NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

LINE AND CIRCLE FORMATION OF DISTRIBUTED AUTONOMOUS MOBILE ROBOTS WITH LIMITED SENSOR RANGE

by

Okay Albayrak

June, 1996

Thesis Advisor: Xiaoping Yun

Approved for public release; distribution is unlimited.

LINE AND CIRCLE FORMATION OF DISTRIBUTED AUTONOMOUS MOBILE ROBOTS WITH LIMITED SENSOR RANGE

Okay Albayrak
Ltjg, Turkish Navy
B.S., Turkish Naval Academy - 1990

In the literature, formation problems for idealized distributed autonomous mobile robots were studied. Idealized robots are represented by a dimensionless point, are able to instantaneously move in any direction and are equipped with perfect range sensors. In this thesis, line and circle formation problems of distributed mobile robots that are subjected to physical constraints are addressed. It is assumed that mobile robots have physical dimensions, and their motions are governed by physical laws. They are equipped with sonar and infrared sensors in which sensor ranges are limited. A new line algorithm based on least-square line fitting, a new circle algorithm, and a merge algorithm are presented. All the algorithms are developed with consideration of physical robots and realistic sensors, and are validated through extensive simulations. Formation problems for mobile robots with limited visibility are also studied. In this case, robots are assumed to be randomly distributed in a large rectangular field such that one robot may not see other robots. An algorithm is developed that makes each robot converge to the center of the field before executing a line or circle algorithm.

Master of Science in Electrical Engineering
June 1996
Advisor: Xiaoping Yun, Department of Electrical and Computer Engineering
Second Reader: Robert G. Hutchins, Department of Electrical and Computer Engineering

Classification of Thesis:
[Unclassified/A]

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 1996	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE LINE AND CIRCLE FORMATION OF DISTRIBUTED AUTONOMOUS MOBILE ROBOTS WITH LIMITED SENSOR RANGE	5. FUNDING NUMBERS
6. AUTHOR(S) Albayrak Okay	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)* In the literature, formation problems for idealized distributed autonomous mobile robots were studied. Idealized robots are represented by a dimensionless point, are able to instantaneously move in any direction and are equipped with perfect range sensors. In this thesis, line and circle formation problems of distributed mobile robots that are subjected to physical constraints are addressed. It is assumed that mobile robots have physical dimensions, and their motions are governed by physical laws. They are equipped with sonar and infrared sensors in which sensor ranges are limited. A new line algorithm based on least-square line fitting, a new circle algorithm, and a merge algorithm are presented. All the algorithms are developed with consideration of physical robots and realistic sensors, and are validated through extensive simulations. Formation problems for mobile robots with limited visibility are also studied. In this case, robots are assumed to be randomly distributed in a large rectangular field such that one robot may not see other robots. An algorithm is developed that makes each robot converge to the center of the field before executing a line or circle algorithm.

14. SUBJECT TERMS Distributed Autonomous Mobile Robotic Systems; Nomad 200 Mobile Robot; Potential Field Method; Nonholonomic Vehicles; Formation Problems			15. NUMBER OF PAGES 94 16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

LINE AND CIRCLE FORMATION OF DISTRIBUTED AUTONOMOUS MOBILE ROBOTS WITH LIMITED SENSOR RANGE

Okay Albayrak
Ltjg, Turkish Navy
B.S., Turkish Naval Academy - 1990

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1996

Author: _____
Okay Albayrak

Approved by: _____
Xiaoping Yun, Thesis Advisor

_____
Robert Gary Hutchins, Second Reader

_____
Herschel H. Loomis, Jr., Chairman
Department of Electrical and Computer Engineering

ABSTRACT

In the literature, formation problems for idealized distributed autonomous mobile robots were studied. Idealized robots are represented by a dimensionless point, are able to instantaneously move in any direction and are equipped with perfect range sensors. In this thesis, line and circle formation problems of distributed mobile robots that are subjected to physical constraints are addressed. It is assumed that mobile robots have physical dimensions, and their motions are governed by physical laws. They are equipped with sonar and infrared sensors in which sensor ranges are limited. A new line algorithm based on least-square line fitting, a new circle algorithm, and a merge algorithm are presented. All the algorithms are developed with consideration of physical robots and realistic sensors, and are validated through extensive simulations. Formation problems for mobile robots with limited visibility are also studied. In this case, robots are assumed to be randomly distributed in a large rectangular field such that one robot may not see other robots. An algorithm is developed that makes each robot converge to the center of the field before executing a line or circle algorithm.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Given a group of mobile robots (say, 20 robots) randomly placed on a laboratory floor, how would one control them to form a geometric pattern such as a circle without using a centralized coordinator? This is the formation problem of distributed mobile robots studied in References 1, 2, 3, 4, and 5. Distributed robots make motion plans based on a given task goal of the group and the perceived information about their environment from onboard sensors without the aid of a centralized coordinator.

Line and circle formation, or formation of any geometric pattern in general, is only one of many issues of distributed mobile robots [Ref. 8]. Representative work addressing other issues of distributed mobile robots includes cellular robotics systems [Ref. 9, 10, 11], and dynamically reconfigurable robotic systems [Ref. 12]. These systems can change their overall shape depending on the task and the environment by autonomously detaching and combining cells.

A. PROBLEM STATEMENT

The formation problem of distributed mobile robots has been studied for idealized mobile robots [Ref. 1, 2, 3, 5]– robots that are represented by a point, able to move in any direction, and equipped with range sensors that can determine the position of all other robots. Since a robot is a point, two or more robots may occupy the same location. Each robot has its own coordinate system and there is no common, global coordinate system. Furthermore, these robots do not communicate with each other. Under these assumptions, Prof. Suzuki and his colleagues have developed a number of distributed formation algorithms. In particular, they developed algorithms for multiple distributed mobile robots to form circles, simple polygons and line segments; to uniformly distribute robots within a circle or a convex polygon; and divide them into groups [Ref. 1, 2, 3, 4, 5].

In the previous studies [Ref. 1, 2], even though the number of robots participating in a given task is assumed to be unknown, the perfect sensor assumption makes it possible for each robot to “see” the location of all other robots, and hence to determine the number

of robots. Perfect sensors are not occluded by the presence of other robots. One of the biggest challenges in implementing existing formation algorithms is the inability to sense the location (or even just the presence) of all other robots by using sonar or infrared sensors. Each robot may see a different number of robots at each instant in time.

Based on earlier work, this thesis studies the line, circle and cluster formation of distributed "physical" mobile robots. The mobile robots considered in this thesis have physical dimensions (hence two robots cannot occupy the same spot), and their motions obey physical laws (hence wheeled mobile robots must satisfy nonholonomic constraints). Furthermore, robots are assumed to be equipped with range sensors having realistic physical properties. The Robot Simulator from Nomadic Technologies, Inc. is used. Robots in the Simulator realistically simulate the motion behavior and sensor systems of Nomad 200 mobile robots [Ref. 22, 23]. The Nomad robot has a synchronous drive mechanism which enables it to translate, steer, and rotate its turret independently. The robot is nonholonomically constrained, thus it is not able to instantaneously move in the lateral direction. The robot's sensor systems include tactile (bumper) sensors, infrared sensors, ultrasonic sensors, and laser sensors. All but the laser sensors are used in the simulations for this study.

To solve the line formation problem considering physical dimensions of mobile robots, a new line algorithm (a least-squares line algorithm) is described in this thesis which is based on least-square line fit computations. Since physical constraints of the robots are considered, robots are able to see only their vicinity, and two or more robots cannot occupy the same spot simultaneously. Each robot finds a least square line fit by using the coordinate information of the visible robots.

The diameter of circles in existing algorithms [Ref. 1, 4, 5] depends on the maximum sensor ranges. Since sensor ranges are limited, a robot can't see robots on the other side of a circle if the desired diameter is larger than the maximum sensor ranges. In this thesis a new algorithm is presented (the merge-then-circle algorithm) to solve this problem. In this new algorithm each robot relies on position information of the two closest

robots, and does not use position information of the furthest robot. So robots can form a circle with a diameter greater than the sensor range limits.

In the previous studies [Ref. 1, 2, 4, 5], formation algorithms of distributed mobile robots are developed without considering the robot's sensor ranges, but in reality robot's sensor ranges are limited. In Reference 3, the formation of a single point by robots and agreement on an x-y coordinate system is examined by assuming that robots have limited visibility. It is also assumed in Reference 3 that a robot is a point (hence robots can occupy the same position simultaneously) and does not block the views of others.

If robots are randomly placed in a large field, a robot may not see other robots due to limited sensor ranges. In this thesis, a scenario where robots are randomly distributed in a large rectangular field is considered. Different from the previous study [Ref. 3], an alternative method (limited range algorithm) is proposed, which is based on the fact that the field is rectangular. This new method converges robots to the center of the field before they execute any formation algorithm.

Different schemes for collision avoidance were examined in References 4, 13, 14, 15, 16, 17, and 18. The method proposed in Reference 4 is discussed in Chapter III. The strategy proposed in Reference 15 is that if a robot detects another robot on its way, it stops and waits some fixed period of time. If a robot is still present, the robot turns left and proceeds forward. The method proposed in Reference 16 adds an initial step to the algorithms from Reference 1 to avoid collisions. Motor schemas [Ref. 19] is another method for navigation and collision avoidance.

Motion control and collision avoidance in this thesis are achieved by implementing the potential field algorithm [Ref. 6, 7]. To each robot of concern, the presence of other robots generates a repulsive force which keeps them apart, and the goal position produces an attractive force. Because the workspace is assumed to be obstacle-free, the shape of robots is circular, and the goal position changes as other robots move, the local minimum problem of the potential field method is rarely encountered in the simulations.

B. OUTLINE OF THE THESIS

This thesis has five chapters and four appendices. The remainder of this thesis is organized as follows: Since the Robot Simulator from Nomadic Technologies, Inc. and the potential field method are used for simulations, the Nomad 200 mobile robot, its simulator, and the potential field method are explained in Chapter II. In Chapter III existing line, circle and point formation algorithms, proposed in References 1, 2, 4, and 5, are described. In Chapter IV, new algorithms for line (least-square line algorithm), circle (merge-then-circle algorithm), cluster formations (merge algorithm), and formations with limited sensor ranges (limited range algorithm) are developed by considering constraints of the physical robots, and the simulation results are depicted. The conclusion and recommendation are discussed in Chapter V. The algorithms proposed in Chapter IV are developed for obstacle-free workspaces. Thus in Chapter V, it is suggested that in future research, the new algorithms developed in this thesis can be improved by considering obstacles in the workspace. The source codes of all the algorithms are listed in the Appendices.

II. LITERATURE REVIEW

A. NOMAD 200 MOBILE ROBOT

The Nomad 200 is an integrated mobile robot system designed for research developments [Ref 22, 23]. The mobile robot has four sensory modules including tactile, infrared, ultrasonic, and laser systems. It has also on-board computers for sensor and motor control and for host computer communication. The mobile base keeps track of its position and orientation through dead-reckoning. The Nomad 200 architecture includes a software package for the host computer with a graphic interface, and a simulator. The Nomad 200 system allows one to switch between the simulator and the real robots. In Figure 1, the distributed architecture of the Nomad 200 system is depicted. Robot behaviors are written in C, including the UNIX operating system functionality. The whole architecture runs on a Sun workstation. The various behaviors have been tested both in simulation and on a Nomad 200 mobile robot. The algorithms described in this thesis have only been tested using the simulator because we have only one Nomad 200 mobile robot in our laboratory.

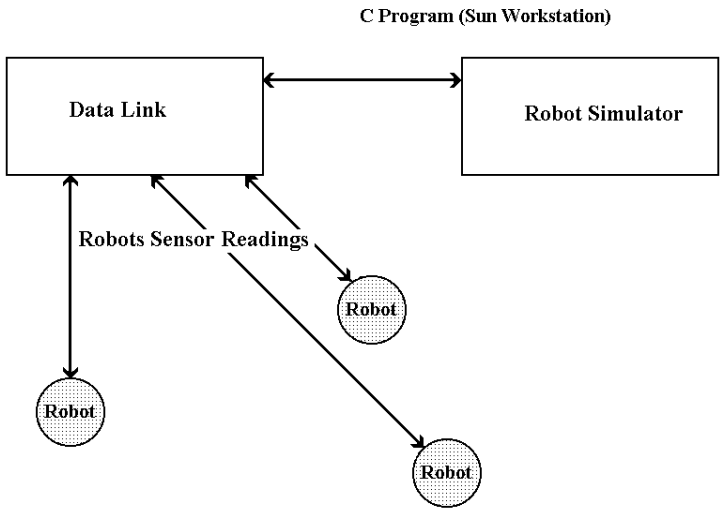

Figure 1. Distributed robotics architecture

1. Mechanical System

The Nomad 200 mobile base is a three servo, three wheel synchronous drive non-holonomic system with zero gyro-radius. The three wheels translate together (controlled by one motor) and rotate together (controlled by a second motor). A third motor controls the angular position of the turret. The robot can only translate along the forward and backward directions along which the three wheels are aligned (this is referred to as non-holonomic constraint, similar to that of a car). The robot has a zero gyro-radius, i.e. the robot can rotate around its center.

The Nomad 200 has a maximum translational speed of 20 inches per second and a maximum rotational speed of 60° per second. It has a diameter of 18 inches and a height of 35 inches.

2. Sensor Systems

The robot's sensor systems include tactile (bumper) sensors, infrared sensors, ultrasonic sensors, and laser sensors. All but the laser sensors are used in simulations for this study. The tactile system which consists of two bumper rings is used to detect contact with any object.

The Nomad 200 has a 16 channel reflective intensity based on an infrared ranging system that provides 360 degree coverage. Each of the 16 sensors are composed of two LED emitters and a photodiode detector. The range to the object(s) is determined by the intensity of the light from the emitter reflected back to the detectors from an object. The infrared sensors are quite accurate at ranges up to 35 inches, but are not reliable at ranges beyond 35 inches.

The Nomad 200 also has a 16 channel sonar ranging system which can give range information from 5 inches to 255 inches with 1% accuracy over the entire range. The sonar system is a time of flight ranging sensor based upon the return time of an acoustic signal. The sensors are standard Polaroid transducers with a beam width of 25°. The circumference of the robot is covered by sixteen sensors.

The user manuals [Ref. 22, 23] for the Nomad 200 robot state that the maximum sonar range is 255 inches. However, by changing two parameters (halfcone and overlap) which are stored in the robot.setup file, users can alter maximum sonar range. Halfcone sets half the angular range of the main lobe of the sonar, while overlap sets the minimal apparent size of a surface to be detected when using the conical model. In the simulations, to get the best result, halfcone is taken as 125, which means 12.5° (the same as default), and overlap is set at 0.08 (default is 0.05). Default values only permit a maximum of 62 inches for the sonar range, but by optimizing the values, robots manage to detect objects out to 206 inches in the simulations.

3. The Robot Simulator

The Nomadic Host Software Development Environment is a full featured object-based mobile robot software development package for the Nomad 200 mobile robot. It consists of two parts: the server and the client. The server performs four functions:

- Host-Robot Interface
- Robot Simulator
- Graphic User Interface
- Client-Server Language User Interface.

The client provides the link between the application program and the server. The Host-Robot interface allows complete control of the robot from a host computer. The Robot Simulator runs on the host computer and simulates the robot's basic motion patterns, such as translation, steering, and turret rotation. It also simulates the five sensor systems, which are tactile, infrared, sonar, laser, and compass. The simulated robot responds to the same set of commands as the real robot. The simulator is capable of simulating up to six robots. That's why the algorithms described in this thesis employ up to six robots in their simulations. The graphics user interface provides graphic displays for various sensory information and interfaces between the robot and the robot simulator. The client-server

language user interface allows users to program in C or Lisp and acts as a client process to access the server.

The simulator has graphics user-interface windows, which provide a graphic display for the robot's position and orientation and its various sensor systems. The graphic environment consists of four main windows: the world window, the robot window, the short sensors window, and the long sensors window. The world window gives an overall view of the environment (real or simulated). The robot window, (one for each robot), contains information about each individual robot, such as current command executed, position, orientation, and sensor data history. There are two windows (the short sensors window and the long sensors window) attached to each robot window that give more detailed information about the current sensor readings. Each time any of the functions that return sensor data is called, the sensor data returned, as well as the current positions of all robots, are displayed graphically on these windows. Users are allowed to draw maps in the world window to simulate the environment. The figures that show the simulation results in the following sections are the snapshots of the world window taken at different time instants during a simulation.

In order to run the simulator, the executable server program (Nserver), the setup files for the world (world.setup) and for each robot (robot.setup), as well as the license file must be in the same directory. To start the server, one simply executes the Nserver. Individual setup files can be specified as command line parameters. If the setup files are not specified, the server will automatically look for world.setup and robot.setup. It is necessary to have a separate setup file for each robot to be created. The name of each robot setup file must be specified in the world setup file. The best way to discriminate between the robots is to set a different color for each robot in its own robot.setup file.

The application program for each robot should run simultaneously as a separate process, by taking advantage of multitasking capabilities of the UNIX operating system. This makes debugging very easy and provides the possibility of testing each behavior independently, as well as the ability to add or remove some robots during simulations.

B. POTENTIAL FIELD METHOD

A robot in the potential field method is treated as a point represented in configuration space as a particle under the influence of an artificial potential field *U* whose local variations reflect the "structure" of the free space. The potential function can be defined over free space as the sum of an attractive potential pulling the robot toward the goal configuration and a repulsive potential pushing the robot away from the obstacles [Ref. 6]. Motion planning is performed in an iterative fashion. At each iteration, the artificial force induced by the potential function at the current configuration is regarded as the most appropriate direction of motion, and path planning proceeds along this direction by some increment.

The general idea is that a robot is attracted toward its goal configuration, while being repulsed by the obstacles. In this section, this idea is illustrated with the definition of one possible potential function, in the case where the robot moves freely in $W=R^N$, with $N=2$, i.e. $C=R^N$. *W* denotes the Robot's workspace, *R* is the set of real numbers, and *C* denotes the configuration space of a robot. An element of *C* is denoted by *(q)*. A more detailed discussion can be found in Reference 6.

The field of artificial forces *F(q)* in *C* is produced by a differentiable potential function:

$$U: C_{free} \rightarrow R, \quad \text{with: } \vec{F}(q) = -\vec{\nabla} U(q) , \qquad \textbf{(1)}$$

where $\vec{\nabla} U(q)$ denotes the gradient vector of *U* at *q*. In $C = R^N (N = 2 \text{ or } 3)$, we can write *q=(x, y) or (x, y, z)*, and:

$$\vec{\nabla}U = \begin{bmatrix} \dfrac{\partial U}{\partial x} \\ \\ \dfrac{\partial U}{\partial y} \end{bmatrix} \quad \text{or} \quad \vec{\nabla}U = \begin{bmatrix} \dfrac{\partial U}{\partial x} \\ \\ \dfrac{\partial U}{\partial y} \\ \\ \dfrac{\partial U}{\partial z} \\ \\ \end{bmatrix} . \qquad \textbf{(2)}$$

In order to attract the robot toward its goal configuration while repulsing it from the obstacles, U is constructed as the sum of two elementary potential functions:

$$U(q) = U_{att}(q) + U_{rep}(q) \ , \qquad \textbf{(3)}$$

where U_{att} is the attractive potential associated with the goal configuration q_{goal} and U_{rep} is the repulsive potential associated with the C-obstacle region. U_{att} is independent of the C-obstacle region, while U_{rep} is independent of the goal configuration. With these conventions, $\vec{F}$ is the sum of two vectors:

$$\vec{F}_{att} = -\vec{\nabla}U_{att} \quad \text{and} \quad \vec{F}_{rep} = -\vec{\nabla}U_{rep} \ , \qquad \textbf{(4)}$$

which are called the attractive and the repulsive forces, respectively.

1. Attractive Potential

The attractive potential field U_{att} can simply be defined as a parabolic-well, i.e.:

$$U_{att}(q) = \frac{1}{2}\xi\rho^2_{goal}(q) \ , \qquad \textbf{(5)}$$

where ξ is a positive scaling factor and $\rho_{\text{goal}}(q)$ denotes the Euclidean distance $\|q - q_{\text{goal}}\|$. The function U_{att} is positive or null, and attains its minimum at q_{goal}, where $U_{\text{att}}(q_{\text{goal}}) = 0$.

The function ρ_{goal} is differentiable everywhere in C. At every configuration q, the attractive force $\vec{F}_{att}$ deriving from U_{att} is:

$$\begin{aligned}\vec{F}_{att}(q) &= -\vec{\nabla}U_{att}(q) \\ &= -\xi\rho_{goal}(q)\vec{\nabla}\rho_{goal}(q) \\ &= -\xi\left(q - q_{goal}\right)\end{aligned} \quad . \qquad \textbf{(6)}$$

The parabolic well demonstrates good stabilizing characteristics. It generates a force $\vec{F}_{att}$ that converges linearly toward 0 when the robot's configuration gets closer to the goal configuration. On the other hand, $\vec{F}_{att}$ increases with the distance to the goal configuration and finally tends toward infinity when $\rho_{\text{goal}}(q) \to \infty$. Alternatively, U_{att} can be defined as a conic-well, i.e.:

$$U_{att}(q) = \xi\rho_{goal}(q) \quad . \qquad \textbf{(7)}$$

Then, the attractive force is:

$$\begin{aligned}\vec{F}_{att}(q) &= -\xi\vec{\nabla}\rho_{goal}(q) \\ &= -\xi\frac{\left(q - q_{goal}\right)}{\left\|q - q_{goal}\right\|}\end{aligned} \quad . \qquad \textbf{(8)}$$

The amplitude of $\vec{F}_{att}(q)$ is constant over C, except at q_{goal}, where U_{att} is singular. Since the amplitude of the force does not tend toward 0 when $q \to q_{\text{goal}}$, the conic-well potential does not have the stabilizing characteristics of the parabolic-well function.

The advantages of both the parabolic and the conic-wells can be combined by defining the attractive potential as a parabolic-well within a distance "d" from the goal configuration and a conic-well beyond that distance. In this case a discontinuity problem is encountered as plotted in Figure 2(a) when it is implemented by using the above conic-well definition. The attractive force must be made continuous at transition point, which

can be achieved simply by multiplying the conic-well attractive potential by "*d*". Then, the resulting attractive force can be defined as follows, which is plotted in Figure 2(b).

$$\vec{F}_{att}(q) = \begin{cases} -\xi\left(q - q_{goal}\right) & \text{if } \left\|q - q_{goal}\right\| \le d \\ -\xi d \dfrac{\left(q - q_{goal}\right)}{\left\|q - q_{goal}\right\|} & \text{if } \left\|q - q_{goal}\right\| > d \end{cases} . \qquad \textbf{(9)}$$

Figure 2 plots of attractive force that is defined by combining the attractive potential as a parabolic-well within a distance "*d*" from the goal configuration and a conic-well beyond that distance. Figure 2(a) illustrates the discontinuity at the transition point, and (b) illustrates the continuity after multiplying the conic-well attractive potential by "*d*".

2. Repulsive Potential

The main idea is to create a potential barrier around the C-obstacle region that cannot be traversed by the robot. In addition, the repulsive potential should not affect the motion of the robot when it is sufficiently far away from the C-obstacles.

These constraints can be achieved by defining the repulsive potential function as follows:

$$U_{rep}(q) = \begin{cases} \dfrac{1}{2}\eta\left(\dfrac{1}{\rho(q)} - \dfrac{1}{\rho_0}\right)^2 & \text{if } \rho(q) \le \rho_0 \\ 0 & \text{if } \rho(q) > \rho_0 \end{cases} , \qquad \textbf{(10)}$$

where η is a positive scaling factor, $\rho(q)$ denotes the distance from q to the C-obstacle region *CB*, i.e.:

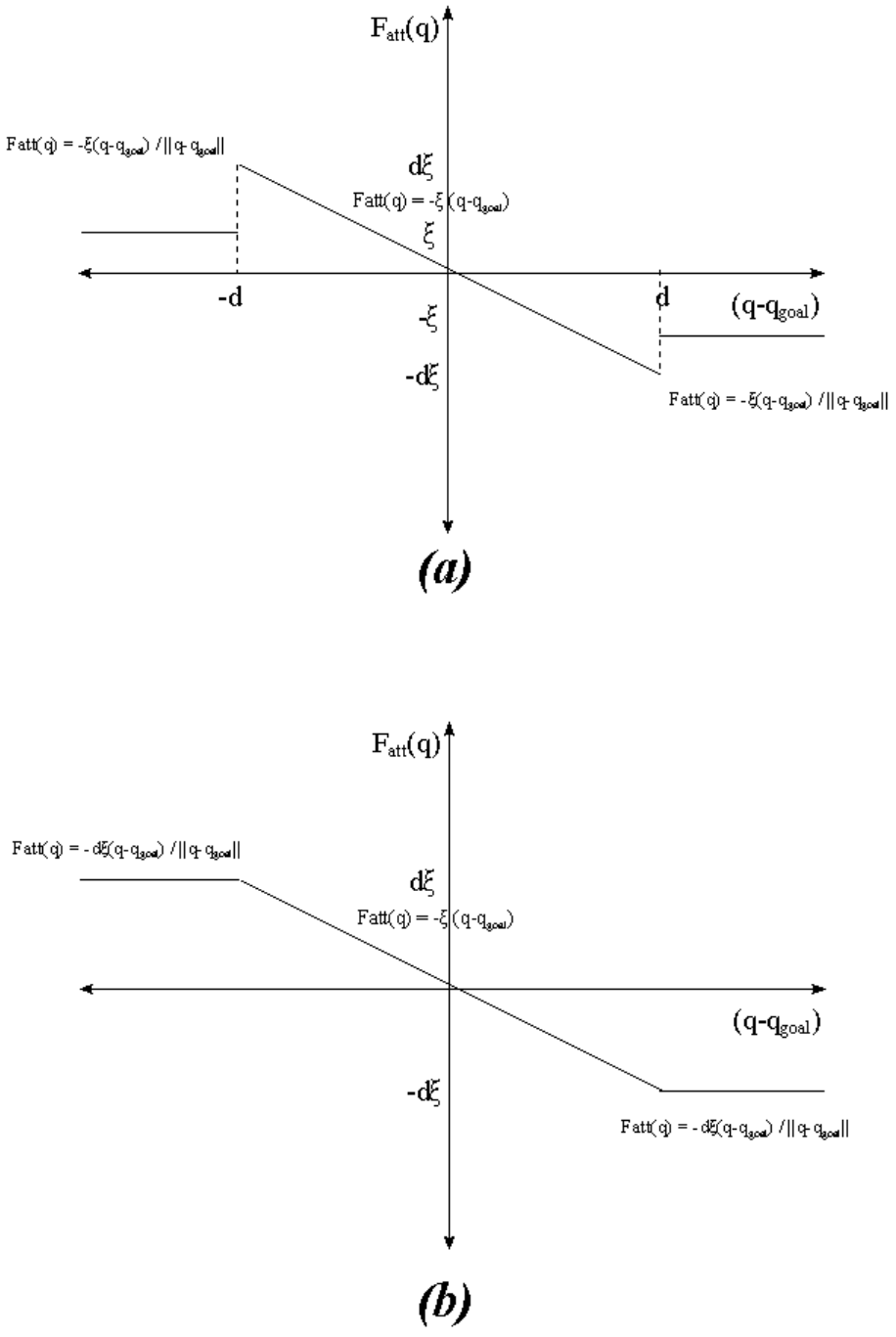

Figure 2. (a) Discontinuity problem, and (b) Continuous attractive force.

$$\rho(q) = \min_{q' \in CB} \left\| q - q' \right\| \quad , \tag{11}$$

and ρ_0 is a positive constant called the distance of influence (or cut-off distance) of the C-obstacle. The function U_{rep} is positive or null, tends to infinity as q gets closer to the C-obstacle region, and is null when the distance of the robot's configuration to the C-obstacle is greater than ρ_0.

If CB is a convex region with a piece wise differentiable boundary, ρ is differentiable everywhere in C_{free}. Then the artificial repulsive force derived from U_{rep} is defined as follows:

$$\begin{aligned} \vec{F}_{rep}(q) &= -\vec{\nabla} U_{rep}(q) \\ &= \begin{cases} \eta\left(\frac{1}{\rho(q)} - \frac{1}{\rho_0}\right)\frac{1}{\rho^2(q)}\vec{\nabla}\rho(q) & \text{if } \rho(q) \le \rho_0 \\ 0 & \text{if } \rho(q) > \rho_0 \end{cases} . \end{aligned} \tag{12}$$

Each obstacle detected by any sensor produces a repulsive force. Equation (12) calculates the repulsive force produced by one obstacle. But in implementation, each sonar sensor produces a repulsive force when something is detected.

The resulting repulsive force is the sum of repulsive potential fields created by each individual sensor contact. Thus, the resulting artificial repulsive force is:

$$\vec{F}_{rep}(total)(q) = \sum_{i=0}^{15} \vec{F}_{rep(i)}(q) \ , \tag{13}$$

where *i* represents the sensor number. In the Nomad 200 robot, there is one infrared and one sonar sensor that can scan the same direction. Infrared sensor information is used if the returned value is within maximum infrared sensor range, and the sonar information is used otherwise.

In the simulations, total repulsive force is calculated by adding the repulsive forces which are produced by sensors that detect obstacles at ranges less than ρ_0 distance. ρ_0 is empirically determined using simulations. In the least-square line algorithm, it is taken as 20 inches. In the merge algorithm and merge-then-circle algorithm, it is taken as 50 inches.

III. DESCRIPTION OF EXISTING ALGORITHMS

In this chapter the existing line, circle and point algorithms, proposed in References 1, 2, 4, and 5, are described. These algorithms have different assumptions. In Reference 1, it is assumed that robots cannot determine their absolute positions in the plane. Also, they don't have a compass to determine absolute directions, and all robots are identical. In Reference 5, the time that it takes for a robot to move to its new position is negligibly small and a robot is a "point," hence two or more robots can occupy the same position simultaneously. In Reference 4, which is different from References 1 and 5, the physical dimensions of robots are considered, and they are represented as discs with diameters of 40 centimeters. It is assumed in Reference 4 that a robot can monitor positions of other robots and move in any desired direction at any speed not exceeding a given maximum of 5 cm per second, and robots do not have a common x-y coordinate system.

In References 1 and 5, collision avoidance is not considered. In Reference 4, a revised version of Reference 1, the physical dimensions of robots are considered, and a simple collision avoidance strategy is implemented. The strategy is: if a robot detects another robot nearby (implemented at 20 cm which is measured between the surfaces of robots) in the direction of its motion, it swerves to the left minimally, provided that it successfully finds a direction that is clear of any such robots. If left swerve is impossible, the robot doesn't move until either its path becomes clear or a suitable left swerve becomes possible.

A. LINE ALGORITHMS

In References 1 and 4, Prof. Suzuki and Prof. Sugihara proposed an algorithm to solve the formation of a line segment problem with distributed autonomous mobile robots. The algorithm depends on two robots that will be endpoints of the line segment. The following steps explain how the algorithm works:

- **Step 1.** Two robots are manually moved to their destination points which are the endpoints of the line segment.

- **Step 2.** Then all other robots execute a *fillpolygon* algorithm which works as follows. If one of the robots sees all other robots arranged in a wedge whose apex angle is less than π, the robot moves into the wedge along the bisector of the apex. If not, the robot moves away from the nearest robot.

In Reference 5, Prof. Suzuki and Prof. Yamashita presented a simple line algorithm. It is assumed that each robot repeatedly becomes active and inactive (sleep mode) at unpredictable time instants. Robots do the following when they become active:

- **Step 1.** Determine the furthest robot R_f and the closest robot R_c.

- **Step 2.** Calculate the distance d from its current position to the point p that is the foot of the perpendicular drop from itself to a line passing through R_c and R_f.

- **Step 3.** Move min $\{d, v\}$ towards point p, where v is the maximum distance robots can move at a time.

B. CIRCLE ALGORITHMS

An algorithm for formation of a circle with a given radius r is proposed by Prof. Suzuki and Prof. Sugihara in References 1 and 4. For convenience, let robot R be any one of the distributed robots participating in the task of circle formation. The algorithm works as follows. Robot R continuously monitors the position of the furthest robot R_f and the closest robot R_c and moves in real time. In this algorithm, a is the distance between R and R_f, and ξ is a small positive constant.

- **Step 1.** Robot R moves toward R_f, if a is greater than $2r$.

- **Step 2.** Robot R moves away from R_f, if a is less than $(2r - \xi)$.

- **Step 3.** Robot R moves away from R_c, if $(2r - \xi) \leq a \leq 2r$.

It is pointed out in References 1 and 4 that robots using this algorithm sometimes converge into a configuration known as Reuleaux's triangle. Furthermore, in Reference 5 a

different, better algorithm is proposed which avoids convergence into a Reuleaux triangle if the number of robots is large. This new circle algorithm works as follows [Ref. 5]. As before, robot R becomes active and inactive at random points in time. Each time robot R becomes active, it:

- **Step 1.** Determines the furthest robot R_f and the closest robot R_c.
- **Step 2.** Calculates the distance d from its current position to the middle point p_m between R_c and R_f.
- **Step 3.** Moves a distance of $\min\{d - r, v\}$ towards p_m if $(d - r) \geq 0$, or a distance of $\min\{r - d, v\}$ away from p_m if $(d - r) < 0$, where v is the maximum distance that a robot can move at a time, r is the desired radius of a circle to be formed.

Both algorithms, described above, do not work when the desired diameter of the circle is larger than sensor range limits (hence a robot is not able to see robots on the other side of the circle).

C. POINT ALGORITHMS

The following algorithm, described in Reference 5, forms a point by moving robot R toward R_f. This algorithm works as follows. Each time robot R becomes active it calculates the distance a to R_f and moves a distance $\min\{a/b, v\}$ towards R_f, where b is a constant greater than one and v is the maximum distance R can move each time.

Prof. Suzuki and Prof. Yamashita also proposed another algorithm in Reference 5, to converge robots towards a point. In this algorithm, robot R calculates the distance d to the centroid g of robot positions, then it moves a distance $\min\{d/b, v\}$ towards g.

D. SUMMARY

Implementing existing formation algorithms on physical robots is very difficult because of their assumptions. The existing line algorithm proposed in References 1 and 4 is not a completely autonomous implementation for line formation, because it depends on two robots which are manually moved to the endpoints of a line segment. The other line

algorithm, presented in Reference 5, depends on detection of furthest and closest robots. The circle algorithms, explained in References 1, 4, and 5, cannot be used to form a large circle whose diameter is larger than sensor range limits of the robots.

Physical robots cannot form a point. In the point algorithms, explained in Reference 5, robots are considered as dimensionless points and they are capable of detecting all the other robots in the operation field. In the next chapter, new formation algorithms are developed for physical robots to solve the problems described in this chapter.

IV. DESCRIPTION OF NEW ALGORITHMS AND SIMULATION RESULTS

In this chapter new algorithms are developed to solve formation problems of distributed mobile robots. It is assumed that robots have physical dimensions (in the simulations robots are represented as disc shapes with radius of r_o = 9 inch), so they cannot occupy the same spot, and their motions obey physical laws (hence wheeled mobile robots must satisfy nonholonomic constraints). Furthermore, robots don't have common coordinate systems, and they are identical. The same program is executed by each robot. In simulations, the potential field method is used for collision avoidance and motion control. The algorithms are simulated using the Robot Simulator software from Nomadic Technologies, Inc. The algorithm source codes are listed in the Appendices.

A. LEAST-SQUARE LINE ALGORITHM

The existing line algorithm explained in References 1 and 4 depends on two robots which are manually moved to their destination points. The line algorithm described in Reference 5 only utilizes position information of the furthest and closest robots. A new line algorithm is described in this section. This new algorithm utilizes position information of all robots detected by each robot.

The basic idea is that, at each iteration, a robot finds the least square line fitting of all visible robots and moves towards the line. Robots don't have a common coordinate system. They only have their own coordinate systems to compute the line fitting. It is noted that at each instant of time, each robot may see a different number of robots and different map.

Assume that robot *R* is any one of the distributed robots and that it sees *n* robots in its vicinity at the current instant of time. Positions of the visible robots are represented in the coordinate system of robot *R* as *n* pairs of data, (x_i , y_i), i = 1, 2, 3, ..., n. Note that in the line fitting computations, coordinate (x_o , y_o) of robot *R* is included. The least-square line fitting of the (n+1) pairs of data can be found by the following standard line equation [Ref. 20].

$$y = mx + b\,, \tag{14}$$

where the values of *m* and *b* minimize the value of the following function:

$$S = \sum \left(mx_i + b - y_i \right)^2 . \tag{15}$$

The values of *m* and *b* that accomplish this are determined with the first and second derivative tests as explained in [Ref. 20]

$$m = \frac{\left(\sum x_i\right)\left(\sum y_i\right) - (n+1)\sum x_i y_i}{\left(\sum x_i\right)^2 - (n+1)\sum x_i^2}\ , \tag{16}$$

$$b = \frac{1}{n+1}\left(\sum y_i - m\sum x_i\right), \tag{17}$$

with all sums running from $i = 0$ to $i = n$.

This representation (Equation 14) of lines has a singularity when the resulting line is parallel to the y-axis. To avoid this singularity, the parametric representation of lines [Ref. 21] is used to compute the least-square line fitting.

$$x\cos\theta + y\sin\theta = r\ , \tag{18}$$

where *r* and θ are two parameters depicted in Figure 3. The derivation of the following least-square procedure is from Reference 21. The sum of the squares of all residuals is:

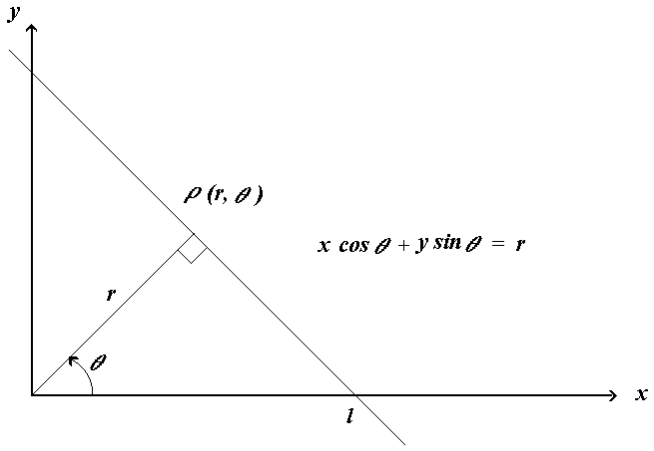

Figure 3. Parametric representation of lines using *r* and *θ*.

$$S = \sum \left(x_i \cos\theta + y_i \sin\theta - r \right)^2 \ , \tag{19}$$

where the summation goes from *0* to *n*. The line which best fits the set of data minimizes *S*. The two parameters that characterize the best line are computed from:

$$\frac{\partial S}{\partial r} = \frac{\partial S}{\partial \theta} = 0 \ . \tag{20}$$

Differentiating *S* with respect to *r*, we have:

$$\frac{\partial S}{\partial r} = 2\left[(n+1)r - \sum x_i \cos\theta - \sum y_i \sin\theta \right] = 0 \ , \tag{21}$$

from which we solve *r*:

$$r = \frac{1}{n+1}\left(\sum x_i \cos\theta + \sum y_i \sin\theta \right) \ . \tag{22}$$

Here *r* may take any value (positive, negative, or zero). Substituting *r* into Equation (19) and differentiating *S* with respect to θ, we get:

$$\frac{\partial S}{\partial \theta} = \mu_1 \sin(2\theta) + 2\mu_2 \cos(2\theta) = 0 \ , \tag{23}$$

where

$$\mu_1 = \sum y_i^2 - \sum x_i^2 + \frac{\left(\sum x_i\right)^2 - \left(\sum y_i\right)^2}{n+1}$$

and **(24)**

$$\mu_2 = \sum \left(x_i y_i\right) - \frac{\sum x_i \sum y_i}{n+1}$$

If we pick θ in such a way that:

$$\sin(2\theta) = -2\mu_2 \quad \text{and} \quad \cos(2\theta) = \mu_1 \ , \tag{25}$$

it clearly satisfies Equation (23). Therefore θ is calculated from:

$$\theta = \frac{1}{2}\text{atan2}\left(-2\mu_2, \mu_1\right) \ . \tag{26}$$

After finding θ and *r*, the robot is directed to move to point *p* on the line as shown in Figure 3.

In the simulations, the potential field method is used for collision avoidance. In the program, to compute the goal point in the line segment, Equations (16) and (17) are evaluated. In Equation (16) when *m* goes to infinity, a problem occurs. This problem is solved in the simulations by checking the conditions and using the if statements. Because of this problem, Equations (22) and (26) are developed. The simulation results show the

program that uses Equations (16) and (17) runs in any distribution of robots. That's why, even though Equations (22) and (26) are developed, they are not implemented in the simulations.

After calculating the goal point, robot *R* moves to that point because of an attractive force produced by that point. The movements of robot *R* also depend on repulsive forces produced by any object within a distance of 20 inches. Even if there are other robots at or near point *p*, robot *R* is able to squeeze into the line because a repulsive force causes other robots to move slightly. Robot *R* finds a new goal point by reason of utilizing position information of all visible robots. This algorithm doesn't uniformly distribute the robots in the line, which means that the distances between robots may not be equal. In the simulations, if robot *R* detects only one robot nearby (which is the case if it is the endpoint of the line segment), it positions itself d_o distance away from the closest robot, where d_o stands for the cut-off distance of repulsive forces in the potential field algorithm (in simulations d_o is taken to be 20 inches).

Figure 4 shows a simulation result of the algorithm. In Figure 4(a) the initial distribution of the robots is depicted. As seen in Figure 4(f) robots formed a line segment in the plane. Note that the position of the line depends on the initial distribution of the robots. The source code of the algorithm is in Appendix A.

B. MERGE ALGORITHM

In the previous chapter, existing point algorithms are discussed. As mentioned before, robots are assumed to be dimensionless points in Reference 5. It is not possible to form a point using existing algorithms if the physical dimensions of robots are considered. That's why in this section a new algorithm is described. The new algorithm simply merges robots together to form a cluster instead of a point.

The algorithm works as follows. Robot *R* continuously monitors the environment and moves to the middle point p_m between the visible furthest robot R_f and closest robot R_c by using the potential field method to avoid collisions. Figure 5 shows a schematic representation of this algorithm. But if robot *R* sees only one robot, which may be the

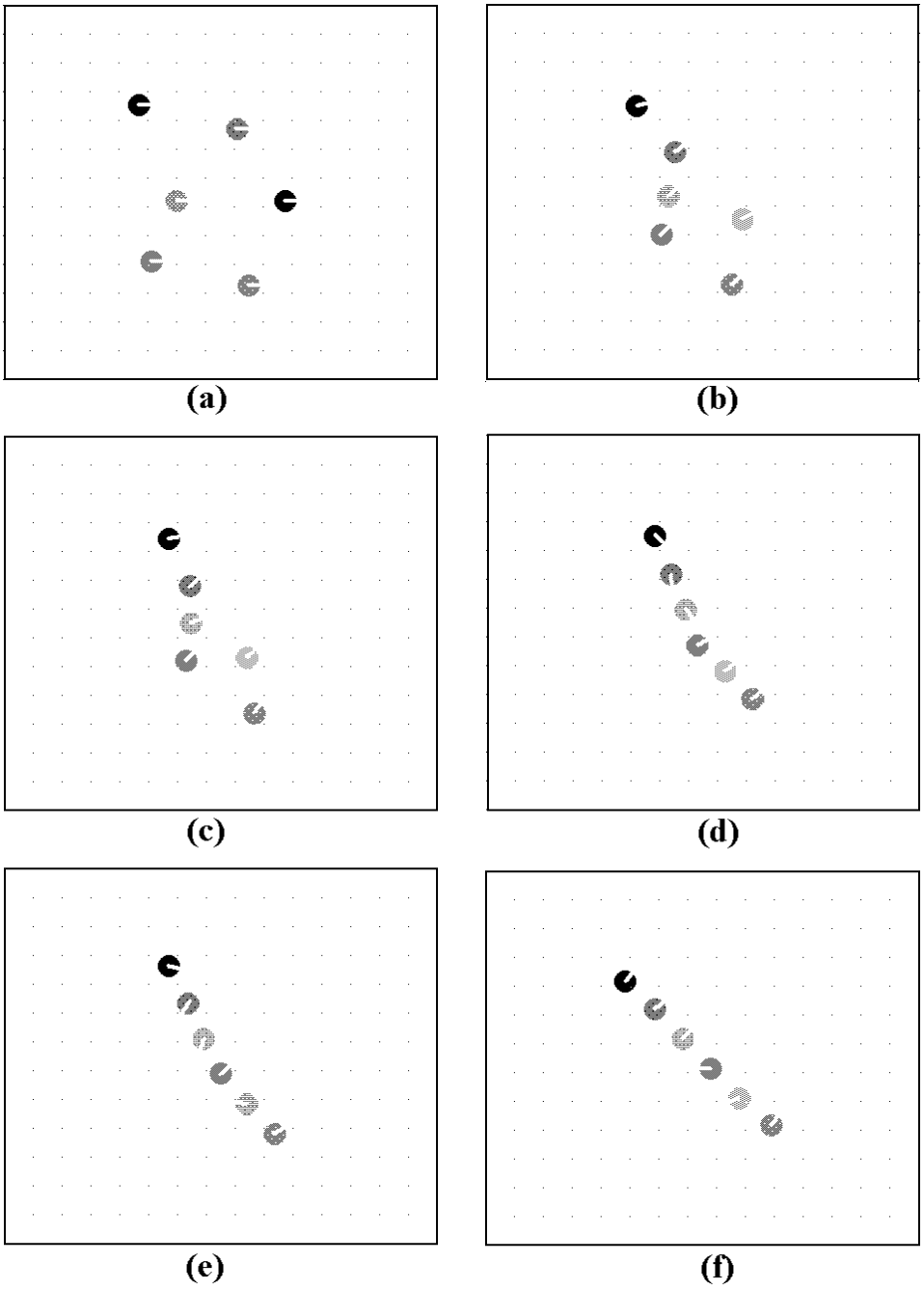

Figure 4. Selected images from a simulation of the least-square line algorithm: (a) the initial distribution, (b)-(e) intermediate steps, and (f) the final distribution of the robots.

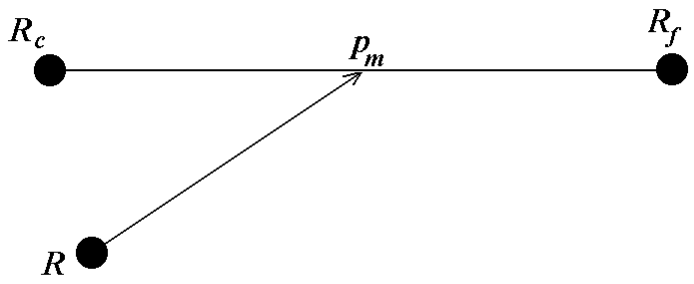

Figure 5. Schematic representation of merge algorithm.

situation if the robots are initially distributed in a line segment and R is at one of the endpoints of the line segment, then robots cannot merge together. To solve this problem, robot R turns $\pi/3$ degree right from R_c and moves to a distance d_o from R_c if it detects only R_c. If robot R doesn't see any robot around, then it doesn't move. So if robots are initially distributed in a large field far away from each other, they may not be able to merge together, or they may form more than one cluster, because the distance between the clusters or the robots are greater than the sensor range limits.

Figure 6 shows a simulation result of the algorithm from initial distribution to final stage. In the simulations d_o is 50 inches. Since the attractive and repulsive forces cancel each other, robots don't move once they are merged. But if only two robots form a cluster, they keep moving around themselves. In simulations, even if robots are initially distributed in a line, they form a cluster by executing this algorithm. Figure 7 shows another simulation result where robots are initially distributed in a line. The source code of the algorithm is in Appendix B.

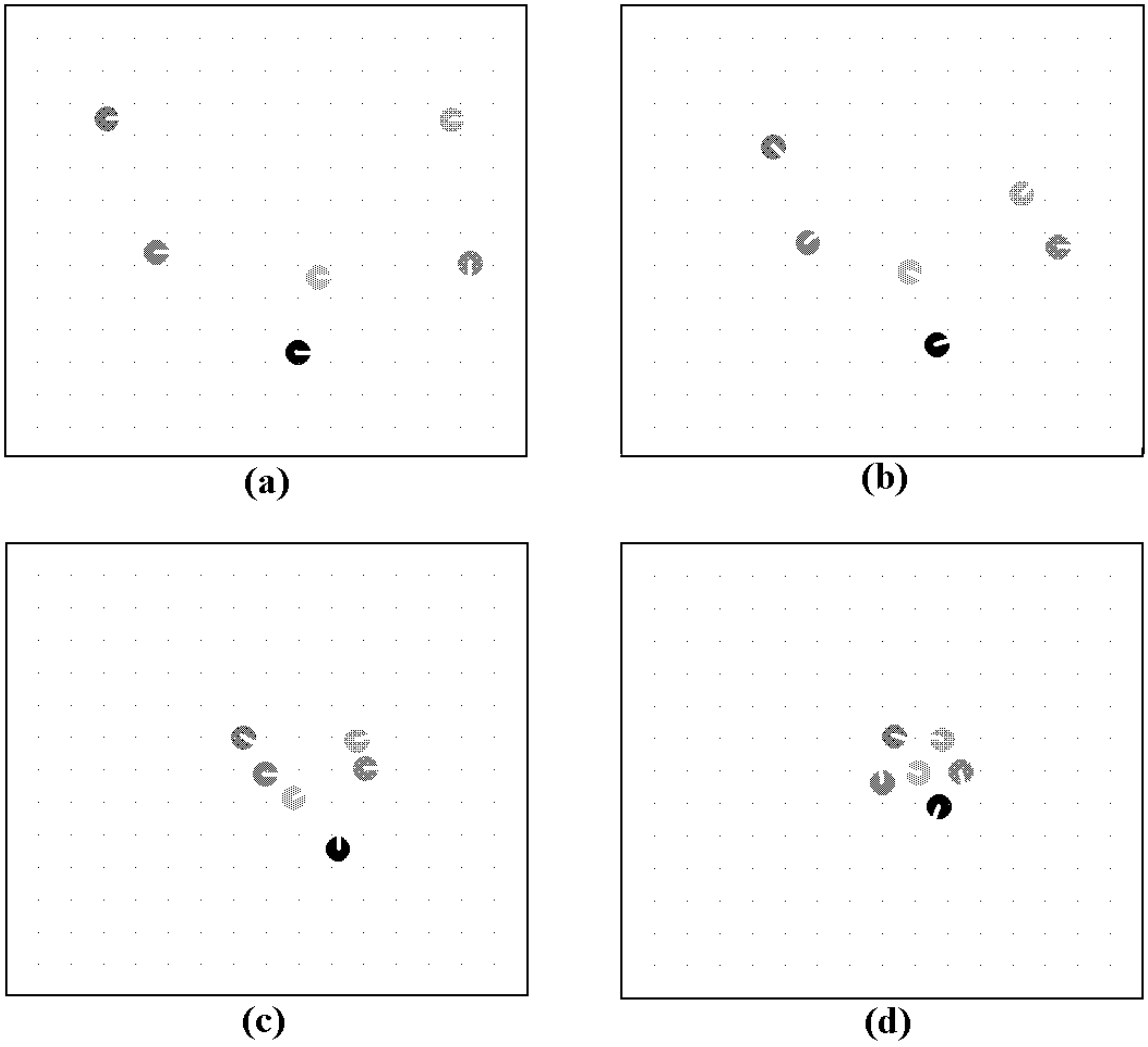

Figure 6. Selected images from a simulation of the merge algorithm: (a) the initial distribution, (b)-(c) intermediate steps, and (d) the final distribution of the robots.

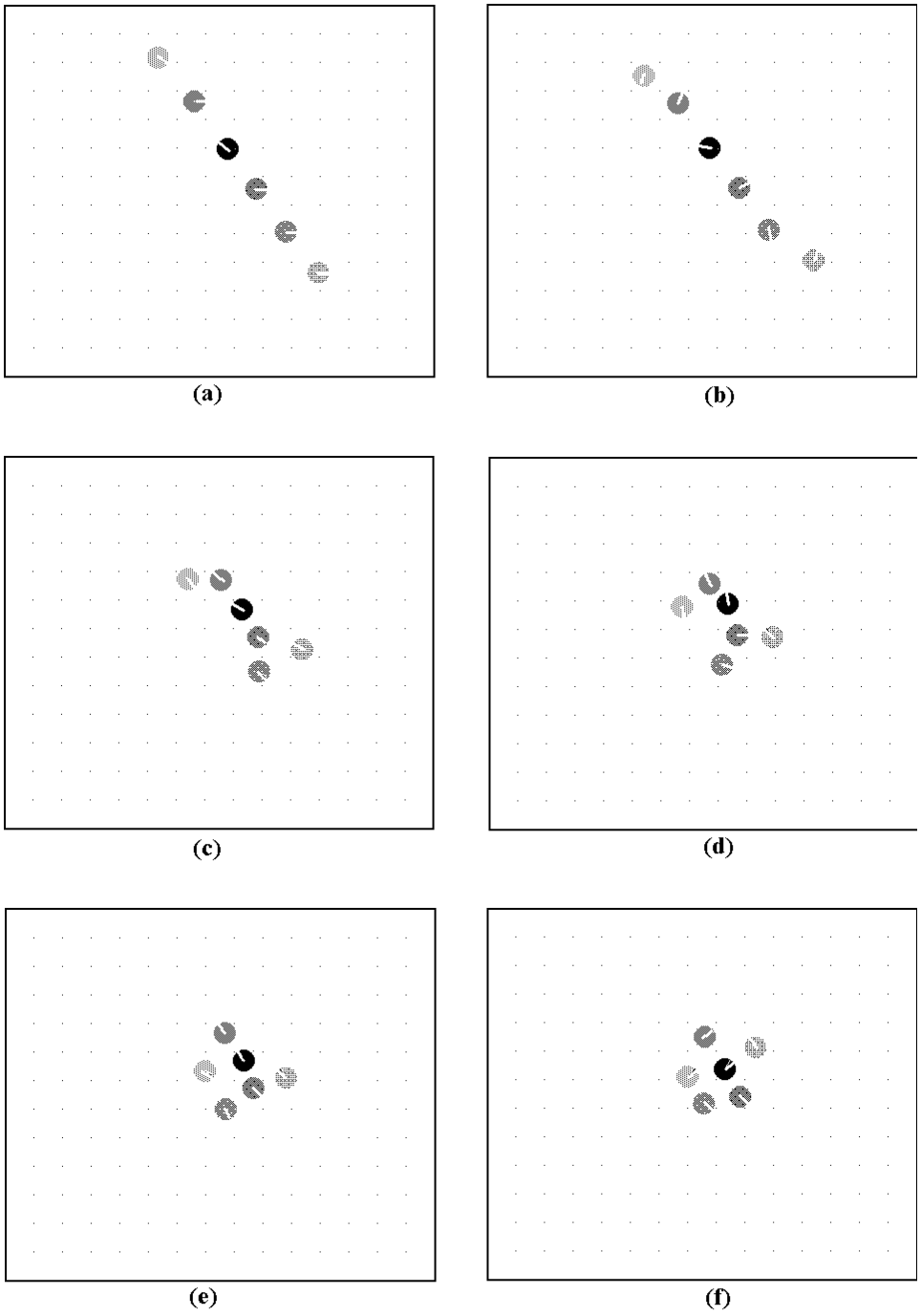

Figure 7. Selected images from a simulation of the merge algorithm: (a) the initial distribution from a line formation, (b)-(e) intermediate steps, and (f) the final distribution of the robots.

C. MERGE-THEN-CIRCLE ALGORITHM

In this section, a new algorithm is presented which allows robots to form a relatively large circle. Since sonar ranges are limited, a robot will not be able to see robots on the other side of a circle if the radius r is relatively large. In this new algorithm, each robot relies on the position information of the two closest robots, and does not use position information of the furthest robot. In this way, robots are able to form a circle with a diameter greater than the sensor range limits.

The algorithm is divided into two stages: first converge all robots into a single cluster by using the merge algorithm and then diverge them from the cluster to form a circle. The algorithm works as follows:

- **Step 1.** Robot R executes merge algorithm.
- **Step 2.** If the speed of robot R is less than some small value (1 inch/sec in the simulations) for N successive iterations (in simulations N is 20), robot R goes to sleep. It wakes up after T seconds to get the sensor data to determine the empty spaces around and sleeps again for another T seconds. T is empirically determined in simulations.
- **Step 3.** After waking up, if robot R sees an empty area based on the sensor data it got between the two sleep periods, it moves a distance r toward the middle of the empty area and goes back to sleep for another period of T seconds. If there is no empty space around, i.e., it is surrounded by other robots, it disregards previous data collected between the two sleep periods. It searches the surrounding area to look for an empty space. As soon as an empty space is detected, the robot travels $(r + d_o + r_o)$ toward the center of the empty space and then sleeps for T seconds.
- **Step 4.** Let R_{c1} and R_{c2} be the closest robots to robot R, one on each side of a line passing through from its position in the merged cluster to its present position. After waking up, robot R moves toward R_{c1} or R_{c2} until the distances to them are equal.
- **Step 5.** Robot R compares the desired diameter of the circle and the maximum sensor range (in simulations it is 206 inches). If the desired diameter is less than maximum sensor range which means robot R is able to detect robots on

the other side of the circle, then it positions itself r distance away from the centroid p_m of R_f, R_{c1} and R_{c2}. If not, it doesn't move.

In Step 2, robot R goes to sleep after N successive iterations, which happens when all robots are nearly merged. By waiting T seconds, robot R ensures all robots are merged. Taking the position information of robots between two sleep periods makes certain that all robots collect data before anyone wakes up. The sleep time at the end of Step 3 ensures that all robots reach their goal positions and form a rough circle. Step 5 utilizes all of the available information robot R can get to form a circle.

In the simulations, six robots are simulated. It is noted that if the number of robots is very large, robots will form a big cluster. In this case, at Step 3 if robot R is surrounded by other robots then it travels $(r + L(d_o+r_o))$, where L will be empirically determined in the simulations to optimize the formation of the rough circle.

In the simulations, at Step 3, robot R finds its goal position in its coordinate system by using the State vectors of the simulator which gives the position information of the robots in their coordinate systems. By extensive simulations, T is found set to 100 seconds.

Figure 8 shows a simulation of the merge-then-circle algorithm with a desired radius r = 120 inches. Figure 8(a) is the initial starting distribution. Figure 8(b) and Figure 8(d) are intermediate stages while robots execute merge algorithm. Figure 8(e) is the merged cluster after Step 2. Figure 8(g) is the rough circle after Step 3. Figure 8(h) is the final distribution of robots on a circle after Step 4. Robots don't move at Step 5 because the desired diameter of the circle (2 x 120 inches) is larger than 206 inches. The source code of the algorithm is in Appendix C.

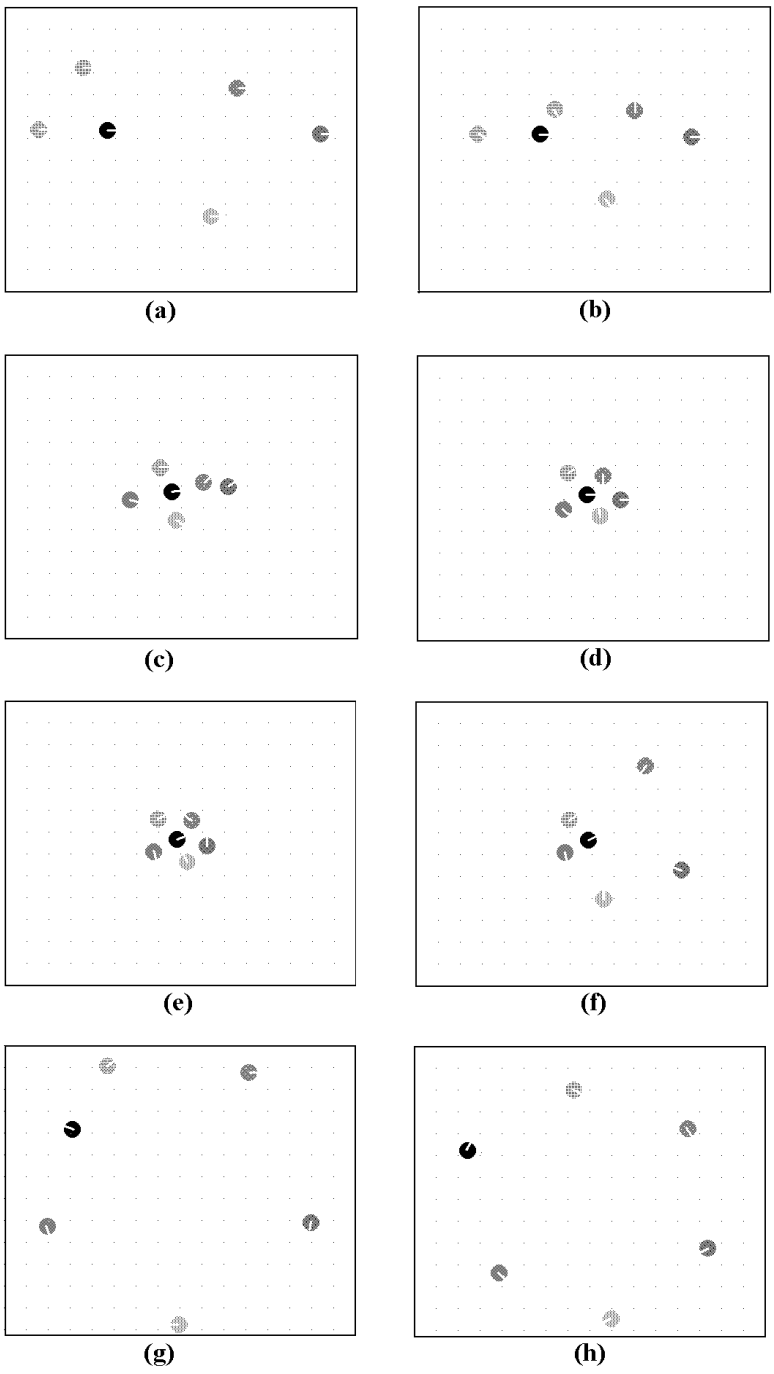

Figure 8. Selected images from a simulation of the merge-then-circle algorithm: (a) the initial distribution, (b)-(g) intermediate steps, and (h) the final distribution of the robots.

D. LIMITED RANGE ALGORITHM

In this section, the robots are initially placed at random in a large rectangular field. The field is so large that a robot may not see other robots due to limited sensor range. The objective is again to form specific geometric patterns with the distributed mobile robots. Even though the field is assumed to be rectangular in shape, its size is unknown. For all robots in the large field to form a circle or a line, one possible method is to have each robot search for all other robots and then execute a formation algorithm. Here an alternative method is proposed which is based heavily on the fact that the field is rectangular. All robots converge to the center of the field before executing any formation algorithm. This method can be described as follows:

- **Step 1**. Starting from its initial position, robot *R* moves straight until it reaches a wall (an edge of the field). It may need to avoid other robots before reaching a wall.
- **Step 2.** Robot *R* follows the edges of the field in counterclockwise direction until it has encountered three corners. It records the coordinates of the first and third corners.
- **Step 3.** It computes the center point p_m of the field, which is the middle point between the first and third corners.
- **Step 4.** It converges to p_m and goes to sleep for *T* seconds. The sleep mode is waiting until all robots converge. Time *T* is determined by a worst case analysis.
- **Step 5.** After waking up, it executes any formation algorithm described in previous sections.

In the simulations, a simple collision avoidance strategy is adopted (it is called panic mode). The strategy is that if robot *R* detects another robot very close to itself (it is implemented as 20 inches) in the direction of its move, it stops and turns left to avoid collisions. In Step 2 robot *R* detects the edge of the field by the following computations. If sonars adjacent to sonar S_{min} return approximately the same or very close values (in simulations robot *R* checks if the difference is less than 10 inches or not), then robot *R*

considers that sonar S_{min} gives the distance to the edge of the field, where S_{min} stands for one of the sonar sensors which returns the minimum value.

In the simulations, after robot *R* detects an edge, it follows the edge counterclockwise using the following behaviors:

- Case 1. If ($12 < S_{min} < 5$), then it turns left.
- Case 2. If ($4 < S_{min} < 12$), then it turns right.
- Case 3. If ($S_{min} = 12$), then it moves straight.

The simulator simulates robots with 16 sonar sensors and they are numbered from zero to 15 counterclockwise.

During the following edges, robot *R* counts corners. If it detects that it is getting closer to an object in the direction of its motion for *N* successive iterations (in simulations *N* is 10), it considers that there is a corner and increases a corner counter. If the distance to the corner becomes less than 20 inches, then robot R turns left, and continues to follow the edge. Only the coordinates of first and third corners are recorded. So when robot *R* reaches the third corner, it is able to compute the coordinates of the center point p_m which is the middle point of the first and third corners.

In simulations, robot *R* uses potential field algorithm to converge to the center of the field while executing Step 4. If robot *R* detects the center is blocked by another robot, then it stops within 20 inches of that robot. If center is not blocked then it stops after arriving at the center. After robots converge at the center, they sleep for a period of *T* seconds, which is determined by a worst case analysis. *T* is the elapsed time between the first and the last robot arrivals at the center. The following situation is the worst-case, which makes *T* a maximum. At the beginning, if robot *R* is very close to one of the corners and finds that corner, and at the same time one of the other robots is close to the opposite corner and cannot detect the edge, then this makes *T* a maximum. Thus *T* is calculated by dividing half of the total circumference of the field by the speed of the robots (in simulations robot speed is set at a constant 10 inches/sec). By waiting *T* seconds in Step 4,

robot *R* ensures all robots merged to the center of the field. It is noted that in simulations robots don't get closer while following the edges because the speeds of the robots don't change.

In this algorithm, robots are considered to have their own coordinate systems. If it is assumed that robots don't have any coordinate system, then robots must use a different algorithm to converge to the center of the field. In this new algorithm, instead of recording the coordinates of the first and third corners at Step 2, robot *R* measures the distance between the corners while following the edges and records the distances. In Step 3 it calculates the distance to p_m from the third corner by using the Pythagorean theorem (Figure 9). Step 4 and Step 5 are the same as before.

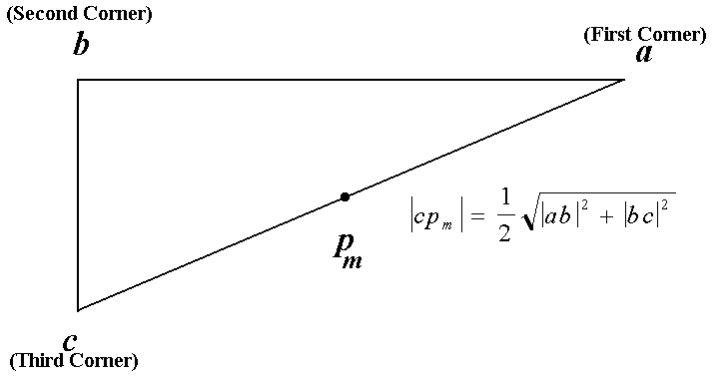

Figure 9. Pythagorean theorem for calculating the center of the rectangularly shaped field.

In simulations, at Step 5, the merge-then-circle algorithm is executed to form a circle. It is noted that the sleep times in the merge-then-circle algorithm will be the same as *T* in Step 4. Figure 10 depicts a simulation result of this algorithm. Figure 10(a) shows an initial distribution of robots. In Figure 10(c), robots are following the edges of the field. Figure 10(f) is a merged cluster at the center of the field. Figure 10(g) is a rough circle occurring in the intermediate steps of the merge-then circle algorithm. Figure 10(h) is the final distribution of the robots on a circle after completion of Step 5. The source code of the algorithm is in Appendix D.

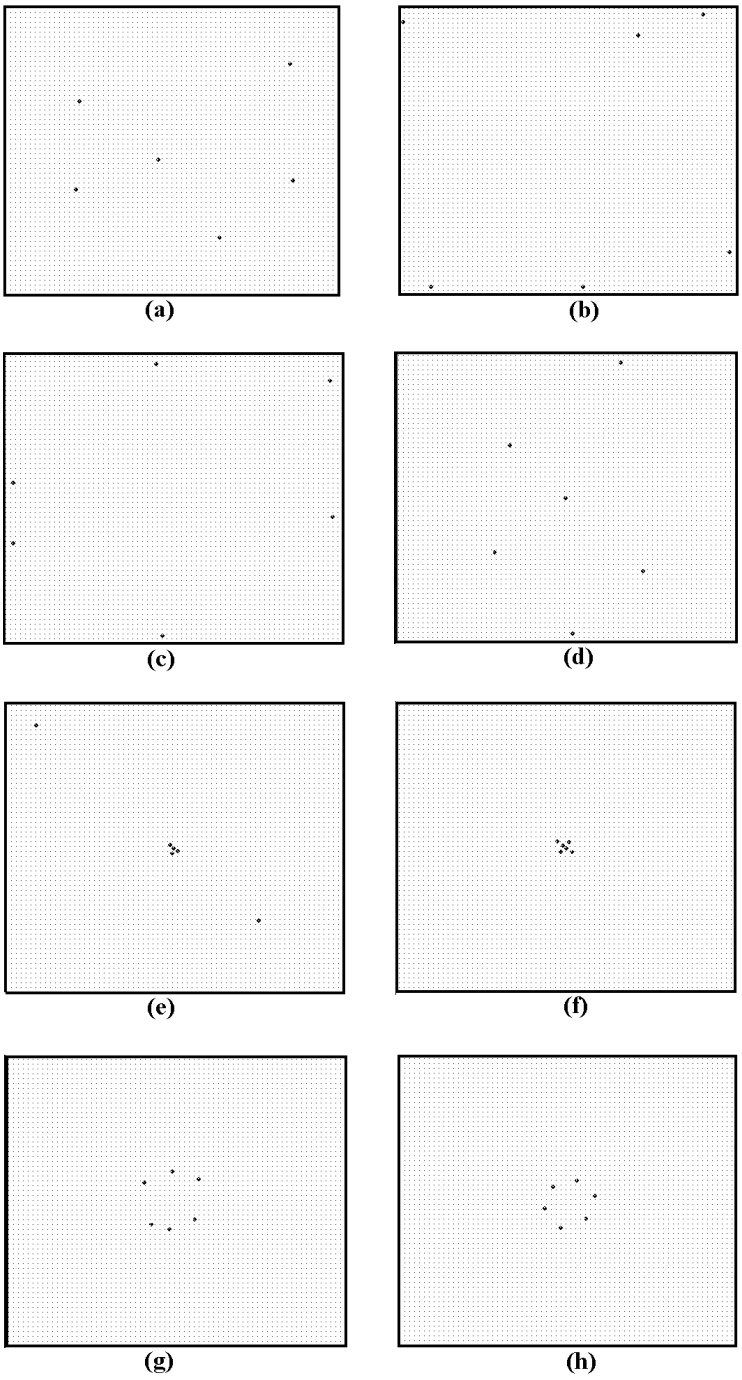

Figure 10. Selected images of a simulation of the limited range algorithm: (a) the initial distribution, (b)-(g) intermediate steps, and (h) the final distribution of the robots.

V. CONCLUSION AND RECOMMENDATION

In this thesis, formation problems for distributed autonomous mobile robots are analyzed by considering the physical dimensions of robots. All algorithms explained in this thesis are simulated using the Nomad 200 Robot Simulator. This simulator represents robots with their physical constraints. By assuming the physical robots, new algorithms for the formation of a line, circle and cluster are developed. Formation problems for mobile robots, distributed in a large rectangular field with limited sensor ranges, are also studied and simulated. In the simulations, the potential field method is adopted for collision avoidance.

The simulation results of the new algorithms, described in this thesis, indicate that these algorithms can be used in various fields currently using centralized control, like factory automation projects, operations in hazardous environments, planetary and space explorations, and military applications, such as so called “Battlefield of the Future” scenarios.

Existing formation algorithms [Ref. 1, 2, 3, 4, 5] do not work well on physical mobile robots because of their assumptions that robots are dimensionless points and are equipped with perfect sensors. However physical robots have limited sensor ranges. The new algorithms described in this thesis are developed for physical robots with consideration of sensor range limitations.

In this thesis the algorithms are developed for obstacle-free spaces. In future research these algorithms can be improved by considering obstacles in the operation field.

APPENDIX A. THE SIMULATION PROGRAM CODE FOR LEAST-SQUARE LINE ALGORITHM

```
/*********************************************************
This is a C program used for the simulations to simulate least-square
line algorithm by using potential field method. Developed by Okay
Albayrak. Last Modified in May 1996. This program has one input
argument. Usage <function> <Robot_ID> number.
*********************************************************/

/*** Include Files ***/
#include "Nclient.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*** Constants ***/
#define TRUE 1
#define FALSE 0
#define PI 3.1415926

/*** Function Prototypes ***/
void GetSensorData(void); /* This function returns sensor data */

void Movement(void);/*This function controls robot motions*/

int  sign(int);

void potential(void); /* This function implements potential field */

/*** Global Variables ***/
long SonarRange[16]; /* array of sonar readings (inches) */

long IRRange[16]; /*array of infrared readings (no units) */

int BumperHit = 0; /* boolean value */

/*the current robot configuration; x, y, steering_angle, turret_angle (x
and y are in tenth of inches, and angles are in tenth of degrees )*/
long robot_config[4];

long goal_config[4]; /*the goal configuration of the robot*/

/* sonar sensor numbers which returns minimum and maximum distances */
int minreturn, maxreturn;

/* minimum and maximum distances returned by the sonar sensors */
long mindist, maxdist;

double xgoal, ygoal; /* coordinates of destination in robot's coordinate
system */

/* the desired translation and steering velocity in 1/10 inch/sec and
1/10 deg/sec */
```

```
int tvel, svel;

int Robot_ID; /* represents robot number */

/*** Main Program ***/
main (unsigned int argc, char* argv[])
{
      int i;
      int order[16];
      int oldx, oldy;
      Robot_ID = atoi(argv[1]) ;
      printf("argv[1]= %s \n",argv[1]);
      printf("Robot_ID= %d \n",Robot_ID);

      /* Enter robot's number */
      if (argc!=2) {
            printf("please enter 1 parameters besides the command\n");
            exit();
      }

      /* This version of Nomad Robot 200 simulator can only simulate up
      to six robots */
      if ( (Robot_ID<1) || ( Robot_ID>6) ) {
            printf("Robot ID must be between 1 and 6 ");
            exit();
      }

      /* Communication port with robot and host computer */
      SERV_TCP_PORT=7772 ;

      /* Connect to Nserver. */
      connect_robot(Robot_ID);

      /* Initialize Smask and send to robot. Smask is a large array that
      controls which data the robot returns back to the server. This
      function tells the robot to give us everything. */
      init_mask();

      /* Configure timeout (given in seconds). This is how long the
      robot will keep moving if you become disconnected. Set this low if
      there are walls nearby. */
      conf_tm(1);

      /* Sonar setup: configure the order in which individual sonar
      units fire. In this case, fire all units in counter-clockwise
      order (units are numbered counter-clockwise starting with the
      front sonar as zero). The conf_sn() function takes an integer and
      an array of at most 16 integers. If less than 16 units are to be
      used, the list must be terminated by a element of value -1. See
      the IR setup below for an example of this. The single integer
      value passed controls the time delay between units in multiples of
      four milliseconds. */
      for (i = 0; i < 16; i++)
            order[i] = i;
      conf_sn(1,order);
```

```
	/* Infrared setup: only use the front 8 sensors as a last resort.
	The IR sensors are not useful for gauging distances accurately,
	and are thus only used to determine the  presence of obstacles
	that are missed by the sonar system. */
	for (i = 0; i < 16; i++)
		order[i] = i;
	conf_ir(1,order);

	/* Unfortunately, the robot can talk... */
	tk("Let's make a line.");

	/* Zero the robot. This aligns the turret and steering angles. The
	repositioning junk is neccessary to allow the user to position the
	robot. This is needed for real robots. */
	/*
	oldx = State[34];  /* State vector 34 and 35 give the coordinates
	of the robot */
	oldy = State[35];
	zr();
	ws(1,1,1,20);
	place_robot(oldx, oldy, 0, 0);
	*/

	/* Main loop. */
	while (!BumperHit)
	{
		GetSensorData();
		Movement();
	} /* end of the while statement */

	/* Disconnect. */
	vm(0,0,0);
	disconnect_robot(Robot_ID);

} /* end of the main function */


/* Movement(). This function is responsible for using the sensor data to
direct the robot's motion appropriately. */
void Movement (void)
{
	/* Variables are defined here. */
	int i;

	int sum = 1; /* This variable is used to find how many robots are
	visible */

	double x[16], y[16]; /* x and y coordinates of the robots seen by
	sonar sensor */

	double Ex , Ex2, Ey, Exy; /* Variables used to calculate least-
	square line fitting */

	double c0, nc1, c1, distance;

	/* Initialization of the variables */
```

```
Ex = 0.0; /* Used to calculate sum of x coordinates of the visible
robots. */

Ex2 = 0.0; /* Used to calculate sum of squares of x coordinates of
the robot. */

Ey = 0.0; /* Used to calculate sum of x coordinates of the visible
robots. */

Exy = 0.0; /* Used to calculate sum of multiplication of  x and y
coordinates. */

/* This loop is for calculating sum, Ex, Ey, Ex2, Exy */
for (i = 0; i<16 ; i++)
{
      x[i] = 0.0;
      y[i] = 0.0;

      if (SonarRange[i] < 255)
      {
            x[i] = ( (double)(SonarRange[i]) + 8.81) * cos (
            (double)(i) * 0.39);

            y[i] = ( (double)(SonarRange[i]) + 8.81) * sin (
            (double)(i) * 0.39);

            sum++; /* calculates the number of visible robots
            including itself */

      } /* end of if statement */

      Ex = Ex + x[i]; /* Calculates sum of x coordinates of the
      visible robots. */

      Ex2 = Ex2 + (x[i]*x[i]); /* Calculates sum of squares of x
      coordinates */

      Ey = Ey + y[i]; /* Calculates sum of x coordinates of the
      visible robots. */

      Exy = Exy+(x[i]*y[i]); /* Calculates sum of multiplication
      of  x and y coordinates. */

} /* end of for loop */

printf ("sum = %d\n",sum);
printf ("Ex = %f\n",Ex);
printf ("Ey = %f\n",Ey);
printf ("Ex2 = %f\n",Ex2);
printf ("Exy = %f\n",Exy);

/* Ex or Ey  equals to zero, if there is no visible robots or all
the robots are in the line */
if ( (fabs(Ex) < 0.01) || (fabs(Ey) < 0.01))
{
      xgoal = 0.0;
```

```
      ygoal = 0.0;

      printf("Ex < 0.01 || Ey < 0.01");

} /* end of if statement */

else
{
      /* c0 is the slope of the least-square line fit */
      c0 = ( (Ex*Ey)-((double)(sum)*Exy) ) / (((Ex*Ex) -
      ((double)(sum)*Ex2) );

      /* nc1 and c1 calculates the same value which is the y-
      intercept of the least square line fit*/

      nc1 = (Ey - (Ex*c0)) / (double)(sum);

      c1= ((Ey*Ex2) - (Exy*Ex)) / ( ((double)(sum)*Ex2) - (Ex*Ex)
      );

      printf ("c0 = %f\n", c0);
      printf("nc1 = %f\n", nc1);
      printf ("c1 = %f\n", c1);

      /* xgoal and ygoal are the coordinates in the line segment
      */

      xgoal = (-c1 * c0) / ((c0*c0) + 1);
      ygoal = (c0*xgoal) + c1;

}/* end of else statement */

/* If only one robot is detected , then the robot moves to that
robot. Because the robot might be at the endpoint of the line
segment */
if ( (sum == 2) && (mindist > 25) )
{
      xgoal = ( (double)(mindist) + 8.81) *
      cos((double)(minreturn) * 0.39);

      ygoal = ( (double)(mindist) + 8.81) *
      sin((double)(minreturn) * 0.39);
}

/* robots uses potential field method to control its movements */
potential() ;

/* The simplest search algorithm; if there isn't any robot
detected, then the robot makes a big circle to search others. */
if (mindist==255){
      svel = 50;
      tvel = 100;
}

/* Set the robot's velocities. The first parameter is the robot's
translational velocity, in tenths of an inch per second. This
```

```
      velocity can be between -240 and 240. The second parameter is the
      steering velocity, and the third is the turret velocity. The units
      of the latter two are tenths of a degree per second, and can be
      between -450 and 450. The same value is given for these  two so
      that the turret is always facing the direction of motion. */
      vm(tvel,svel,svel);

} /* end  of the function */


/* Read in sensor data and load into arrays. */
void GetSensorData (void)
{
      int i;

      /* Read all sensors and load data into State array. */
      gs();

      /* Read State array data and put readings into individual arrays.
      */
      for (i = 0; i < 16; i++)
      {
            /* Sonar ranges are given in inches, and can be between 6
            and 255, inclusive. */
            SonarRange[i] = State[17+i];

            printf("SonarRange[%d] : %d \n", i, SonarRange[i]);

            /* IR readings are between 0 and 15, inclusive. This value
            is inversely proportional to the light reflected by the
            detected object, and is thus proportional to the distance of
            the object. Due to the many environmental variables
            effecting the reflectance of infrared light, distances
            cannot be accurately ascribed to the IR readings. */
            IRRange[i] = State[1+i];
      }

      /* The robot configuration parameters (x,y,steering_angle,and
      turret_angle) are stored in State[34], State[35], State[36], and
      State[37].  */
      for (i = 0; i < 4; i++)
            robot_config[i] = State[34+i];

      /* Check for bumper hit. If a bumper is activated, the
      corresponding bit in State[33] will be turned on. Since we don't
      care which bumper is hit, we thus only need to check if State[33]
      is greater than zero. */
      if (State[33] > 0) {
            BumperHit = 1;
            tk("Ouch.");
            printf("Bumper hit!\n");
      }

      /* Calculate which sonar returns minimum distance */
      minreturn = 0;
      for (i = 1 ; i < 16 ; i++)
```

```
	{
		if (SonarRange[i] < SonarRange[minreturn])
			minreturn = i;
	}

	mindist = SonarRange[minreturn];

	printf("minreturn : %d mindist: %d \n",minreturn,mindist);

	/* Calculate which sonar returns maximum distance */
	maxreturn = minreturn ;
	for (i = 0 ; i < 16 ; i++)
	{
		if ((SonarRange[i] >= SonarRange[maxreturn]) &&
		(SonarRange[i]<255))
			maxreturn = i;
	} /* end of for loop */

	maxdist = SonarRange[maxreturn];

	printf("maxreturn : %d maxdist: %d \n", maxreturn,maxdist);

	/* Notice the user if there is no contact */
	if (mindist == 255)
		printf("There is no object around");

} /* End of the GetSensorData() function */


/* Sign function. It returns 1 if x is positive, and returns -1
otherwise */
int sign(int x)
{
	return x>0?1:-1;
} /* end of sign function */


/* The potential field method is used for motion control and collision
avoidance */
void potential() {

	/* Various constants for computing attractive and repulsive forces
	should be  defined here, e.g., */

	double rho_0 = 20.0; /* cut-off distance of the repulsive force */

	double scale = 15.0 ; /* scaling factor for attractive force */

	double eta = 12000.0; /* repulsive force scaling factor */

	double d  = 100.0 ; /* saturation in attractive force */

	double gain_tvel = 0.1; /* translational velocity gain */

	double gain_svel = 200.0; /* rotational velocity gain */
```

```
int i;

/* attractive and repulsive forces are defined */
double F_att[2], F_rep[2], F_tol[2] ;

double rho_float;
double distance ;

printf ("xgoal = %f\n", xgoal);
printf ("ygoal = %f\n", ygoal);

/* the distance between present location and destination is
calculated */
distance = hypot(xgoal,ygoal) ;

printf("distance : %f\n ", distance);

/* parabolic-well definition of the attractive force */
if (distance <= d)
{
      F_att[0] = scale*xgoal ;
      F_att[1] = scale*ygoal ;
}

/* conic-well definition of attractive force */
else
{
      F_att[0] = scale*d*(xgoal/distance) ;
      F_att[1] = scale*d*(ygoal/distance) ;
}

/* compute the repulsive force in the robot coordinate */
F_rep[0] = 0.0; /* repulsive force implied on x-axis */
F_rep[1] = 0.0; /* repulsive force implied on y-axis */
for (i = 0; i <= 15; i++)
{
      rho_float = (double) (SonarRange[i]);
      if (rho_float < rho_0)
      {
            F_rep[0] += -eta * (1.0/rho_float - 1.0/rho_0) *
            cos((double)(i) * 0.392699) / (rho_float);

            F_rep[1] += -eta * (1.0/rho_float - 1.0/rho_0) *
            sin((double)(i) * 0.392699) / (rho_float);

      } /* end of if  statement */

} /* end of for loop */

/* compute the total force in the robot coordinates */
F_tol[0] = F_att[0] + F_rep[0];
F_tol[1] = F_att[1] + F_rep[1];


/* set the translational velocity */
tvel = (int) (gain_tvel * F_tol[0]);
```

```
        /* set the rotational velocity */
        if (F_tol[0] == 0.0) svel = 0;

        else
        {
              svel = (int)(gain_svel * sin(atan2(F_tol[1],F_tol[0])));

              svel = svel * sign( (int)(F_tol[0]) );
        }

        /* limit the translational and rotational velocities */
        if (abs(tvel) > 230)
              tvel = 230 * sign(tvel);

        if (abs(svel) > 450)
              svel = 450 * sign(svel);

} /* end of potential() function */
```

APPENDIX B. THE SIMULATION PROGRAM CODE FOR MERGE ALGORITHM

```
/*********************************************************
This is a C program used for the simulations of merge algorithm.
Developed Okay Albayrak. Last modified in May 1996. Usage <function>
<Robot_ID>. This program merges the robots to a cluster.
*********************************************************/

/*** Include Files ***/
#include "Nclient.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*** Constants ***/
#define TRUE 1
#define FALSE 0
#define PI 3.1415926

/*** Function Prototypes ***/
void GetSensorData(void); /* This function returns sensor data */

void Movement(void); /* This function controls robot motions */

int  sign(int); /* This functions returns 1 for positive values and -1
for negative values */

void potential(void); /* This function implements potential field */

/*** Global Variables ***/
long SonarRange[16]; /* array of sonar readings (inches) */

long IRRange[16]; /* array of infrared readings (no units) */

int  BumperHit = 0; /* boolean value */

/*the current robot configuration; x, y, steering_angle, turret_angle (x
and y are in tenth of inches, and angles are in tenth of degrees )*/
long robot_config[4];

long goal_config[4]; /* the goal configuration of the robot */

/* constants for computing attractive and repulsive forces in potential
field*/
double F_att[2], F_rep[2], F_tol[2];

/* sonar sensor numbers which returns minimum and maximum distances */
int minreturn, maxreturn;

/* minimum and maximum distances returned by the sonar sensors */
long mindist, maxdist;
```

```
double xgoal, ygoal; /* coordinates of destination in robot's coordinate
system */

/* the desired translation and steering velocity in 1/10 inch/sec and
1/10 deg/sec */
int tvel, svel;

int Robot_ID; /* represents robot number */

/*** Main Program ***/
main (unsigned int argc, char* argv[])
{
      int i, index;
      int order[16];
      int oldx, oldy;
      Robot_ID = atoi(argv[1]) ;
      printf("argv[1]= %s \n",argv[1]);
      printf("Robot_ID= %d \n",Robot_ID);

      /* Enter robot's number */
      if (argc!=2) {
            printf("please enter 1 parameters besides the command\n");

            exit();
      }

      /* Nomad Robot 200 simulator can only simulate up to six robots */
      if ( (Robot_ID<1) || ( Robot_ID>6) ) {
            printf("Robot ID must be between 1 and 6 ");
            exit();
      }

      /* This is the communication port between robot and host server.
      */
      SERV_TCP_PORT=7772;

      /* Connect to Nserver. */
      connect_robot(Robot_ID);

      /* Initialize Smask and send to robot. Smask is a large array that
      controls which data the  robot returns back to the server. This
      function tells the robot to give us everything. */
      init_mask();

      /* Configure timeout (given in seconds). This is how long the
      robot will keep moving if you become disconnected. Set this low if
      there are walls nearby. */
      conf_tm(1);

      /* Sonar setup: configure the order in which individual sonar
      units fire. In this case, fire all units in counter-clockwise
      order (units are numbered counter-clockwise starting with the
      front sonar as zero). The conf_sn() function takes an integer and
      an array of at most 16 integers. If less than 16 units are to be
      used, the list must be terminated by a element of value -1. See
      the IR setup below for an example of this. The single integer
```

```
      value passed controls the time delay between units in multiples of
      four milliseconds. */
      for (i = 0; i < 16; i++)
            order[i] = i;
      conf_sn(1,order);

      /* Infrared setup: only use the front 8 sensors as a last resort.
      The IR sensors are not useful for gauging distances accurately,
      and are thus only used to determine the presence of obstacles that
      are missed by the sonar system. */
      for (i = 0; i < 16; i++)
            order[i] = i;
      conf_ir(1,order);

      /* Unfortunately, the robot can talk... */
      tk("Let's make a cluster.");

      /* Zero the robot. This aligns the turret and steering angles. The
      repositioning junk is neccessary to allow the user to position the
      robot. This is needed for real robots. */
      /*
      oldx = State[34];
      oldy = State[35];
      zr();
      ws(1,1,1,20);
      place_robot(oldx, oldy, 0, 0);
      */

      /* Main loop. */
      while (!BumperHit)
      {
            GetSensorData();
            Movement();
      } /* end of the while statement */

      /* Disconnect. */
      vm(0,0,0); /* before disconnecting zero all the velocities */

      disconnect_robot(Robot_ID);

} /* end of the main function */


/* Movement(). This function is responsible for using the sensor data to
direct the robot's motion appropriately. */
void Movement (void)
{
      /* Variables are defined here. */
      int i;
      double x1, x2, y1, y2 ;

      /* Coordinates of the closest robot */
      x1 = ( (double)(mindist) + (8.81)) * cos ( (double)(minreturn) *
      0.39);
```

```
        y1 = ( (double)(mindist) + (8.81)) * sin ( (double)(minreturn) *
        0.39);

        /* Coordinates of the furthest  visible robot */
        x2 = ( (double)(maxdist) + (8.81)) * cos ( (double)(maxreturn) *
        0.39);
        y2 = ( (double)(maxdist) + (8.81)) * sin ( (double)(maxreturn) *
        0.39);

        /* Coordinates of the goal points in robot coordinate system */
        xgoal = (x2 + x1)/2.0;
        ygoal = (y2 + y1)/2.0;

        /* If initial distribution is a line this will break the line */

        if (abs(minreturn - maxreturn) == 1 ) || (abs(minreturn -
        maxreturn) == 15) || (minreturn==maxreturn) )
        {
              if ( abs(mindist-maxdist)<=4 )
              {
                    xgoal = ((double)(mindist)+8.81) * cos(
                    ((double)(minreturn)*0.39) - (60.0 *(PI/180.0)) );

                    ygoal = ((double)(mindist)+8.81) * sin(
                    ((double)(minreturn)*0.39) - (60.0 *(PI/180.0)) );
              }
        }

        /* The robot uses potential field method to move to its goal point
        */
        potential() ;

        /* The simplest search algorithm; if there isn't any robot
        detected, then the robot makes a big circle to search others. */
        if (mindist==255){
              svel = 50;
              tvel = 100;
       }

        /* Set the robot's velocities. The first parameter is the robot's
        translational velocity, in tenths of an inch per second. This
        velocity can be between -240 and 240. The second parameter is the
        steering velocity, and the third is the turret velocity. The units
        of the latter two are tenths of a degree per second, and can be
        between -450 and 450. The same value is given for these  two so
        that the turret is always facing the direction of motion. */
        vm(tvel,svel,svel);

} /* end of function */


/* Read in sensor data and load into arrays. */
void GetSensorData (void)
{
        int i;
```

```
/* Read all sensors and load data into State array. */
gs();

/* Read State array data and put readings into individual arrays.
*/
for (i = 0; i < 16; i++)
{
      /* Sonar ranges are given in inches, and can be between 6
      and 255, inclusive. */
      SonarRange[i] = State[17+i];

      printf("SonarRange[%d] : %d\n", i, SonarRange[i]);

      /* IR readings are between 0 and 15, inclusive. This value
      is inversely proportional to the light reflected by the
      detected object, and is thus proportional to the distance of
      the object. Due to the many environmental variables
      effecting the reflectance of infrared light, distances
      cannot be accurately ascribed to the IR readings. */
      IRRange[i] = State[1+i];
}

/* The robot configuration parameters (x,y,steering_angle,and
turret_angle) are stored in State[34], State[35], State[36], and
State[37].  */
for (i = 0; i < 4; i++)
      robot_config[i] = State[34+i];

/* Check for bumper hit. If a bumper is activated, the
corresponding bit in State[33] will be turned on. Since we don't
care which bumper is hit, we thus only need to check if State[33]
is greater than zero. */
if (State[33] > 0)
{
      BumperHit = 1;
      tk("Ouch.");
      printf("Bumper hit!\n");
}

/* Calculate which sonar returns minimum distance */
minreturn = 0;
for (i = 1 ; i < 16 ; i++) {
      if (SonarRange[i] < SonarRange[minreturn])
      minreturn = i;
}

mindist = SonarRange[minreturn];
printf("minreturn:%d mindist: %d\n",minreturn,mindist);


/* Calculate which sonar returns maximum distance */
maxreturn = minreturn ;
for (i = 0 ; i < 16 ; i++)
{
      if ((SonarRange[i] >= SonarRange[maxreturn]) &&
      (SonarRange[i]<255))
```

```
                maxreturn = i;
        }

        maxdist = SonarRange[maxreturn];
        printf("maxreturn:%d maxdist: %d\n",maxreturn,maxdist);

        /* Notice the user if there is no contact */
        if (mindist == 255)
        printf("There is no object around");

} /* end of the GetSensorData() function */


/* Sign function.  It returns 1 if x is positive, and returns -1
otherwise */
int sign(int x)
{
        return x>0?1:-1;
} /* end of the sign() function */


/* The potential field method is used for motion control and collision
avoidance */
void potential() {

        /* Various constants for computing attractive and repulsive forces
        should be defined here, e.g., */

        double rho_0 = 50.0; /* cut-off distance of the repulsive force */

        double scale = 10.0 ; /* scaling factor for attractive force */

        double eta = 12000.0; /* repulsive force scaling factor */

        double d  = 100.0 ; /* saturation in attractive force */

        double gain_tvel = 0.1; /* translational velocity gain */

        double gain_svel = 200.0; /* rotational velocity gain */

        int i;

        /* attractive and repulsive forces are defined */
        double F_att[2], F_rep[2], F_tol[2] ;

        double rho_float;

        double distance ;

        printf ("xgoal = %f\n", xgoal);
        printf ("ygoal = %f\n", ygoal);

        /* the distance between present location and destination is
        calculated */
        distance = hypot(xgoal,ygoal) ;
```

```
printf("distance : %f\n ", distance);

/* parabolic-well definition of the attractive force */
if (distance <= d)
{
      F_att[0] = scale*xgoal ;
      F_att[1] = scale*ygoal ;
}

/* conic-well definition of attractive force */
else
{
      F_att[0] = scale*d*(xgoal/distance) ;
      F_att[1] = scale*d*(ygoal/distance) ;
}

/* compute the repulsive force in the robot coordinate system */
F_rep[0] = 0.0;
F_rep[1] = 0.0;
for (i = 0; i <= 15; i++)
{
      rho_float = (double) (SonarRange[i]);
      if (rho_float < rho_0)
      {
            F_rep[0]+= -eta * (1.0/rho_float - 1.0/rho_0) *
            cos((double)(i) * 0.392699)/(rho_float);

            F_rep[1]+= -eta * (1.0/rho_float - 1.0/rho_0) *
            sin((double)(i) * 0.392699)/(rho_float);

      } /* end of if  statement */

} /* end of for loop */

/* compute the total force in the robot coordinates */
F_tol[0] = F_att[0] + F_rep[0];
F_tol[1] = F_att[1] + F_rep[1];


/* set the translational velocity */
tvel = (int) (gain_tvel * F_tol[0]);

/* set the rotational velocity */
if (F_tol[0] == 0.0) svel = 0;

else
{
      svel=(int)(gain_svel*sin (atan2(F_tol[1],F_tol[0])));

      svel = svel * sign( (int) (F_tol[0]) );
}

/*limit the translational and rotational velocities */
if (abs(tvel) > 230)
tvel = 230 * sign(tvel);
```

```
      if (abs(svel) > 450)
      svel = 450 * sign(svel) ;


} /* end of potential() function */
```

APPENDIX C. THE SIMULATION PROGRAM CODE FOR MERGE-THEN-CIRCLE ALGORITHM

```
/*********************************************************
This is a C program used for the simulations of merge-then-circle
algorithm. It is developed by Okay Albayrak, and it is last modified in
May 1996. Usage <function> <Robot_ID> <Radius of the Circle in 1/10
inch>. This function needs 2 parameters  first parameter is the robot id
and second one is the desired radius of the circle.
*********************************************************/

/*** Include Files ***/
#include "Nclient.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*** Constants ***/
#define TRUE 1
#define FALSE 0
#define PI 3.1415926

/*** Function Prototypes ***/
void GetSensorData(void); /* This function returns sensor data */

void Movement(void); /* This function controls robot motions */

int  sign(int); /* This function returns 1 for positive values, -1
otherwise */

void SwichToCircle(void);/*This functions forms a rough circle after all
robots merged*/

void Movement1(void);

void GoCircle(); /* This function makes robots form a homogeneous circle
after they form a rough circle */

void min2max(void);

void potential(void); /* This function implements potential field method
*/

/*** Global Variables ***/
long SonarRange[16]; /* array of sonar readings (inches) */

long IRRange[16]; /* array of infrared readings (no units)*/

int  BumperHit = 0; /* boolean value */

/* the current robot configuration; x, y, steering angle, turret angle.
x and y are in tenth of inches, and angles are in tenth of degrees */
long robot_config[4];
```

```
long goal_config[4];/*the goal configuration of the robot*/

double xgoal, ygoal; /* coordinates of the goal position in robot
coordinate system */

int tvel, svel; /* translation and steering velocities 1/10 inch/sec and
1/10 deg/sec */

int minreturn, maxreturn, secondminreturn ;

long mindist, maxdist, secondmindist ;

double r, R; /* desired radius of the circle */

int Robot_ID; /*Robot ID number can be between one and six*/

long SortedSonarRange[16]; /* This array has the sorted values of
SonarRange[16] */

int SortedSonarReturn[16];

int SleepTime = 100;

/*** Main Program ***/
main (unsigned int argc, char* argv[])
{
      int i, index;
      int order[16];
      int oldx, oldy;
      int SumStop = 0;

      Robot_ID = atoi(argv[1]);

      /* r is radius */
      r = (double)(atoi(argv[2])) ;
      R = r;

      printf("argv[1]= %s \n",argv[1]);
      printf("Robot_ID= %d \n",Robot_ID);

      if (argc!=3) {
            printf("please enter 2 parameters besides the command\n");
            exit();
      }

      if ( (Robot_ID<1) || ( Robot_ID>6) ) {
            printf("Robot ID must be between 1 and 6 ");
            exit();
      }

      /* This is the communication port between robot and host server.
      */
      SERV_TCP_PORT=7772;

      /* Connect to Nserver. */
      connect_robot(Robot_ID);
```

```
/* Initialize Smask and send to robot. Smask is a large array that
controls which data the  robot returns back to the server. This
function tells the robot to give us everything. */
init_mask();

/* Configure timeout (given in seconds). This is how long the
robot will keep moving if you become disconnected. Set this low if
there are walls nearby. */
conf_tm(1);

/* Sonar setup: configure the order in which individual sonar
units fire. In this case, fire all units in counter-clockwise
order (units are numbered counter-clockwise starting with the
front sonar as zero). The conf_sn() function takes an integer and
an array of at most 16 integers. If less than 16 units are to be
used, the list must be terminated by a element of value -1. See
the IR setup below for an example of this. The single integer
value passed controls the time delay between units in multiples of
four milliseconds. */
for (i = 0; i < 16; i++)
      order[i] = i;
conf_sn(1,order);

/* Infrared setup: only use the front 8 sensors as a last resort.
The IR sensors are not useful for gauging distances accurately,
and are thus only used to determine the presence of obstacles that
are missed by the sonar system. */
for (i = 0; i < 16; i++)
      order[i] = i;
conf_ir(1,order);

 /* Unfortunately, the robot can talk... */
tk("Start the program.");

/* Zero the robot. This aligns the turret and steering angles. The
repositioning junk is neccessary to allow the user to position the
robot. This is needed for real robots. */
/*
oldx = State[34];
oldy = State[35];
zr();
ws(1,1,1,20);
place_robot(oldx, oldy, 0, 0);
*/

/* Main loop. */
while (!BumperHit)
{
      GetSensorData();
      Movement();
      if ( (abs(tvel) < 11) && (mindist != 255) )
            SumStop++;

      printf("Merge->SwitchToCircle SumStop = %d\n",SumStop);
```

```
            if (SumStop > 25)
            {
                  st();
                  GetSensorData();
                  sleep(SleepTime);
                  SwichToCircle();

            } /* end of if statement */

      } /* end of while loop */

      /* Disconnect. */
      vm(0,0,0) ;
      disconnect_robot(Robot_ID);

} /* end of main function */


/* Movement(). This function is responsible for using the sensor data to
direct the robot's motion appropriately. */
void Movement (void)
{
       /* Variables are defined here. */
      int i;

      double x1, x2, y1, y2 ;

      /* Coordinates of the closest robot */
      x1 = ((double)(mindist)+(8.81)) * cos((double)(minreturn)*0.39);

      y1 = ((double)(mindist)+(8.81)) * sin((double)(minreturn)*0.39);

      /* Coordinates of the furthest  visible robot */
      x2 = ((double)(maxdist)+(8.81)) * cos((double)(maxreturn)*0.39);

      y2 = ((double)(maxdist)+(8.81)) * sin((double)(maxreturn)*0.39);

      /* Coordinates of the goal points in robot coordinate system */
      xgoal = (x2 + x1)/2.0;
      ygoal = (y2 + y1)/2.0;

      /* If initial distribution is a line this will help to break the
      line */
      if (abs(minreturn-maxreturn) == 1 ) || (abs(minreturn - maxreturn)
      == 15) || (minreturn==maxreturn) )
      {
            if ( abs(mindist-maxdist)<=4 )
            {
                  xgoal = ((double)(mindist) + 8.81) *
                  cos(((double)(minreturn)*0.39) - (60.0 *(PI/180.0)) );

                  ygoal = ((double)(mindist) + 8.81) *
                  sin(((double)(minreturn)*0.39) - (60.0 *(PI/180.0)) );
            }
      }
```

```
	/* The robot uses potential field method to move to its goal point
	*/
	potential() ;

	/* The simplest search algorithm; if there isn't any robot
	detected, then the robot makes a big circle to search others. */
	if (mindist==255)
	{
		svel = 50;
		tvel = 100;
	}

	/* Set the robot's velocities. The first parameter is the robot's
	translational velocity, in tenths of an inch per second. This
	velocity can be between -240 and 240. The second parameter is the
	steering velocity, and the third is the turret velocity. The units
	of the latter two are tenths of a degree per second, and can be
	between -450 and 450. The same value is given for these  two so
	that the turret is always facing the direction of motion. */
	vm(tvel,svel,svel);

}


/* Read in sensor data and load into arrays. */
void GetSensorData (void)
{
	int i;

	/* Read all sensors and load data into State array. */
	gs();

	/* Read State array data and put readings into individual arrays.
	*/
	for (i = 0; i < 16; i++)
	{
		/* Sonar ranges are given in inches, and can be between 6
		and 255, inclusive. */
		SonarRange[i] = State[17+i];
		printf("SonarRange[%d] : %d\n", i, SonarRange[i]);

		/* IR readings are between 0 and 15, inclusive. This value
		is inversely proportional to the light reflected by the
		detected object, and is thus proportional to the distance of
		the object. Due to the many environmental variables
		effecting the reflectance of infrared light, distances
		cannot be accurately ascribed to the IR readings. */
		IRRange[i] = State[1+i];
	}

	/* The robot configuration parameters (x,y,steering_angle,and
	turret_angle) are stored in State[34], State[35], State[36], and
	State[37].  */
	for (i = 0; i < 4; i++)
		robot_config[i] = State[34+i];
```

```
/* Check for bumper hit. If a bumper is activated, the
corresponding bit in State[33] will be turned on. Since we don't
care which bumper is hit, we thus only need to check if State[33]
is greater than zero. */
if (State[33] > 0)
{
      BumperHit = 1;
      tk("Ouch.");
      printf("Bumper hit!\n");
}

/* Calculate which sonar returns minimum distance */
minreturn = 0;
for (i = 1 ; i < 16 ; i++) {
      if (SonarRange[i] < SonarRange[minreturn])
            minreturn = i;
}

mindist = SonarRange[minreturn];
printf("minreturn:%d mindist: %d\n",minreturn,mindist);


/* Calculate which sonar returns maximum distance */
maxreturn = minreturn ;
for (i = 0 ; i < 16 ; i++) {
      if ((SonarRange[i] >= SonarRange[maxreturn]) &&
      (SonarRange[i]<255))
            maxreturn = i;
}

maxdist = SonarRange[maxreturn];
printf("maxreturn:%d maxdist: %d\n",maxreturn,maxdist);

/* find the second closest robot */
min2max();
secondminreturn = SortedSonarReturn[1];
secondmindist = SortedSonarRange[1] ;

if ((abs(SortedSonarReturn[0]-SortedSonarReturn[1])==1) || (
      abs(SortedSonarReturn[0]-SortedSonarReturn[1])==15) )
{
      secondminreturn = SortedSonarReturn[2];
      secondmindist = SortedSonarRange[2] ;

      if ( (abs(SortedSonarReturn[0]-SortedSonarReturn[2])==1) ||
      (abs(SortedSonarReturn[0]-SortedSonarReturn[2])==15) )
      {
            secondminreturn = SortedSonarReturn[3];
            secondmindist = SortedSonarRange[3] ;
      }
}

/* Notice the user if there is no contact */
if (mindist == 255)
printf("There is no object around");
```

```
} /* End of the GetSensorData() function */


/* Sign function.  It returns 1 if x is positive, and returns -1
otherwise */
int sign(int x)
{
      return x>0?1:-1;
}


/* The potential field method is used for motion control and collision
avoidance */
void potential() {

      /* Various constants for computing attractive and repulsive forces
      should be defined here, e.g., */

      double rho_0 = 50.0; /* cut-off distance of the repulsive force */

      double scale = 10.0 ; /* scaling factor for attractive force */

      double eta = 12000.0;/*repulsive force scaling factor*/

      double d  = 100.0 ; /*saturation in attractive force */

      double gain_tvel = 0.1; /* translational velocity gain */

      double gain_svel = 200.0;/*rotational velocity gain */

      int i;

      /* attractive and repulsive forces are defined */
      double F_att[2], F_rep[2], F_tol[2] ;

      double rho_float;

      double distance ;

      printf ("xgoal = %f\n", xgoal);
      printf ("ygoal = %f\n", ygoal);

      /* the distance between present location and destination is
      calculated */
      distance = hypot(xgoal,ygoal) ;
      printf("distance : %f\n ", distance);

      /* parabolic-well definition of the attractive force */
      if (distance <= d)
      {
            F_att[0] = scale*xgoal ;
      F_att[1] = scale*ygoal ;
      }

      /* conic-well definition of attractive force */
      else
```

```
      {
            F_att[0] = scale*d*(xgoal/distance) ;
            F_att[1] = scale*d*(ygoal/distance) ;
      }

      /*compute the repulsive force in the robot coordinate*/
      F_rep[0] = 0.0;
      F_rep[1] = 0.0;
      for (i = 0; i <= 15; i++)
      {
            rho_float = (double) (SonarRange[i]);
            if (rho_float < rho_0)
            {
                  F_rep[0] += -eta * (1.0/rho_float - 1.0/rho_0) * cos
                  ((double)(i) * 0.392699)/(rho_float);

                  F_rep[1] += -eta * (1.0/rho_float - 1.0/rho_0) * sin
                  ((double)(i) * 0.392699)/(rho_float);

            } /* end of if  statement */

      } /* end of for loop */

      /* compute the total force in the robot coordinates */
      F_tol[0] = F_att[0] + F_rep[0];
      F_tol[1] = F_att[1] + F_rep[1];


      /* set the translational velocity */
            tvel = (int) (gain_tvel * F_tol[0]);

      /* set the rotational velocity */
      if (F_tol[0] == 0.0) svel = 0;

      else
      {
            svel = (int)(gain_svel * sin(atan2(F_tol[1],F_tol[0])));

            svel = svel * sign((int)(F_tol[0]));

      }

      /*limit the translational  and rotational velocities */
      if (abs(tvel) > 230)
            tvel = 230 * sign(tvel);

      if (abs(svel) > 450)
            svel = 450 * sign(svel) ;

} /* end of potential() function */


/* This function forms a rough circle */
void SwichToCircle(void)
{
      int i;
```

```
int s, sm, opposite;
double xrgoal, yrgoal, alpha;
int SumStop = 0;
int CutOff = 0;

s = 0;
sm = 0;
for (i = 0; i<16; i++)
{
      if (SonarRange[i]==255)
      {
            s = s + i;
            sm++;

      } /* end of if statement */

} /* end of for loop */

/* If  the robot surrendered by other robots then the r distance
of this robot should be larger */
if ( sm<9 )
      r = r + ( (double)(mindist) * 10.0) ;

printf("s = %d\n", s);
printf("sm = %d\n",sm);

if (mindist != 255)
{
if (SonarRange[15]==255)
      {
            i = 0;
            while(SonarRange[i]==255 )
            {
                  s = s + 16;
                  i++;

            } /* end of while loop */
      }
}

printf("s = %d\n", s);
printf("sm = %d\n",sm);

if (sm == 0)
      opposite = 0;
else
      opposite = s / sm;  /* this gives the direction of the empty
      space */

if (opposite == 0) opposite = 16;

/* If there is no empty space available search for it 33 times */
while ( ((SonarRange[(opposite+1)%16] != 255) ||
(SonarRange[(opposite-1)%16] != 255) || (SonarRange[opposite % 16]
!= 255)) && (CutOff < 33) )
{
```

```
        CutOff++;
        printf("CutOff = %d\n",CutOff);
        opposite++;
        GetSensorData();
 }


 /*if empty space is not found look for empty direction*/
 while (SonarRange[opposite % 16] != 255)
 {
        opposite++;
        GetSensorData();
        printf("Second Chance");
}

 opposite = opposite % 16;
 printf("opposite = %d\n",opposite);
 printf("r = %f\n",r);

 /* calculate the destination point to form a rough circle */
 xrgoal = r * cos( (double)(opposite) * 0.392699);
 yrgoal = r * sin( (double)(opposite) * 0.392699);
 alpha  = (double)(robot_config[2]) * PI/(10.0*180.0);


 printf("xrgoal = %f\n",xrgoal);
 printf("yrgoal = %f\n",yrgoal) ;
 printf("robot_config[0] = %d\n",robot_config[0]);
 printf("robot_config[1] = %d\n",robot_config[1]);
 printf("robot_config[2] = %d\n",robot_config[2]);

 /* following gives the destination points for the robot to form
 the rough circle */

 goal_config[0] = (cos(alpha) * xrgoal) - (sin(alpha) * yrgoal) +
 robot_config[0] ;

 goal_config[1] = (sin(alpha) * xrgoal) + (cos(alpha) * yrgoal) +
 robot_config[1];

 printf("goal_config[0] = %d\n",goal_config[0]);
 printf("goal_config[1] = %d\n",goal_config[1]);

 /* Main loop. */
 while (!BumperHit)
{
        GetSensorData();
        Movement1();
        if ( abs(tvel) < 11 )
              SumStop++;

        printf("SwitchToCircle->GoCircle SumStop = %d\n",SumStop);

        if(SumStop > 25)
        {
              st();
```

```
                  sleep(33);
                  GoCircle();

            }/* end of if statement */

      } /* end of while loop */

      /* Disconnect. */
      vm(0,0,0) ;
      disconnect_robot(Robot_ID);

} /* end of the function */


/* Movement1(). This function is responsible for using the sensor data
to direct the robot's motion appropriately. */
void Movement1 (void)
{
      int i;
      int panic;
      double phi;
      double D_att[2];

      /* Make sure we are not about to plow into something; check the
      front sonar and infrared sensors. If it looks bad, set panic flag.
      The threshold value for IRRangle has no exact physical relevance,
      and was empirically determined. */
      panic = FALSE;
      for (i = 12; i <= 15; i++)
            if (SonarRange[i] < 8 || IRRange[i] < 10)
                  panic = TRUE;
      for (i = 0; i <= 4; i++)
            if (SonarRange[i] < 8 || IRRange[i] < 10)
                  panic = TRUE;

      /*attractive force Direction in the world coordinates*/
      D_att[0] = (double)(goal_config[0]-robot_config[0]);
      D_att[1] = (double)(goal_config[1]-robot_config[1]);

      /* convert the attractive force Direction into robot coordinates
      */
      phi = ((double)robot_config[2])*PI/(10.0*180.0);
      xgoal =  cos(phi)*D_att[0] + sin(phi)*D_att[1];
      ygoal = -sin(phi)*D_att[0] + cos(phi)*D_att[1];
      potential();
      vm(tvel,svel,svel);
}


/* This function forms a homogeneous circle after a rough circle is
formed. */
void GoCircle(void)
{
      /* Variables are defined here */
      int i;
      double x1, x2, x3, y1, y2, y3;
```

```
double xM, yM ;
double teta;
double dist;

while (!BumperHit)
{
      GetSensorData();

      x1 = ((double)(mindist)+8.81)*cos((double)(minreturn)*0.39);

      y1 = ((double)(mindist)+8.81)*sin((double)(minreturn)*0.39);

      x2 = ((double)(maxdist)+8.81)*cos((double)(maxreturn)*0.39);

      y2 = ((double)(maxdist)+8.81)*sin((double)(maxreturn)*0.39);

      x3 = ((double)(secondmindist) + 8.81) *
      cos((double)(secondminreturn) * 0.39);

      y3 = ((double)(secondmindist) + 8.81) *
      sin((double)(secondminreturn) * 0.39);

      /* middle point coordinates of  the centroid of furthest,
      closest and second closest robots */
      xM = (x1+x2+x3)/3.0;
      yM = (y1+y2+y3)/3.0;

      dist = hypot(xM,yM);  /* distance to the middle point */

      teta = atan2(yM, xM); /* angle between middle point and the
      robot */

      printf("dist = %f\n",dist);
      printf("R = %f\n",(R/10.0));

      /* If the furthest robot can be seen, then move r distance
      of middle point */
      xgoal = (dist - ((R+100.0)/10.0)) * cos(teta) ;
      ygoal = (dist - ((R+100.0)/10.0)) * sin(teta) ;

      /* If  the desired radius is larger than 81 inches, the
      robot doesn't move */
      if ( (int)(R/10.0) > 81)
      {
            xgoal = 0.0;
            ygoal = 0.0;
      }

      /* If  the distances to the closest and second closest
      robots are not equal, move to the closest robot */
      if (abs(secondmindist - mindist) > 8)
      {
            xgoal = x3;
            ygoal = y3;
      }
```

```
            potential();

            vm(tvel,svel,svel);

      } /* end of while loop */

} /* end of function */


/* This function returns sorted sonar distances and the sonar numbers
that gives this distances */
void min2max (void)
{
      int i, j;
      int tmp , tmp1;

      for(i=0; i<16; i++)
      {
            SortedSonarRange[i] = SonarRange[i];
            SortedSonarReturn[i] = i;
      }

      for( i = 0; i<16; i++)
      {
            for(j = i ; j <16; j++)
            {
                  if (SortedSonarRange[i] > SortedSonarRange[j])
                  {
                        tmp = SortedSonarRange[i];
                        tmp1 = SortedSonarReturn[i];
                        SortedSonarRange[i] = SortedSonarRange[j];
                        SortedSonarReturn[i] =  SortedSonarReturn[j];
                        SortedSonarRange[j] = tmp;
                        SortedSonarReturn[j] = tmp1;

                  }/* end of if */

            }

      }

} /* end of function */
```

APPENDIX D. THE SIMULATION PROGRAM CODE FOR LIMITED RANGE ALGORITHM

```
/*********************************************************
This is a C program used for the simulations to simulate limited range
algorithm. This algorithms forms a circle at the center of the field,
even though at initial distribution robots cannot see each other. Usage
is the same as merge-then-circle algorithm. Last modified May 1996.
*********************************************************/

/*** Include Files ***/
#include "Nclient.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*** Constants ***/
#define TRUE 1
#define FALSE 0
#define PI 3.1415926


/*** Function Prototypes ***/

void GetSensorData(void); /* This function returns sensor data */

void Movement(void); /*This function control robot motions*/

int  sign(int) ; /* This function returns 1 for positive values, -1 for
negative values */

void SwichToCircle(void) ; /* This function forms a rough circle after
robots merged */

void Movement1(void);

void GoCircle(); /* This function forms a homogeneous circle after
robots formed a rough circle */

void min2max(void);/*This function sorts the sonar returns*/

void GoCenter(void); /* This function merges robots at the center of the
rectangular shaped field */

void potential(void); /* Potential field method */

void Converge(void);

/*** Global Variables ***/
long SonarRange[16]; /* array of sonar readings (inches) */
long IRRange[16]; /* array of infrared readings (no units)*/
double fused_range[16];  /* fused range data */
int  BumperHit = 0; /* boolean value */
```

```
long robot_config[4]; /* the current robot configuration; x, y,
steering_angle, turret_angle x and y are in tenth of inches, and angles
are in tenth of degrees */

long goal_config[4];/* the goal configuration of the robot*/

long SortedSonarRange[16]; /* sorted values of SonarRange[16] */

int SortedSonarReturn[16];

double xcorner1, ycorner1, xcorner2, ycorner2;
double xcorner3, ycorner3;
int minreturn, maxreturn, secondminreturn;
long mindist, maxdist, secondmindist;
double xgoal, ygoal;

int tvel, svel; /*desired translation and steering velocity in 1/10
inch/sec and deg/sec */

double side1, side2;
int Robot_ID;
double r, R;
int SleepTime;
int count = 0;
int corner = 0;
int CloseCor = 0;


/*** Main Program ***/
main (unsigned int argc, char* argv[])
{
      int i, index;
      int order[16];
      int oldx, oldy;

      Robot_ID = atoi(argv[1]);

      /* r is desired radius of the circle */
      r = (double)(atoi(argv[2]));
      R = r;

      printf("argv[1]= %s \n",argv[1]);
      printf("Robot_ID= %d \n",Robot_ID);

      if (argc!=3)
      {
            printf("please enter 2 parameters besides the command\n");
            exit();
      }

      if ((Robot_ID<1) || (Robot_ID>6))
      {
            printf("Robot ID must be between 1 and 6 ");
            exit();
     }
```

```
        /* Communication port between client and server */
        SERV_TCP_PORT=7772 ;

         /* Connect to Nserver */
        connect_robot(Robot_ID);

        /* Initialize Smask and send to robot. Smask is a large array that
        controls which data the robot returns back to the server. This
        function tells the robot to give us everything. */
        init_mask();

        /* Configure timeout (given in seconds). This is how long the
        robot will keep moving if you become disconnected. Set this low if
        there are walls nearby. */
        conf_tm(1);

        /* Sonar setup: configure the order in which individual sonar
        units fire. In this case, fire all units in counter-clockwise
        order (units are numbered counter-clockwise starting with the
        front sonar as zero). The conf_sn() function takes an integer and
        an array of at most 16 integers. If less than 16 units are to be
        used, the list must be terminated by a element of value -1. See
        the IR setup below for an example of this. The single integer
        value passed controls the time delay between units in multiples of
        four milliseconds. */
        for (i = 0; i < 16; i++)
              order[i] = i;
        conf_sn(1,order);

        /* Infrared setup: only use the front 8 sensors as a last resort.
        The IR sensors are not useful for gauging distances accurately,
        and are thus only used to determine the presence of obstacles that
        are missed by the sonar system. */
        for (i = 0; i < 16; i++)
              order[i] = i;
        conf_ir(1,order);


        /* Unfortunately, the robot can talk... */
        tk("I can't see anyone, okay let's go center.");

        /* Get the sensor information. */
        GetSensorData();

        /* Main loop. */
        while (!BumperHit)
        {
              Movement();
        }

        /* Disconnect. */
        vm (0,0,0);
        disconnect_robot(Robot_ID);

}
```

```
/* Movement(). This function is responsible for using the sensor data to
direct the robot's motion appropriately. */
void Movement (void)
{
      double xrobcorner1, yrobcorner1, xrobcorner2, yrobcorner2,
      xrobcorner3;
      double yrobcorner3 ;
      double alpha1, alpha2, alpha3;
      int i;
      int panic;
      int precorner;
      int preSonarRange0;

      preSonarRange0 = SonarRange[0]; /* previous value of  the front
      sonar sensor */

      GetSensorData();

      /* This procedure checks if the robot get close to any object */
      if ( (minreturn == 12) && ( (preSonarRange0-SonarRange[0])>4) &&
      (SonarRange[0] < 120) )
            CloseCor++;

      if (SonarRange[0] == 255)
            CloseCor = 0;

      /* panic mode */
      panic = FALSE;
      for (i = 0; i <= 2; i++)
            if (fused_range[i] < 20) panic = TRUE;
      for (i = 14; i <= 15; i++)
            if (fused_range[i] < 20) panic = TRUE;

      svel = 0;
      tvel = 100;
      precorner = corner;

      /* If there is a wall near the robot, then robot follows this wall
      */

      if (((SonarRange[(minreturn+15)%16]-
      SonarRange[(minreturn+17)%16])<11)&&(mindist < 150))
      {
            if ((minreturn == 12) && (CloseCor <= 3))
            {
                  count++ ;

                  /* when following the wall state the distance to wall
                  will be between 33 and 28 inches */
                  if (SonarRange[12] > 33)
                  {
                        svel = -25 ;
                        tvel = 100 ;
                  }
```

```
            else if (SonarRange[12] < 28)
            {
                  svel = 25;
                  tvel = 100;
            }

            else
            {
                  svel = 0;
                  tvel = 100;
            }
      }

      else if ( (minreturn == 12) && (CloseCor > 3) )
      {
            svel = 50;
            tvel = 100;

            if (count > 7)
            {
                  corner++ ;
                  count = 0;
            }
      }

      /* If minreturn is not 12, then turn counterclockwise till
      minreturn becomes 12. */
      else
      {
            for (i=13; i<=20; i++)
            {
                  if ( (i % 16) == minreturn )
                  {
                        svel = (i * 49) - 537;
                        /*i=13, svel=100; i=20, svel=443 */

                        tvel = (-4*i) + 152 ;
                        /*i = 13, tvel=100; i=20,tvel=72 */
                  }
            }

            for (i=5; i<12; i++)
            {
                  if (i == minreturn)
                  {
                        svel = (i*50) - 650;
                        /*i=5,svel=-400; i=11,svel= -100 */

                        tvel = 100 ;
                  }
            }
      }
}

/* the simple collision avoidance strategy. */
if (panic) { tvel = 0; svel = 200;}
```

```
printf("svel = %d \n",svel) ;
printf("tvel = %d \n",tvel) ;
printf("count = %d \n",count);
printf("corner = %d \n",corner);
printf("CloseCor = %d \n",CloseCor);

/* Set the robot's velocities. The first parameter is the robot's
translational velocity, in tenths of an inch per second. This
velocity can be between -240 and 240. The second parameter is the
steering velocity, and the third is the turret velocity. The units
of the latter two are tenths of a degree per second, and can be
between -450 and 450. The same value is given for these two so
that the turret is always facing the direction of motion. */
vm(tvel,svel,svel);

if ((precorner == 0) && (corner == 1))
{
      xrobcorner1 = (double)(SonarRange[0]) * 10.0;
      yrobcorner1 = -(double)(mindist) * 10.0;
      alpha1 =  (double)(robot_config[2]) * PI/(10.0*180.0) ;

      xcorner1 = (cos(alpha1) * xrobcorner1) - (sin(alpha1) *
      yrobcorner1) + (double)(robot_config[0]);

      ycorner1 = (sin(alpha1) * xrobcorner1) + (cos(alpha1) *
      yrobcorner1) + (double)(robot_config[1]);

}


if ((precorner == 1) && (corner == 2)) {
      xrobcorner2 = (double)(SonarRange[0]) * 10.0;
      yrobcorner2 = -(double)(mindist) * 10.0 ;
      alpha2 =  (double)(robot_config[2]) * PI/(10.0*180.0) ;

      xcorner2 = (cos(alpha2) * xrobcorner2) - (sin(alpha2) *
      yrobcorner2) + (double)(robot_config[0]);

      ycorner2 = (sin(alpha2) * xrobcorner2) + (cos(alpha2) *
      yrobcorner2) + (double)(robot_config[1]);

}

if ((precorner == 2) && (corner == 3))
{
      xrobcorner3 = (double)(SonarRange[0]) * 10.0 ;
      yrobcorner3 = -(double)(mindist) * 10.0 ;
      alpha3 =  (double)(robot_config[2]) * PI/(10.0*180.0) ;

      xcorner3 = (cos(alpha3) * xrobcorner3) - (sin(alpha3) *
      yrobcorner3) + (double)(robot_config[0]);

      ycorner3 = (sin(alpha3) * xrobcorner3) + (cos(alpha3) *
      yrobcorner3) + (double)(robot_config[1]);
```

```
            /* calculate the distance between corners */
            side1 = hypot((xcorner1-xcorner2) , (ycorner1-ycorner2));

            side2 = hypot((xcorner2-xcorner3) , (ycorner2-ycorner3));

            SleepTime = (int)((side1+side2)/100.0);

            GoCenter();
      }

}/* end of function */


/* Read in sensor data and load into arrays. */
void GetSensorData (void)
{
      int i;
      double corrected_ir[16];  /* correlate IR reading to distance. */
      double corrected_sonar[16];
      double norm[16];

      /* Read all sensors and load data into State array. */
      gs();

      /* Read State array data and put readings into individual arrays.
      */
      for (i = 0; i < 16; i++)
      {
            /* Sonar ranges are given in inches, and can be between 6
            and 255, inclusive. */
            SonarRange[i] = State[17+i];

            /* IR readings are between 0 and 15, inclusive. This value
            is inversely proportional to the light reflected by the
            detected object, and is thus proportional to the distance of
            the object. Due to the many environmental variables
            effecting the reflectance of infrared light, distances
            cannot be accurately ascribed to the IR readings. */
            IRRange[i] = State[1+i];
      }

      /* to correlate the IR reading to physical distance. The numbers
      are obtained by least square linear regression of measurement
      data. */
      for (i = 0; i < 16; i++)
            corrected_ir[i] = 2.2508 * ((double) IRRange[i] + 0.8602);

      for (i = 0; i < 16; i++)
            corrected_sonar[i] = (double) SonarRange[i];
            /* to fuse the sonar and IR data */

      for (i = 0; i < 16; i++)
      {
            if (IRRange[i] <= 14)
            {
```

```
            norm[i] = corrected_sonar[i] * corrected_sonar[i] +
            corrected_ir[i] * corrected_ir[i];

            fused_range[i] = (corrected_sonar[i] *
            corrected_sonar[i] * corrected_ir[i]
            + corrected_ir[i] * corrected_ir[i] *
            corrected_sonar[i]) / norm[i];

            if (fused_range[i] <= 5.0)
                  fused_range[i] = 0.0;
      }

      else
      {
            fused_range[i] = corrected_sonar[i];
      }
}

/* The robot configuration parameters (x,y,steering_angle,and
turret_angle) are stored in State[34], State[35], State[36], and
State[37].  */
for (i = 0; i < 4; i++)
      robot_config[i] = State[34+i];

/* Check for bumper hit. If a bumper is activated, the
corresponding State[33] will be turned on. Since we don't care
which bumper is hit, we thus only need to check if State[33] is
greater zero. */
if (State[33] > 0)
{
      BumperHit = 1;
      tk("Ouch.");
      printf("Bumper hit!\n");
}

minreturn = 0;
for (i = 1 ; i < 16 ; i++)
{
      if (SonarRange[i] < SonarRange[minreturn])
            minreturn = i;
}
mindist = SonarRange[minreturn];

maxreturn = minreturn ;
for (i = 0 ; i < 16 ; i++)
{
      if ((SonarRange[i] >= SonarRange[maxreturn]) &&
      (SonarRange[i]<255))
            maxreturn = i;
}
maxdist = SonarRange[maxreturn];

min2max();
secondminreturn = SortedSonarReturn[1];
secondmindist = SortedSonarRange[1] ;
```

```
      if ((abs(SortedSonarReturn[0]-SortedSonarReturn[1])==1) || (
            abs(SortedSonarReturn[0]-SortedSonarReturn[1])==15) )
      {
            secondminreturn = SortedSonarReturn[2];
            secondmindist = SortedSonarRange[2] ;

            if ( (abs(SortedSonarReturn[0]-SortedSonarReturn[2])==1) ||
            (abs(SortedSonarReturn[0]-SortedSonarReturn[2])==15) )
            {
                  secondminreturn = SortedSonarReturn[3];
                  secondmindist = SortedSonarRange[3] ;
            }
      }

      if (mindist == 255)
            printf("There is no object around");

}


/* Sign function.  It returns 1 if x is positive, and returns -1
otherwise */
int sign(int x)
{
      return x>0?1:-1;
}


void min2max (void)
{
      int i, j;
      int tmp , tmp1;

      for (i=0; i<16; i++)
      {
            SortedSonarRange[i] = SonarRange[i];
            SortedSonarReturn[i] = i;
      }

      for ( i = 0; i<16; i++)
      {
            for (j = i ; j <16; j++)
            {
                  if (SortedSonarRange[i] > SortedSonarRange[j])
                  {
                        tmp = SortedSonarRange[i];
                        tmp1 = SortedSonarReturn[i];

                        SortedSonarRange[i] = SortedSonarRange[j];

                        SortedSonarReturn[i] =  SortedSonarReturn[j];

                        SortedSonarRange[j] = tmp;
                        SortedSonarReturn[j] = tmp1;
                  }
            }
```

```
	}

} /* end of the function */


void GoCenter(void)
{
	int i;
	int panic;
	int SumStop = 0;
	double phi;
	double D_att[2];

	/* Make sure we are not about to plow into something; check the
	front sonar and infrared sensors. If it looks bad, set panic flag.
	The threshold value for IRRangle has no exact physical relevance,
	and was empirically determined. */
	panic = FALSE;
	for (i = 0; i <= 2; i++)
		if (fused_range[i] < 20) panic = TRUE;
	for (i = 14; i <= 15; i++)
		if (fused_range[i] < 20) panic = TRUE;

	goal_config[0] = (xcorner1+xcorner3)/2;
	goal_config[1] = (ycorner1+ycorner3)/2;

	while(!BumperHit)
	{
	GetSensorData();

		/* attractive force direction in the world coordinates */
		D_att[0] = (double)(goal_config[0]-robot_config[0]); /* x
		component */

		D_att[1] = (double)(goal_config[1]-robot_config[1]); /* y
		component */

		/* convert the atrtractive force direction into robot
		coordinates. */
		phi = ((double)robot_config[2])*PI/(10.0*180.0);
		xgoal =  cos(phi)*D_att[0] + sin(phi)*D_att[1];
		ygoal = -sin(phi)*D_att[0] + cos(phi)*D_att[1];

		potential();

		if (panic)
		{
			tvel = 0;
			svel = 0;
		}

		vm(tvel,svel,svel);

		if ( (abs(tvel)<11) && (mindist != 255) )
			SumStop++;
```

```
            printf("GoCenter->Converge SumStop = %d\n", SumStop);

            if (SumStop > 10)
            {
            st();

                  printf("GoCenter->Converge SleepTime =
                  %d\n",(SleepTime+100));

                  sleep(SleepTime+100) ;

                  Converge();
            }

      } /* end of while loop */

}



void SwichToCircle(void)
{

      int i;
      int s, sm, opposite;
      double xrgoal, yrgoal, alpha;
      int SumStop = 0;
      int CutOff = 0;

      s = 0;
      sm = 0;
      for (i = 0; i<16; i++) {
            if (SonarRange[i]==255)
            {
                  s = s + i;
                  sm++;
            }
      }

      if ( sm<9 )  r = r + ( (double)(mindist) * 10.0) ;

      printf("s = %d\n", s);
      printf("    sm = %d\n",sm);

      if (mindist != 255)
      {
            if (SonarRange[15]==255)
            {
                  i = 0;
                  while(SonarRange[i]==255 )
                  {
                        s = s + 16;
                        i++;
                  }
            }
      }
```

```
printf ("s = %d\n", s);
printf("sm = %d\n",sm);

if (sm == 0)
      opposite = 0;
else
      opposite = s / sm;

if (opposite == 0) opposite = 16;

while ( ((SonarRange[(opposite+1)%16] != 255) ||
(SonarRange[(opposite-1)%16] != 255) || (SonarRange[opposite % 16]
!= 255)) && (CutOff < 33) )
{
      CutOff++;
      printf("CutOff = %d\n",CutOff);
      opposite++;
      GetSensorData();
}

while (SonarRange[opposite % 16] != 255)
{
      opposite++;
      GetSensorData();
      printf("Second Chance");
}

opposite = opposite % 16;

printf("opposite = %d\n",opposite) ;
printf("r = %f\n",r);

xrgoal = r * cos((double)(opposite) * 0.392699);
yrgoal = r * sin((double)(opposite) * 0.392699);
alpha  = (double)(robot_config[2]) * PI/(10.0*180.0);

goal_config[0] = (cos(alpha) * xrgoal) - (sin(alpha) * yrgoal) +
robot_config[0];

goal_config[1] = (sin(alpha) * xrgoal) + (cos(alpha) * yrgoal) +
robot_config[1];

printf("goal_config[0] = %d\n",goal_config[0]);
printf("goal_config[1] = %d\n",goal_config[1]);


/* Main loop. */
while (!BumperHit)
{
      GetSensorData();
      Movement1();
      if ( abs(tvel) < 11 )
            SumStop++;

      printf("SwichToCircle->GoCircle SumStop = %d\n",SumStop);
```

```
            if(SumStop > 5)
            {
                  st();
                  printf("SwichToCircle->GoCircle SleepTime =
                  %d\n",(SleepTime + 100));
                  sleep(SleepTime + 100);
                  GoCircle();
            }
      }

       /* Disconnect. */
       vm(0,0,0) ;
       disconnect_robot(Robot_ID);

}


/* Movement1(). This function is responsible for using the sensor data
to direct the robot's motion appropriately. */
void Movement1 (void)
{
       int i;
       double phi;
       double D_att[2];

       /* attractive force Direction in the world coordinates */
       D_att[0] = (double)(goal_config[0]-robot_config[0]);
       D_att[1] = (double)(goal_config[1]-robot_config[1]);

       /* convert the attractive force Direction into robot coordinates
       */
       phi = ((double)robot_config[2])*PI/(10.0*180.0);
       xgoal =  cos(phi)*D_att[0] + sin(phi)*D_att[1];
       ygoal = -sin(phi)*D_att[0] + cos(phi)*D_att[1];

       potential();

       vm(tvel,svel,svel);
}


 void GoCircle(void)
 {
       int i;
       double x1, x2, x3, y1, y2, y3;
       double xM, yM ;
       double teta;
       double dist;

       while (!BumperHit) {
             GetSensorData();

             x1 = ((double)(mindist)+8.81)*cos((double)(minreturn)*0.39);

             y1 = ((double)(mindist)+8.81)*sin((double)(minreturn)*0.39);
```

```
            x2 = ((double)(maxdist)+8.81)*cos((double)(maxreturn)*0.39);

            y2 = ((double)(maxdist)+8.81)*sin((double)(maxreturn)*0.39);

            x3 = ( (double)(secondmindist) + 8.81) *
            cos((double)(secondminreturn) * 0.39);

            y3 = ( (double)(secondmindist) + 8.81) *
            sin((double)(secondminreturn) * 0.39);

            xM = (x1+x2+x3)/3.0;
            yM = (y1+y2+y3)/3.0;

            dist = hypot(xM,yM);

            teta = atan2(yM, xM);

            printf("dist = %f\n",dist);
            printf("R = %f\n",(R/10.0));

            xgoal = (dist - ((R+100.0)/10.0)) * cos(teta) ;
            ygoal = (dist - ((R+100.0)/10.0)) * sin(teta) ;

            if ( (int)(R/10.0) > 81){
                  xgoal = 0.0;
                  ygoal = 0.0;
            }

            if (abs(secondmindist - mindist) > 8) {
                  xgoal = x3;
                  ygoal = y3;
            }

            potential();
            vm(tvel,svel,svel);

      }
}


/* This function merges all robots to a cluster and works as the same as
merge algorithm */
void Converge(void)
{
      /* Variables are defined here. */
      int i;
      int panic;
      int SumStop = 0;
      double x1, x2, y1, y2 ;


      while (!BumperHit) {
            GetSensorData();

            /* Compute the goal points in robot coordinate system */
```

```
            x1 = ( (double)(mindist) + (8.81)) * cos (
            (double)(minreturn) * 0.39);

            y1 = ( (double)(mindist) + (8.81)) * sin (
            (double)(minreturn) * 0.39);

            x2 = ( (double)(maxdist) + (8.81)) * cos (
            (double)(maxreturn) * 0.39);

            y2 = ( (double)(maxdist) + (8.81)) * sin (
            (double)(maxreturn) * 0.39);

            xgoal = (x2 + x1)/2;
            ygoal = (y2 + y1)/2;

            if ( (abs(minreturn-maxreturn)==1)||(abs(minreturn-
            maxreturn)==15)||(minreturn==maxreturn) )
            {
                  if ( abs(mindist-maxdist)<=4 )
                  {
                        xgoal = ((double)(mindist)+8.81) * cos(
                        ((double)(minreturn)*0.39)-(60.0 *(PI/180.0)));

                        ygoal = ((double)(mindist)+8.81) * sin(
                        ((double)(minreturn)*0.39)-(60.0 *(PI/180.0)));
                  }
            }

            potential();

            if (mindist==255)
            {
                  svel = 50;
                  tvel = 100;
            }

            vm(tvel,svel,svel);

            if ( (abs(tvel)<11) && (mindist != 255) )
                  SumStop++;
            printf("Converge->SwitchToCircle SumStop = %d\n",SumStop);

            if (SumStop > 10)
            {
                  st() ;
                  printf("Converge->SwitchToCircle SleepTime =
                  %d\n",(SleepTime+100));
                  GetSensorData();
                  sleep(SleepTime + 100) ;
                  SwichToCircle();
            }
      }
}
```

```
/* The potential field method is used for motion control and collision
avoidance */
void potential() {

      /* Various constants for computing attractive and repulsive forces
      should be defined here, e.g., */

      double rho_0 = 50.0; /* cut-off distance of the repulsive force */

      double scale = 10.0 ; /* scaling factor for attractive force */

      double eta = 12000.0;/*repulsive force scaling factor*/

      double d  = 100.0 ; /*saturation in attractive force */

      double gain_tvel = 0.1; /* translational velocity gain, can be
      adjusted */

      double gain_svel = 200.0;/*rotational velocity gain */

      int i;

      /* attractive and repulsive forces are defined */
      double F_att[2], F_rep[2], F_tol[2] ;
      double rho_float;
      double distance ;

      printf ("xgoal = %f\n", xgoal);
      printf ("ygoal = %f\n", ygoal);

      /* the distance between present location and destination is
      calculated */
      distance = hypot(xgoal,ygoal);
      printf("distance : %f\n ", distance);

      /* parabolic-well definition of the attractive force */
      if (distance <= d)
      {
            F_att[0] = scale*xgoal ;
            F_att[1] = scale*ygoal ;
      }

      /* conic-well definition of attractive force */
      else
      {
            F_att[0] = scale*d*(xgoal/distance) ;
            F_att[1] = scale*d*(ygoal/distance) ;
      }

      /*compute the repulsive force in the robot coordinate*/
      F_rep[0] = 0.0;
      F_rep[1] = 0.0;
      for (i = 0; i <= 15; i++)
      {
            rho_float = (double) (SonarRange[i]);
            if (rho_float < rho_0)
```

```
            {
                  F_rep[0] += -eta * (1.0/rho_float - 1.0/rho_0) * cos
                  ((double)(i) * 0.392699)/(rho_float);

                  F_rep[1] += -eta * (1.0/rho_float - 1.0/rho_0) * sin
                  ((double)(i) * 0.392699)/(rho_float);

            } /* end of if  statement */

      } /* end of for loop */

      /* compute the total force in the robot coordinates */
      F_tol[0] = F_att[0] + F_rep[0];
      F_tol[1] = F_att[1] + F_rep[1];


      /* set the translational velocity */
      tvel = (int) (gain_tvel * F_tol[0]);

      /* set the rotational velocity */
      if (F_tol[0] == 0.0) svel = 0;

      else
      {
            svel = (int) (gain_svel * sin(atan2(F_tol[1],F_tol[0])));

            svel = svel * sign( (int) (F_tol[0]) );

      }

      /*limit the translational  and rotational velocities */
      if (abs(tvel) > 230)
            tvel = 230 * sign(tvel);
      if (abs(svel) > 450)
            svel = 450 * sign(svel) ;

} /* end of potential() function */
```

LIST OF REFERENCES

1. K. Sugihara and I. Suzuki, "Distributed Motion Coordination of Multiple Mobile Robots," *Proocedings IEEE International Symposium on Intellegence and Contro*l, Philadelphia, PA, 1990, pp. 138-143.

2. I. Suzuki and M. Yamashita, "Formation and Agreement Problems for Anonymous Mobile Robots," *Proceedings of the 31st Annual Allerton Conference on Communication, Control, and Computer*, University of Illinois, Urbana, IL, 1993, pp. 93-102.

3. H. Ando, I. Suzuki, and M. Yamashita, "Formation and Agreement Problems for Synchronous Mobile Robots with Limited Visibility," *Proceedings of International Symposium on Intelligent Control*, Monterey, CA, August 1995, pp. 453-460.

4. K. Sugihara and I. Suzuki, "Distributed Algorithms for Formation of Geometric Patterns with Many Mobile Robots," *Journal of Robotic Systems*, vol.13, no.3, 1996, pp. 127-139.

5. I. Suzuki and M. Yamashita, "A Theory of Distributed Anonymous Mobile Robots -Formation and Agreement Problems," Technical Report TR-94-07-01, Department of Electrical Engineering and Computer Science, University of Wisconsin, Milwaukee, 1994.

6. J. Latombe, *Robot Motion Planning*, Kluver Academic Publisher, Norwell, MA, 1991.

7. Y. Koren and J. Borenstein, "Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation," *Proceedings of IEEE International Conference on Robotics and Automation*, April 1991, pp. 1398-1404.

8. T. Arai, H. Asama, T. Fukuda, and I. Endo, *Distributed Autonomous Robotic Systems*, Springer Verlag, New York City, NY, 1994.

9. T. Ueyama, T. Fukuda, G. Iritani, and F. Arai, "Optimization of Group Behavior on Cellular Robotics System in Dynamic Environment," *Proceedings of IEEE International Conference on Robotics and Automation*, San Diego, CA, May 1994, pp. 1027-1032.

10. M. Inaba, Y. Kawauchi, and T. Fukuda, "A Princible of Decision Making of Cellular Robotics System (CEBOT)," *Proceedings of IEEE International Conference on Robotics and Automation*, Los Alamitos, CA, May 1993, pp. 833-838.

11. J. Wang and G. Beni, "Cellular Robotics Systems: Self Organizing Robots and Kinetic Pattern Generation," *Proceedings of IEEE International Workshop on Intelligent Robotics Systems*, Tokyo, Japan, 1988, pp. 139-144.

12. T. Fukuda and S. Nakagawa, "Approach to the Dynamically Reconfigurable Robot Systems," *Journal of Intelligent Robtics Systems,* vol.1, 1988, pp. 55- 72.

13. M. J. Mataric, "Minimizing Complexity in Controlling a Mobile Robot Population," *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 830-853, Nice, France, May 1992.

14. P. Tournassoud, "A Strategy For Obstacle Avoidance And Its Applicationto Multi-Robot Systems," *Proceedings of IEEE International Conference on Robotics and Automation*, San Francisco, CA, 1986, pp. 1224-1229.

15. M. J. Mataric, "Designing Emergent Behaviors: from Local Interactions to Collective Intelligence," *From Animals to Animals 2: International Conference on Simulation of Adaptive Behavior*, MIT Press, Cambridge, MA, 1993, pp. 432-441.

16. Q. Chen and J. Y. S. Luh, "Coordination And Control of a Group of Small Mobile Robots," *Proceedings of IEEE International Conference on Robotics and Automation*, San Diego, CA, 1994, pp. 2315-2320.

17. J. Borenstein and Y. Koren, "Real-Time Obstacle Avoidance for Fast Mobile Robots," *IEEE Transactions on Systems, Man, and Cybernatics,* vol.19, no.5, Sep/Oct 1989, pp. 1179-1187.

18. L. Parker, "Adaptive Action Selection for Cooperative Agent Teams," *From Animals to Animals 2: International Conference on Simulations of Adaptive Behavior*, MIT Press, Cambridge, MA, 1993, pp. 442-450.

19. R. C. Arkin, "Motor Schema Based Mobile Robot Navigation," *International Journal of Robotics Research*, vol.8, no.4, 1989, pp. 92-112.

20. R. L. Finney and G. B. Thomas, *Calculus,* Addison-Wesley-Publishing Company, New York, 2nd edition, 1994, pp. 862.

21. Y. Kanayama and T. Noguchi, "Spatial Learning By Autonomous Mobile Robot With Ultrasonic Sensors," Technical Report TRCS89-06, University of California Santa Barbara Department of Computer Science, Santa Barbara, CA, February, 1989.

22. *Nomad 200 Mobile Robot User's Guide*, Nomadic Technologies, Inc., Mountain View, CA.

23. *Nomad 200 Mobile Robot Simulator Language Manual*, Nomad Host Software Development Environment, Release 2.1, Nomadic Technologies, Inc., Mountain View CA.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center 8725 John J. Kingman Rd., STE 0944 Ft. Belvoir, VA 22060-6218	2
2.	Dudley Knox Library Naval Postgraduate School 411 Dyer Rd. Monterey, California 93943-5101	2
3.	Chairman, Coded EC Department of Electricel and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
4.	Professor Xiaoping Yun, Code EC/Yx Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5101	4
5.	Professor Robert G. Hutchins, Code EC/Hu Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5101	1
6.	Professor Ichiro Suzuki Department of Electrical Engineering and Computer Science University of Wisconsin - Milwaukee Milwaukee, WI 53201	1
7.	Deniz Kuvvetleri Komutanligi Personel Daire Baskanligi Bakanliklar, Ankara 06100 Turkey	2
8.	Gölcük Tersanesi Komutanligi Gölcük, Kocaeli, Turkey	1

9.	Deniz Harp Okulu Komutanligi Kutuphanesi Tuzla, Istanbul, Turkey 81704	1
10.	Ltjg. Okay Albayrak Buyukkumla Koyu No 174/3 Gemlik, Bursa, Turkey	1