792 void
793 x80::SetGeometry()
794 {
795   int index = 0;
796
797   // Geometry
798   _robot2d_geom.size.sh      = 0.255;
```

```
799   _robot2d_geom.size.sl      = 0.38;
800   _robot2d_geom.size.sw      = 0.36;
801
802   _robot2d_geom.pose.ppitch    = 0.0;
803   _robot2d_geom.pose.proll   = 0.0;
804   _robot2d_geom.pose.px      = 0.0;
805   _robot2d_geom.pose.py      = 0.0;
806   _robot2d_geom.pose.pyaw    = 0.0;
807   _robot2d_geom.pose.pz      = 0.0;
808
809   // also initialize data
810   pthread_mutex_init(&_data.lock, NULL);
811
812   _data.encoder1PulseLast = -1;
813   _data.encoder2PulseLast = -1;
814   _data.new_pos = 0;
815   _data.new_sonar = 0;
816   _data.new_ir = 0;
817
818   /*
819    * sensor numbering will start at 0 for the front left sensor and
          will
820    * increase clockwise around the vehicle
821    */
822
823   _sonar_geom.poses_count = X80_NUM_SONAR;
824   _data.sonar.ranges_count = _sonar_geom.poses_count;
825   _data.sonar.ranges = new float[_sonar_geom.poses_count];
826   _sonar_geom.poses = new player_pose3d_t[_sonar_geom.poses_count];
827
828   // set to -1 so we can tell when the first values have arrived
829   memset(_data.sonar.ranges, -1, _sonar_geom.poses_count*sizeof(float))
          ;
830
831   // TODO: measure actual angles -- currently calculated using arctan(
          py/px)
832
833   index = 0;  // front left
834   _sonar_geom.poses[index].ppitch = 0.0;
835   _sonar_geom.poses[index].proll = 0.0;
836   _sonar_geom.poses[index].px = 0.14;
837   _sonar_geom.poses[index].py = 0.14;
838   _sonar_geom.poses[index].pyaw = PI / 4.0;
839   _sonar_geom.poses[index].pz = 0.12;
840   _data.sonar.ranges[index] = 0.0;
841
842   index = 1;  // front center
843   _sonar_geom.poses[index].ppitch = 0.0;
844   _sonar_geom.poses[index].proll = 0.0;
845   _sonar_geom.poses[index].px = 0.19;
846   _sonar_geom.poses[index].py = 0.0;
847   _sonar_geom.poses[index].pyaw = 0.0;
848   _sonar_geom.poses[index].pz = 0.12;
849   _data.sonar.ranges[index] = 0.0;
```

```
850
851   index = 2;  // front right
852   _sonar_geom.poses[index].ppitch = 0.0;
853   _sonar_geom.poses[index].proll = 0.0;
854   _sonar_geom.poses[index].px = 0.14;
855   _sonar_geom.poses[index].py = -0.14;
856   _sonar_geom.poses[index].pyaw = - PI / 4.0;
857   _sonar_geom.poses[index].pz = 0.12;
858   _data.sonar.ranges[index] = 0.0;
859
860   index = 3;  // back right
861   _sonar_geom.poses[index].ppitch = 0.0;
862   _sonar_geom.poses[index].proll = 0.0;
863   _sonar_geom.poses[index].px = -0.12;
864   _sonar_geom.poses[index].py = -0.125;
865   _sonar_geom.poses[index].pyaw = - (PI / 2.0) - 0.8058;
866   _sonar_geom.poses[index].pz = 0.16;
867   _data.sonar.ranges[index] = 0.0;
868
869   index = 4;  // back
870   _sonar_geom.poses[index].ppitch = 0.0;
871   _sonar_geom.poses[index].proll = 0.0;
872   _sonar_geom.poses[index].px = -0.18;
873   _sonar_geom.poses[index].py = 0.0;
874   _sonar_geom.poses[index].pyaw = PI;
875   _sonar_geom.poses[index].pz = 0.16;
876   _data.sonar.ranges[index] = 0.0;
877
878   index = 5;  // back left
879   _sonar_geom.poses[index].ppitch = 0.0;
880   _sonar_geom.poses[index].proll = 0.0;
881   _sonar_geom.poses[index].px = -0.12;
882   _sonar_geom.poses[index].py = 0.125;
883   _sonar_geom.poses[index].pyaw = (PI / 2.0) + 0.8058;
884   _sonar_geom.poses[index].pz = 0.16;
885   _data.sonar.ranges[index] = 0.0;
886
887   _ir_geom.poses_count = X80_NUM_IR;
888   _data.ir.ranges_count = _ir_geom.poses_count;
889   _ir_geom.poses = new player_pose3d_t[_ir_geom.poses_count];
890   _data.ir.ranges = new float[_ir_geom.poses_count];
891
892   // set to -1 so we can tell when the first values have arrived
893   memset(_data.ir.ranges, -1, _ir_geom.poses_count*sizeof(float));
894
895   index = 0;  // outer front left
896   _ir_geom.poses[index].ppitch = 0.0;
897   _ir_geom.poses[index].proll = 0.0;
898   _ir_geom.poses[index].px = 0.16;
899   _ir_geom.poses[index].py = 0.10;
900   _ir_geom.poses[index].pyaw = 0.5586;
901   _ir_geom.poses[index].pz = 0.12;
902   _data.ir.ranges[index] = 0.0;
903
```

```
904   index = 1;  // inner front left
905   _ir_geom.poses[index].ppitch = 0.0;
906   _ir_geom.poses[index].proll = 0.0;
907   _ir_geom.poses[index].px = 0.17;
908   _ir_geom.poses[index].py = 0.04;
909   _ir_geom.poses[index].pyaw = 0.2311;
910   _ir_geom.poses[index].pz = 0.12;
911   _data.ir.ranges[index] = 0.0;
912
913   index = 2;  // inner front right
914   _ir_geom.poses[index].ppitch = 0.0;
915   _ir_geom.poses[index].proll = 0.0;
916   _ir_geom.poses[index].px = 0.17;
917   _ir_geom.poses[index].py = -0.04;
918   _ir_geom.poses[index].pyaw = -0.2311;
919   _ir_geom.poses[index].pz = 0.12;
920   _data.ir.ranges[index] = 0.0;
921
922   index = 3;  // outer front right
923   _ir_geom.poses[index].ppitch = 0.0;
924   _ir_geom.poses[index].proll = 0.0;
925   _ir_geom.poses[index].px = 0.16;
926   _ir_geom.poses[index].py = -0.1;
927   _ir_geom.poses[index].pyaw = -0.5586;
928   _ir_geom.poses[index].pz = 0.12;
929   _data.ir.ranges[index] = 0.0;
930
931   index = 4;  // right
932   _ir_geom.poses[index].ppitch = 0.0;
933   _ir_geom.poses[index].proll = 0.0;
934   _ir_geom.poses[index].px = 0.0;
935   _ir_geom.poses[index].py = -0.12;
936   _ir_geom.poses[index].pyaw = -PI / 2.0;
937   _ir_geom.poses[index].pz = 0.205;
938   _data.ir.ranges[index] = 0.0;
939
940   index = 5;  // back
941   _ir_geom.poses[index].ppitch = 0.0;
942   _ir_geom.poses[index].proll = 0.0;
943   _ir_geom.poses[index].px = -0.18;
944   _ir_geom.poses[index].py = 0.0;
945   _ir_geom.poses[index].pyaw = PI;
946   _ir_geom.poses[index].pz = 0.06;
947   _data.ir.ranges[index] = 0.0;
948
949   index = 6;  // left
950   _ir_geom.poses[index].ppitch = 0.0;
951   _ir_geom.poses[index].proll = 0.0;
952   _ir_geom.poses[index].px = 0.0;
953   _ir_geom.poses[index].py = 0.12;
954   _ir_geom.poses[index].pyaw = PI / 2.0;
955   _ir_geom.poses[index].pz = 0.205;
956   _data.ir.ranges[index] = 0.0;
957
```

```
958 }
959
960 void
961 x80::SetWheelVelocities(player_position2d_cmd_vel_t *vcmd)
962 {
963   /*
964    * Take a desired velocity and rotation rate and convert it to wheel
       speeds
965    * Command the platform
966    */
967
968   double vLeft = vcmd->vel.px - X80_MIN_TURN_RADIUS * vcmd->vel.pa;
969   double vRight = vcmd->vel.px + X80_MIN_TURN_RADIUS * vcmd->vel.pa;
970
971   // Sending a -vRight because the encoder directions are switched
972   _driver->SetMotorSpeed(X80_WHEEL_LEFT, vLeft);
973   _driver->SetMotorSpeed(X80_WHEEL_RIGHT, -vRight);
974
975 }
976
977 ////////////////////////////////////////////
978 // Main function for device thread
979 void
980 x80::Main()
981 {
982   // The main loop; interact with the device here
983   for(;;)
984   {
985   // test if we are supposed to cancel
986   pthread_testcancel();
987
988   // Process incoming messages.  x80::ProcessMessage() is
989   // called on each message.
990
991   Lock();
992
993   // ping X80 or else it will stop sending data
994   _driver->PingPMS5005();
995
996   if(InQueue->Empty() == false)
997   {
998     ProcessMessages();
999   }
1000
1001   Unlock();
1002
1003     this->PublishData();
1004
1005     // Sleep (you might, for example, block on a read() instead)
1006     usleep(10000);
1007   }
1008 }
1009
1010 ////////////////////////////////////////
```

```
1011 // Extra stuff for building a shared object.
1012
1013 /* need the extern to avoid C++ name-mangling  */
1014 extern "C" {
1015   int player_driver_init(DriverTable* table)
1016   {
1017     puts("X80  initializing");
1018     x80_Register(table);
1019     puts("X80 Driver done");
1020     return(0);
1021   }
1022 }
```

Appendix G

MATLAB: Single Robot Collision Avoidance Controller Code

G.1 Collision avoidance with common obstacles

G.1.1 Main Function

```
% Clear all windows
clc
clf

% Initial Robot Values
robot_pose = [10, 15, 0];
sensor_radius = 4;
max_vel = 4;
max_ang_vel = 1;

% Map Dimensions
map_max = 30;
map_min = 0;

% Goal Position on the map
goal = [map_max, map_max];
goal_radius = sensor_radius;

% Time settings
t_sim = 100;
T = 0.05;
t_steps = floor(t_sim/T);


% Movie
j = 0;
for t = 1 : t_steps

    % Axis properties
    clf;
    xmax = map_max;
```

```
32      xmin = map_min;
33      ymax = map_max;
34      ymin= 0;
35      axis([xmin,xmax+2*goal_radius,ymin,ymax+2*goal_radius]);
36      axis equal; axis manual;     grid on;     hold on;
37
38      % Plot goal and the circle of interest around the goal
39      scatter(goal(1),goal(2), 100, 'd', 'm', 'filled');
40      rectangle('Position',[goal(1)-goal_radius,...
41          goal(2)-goal_radius,2*goal_radius,2*goal_radius],...
42         'Curvature',[1,1],'EdgeColor','m');
43
44      % Plot the concave or convex obstacle
45      % Check if sensor readings intersect with an obstacle
46      % If an obstacle is in range
47          % Set obst_dist and obst_angle values
48      % If no obstacle is in view
49          % Set obst_dist=0
50      [concave_convex] = concave(robot_pose,sensor_radius,...
51          map_max);
52      obst_dist = concave_convex(1);
53      obst_angle = concave_convex(2);
54
55      % Calculate the new robot pose
56    robot = [robot_pose(1),robot_pose(2),robot_pose(3)];
57    [new_pose] = obstacle_controller(robot,goal,...
58          max_vel,max_ang_vel,goal_radius,sensor_radius,...
59          obst_dist,obst_angle,T);
60    robot_pose = [new_pose(1),new_pose(2),new_pose(3)];
61
62      % Plot the robot and its sensors
63    [circle] = sensor_view(robot_pose,sensor_radius);
64    [XL,YL] = moving_arrows(robot_pose);
65    fill(XL,YL,'r');
66  %{
67      % Save a picture every 10 seconds
68      if ( mod(t,10) == 0 )
69          filename = strcat('pics/',int2str(t),'.png');
70          saveas(gcf,filename)
71      end
72  %}
73      hold off;
74
75      j = j + 1;
76      M(j) = getframe;
77
78  end
79
80
81  % Create a movie and save it as an AVI file
82  movie2avi(M,'video/simulation.avi');
```

G.1.2 Plot obstacle and check robot sensors detect the obstacle

```
function [concave_convex] = concave(robot_pose,...
    sensor_radius,map_max)

% Obstacle Parameters
obst_radius = sensor_radius;
x_dist = map_max/2;
y_dist = map_max/2;
ob_width = 5;
ob_height = 10;
ob_thickness = 2.5;

% Concave obstacle dimensions
%xlimit = [x_dist,x_dist+ob_width,...
%    x_dist+ob_width-ob_thickness];
%ylimit = [y_dist, y_dist+ob_height,...
%    y_dist+ob_height-ob_thickness,y_dist+ob_thickness];
%concaveX = xlimit([1 2 2 1 1 3 3 1]);
%concaveY = ylimit([1 1 2 2 3 3 4 4]);

% Convex obstacle dimensions
xlimit = [x_dist,x_dist+ob_thickness,...
    x_dist+ob_width+ob_thickness,...
    x_dist+ob_width+2*ob_thickness];
ylimit = [y_dist, y_dist-ob_height/2, y_dist+ob_height/2];
concaveX = xlimit([1 2 3 4 3 2 1]);
concaveY = ylimit([1 2 2 1 3 3 1]);

% Draw obstacles in the simulation
% Obstacles are colored black -> k
fill(concaveX, concaveY, 'k');

% Check to see if the robot sensors intersect with an obstacle
obst_dist = 0;
obst_angle = 0;
for beam_angle = (robot_pose(3)-pi/2) : pi/12 : ...
        (robot_pose(3)+pi/2);

    x_ob = [robot_pose(1),...
        robot_pose(1)+obst_radius*cos(beam_angle)];
    y_ob = [robot_pose(2),...
        robot_pose(2)+obst_radius*sin(beam_angle)];
    [xi, yi] = polyxpoly(x_ob, y_ob, concaveX, concaveY);

    % Plot a red circle where the intersection points occur
    mapshow(xi,yi,'DisplayType','point','Marker','o');

    dist = sqrt(xi.^2 + yi.^2);
    if (size(dist,1) == 2)
        dist = dist(1);
    end
```

```
51
52      if (isempty(dist) == false)
53          obst_dist = dist;
54          obst_angle = beam_angle;
55          if (obst_angle > 2*pi)
56              obst_angle = obst_angle - 2*pi;
57          elseif (obst_angle < 0)
58              obst_angle = obst_angle + 2*pi;
59          end
60      end
61  end
62
63  % Return obst_dist and obst_angle values to the main
64  concave_convex = [obst_dist, obst_angle];
```

G.1.3 Apply collision avoidance controller

```
function [new_position] = obstacle_controller(robot,goal,...
    max_vel,max_ang_vel,goal_radius,obst_radius,...
    obst_dist,obst_angle,T)

% Constants
k_a = 2;
k_n = 0.5;
k_t = 0.5;

% Calculate the x and y components of the distance to goal
delta_x = goal(1) - robot(1);
delta_y = goal(2) - robot(2);

% Calculate the distance to the goal
dist_goal = sqrt( delta_x^2 + delta_y^2 );

% Calculate angle to goal
theta = atan(delta_y/delta_x);

% Calculate the change in the angle between the current and goal
    position
delta_theta = theta - robot(3);

% Calculate the distance relation
dist_relation = exp(-dist_goal/goal_radius);

% Calculate attractive velocity
u_a = k_a * max_vel * (1-dist_relation);

% Find new theta value
close_to_zero = 0.2;
if ( abs(delta_theta) < close_to_zero)
    theta = theta;
elseif ( delta_theta >= 0 && abs(delta_theta) < pi)
    theta = robot(3) + max_ang_vel*T;
elseif ( delta_theta >= 0 && abs(delta_theta) >= pi)
    theta = robot(3) - max_ang_vel*T;
elseif ( delta_theta < 0 && abs(delta_theta) < pi)
    theta = robot(3) - max_ang_vel*T;
elseif ( delta_theta < 0 && abs(delta_theta) >= pi)
    theta = robot(3) + max_ang_vel*T;
end

% Calculate repulsive velocity
% If no obstacle is in view
    % set normal and tangent values to zero
% If an obstacle is in view
    % calculate normal and tangent values
if (obst_dist == 0)
    normal = 0;
    tangent = 0;
    obst_relation = 0;
```

```
52      normal_angle = 0;
53      tangent_angle = 0;
54      theta_obst = 0;
55
56  else
57      normal = 1/obst_dist;
58      tangent = 1/obst_dist;
59      obst_relation = exp(-obst_dist/obst_radius);
60
61      % Calculate normal angle
62      normal_angle = obst_angle + pi;
63      if (normal_angle > 3*pi/2)
64        normal_angle = normal_angle - 2*pi;
65      end
66
67      % Calculate tangent angle
68      tangent_angle = normal_angle - pi/2;
69      if (tangent_angle > 3*pi/2)
70        tangent_angle = tangent_angle - 2*pi;
71      end
72
73      % Calculate obstacle angle
74      theta_obst = (normal_angle + tangent_angle) / 2;
75      if (theta_obst < 0)
76          theta_obst = theta_obst + 2*pi;
77      end
78
79      % Calculate change in angle between the robot and obstacle
80      delta_theta_obst = theta_obst - robot(3);
81      if ( abs(delta_theta_obst) < close_to_zero )
82          theta_obst = theta_obst;
83      elseif ( delta_theta_obst >= 0 && abs(delta_theta_obst) < pi)
84          theta_obst = robot(3) + max_ang_vel*T;
85      elseif ( delta_theta_obst >= 0 && abs(delta_theta_obst) >= pi)
86          theta_obst = robot(3) - max_ang_vel*T;
87      elseif ( delta_theta_obst < 0 && abs(delta_theta_obst) < pi)
88          theta_obst = robot(3) - max_ang_vel*T;
89      elseif ( delta_theta_obst < 0 && abs(delta_theta_obst) >= pi)
90          theta_obst = robot(3) + max_ang_vel*T;
91      end
92
93      % Ignore goal while obstacle is in view
94      u_a = 0;
95      theta = 0;
96  end
97
98  % Calculate normal and tangential velocity components
99  u_n = k_n * normal * obst_relation;
100 u_t = k_t * tangent * obst_relation;
101
102 % Calculate the sum of all velocity components
103 u_totX = u_a*cos(theta) + u_n*cos(normal_angle) ...
104     + u_t*cos(tangent_angle);
105 u_totY = u_a*sin(theta) + u_n*sin(normal_angle) ...
```

```
106     + u_t*sin(tangent_angle);
107
108 % Calculate the change in x and y distance for the current time step
109 delta_d = [u_totX*T, u_totY*T];
110 delta_theta = theta + theta_obst;
111
112 % Send the robot's next position to the main function
113 new_position = [robot(1)+delta_d(1), robot(2)+delta_d(2) ,...
114     delta_theta, u_totX, u_totY];
```

G.1.4 Arrow geometry

```
function [X,Y] = moving_arrows(Q)
x     = Q(1);
y     = Q(2);
theta = Q(3);

l = 1;
X = [x,x+l*cos(theta-2*pi/3),x+l*cos(theta),...
     x+l*cos(theta+2*pi/3),x];
Y = [y,y+l*sin(theta-2*pi/3),y+l*sin(theta),...
     y+l*sin(theta+2*pi/3),y];
```

G.1.5 Senor geometry around the robot

```
function [circle] = sensor_view(Q,radius)
x     = Q(1);
y     = Q(2);
theta = Q(3);

% Plot beam
for beam_angle = (theta - pi/2) : pi/12 : (theta + pi/2);
    beamX = [x, x + radius*cos(beam_angle)];
    beamY = [y, y + radius*sin(beam_angle)];
    plot(beamX,beamY,'c');
end

% Plot circle
circle = rectangle('Position',...
    [x-radius,y-radius,2*radius,2*radius],...
    'Curvature',[1,1],'EdgeColor','c');
```

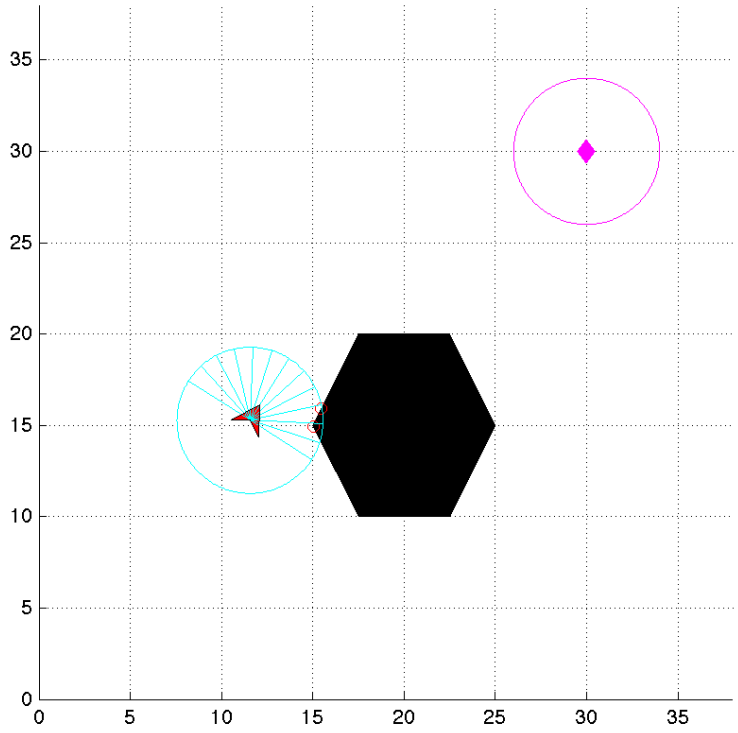


Figure G.1: MATLAB figure for a single robot obstacle avoidance simulation

G.2 Collision avoidance through narrow passages

G.2.1 Main Function

```
% Clear all windows
clc
clf

% Initial Robot Values
robot_pose = [10, 15, 0];
sensor_radius = 4;
max_vel = 4;
max_ang_vel = 1;

% Map Dimensions
map_max = 30;
map_min = 0;

% Goal Position on the map
goal = [map_max,map_max];
goal_radius = sensor_radius;

% Time settings
t_sim = 100;
T = 0.05;
t_steps = floor(t_sim/T);


% Movie
j = 0;
for t = 1 : t_steps

    % Set Axis properties
    clf;
    xmax = map_max;
    xmin = map_min;
    ymax = map_max;
    ymin= 0;
    axis([xmin, xmax+2*goal_radius, ymin, ymax+2*goal_radius]);
    axis equal; axis manual;    grid on;    hold on;

    % Plot goal and the circle of interest around the goal
    scatter(goal(1),goal(2), 100, 'd', 'm', 'filled');
    rectangle('Position',[goal(1)-goal_radius,goal(2)-goal_radius,2*
        goal_radius,2*goal_radius], 'Curvature',[1,1],'EdgeColor','m');

    % Plot the narrow passage
    % Check if sensor readings intersect with an obstacle
    % If an obstacle is in range, set obst_dist and obst_angle values
    % If no obstacle is in view, obst_dist=0
    [narrow_opening] = narrow_passage(robot_pose, sensor_radius,
        map_max);
    obst_dist = narrow_opening(1);
```

```
48      obst_angle = narrow_opening(2);
49
50      % Calculate the new robot pose
51    robot = [robot_pose(1), robot_pose(2), robot_pose(3)];
52    [new_pose] = obstacle_controller(robot, goal, max_vel, max_ang_vel,
          goal_radius, obst_radius, obst_dist, obst_angle, T);
53    robot_pose = [new_pose(1), new_pose(2), new_pose(3)];
54
55      % Plot the robot and its sensors
56    [circle] = sensor_view(robot_pose ,sensor_radius);
57    [XL,YL] = moving_arrows(robot_pose);
58    fill(XL,YL,'r');
59
60      % Save a picture every 10 seconds
61      if ( mod(t,10) == 0 )
62          filename = strcat( 'pics/', int2str(t), '.png');
63          saveas(gcf,filename)
64      end
65
66      hold off;
67
68      j = j + 1;
69      M(j) = getframe;
70
71  end
72
73
74  % Save the simulation as an AVI file
75  movie2avi(M,'video/simulation.avi');
```

G.2.2 Plot obstacle and check robot sensors detect the obstacle

```
function [narrow_opening] = narrow_passage(leader_robot, sensor_radius,
    map_max)

% Obstacle Parameters
obst_opening = 10;
obst_radius = sensor_radius;
ob_width = 10;
ob_height = map_max/2;
x_dist = 0;
y_dist = 0.6*map_max;

% Set the dimensions of the obstacle
xlimit = [x_dist, x_dist+ob_width, x_dist+2*ob_width, x_dist+3*ob_width
    ];
ylimit = [y_dist-obst_opening/2-ob_height, y_dist-obst_opening/2,
    y_dist+obst_opening/2+ob_height, y_dist+obst_opening/2];
ob_bottomX = xlimit([1 2 3 4]);
ob_bottomY = ylimit([1 2 2 1]);
ob_topX = ob_bottomX;
ob_topY = ylimit([3 4 4 3]);

% Draw obstacles in the simulation
% Obstacles are colored black -> k
fill(ob_bottomX, ob_bottomY, 'k');
fill(ob_topX, ob_topY, 'k');

% Check to see if the robot sensors intersect with an obstacle
obst_dist = 0;
obst_angle = 0;
passage_dist = 0;
passage_distT = 0;  passage_distB = 0;

for beam_angle = (leader_robot(3)-pi/2) : pi/12 : (leader_robot(3)+pi
    /2);

    x_ob = [leader_robot(1), leader_robot(1)+obst_radius*cos(beam_angle
        )];
    y_ob = [leader_robot(2), leader_robot(2)+obst_radius*sin(beam_angle
        )];
    [xT, yT] = polyxpoly(x_ob, y_ob, ob_topX, ob_topY);
    [xB, yB] = polyxpoly(x_ob, y_ob, ob_bottomX, ob_bottomY);
    mapshow(xT,yT,'DisplayType','point','Marker','o');
    mapshow(xB,yB,'DisplayType','point','Marker','o');

    obst_distT = sqrt(xT.^2 + yT.^2);
    obst_distB = sqrt(xB.^2 + yB.^2);

    if (size(obst_distT,1) == 2)
        obst_distT = obst_distT(1);
    elseif (size(obst_distB,1) == 2)
```

```
45          obst_distB = obst_distB(1);
46      end
47
48      if (isempty(obst_distT) == false && isempty(obst_distB) == true)
49          obst_dist = obst_distT;
50      elseif (isempty(obst_distT) == true && isempty(obst_distB) == false
            )
51        obst_dist = obst_distB;
52      end
53
54      if (isempty(obst_distT) == false || isempty(obst_distB) == false)
55          obst_angle = beam_angle;
56          if (obst_angle >= 2*pi)
57              obst_angle = obst_angle - 2*pi;
58          elseif (obst_angle < 0)
59              obst_angle = obst_angle + 2*pi;
60          end
61      end
62
63      % Calculate the passage distance
64      x_pass = [leader_robot(1), leader_robot(1)+sensor_radius*cos(
            beam_angle)];
65      y_pass = [leader_robot(2), leader_robot(2)+sensor_radius*sin(
            beam_angle)];
66      [xT_pass, yT_pass] = polyxpoly(x_pass, y_pass, ob_topX, ob_topY);
67      [xB_pass, yB_pass] = polyxpoly(x_pass, y_pass, ob_bottomX,
            ob_bottomY);
68
69      % Plot a red circle where the intersection points occur
70      mapshow(xT_pass,yT_pass,'DisplayType','point','Marker','o');
71      mapshow(xB_pass,yB_pass,'DisplayType','point','Marker','o');
72
73      if (size(yT_pass,1) == 2)
74          yT_pass = yT_pass(1);
75      elseif (size(yB_pass,1) == 2)
76          yB_pass = yB_pass(1);
77      end
78
79
80      if (isempty(yT_pass) == false && isempty(yB_pass) == true)
81          passage_distT = yT_pass;
82      elseif (isempty(yT_pass) == true && isempty(yB_pass) == false)
83          passage_distB = yB_pass;
84      end
85
86      % Calculate passage distance
87      passage_dist = passage_distT - passage_distB;
88
89  end
90
91  % Return obst_dist, obst_angle, and passage_dist values to the main
        function
92  narrow_opening = [obst_dist, obst_angle, passage_dist];
```

G.2.3 Apply collision avoidance controller

```
function [new_position] = obstacle_controller(robot,goal,...
    max_vel,max_ang_vel,goal_radius,obst_radius,...
    obst_dist,obst_angle,T)

% Constants
k_a = 2;
k_n = 0.5;
k_t = 0.5;

% Calculate the x and y components of the distance to goal
delta_x = goal(1) - robot(1);
delta_y = goal(2) - robot(2);

% Calculate the distance to the goal
dist_goal = sqrt( delta_x^2 + delta_y^2 );

% Calculate angle to goal
theta = atan(delta_y/delta_x);

% Calculate the change in the angle between the current and goal
    position
delta_theta = theta - robot(3);

% Calculate the distance relation
dist_relation = exp(-dist_goal/goal_radius);

% Calculate attractive velocity
u_a = k_a * max_vel * (1-dist_relation);

% Find new theta value
close_to_zero = 0.2;
if ( abs(delta_theta) < close_to_zero)
    theta = theta;
elseif ( delta_theta >= 0 && abs(delta_theta) < pi)
    theta = robot(3) + max_ang_vel*T;
elseif ( delta_theta >= 0 && abs(delta_theta) >= pi)
    theta = robot(3) - max_ang_vel*T;
elseif ( delta_theta < 0 && abs(delta_theta) < pi)
    theta = robot(3) - max_ang_vel*T;
elseif ( delta_theta < 0 && abs(delta_theta) >= pi)
    theta = robot(3) + max_ang_vel*T;
end

% Calculate repulsive velocity
% If no obstacle is in view
    % set normal and tangent values to zero
% If an obstacle is in view
    % calculate normal and tangent values
if (obst_dist == 0)
    normal = 0;
    tangent = 0;
    obst_relation = 0;
```

```
52      normal_angle = 0;
53      tangent_angle = 0;
54      theta_obst = 0;
55
56  else
57      normal = 1/obst_dist;
58      tangent = 1/obst_dist;
59      obst_relation = exp(-obst_dist/obst_radius);
60
61      % Calculate normal angle
62      normal_angle = obst_angle + pi;
63      if (normal_angle > 3*pi/2)
64        normal_angle = normal_angle - 2*pi;
65      end
66
67      % Calculate tangent angle
68      tangent_angle = normal_angle - pi/2;
69      if (tangent_angle > 3*pi/2)
70        tangent_angle = tangent_angle - 2*pi;
71      end
72
73      % Calculate obstacle angle
74      theta_obst = (normal_angle + tangent_angle) / 2;
75      if (theta_obst < 0)
76          theta_obst = theta_obst + 2*pi;
77      end
78
79      % Calculate change in angle between the robot and obstacle
80      delta_theta_obst = theta_obst - robot(3);
81      if ( abs(delta_theta_obst) < close_to_zero )
82          theta_obst = theta_obst;
83      elseif ( delta_theta_obst >= 0 && abs(delta_theta_obst) < pi)
84          theta_obst = robot(3) + max_ang_vel*T;
85      elseif ( delta_theta_obst >= 0 && abs(delta_theta_obst) >= pi)
86          theta_obst = robot(3) - max_ang_vel*T;
87      elseif ( delta_theta_obst < 0 && abs(delta_theta_obst) < pi)
88          theta_obst = robot(3) - max_ang_vel*T;
89      elseif ( delta_theta_obst < 0 && abs(delta_theta_obst) >= pi)
90          theta_obst = robot(3) + max_ang_vel*T;
91      end
92
93      % Ignore goal while obstacle is in view
94      u_a = 0;
95      theta = 0;
96  end
97
98  % Calculate normal and tangential velocity components
99  u_n = k_n * normal * obst_relation;
100 u_t = k_t * tangent * obst_relation;
101
102 % Calculate the sum of all velocity components
103 u_totX = u_a*cos(theta) + u_n*cos(normal_angle) ...
104     + u_t*cos(tangent_angle);
105 u_totY = u_a*sin(theta) + u_n*sin(normal_angle) ...
```

```
106     + u_t*sin(tangent_angle);
107
108 % Calculate the change in x and y distance for the current time step
109 delta_d = [u_totX*T, u_totY*T];
110 delta_theta = theta + theta_obst;
111
112 % Send the robot's next position to the main function
113 new_position = [robot(1)+delta_d(1), robot(2)+delta_d(2) ,...
114     delta_theta, u_totX, u_totY];
```

G.2.4 Arrow geometry

```
function [X,Y] = moving_arrows(Q)
x     = Q(1);
y     = Q(2);
theta = Q(3);

l = 1;
X = [x,x+l*cos(theta-2*pi/3),x+l*cos(theta),...
     x+l*cos(theta+2*pi/3),x];
Y = [y,y+l*sin(theta-2*pi/3),y+l*sin(theta),...
     y+l*sin(theta+2*pi/3),y];
```

G.2.5 Senor geometry around the robot

```
function [circle] = sensor_view(Q,radius)
x      = Q(1);
y      = Q(2);
theta = Q(3);

% Plot beam
for beam_angle = (theta - pi/2) : pi/12 : (theta + pi/2);
    beamX = [x, x + radius*cos(beam_angle)];
    beamY = [y, y + radius*sin(beam_angle)];
    plot(beamX,beamY,'c');
end

% Plot circle
circle = rectangle('Position',...
    [x-radius,y-radius,2*radius,2*radius],...
    'Curvature',[1,1],'EdgeColor','c');
```

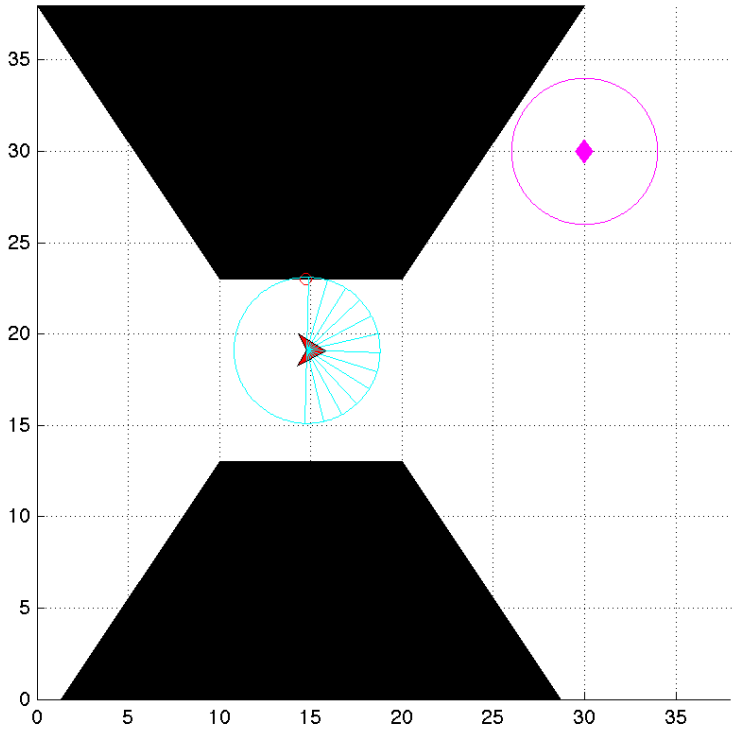


Figure G.2: MATLAB figure for a single robot obstacle avoidance simulation through a narrow passage

Appendix H

PLAYER: Single Robot Collision Avoidance Controller Code

H.1 Robot Controller

```
1 //============================================================
2 // Name        : hydro1.cpp
3 // Author      :
4 // Version     :
5 // Copyright   :
6 // Description :
7 //============================================================
8
9 #include <iostream>
10 #include <cmath>
11 #include <csignal>
12 #include "libplayerc++/playerc++.h"
13
14 using namespace std;
15 using namespace PlayerCc;
16
17
18 // ------------------------------------------------------------
19
20
21 // FUNCTION PROTOTYPES
22
23 volatile sig_atomic_t sigShutdown;
24 void UnexpectedShutdown (void);
25 void HandleShutdownSignal(int);
26
27 void CheckObstacle(double&, double&, int&, SonarProxy&, SonarProxy&,
       double);
28 double Velocity (double, double, double, double, double, double, int);
29 double AngularVelocity(double, double, double, double, double, double,
       double, int);
30
31
```

```
32 // ------------------------------------------------------------
33
34
35 // GLOBAL VARIABLES
36
37 double xGoal = 4, yGoal = 0, goalRadius = 1, obstRadius = 1;
38 double vmax = 0.4, omega_max = 0.5, t = 1;
39 double const PI = 3.14159, close_to_zero = 0.1;
40 double k_a = 0.8, k_n = 1, k_t = 1, k_o = 1;
41
42 double obstDist1, obstAngle1;
43 double xR1, yR1, thetaR1;
44 double vR1, omegaR1;
45 int turnDirection1 = 1;
46
47 bool updateSim = false;
48
49 // ------------------------------------------------------------
50
51
52 // MAIN FUNCTION
53
54 int main()
55 {
56   try
57   {
58 /*
59     // Get final pose from user
60     cout << "Robot goal x: "<< endl;
61     cin >> xGoal;
62     cout << "Robot goal y: " << endl;
63     cin >> yGoal;
64 */
65     cout << endl << endl << "executing new program...." << endl;
66
67     // Call the player client, name the vehicle, and include any
       neccesary proxies
68     PlayerClient robot1("localhost", 6665);
69     Position2dProxy pose1(&robot1, 0);
70     SonarProxy  sonar1(&robot1, 0);
71     SonarProxy  ir1(&robot1, 1);
72     char robotName[] = "leader";
73
74     robot1.SetReplaceRule(true, PLAYER_MSGTYPE_DATA, -1, -1);
75
76     SimulationProxy *sim = NULL;
77     if(updateSim == true)
78     {
79       sim = new SimulationProxy(&robot1, 0);
80     }
81
82     // Handle unexpected shutdown
83     UnexpectedShutdown();
84
```

```
85      while (sigShutdown == 0)
86      {
87        // read from the proxies
88        robot1.Read();
89        pose1.RequestGeom();
90        sonar1.RequestGeom();
91        ir1.RequestGeom();
92
93        // get the x, y, and theta position
94        xR1 = pose1.GetXPos();
95        yR1 = pose1.GetYPos();
96        thetaR1 = pose1.GetYaw();
97
98        // if the robot has reached the goal then stop
99        if ( ((xGoal-xR1) <= close_to_zero && (yGoal-yR1) <=
              close_to_zero) )
100       {
101         pose1.SetSpeed(0, 0);
102         break;
103       }
104
105       // See if any obstacles are in view, if so find its distance and
              angle
106       CheckObstacle(obstDist1, obstAngle1, turnDirection1, sonar1, ir1,
               thetaR1);
107
108       // Calculate the velocity commands
109       vR1 = Velocity(xR1, yR1, xGoal, yGoal, obstDist1, obstAngle1,
              turnDirection1);
110       omegaR1 = AngularVelocity(xR1, yR1, thetaR1, xGoal, yGoal,
              obstDist1, obstAngle1, turnDirection1);
111
112       // set the speed of the robot
113       pose1.SetSpeed(vR1, omegaR1);
114       //pose1.SetSpeed(0, 0);
115
116       if(updateSim == true)
117       {
118         sim->SetPose2d(robotName, xR1, yR1, thetaR1);
119       }
120     }
121
122     // stop the vehicle by using a keyboard stroke
123     pose1.SetSpeed(0, 0);
124     cout << "robot stopped" <<endl;
125
126     if(updateSim == true)
127       delete sim;
128
129     sleep(100000);
130   }
131
132   catch (PlayerCc::PlayerError e)
133   {
```

```
134     std::cerr << e << std::endl;
135     return -1;
136   }
137
138   return 0;
139 }   // end of main
140
141
142 // ------------------------------------------------------
143
144
145 // FUNCTIONS
146
147 void CheckObstacle(double &obstDist, double &obstAngle, int &
      turnDirection, SonarProxy &sonar, SonarProxy &ir, double theta)
148 {
149   double minDist = obstRadius, minAngle;
150   int sonarMin = 0, irMin = 0;
151
152   // sonar sensors
153   for(int i=0; i<=5; i++)
154   {
155     if(sonar[i] < minDist && sonar[i] > 0)
156     {
157       minDist = sonar[i];
158       sonarMin = i+1;
159       //cout << "sonar reading: " << i+1 << endl;
160     }
161   }
162
163   // ir sensors
164   for(int j=0; j<=6; j++)
165   {
166     if(ir[j] < minDist && ir[j] > 0)
167     {
168       minDist = ir[j];
169       irMin = j+1;
170       sonarMin = 0;
171       //cout << "ir reading: " << j+1 << endl;
172     }
173   }
174
175   //get the min angle
176   if (sonarMin > 0)
177   {
178     switch (sonarMin)
179     {
180       case 1:
181       {
182         minAngle = theta - dtor(45);
183         turnDirection = -1;
184         break;
185       }
186       case 2:
```

```
187         {
188           minAngle = theta;
189           turnDirection = 1;
190           //minDist = obstRadius;
191           break;
192         }
193         case 3:
194         {
195           minAngle = theta - dtor(45);
196           turnDirection = 1;
197           break;
198         }
199         case 4:
200         {
201           minAngle = theta - dtor(135);
202           turnDirection = 1;
203           break;
204         }
205         case 5:
206         {
207           //minAngle = theta - dtor(180);
208           //turnDirection = 1;
209           minDist = obstRadius;
210           break;
211         }
212         case 6:
213         {
214           minAngle = theta - dtor(135);
215           turnDirection = -1;
216           break;
217         }
218       }
219       //cout << "Sonar Angle: " << minAngle << endl;
220     }
221     else if (irMin > 0)
222     {
223       switch (irMin)
224       {
225         case 1:
226         {
227           minAngle = theta - dtor(40);
228           turnDirection = -1;
229           break;
230         }
231         case 2:
232         {
233           minAngle = theta - dtor(10);
234           turnDirection = -1;
235           break;
236         }
237         case 3:
238         {
239           minAngle = theta - dtor(10);
240           turnDirection = 1;
```

```
241         break;
242       }
243       case 4:
244       {
245         minAngle = theta - dtor(40);
246         turnDirection = 1;
247         break;
248       }
249       case 5:
250       {
251         minAngle = theta - dtor(90);
252         turnDirection = 1;
253         break;
254       }
255       case 6:
256       {
257         //minAngle = theta - dtor(180);
258         //turnDirection = 1;
259         minDist = obstRadius;
260         break;
261       }
262       case 7:
263       {
264         minAngle = theta - dtor(90);
265         turnDirection = -1;
266         break;
267       }
268     }
269     //cout << "IR Angle: " << rtod(minAngle) << endl;
270   }
271
272   obstDist = 0;
273   if (minDist < obstRadius && minDist > 0)
274   {
275     obstDist = minDist;
276     obstAngle = minAngle;
277     if(obstAngle < 0)
278       obstAngle = obstAngle + 2*PI;
279   }
280 }
281
282 double Velocity (double x, double y, double goal_x, double goal_y,
      double obstDist, double obstAngle, int turnDirection)
283 {
284   // distance to goal
285   double delta_x = goal_x - x;
286   double delta_y = goal_y - y;
287   double distGoal = sqrt( delta_x*delta_x + delta_y*delta_y );
288   double thetaCalc = atan(delta_y/delta_x);
289   double distRelation = exp(-distGoal/goalRadius);
290
291   // attractive velocity
292   double u_a = k_a * vmax * (1-distRelation);
293
```

```
294   // repulsive velocity
295   double normal, tangent, obstRelation, normalAngle, tangentAngle;
296   if (obstDist == 0)
297   {
298     normal = 0, tangent = 0;
299     obstRelation = 0;
300     normalAngle = 0, tangentAngle = 0;
301   }
302
303   else
304   {
305     normal = 1/(obstDist), tangent = 1/(obstDist);
306     obstRelation = exp(-obstDist/obstRadius);
307
308     //u_a = 0;
309
310     // normal angle
311     normalAngle = obstAngle + PI;
312     if (normalAngle > 2*PI)
313       normalAngle = normalAngle - 2*PI;
314
315     // tangent angle
316     tangentAngle = normalAngle - (PI/2 *turnDirection);
317   }
318
319   // normal and tangent velocities
320   double u_n = k_n * normal * obstRelation;
321   double u_t = k_t * tangent * obstRelation;
322
323   // sum of velocity terms
324   double v, vx, vy;
325   vx = u_a*cos(thetaCalc) + u_n*cos(normalAngle) + u_t*cos(tangentAngle
        );
326   vy = u_a*sin(thetaCalc) + u_n*sin(normalAngle) + u_t*sin(tangentAngle
        );
327   v = sqrt(vx*vx + vy*vy);
328   if (v > vmax)
329     v = vmax;
330   else if (v < -vmax)
331     v = -vmax;
332
333   return (v);
334 }
335
336 double AngularVelocity(double x, double y, double theta, double goal_x,
       double goal_y, double obstDist, double obstAngle, int
      turnDirection)
337 {
338   double omega_a, omega_o = 0, omega;
339
340   // distance to goal
341   double delta_x = goal_x - x;
342   double delta_y = goal_y - y;
343   double thetaCalc = atan(delta_y/delta_x);
```

```
344   double delta_theta = thetaCalc - theta;
345
346   // attractive angular velocity
347   omega_a = k_a * delta_theta/t;
348   if (omega_a >= omega_max)
349     omega_a = omega_max;
350   else if (omega_a < - omega_max)
351     omega_a = - omega_max;
352
353   // repulsive velocity
354   double normalAngle, tangentAngle;
355   if (obstDist == 0)
356     normalAngle = 0, tangentAngle = 0;
357
358   else
359   {
360     // normal angle
361     normalAngle = obstAngle + PI;
362     if (normalAngle > 2*PI)
363       normalAngle = normalAngle - 2*PI;
364
365     // tangent angle
366     tangentAngle = normalAngle - (PI/2 * turnDirection);
367
368     // get the resultant angle
369     double thetaObst = (normalAngle + tangentAngle) / 2;
370     if (thetaObst > 2*PI)
371       thetaObst = thetaObst - 2*PI;
372     double delta_thetaObst = (thetaObst - theta) * turnDirection;
373
374     // repulsive velocity
375     omega_o = k_o * delta_thetaObst/t;
376     if (omega_o >= omega_max)
377       omega_o = omega_max;
378     else if (omega_o < - omega_max)
379       omega_o = - omega_max;
380
381     omega_a = 0;
382
383   }
384
385   omega = omega_a + omega_o;
386
387   return (omega);
388 }
389
390 void UnexpectedShutdown (void)
391 {
392   sigShutdown = 0;
393   sigset_t blockmask;
394   struct sigaction sa;
395   sigfillset(&blockmask);
396   sa.sa_handler = HandleShutdownSignal;
397   sa.sa_mask = blockmask;
```

```
398   sa.sa_flags = 0;
399   sigaction(SIGINT, &sa, NULL);
400   sigaction(SIGQUIT, &sa, NULL);
401 }
402
403 void HandleShutdownSignal (int sig __attribute__ ((unused)))
404 {
405    sigShutdown = 1;
406 }
```

Appendix I

MATLAB: Geometric Formation Controller Code

I.1 Leader-Follower Simulation with 2 Robots

I.1.1 Main Function

```
% ------------------------------------------------------------

% INITIAL CONDITIONS

% Clear windows
clc
clf

% Simulation time
tsim = 80;

% Leader initial position and trajectory velocity
xL_init = [0,0,0];
R = 10;
gamma = 0.1;
vL = gamma*R;
wL = gamma;

% Follower initial position
xF_init = [-5,-5,pi];

% ------------------------------------------------------------

% LEADER AND FOLLOWER POSITION CALCULATIONS

% Convert to cartesian coordinates to polar coordinates
rho = sqrt((xL_init(2)-xF_init(2))^2 ...
    + (xL_init(1)-xF_init(1))^2);
alpha = atan((xL_init(2)-xF_init(2)) ...
    / (xL_init(1)-xF_init(1))) - xF_init(3);
psi = alpha + xF_init(3) - xL_init(3) + pi;
```

```
32
33 % Create a vector with the three polar coordinates
34 polar_init = [rho, alpha, psi];
35
36 % Use the ordinary differential equation function
37 % to determine the position of the follower robot,
38 % based on the follower's velocity calculated in the
39 % function follower_sim.m
40 options = odeset('RelTol',1e-3);
41 [tF,pF] = ode45('follower_controller',tsim,polar_init,...
42     options,vL,wL);
43
44 % Calculate the leader positions for the entire time interval
45 xL = zeros(size(tF),3);
46 for i = 1:size(tF)
47     xL(i,1) = R*sin(gamma*tF(i));
48     xL(i,2) = R*(1-cos(gamma*tF(i)));
49     xL(i,3) = gamma*tF(i);
50 end
51
52 % Create a matrix for the follower position for the entire
53 % time interval. Set the first row of the matrix to the
54 % follower initial position
55 xF = zeros(size(tF),3);
56 xF(1,1) = xF_init(1);
57 xF(1,2) = xF_init(2);
58 xF(1,3) = xF_init(3);
59
60 % Calculate the remaining follower positions for the entire
61 % time interval. convert polar position coordinates to
62 % cartesian position coordinates
63 for i = 2:size(tF)
64     thetaF = pF(i,3) - pF(i,2) + xL(i,3) - pi;
65
66     xF(i,1) = xL(i,1) - pF(i,1)*cos(pF(i,2)+thetaF);
67     xF(i,2) = xL(i,2) - pF(i,1)*sin(pF(i,2)+thetaF);
68     xF(i,3) = thetaF;
69 end
70
71 % ------------------------------------------------------------
72
73 % CREATE A SIMULATION ANIMATION
74
75 figure;
76 clf reset;
77 xmax = max(xL(:,1));      xmin = min(xL(:,1));
78 ymax = max(xL(:,2));      ymin = min(xL(:,2));
79
80 j = 0;
81 for i = 1:size(tF)
82
83     % Clear figure and set axes
84     clf;
85     axis([xmin-10 xmax+10 ymin-10 ymax+10]);
```

```
86      axis equal;
87      axis manual;
88
89      % Plot the leader with a red arrow and a red trail
90      [X,Y] = moving_arrows(xL(i,:),axis(gca));
91      hold on;
92      plot(xL(1:i,1), xL(1:i,2), 'r');
93      fill(X,Y,'r');
94      hold off;
95
96      % Plot the follower moving with a blue arrow and a
97      % blue trail
98      [X,Y] = moving_arrows(xF(i,:),axis(gca));
99      hold on;
100     plot(xF(1:i,1), xF(1:i,2), 'b');
101     fill(X,Y,'b');
102     hold off;
103
104     % Set the labels for the figure
105     xlabel('x [m]');
106     ylabel('y [m]');
107
108     % Save a picture every 10 seconds
109     if ( mod(t,10) == 0 )
110         filename = strcat( 'pics/', int2str(t), '.png');
111         saveas(gcf,filename)
112     end
113
114     j = j + 1;
115     M(j) = getframe(gcf);
116
117 end
118
119 % Create an avi movie file
120 movie2avi(M,'leader_follower.avi');
```

I.1.2 Follower Controller Function

```
function [ dx ] = follower_controller( t,p,flag,vL,wL )

% Initialize the return value dx
dx = zeros(3,1);

% Set the desired distance and angle values
rho_d = 1;
alpha_d = 0;

% Proportional control constant K
K = 1;

% Create the error vector, and apply controller
e = zeros(2,1);
e(1) = K*(rho_d - p(1));
e(2) = K*(alpha_d - p(2));

% Calculate follower velocity and angular velocity according to the
    control
% algorithm
vF = e(1)*(-1/cos(p(2))) + vL*(cos(p(3))/cos(p(2)));
wF = e(1)*(-tan(p(2))/p(1)) - e(2) + vL*(tan(p(2))*cos(p(3))/p(1) + sin
    (p(3))/p(1));

% Calculate the output return dx values
dx(1) = e(1);
dx(2) = e(2);
dx(3) = vF*sin(p(2))/p(1) + vL*sin(p(3))/p(1) - wL;
```

I.2 5 Robots in a Platoon Formation

I.2.1 Main Function

```
% ------------------------------------------------------------

% INITIAL CONDITIONS

% Clear windows
clc
clf

% Simulation time
tsim = 80;

% Leader initial position and trajectory velocity
% Leader with a straight line
xL_init = [0,0,0];
R = 10;
gamma = 0;
vL = 10;
wL = 0;

% Follower initial positions
xF_init1 = [-1,-1,0];
xF_init2 = [-2,-5,pi/2];
xF_init3 = [-3,-3,pi];
xF_init4 = [-6,-4,pi/2];

% Desired distance and angle from the leader
rho_d1 = 2;
rho_d2 = 4;
rho_d3 = 6;
rho_d4 = 8;
alpha_d = 0;

% ------------------------------------------------------------

% LEADER AND FOLLOWER POSITION CALCULATIONS

% Convert leader coordinates from cartesian to polar coordinates
[polar1] = polar_robot(xL_init, xF_init1);
[polar2] = polar_robot(xL_init, xF_init2);
[polar3] = polar_robot(xL_init, xF_init3);
[polar4] = polar_robot(xL_init, xF_init4);

% Use the ordinary differential equation function
% to determine the position of the follower robot,
% based on the follower's velocity calculated in the
% function follower_sim.m
options = odeset('RelTol',1e-3);
[tF1,pF1] = ode45('follower_controller',tsim,polar1,options,...
    vL,wL,rho_d1,alpha_d);
```

```
50 [tF2,pF2] = ode45('follower_controller',tsim,polar2,options,...
51     vL,wL,rho_d2,alpha_d);
52 [tF3,pF3] = ode45('follower_controller',tsim,polar3,options,...
53     vL,wL,rho_d3,alpha_d);
54 [tF4,pF4] = ode45('follower_controller',tsim,polar4,options,...
55     vL,wL,rho_d4,alpha_d);
56
57 % Calculate the leader path
58 [xL1] = leader_path(tF1,R,gamma);
59 [xL2] = leader_path(tF2,R,gamma);
60 [xL3] = leader_path(tF3,R,gamma);
61 [xL4] = leader_path(tF4,R,gamma);
62
63 % Find the number of steps for each of the followers
64 size(tF1)
65 size(tF2)
66 size(tF3)
67 size(tF4)
68
69 % Choose the the time with the smallest number of steps
70 tF = tF4;
71 xL = xL4;
72
73 % Convert follower coordinates from polar to cartesian
74 [xF1] = convert_cartesian(xF_init1,pF1,tF,xL);
75 [xF2] = convert_cartesian(xF_init2,pF2,tF,xL);
76 [xF3] = convert_cartesian(xF_init3,pF3,tF,xL);
77 [xF4] = convert_cartesian(xF_init4,pF4,tF,xL);
78
79 % ---------------------------------------------------------
80
81 % CREATE A SIMULATION ANIMATION
82
83 figure;
84 clf reset;
85
86 % Map dimensions
87 xmax = 15;      xmin = 0;
88 ymax = 15;      ymin = 0;
89
90 j = 0;
91 for i = 2:size(tF,1)
92     clf;
93     box on;
94
95     % Set axes
96     axis([xmin-10 xmax+5 ymin-10 ymax+5]);
97     axis equal;
98     axis manual;
99
100    % Plot line between the leader and follower
101    hold on;
102    plot([xL(i,1),xF1(i,1)],[xL(i,2),xF1(i,2)],'g');
103    plot([xL(i,1),xF2(i,1)],[xL(i,2),xF2(i,2)],'g');
```

```
104     plot([xF1(i,1),xF2(i,1)],[xF1(i,2),xF2(i,2)],'g');
105     hold off;
106     hold on;
107     plot([xF1(i,1),xF3(i,1)],[xF1(i,2),xF3(i,2)],'g');
108     plot([xF2(i,1),xF4(i,1)],[xF2(i,2),xF4(i,2)],'g');
109     plot([xF3(i,1),xF4(i,1)],[xF3(i,2),xF4(i,2)],'g');
110     hold off;
111
112     % Plot the leader
113     [X,Y] = moving_arrows(xL(i,:),axis(gca));
114     hold on;
115     plot(xL(2:i,1),xL(2:i,2),'r');
116     fill(X,Y,'r');
117     hold off;
118
119     % Plot follower 1
120     [X,Y] = moving_arrows(xF1(i,:),axis(gca));
121     hold on;
122     plot(xF1(2:i,1), xF1(2:i,2), 'b');
123     fill(X,Y,'b');
124     hold off;
125
126     % Plot follower 2
127     [X,Y] = moving_arrows(xF2(i,:),axis(gca));
128     hold on;
129     plot(xF2(2:i,1), xF2(2:i,2), 'b');
130     fill(X,Y,'b');
131     hold off;
132
133     % Plot follower 3
134     [X,Y] = moving_arrows(xF3(i,:),axis(gca));
135     hold on;
136     plot(xF3(2:i,1), xF3(2:i,2), 'b');
137     fill(X,Y,'b');
138     hold off;
139
140     % Plot follower 4
141     [X,Y] = moving_arrows(xF4(i,:),axis(gca));
142     hold on;
143     plot(xF4(2:i,1), xF4(2:i,2), 'b');
144     fill(X,Y,'b');
145     hold off;
146
147     % Axes labels
148     xlabel('x [m]');
149     ylabel('y [m]');
150
151     % Save a picture every 10 seconds
152     if ( mod(j,10) == 0 )
153         filename = strcat( 'pics/', int2str(j), '.png');
154         saveas(gcf,filename)
155     end
156
157     j = j + 1;
```

```
158     M(j) = getframe(gcf);
159
160 end
161
162 % Create an avi movie file
163 movie2avi(M, 'platoon_mult.avi');
```

I.2.2 Leader Path Function

```
function [xL] = leader_path(tF,R,gamma)

%{
% Leader with a circular path
xL = zeros(size(tF,1),3);
for i = 1:size(tF,1)
    xL(i,1) = R*sin(gamma*tF(i,1));
    xL(i,2) = R*(1-cos(gamma*tF(i,1)));
    xL(i,3) = gamma*tF(i,1);
end
%}
% Leader with a straight path
xL = zeros(size(tF,1),3);
for i = 1:size(tF,1)
    xL(i,1) = i/5;
    xL(i,2) = i/5;
    xL(i,3) = pi/4;
end
```

I.2.3 Convert from Cartesian Coordinates to Polar Coordinates Function

```
function [polar] = polar_robot( xL_init, xF_init)

polar = zeros(3,1);

% Convert cartesian coordinates to polar coordinates
rho = sqrt((xL_init(2)-xF_init(2))^2 ...
    + (xL_init(1)-xF_init(1))^2);
alpha = atan((xL_init(2)-xF_init(2)) ...
    / (xL_init(1)-xF_init(1))) - xF_init(3);
zhi = alpha + xF_init(3) - xL_init(3) + pi;

polar(1) = rho;
polar(2) = alpha;
polar(3) = zhi;
```

I.2.4 Follower Controller Function

```
function [ dx ] = follower_controller( t,polar,flag,vL,wL,rho_d,alpha_d
    )

% Assume
k = 1;
rho = polar(1);
alpha = polar(2);
psi = polar(3);

dx = zeros(3,1);

err_rho = k * (rho_d - rho);
err_alpha = k * (alpha_d - alpha);

vF = -( err_rho + vL*cos(psi) ) / cos(alpha);
omegaF = -( err_rho*tan(alpha) + vL*cos(psi)*tan(alpha) ) ...
    / rho + ( vL*sin(psi) ) / rho - err_alpha;

dx(1) = err_rho;
dx(2) = err_alpha;
dx(3) = vF*sin(alpha)/rho + vL*sin(psi)/rho - wL;
```

I.2.5 Convert from Polar Coordinates to Cartesian Coordinates Function

```
function [xF] = convert_cartesian( xF_init, pF, tF, xL )

xF = zeros(size(tF,1),3);
xF(1,1) = xF_init(1);
xF(1,2) = xF_init(2);
xF(1,3) = xF_init(3);

% Convert from polar coordinates to cartesian coordinates
for i = 2:size(tF,1)

    thetaF = pF(i,3) - pF(i,2) + xL(i,3) - pi;

    xF(i,1) = xL(i,1) - pF(i,1)*cos(pF(i,2)+thetaF);
    xF(i,2) = xL(i,2) - pF(i,1)*sin(pF(i,2)+thetaF);
    xF(i,3) = thetaF;
end
```

I.3 5 Robots in a V-Formation

I.3.1 Main Function

```
% ------------------------------------------------------------

% INITIAL CONDITIONS

% Clear windows
clc
clf

% Simulation time
tsim = 30;

% Leader initial position and trajectory velocity
% Leader with a straight line
xL_init = [0,0,0];
R = 10;
gamma = 1;
vL = gamma*R;
wL = gamma;

% Follower initial positions
xF_init1 = [-1,-1,0];
xF_init2 = [-2,-2,pi/2];
xF_init3 = [-3,-3,-pi/2];
xF_init4 = [-4,-4,pi];

% Desired distance and angle from the leader
rho_d1 = 2;
rho_d2 = 4;
alpha_d1 = pi/6;
alpha_d2 = -pi/6;

% ------------------------------------------------------------

% LEADER AND FOLLOWER POSITION CALCULATIONS

% Convert to polar coordinates
[polar1] = polar_robot(xL_init, xF_init1);
[polar2] = polar_robot(xL_init, xF_init2);
[polar3] = polar_robot(xL_init, xF_init3);
[polar4] = polar_robot(xL_init, xF_init4);

% Use the ordinary differential equation function
% to determine the position of the follower robot,
% based on the follower's velocity calculated in the
% function follower_sim.m
options = odeset('RelTol',1e-3);
[tF1,pF1] = ode45('follower_controller',tsim,polar1,options,...
    vL,wL,rho_d1,alpha_d1);
[tF2,pF2] = ode45('follower_controller',tsim,polar2,options,...
```

```
50      vL,wL,rho_d1,alpha_d2);
51  [tF3,pF3] = ode45('follower_controller',tsim,polar3,options,...
52      vL,wL,rho_d2,alpha_d1);
53  [tF4,pF4] = ode45('follower_controller',tsim,polar4,options,...
54      vL,wL,rho_d2,alpha_d2);
55
56  % Calculate the leader path
57  [xL1] = leader_path(tF1,R,gamma);
58  [xL2] = leader_path(tF2,R,gamma);
59  [xL3] = leader_path(tF3,R,gamma);
60  [xL4] = leader_path(tF4,R,gamma);
61
62  % Find the number of steps for each of the followers
63  size(tF1)
64  size(tF2)
65  size(tF3)
66  size(tF4)
67
68  % Choose the the time with the smallest number of steps
69  tF = tF4;
70  xL = xL4;
71
72  % Convert follower coordinates from polar to cartesian
73  [xF1] = convert_cartesian(xF_init1,pF1,tF,xL);
74  [xF2] = convert_cartesian(xF_init2,pF2,tF,xL);
75  [xF3] = convert_cartesian(xF_init3,pF3,tF,xL);
76  [xF4] = convert_cartesian(xF_init4,pF4,tF,xL);
77
78  % ---------------------------------------------------------
79
80  % CREATE A SIMULATION ANIMATION
81
82  figure;
83  clf reset;
84
85  % Map dimensions
86  xmax = 10;      xmin = -10;
87  ymax = 20;      ymin = -5;
88
89  j = 0;
90  for i = 1:size(tF,1)
91      clf;
92      box on;
93
94      % Set axes
95      axis([xmin-5 xmax+5 ymin-5 ymax+5]);
96      axis equal;
97      axis manual;
98
99      % plot line between the leader and follower
100     hold on;
101     plot([xL(i,1),xF1(i,1)],[xL(i,2),xF1(i,2)],'g');
102     plot([xL(i,1),xF2(i,1)],[xL(i,2),xF2(i,2)],'g');
103     plot([xF1(i,1),xF2(i,1)],[xF1(i,2),xF2(i,2)],'g');
```

```
104     hold off;
105     hold on;
106     plot([xF1(i,1),xF3(i,1)],[xF1(i,2),xF3(i,2)],'g');
107     plot([xF2(i,1),xF4(i,1)],[xF2(i,2),xF4(i,2)],'g');
108     plot([xF3(i,1),xF4(i,1)],[xF3(i,2),xF4(i,2)],'g');
109     hold off;
110
111     % Plot the leader
112     [X,Y] = moving_arrows(xL(i,:),axis(gca));
113     hold on;
114     plot(xL(1:i,1),xL(1:i,2),'r');
115     fill(X,Y,'r');
116     hold off;
117
118     % Plot follower 1
119     [X,Y] = moving_arrows(xF1(i,:),axis(gca));
120     hold on;
121     plot(xF1(1:i,1), xF1(1:i,2), 'b');
122     fill(X,Y,'b');
123     hold off;
124
125     % Plot follower 2
126     [X,Y] = moving_arrows(xF2(i,:),axis(gca));
127     hold on;
128     plot(xF2(1:i,1), xF2(1:i,2), 'b');
129     fill(X,Y,'b');
130     hold off;
131
132     % Plot follower 3
133     [X,Y] = moving_arrows(xF3(i,:),axis(gca));
134     hold on;
135     plot(xF3(1:i,1), xF3(1:i,2), 'b');
136     fill(X,Y,'b');
137     hold off;
138
139     % Plot follower 4
140     [X,Y] = moving_arrows(xF4(i,:),axis(gca));
141     hold on;
142     plot(xF4(1:i,1), xF4(1:i,2), 'b');
143     fill(X,Y,'b');
144     hold off;
145
146     % Axes labels
147     xlabel('x [m]');
148     ylabel('y [m]');
149
150     % Save a picture every 10 seconds
151     if ( mod(j,10) == 0 )
152         filename = strcat( 'pics/', int2str(j), '.png');
153         saveas(gcf,filename)
154     end
155
156     j = j + 1;
157     M(j) = getframe(gcf);
```

```
158
159 end
160
161 % Create an avi movie file
162 movie2avi(M,'Vshape_mult_leader_follower.avi');
```

I.4 5 Robots in a Reconfiguring V-Formation

I.4.1 Main Function

```
% Clear command window
clc
clf

% Initial Robot Values
num_robot = 5;
sensor_radius = 15;
max_vel = 15;
max_ang_vel = 1;

% Time settings
t_sim = 20;
T = 0.1;
t_steps = floor(t_sim/T);

% Map
map_max = 40;
map_min = 0;

% Obstacles
obst_opening =15;
obst_radius = sensor_radius;
x_dist = 1.5*map_max;
y_dist = 0.4*map_max;

% Goal Position
goal = [3*map_max,0.75*map_max];
goal_radius = sensor_radius;

% V formation parameters
rho_desired = 10;
alpha_desired = pi/4;
min_alpha = 0;

% ------------------------------------------------------------

% Vehicle Positions

follower_robot = zeros(num_robot-1,3);

% randomly select leader vehicle
leader_robot = [rand*map_max, rand*map_max, 0];

% randomly select follower vehicles
for i = 1:(num_robot-1)
    follower_robot(i,:) = [rand*map_min, rand*map_max, 0];
end

% ------------------------------------------------------------
```

```
50 % ------------------------------------------------------------
51
52 % Movie
53 j = 0;
54 for t = 1:t_steps
55
56     % Axis properties
57     clf;
58     xmax = 3*map_max;    xmin = map_min;
59     ymax = map_max;      ymin= map_min;
60     axis([xmin, xmax+2*goal_radius,...
61         ymin, ymax+2*goal_radius]);
62     axis equal; axis manual;    grid on;    hold on;
63
64 % ------------------------------------------------------------
65
66     % Leader Motion with Obstacles
67
68     % robots travelling through a narrow passage
69     [narrow_opening] = narrow_passage(leader_robot, obst_radius,
           sensor_radius, obst_opening, map_max, map_min);
70     obst_dist = narrow_opening(1);
71     obst_angle = narrow_opening(2);
72     passage_dist = narrow_opening(3);
73 %{
74     % concave or convex obstacles
75     [concave_convex] = concave(leader_robot, obst_radius, x_dist,
           y_dist, map_max);
76     obst_dist = concave_convex(1);
77     obst_angle = concave_convex(2);
78 %}
79   % make sure that leader angles are less than 2*pi
80     if (leader_robot(3) > 2*pi)
81         leader_robot(3) = leader_robot(3) - 2*pi;
82     end
83
84   position = [leader_robot(1), leader_robot(2),...
85         leader_robot(3)];
86   [new_position] = obstacle_controller(position,...
87         goal, max_vel, max_ang_vel, goal_radius,...
88         obst_radius, obst_dist, obst_angle, T);
89   leader_robot = [new_position(1), new_position(2),...
90         new_position(3)];
91     vL = sqrt( new_position(4)^2 + new_position(5)^2 );
92
93     % Plot Leader Robot
94   [circle] = sensor_view(leader_robot ,sensor_radius);
95   [XL,YL] = moving_arrows(leader_robot);
96   fill(XL,YL,'r');
97
98 % ------------------------------------------------------------
99
100    % Follower Formation
101
```

```
102     for i = 1:(num_robot-1)
103
104         current_follower = [follower_robot(i,1) ,...
105             follower_robot(i,2), follower_robot(i,3)];
106
107         %[rho_alpha_psi] = polar_narrowpassage(i ,...
108         %    leader_robot, current_follower, num_robot ,...
109         %    passage_dist, rho_desired, alpha_desired ,...
110         %    min_alpha);
111         [rho_alpha_psi] = polar_concave(i, leader_robot ,...
112             current_follower, rho_desired, alpha_desired ,...
113             obst_dist);
114
115         polar = [rho_alpha_psi(1), rho_alpha_psi(2) ,...
116             rho_alpha_psi(3)];
117         rho_d = rho_alpha_psi(4);
118         alpha_d = rho_alpha_psi(5);
119
120         [formation_position] = formation_controller...
121             (current_follower, leader_robot, polar, rho_d ,...
122             alpha_d, vL, max_vel, max_ang_vel, T);
123         follower_robot(i,:) = formation_position;
124
125         %[circle] = sensor_view(current_follower ,...
126         %    obst_radius);
127         [XF,YF] = moving_arrows(follower_robot(i,:));
128         fill(XF,YF,'b');
129     end
130
131 % ------------------------------------------------------------
132
133     % plot goal
134     scatter(goal(1),goal(2), 100, 'd', 'm', 'filled');
135     rectangle('Position',[goal(1)-goal_radius ,...
136         goal(2)-goal_radius,2*goal_radius,2*goal_radius],...
137         'Curvature',[1,1],'EdgeColor','m');
138
139     % Save a picture every 10 seconds
140     if ( mod(t,10) == 0 )
141         filename = strcat( 'pics/', int2str(t), '.png');
142         saveas(gcf,filename)
143     end
144
145     hold off;
146
147     j = j + 1;
148     M(j) = getframe;
149
150 end
151
152 % Create a movie and save it as an AVI file
153 movie2avi(M,'video/simulation.avi');
```

I.4.2 Concave or Convex Obstacle Function

```
function [concave_convex] = concave(leader_robot,...
    obst_radius, x_dist, y_dist, map_max)

ob_width = 10;
ob_height = 30;
ob_thickness = 5;

%{
% concave obstacle
xlimit = [x_dist, x_dist+ob_width,...
    x_dist+ob_width-ob_thickness];
ylimit = [y_dist, y_dist+ob_height,...
    y_dist+ob_height-ob_thickness, y_dist+ob_thickness];
concaveX = xlimit([1 2 2 1 1 3 3 1]);
concaveY = ylimit([1 1 2 2 3 3 4 4]);
%}

% convex obstacle
xlimit = [x_dist, x_dist+ob_thickness,...
    x_dist+ob_width+ob_thickness,...
    x_dist+ob_width+2*ob_thickness ];
ylimit = [y_dist, y_dist-ob_height/3,...
    y_dist+ob_height/3];
concaveX = xlimit([1 2 3 4 3 2 1]);
concaveY = ylimit([1 2 2 1 3 3 1]);


fill (concaveX, concaveY, 'k');

% Sensors
obst_dist = 0;
obst_angle = 0;
for beam_angle = (leader_robot(3)-pi/2) : pi/12 :...
        (leader_robot(3)+pi/2);

    % avoid obstacles
    x_ob = [leader_robot(1),...
        leader_robot(1)+obst_radius*cos(beam_angle)];
    y_ob = [leader_robot(2),...
        leader_robot(2)+obst_radius*sin(beam_angle)];
    [xi, yi] = polyxpoly(x_ob, y_ob, concaveX, concaveY);
    mapshow(xi,yi,'DisplayType','point','Marker','o');

    dist = sqrt(xi.^2 + yi.^2);
    if (size(dist,1) == 2)
        dist = dist(1);
    end

    if (isempty(dist) == false)
        obst_dist = dist;
        obst_angle = beam_angle;
        if (obst_angle > 2*pi)
```

```
53              obst_angle = obst_angle - 2*pi;
54          elseif (obst_angle < 0)
55              obst_angle = obst_angle + 2*pi;
56          end
57      end
58  end
59
60  concave_convex = [obst_dist, obst_angle];
```

I.4.3 Polar Calculations for Concave or Convex Obstacle Function

```
function [rho_alpha_psi] = polar_concave(i, leader_robot,...
    follower, rho_desired, alpha_desired, obst_dist)

rho = sqrt( (leader_robot(1) - follower(1))^2 ...
    + (leader_robot(2) - follower(2))^2 );
alpha = atan( (leader_robot(2) - follower(2)) ...
    / (leader_robot(1) - follower(1)) ) - follower(3);
psi = alpha + follower(3) - leader_robot(3) + pi;

if (obst_dist == 0)
  if ( mod(i,2) == 0 )
        alpha_d = alpha_desired;
        rho_d = rho_desired + rho_desired*(i-2)/2;
    else
        alpha_d = - alpha_desired;
        rho_d = rho_desired + rho_desired*(i-1)/2;
    end
else
    alpha_d = 0;
    rho_d = rho_desired + rho_desired*(i-1)/2;
end

rho_alpha_psi = [rho, alpha, psi, rho_d, alpha_d];
```

I.4.4 Narrow Passage Obstacle Function

```
function [narrow_opening] = narrow_passage(leader_robot,...
    obst_radius, sensor_radius, obst_opening, map_max, map_min)

ob_width = 30;
ob_height = map_max;
x_dist = 0.75*map_max;
y_dist = 0.75*map_max;

xlimit = [x_dist, x_dist+ob_width, x_dist+2*ob_width,...
    x_dist+3*ob_width];
ylimit = [y_dist-obst_opening/2-ob_height,...
    y_dist-obst_opening/2, y_dist+obst_opening/2+ob_height,...
    y_dist+obst_opening/2];
ob_bottomX = xlimit([1 2 3 4]);
ob_bottomY = ylimit([1 2 2 1]);
ob_topX = ob_bottomX;
ob_topY = ylimit([3 4 4 3]);

fill(ob_bottomX, ob_bottomY, 'k');
fill(ob_topX, ob_topY, 'k');


% Sensors
obst_dist = 0;
obst_angle = 0;
passage_dist = 0;
passage_distT = 0;  passage_distB = 0;

for beam_angle = (leader_robot(3)-pi/2) : pi/12 :...
        (leader_robot(3)+pi/2);

    % avoid obstacles
    x_ob = [leader_robot(1),...
        leader_robot(1)+obst_radius*cos(beam_angle)];
    y_ob = [leader_robot(2),...
        leader_robot(2)+obst_radius*sin(beam_angle)];
    [xT, yT] = polyxpoly(x_ob, y_ob, ob_topX, ob_topY);
    [xB, yB] = polyxpoly(x_ob, y_ob, ob_bottomX, ob_bottomY);
    mapshow(xT,yT,'DisplayType','point','Marker','o');
    mapshow(xB,yB,'DisplayType','point','Marker','o');

    obst_distT = sqrt(xT.^2 + yT.^2);
    obst_distB = sqrt(xB.^2 + yB.^2);

    if (size(obst_distT,1) == 2)
        obst_distT = obst_distT(1);
    elseif (size(obst_distB,1) == 2)
        obst_distB = obst_distB(1);
    end

    if (isempty(obst_distT) == false ...
            && isempty(obst_distB) == true)
```

```
53          obst_dist = obst_distT;
54      elseif (isempty(obst_distT) == true ...
55              && isempty(obst_distB) == false)
56        obst_dist = obst_distB;
57      end
58
59      if (isempty(obst_distT) == false ...
60              || isempty(obst_distB) == false)
61          obst_angle = beam_angle;
62          if (obst_angle >= 2*pi)
63              obst_angle = obst_angle - 2*pi;
64          elseif (obst_angle < 0)
65              obst_angle = obst_angle + 2*pi;
66          end
67      end
68
69      % calculate passage distance
70      x_pass = [leader_robot(1) ,...
71          leader_robot(1)+sensor_radius*cos(beam_angle)];
72      y_pass = [leader_robot(2) ,...
73          leader_robot(2)+sensor_radius*sin(beam_angle)];
74      [xT_pass, yT_pass] = polyxpoly(x_pass, y_pass ,...
75          ob_topX, ob_topY);
76      [xB_pass, yB_pass] = polyxpoly(x_pass, y_pass ,...
77          ob_bottomX, ob_bottomY);
78      mapshow(xT_pass,yT_pass,'DisplayType','point',...
79          'Marker','o');
80      mapshow(xB_pass,yB_pass,'DisplayType','point',...
81          'Marker','o');
82
83      if (size(yT_pass,1) == 2)
84          yT_pass = yT_pass(1);
85      elseif (size(yB_pass,1) == 2)
86          yB_pass = yB_pass(1);
87      end
88
89
90      if (isempty(yT_pass) == false ...
91              && isempty(yB_pass) == true)
92          passage_distT = yT_pass;
93      elseif (isempty(yT_pass) == true ...
94              && isempty(yB_pass) == false)
95          passage_distB = yB_pass;
96      end
97
98      passage_dist = passage_distT - passage_distB;
99
100 end
101
102 narrow_opening = [obst_dist, obst_angle, passage_dist];
```

I.4.5 Polar Calculations for Narrow Passage Function

```
function [rho_alpha_psi] = polar_narrowpassage(i,...
    leader_robot, follower, num_robot, passage_dist,...
    rho_desired, alpha_desired, min_alpha)

rho = sqrt( (leader_robot(1) - follower(1))^2 ...
    + (leader_robot(2) - follower(2))^2 );
alpha = atan( (leader_robot(2) - follower(2)) ...
    / (leader_robot(1) - follower(1)) ) - follower(3);
psi = alpha + follower(3) - leader_robot(3) + pi;

if ( passage_dist == 0 || (passage_dist/4) ...
        >= ((num_robot-1)*rho_desired/2) )
  if ( mod(i,2) == 0 )
        alpha_d = alpha_desired;
        rho_d = rho_desired + rho_desired*(i-2)/2;
    else
        alpha_d = - alpha_desired;
        rho_d = rho_desired + rho_desired*(i-1)/2;
  end
else
  new_alpha = asin( (passage_dist/4) ...
        / ((num_robot-1)*rho_desired/2) );

  if ( new_alpha < min_alpha )
        alpha_d = 0;
        rho_d = rho_desired + rho_desired*(i-1);
    else
        if ( mod(i,2) == 0 )
            alpha_d = new_alpha;
            rho_d = rho_desired + rho_desired*(i-2)/2;
        else
            alpha_d = - new_alpha;
            rho_d = rho_desired + rho_desired*(i-1)/2;
        end
  end
end

rho_alpha_psi = [rho, alpha, psi, rho_d, alpha_d];
```

I.4.6 Obstacle Controller Function

```
function [new_position] = obstacle_controller(robot,...
    goal, max_vel, max_ang_vel, goal_radius, obst_radius,...
    obst_dist, obst_angle, T)

% distance to goal
delta_x = goal(1) - robot(1);
delta_y = goal(2) - robot(2);

theta = atan(delta_y/delta_x);
delta_theta = theta - robot(3);

dist_goal = sqrt( delta_x^2 + delta_y^2 );
dist_relation = exp(-dist_goal/goal_radius);


% Constants
k_a = 1;
k_n = 1;
k_t = 1;

% Attractive Velocity
u_a = k_a * max_vel * (1-dist_relation);

% Theta
close_to_zero = 0.2;
if ( abs(delta_theta) < close_to_zero)
    theta = theta;
elseif ( delta_theta >= 0 && abs(delta_theta) < pi)
    theta = robot(3) + max_ang_vel*T;
elseif ( delta_theta >= 0 && abs(delta_theta) >= pi)
    theta = robot(3) - max_ang_vel*T;
elseif ( delta_theta < 0 && abs(delta_theta) < pi)
    theta = robot(3) - max_ang_vel*T;
elseif ( delta_theta < 0 && abs(delta_theta) >= pi)
    theta = robot(3) + max_ang_vel*T;
end


% Repulsive Velocity
if (obst_dist == 0)
    normal = 0;
    tangent = 0;
    obst_relation = 0;
    normal_angle = 0;
    tangent_angle = 0;
    theta_obst = 0;

else
    normal = 1/obst_dist;
    tangent = 1/obst_dist;
    obst_relation = exp(-obst_dist/obst_radius);

```

```
53      % normal angle
54      normal_angle = obst_angle + pi;
55      if (normal_angle > 3*pi/2)
56        normal_angle = normal_angle - 2*pi;
57      end
58
59      % tangent angle
60      tangent_angle = normal_angle - pi/2;
61
62      if (tangent_angle > 3*pi/2)
63        tangent_angle = tangent_angle - 2*pi;
64      end
65
66
67      theta_obst = (normal_angle + tangent_angle) / 2;
68      if (theta_obst < 0)
69          theta_obst = theta_obst + 2*pi;
70      end
71      delta_theta_obst = theta_obst - robot(3);
72
73
74      if ( abs(delta_theta_obst) < close_to_zero )
75          theta_obst = theta_obst;
76      elseif ( delta_theta_obst >= 0 ...
77              && abs(delta_theta_obst) < pi)
78          theta_obst = robot(3) + max_ang_vel*T;
79      elseif ( delta_theta_obst >= 0 ...
80              && abs(delta_theta_obst) >= pi)
81          theta_obst = robot(3) - max_ang_vel*T;
82      elseif ( delta_theta_obst < 0 ...
83              && abs(delta_theta_obst) < pi)
84          theta_obst = robot(3) - max_ang_vel*T;
85      elseif ( delta_theta_obst < 0 ...
86              && abs(delta_theta_obst) >= pi)
87          theta_obst = robot(3) + max_ang_vel*T;
88      end
89
90
91      % robot stops looking for goal
92      u_a = 0;
93      theta = 0;
94
95  end
96
97  u_n = k_n * normal * obst_relation;
98  u_t = k_t * tangent * obst_relation;
99
100
101 % Sum of velocity terms
102 u_totX = u_a*cos(theta) + u_n*cos(normal_angle) ...
103     + u_t*cos(tangent_angle);
104 u_totY = u_a*sin(theta) + u_n*sin(normal_angle) ...
105     + u_t*sin(tangent_angle);
106
```

```
107
108 delta_d = [u_totX*T, u_totY*T];
109 delta_theta = theta + theta_obst;
110
111
112 new_position = [robot(1)+delta_d(1), robot(2)+delta_d(2) ,...
113     delta_theta, u_totX, u_totY];
```

I.4.7 Formation Controller Function

```
function [formation_position] = formation_controller...
    (robot, leader, polar, rho_desired, alpha_desired,...
    vL, max_vel, max_ang_vel, T)

k = 1;
rho = polar(1);
alpha = polar(2);
psi = polar(3);

err_rho = k * (rho_desired - rho);
err_alpha = k * (alpha_desired - alpha);

vF = - ( err_rho + vL*cos(psi) ) / cos(alpha);
omegaF = - (err_rho*tan(alpha) + vL*cos(psi)*tan(alpha))...
    / rho + ( vL*sin(psi) ) / rho - err_alpha;

delta_d = [vF*T*cos(robot(3)), vF*T*sin(robot(3))];
delta_theta = omegaF*T;


formation_position = [robot(1)+delta_d(1), robot(2)+delta_d(2),...
    robot(3)+delta_theta];
```

Appendix J

PLAYER: Geometric Formation Controller Code

J.1 geo_obst.cpp

```
1 //============================================================
2 // Name        : geom.cpp
3 // Author      :
4 // Version     :
5 // Copyright   :
6 // Description :
7 //============================================================
8
9 #include <iostream>
10 #include <cmath>
11 #include <csignal>
12 #include "libplayerc++/playerc++.h"
13
14 using namespace std;
15 using namespace PlayerCc;
16
17
18 // ----------------------------------------------------------
19
20
21 // FUNCTION PROTOTYPES
22
23 volatile sig_atomic_t sigShutdown;
24 void UnexpectedShutdown (void);
25 void HandleShutdownSignal (int);
26
27 void CheckObstacle (double&, double&, int&, SonarProxy&, SonarProxy&,
      double, double);
28 double LeaderVelocity (double, double, double, double, double, double,
      double, int);
29 double LeaderAngVelocity(double, double, double, double, double, double
      , int);
30
```

```
31 double FollowerVelocity (double, double, double, double, double, double
       , double, double, int, int);
32 double FollowerAngVelocity (double, double, double, double, double,
       double, double, double, int, int);
33
34
35 // ------------------------------------------------
36
37
38 // GLOBAL VARIABLES
39 double const PI = 3.14159, close_to_zero = 0.1;
40
41 double xGoal = 2, yGoal = -0.25, goalRadius = 1, obstRadius = 1;
42 double vmax = 0.6, omega_max = 1, t = 1;
43 double rho_desired = 0.75, angle_degrees = 0, alpha_desired = dtor(
       angle_degrees);
44
45 double obstDist1, obstAngle1, passageDist;
46 double xL, yL, thetaL, xF1, yF1, thetaF1, xF2, yF2, thetaF2;
47 double vL, omegaL, vF1, omegaF1, vF2, omegaF2;
48 int turnDirection1 = 1, turnDirection2 = 1, turnDirection3 = 1;
49
50 double K_p = 0.2, K_d = 0.2;
51 //int updateSim = 0;
52
53 // ------------------------------------------------
54
55
56 // MAIN FUNCTION
57
58 int main()
59 {
60   try
61   {
62 /*
63     // Get final pose from user
64     cout << "Robot goal x: "<< endl;
65     cin >> xGoal;
66     cout << "Robot goal y: " << endl;
67     cin >> yGoal;
68 */
69     // Call the player client, name the vehicle, and include any
         neccesary proxies
70     PlayerClient robot1("localhost", 6665);
71     Position2dProxy pose1(&robot1, 0);
72     SonarProxy  sonar1(&robot1, 0);
73     SonarProxy  ir1(&robot1, 1);
74
75     PlayerClient robot2("localhost", 6666);
76     Position2dProxy pose2(&robot2, 0);
77     SonarProxy  sonar2(&robot2, 0);
78     SonarProxy  ir2(&robot2, 1);
79
80     PlayerClient robot3("localhost", 6667);
```

```
81      Position2dProxy pose3(&robot3, 0);
82      SonarProxy  sonar3(&robot3, 0);
83      SonarProxy  ir3(&robot3, 1);
84
85      //SimulationProxy *sim = NULL;
86      //if(updateSim == 1)
87      //{
88        //sim = new SimulationProxy(&robot1, 0);
89      //}
90
91      // Handle unexpected shutdown
92      UnexpectedShutdown();
93
94
95      while (sigShutdown == 0)
96      {
97        // read from the proxies
98        robot1.Read();
99        pose1.RequestGeom();
100       sonar1.RequestGeom();
101       ir1.RequestGeom();
102
103       robot2.Read();
104       pose2.RequestGeom();
105       sonar2.RequestGeom();
106       ir2.RequestGeom();
107
108       robot3.Read();
109       pose3.RequestGeom();
110       sonar3.RequestGeom();
111       ir3.RequestGeom();
112
113
114       // get the x, y, and theta position
115       xL = pose1.GetXPos();
116       yL = pose1.GetYPos();
117       thetaL = pose1.GetYaw();
118
119       xF1 = pose2.GetXPos();
120       yF1 = pose2.GetYPos();
121       thetaF1 = pose2.GetYaw();
122
123       xF2 = pose3.GetXPos();
124       yF2 = pose3.GetYPos();
125       thetaF2 = pose3.GetYaw();
126
127
128       // if the robot has reached the goal then stop
129       if ( ((xGoal-xL) <= close_to_zero && (yGoal-yL) <= close_to_zero)
              )
130       {
131         pose1.SetSpeed(0, 0);
132         pose2.SetSpeed(0, 0);
133         pose3.SetSpeed(0, 0);
```

```
134         break;
135       }
136
137       // See if any obstacles are in view around the leader, if so find
              its distance and angle
138       CheckObstacle(obstDist1, obstAngle1, turnDirection1, sonar1, ir1,
              thetaL, obstRadius);
139
140       // Calculate the velocity commands
141       vL = LeaderVelocity(xL, yL,vmax, goalRadius, obstDist1,
              obstAngle1,obstRadius, turnDirection1);
142       vF1 = FollowerVelocity(vL, xL, yL, thetaL, xF1, yF1, thetaF1,
              passageDist, -1, 1);
143       vF2 = FollowerVelocity(vL, xL, yL, thetaL, xF2, yF2, thetaF2,
              passageDist, 1, 2);
144
145       omegaL = LeaderAngVelocity(xL, yL, thetaL, omega_max, obstDist1,
              obstAngle1, turnDirection1);
146       omegaF1 = FollowerAngVelocity(vL, xL, yL, thetaL, xF1, yF1,
              thetaF1, passageDist, -1, 1);
147       omegaF2 = FollowerAngVelocity(vL, xL, yL, thetaL, xF2, yF2,
              thetaF2, passageDist, 1, 2);
148
149
150       // set the speed of the robot
151       pose1.SetSpeed(vL, omegaL);
152       pose2.SetSpeed(vF1, omegaF1);
153       pose3.SetSpeed(vF2, omegaF2);
154       //pose1.SetSpeed(vmax,0);
155       //pose2.SetSpeed(0, 0);
156       //pose3.SetSpeed(0, 0);
157
158     }
159
160     // stop the vehicle by using a keyboard stroke
161     pose1.SetSpeed(0, 0);
162     pose2.SetSpeed(0, 0);
163     pose3.SetSpeed(0, 0);
164     cout << "robot stopped" <<endl;
165
166     //if(updateSim == 1)
167       //delete sim;
168     sleep(100000);
169   }
170
171   catch (PlayerCc::PlayerError e)
172   {
173     std::cerr << e << std::endl;
174     return -1;
175   }
176
177   return 0;
178 }   // end of main
179
```

```
180
181 // ------------------------------------------------------
182
183
184 // FUNCTIONS
185
186 double FollowerVelocity (double vL, double xL, double yL, double thetaL
        , double xF, double yF, double thetaF,
187     double obstDist, int whichSide, int robotPosition)
188 {
189   double rho, alpha, psi;
190   double vF, err_rho, rho_d = rho_desired;
191
192   rho = sqrt( (xL - xF)*(xL - xF) + (yL - yF)*(yL - yF) );
193   alpha = atan( (yL-yF)/(xL-xF) ) - thetaF;
194   psi = alpha + thetaF - thetaL + PI;
195
196   if (obstDist != 0 || alpha_desired == 0)
197     rho_d = rho_desired * robotPosition;
198
199   err_rho = rho - rho_d;
200   vF = (K_p/(1 + K_d*cos(alpha))) * err_rho - (cos(psi)/cos(alpha)) *
        vL;
201
202   return (vF);
203 }
204
205
206 double FollowerAngVelocity (double vL, double xL, double yL, double
       thetaL, double xF, double yF, double thetaF,
207     double obstDist, int whichSide, int robotPosition)
208 {
209   double rho, alpha, psi;
210   double omegaF, err_rho, err_alpha;
211   double alpha_d, rho_d;
212
213   rho = sqrt( (xL - xF)*(xL - xF) + (yL - yF)*(yL - yF) );
214   alpha = atan( (yL-yF)/(xL-xF) ) - thetaF;
215   psi = alpha + thetaF - thetaL + PI;
216
217   rho_d = rho_desired;
218   alpha_d = alpha_desired * whichSide;
219
220   if (obstDist != 0 || alpha_desired == 0)
221   {
222     alpha_d = 0;
223     rho_d = rho_desired * robotPosition;
224   }
225
226   err_rho = rho - rho_d;
227   err_alpha = alpha - alpha_d;
228
229   omegaF = (K_p * tan(alpha) / rho) * err_rho + (K_p / (1 + K_d)) *
        err_alpha +
```

```
230       (1 + K_d) * ((-cos(psi) * tan(alpha) + sin(psi)) / rho) * vL;
231
232   return (omegaF);
233 }
234
235
236 void CheckObstacle(double &obstDist, double &obstAngle, int &
      turnDirection, SonarProxy &sonar, SonarProxy &ir, double theta,
      double obstRadius)
237 {
238   double minDist = obstRadius, minAngle;
239   int sonarMin = 0, irMin = 0;
240
241   // sonar sensors
242   for(int i=0; i<=2; i++)
243   {
244     if(sonar[i] < minDist && sonar[i] > 0)
245     {
246       minDist = sonar[i];
247       sonarMin = i+1;
248       //cout << "sonar reading: " << i+1 << endl;
249     }
250   }
251
252   // ir sensors
253   for(int j=0; j<=6; j++)
254   {
255     if(ir[j] < minDist && ir[j] > 0)
256     {
257       minDist = ir[j];
258       irMin = j+1;
259       sonarMin = 0;
260       //cout << "ir reading: " << j+1 << endl;
261     }
262   }
263
264   //get the min angle
265   if (sonarMin > 0)
266   {
267     switch (sonarMin)
268     {
269       case 1:
270       {
271         minAngle = theta - dtor(45);
272         turnDirection = -1;
273         break;
274       }
275       case 2:
276       {
277         minAngle = theta;
278         turnDirection = 1;
279         break;
280       }
281       case 3:
```

```
282         {
283           minAngle = theta - dtor(45);
284           turnDirection = 1;
285           break;
286         }
287         case 4:
288         {
289           minAngle = theta - dtor(135);
290           turnDirection = 1;
291           break;
292         }
293         case 5:
294         {
295           //minAngle = theta - dtor(180);
296           //turnDirection = 1;
297           minDist = obstRadius;
298           break;
299         }
300         case 6:
301         {
302           minAngle = theta - dtor(135);
303           turnDirection = 1;
304           break;
305         }
306       }
307       //cout << "Sonar Angle: " << minAngle << endl;
308     }
309     else if (irMin > 0)
310     {
311       switch (irMin)
312       {
313         case 1:
314         {
315           minAngle = theta - dtor(40);
316           turnDirection = -1;
317           break;
318         }
319         case 2:
320         {
321           minAngle = theta - dtor(10);
322           turnDirection = -1;
323           break;
324         }
325         case 3:
326         {
327           minAngle = theta - dtor(10);
328           turnDirection = 1;
329           break;
330         }
331         case 4:
332         {
333           minAngle = theta - dtor(40);
334           turnDirection = 1;
335           break;
```

```
336         }
337         case 5:
338         {
339           minAngle = theta - dtor(90);
340           turnDirection = 1;
341           break;
342         }
343         case 6:
344         {
345           //minAngle = theta - dtor(180);
346           //turnDirection = 1;
347           minDist = obstRadius;
348           break;
349         }
350         case 7:
351         {
352           minAngle = theta - dtor(90);
353           turnDirection = -1;
354           break;
355         }
356       }
357       //cout << "IR Angle: " << rtod(minAngle) << endl;
358     }
359
360     obstDist = 0;
361     if (minDist < obstRadius && minDist > 0)
362     {
363       obstDist = minDist;
364       obstAngle = minAngle;
365       if(obstAngle < 0)
366         obstAngle = obstAngle + 2*PI;
367     }
368 }
369
370 double LeaderVelocity (double x, double y, double vmax, double
      goalRadius, double obstDist, double obstAngle, double obstRadius,
      int turnDirection)
371 {
372   // distance to goal
373   double delta_x = xGoal - x;
374   double delta_y = yGoal - y;
375   double distGoal = sqrt( delta_x*delta_x + delta_y*delta_y );
376   double thetaCalc = atan(delta_y/delta_x);
377   double distRelation = exp(-distGoal/goalRadius);
378
379   // constants
380   double k_a = 1, k_n = 1, k_t = 1;
381
382   // attractive velocity
383   double u_a = k_a * vmax * (1-distRelation);
384
385   // repulsive velocity
386   double normal, tangent, obstRelation, normalAngle, tangentAngle;
387   if (obstDist == 0)
```

```
388   {
389     normal = 0, tangent = 0;
390     obstRelation = 0;
391     normalAngle = 0, tangentAngle = 0;
392   }
393
394   else
395   {
396     normal = 0.5/(obstDist), tangent = 0.5/(obstDist);
397     obstRelation = exp(-obstDist/obstRadius);
398
399     u_a = 0;
400
401     // normal angle
402     normalAngle = obstAngle + PI;
403     if (normalAngle > 2*PI)
404       normalAngle = normalAngle - 2*PI;
405
406     // tangent angle
407     tangentAngle = normalAngle - (PI/2 *turnDirection);
408   }
409
410   // normal and tangent velocities
411   double u_n = k_n * normal * obstRelation;
412   double u_t = k_t * tangent * obstRelation;
413
414   // sum of velocity terms
415   double v, vx, vy;
416   vx = u_a*cos(thetaCalc) + u_n*cos(normalAngle) + u_t*cos(tangentAngle
        );
417   vy = u_a*sin(thetaCalc) + u_n*sin(normalAngle) + u_t*sin(tangentAngle
        );
418   v = sqrt(vx*vx + vy*vy);
419   if (v > vmax)
420     v = vmax;
421   else if (v < -vmax)
422     v = -vmax;
423
424   return (v);
425 }
426
427 double LeaderAngVelocity(double x, double y, double theta, double
      omega_max, double obstDist, double obstAngle, int turnDirection)
428 {
429   double omega_a, omega_o = 0, omega;
430   double k_a = 1, k_o = 1;
431
432   // distance to goal
433   double delta_x = xGoal - x;
434   double delta_y = yGoal - y;
435   double thetaCalc = atan(delta_y/delta_x);
436   double delta_theta = thetaCalc - theta;
437
438   // attractive angular velocity
```

```
439   omega_a = k_a * delta_theta/t;
440   if (delta_theta >= omega_max)
441     omega_a = omega_max;
442   else if (delta_theta < - omega_max)
443     omega_a = - omega_max;
444
445   // repulsive velocity
446   double normalAngle, tangentAngle;
447   if (obstDist == 0)
448     normalAngle = 0, tangentAngle = 0;
449
450   else
451   {
452     // normal angle
453     normalAngle = obstAngle + PI;
454     if (normalAngle > 2*PI)
455       normalAngle = normalAngle - 2*PI;
456
457     // tangent angle
458     tangentAngle = normalAngle - (PI/2 * turnDirection);
459
460     // get the resultant angle
461     double thetaObst = (normalAngle + tangentAngle) / 2;
462     if (thetaObst > 2*PI)
463       thetaObst = thetaObst - 2*PI;
464     double delta_thetaObst = (thetaObst - theta) * turnDirection;
465
466     // repulsive velocity
467     omega_o = k_o * delta_thetaObst/t;
468     if (delta_thetaObst >= omega_max)
469       omega_o = omega_max;
470     else if (delta_thetaObst < - omega_max)
471       omega_o = - omega_max;
472
473     // set omega_a = 0
474     omega_a = 0;
475
476   }
477
478   omega = omega_a + omega_o;
479
480   return (omega);
481 }
482
483
484
485 void UnexpectedShutdown (void)
486 {
487   sigShutdown = 0;
488   sigset_t blockmask;
489   struct sigaction sa;
490   sigfillset(&blockmask);
491   sa.sa_handler = HandleShutdownSignal;
492   sa.sa_mask = blockmask;
```

```
493   sa.sa_flags = 0;
494   sigaction(SIGINT, &sa, NULL);
495   sigaction(SIGQUIT, &sa, NULL);
496 }
497
498 void HandleShutdownSignal (int sig __attribute__ ((unused)))
499 {
500    sigShutdown = 1;
501 }
```

J.2 geo_narrow.cpp

```
 1 //============================================
 2 // Name        : geom.cpp
 3 // Author      :
 4 // Version     :
 5 // Copyright   :
 6 // Description :
 7 //============================================
 8
 9 #include <iostream>
10 #include <cmath>
11 #include <csignal>
12 #include "libplayerc++/playerc++.h"
13
14 using namespace std;
15 using namespace PlayerCc;
16
17
18 // ----------------------------------------------
19
20
21 // FUNCTION PROTOTYPES
22
23 volatile sig_atomic_t sigShutdown;
24 void UnexpectedShutdown (void);
25 void HandleShutdownSignal (int);
26
27 void CheckObstacle (double&, double&, int&, SonarProxy&, SonarProxy&,
      double);
28 double LeaderVelocity (double, double, double, double, int);
29 double LeaderAngVelocity (double, double, double, double, double,
      double, int);
30
31 double PassageDistance (SonarProxy&, SonarProxy&);
32 double FollowerVelocity (double, double, double, double, double, double
      , double, double, int, int);
33 double FollowerAngVelocity (double, double, double, double, double,
      double, double, double, int, int);
34
35
36 // ----------------------------------------------
37
```

```
38
39 // GLOBAL VARIABLES
40 double const PI = 3.14159, close_to_zero = 0.2;
41
42 double xGoal = 5, yGoal = 0, goalRadius = 1, obstRadius = 1, beamRadius
       = 1;
43 double vmax = 0.3, omega_max = 1, t = 1;
44 double rho_desired = 0.75, angle_degrees = 30, alpha_desired = dtor(
      angle_degrees);
45
46 double obstDist1, obstAngle1, passageDist;
47 double xL, yL, thetaL, xF1, yF1, thetaF1, xF2, yF2, thetaF2;
48 double vL, omegaL, vF1, omegaF1, vF2, omegaF2;
49 int turnDirection1 = 1, turnDirection2 = 1, turnDirection3 = 1;
50
51 double K_p = 0.3, K_d = 0.3;
52
53 // ------------------------------------------------------------
54
55
56 // MAIN FUNCTION
57
58 int main()
59 {
60   try
61   {
62 /*
63     // Get final pose from user
64     cout << "Robot goal x: "<< endl;
65     cin >> xGoal;
66     cout << "Robot goal y: " << endl;
67     cin >> yGoal;
68 */
69
70     cout << endl << "running program .. " << endl;
71
72     // Call the player client, name the vehicle, and include any
         neccesary proxies
73     PlayerClient robot1("localhost", 6665);
74     Position2dProxy pose1(&robot1, 0);
75     SonarProxy  sonar1(&robot1, 0);
76     SonarProxy  ir1(&robot1, 1);
77
78     PlayerClient robot2("localhost", 6666);
79     Position2dProxy pose2(&robot2, 0);
80     SonarProxy  sonar2(&robot2, 0);
81     SonarProxy  ir2(&robot2, 1);
82
83     PlayerClient robot3("localhost", 6667);
84     Position2dProxy pose3(&robot3, 0);
85     SonarProxy  sonar3(&robot3, 0);
86     SonarProxy  ir3(&robot3, 1);
87
88     // Handle unexpected shutdown
```

```
89      UnexpectedShutdown();
90
91      while (sigShutdown == 0)
92      {
93        // read from the proxies
94        robot1.Read();
95        pose1.RequestGeom();
96        sonar1.RequestGeom();
97        ir1.RequestGeom();
98
99        robot2.Read();
100       pose2.RequestGeom();
101       sonar2.RequestGeom();
102       ir2.RequestGeom();
103
104       robot3.Read();
105       pose3.RequestGeom();
106       sonar3.RequestGeom();
107       ir3.RequestGeom();
108
109       // get the x, y, and theta position
110       xL = pose1.GetXPos();
111       yL = pose1.GetYPos();
112       thetaL = pose1.GetYaw();
113
114       xF1 = pose2.GetXPos();
115       yF1 = pose2.GetYPos();
116       thetaF1 = pose2.GetYaw();
117
118       xF2 = pose3.GetXPos();
119       yF2 = pose3.GetYPos();
120       thetaF2 = pose3.GetYaw();
121
122       // if the robot has reached the goal then stop
123       if ( (abs(xGoal-xL) <= close_to_zero) && (abs(yGoal-yL) <=
              close_to_zero) )
124       {
125         pose1.SetSpeed(0, 0);
126         pose2.SetSpeed(0, 0);
127         pose3.SetSpeed(0, 0);
128         break;
129       }
130
131       // See if any obstacles are in view around the leader, if so find
            its distance and angle
132       //CheckObstacle(obstDist1, obstAngle1, turnDirection1, sonar1,
            ir1, thetaL);
133       obstDist1 = 0, obstAngle1 = 0;
134
135       // If there is a narrow passage, calculate the distance between
            the obstacles
136       passageDist = PassageDistance(sonar1, ir1);
137       //passageDist = 0;
138
```

```
139       // Calculate the velocity commands
140       vL = LeaderVelocity(xL, yL, obstDist1, obstAngle1, turnDirection1
              );
141       vF1 = FollowerVelocity(vL, xL, yL, thetaL, xF1, yF1, thetaF1,
              passageDist, -1, 1);
142       vF2 = FollowerVelocity(vL, xL, yL, thetaL, xF2, yF2, thetaF2,
              passageDist, 1, 2);
143
144       omegaL = LeaderAngVelocity(xL, yL, thetaL, omega_max, obstDist1,
              obstAngle1, turnDirection1);
145       omegaF1 = FollowerAngVelocity(vL, xL, yL, thetaL, xF1, yF1,
              thetaF1, passageDist, -1, 1);
146       omegaF2 = FollowerAngVelocity(vL, xL, yL, thetaL, xF2, yF2,
              thetaF2, passageDist, 1, 2);
147
148       // set the speed of the robot
149       pose1.SetSpeed(vL, omegaL);
150       //pose2.SetSpeed(vF1, omegaF1);
151       //pose3.SetSpeed(vF2, omegaF2);
152       //pose1.SetSpeed(vmax, 0);
153       pose2.SetSpeed(0, 0);
154       pose3.SetSpeed(0, 0);
155     }
156
157     // stop the vehicle by using a keyboard stroke
158     pose1.SetSpeed(0, 0);
159     pose2.SetSpeed(0, 0);
160     pose3.SetSpeed(0, 0);
161     cout << "robot stopped" <<endl;
162     sleep(100000);
163   }
164
165   catch (PlayerCc::PlayerError e)
166   {
167     std::cerr << e << std::endl;
168     return -1;
169   }
170
171   return 0;
172 }    // end of main
173
174
175 // ------------------------------------------------------
176
177
178 // FUNCTIONS
179
180
181 double PassageDistance (SonarProxy &sonar, SonarProxy &ir)
182 {
183   double passage = 0;
184
185   //if (sonar[0] < beamRadius && sonar[2] < beamRadius)
186     //passage = sonar[0]*sin(PI/4) + sonar[2]*sin(PI/4);
```

```
187
188   //if ((ir[4] < beamRadius && ir[6] < beamRadius))
189     //passage = ir[4] + ir[6];
190
191   if ((ir[4] < beamRadius || ir[6] < beamRadius))
192     cout << "IR detected at sides: ir[4] = " << ir[4] << ", ir[6] = "
          << ir[6] << endl;
193
194   if (ir[0] < beamRadius || ir[1] < beamRadius || ir[2] < beamRadius ||
        ir[3] < beamRadius)
195     cout << "IR detected at front: ir[0] = " << ir[0] << ", ir[1] = "
          << ir[1] << ", ir[2] = " << ir[2] << ", ir[3] = " << ir[3] <<
          endl;
196
197   //else if (sonar[3] < beamRadius && sonar[5] < beamRadius)
198     //passage = sonar[3]*sin(PI/4) + sonar[5]*sin(PI/4);
199
200   if (passage < 0)
201     passage = 0;
202
203   //if (passage == 0)
204     //cout << "open space" << endl;
205
206   //if (passage != 0)
207     //cout << "passage detected - passage distance: " << passage <<
          endl;
208
209   return (passage);
210 }
211
212
213 double FollowerVelocity (double vL, double xL, double yL, double thetaL
      , double xF, double yF, double thetaF,
214     double passageDist, int whichSide, int robotPosition)
215 {
216   double rho, alpha, psi;
217   double vF, err_rho, rho_d;
218
219   rho = sqrt( (xL - xF)*(xL - xF) + (yL - yF)*(yL - yF) );
220   alpha = atan( (yL-yF)/(xL-xF) ) - thetaF;
221   psi = alpha + thetaF - thetaL + PI;
222
223   if (passageDist != 0 || alpha_desired == 0)
224     rho_d = rho_desired * robotPosition;
225   else
226     rho_d = rho_desired;
227
228   err_rho = rho - rho_d;
229
230   vF = (K_p/(1 + K_d*cos(alpha))) * err_rho - (cos(psi)/cos(alpha)) *
        vL;
231
232   return (vF);
233 }
```

```
234
235
236 double FollowerAngVelocity (double vL, double xL, double yL, double
        thetaL, double xF, double yF,
237     double thetaF, double passageDist, int whichSide, int robotPosition
            )
238 {
239   double rho, alpha, psi;
240   double omegaF, err_rho, err_alpha, alpha_d, rho_d;
241
242   rho = sqrt( (xL - xF)*(xL - xF) + (yL - yF)*(yL - yF) );
243   alpha = atan( (yL-yF)/(xL-xF) ) - thetaF;
244   psi = alpha + thetaF - thetaL + PI;
245
246   if (passageDist != 0 || alpha_desired == 0)
247   {
248     alpha_d = alpha_desired * whichSide;
249     rho_d = rho_desired * robotPosition;
250     //alpha_d = 0;
251       //rho_d = rho_desired;
252   }
253   else
254   {
255     alpha_d = alpha_desired * whichSide;
256     rho_d = rho_desired;
257   }
258
259   err_rho = rho - rho_d;
260   err_alpha = alpha - alpha_d;
261
262   omegaF = (K_p * tan(alpha) / rho) * err_rho + (K_p / (1 + K_d)) *
        err_alpha +
263       (1 + K_d) * ((-cos(psi) * tan(alpha) + sin(psi)) / rho) * vL;
264
265   return (omegaF);
266 }
267
268 void CheckObstacle (double &obstDist, double &obstAngle, int &
        turnDirection, SonarProxy &sonar, SonarProxy &ir,double theta)
269 {
270   double minDist = obstRadius, minAngle;
271   int sonarMin = 0, irMin = 0;
272
273   // sonar sensors
274   for(int i=0; i<=2; i++)
275   {
276     if(sonar[i] < minDist && sonar[i] > 0)
277     {
278       minDist = sonar[i];
279       sonarMin = i+1;
280       //cout << "sonar reading: " << i+1 << endl;
281     }
282   }
283
```

```
284   // ir sensors
285   for(int j=0; j<=6; j++)
286   {
287     if(ir[j] < minDist && ir[j] > 0)
288     {
289       minDist = ir[j];
290       irMin = j+1;
291       sonarMin = 0;
292       //cout << "ir reading: " << j+1 << endl;
293     }
294   }
295
296   //get the min angle
297   if (sonarMin > 0)
298   {
299     switch (sonarMin)
300     {
301       case 1:
302       {
303         minAngle = theta - dtor(45);
304         turnDirection = -1;
305         break;
306       }
307       case 2:
308       {
309         minAngle = theta;
310         turnDirection = 1;
311         break;
312       }
313       case 3:
314       {
315         minAngle = theta - dtor(45);
316         turnDirection = 1;
317         break;
318       }
319       case 4:
320       {
321         minAngle = theta - dtor(135);
322         turnDirection = 1;
323         break;
324       }
325       case 5:
326       {
327         //minAngle = theta - dtor(180);
328         //turnDirection = 1;
329         minDist = obstRadius;
330         break;
331       }
332       case 6:
333       {
334         minAngle = theta - dtor(135);
335         turnDirection = 1;
336         break;
337       }
```

```
338     }
339     //cout << "Sonar Angle: " << minAngle << endl;
340   }
341   else if (irMin > 0)
342   {
343     switch (irMin)
344     {
345       case 1:
346       {
347         minAngle = theta - dtor(40);
348         turnDirection = -1;
349         break;
350       }
351       case 2:
352       {
353         minAngle = theta - dtor(10);
354         turnDirection = -1;
355         break;
356       }
357       case 3:
358       {
359         minAngle = theta - dtor(10);
360         turnDirection = 1;
361         break;
362       }
363       case 4:
364       {
365         minAngle = theta - dtor(40);
366         turnDirection = 1;
367         break;
368       }
369       case 5:
370       {
371         minAngle = theta - dtor(90);
372         turnDirection = 1;
373         break;
374       }
375       case 6:
376       {
377         //minAngle = theta - dtor(180);
378         //turnDirection = 1;
379         minDist = obstRadius;
380         break;
381       }
382       case 7:
383       {
384         minAngle = theta - dtor(90);
385         turnDirection = -1;
386         break;
387       }
388     }
389     //cout << "IR Angle: " << rtod(minAngle) << endl;
390   }
391
```

```
392   obstDist = 0;
393   if (minDist < obstRadius && minDist > 0)
394   {
395     obstDist = minDist;
396     obstAngle = minAngle;
397     if(obstAngle < 0)
398       obstAngle = obstAngle + 2*PI;
399   }
400 }
401
402 double LeaderVelocity (double x, double y, double obstDist, double
      obstAngle, int turnDirection)
403 {
404   // distance to goal
405   double delta_x = xGoal - x;
406   double delta_y = yGoal - y;
407   double distGoal = sqrt( delta_x*delta_x + delta_y*delta_y );
408   double thetaCalc = atan(delta_y/delta_x);
409   double distRelation = exp(-distGoal/goalRadius);
410
411   // constants
412   double k_a = 1, k_n = 1, k_t = 1;
413
414   // attractive velocity
415   double u_a = k_a * vmax * (1-distRelation);
416
417   // repulsive velocity
418   double normal, tangent, obstRelation, normalAngle, tangentAngle;
419   if (obstDist == 0)
420   {
421     normal = 0, tangent = 0;
422     obstRelation = 0;
423     normalAngle = 0, tangentAngle = 0;
424   }
425
426   else
427   {
428     normal = 0.5/(obstDist), tangent = 0.5/(obstDist);
429     obstRelation = exp(-obstDist/obstRadius);
430
431     u_a = 0;
432
433     // normal angle
434     normalAngle = obstAngle + PI;
435     if (normalAngle > 2*PI)
436       normalAngle = normalAngle - 2*PI;
437
438     // tangent angle
439     tangentAngle = normalAngle - (PI/2 *turnDirection);
440   }
441
442   // normal and tangent velocities
443   double u_n = k_n * normal * obstRelation;
444   double u_t = k_t * tangent * obstRelation;
```

```
445
446   // sum of velocity terms
447   double v, vx, vy;
448   vx = u_a*cos(thetaCalc) + u_n*cos(normalAngle) + u_t*cos(tangentAngle
          );
449   vy = u_a*sin(thetaCalc) + u_n*sin(normalAngle) + u_t*sin(tangentAngle
          );
450   v = sqrt(vx*vx + vy*vy);
451   if (v > vmax)
452     v = vmax;
453   else if (v < -vmax)
454     v = -vmax;
455
456   return (v);
457 }
458
459 double LeaderAngVelocity (double x, double y, double theta, double
       omega_max, double obstDist, double obstAngle, int turnDirection)
460 {
461   double omega_a, omega_o = 0, omega;
462   double k_a = 1, k_o = 1;
463
464   // distance to goal
465   double delta_x = xGoal - x;
466   double delta_y = yGoal - y;
467   double thetaCalc = atan(delta_y/delta_x);
468   double delta_theta = thetaCalc - theta;
469
470   // attractive angular velocity
471   omega_a = k_a * delta_theta/t;
472   if (delta_theta >= omega_max)
473     omega_a = omega_max;
474   else if (delta_theta < - omega_max)
475     omega_a = - omega_max;
476
477   // repulsive velocity
478   double normalAngle, tangentAngle;
479   if (obstDist == 0)
480     normalAngle = 0, tangentAngle = 0;
481
482   else
483   {
484     // normal angle
485     normalAngle = obstAngle + PI;
486     if (normalAngle > 2*PI)
487       normalAngle = normalAngle - 2*PI;
488
489     // tangent angle
490     tangentAngle = normalAngle - (PI/2 * turnDirection);
491
492     // get the resultant angle
493     double thetaObst = (normalAngle + tangentAngle) / 2;
494     if (thetaObst > 2*PI)
495       thetaObst = thetaObst - 2*PI;
```

```
496     double delta_thetaObst = (thetaObst - theta) * turnDirection;
497
498     // repulsive velocity
499     omega_o = k_o * delta_thetaObst/t;
500     if (delta_thetaObst >= omega_max)
501       omega_o = omega_max;
502     else if (delta_thetaObst < - omega_max)
503       omega_o = - omega_max;
504
505     // set omega_a = 0
506     omega_a = 0;
507
508   }
509
510   omega = omega_a + omega_o;
511
512   return (omega);
513 }
514
515
516
517 void UnexpectedShutdown (void)
518 {
519   sigShutdown = 0;
520   sigset_t blockmask;
521   struct sigaction sa;
522   sigfillset(&blockmask);
523   sa.sa_handler = HandleShutdownSignal;
524   sa.sa_mask = blockmask;
525   sa.sa_flags = 0;
526   sigaction(SIGINT, &sa, NULL);
527   sigaction(SIGQUIT, &sa, NULL);
528 }
529
530 void HandleShutdownSignal (int sig __attribute__ ((unused)))
531 {
532    sigShutdown = 1;
533 }
```

Appendix K

How to Make a C++ Program to Automatically Generate Multi-Robot Configuration Files for Player/Stage

K.1 AutoConfig.cc

```
1 //============================================
2 // Name        : AutoConfig.cpp
3 // Author      :
4 // Version     :
5 // Copyright   :
6 // Description :
7 //============================================
8
9 #include <iostream>
10 #include <cstdlib>
11 #include <string>
12 #include <fstream>
13 #include <sstream>
14 #include <ctime>
15
16 using namespace std;
17
18 //Function Prototypes
19 void MakeConfigFile(int);
20 void MakeWorldFile(int);
21
22
23 int main()
24 {
25   // how many robots?
26   int numRobots = 50;
27
28   cout << endl << "------ Creating the new configuration file" << endl;
```

```
29   cout << "number of robots: " << numRobots << endl;
30
31   srand((int) time(0));
32
33   // functions that make configuration and world files
34   MakeConfigFile(numRobots);
35   MakeWorldFile(numRobots);
36
37   return 0;
38 }
39
40
41 void MakeConfigFile(int numRobots)
42 {
43   int port = 6665;
44
45   for (int i = 1; i <= numRobots; i++)
46   {
47     int currentPort = port + (i-1);
48     ostringstream portToString;
49     portToString << currentPort;
50     string portNumber = portToString.str();
51
52     ostringstream intToString;
53     intToString << i;
54     string robotName = "robot" + intToString.str();
55
56     string fileName = "/home/elisha/Documents/Player/configs/
         AutoConfigFiles/";
57     fileName.append(robotName + ".cfg");
58     ofstream myConfig(fileName.c_str());
59
60     if (!myConfig)
61       cout << endl << "error opening file" << endl << endl;
62
63     string stageDriver = "# STAGE DRIVER ------";
64     stageDriver.append("\n\ndriver\n(\n\tname \"stage\"\n\tprovides [\"
         simulation:0\"]\n\tplugin \"stageplugin\"");
65     stageDriver.append("\n\tworldfile \"AutoWorldFiles/" + robotName +
         ".world\"\n)\n\n");
66     stageDriver.append("# X80 DRIVERS ------");
67
68     myConfig << stageDriver << endl;
69
70     for (int j = 1; j <= i; j++)
71     {
72       int currentPort = port + (j-1);
73       ostringstream portToString;
74       portToString << currentPort;
75       string portNumber = portToString.str();
76
77       ostringstream intToString;
78       intToString << j;
79       string robotName = "robot" + intToString.str();
```

```
80
81        string x80Driver = "\ndriver\n(\n\tname \"stage\"\n\tprovides [\"
              ";
82        x80Driver.append(portNumber + ":position2d:0\" \"" + portNumber +
              ":sonar:0\" \"" + portNumber + ":sonar:1\"]");
83        x80Driver.append("\n\tmodel \"" + robotName + "\"\n)");
84
85        myConfig << x80Driver << endl;
86      }
87
88      myConfig.close();
89    }
90 }
91
92 void MakeWorldFile(int numRobots)
93 {
94   int yWindowSize = 700;
95   int xWindowSize = 2 * yWindowSize;
96   int scale = 50;
97
98   ostringstream scaleToString;
99   scaleToString << scale;
100  string scaleString = scaleToString.str();
101
102  ostringstream xWindowToString;
103  xWindowToString << xWindowSize;
104  string xWindowString = xWindowToString.str();
105
106  ostringstream yWindowToString;
107  yWindowToString << yWindowSize;
108  string yWindowString = yWindowToString.str();
109
110  ostringstream xWindowScaleToString;
111  xWindowScaleToString << (xWindowSize/scale-4);
112  string xWindowScale = xWindowScaleToString.str();
113
114  ostringstream yWindowScaleToString;
115  yWindowScaleToString << (yWindowSize/scale-2);
116  string yWindowScale = yWindowScaleToString.str();
117
118  // Randomly select positions of follower robots
119  int xPose[numRobots-1];
120  int yPose[numRobots-1];
121  int yawPose[numRobots-1];
122
123  for (int k = 0; k <= (numRobots-2); k++)
124  {
125    xPose[k] = - rand() % (xWindowSize/150 - 4) - 6;
126    yPose[k] = - rand() % (yWindowSize/75) + 4;
127    yawPose[k] = 0;
128
129    cout << "xPose: " << xPose[k] << "\t yPose: " << yPose[k] << endl;
130
131    // Check to see that all robot positions are unique
```

```
132     if (numRobots > 2)
133     {
134       for (int check1 = 0; check1 <= (k-1); check1++)
135       {
136         if (xPose[k]==xPose[check1] && yPose[k]==yPose[check1])
137         {
138           cout << "Two robots in the same spot! Find new position.." <<
                endl;
139           xPose[k] = - rand() % (xWindowSize/150 - 4) - 6;
140           yPose[k] = - rand() % (yWindowSize/100) + 4;
141
142           cout << "New xPose: " << xPose[k] << "\t New yPose: " <<
                yPose[k] << endl;
143         }
144       }
145       // Second check
146       for (int check2 = 0; check2 <= (k-1); check2++)
147       {
148         if (xPose[k]==xPose[check2] && yPose[k]==yPose[check2])
149         {
150           cout << "Two robots in the same spot again! Find new position
                .." << endl;
151           xPose[k] = - rand() % (xWindowSize/150 - 4) - 6;
152           yPose[k] = - rand() % (yWindowSize/100) + 4;
153
154           cout << "Second New xPose: " << xPose[k] << "\t Second New
                yPose: " << yPose[k] << endl;
155         }
156       }
157     }
158   }
159
160   // MAKE THE FILE
161   for (int i = 1; i <= numRobots; i++)
162   {
163     ostringstream robotToString;
164     robotToString << i;
165     string robotName = "robot" + robotToString.str();
166
167     string fileName = "/home/elisha/Documents/Player/configs/
            AutoConfigFiles/AutoWorldFiles/";
168     fileName.append(robotName + ".world");
169     ofstream myWorld(fileName.c_str());
170
171     if (!myWorld)
172       cout << endl << "error opening file" << endl << endl;
173
174     string include = "include \"IncludeFiles/map.inc\"\ninclude \"
            IncludeFiles/X80.inc\"\n";
175     myWorld << include << endl;
176
177     string map = "# OPEN ENVIRONMENT FILE -----\n";
178     map.append("map\n(\n\tbitmap \"IncludeFiles/bitmap/warehouse.png\""
            );
```

```
179     map.append("\n\tsize [" + xWindowScale + " " + yWindowScale + " 1]"
            );
180     map.append("\n\torigin [0 0 0 0]\n)\n");
181     myWorld << map << endl;
182
183     string addGoal = "# ADD GOAL POSITION IN THE TERRAIN -----\n";
184     addGoal.append("rect_goal(name \"goal\" pose [8 2 0 0])\n\n");
185     myWorld << addGoal << endl;
186
187
188     // LEADER ROBOT
189     string leaderPose =  "# ADD ROBOTS TO THE MAP -----\n";
190     leaderPose.append("X80\n(\n\tname \"robot1\"\n\tpose [-5 0 0 0]\n\
            tcolor \"red\"\n)\n\n");
191     myWorld << leaderPose << endl;
192
193
194     // FOLLOWER ROBOTS
195     for (int j = 2; j <= i; j++)
196     {
197       ostringstream intToString;
198       intToString << j;
199       string robotName = "robot" + intToString.str();
200
201       ostringstream xPoseToString;
202       xPoseToString << xPose[j-2];
203       string xPoseString = xPoseToString.str();
204
205       ostringstream yPoseToString;
206       yPoseToString << yPose[j-2];
207       string yPoseString = yPoseToString.str();
208
209       ostringstream yawPoseToString;
210       yawPoseToString << yawPose[j-2];
211       string yawPoseString = yawPoseToString.str();
212
213
214       // PICK RANDOM COLOR FOR ROBOT
215       ifstream colorFile ("/home/elisha/Documents/Player/configs/
              AutoConfigFiles/AutoWorldFiles/IncludeFiles/color.txt");
216       if (!colorFile)
217         cout << endl << "error opening color file" << endl << endl;
218
219       string colorName;
220       int desiredLine = rand() % 550;
221       int currentLine = 0;
222       while ( getline(colorFile, colorName) )
223       {
224         if (currentLine == desiredLine)
225           break;
226         currentLine++;
227       }
228
229       // MAKE LINES IN THE FILE
```

```
230         string followerPose = "X80\n(\n\tname \"";
231         followerPose.append(robotName);
232         followerPose.append("\"\n\tpose [" + xPoseString + " " +
                yPoseString + " 0 " + yawPoseString + "]");
233         followerPose.append("\n\tcolor \"" + colorName + "\"\n)\n\n");
234
235         myWorld << followerPose << endl;
236       }
237
238
239       string guiWindow = "# CONFIGURE GUI WINDOW -----\n";
240       guiWindow.append("window\n(\n\tsize [" + xWindowString + " " +
              yWindowString + "]\n\t");
241       guiWindow.append("scale " + scaleString + "\n\tcenter [0 0]\n\
              trotate [0 0]\n\n\t");
242       guiWindow.append("show_data 1 \n\t");
243       guiWindow.append("show_flags 0 \n\t");
244       guiWindow.append("show_blocks 1 \n\t");
245       guiWindow.append("show_clock 0 \n\t");
246       guiWindow.append("show_footprints 0 \n\t");
247       guiWindow.append("show_grid 1 \n\t");
248       guiWindow.append("show_trailarrows 0 \n\t");
249       guiWindow.append("show_trailrise 0 \n\t");
250       guiWindow.append("show_trailfast 0 \n\t");
251       guiWindow.append("show_occupancy 0 \n\t");
252       guiWindow.append("pcam_on 0 \n\t");
253       guiWindow.append("screenshots 0 \n");
254       myWorld << guiWindow << endl;
255
256
257       myWorld.close();
258     }
259 }
260
261
262 /*
263       string addStart = "# ADD START POSITION IN THE TERRAIN -----\n";
264       addStart.append("rect_goal(name \"goal\" pose [-8 -0 0 0])\n\n");
265       myWorld << addStart << endl;
266
267
268       string addObst = "# ADD OBSTACLE IN THE TERRAIN -----\n";
269       addObst.append("rect_obst(name \"box\" pose [1 2 0 0])\n");
270       addObst.append("rect_obst(name \"box\" pose [1 -2 0 0])\n");
271       addObst.append("rect_obst2(name \"box\" pose [7 2 0 0])\n");
272       addObst.append("rect_obst2(name \"box\" pose [7 -2 0 0])\n\n");
273       myWorld << addObst << endl;
274 */
```

Appendix L

Self-Organizing Formation Controller Code

L.1 SelfOrganizing.cpp

```
1 //============================================================
2 // Name        : SelfOrganizing.cpp
3 // Author      :
4 // Version     :
5 // Copyright   :
6 // Description :
7 //============================================================
8
9 #include <iostream>
10 #include <csignal>
11
12 #include "libplayerc++/playerc++.h"
13 #include "RobotParameters.h"
14 #include "CheckObstacles.h"
15 #include "CalcSpeed.h"
16
17 using namespace std;
18 using namespace PlayerCc;
19
20 // Function prototypes
21 volatile sig_atomic_t sigShutdown;
22 void UnexpectedShutdown (void);
23 void HandleShutdownSignal(int);
24
25
26 // MAIN FUNCTION
27
28 int main()
29 {
30   try
31   {
32     int numRobots;
33     int port = 6665;
```

```
34      double xGoal = 8, yGoal = 2;
35      double vmax = 0.1, omega_max = 1;
36      double close_to_zero = 0.5;
37
38      cout << endl << endl << "executing new program...." << endl;
39      cout << "enter the number of robots: " << endl;
40      cin >> numRobots;
41
42
43      // Declare a pointer variable called robot
44      RobotParameters* robot[numRobots];
45      for (int i = 0; i <= (numRobots-1); i++)
46      {
47        // set robot constructor
48        robot[i] = new RobotParameters(port+i);
49      }
50
51      // Handle unexpected shutdown
52      UnexpectedShutdown();
53
54      while (sigShutdown == 0)
55      {
56        for (int i = 0; i <= (numRobots-1); i++)
57        {
58          // get robot's current state
59          robot[i]->getCurrentState();
60
61          // if a robot reaches the goal stop all robots
62          if ( ((xGoal - robot[i]->xPose) <= close_to_zero && (yGoal -
                robot[i]->yPose) <= close_to_zero) )
63          {
64            robot[i]->pose->SetSpeed(0, 0);
65            robot[i]->~RobotParameters();
66            break;
67          }
68
69          // Get sensor values
70          robot[i]->getSensorValues();
71          CheckObstacles obst(robot[i]->sonarNum, robot[i]->sonarDist,
                robot[i]->irNum, robot[i]->irDist,
72            robot[i]->thetaPose);
73
74          // Get velocity and angular velocity
75          CalcSpeed robotSpeed(robot[i]->xPose, robot[i]->yPose, robot[i
                ]->thetaPose, xGoal, yGoal,
76            vmax, omega_max, obst.obstDist, obst.obstAngle, obst.
                turnDirection);
77
78          // Set the motor speed
79          robot[i]->pose->SetSpeed(robotSpeed.v, robotSpeed.omega);
80        }
81      }
82      sleep(5);
83    }
```

```
84
85    catch (PlayerCc::PlayerError e)
86    {
87      std::cerr << e << std::endl;
88      return -1;
89    }
90
91    return 0;
92  }
93
94  //-------------------------------------------------------------------
95
96  void UnexpectedShutdown (void)
97  {
98    sigShutdown = 0;
99    sigset_t blockmask;
100   struct sigaction sa;
101   sigfillset(&blockmask);
102   sa.sa_handler = HandleShutdownSignal;
103   sa.sa_mask = blockmask;
104   sa.sa_flags = 0;
105   sigaction(SIGINT, &sa, NULL);
106   sigaction(SIGQUIT, &sa, NULL);
107 }
108
109 void HandleShutdownSignal (int sig __attribute__ ((unused)))
110 {
111    sigShutdown = 1;
112 }
```

L.2 RobotParameters.h

```
1  #include <iostream>
2  #include <cstring>
3  #include "libplayerc++/playerc++.h"
4
5  using namespace PlayerCc;
6
7  class RobotParameters
8  {
9  public:
10
11   // Constructor
12   RobotParameters(int port);
13   ~RobotParameters();
14
15   // Method prototypes
16   void getCurrentState();
17   void getSensorValues();
18
19   // Player proxy pointers
20   PlayerClient* client;
21   Position2dProxy* pose;
```

```
22   SonarProxy *sonar;
23   SonarProxy *ir;
24
25   // Position variables
26   double xPose;
27   double yPose;
28   double thetaPose;
29
30   // Sensor variables
31   double static const sonarMax = 2.55;
32   double static const irMax = 1.0;
33   int sonarNum;
34   int irNum;
35   double sonarDist;
36   double irDist;
37 };
```

L.3 RobotParameters.cpp

```
1 #include "RobotParameters.h"
2
3 RobotParameters::RobotParameters(int port)
4 {
5   client = new PlayerClient("localhost", port);
6   pose = new Position2dProxy(client, 0);
7   sonar = new SonarProxy(client, 0);
8   ir = new SonarProxy(client, 1);
9 }
10
11
12 RobotParameters::~RobotParameters()
13 {
14   delete [] client;
15   delete [] pose;
16   delete [] sonar;
17   delete [] ir;
18 }
19
20 void RobotParameters::getCurrentState()
21 {
22   client->Read();
23   xPose = pose->GetXPos();
24   yPose = pose->GetYPos();
25   thetaPose = pose->GetYaw();
26 }
27
28 void RobotParameters:: getSensorValues()
29 {
30   // Initialize
31   sonarDist = 0;
32   irDist = 0;
33   sonarNum = 0;
34   irNum = 0;
```

```
35
36    // Read data from robot
37    client->Read();
38
39    // Go through sonar sensors
40    for (int k = 0; k < 3; k++)
41    {
42      if(sonar->GetScan(k) < sonarMax && sonar->GetScan(k) > 0)
43      {
44        sonarDist = sonar->GetScan(k);
45        sonarNum = k+1;
46      }
47    }
48
49    // Go through ir sensors
50    for (int k = 0; k < 7; k++)
51    {
52      if(ir->GetScan(k) < irMax && ir->GetScan(k) > 0)
53      {
54        irDist = ir->GetScan(k);
55        irNum = k+1;
56      }
57    }
58 }
```

L.4 CheckObstacles.h

```
1 #include <iostream>
2 #include "libplayerc++/playerc++.h"
3
4 using namespace std;
5 using namespace PlayerCc;
6
7 class CheckObstacles
8 {
9 public:
10   // Constructor
11   CheckObstacles(int sonarNum, double sonarDist, int irNum, double
         irDist, double theta);
12   ~CheckObstacles();
13
14   // Methods
15   void getSonarAngle(int sonarNum, double theta);
16   void getIrAngle(int irNum, double theta);
17
18   // Variables
19   double static const PI = 3.14159;
20   double obstDist;
21   double obstAngle;
22   double minAngle;
23   double minDist;
24   int turnDirection;
25 };
```

L.5 CheckObstacles.cpp

```
1 #include "CheckObstacles.h"
2
3 CheckObstacles::CheckObstacles(int sonarNum, double sonarDist, int
    irNum, double irDist, double theta)
4 {
5   // Initialize
6   minDist = 0;
7   obstDist = 0;
8   obstAngle = 0;
9
10  // If there is a sensor reading
11  if (sonarNum > 0 || irNum > 0)
12  {
13    if (sonarDist <= irDist)
14    {
15      minDist = sonarDist;
16      getSonarAngle(sonarNum, theta);
17    }
18
19    else
20    {
21      minDist = irDist;
22      getIrAngle(irNum, theta);
23    }
24  }
25
26  if (minDist > 0)
27  {
28    obstDist = minDist;
29    obstAngle = minAngle;
30
31    if(obstAngle < 0)
32      obstAngle = obstAngle + 2*PI;
33  }
34 }
35
36 CheckObstacles::~CheckObstacles()
37 {
38
39 }
40
41 void CheckObstacles::getSonarAngle(int sonarNum, double theta)
42 {
43  switch (sonarNum)
44  {
45    case 1:
46    {
47      minAngle = theta - dtor(45);
48      turnDirection = -1;
49      break;
50    }
51    case 2:
```

```
52      {
53        minAngle = theta;
54        turnDirection = 1;
55        //minDist = 0;
56        break;
57      }
58      case 3:
59      {
60        minAngle = theta - dtor(45);
61        turnDirection = 1;
62        break;
63      }
64      case 4:
65      {
66        minAngle = theta - dtor(135);
67        turnDirection = 1;
68        break;
69      }
70      case 5:
71      {
72        //minAngle = theta - dtor(180);
73        //turnDirection = 1;
74        minDist = 0;
75        break;
76      }
77      case 6:
78      {
79        minAngle = theta - dtor(135);
80        turnDirection = -1;
81        break;
82      }
83    }
84  }
85
86  void CheckObstacles::getIrAngle(int irNum, double theta)
87  {
88    switch (irNum)
89    {
90      case 1:
91      {
92        minAngle = theta - dtor(40);
93        turnDirection = -1;
94        break;
95      }
96      case 2:
97      {
98        minAngle = theta - dtor(10);
99        turnDirection = -1;
100       break;
101     }
102     case 3:
103     {
104       minAngle = theta - dtor(10);
105       turnDirection = 1;
```

```
106         break;
107       }
108       case 4:
109       {
110         minAngle = theta - dtor(40);
111         turnDirection = 1;
112         break;
113       }
114       case 5:
115       {
116         minAngle = theta - dtor(90);
117         turnDirection = 1;
118         break;
119       }
120       case 6:
121       {
122         //minAngle = theta - dtor(180);
123         //turnDirection = 1;
124         minDist = 0;
125         break;
126       }
127       case 7:
128       {
129         minAngle = theta - dtor(90);
130         turnDirection = -1;
131         break;
132       }
133     }
134 }
```

L.6 CalcSpeed.h

```
1 #include <iostream>
2 #include "libplayerc++/playerc++.h"
3
4 using namespace std;
5 using namespace PlayerCc;
6
7 class CalcSpeed
8 {
9 public:
10   // Constructor
11   CalcSpeed(double x, double y, double theta, double xGoal, double
         yGoal, double vmax, double omega_max,
12         double obstDist, double obstAngle, int turnDirection);
13   ~CalcSpeed();
14
15   // Variables
16   double v;
17   double omega;
18
19   // Constants
20   double static const PI = 3.14159;
```

```
21   double static const sonarMax = 2.55;
22   double static const irMax = 0.8;
23
24   const static double k_a = 0.5;
25   const static double k_o = 0.01;
26   const static double k_n = 0.01;
27   const static double k_t = 0.01;
28 };
```

L.7 CalcSpeed.cpp

```
1 #include "CalcSpeed.h"
2
3 CalcSpeed::CalcSpeed(double x, double y, double theta, double xGoal,
      double yGoal, double vmax, double omega_max,
4     double obstDist, double obstAngle, int turnDirection)
5 {
6   // Variables
7   double obstRadius = irMax;
8   double goalRadius = irMax;
9   double t = 1;
10
11  double u_a = 0, u_n = 0, u_t = 0;
12  double omega_a = 0, omega_o = 0;
13
14
15 /* ---------------------------------------------------------------- */
16
17
18  // Calculate distance and angle variables
19  double delta_x = xGoal - x;
20  double delta_y = yGoal - y;
21  double distGoal = sqrt( delta_x*delta_x + delta_y*delta_y );
22  double thetaCalc = atan(delta_y/delta_x);
23  double delta_theta = thetaCalc - theta;
24
25
26 /* ---------------------------------------------------------------- */
27
28
29  // calculate the attractive velocity
30  double distRelation = exp(-distGoal/goalRadius);
31  u_a = k_a * vmax * (1-distRelation);
32
33  omega_a = (k_a * delta_theta * (1-distRelation)) / (10*t);
34  if (omega_a >= omega_max)
35    omega_a = omega_max;
36  else if (omega_a < - omega_max)
37    omega_a = - omega_max;
38
39 /* ---------------------------------------------------------------- */
40
41
```

```
42   // repulsive velocity
43   double normal, tangent, normalAngle, tangentAngle;
44   double obstRelation = exp(-obstDist/obstRadius);
45
46   if (obstDist == 0)
47   {
48     normal = 0, tangent = 0;
49     obstRelation = 0;
50     normalAngle = 0, tangentAngle = 0;
51
52     // all velocities
53     u_a = k_a * vmax * (1-distRelation);
54     u_n = 0;
55     u_t = 0;
56     omega_o = 0;
57   }
58
59   else
60   {
61     normal = 1/obstDist, tangent = 1/obstDist;
62
63     // normal angle
64     normalAngle = obstAngle + PI;
65     if (normalAngle > 2*PI)
66       normalAngle = normalAngle - 2*PI;
67
68     // tangent angle
69     tangentAngle = normalAngle - (PI/2 *turnDirection);
70
71     // get the resultant angle
72     double thetaObst = (normalAngle + tangentAngle) / 2;
73     if (thetaObst > 2*PI)
74       thetaObst = thetaObst - 2*PI;
75     double delta_thetaObst = (thetaObst - theta) * turnDirection;
76
77     // repulsive velocity
78     omega_o = (k_o * delta_thetaObst * obstRelation) / (2*t);
79     if (omega_o >= omega_max)
80       omega_o = omega_max;
81     else if (omega_o < - omega_max)
82       omega_o = - omega_max;
83
84     // all velocities
85     u_a = 0;
86     u_n = k_n * normal * obstRelation;
87     u_t = k_t * tangent * obstRelation;
88
89     omega_a = 0;
90   }
91
92
93
94 /* ------------------------------------------------------------ */
95
```

```
96
97    // sum of velocity terms
98    double vx, vy;
99    vx = u_a*cos(thetaCalc) + u_n*cos(normalAngle) + u_t*cos(tangentAngle
          );
100   vy = u_a*sin(thetaCalc) + u_n*sin(normalAngle) + u_t*sin(tangentAngle
          );
101
102   v = sqrt(vx*vx + vy*vy);
103
104   if (v > vmax)
105     v = vmax;
106   else if (v < -vmax)
107     v = -vmax;
108
109   // angular velocity terms
110   omega = omega_a + omega_o;
111 }
112
113 CalcSpeed::~CalcSpeed()
114 {
115
116 }
```

Appendix M

How to Check the Sensor Readings on the X80 Robot

M.1 X80Test.cc

```
1
2 /*
3  * X80Test.cc
4  *
5  *  Created on: 2010-02-08
6  *      Author: lbrunet
7  *
8  *
9  *      This is a test program for the X80_Driver.
10 *      It registers callbacks and can print out data that it receives
11 *      It can also be used to test the sending of commands to the
       vehicle
12 */
13
14 #include "X80_Driver.h"
15
16 #include <stdio.h>
17 #include <unistd.h>
18
19
20 /********************** X80 User Callbacks ************/
21
22 void
23 SENSOR_DATA_Callback(X80_Message *message, void *userData)
24 {
25   X80_Sensor_Data *data;
26
27   data = (X80_Sensor_Data *) message->data;
28
29   printf("Sonars: %i, %i, %i, %i, %i, %i -- IR: %i\n",
30       data->sonar1, data->sonar2, data->sonar3,
31       data->sonar4, data->sonar5, data->sonar6,
32       data->infraRedValue);
```

```
33
34    printf("Temp: %d, BAT: %d, tiltX: %d, tiltY: %d\n",
35        data->temperature, data->mainBAT,
36        data->tiltX, data->tiltY);
37 }
38
39 void
40 MOTOR_DATA_Callback(X80_Message *message, void *userData)
41 {
42    X80_Motor_Control_Data *data;
43
44    double vLeft = 0.0, vRight = 0.0;
45    double xLeft = 0.0, xRight = 0.0;
46
47    data = (X80_Motor_Control_Data *) message->data;
48
49    /* Convert encoder angular rate to linear wheel velocities*/
50    vLeft = PI * X80_WHEEL_DIAMETER * ((float)data->encoder1Speed / (
          float)X80_WHEEL_TICKS_PER_REV);
51    vRight = PI * X80_WHEEL_DIAMETER * ((float)data->encoder2Speed / (
          float)X80_WHEEL_TICKS_PER_REV);
52
53    /* Convert encoder angular position to linear wheel position */
54    xLeft = PI * X80_WHEEL_DIAMETER * ((float)data->encoder1Pulse / (
          float)X80_WHEEL_TICKS_PER_REV);
55    xRight = PI * X80_WHEEL_DIAMETER * ((float)data->encoder2Pulse / (
          float)X80_WHEEL_TICKS_PER_REV);
56
57    if((data->encoderDirection & (1 << 0)) == 0)
58    {
59      vLeft = -vLeft;
60    }
61
62    // right encoder goes in the reverse direction
63    if((data->encoderDirection & (1 << 1)) == (1 << 1))
64    {
65      vRight = -vRight;
66    }
67    printf("Dir: %d, xLeft: %.5f xRight: %.5f vLeft: %.5f vRight: %.5f\n"
          ,
68        data->encoderDirection, xLeft, xRight, vLeft, vRight);
69 }
70
71 void
72 CUSTOM_DATA_Callback(X80_Message *message, void *userData)
73 {
74    X80_Custom_Data *data;
75
76    data = (X80_Custom_Data *) message->data;
77
78    printf("IR: %i, %i, %i, %i, %i, %i\n",
79        data->adChannel3,data->adChannel4, data->adChannel5,
80        data->adChannel6, data->adChannel7, data->adChannel8);
81
```

```
82    fflush(stdout);
83 }
84
85 void
86 ACK_DATA_Callback(X80_Message *message, void *userData)
87 {
88    X80_Custom_Data *data;
89
90    data = (X80_Custom_Data *) message->data;
91
92    printf("ACK Received\n");
93    fflush(stdout);
94 }
95
96 int
97 main(int argc, char **argv)
98 {
99    X80_Driver *driver;
100
101   /* Create driver */
102   driver = new X80_Driver();
103   if(driver == NULL)
104   {
105      printf("Error Creating Driver\n");
106      return -1;
107   }
108
109   /* Connect to the stream
110    * If a serial connection is desired, comment out the UDP code below
         and
111    * uncomment serial connection code found below it
112    */
113   char test[] = "192.168.0.205";
114   if(driver->ConnectUDP(test, 10001) < 0)
115   {
116      printf("Error connecting to UDP socket\n");
117      delete driver;
118      return -1;
119   }
120
121   /* Use this to connect to through the Serial Port */
122   /*char port[] = "/dev/ttyS0";
123   if(driver->ConnectSerial(port) < 0)
124   {
125      printf("Error connecting to serial port\n");
126      delete driver;
127      return -1;
128   }*/
129
130   /* Register Callbacks
131    * Can comment whatever callbacks you do not want to see printed to
132    * the screen
133    */
```

```
134   driver->RegisterCallback(X80_GET_SENSOR_FEEDBACK, &
          SENSOR_DATA_Callback, NULL);
135   // driver->RegisterCallback(X80_GET_MOTOR_CONTROL_SIGNAL, &
          MOTOR_DATA_Callback, NULL);
136   // driver->RegisterCallback(X80_GET_CUSTOM, &CUSTOM_DATA_Callback,
          NULL);
137   // driver->RegisterCallback(X80_SET_SYSTEM_COMMS, &ACK_DATA_Callback,
           NULL);
138
139   /* Make sure platform is stopped at startup -- safety*/
140   driver->StopPlatform();
141   driver->Run(1000);
142
143   if(driver->RequestAllSensorDataBegin() < 0)
144   {
145     return -1;
146   }
147
148   /*
149    * This will loop such that the user can verify the data printed to
           the screen
150    * If you want to control the platform wheels, comment out the
           infinite loop
151    * and let the remaining code execute
152    *
153    */
154   while(true)
155   {
156     driver->PingPMS5005();
157     usleep(10000);
158   }
159
160   /* This should run for 2 seconds */
161   for(int i=0; i<200; i++)
162   {
163     usleep(10000);
164   }
165
166   /* Now play with the wheels */
167
168   /*
169    * PWM Control
170    */
171   /* Set Control Sensor */
172   driver->SetControlSensor(X80_WHEEL_LEFT, X80_DOUBLE_POTENTIOMETER);
173   driver->SetControlSensor(X80_WHEEL_RIGHT, X80_DOUBLE_POTENTIOMETER);
174
175   /* Set control method */
176   driver->SetControlMethod(X80_WHEEL_LEFT, X80_CONTROL_PWM);
177   driver->SetControlMethod(X80_WHEEL_RIGHT, X80_CONTROL_PWM);
178
179   /* Set for wheel polarity */
180   driver->SetWheelPolarity(X80_WHEEL_LEFT,X80_WHEEL_POSITIVE);
181   driver->SetWheelPolarity(X80_WHEEL_RIGHT,X80_WHEEL_POSITIVE);
```

```
182
183   /* Command the platform
184    * vRight is negative because the encoder goes backwards on that side
185    */
186   uint16_t val = 20000;
187   driver->SetMotorPWM(X80_WHEEL_LEFT, val);
188   driver->SetMotorPWM(X80_WHEEL_RIGHT, -val);
189
190   /*
191    * Velocity Control
192    */
193
194   /*driver->SetControlMethod(X80_WHEEL_LEFT, X80_CONTROL_VELOCITY);
195   driver->SetControlMethod(X80_WHEEL_RIGHT, X80_CONTROL_VELOCITY);
196
197   driver->SetWheelPolarity(X80_WHEEL_LEFT,X80_WHEEL_POSITIVE);
198   driver->SetWheelPolarity(X80_WHEEL_RIGHT,X80_WHEEL_POSITIVE);
199
200   driver->SetVelocityPIDGains(X80_WHEEL_LEFT, 30, 10, 0);
201   driver->SetVelocityPIDGains(X80_WHEEL_RIGHT, 30, 10, 0);
202
203   double speed = 0.2, turn = 0;
204
205   double vLeft = speed - X80_MIN_TURN_RADIUS * turn;
206   double vRight = speed + X80_MIN_TURN_RADIUS * turn;
207
208   driver->SetMotorSpeed(X80_WHEEL_LEFT, vLeft);
209   driver->SetMotorSpeed(X80_WHEEL_RIGHT, -vRight);*/
210
211
212   /* Run for 1 second */
213   usleep(2000000);
214
215
216   /* Cleanup */
217   driver->StopPlatform();
218   driver->Stop();
219   driver->Disconnect();
220   delete driver;
221 }
```

M.2 X80_Constants.h

```
1  /*
2   * X80_Constants.h
3   *
4   * This header file contains all the constants that are needed
5   * for the Dr. Robot X80 platform driver
6   *
7   *  Created on: 2010-02-08
8   *      Author: lbrunet
9   */
10
11 #ifndef X80_CONSTANTS_H_
12 #define X80_CONSTANTS_H_
13
14 // these are used to help parse out messages
15 #define STX1       0x5E
16 #define STX2       0x02
17 #define ETX1       0x5E
18 #define ETX2       0x0D
19
20 // serial baud rate
21 #define BAUD       B115200
22
23 // integer constants
24 #define X80_MAX_MSG_LEN    512;
25 #define X80_CONST_MSG_BYTES 9
26 #define X80_MAX_CALLBACKS 4
27 #define X80_MSG_WAIT_PERIOD 4000
28
29 // platform specs
30 #define X80_WHEEL_DIAMETER     0.17       // meters
31 #define X80_MIN_TURN_RADIUS    0.155        // meters - distance from the
       wheel to the center of rotation
32 #define X80_MAX_ENCODER_TICKS 32767      // number of ticks until the
      counter restarts
33 #define X80_WHEEL_TICKS_PER_REV 800        // number of wheel ticks in 1
       revolution
34
35 // sensor constants
36 #define X80_AD_MULTIPLIER    (3.0 / 4095.0)  // converts AD value to
      voltage
37 #define X80_SONAR_MULTIPLIER  1.0 / 100.0    // converts cm to meters
38 #define X80_NUM_SONAR      6
39 #define X80_NUM_IR         7
40
41 // math constants
42 #define PI  3.14159
43
44 typedef enum _RobotID
45 {
46   X80_HOST_ID    = 0,
47   X80_PMS5005_ID = 1,
48   X80_PMB5010_ID = 8,
```

```
49 } X80_RobotID;
50
51 typedef enum _DataID
52 {
53   X80_MOVE_MOTOR_TO_TARGET      = 3,
54   X80_MOVE_MOTOR_TO_TARGET_ALL  = 4,
55   X80_MOVE_MOTOR_WITH_PWM       = 5,
56   X80_MOVE_MOTOR_WITH_PWM_ALL   = 6,
57   X80_SYSTEM_PARAMS             = 7,
58   X80_SEND_TO_POWER_CONTROLLER  = 22,
59   X80_CONTROL_LCD               = 23,
60   X80_SET_MOTOR_VELOCITY        = 26,
61   X80_SET_MOTOR_VELOCITY_ALL    = 27,
62   X80_MOVE_SERVO_TO_TARGET      = 28,
63   X80_MOVE_SERVO_TO_TARGET_ALL  = 29,
64   X80_SUSPEND_RESUME            = 30,
65   X80_SET_CONST_SENSOR_ID       = 80,
66   X80_GET_MOTOR_CONTROL_SIGNAL  = 123,
67   X80_GET_CUSTOM                = 124,
68   X80_GET_SENSOR_FEEDBACK       = 125,
69   X80_GET_SENSOR_FEEDBACK_ALL   = 127,
70   X80_SET_SYSTEM_COMMS          = 255,
71 } X80_DataID;
72
73 typedef enum _SystemParams
74 {
75   X80_MOTOR_POLARITY    = 0x06,
76   X80_PID_POSITION      = 0x07,
77   X80_PID_VELOCITY      = 0x08,
78   X80_SENSOR_USEAGE     = 0x0d,
79   X80_CONTROL_METHOD    = 0x0e,
80 } X80_SystemParams;
81
82 typedef enum _GainConstants
83 {
84   X80_KP    = 0x01,
85   X80_KD    = 0x02,
86   X80_KI    = 0x03,
87 } X80_GainConstants;
88
89 typedef enum _SensorUseage
90 {
91   X80_SINGLE_POTENTIOMETER  = 0x00,
92   X80_DOUBLE_POTENTIOMETER  = 0x01,
93   X80_QUAD_ENCODER          = 0x02,
94 } X80_SensorUseage;
95
96 typedef enum _ControlMethod
97 {
98   X80_CONTROL_PWM       = 0x00,
99   X80_CONTROL_POSITION  = 0x01,
100  X80_CONTROL_VELOCITY  = 0x02,
101 } X80_ControlMethod;
102
```

```
103 typedef enum _WheelIndex
104 {
105   X80_WHEEL_LEFT  = 0,
106   X80_WHEEL_RIGHT = 1,
107 }X80_WheelIndex;
108
109 typedef enum _WheelPolarity
110 {
111   X80_WHEEL_POSITIVE  = 1,
112   X80_WHEEL_NEGATIVE  = -1,
113 }X80_WheelPolarity;
114
115 typedef enum _ConnectionType
116 {
117   X80_CONNECT_UDP    = 0,
118   X80_CONNECT_SERIAL  = 1,
119 }X80_ConnectionType;
120
121 #endif /* X80_CONSTANTS_H_ */
```

M.3 X80_Message_Types.h

```
1  /*
2   * X80_Message_Types.h
3   *
4   *  Created on: 2010-02-08
5   *      Author: lbrunet
6   */
7
8  #ifndef X80_MESSAGE_TYPES_H_
9  #define X80_MESSAGE_TYPES_H_
10
11 #include <sys/types.h>
12 #include <stdint.h>
13
14 ////////////////////////////////////////////////
15
16 typedef struct _X80_Message_Header
17 {
18   uint8_t   stx[2];   // start transmission
19   uint8_t   robot_id; // sender id
20   uint8_t   reserved; // reserved byte --> set to 0
21   uint8_t   data_id;  // id of the data packet contained in the message
22   uint8_t   length;   // length of the data in the message
23 } __attribute__ ((packed)) X80_Message_Header;
24
25 typedef struct _X80_Message_Footer
26 {
27   uint8_t   checksum;   // message checksum
28   uint8_t   etx[2];     // end of transmission
29 } __attribute__ ((packed)) X80_Message_Footer;
30
31
32 /*
33  * Message Structure for the X80 Motor Data
34  */
35 typedef struct _X80_Motor_Control_Data
36 {
37   uint16_t potentiometer1;
38   uint16_t potentiometer2;
39   uint16_t potentiometer3;
40   uint16_t potentiometer4;
41   uint16_t potentiometer5;
42   uint16_t potentiometer6;
43
44   uint16_t currentMotor1;
45   uint16_t currentMotor2;
46   uint16_t currentMotor3;
47   uint16_t currentMotor4;
48   uint16_t currentMotor5;
49   uint16_t currentMotor6;
50
51   uint16_t encoder1Pulse;
52   uint16_t encoder1Speed;
```

```
53   uint16_t encoder2Pulse;
54   uint16_t encoder2Speed;
55   uint8_t  encoderDirection;
56 } __attribute__ ((packed)) X80_Motor_Control_Data;
57
58
59 /*
60  * Message Structure for the X80 Custom Data
61  * A/D channels 2-8 are currently connected to IR Sensors 1-7
62  */
63 typedef struct _X80_Custom_Data
64 {
65   uint16_t  adChannel1;
66   uint16_t  adChannel2;
67   uint16_t  adChannel3;
68   uint16_t  adChannel4;
69   uint16_t  adChannel5;
70   uint16_t  adChannel6;
71   uint16_t  adChannel7;
72   uint16_t  adChannel8;
73
74   uint8_t   inputIOPort;
75
76   uint16_t  distLeftID1;
77   uint16_t  distLeftID2;
78   uint16_t  distLeftID3;
79   uint16_t  distLeftID4;
80   uint16_t  distRightID1;
81   uint16_t  distRightID2;
82   uint16_t  distRightID3;
83   uint16_t  distRightID4;
84
85   uint8_t   idTranspID1;
86   uint8_t   idTranspID2;
87   uint8_t   idTranspID3;
88   uint8_t   idTranspID4;
89 } __attribute__ ((packed)) X80_Custom_Data;
90
91
92 /*
93  * Message Structure for the X80 Sensor Data
94  */
95 typedef struct _X80_Sensor_Data
96 {
97   uint8_t sonar1;
98   uint8_t sonar2;
99   uint8_t sonar3;
100  uint8_t sonar4;
101  uint8_t sonar5;
102  uint8_t sonar6;
103
104  uint16_t humanSensorAlarm1;
105  uint16_t humanSensorDetect1;
106  uint16_t humanSensorAlarm2;
```

```
107     uint16_t humanSensorDetect2;
108
109     uint16_t tiltX;
110     uint16_t tiltY;
111
112     uint16_t overheatMotor1;
113     uint16_t overheatMotor2;
114
115     uint16_t temperature;
116     uint16_t infraRedValue;
117     uint32_t infraRedCommand;
118     uint16_t mainBAT;
119     uint16_t dcBAT;
120     uint16_t servoBAT;
121     uint16_t refVoltage;
122     uint16_t potDetectVoltage;
123
124 } __attribute__ ((packed)) X80_Sensor_Data;
125
126 #endif /* X80_MESSAGE_TYPES_H_ */
```

M.4 X80_Driver.h

```
1 /*
2  * X80_Driver.h
3  *
4  *  Created on: 2010-02-05
5  *      Author: lbrunet
6  */
7
8 #ifndef X80_DRIVER_H_
9 #define X80_DRIVER_H_
10
11 #include "X80_Message_Types.h"
12 #include "X80_Constants.h"
13 #include "X80_Serial.h"
14 #include "X80_Udp.h"
15
16 #include <pthread.h>
17
18 /*
19  * X80 Message structure
20  */
21 typedef struct _X80_Message
22 {
23   X80_Message_Header  header;   // header for the message
24   void          *data;    // message specific data
25   X80_Message_Footer  footer;   // message ending
26 } X80_Message;
27
28 /*
29  * Structure to hold callback resistrations
30  */
31 typedef struct _X80_Message_Register
32 {
33    uint8_t messageId;          // ID of the message
34    void (*callback) (X80_Message * message, void *data); // Callback
       called when message received
35    void *callbackData;
36 } X80_Message_Register;
37
38 ////////////////////////////////////////////////
39
40 // The X80 device driver class.
41 class X80_Driver
42 {
43   public:
44
45     // Constructor
46     X80_Driver();
47     ~X80_Driver();
48
49     /* Method to register functions to handle incoming data */
50     int RegisterCallback(int messageID, void (*callback) (X80_Message
       *, void *), void *userData);
```

```
51
52      /* Connection Methods*/
53      int ConnectSerial(const char* port);
54      int ConnectUDP(const char* address, int port);
55      int Disconnect();
56
57      /* Threads to read messages and run user callbacks*/
58      int Run(int timeout);
59    int Stop();
60
61    /* Internal threads --> do not call from main program */
62    static void *StartThread(void *arg)
63    {
64      reinterpret_cast < X80_Driver * >(arg)->rcvThread();
65      return NULL;
66    }
67    void rcvThread();
68
69      /* Data Request Messages*/
70    int RequestAllSensorDataBegin();
71    int RequestAllSensorDataStop();
72      int RequestSensorDataBegin();
73      int RequestSensorDataStop();
74      int RequestMotorDataBegin();
75      int RequestMotorDataStop();
76      int RequestCustomDataBegin();
77      int RequestCustomDataStop();
78      int PingPMS5005();
79
80      /* Platform Motion Methods */
81      int SetMotorSpeed(X80_WheelIndex index, double val);
82      int SetMotorPWM(X80_WheelIndex index, uint16_t val);
83      int StopPlatform();
84      int SetWheelPolarity(X80_WheelIndex index, X80_WheelPolarity
            polarity);
85      int SetControlSensor(X80_WheelIndex index, X80_SensorUseage sensor)
            ;
86      int SetControlMethod(X80_WheelIndex index, X80_ControlMethod method
            );
87
88      /* Controller Parameters*/
89      int SetVelocityPIDGains(X80_WheelIndex index, uint16_t kp, uint16_t
            kd, uint16_t kix100);
90
91    private:
92
93      X80_Serial  *serial;     // serial connection object
94      X80_Udp   *udp;      // udp connection object
95      int     readTimeout;  // timeout to wait for incoming data
96
97      X80_ConnectionType connectionType;  // serial or UDP
98
99      /* Registered message callbacks */
100     X80_Message_Register MessageCallbacks[X80_MAX_CALLBACKS];
```

```
101     int callbackCount;
102
103     /* pthread data*/
104     pthread_t receiveThreadId;
105     pthread_mutex_t lock;
106     bool threadRunning;
107
108     /* Sends a data stream to the X80 Platform*/
109     int SendMessage(X80_RobotID robot_id, X80_DataID dataID, uint8_t *
            data, uint8_t length);
110
111     /* Calculates the CRC of an X80 Message */
112     uint8_t CalculateCRC(uint8_t start, uint8_t *data, int len);
113
114 };
115
116 #endif /* X80_DRIVER_H_ */
```

M.5 X80_Driver.cc

```
1 /*
2  * X80_Driver.cc
3  *
4  *  Created on: 2010-02-05
5  *      Author: lbrunet
6  */
7
8
9 #include "X80_Driver.h"
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <termios.h>
14 #include <fcntl.h>
15 #include <errno.h>
16 #include <string.h>
17 #include <unistd.h>
18 #include <sys/types.h>
19
20 // TODO: Add wait for ACK to return when sending a command to the X80
21
22 /********************** Constructors ********************/
23 /*
24  * Constructor for the X80 Driver
25  * initializes class data
26  */
27 X80_Driver::X80_Driver()
28 {
29   serial = NULL;
30   udp = NULL;
31
32   callbackCount = 0;
33   connectionType = X80_CONNECT_UDP; // UDP is the default
34
35   threadRunning  = false;
36   receiveThreadId = -1;
37
38   pthread_mutex_init(&this->lock, NULL);
39 }
40
41 /*
42  * Destructor for the X80 Driver
43  * disconnects from the platform and cleans up any memory that was
       allocated
44  */
45 X80_Driver::~X80_Driver()
46 {
47   /* Ensure that we are properly disconnected */
48   this->Disconnect();
49
50   if(serial != NULL)
51   {
```

```
52      delete serial;
53    }
54
55    if(udp != NULL)
56    {
57      delete udp;
58    }
59
60    serial = NULL;
61    udp = NULL;
62  }
63
64  /************************ Connection Routines ***************/
65
66  /*
67   * Closes the platform's data stream
68   *
69   * return: 1 on success
70   */
71  int
72  X80_Driver::Disconnect()
73  {
74
75    /* Stop the receiving thread first */
76    this->Stop();
77
78    /* Disconnect from the robot*/
79    switch(connectionType)
80    {
81      case X80_CONNECT_UDP:
82        if(udp != NULL)
83        {
84          udp->Disconnect();
85          delete udp;
86        }
87        break;
88      case X80_CONNECT_SERIAL:
89        if(serial != NULL)
90        {
91          serial->Disconnect();
92          delete serial;
93        }
94        break;
95      default:
96        break;
97    }
98
99    serial = NULL;
100   udp = NULL;
101
102   return 1;
103 }
104
105 /*
```

```
106  * Connects to the X80 via UDP
107  *
108  * const char* address: ip address of the platform (ex:
       "192.168.0.202")
109  * int port: socket to listen and write to (ex: 10001)
110  *
111  * return: 1 on success, -1 on failure
112  */
113 int
114 X80_Driver::ConnectUDP(const char* address, int port)
115 {
116   udp = new X80_Udp();
117   if(udp == NULL)
118   {
119     printf("%s: Unable to create UDP Connection object\n", __func__);
120     return -1;
121   }
122   if(udp->Connect(address, port) < 0)
123   {
124     delete udp;
125     return -1;
126   }
127
128   connectionType = X80_CONNECT_UDP;
129   return 1;
130 }
131
132 /*
133  * Connects to the X80 via Serial
134  *
135  * const char* port: the serial port that we are connecting to (ex: "/
       dev/ttyS0")
136  *
137  * return: 1 on success, -1 on failure
138  */
139 int
140 X80_Driver::ConnectSerial(const char* port)
141 {
142   serial = new X80_Serial();
143   if(serial == NULL)
144   {
145     printf("%s: Unable to create Serial Connection object\n", __func__)
           ;
146     return -1;
147   }
148   if(serial->Connect(port) < 0)
149   {
150     delete serial;
151     return -1;
152   }
153
154   connectionType = X80_CONNECT_SERIAL;
155   return 1;
156 }
```

```
157
158 /************************ Read and Write Methods
    ***********************/
159
160 /*
161  * Calculates the checksum over all bytes in an X80 message
162  * excluding the STX and ETX
163  *
164  * uint8_t start: what the shift register should start at (0 for the
     start of the CRC)
165  *            this enables partial CRC calculation, with the rest
     calculated with
166  *          a second call
167  * uint8_t *data: array of bytes in the message
168  * int len: number of bytes to be included in the CRC calculation
169  *
170  * return: the calculated CRC value
171  */
172 uint8_t
173 X80_Driver::CalculateCRC(uint8_t start, uint8_t *data, int len)
174 {
175   uint8_t shift_reg, v, data_bit, sr_lsb, fb_bit;
176   int i, j;
177
178   /*CRC calculation given by Dr. Robot (PMS5005 protocol) */
179
180   // start crc calculation
181   shift_reg = start; // initialize the shift register
182   v = 0;
183   data_bit = 0;
184   sr_lsb = 0;
185   fb_bit = 0;
186
187   for(i=0; i<len; i++)
188   {
189     v = (uint8_t) (data[i] & 0x0000FFFF);
190
191     for (j = 0 ; j < 8 ; j++) // for each bit
192     {
193       data_bit = v & 0x01; // isolate least sign bit
194       sr_lsb = shift_reg & 0x01;
195       fb_bit = (data_bit ^ sr_lsb) & 0x01; // calculate the feed back
          bit
196       shift_reg = shift_reg >> 1;
197       if (fb_bit == 1)
198       {
199          shift_reg = shift_reg ^ 0x8C;
200       }
201
202       v = v >> 1;
203     }
204   }
205
206   return shift_reg;
```

```
207 }
208
209 /*
210  * Sends a message to the X80 platform by taking an array of bytes and
       adding an
211  * STX, the calculated CRC, and an ETX
212  *
213  * X80_RobotID robot: destination of the message (see X80_Constants.h
       for possible values)
214  * X80_DataID dataID: data payload type (see X80_Constants.h for
       possible values)
215  * uint8_t *data: array of bytes to be sent to the platform
216  * uint8_t len: the number of bytes to be transmitted in the message
       payload
217  *
218  * return: message length on success, -1 on failure
219  */
220 int
221 X80_Driver::SendMessage(X80_RobotID robotID, X80_DataID dataID, uint8_t
       *data, uint8_t len)
222 {
223   X80_Message_Header header;
224   X80_Message_Footer footer;
225   uint8_t *bytes;
226
227   /* Create header */
228   header.stx[0] = 0;          // STX = 0 in order to exclude it from the
       CRC
229   header.stx[1] = 0;          // will add the proper STX later
230   header.robot_id = robotID;     // destination
231   header.reserved = 0;        // always
232   header.data_id = dataID;
233   header.length = len;
234
235   // calculate the checksum over entire message excluding STX and ETX
236   footer.checksum = this->CalculateCRC(0, (uint8_t *)&header, sizeof(
       X80_Message_Header));
237   footer.checksum = this->CalculateCRC(footer.checksum, data, header.
       length);
238
239   /* Add the proper STX and ETX now that the CRC has been calculated */
240   footer.etx[0] = ETX1;
241   footer.etx[1] = ETX2;
242   header.stx[0] = STX1;
243   header.stx[1] = STX2;
244
245   /* Copy message into a buffer */
246   bytes = (uint8_t *)malloc(len+X80_CONST_MSG_BYTES);
247   memcpy(&bytes[0], &header, sizeof(X80_Message_Header));
248   memcpy(&bytes[sizeof(X80_Message_Header)], data, len);
249   memcpy(&bytes[sizeof(X80_Message_Header)+len], &footer, sizeof(
       X80_Message_Footer));
250
251   /* Write data to data stream */
```

```
252   switch(connectionType)
253   {
254   case X80_CONNECT_UDP:
255   {
256     if(udp->WriteData(bytes, X80_CONST_MSG_BYTES+len) < 0)
257     {
258       return -1;
259     }
260     break;
261   }
262   case X80_CONNECT_SERIAL:
263   {
264     if(serial->WriteData(bytes, X80_CONST_MSG_BYTES+len) < 0)
265     {
266       return -1;
267     }
268     break;
269   }
270   default:
271   {
272     printf("%s:Write failed -- Unknown Connection Type\n", __func__);
273     return -1;
274   }
275   }
276
277   free(bytes);
278
279   #ifdef X80_DEBUG
280     printf("%s: Sent message: ID %d Length %d CRC %d\n", __func__,
          dataID, len, footer.checksum);
281   #endif
282
283   return X80_CONST_MSG_BYTES+len;
284 }
285
286 /*
287  * Starts the receive thread -- messages will continuously be read and
        the user callbacks
288  * will be called where applicable
289  *
290  * int timeout: time in ms to wait for new data on the data stream
291  *
292  * return: 1 on success, -1 on failure
293  */
294 int
295 X80_Driver::Run(int timeout)
296 {
297   /* Run thread to receive X80 Messages */
298   this->readTimeout = timeout;
299   if (this->threadRunning == false)
300   {
301     this->threadRunning = true;
302     pthread_create(&this->receiveThreadId, NULL, &X80_Driver::
          StartThread, this);
```

```
303     return 1;
304   }
305
306   return -1;
307 }
308
309 /*
310  * Stops the receive thread -- messages will no longer be read from
311  * the data stream
312  *
313  * return: 1 on success, -1 on failure
314  */
315 int
316 X80_Driver::Stop()
317 {
318   /* Stop the receiving thread from running */
319   if (this->threadRunning == true)
320   {
321     this->threadRunning = false;
322     pthread_join(this->receiveThreadId, NULL);
323
324     return 1;
325   }
326
327   return -1;
328 }
329
330 /*
331  * Thread that continuously reads X80 messages from the data stream and
332  * if a message is received, the callback registry is checked and
       called if a
333  * matching callback has been found
334  */
335 void
336 X80_Driver::rcvThread()
337 {
338   int pos = 0, left = 0;
339   int maxSize = X80_MAX_MSG_LEN;
340   uint8_t buf[maxSize], crc;
341
342   while(this->threadRunning)
343   {
344     pthread_mutex_lock(&this->lock);
345
346     /* Read bytes for an entire X80 message */
347     switch(connectionType)
348     {
349       case X80_CONNECT_UDP:
350         if(!udp->isConnected())
351         {
352           printf("%s: udp not connected\n", __func__);
353           pthread_mutex_unlock(&this->lock);
354           this->threadRunning = false;
355           return;
```

```
356         }
357         left = udp->ReadData((uint8_t *)buf, maxSize, this->readTimeout
                );
358         break;
359       case X80_CONNECT_SERIAL:
360         if(!serial->isConnected())
361         {
362           printf("%s: serial not connected\n", __func__);
363           pthread_mutex_unlock(&this->lock);
364           this->threadRunning = false;
365           return;
366         }
367         left = serial->ReadData((uint8_t *)buf, maxSize, this->
                readTimeout);
368         break;
369       default:
370         printf("%s: Unknown Connection Type\n", __func__);
371         pthread_mutex_unlock(&this->lock);
372         this->threadRunning = false;
373         break;
374     }
375
376     /* Process bytes */
377     pos = 0;
378     while(left >= X80_CONST_MSG_BYTES)
379     {
380       X80_Message msg;
381
382       msg.header.stx[0] = 0; // STX = 0 to start for checksum
383       msg.header.stx[1] = 0; // will add the actual STX afterwards
384       msg.header.robot_id = buf[pos+2];
385       msg.header.reserved = buf[pos+3];
386       msg.header.data_id = buf[pos+4];
387       msg.header.length = buf[pos+5];
388
389       /* Check if the message length is correct */
390       if(left < X80_CONST_MSG_BYTES+msg.header.length)
391       {
392         printf("Invalid msg size -- %i instead of %i\n",
393             left, X80_CONST_MSG_BYTES+msg.header.length);
394         break;
395       }
396
397       // copy the message payload
398       msg.data = malloc(msg.header.length);
399       memcpy((uint8_t *)msg.data, &buf[pos+6], msg.header.length);
400
401       // get the checksum
402       msg.footer.checksum = buf[pos+6+msg.header.length];
403
404       // verify checksum
405       crc = this->CalculateCRC(0, (uint8_t *)&msg.header, sizeof(
              X80_Message_Header));
```

```
406         crc = this->CalculateCRC(crc, (uint8_t *)msg.data, msg.header.
                length);
407         if(crc == msg.footer.checksum)
408         {
409           // add proper STX and ETX
410           msg.footer.etx[0] = ETX1;
411           msg.footer.etx[1] = ETX2;
412           msg.header.stx[0] = STX1;
413           msg.header.stx[1] = STX2;
414
415           // find the appropriate user callback to issue
416           for(int i=0; i<callbackCount; i++)
417           {
418             if(MessageCallbacks[i].messageId == msg.header.data_id)
419             {
420               MessageCallbacks[i].callback(&msg, MessageCallbacks[i].
                    callbackData);
421               break;
422             }
423           }
424
425           #ifdef X80_DEBUG
426             printf("%s: Received message ID %d Length %d CRC %d\n",
427                 __func__, msg.header.data_id, msg.header.length, msg.
                      footer.checksum);
428           #endif
429         }
430         else
431         {
432           printf("Invalid checksum (%d, %d) on Read - id: %i\n", crc, msg
                .footer.checksum, msg.header.data_id);
433         }
434         free(msg.data);
435         pos += (X80_CONST_MSG_BYTES+msg.header.length);
436         left -= (X80_CONST_MSG_BYTES+msg.header.length);
437       }
438
439       pthread_mutex_unlock(&this->lock);
440       usleep(10000);
441     }
442 }
443
444 /************************ Callback Registration ******************/
445
446 /*
447  * This function registers a callback for the user, and adds a pointer
        to user data
448  * that will be added as a parameter in the callback
449  *
450  * int id: message id that the callback is for
451  * void (*callback) (X80_Message *, void *): pointer to the callback
452  * void* data: user data to be passed when the callback is called
453  *
454  * return: 1 on success, -1 on failure
```

```
455  */
456 int
457 X80_Driver::RegisterCallback(int id, void (*callback)(X80_Message *,
        void *), void* data)
458 {
459   int i;
460
461   // check for a NULL callback
462   if(callback == NULL)
463   {
464     printf("%s: NULL Callback\n", __func__);
465     return -1;
466   }
467
468   /* Can only have one callback for every message (4 possible right now
        ) */
469   if(callbackCount >= X80_MAX_CALLBACKS)
470   {
471     printf("%s: MAX Callbacks have been exceeded\n", __func__);
472     return -1;
473   }
474
475   // see if the callback has already been issued
476   for(i=0; i<callbackCount; i++)
477   {
478     if(MessageCallbacks[i].messageId == id)
479     {
480       // just update the callback
481       MessageCallbacks[i].callback = callback;
482       MessageCallbacks[i].callbackData = data;
483       return 1;
484     }
485   }
486
487   callbackCount++;
488   MessageCallbacks[i].messageId = id;
489   MessageCallbacks[i].callback = callback;
490   MessageCallbacks[i].callbackData = data;
491   return 1;
492 }
493
494 /************************ X80 Specific Messages ********************/
495
496 /*
497  * Sends a message to the X80 requesting that it continuously send
498  * all available data (CUSTOM, MOTOR, SENSOR)
499  *
500  * return: 1 on success, -1 on failure
501  */
502 int
503 X80_Driver::RequestAllSensorDataBegin()
504 {
505   if(this->SendMessage(X80_PMS5005_ID, X80_GET_SENSOR_FEEDBACK_ALL,
        NULL, 0) < 0)
```

```
506   {
507     printf("%s: Error Sending Message\n", __func__);
508     return -1;
509   }
510
511   return 1;
512 }
513
514 /*
515  * Sends a message to the X80 requesting that it stop sending
516  * data (CUSTOM, MOTOR, SENSOR)
517  *
518  * return: 1 on success, -1 on failure
519  */
520 int
521 X80_Driver::RequestAllSensorDataStop()
522 {
523   uint8_t data = 0;
524
525   if(this->SendMessage(X80_PMS5005_ID, X80_GET_SENSOR_FEEDBACK_ALL, &
        data, 1) < 0)
526   {
527     printf("%s: Error Sending Message\n", __func__);
528     return -1;
529   }
530
531   return 1;
532 }
533
534 /*
535  * Sends a message to the X80 requesting that it continuously send
536  * SENSOR data
537  *
538  * return: 1 on success, -1 on failure
539  */
540 int
541 X80_Driver::RequestSensorDataBegin()
542 {
543   if(this->SendMessage(X80_PMS5005_ID, X80_GET_SENSOR_FEEDBACK, NULL,
        0) < 0)
544   {
545     printf("%s: Error Sending Message\n", __func__);
546     return -1;
547   }
548
549   return 1;
550 }
551
552 /*
553  * Sends a message to the X80 requesting that it stop sending
554  * SENSOR data
555  *
556  * return: 1 on success, -1 on failure
557  */
```

```
558 int
559 X80_Driver::RequestSensorDataStop()
560 {
561   uint8_t data = 0;
562
563   if(this->SendMessage(X80_PMS5005_ID, X80_GET_SENSOR_FEEDBACK, &data,
        1) < 0)
564   {
565     printf("%s: Error Sending Message\n", __func__);
566     return -1;
567   }
568
569   return 1;
570 }
571
572 /*
573  * Sends a message to the X80 requesting that it continuously send
574  * MOTOR data
575  *
576  * return: 1 on success, -1 on failure
577  */
578 int
579 X80_Driver::RequestMotorDataBegin()
580 {
581
582   if(this->SendMessage(X80_PMS5005_ID, X80_GET_MOTOR_CONTROL_SIGNAL,
        NULL, 0) < 0)
583   {
584     printf("%s: Error Sending Message\n", __func__);
585     return -1;
586   }
587
588   return 1;
589 }
590
591 /*
592  * Sends a message to the X80 requesting that it stop sending
593  * MOTOR data
594  *
595  * return: 1 on success, -1 on failure
596  */
597 int
598 X80_Driver::RequestMotorDataStop()
599 {
600   uint8_t data;
601
602   data = 0;
603   if(this->SendMessage(X80_PMS5005_ID, X80_GET_MOTOR_CONTROL_SIGNAL, &
        data, 1) < 0)
604   {
605     printf("%s: Error Sending Message\n", __func__);
606     return -1;
607   }
608
```

```
609   return 1;
610 }
611
612 /*
613  * Sends a message to the X80 requesting that it continuously send
614  * CUSTOM data
615  *
616  * return: 1 on success, -1 on failure
617  */
618 int
619 X80_Driver::RequestCustomDataBegin()
620 {
621   if(this->SendMessage(X80_PMS5005_ID, X80_GET_CUSTOM, NULL, 0) < 0)
622   {
623     printf("%s: Error Sending Message\n", __func__);
624     return -1;
625   }
626
627   return 1;
628 }
629
630 /*
631  * Sends a message to the X80 requesting that it stop sending
632  * CUSTOM data
633  *
634  * return: 1 on success, -1 on failure
635  */
636 int
637 X80_Driver::RequestCustomDataStop()
638 {
639   uint8_t data;
640
641   data = 0;
642   if(this->SendMessage(X80_PMS5005_ID, X80_GET_CUSTOM, &data, 1) < 0)
643   {
644     printf("%s: Error Sending Message\n", __func__);
645     return -1;
646   }
647
648   return 1;
649 }
650
651 /*
652  * Ping's the PMS5005 board
653  * must do this periodically or the board will stop transmitting data
654  * if it has not received a message from the host
655  *
656  * return: 1 on success, -1 on failure
657  */
658 int
659 X80_Driver::PingPMS5005()
660 {
661   uint8_t data;
662
```

```
663   data = 1;
664   if(this->SendMessage(X80_PMS5005_ID, X80_SET_SYSTEM_COMMS, &data, 1)
          < 0)
665   {
666     printf("%s: Error Sending Message\n", __func__);
667     return -1;
668   }
669
670   return 1;
671 }
672
673 /*
674  * Sets the velocity of the given wheel
675  *
676  * X80_WheelIndex index: wheel that we are commanding (left or right),
        see X80_Constants.h
677  * double speed: linear wheel speed in m/s
678  *
679  * return: 1 on success, -1 on failure
680  */
681 int
682 X80_Driver::SetMotorSpeed(X80_WheelIndex index, double speed)
683 {
684   uint8_t data[] = {0,0,0};
685   uint8_t val[2];
686   uint16_t encSpeed = 0;
687
688   /* Convert linear speed to encoder angular velocity */
689   encSpeed = (uint16_t)((speed  * (double)X80_WHEEL_TICKS_PER_REV) / (
          PI * (double)X80_WHEEL_DIAMETER));
690
691   data[0] = index;
692   memcpy(&val, (uint16_t *)&encSpeed, sizeof(uint16_t));
693
694   data[1] = val[0];
695   data[2] = val[1];
696
697   if(this->SendMessage(X80_PMS5005_ID, X80_SET_MOTOR_VELOCITY, (uint8_t
          *)&data, 3) < 0)
698   {
699     printf("%s: Error Sending Message\n", __func__);
700     return -1;
701   }
702
703   return 1;
704 }
705
706 /*
707  * Sends a PWM signal to the specified wheel.  See PMS5005 protocol
708  * documentation for information on PWM signal and values
709  *
710  * X80_WheelIndex index: wheel that we are commanding (left or right),
        see X80_Constants.h
711  * uint16_t val: PWM value
```

```
712  *
713  * return: 1 on success, -1 on failure
714  */
715 int
716 X80_Driver::SetMotorPWM(X80_WheelIndex index, uint16_t val)
717 {
718   uint8_t data[] = {0,0,0};
719   uint8_t bytes[2];
720
721   data[0] = index;
722   memcpy(&bytes, (uint16_t *)&val, sizeof(uint16_t));
723
724   data[1] = bytes[0];
725   data[2] = bytes[1];
726   if(this->SendMessage(X80_PMS5005_ID, X80_SET_MOTOR_VELOCITY, (uint8_t
          *)&data, 3) < 0)
727   {
728     printf("%s: Error Sending Message\n", __func__);
729     return -1;
730   }
731
732   return 1;
733 }
734
735 /*
736  * Commands the platform to stop moving and shuts off the motors
737  *
738  * return: 1 on success, -1 on failure
739  */
740 int
741 X80_Driver::StopPlatform()
742 {
743   uint8_t data;
744
745   data = 0;
746   if(this->SendMessage(X80_PMS5005_ID, X80_SUSPEND_RESUME, &data, 1) <
          0)
747   {
748     printf("%s: Error Sending Message\n", __func__);
749     return -1;
750   }
751
752   return 1;
753 }
754
755 /*
756  * Sets the wheel polarity, which is used to identify if the encoder
757  * on the specified wheel is increasing in a positive or negative
758  * direction.  See PMS5005 protocol documentation for more info
759  *
760  * X80_WheelIndex index: wheel that we are commanding (left or right),
          see X80_Constants.h
761  * X80_WheelPolarity polarity: polarity of the wheel(+ve or -ve), see
          X80_Constants.h
```

```
762  *
763  * return: 1 on success, -1 on failure
764  */
765 int
766 X80_Driver::SetWheelPolarity(X80_WheelIndex index, X80_WheelPolarity
       polarity)
767 {
768   uint8_t data[] = {0,0,0};
769
770   data[0] = X80_MOTOR_POLARITY;
771   data[1] = index;
772   data[2] = polarity;
773
774   if(this->SendMessage(X80_PMS5005_ID, X80_SYSTEM_PARAMS, (uint8_t *)
        data, 3) < 0)
775   {
776     printf("%s: Error Sending Message\n", __func__);
777     return -1;
778   }
779
780   return 1;
781 }
782
783 /*
784  * Sets the type of sensor used to control the wheel.  Each sensor type
        has
785  * a specified application.  For robot motion, the quadrature encoder
       or
786  * the double potentiometer should be used. See PMS5005 protocol
       documentation
787  * or X80 user manual for more info
788  *
789  * X80_WheelIndex index: wheel that we are commanding (left or right),
       see X80_Constants.h
790  * X80_SensorUseage sensor: sensor to be used, see X80_Constants.h
791  *
792  * return: 1 on success, -1 on failure
793  */
794 int
795 X80_Driver::SetControlSensor(X80_WheelIndex index, X80_SensorUseage
       sensor)
796 {
797   uint8_t data[] = {0,0,0};
798
799   data[0] = X80_SENSOR_USEAGE;
800   data[1] = index;
801   data[2] = sensor;
802   if(this->SendMessage(X80_PMS5005_ID, X80_SYSTEM_PARAMS, (uint8_t *)
        data, 3) < 0)
803   {
804     printf("%s: Error Sending Message\n", __func__);
805     return -1;
806   }
807
```

```
808   return 1;
809 }
810
811 /*
812  * Sets the control method for each wheel.  The value set depends on
       how the vehicle is commanded, either
813  * wheel position, wheel velocity, or wheel PWM.
814  * See PMS5005 protocol documentation or X80 user manual for more info
815  *
816  * X80_WheelIndex index: wheel that we are commanding (left or right),
       see X80_Constants.h
817  * X80_ControlMethod method: control method to be used, see
       X80_Constants.h
818  *
819  * return: 1 on success, -1 on failure
820  */
821 int
822 X80_Driver::SetControlMethod(X80_WheelIndex index, X80_ControlMethod
      method)
823 {
824   uint8_t data[] = {0,0,0};
825
826   data[0] = X80_CONTROL_METHOD;
827   data[1] = index;
828   data[2] = method;
829   if(this->SendMessage(X80_PMS5005_ID, X80_SYSTEM_PARAMS, (uint8_t *)
        data, 3) < 0)
830   {
831     printf("%s: Error Sending Message\n", __func__);
832     return -1;
833   }
834
835   return 1;
836 }
837
838 /*
839  * Sets the velocity PID gains for each wheel.
840  *
841  * X80_WheelIndex idx: wheel that we are commanding (left or right),
       see X80_Constants.h
842  * uint16_t kp: proportional gain
843  * uint16_t kd: derivative gain
844  * uint16_t kix100: integral gain, (will be divided by 100 when
       received by the controller)
845  *
846  * return: 1 on success, -1 on failure
847  */
848 int
849 X80_Driver::SetVelocityPIDGains(X80_WheelIndex idx, uint16_t kp,
      uint16_t kd, uint16_t kix100)
850 {
851   uint8_t data[] = {0,0,0,0,0};
852   uint8_t val[2];
853
```

```
854   data[0] = X80_PID_VELOCITY;
855   data[1] = idx;
856
857   /* Send P gain*/
858   data[2] = X80_KP;
859   memcpy(&val, (uint16_t *)&kp, sizeof(uint16_t));
860   data[3] = val[0];
861   data[4] = val[1];
862
863   if(this->SendMessage(X80_PMS5005_ID, X80_SYSTEM_PARAMS, (uint8_t *)
          data, 5) < 0)
864   {
865     printf("%s: Error Sending Message\n", __func__);
866     return -1;
867   }
868
869   /* Send D gain*/
870   data[2] = X80_KD;
871   memcpy(&val, (uint16_t *)&kd, sizeof(uint16_t));
872   data[3] = val[0];
873   data[4] = val[1];
874
875   if(this->SendMessage(X80_PMS5005_ID, X80_SYSTEM_PARAMS, (uint8_t *)
          data, 5) < 0)
876   {
877     printf("%s: Error Sending Message\n", __func__);
878     return -1;
879   }
880
881   /* Send I gain*/
882   data[2] = X80_KI;
883   memcpy(&val, (uint16_t *)&kix100, sizeof(uint16_t));
884   data[3] = val[0];
885   data[4] = val[1];
886
887   if(this->SendMessage(X80_PMS5005_ID, X80_SYSTEM_PARAMS, (uint8_t *)
          data, 5) < 0)
888   {
889     printf("%s: Error Sending Message\n", __func__);
890     return -1;
891   }
892
893   return 1;
894 }
```

M.6 X80_Serial.h

```
1 /*
2  * X80_Serial.h
3  *
4  *  Created on: 2010-02-11
5  *      Author: lbrunet
6  */
7
8 #ifndef X80_SERIAL_H_
9 #define X80_SERIAL_H_
10
11 #include <stdint.h>
12
13 // The X80 serial class.
14 class X80_Serial
15 {
16   public:
17
18     // Constructor
19     X80_Serial();
20     ~X80_Serial();
21
22     /* Connection Methods*/
23     int Connect(const char* port);
24     int Disconnect();
25     bool isConnected() {return (fd > 0) ? true:false;};
26
27     /* Read and Write X80 Messages in byte form*/
28     int ReadData(uint8_t *buf, int size, int timeout);  // reads entire
           x80 messages
29     int WriteData(uint8_t *buf, int nBytes);
30
31   private:
32
33     int fd;           // file descriptor for the com port
34
35     int readBytes(uint8_t *buf, int nBytes, int timeout); // reads a
          set number of bytes
36     int readChar(uint8_t c, int timeout);   // reads until a set char
          is found (returns 0 if not found)
37
38 };
39
40 #endif /* X80_SERIAL_H_ */
```

M.7 X80_Serial.cc

```
1 /*
2  * X80_Serial.cc
3  *
4  *  Created on: 2010-02-11
5  *      Author: lbrunet
6  */
7
8 #include "X80_Serial.h"
9 #include "X80_Constants.h"
10
11 #include <stdio.h>
12 #include <termios.h>
13 #include <errno.h>
14 #include <unistd.h>
15 #include <sys/types.h>
16 #include <fcntl.h>
17 #include <string.h>
18
19 /*
20  * Constructor for the X80_Serial object
21  * initializes class data
22  */
23 X80_Serial::X80_Serial()
24 {
25   fd = -1;
26 }
27
28 /*
29  * Destructor for the X80_Serial object
30  * disconnects from the platform and cleans up any memory that was
       allocated
31  */
32 X80_Serial::~X80_Serial()
33 {
34   if(fd > 0)
35   {
36     this->Disconnect();
37   }
38 }
39
40 /*
41  * Connects to the platform via a Serial Connection
42  *
43  * const char* port: the serial port that we are connecting to (ex: "/
       dev/ttyS0")
44  *
45  * return: 1 on success, -1 on failure
46  */
47 int
48 X80_Serial::Connect(const char* port)
49 {
50   struct termios tio;
```

```
51
52   /* Disconnect first just in case something went wrong previously*/
53   this->Disconnect();
54
55   /* open the file descriptor for the serial port */
56     fd = open(port, O_RDWR | O_NOCTTY | O_NDELAY);
57
58   if( fd < 0 )
59   {
60     printf("%s: Unable to open Serial port '%s' - %s", __func__, port,
         strerror(errno));
61     return -1;
62   }
63
64   /* get the serial port attributes so that we can set the proper baud
       rate */
65   if (tcgetattr(fd, &tio) < 0)
66   {
67     printf("%s: Unable to get Serial port attributes - %s", __func__,
         strerror(errno));
68     this->Disconnect();
69     return -1;
70   }
71
72   //115200kbps, no parity bit, 8 data bits, 1 stop bit
73   cfmakeraw(&tio);
74   cfsetospeed(&tio, BAUD);
75   cfsetispeed(&tio, BAUD);
76
77   // set attributes
78    if (tcsetattr(fd, TCSAFLUSH, &tio) != 0)
79    {
80     printf("%s: Failed to set serial port attributes: %s\n", __func__,
         strerror(errno));
81     this->Disconnect();
82     return -1;
83    }
84
85    /* Need to flush out the I/O line before we start to use it */
86
87    tcflush(fd, TCOFLUSH);
88    tcflush(fd, TCIFLUSH);
89
90    return 1;
91 }
92
93
94 /*
95  * Closes the platform's Serial connection
96  *
97  * return: 1 on success
98  */
99 int
100 X80_Serial::Disconnect()
```

```
101 {
102   /*
103    * Close the port if it is open and return
104    */
105   if (fd >= 0)
106   {
107     // close the serial port
108     close(fd);
109   }
110
111   fd = -1;
112   return 1;
113 }
114
115 /*
116  * Writes an array of bytes to the data stream (Serial port)
117  *
118  * uint8_t *buf: array of bytes to be written
119  * int nBytes: number of bytes to be written
120  *
121  * return: number of bytes written on success, -1 on failure
122  */
123 int
124 X80_Serial::WriteData(uint8_t *buf, int nBytes)
125 {
126   int pos, num, left;
127
128   /* Check if a connection has already been made */
129   if(fd < 0)
130   {
131     return -1;
132   }
133
134 #ifdef X80_DEBUG
135   printf("Writing: ");
136   for(int i=0; i<nBytes; i++)
137   {
138     printf("%d ", buf[i]);
139   }
140   printf("\n");
141 #endif
142
143   // write bytes to the serial port
144   pos = 0;
145   left = nBytes;
146   while (left > 0)
147   {
148     num = write(fd, &buf[pos], left);
149     if (num < 0)
150     {
151       printf("%s: write failed: %s\n", __func__, strerror(errno));
152       return -1;
153     }
154     pos += num;
```

```
155       left -= num;
156     }
157
158     /* Drain the serial port before returning */
159     tcdrain(fd);
160
161     return pos;
162   }
163
164   /*
165    * Reads N bytes from the data stream
166    *
167    * uint8_t *buf: array to store the read data bytes
168    * int nBytes: the number of bytes to read
169    * int timeout: time to wait for data at the serial port
170    *
171    * return: number of bytes read on success, -1 on failure
172    */
173   int
174   X80_Serial::readBytes(uint8_t *buf, int nBytes, int timeout)
175   {
176     int num, ret, pos, left;
177     fd_set rfd;
178     struct timeval tv;
179
180     /* Check if a connection has already been made */
181     if(fd < 0)
182     {
183       printf("%s: NULL File Descriptor -- Must Connect to Serial Device
              first\n", __func__);
184       return -1;
185     }
186
187     pos = 0;
188     left = nBytes;
189     while (left > 0)
190     {
191       /* Set the read timeout */
192       tv.tv_sec = 0;
193       tv.tv_usec = 1000 * timeout;
194
195       /* Check if there is data at the socket */
196       FD_ZERO(&rfd);
197       FD_SET(fd, &rfd);
198       ret = select(fd + 1, &rfd, NULL, NULL, &tv);
199
200       if (ret == 0)
201       {
202         // no data at the port
203         return 0;
204       }
205       else if (ret < 0)
206       {
```

```
207         printf("%s: Serial Port Error - %s\n", __func__, strerror(errno))
                ;
208         return -1;
209       }
210
211       /* Read Bytes */
212       num = read(fd, &buf[pos], left);
213       if(num < 0)
214       {
215         // error
216         return -1;
217       }
218
219       pos += num;
220       left -= num;
221     }
222     return pos;
223 }
224
225 /*
226  * Continuously reads from the data stream until a specified character
227  * is found or there is no data remaining on the serial port
228  *
229  * uint8_t c: the character that we are looking for
230  * int timeout: time to wait for data at the serial port
231  *
232  * return: 1 on success, 0 if not found
233  */
234 int
235 X80_Serial::readChar(uint8_t c, int timeout)
236 {
237   uint8_t byte;
238   int num, ret;
239   fd_set rfd;
240   struct timeval tv;
241
242   if(fd < 0)
243   {
244     printf("%s: NULL File Descriptor -- Must Connect to Serial Device
          first\n", __func__);
245     return -1;
246   }
247   while (true)
248   {
249     tv.tv_sec = 0;
250     tv.tv_usec = 1000 * timeout;
251
252     FD_ZERO(&rfd);
253     FD_SET(fd, &rfd);
254     ret = select(fd + 1, &rfd, NULL, NULL, &tv);
255
256     if (ret == 0)
257     {
258       // no data at the port
```

```
259         return 0;
260       }
261       else if (ret < 0)
262       {
263         printf("%s: Serial Port Error - %s\n", __func__, strerror(errno))
              ;
264         return -1;
265       }
266
267       num = read(fd, &byte, 1);
268       if(num < 0)
269       {
270         // not found
271         return 0;
272       }
273       else if(byte == c)
274       {
275         break;
276       }
277     }
278     return 1;
279 }
280
281 /*
282  * Reads an entire X80 message and returns without checking CRC
283  *
284  * uint8_t *buf: array to store the read data bytes
285  * int size: size of the buffer (to prevent overflow)
286  * int timeout: time to wait for data on the socket
287  *
288  * return: number of bytes read on success, -1 on failure
289  */
290 int
291 X80_Serial::ReadData(uint8_t *buf, int size, int timeout)
292 {
293   int ret, pos, num;
294   fd_set rfd;
295   struct timeval tv;
296   int length;
297
298   /* Check if a connection has already been made */
299   if(fd < 0)
300   {
301     printf("%s: NULL File Descriptor -- Must Connect to Serial Device
          first\n", __func__);
302     return -1;
303   }
304
305   if(buf == NULL)
306   {
307     printf("%s: NULL Buffer\n", __func__);
308     return -1;
309   }
310
```

```
311   /* Set the read timeout */
312   tv.tv_sec = 0;
313   tv.tv_usec = 1000 * timeout;
314
315   /* Check if there is data at the socket */
316   FD_ZERO(&rfd);
317   FD_SET(fd, &rfd);
318   ret = select(fd + 1, &rfd, NULL, NULL, &tv);
319
320   if (ret == 0)
321   {
322     // no data at the port
323     return 0;
324   }
325   else if (ret < 0)
326   {
327     printf("%s: Serial Port Error - %s\n", __func__, strerror(errno));
328     return -1;
329   }
330
331   num = 0;
332   pos = 0;
333
334   /* Read in STX first */
335   if(this->readChar(STX1, timeout) == 0)
336   {
337     return -1;
338   }
339   if(this->readChar(STX2, timeout) == 0)
340   {
341     return -1;
342   }
343   pos += 2;
344   buf[0] = STX1;
345   buf[1] = STX2;
346
347   /* Read in the rest of the header*/
348   if(this->readBytes(&buf[2], 4, timeout) < 0)
349   {
350     return -1;
351   }
352   length = buf[5];
353
354   if(9+length > size)
355   {
356     printf("%s: Read Failed -- Not enough buffer space\n", __func__);
357     return -1;
358   }
359
360   /* Read in the data and CRC*/
361   if(this->readBytes(&buf[6], length+1, timeout) < 0)
362   {
363     return -1;
364   }
```

```
365
366   /* Read in the ETX */
367   if(this->readChar(ETX1, timeout) == 0)
368   {
369     return -1;
370   }
371   if(this->readChar(ETX2, timeout) == 0)
372   {
373     return -1;
374   }
375   buf[7+length] = ETX1;
376   buf[8+length] = ETX2;
377
378   return (length+9);
379 }
```

M.8 X80_Udp.h

```
1 /*
2  * X80_UDP.h
3  *
4  *  Created on: 2010-02-11
5  *      Author: lbrunet
6  */
7
8 #ifndef X80_UDP_H_
9 #define X80_UDP_H_
10
11 #include <stdint.h>
12 #include <netinet/in.h>
13
14 // The X80 UDP class.
15 class X80_Udp
16 {
17   public:
18
19     // Constructor
20   X80_Udp();
21     ~X80_Udp();
22
23     /* Connection Methods */
24     int Connect(const char* address, int port);
25     int Disconnect();
26     bool isConnected() {return (sock > 0) ? true:false;};
27
28     /* Read and Write X80 Messages in byte form*/
29     int ReadData(uint8_t *buf, int size, int timeout);  // reads entire
           x80 messages
30     int WriteData(uint8_t *buf, int nBytes);
31
32   private:
33
34     int sock;          // socket for UDP messages
35     struct sockaddr_in robot; // X80 address stucture
36
37     int dataValid(uint8_t *buf, int nBytes); // returns 1 if it is a
          valid X80 message -- no crc checking
38 };
39
40 #endif /* X80_UDP_H_ */
```

M.9 X80_Udp.cc

```
1 /*
2  * X80_Udp.cc
3  *
4  *  Created on: 2010-02-11
5  *      Author: lbrunet
6  */
7
8 #include "X80_Udp.h"
9 #include "X80_Constants.h"
10
11 #include <stdio.h>
12 #include <string.h>
13 #include <termios.h>
14 #include <sys/socket.h>
15 #include <arpa/inet.h>
16 #include <netdb.h>
17 #include <errno.h>
18 #include <unistd.h>
19 #include <termios.h>
20
21 /*
22  * Constructor for the X80_Udp object
23  * initializes class data
24  */
25 X80_Udp::X80_Udp()
26 {
27   sock = -1;
28 }
29
30 /*
31  * Destructor for the X80_Udp object
32  * disconnects from the platform and cleans up any memory that was
       allocated
33  */
34 X80_Udp::~X80_Udp()
35 {
36   if(sock >= 0)
37   {
38     this->Disconnect();
39   }
40 }
41
42 /*
43  * Connects to the platform via a UDP Connection
44  *
45  * const char* address: ip address of the platform (ex:
       "192.168.0.202")
46  * int port: socket to listen and write to (ex: 10001)
47  *
48  * return: 1 on success, -1 on failure
49  */
50 int
```

```
51 X80_Udp::Connect(const char* address, int port)
52 {
53   /* Create the socket */
54   sock = socket(AF_INET, SOCK_DGRAM, 0);
55   if (sock < 0)
56   {
57     printf("%s: Unable to open UDP socket - %s", __func__, strerror(
         errno));
58     return -1;
59   }
60
61   /* Construct the server sockaddr_in structure */
62   memset(&robot, 0, sizeof(sockaddr));        /* Clear struct */
63   robot.sin_family = AF_INET;                   /* Internet/IP */
64   robot.sin_addr.s_addr = inet_addr(address);   /* IP address */
65   robot.sin_port = htons(port);               /* server port */
66
67   return 1;
68 }
69
70 /*
71  * Disconnects from the platform's UDP connection
72  *
73  * return: 1 on success
74  */
75 int
76 X80_Udp::Disconnect()
77 {
78   /*
79    * Close the port if it is open and return
80    */
81   if (sock >= 0)
82   {
83     // close the serial port
84     close(sock);
85   }
86
87   sock = -1;
88   return 1;
89 }
90
91 /*
92  * Writes an array of bytes to the data stream (UDP socket)
93  *
94  * uint8_t *buf: array of bytes to be written
95  * int nBytes: number of bytes to be written
96  *
97  * return: number of bytes written on success, -1 on failure
98  */
99 int
100 X80_Udp::WriteData(uint8_t *buf, int nBytes)
101 {
102   int num;
103
```

```
104   /* Check if a connection has already been made */
105   if(sock < 0)
106   {
107     return -1;
108   }
109
110 #ifdef X80_DEBUG
111   printf("Writing: ");
112   for(int i=0; i<nBytes; i++)
113   {
114     printf("%d ", buf[i]);
115   }
116   printf("\n");
117 #endif
118
119   /* Write to the port and wait for the message to be sent */
120   num = sendto(sock, buf, nBytes, 0, (struct sockaddr *)&robot, sizeof(
          sockaddr));
121   usleep(X80_MSG_WAIT_PERIOD); // used to replace tcdrain
122
123   return num;
124 }
125
126 /*
127  * Reads an entire X80 message and returns without checking CRC
128  *
129  * uint8_t *buf: array to store the read data bytes
130  * int size: size of the buffer (to prevent overflow)
131  * int timeout: time to wait for data on the socket
132  *
133  * return: number of bytes read on success, -1 on failure
134  */
135 int
136 X80_Udp::ReadData(uint8_t *buf, int size, int timeout)
137 {
138   int num, ret;
139   int length;
140   fd_set rfd;
141   struct sockaddr from;
142   struct timeval tv;
143
144   /* Check if a connection has already been made */
145   if(sock < 0)
146   {
147     return -1;
148   }
149
150
151   /* Set the read timeout */
152   tv.tv_sec = 0;
153   tv.tv_usec = 1000 * timeout;
154
155   /* Check if there is data at the socket */
156   FD_ZERO(&rfd);
```

```
157   FD_SET(sock, &rfd);
158   ret = select(sock + 1, &rfd, NULL, NULL, &tv);
159
160   if (ret == 0)
161   {
162     // no data at the port
163     return 0;
164   }
165   else if (ret < 0)
166   {
167     printf("%s: Socket Error - %s\n", __func__, strerror(errno));
168     return -1;
169   }
170
171   /* Read data */
172   length = sizeof(sockaddr);
173   num = recvfrom(sock,buf,size,0, &from, (socklen_t*)&length);
174   if(num < 0)
175   {
176     printf("%s: Error reading from socket -- error %i - %s\n", __func__
          ,
177            errno, strerror(errno));
178   }
179
180   /* Check if the UDP packet that was read is a valid X80 message*/
181   if(this->dataValid(buf, num))
182   {
183     return num;
184   }
185   return -1;
186 }
187
188 /*
189  * Checks if an array of bytes is a valid X80 Message or not
190  * no CRC checking, this just checks the STX and ETX
191  *
192  * uint8_t *buf: array of bytes to check
193  * int nBytes: number of bytes in the array
194  *
195  * return: 1 for valid, 0 for invalid
196  */
197 int
198 X80_Udp::dataValid(uint8_t *buf, int nBytes)
199 {
200   if(nBytes < X80_CONST_MSG_BYTES)
201   {
202     return 0;
203   }
204
205   if((buf[0] != STX1) || (buf[1] != STX2)
206       || (buf[nBytes -2] != ETX1) || (buf[nBytes -1] != ETX2))
207   {
208     return 0;
209   }
```

```
210
211   return 1;
212 }
```

M.10 MSFF2_Constants.h

```
1  /*
2   * MSFF2_Constants.h
3   *
4   *  Created on: 2010-02-12
5   *      Author: lbrunet
6   */
7
8  #ifndef MSFF2_CONSTANTS_H_
9  #define MSFF2_CONSTANTS_H_
10
11 /* Constants to make sure it is the proper joystick */
12 #define MSFF2_NAME       "Microsoft SideWinder Force Feedback 2 Joystick
       "
13 #define MSFF2_NUM_AXIS     6
14 #define MSFF2_NUM_BUTTONS  9
15
16 /* Joystick value constants */
17 #define MSFF2_NAME_LEN     128
18 #define MSFF2_AXIS_MIN     -32767
19 #define MSFF2_AXIS_MAX     32767
20
21
22 #endif /* MSFF2_CONSTANTS_H_ */
```

M.11 MSFF2_Driver.h

```
1 /*
2  * MSFF2_Driver.h
3  *
4  *  Created on: 2010-02-05
5  *      Author: lbrunet
6  */
7
8 #ifndef MSFF2_DRIVER_H_
9 #define MSFF2_DRIVER_H_
10
11 #include "MSFF2_Constants.h"
12
13 #include <stdint.h>
14 #include <pthread.h>
15
16 /*
17  * Force Feedback 2 Message Structure
18  */
19 typedef struct _MSFF2_Message
20 {
21   /* Joystick Axis Values */
22   int x_axis;   // horizontal {left-to-right: 32767 to -32767}
23   int y_axis;   // vertical {bottom-to-top: -32767 to 32767}
24   int z_axis;   // ignore for now
25
26   /* Joystick Rotary Value */
27   int rotary;   // vertical {low-to-high: -32767 to 32767}
28
29   /* Joystick Button Values */
30   uint8_t button1;
31   uint8_t button2;
32   uint8_t button3;
33   uint8_t button4;
34   uint8_t button5;
35   uint8_t button6;
36   uint8_t button7;
37   uint8_t button8;
38   uint8_t button9;
39
40 } MSFF2_Message;
41
42
43 /*
44  * Structure to hold callback resistrations
45  */
46 typedef struct _MSFF2_Message_Register
47 {
48    void (*callback) (MSFF2_Message * message, void *data); // Callback
       called when message received
49    void *callbackData;
50 } MSFF2_Message_Register;
51
```

```
52 ////////////////////////////////////////////////////
53
54 // The MSFF2 device driver class.
55 class MSFF2_Driver
56 {
57   public:
58
59   int *axis;     // array to store axis values
60   char *buttons;  // array to store button values
61
62     // Constructor
63     MSFF2_Driver();
64     ~MSFF2_Driver();
65
66     /* Method to register joystick data callback */
67     int RegisterCallback(void (*callback) (MSFF2_Message *, void *),
          void *userData);
68
69     /* Connection Methods*/
70     int Connect(char* port);
71     int Disconnect();
72
73     /* Threads to read messages and run user callbacks*/
74     int Run();
75     int Stop();
76
77     static void *StartThread(void *arg)
78   {
79     reinterpret_cast < MSFF2_Driver * >(arg)->rcvThread();
80     return NULL;
81   }
82   void rcvThread();
83
84   private:
85
86     int jfd;       // Joystick input file descriptor
87
88     /* User callback to be called when joystick event is received*/
89     MSFF2_Message_Register MessageCallback;
90
91     /* pthread data*/
92     pthread_t receiveThreadId;
93     pthread_mutex_t threadLock;
94     bool threadRunning;
95
96   protected:
97
98 };
99
100 #endif /* MSFF2_DRIVER_H_ */
```

M.12 X80_Driver.cc

```
1 /*
2  * MSFF2_Driver.cc
3  *
4  *  Created on: 2010-02-12
5  *      Author: lbrunet
6  */
7
8 #include "MSFF2_Driver.h"
9
10 #include <unistd.h>
11 #include <stdlib.h>
12 #include <stdio.h>
13 #include <errno.h>
14 #include <fcntl.h>
15 #include <string.h>
16 #include <linux/joystick.h>
17 #include <termios.h>
18
19 /********************** Constructors ***************/
20 /*
21  * Constructor for the joystick driver
22  * initializes class data
23  */
24 MSFF2_Driver::MSFF2_Driver()
25 {
26   jfd = -1;
27   axis = NULL;
28   buttons = NULL;
29   threadRunning  = false;
30   receiveThreadId = -1;
31
32   MessageCallback.callback = NULL;
33   MessageCallback.callbackData = NULL;
34
35   pthread_mutex_init(&this->threadLock, NULL);
36 }
37
38 /*
39  * Destructor for the joystick driver
40  * disconnects from the platform and cleans up any memory that was
       allocated
41  */
42 MSFF2_Driver::~MSFF2_Driver()
43 {
44
45   if( jfd > 0)
46   {
47     this->Stop();
48     close(jfd);
49   }
50
51   if(axis != NULL)
```

```
52   {
53     free(axis);
54   }
55
56   if(buttons != NULL)
57   {
58     free(buttons);
59   }
60
61   axis = NULL;
62   buttons = NULL;
63 }
64
65 /*
66  * Connects to the Joystick Driver via USB
67  *
68  * char* post: usb port that the joystick is connected to (ex: "/dev/
        input/js0")
69  *
70  * return: 1 on success, -1 on failure
71  */
72 int
73 MSFF2_Driver::Connect(char* port)
74 {
75   int num_axis = 0, num_buttons = 0;
76   char joystick_name[MSFF2_NAME_LEN] = "\0";
77   //char target_name[] = MSFF2_NAME;
78
79   /* Disconnect if already connected to a joystick */
80   this->Disconnect();
81
82   /* Open a non-blocking read-only port to read joystick events from */
83   jfd = open(port , O_RDONLY | O_NDELAY);
84   if(  jfd < 0 )
85   {
86     printf("%s: Unable to open Joystick at '%s' - %s\n", __func__, port
          , strerror(errno));
87     return -1;
88   }
89
90   ioctl( jfd, JSIOCGAXES, &num_axis );
91   ioctl( jfd, JSIOCGBUTTONS, &num_buttons );
92   ioctl( jfd, JSIOCGNAME(MSFF2_NAME_LEN), &joystick_name );
93
94   printf("Joystick detected: %s\n\t%d axis\n\t%d buttons\n\n"
95       , joystick_name, num_axis, num_buttons );
96
97   // Make sure it is the proper joystick
98   if(strcmp(joystick_name, MSFF2_NAME) != 0)
99   {
100    printf("%s: Unsupported Joystick Name: '%s'\n", __func__,
         joystick_name);
101    return -1;
102  }
```

```
103
104   if(num_axis != MSFF2_NUM_AXIS)
105   {
106     printf("%s: Unsupported Number of Axis: '%d' -- Should be %d\n",
107         __func__, num_axis, MSFF2_NUM_AXIS);
108     return -1;
109   }
110
111   if(num_buttons != MSFF2_NUM_BUTTONS)
112   {
113     printf("%s: Unsupported Number of Buttons: '%d' -- Should be %d\n",
114         __func__, num_buttons, MSFF2_NUM_BUTTONS);
115     return -1;
116   }
117
118   // allocate memory
119   axis = (int *) calloc( num_axis, sizeof( int ) );
120   buttons = (char *) calloc( num_buttons, sizeof( char ) );
121
122   fcntl( jfd, F_SETFL, O_NONBLOCK );  /* use non-blocking mode */
123
124   return 1;
125 }
126
127 /*
128  * Closes the data stream to the joystick and cleans up memory useage
129  *
130  * return: 1 on success
131  */
132 int
133 MSFF2_Driver::Disconnect()
134 {
135   if( jfd > 0)
136   {
137     this->Stop();
138     close(jfd);
139   }
140
141   if(axis != NULL)
142   {
143     free(axis);
144   }
145
146   if(buttons != NULL)
147   {
148     free(buttons);
149   }
150
151   axis = NULL;
152   buttons = NULL;
153 }
154
155 /*
156  * Thread that continuously reads Joystick events from the USB port and
```

```
157  * if a message is received, the user callback function is called
158  */
159 void
160 MSFF2_Driver::rcvThread()
161 {
162   struct js_event js;
163
164   while(this->threadRunning)
165   {
166     pthread_mutex_lock(&this->threadLock);
167     if(this->jfd < 0)
168     {
169       pthread_mutex_unlock(&this->threadLock);
170       this->threadRunning = false;
171       break;
172     }
173
174     // read data and send messages
175     read(jfd, &js, sizeof(struct js_event));
176
177     switch (js.type & ~JS_EVENT_INIT)
178     {
179       case JS_EVENT_AXIS:
180         this->axis[js.number ] = js.value;
181         break;
182       case JS_EVENT_BUTTON:
183         this->buttons[js.number] = js.value;
184         break;
185       default:
186         this->threadRunning = false;
187         return;
188     }
189
190     MSFF2_Message msg;
191     msg.x_axis = this->axis[0];
192     msg.y_axis = this->axis[1];
193     msg.z_axis = this->axis[2];
194     msg.rotary = this->axis[3];
195
196     msg.button1 = this->buttons[0];
197     msg.button2 = this->buttons[1];
198     msg.button3 = this->buttons[2];
199     msg.button4 = this->buttons[3];
200     msg.button5 = this->buttons[4];
201     msg.button6 = this->buttons[5];
202     msg.button7 = this->buttons[6];
203     msg.button8 = this->buttons[7];
204     msg.button9 = this->buttons[8];
205
206     /* Check that the user has given a callback to call */
207     if(MessageCallback.callback != NULL)
208     {
209       this->MessageCallback.callback(&msg, this->MessageCallback.
          callbackData);
```

```
210       }
211
212       pthread_mutex_unlock(&this->threadLock);
213       usleep(1000);
214     }
215 }
216
217 /*
218  * Starts the receive thread -- js events will continuously be read and
219  * the user callback will be called
220  *
221  * return: 1 on success, -1 on failure
222  */
223 int
224 MSFF2_Driver::Run()
225 {
226   if( jfd < 0 )
227   {
228     this->Stop();
229     return -1;
230   }
231   else if (this->threadRunning == false)
232   {
233     this->threadRunning = true;
234     pthread_create(&this->receiveThreadId, NULL, &MSFF2_Driver::
          StartThread, this);
235
236     return 1;
237   }
238
239   return -1;
240 }
241
242 /*
243  * Stops the receive thread -- js events will no longer be read from
244  * the USB port
245  *
246  * return: 1 on success, -1 on failure
247  */
248 int
249 MSFF2_Driver::Stop()
250 {
251   if (this->threadRunning == true)
252   {
253     this->threadRunning = false;
254     pthread_join(this->receiveThreadId, NULL);
255
256     return 1;
257   }
258
259   return -1;
260 }
261
262 /************************ Callback Registration *************/
```

```
263
264 /*
265  * This function registers a callback for the user, and adds a pointer
       to user data
266  * that will be added as a parameter in the callback
267  *
268  * void (*callback) (MSFF2_Message *, void *): pointer to the callback
269  * void* data: user data to be passed when the callback is called
270  *
271  * return: 1 on success, -1 on failure
272  */
273 int
274 MSFF2_Driver::RegisterCallback(void (*callback) (MSFF2_Message *, void
       *), void* data)
275 {
276   int i;
277
278   if(callback == NULL)
279   {
280     printf("%s: NULL Callback\n", __func__);
281     return -1;
282   }
283
284   pthread_mutex_lock(&this->threadLock);
285   MessageCallback.callback = callback;
286   MessageCallback.callbackData = data;
287   pthread_mutex_unlock(&this->threadLock);
288
289   return 1;
290 }
```

Appendix N

Teleoperation with the X80 Robots

N.1 Teleoperation.cc

```
1 //============================================================
2 // Name        : Teleoperation.cc
3 // Author      :
4 // Version     :
5 // Copyright   :
6 // Description : Hello World in C++, Ansi-style
7 //============================================================
8
9
10 #include "X80_Driver.h"
11 #include "MSFF2_Driver.h"
12
13 #include <stdlib.h>
14 #include <signal.h>
15 #include <stdio.h>
16 #include <unistd.h>
17 #include <pthread.h>
18 #include <math.h>
19 #include <string.h>
20
21 #define JOYSTICK_DEV  "/dev/input/js0" // add -j port at the cmd prompt
       to change
22 #define X80_ADDRESS   "192.168.0.203"  // add -ip address at the cmd
     prompt to change
23 #define X80_PORT     10001       // add -p at the cmd prompt to change
     value
24 #define SERIAL_ADDRESS  "/dev/ttyS0"   // add -s at the cmd prompt to
     change
25
26 /* Joystick Constants Values */
27 #define MAX_JOY_THROTTLE   32767.0
28 #define MIN_JOY_THROTTLE   -32767.0
29 #define MAX_JOY_LEFT      -32767.0
```

```
30 #define MAX_JOY_RIGHT    32767.0
31
32 /* Teleop Motion Constants */
33 #define MAX_THROTTLE     200.0
34 #define MIN_THROTTLE     100.0
35
36 /* Used to catch shutdown signal --> needed to stop the platform on
       program termination*/
37 volatile sig_atomic_t sigShutdown;
38
39 /*
40  * Data structure that will hold all teleop information
41  */
42 typedef struct _Teleop_Data
43 {
44   int          joystickX;
45   int          joystickY;
46   int          direction;  //0 = reverse, 1 = forward
47   int          dead_man; // 0 = off, 1 = on
48   pthread_mutex_t   lock;   // data lock
49 } X80_Teleop_Data;
50
51
52 /*
53  * Callback when the user presses Crtl-C on the keyboard
54  */
55 void
56 HandleShutdownSignal(int sig __attribute__ ((unused)))
57 {
58    sigShutdown = 1;
59 }
60
61 /*
62  * Callback for joystick events
63  *
64  * MSFF2_Message *msg: joystick driver message
65  * void *userData: data structure passed to the callback --> teleop
       data
66  */
67 void
68 JOYSTICK_DATA_Callback(MSFF2_Message *msg, void *userData)
69 {
70   X80_Teleop_Data *data = NULL;
71
72   data = (X80_Teleop_Data *) userData;
73   pthread_mutex_lock(&data->lock);
74   data->joystickX = (int)((double)msg->x_axis); // -32767 to 32767
75   data->joystickY = abs((double)msg->y_axis); // -32767 to 32767
76   data->direction = ((double)msg->y_axis >= 0) ? 0 : 1;
77   data->dead_man = msg->button1;
78   pthread_mutex_unlock(&data->lock);
79 }
80
81 /*
```

```
82  * Callback for the X80 motor Data
83  * this callback serves a debugging purpose
84  *
85  * X80_Message *message: message from the X80 driver
86  * void *userData: not used --> teleop control is open loop
87  */
88 void
89 MOTOR_DATA_Callback(X80_Message *message, void *userData)
90 {
91   X80_Motor_Control_Data *data = NULL;
92
93   data = (X80_Motor_Control_Data *) message->data;
94
95   printf("Speed 1: %d, Speed 2: %d Direction: %d\n",
96       data->encoder1Speed, data->encoder2Speed, data->encoderDirection)
          ;
97 }
98
99 /*  Main thread for teleop
100 *  Args: -j  -- joystick port (default "/dev/input/js0")
101 *      -i  -- ip address (default "192.168.0.202")
102 *      -s  -- serial port (default "/dev/ttyS0")
103 *      -p  -- udp port (default 10001)
104 *      --help -- lists the args available
105 */
106 int main(int argc, char *argv[])
107 {
108
109   sigset_t blockmask;
110   struct sigaction sa;
111
112   MSFF2_Driver *joystick = NULL;
113   X80_Driver *robot = NULL;
114   char joy_port[64] = JOYSTICK_DEV;
115   char robot_address[64] = X80_ADDRESS;
116   char serial_address[64] = SERIAL_ADDRESS;
117   int port = X80_PORT;
118
119   X80_Teleop_Data data;
120
121   /* Read in args first */
122   for(int i=1; i<argc; i++)
123   {
124     if(strcmp(argv[i], "-j") == 0)
125     {
126       strcpy((char*)&joy_port, argv[i+1]);
127     }
128     else if(strcmp(argv[i], "-i") == 0)
129     {
130       strcpy((char*)&robot_address, argv[i+1]);
131     }
132     else if(strcmp(argv[i], "-s") == 0)
133     {
134       strcpy((char*)&serial_address, argv[i+1]);
```

```
135     }
136     else if(strcmp(argv[i], "-p") == 0)
137     {
138       port = atoi(argv[i+1]);
139     }
140     i++;
141   }
142
143   /* Setup signal handlers for shutdown */
144   sigShutdown = 0;
145   sigfillset(&blockmask);
146   sa.sa_handler = HandleShutdownSignal;
147   sa.sa_mask = blockmask;
148   sa.sa_flags = 0;
149   sigaction(SIGINT, &sa, NULL);
150   sigaction(SIGQUIT, &sa, NULL);
151
152   /* Initialize Robot and Joystick Data */
153   data.dead_man = 0;
154   data.direction = 1;
155   data.joystickX = 0;
156   data.joystickY = 0;
157   pthread_mutex_init(&data.lock, NULL);
158
159   /* Create X80 driver */
160   robot = new X80_Driver();
161
162   /* Connect to robot */
163   if(robot->ConnectUDP(robot_address, port) < 0)
164   {
165     delete robot;
166     return -1;
167   }
168
169   /* Setup Double Potentiometer*/
170   robot->SetControlSensor(X80_WHEEL_LEFT, X80_DOUBLE_POTENTIOMETER);
171   robot->SetControlSensor(X80_WHEEL_RIGHT, X80_DOUBLE_POTENTIOMETER);
172
173   /* Setup PWM Control --> will map pulse width to joystick values*/
174   robot->SetControlMethod(X80_WHEEL_LEFT, X80_CONTROL_PWM);
175   robot->SetControlMethod(X80_WHEEL_RIGHT, X80_CONTROL_PWM);
176
177   /* Set initial wheel directions */
178   robot->SetWheelPolarity(X80_WHEEL_LEFT,X80_WHEEL_POSITIVE);
179   robot->SetWheelPolarity(X80_WHEEL_RIGHT,X80_WHEEL_POSITIVE);
180
181   /* Ensure Platform is stopped for safety*/
182   robot->StopPlatform();
183   usleep(100000);
184
185   /* Create joystick driver */
186   joystick = new MSFF2_Driver();
187
188   /* Connect to joystick */
```

```
189   if(joystick->Connect(joy_port) < 0)
190   {
191     delete robot;
192     delete joystick;
193     return -1;
194   }
195
196   /* Set system callbacks */
197   joystick->RegisterCallback(&JOYSTICK_DATA_Callback, &data);
198   robot->RegisterCallback(X80_GET_MOTOR_CONTROL_SIGNAL, &
          MOTOR_DATA_Callback, NULL);
199
200   /* Run Drivers */
201   joystick->Run();
202   robot->Run(1000);
203   while( sigShutdown == 0 )
204   {
205     // will compare with old data in order to not keep sending commands
206     pthread_mutex_lock(&data.lock);
207
208     if(data.dead_man == 1)
209     {
210
211       /*
212        * The throttle is proportional to the joystick position
213        * It is a percentage of a pre-define maximum throttle value
214        * minimum throttle is used since there is a dead zone caused by
215        * wheel friction
216        */
217       double v = (double)data.joystickY * MAX_THROTTLE /
              MAX_JOY_THROTTLE; // max is 6553 now
218       int vRight = MIN_THROTTLE, vLeft = MIN_THROTTLE;
219
220       if(v < MIN_THROTTLE)
221       {
222         robot->StopPlatform();
223       }
224       else if(data.joystickX >= 0)
225       {
226         vLeft += v;
227         vRight += v - v* sin(0.5 * PI * ((double)data.joystickX /
                MAX_JOY_RIGHT));
228       }
229       else
230       {
231         vRight += v;
232         vLeft += v - v* sin(0.5 * PI * ((double)data.joystickX /
                MAX_JOY_LEFT));
233       }
234       printf("Sending %d and %d\n", vRight, vLeft);
235
236       /* check if we are in the reverse gear and switch the velocities
              signs accordingly */
237       if(data.direction == 0)
```

```
238         {
239           vLeft = -vLeft;
240           vRight = -vRight;
241         }
242 
243         /* Send PWM command to the motors */
244         robot->SetMotorPWM(X80_WHEEL_LEFT, vLeft);
245         robot->SetMotorPWM(X80_WHEEL_RIGHT, -vRight);
246       }
247       else
248       {
249         /* Stop the platform if the dead-man switch is not pressed */
250         robot->StopPlatform();
251       }
252 
253       pthread_mutex_unlock(&data.lock);
254 
255       usleep(10000);
256     }
257 
258     /* Disconnect drivers */
259     robot->StopPlatform();
260     robot->Disconnect();
261     joystick->Disconnect();
262 
263     delete robot;
264     delete joystick;
265     return 0;
266 }
```