NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

GEOMETRIC FORMATION WITH UNIFORM DISTRIBUTION AND MOVEMENT IN FORMATION OF DISTRIBUTED MOBILE ROBOTS

by

Gokhan Alptekin
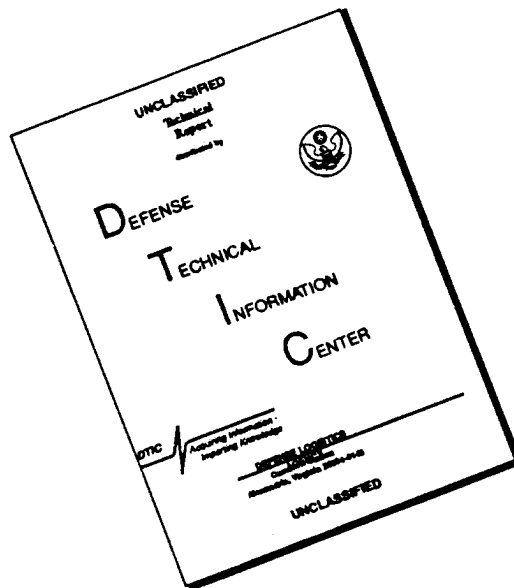
June, 1996

Thesis Advisor: Xiaoping Yun

Approved for public release; distribution is unlimited.

19960910 145

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

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

4. TITLE AND SUBTITLE **GEOMETRIC FORMATION WITH UNIFORM DISTRIBUTION AND MOVEMENT IN FORMATION OF DISTRIBUTED MOBILE ROBOTS**	5. FUNDING NUMBERS
6. AUTHOR(S) Gokhan Alptekin	

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)*

The formation problem of distributed mobile robots was studied in the literature for idealized robots. Idealized robots are able to instantaneously move in any direction, and are equipped with perfect range sensors. In this study, the formation problem of distributed mobile robots that are subject to physical constraints is addressed. Mobile robots considered in this study have physical dimensions and their motions are governed by physical laws. They are equipped with sonar and infrared range sensors. The formation of lines and circles by using the potential field method is investigated in detail. It is demonstrated that line and circle algorithms developed for idealized robots do not work well for physical robots. New line and circle algorithms, with consideration of physical robots and sensors, are presented and validated through extensive simulations. Movement in formation of a small group of physical mobile robots is also studied. An algorithm is developed using the potential field method that makes robots move through a workspace filled with many obstacles while maintaining the formation.

SUBJECT TERMS Distributed Autonomous Mobile Robots, Line and Circle Formation, Nomad 200 Mobile Robot, Physical Robots, Potential Field, Position Maintenance in Formation	15. NUMBER OF PAGES 95
	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

i

DTIC QUALITY INSPECTED 3

ii

GEOMETRIC FORMATION WITH UNIFORM DISTRIBUTION AND MOVEMENT IN FORMATION OF DISTRIBUTED MOBILE ROBOTS

Gokhan Alptekin
Lieutenant JG, Turkish Navy
B.S. Turkish Naval Academy - 1990

Submitted in partial fulfillment
of the requirements for the degree of

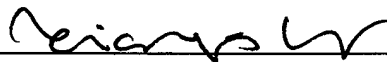**MASTER OF SCIENCE
IN
ELECTRICAL ENGINEERING**

from the

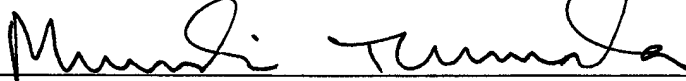**NAVAL POSTGRADUATE SCHOOL
June 1996**
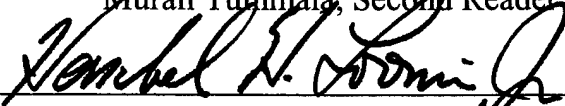
Author: _____

Gokhan Alptekin

Approved by: _____

Xiaoping Yun, Thesis Advisor

_____

Murali Tummala, Second Reader

_____

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

ABSTRACT

The formation problem of distributed mobile robots was studied in the literature for idealized robots. Idealized robots are able to instantaneously move in any direction, and are equipped with perfect range sensors. In this study, the formation problem of distributed mobile robots that are subject to physical constraints is addressed. Mobile robots considered in this study have physical dimensions and their motions are governed by physical laws. They are equipped with sonar and infrared range sensors. The formation of lines and circles by using the potential field method is investigated in detail. It is demonstrated that line and circle algorithms developed for idealized robots do not work well for physical robots. New line and circle algorithms, with consideration of physical robots and sensors, are presented and validated through extensive simulations. Movement in formation of a small group of physical mobile robots is also studied. An algorithm is developed using the potential field method that makes robots move through a workspace filled with many obstacles while maintaining the formation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

A. GENERAL

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 1-5. 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.

The formation problem of distributed mobile robots has been studied for idealized mobile robots [Ref. 1, 2, 4] -- 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, Suzuki et. al 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, 4, 5].

In the previous study [Ref. 1, 2, 4], 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 of time.

1

Line and circle formation, or formation of any geometric patterns in general, is only one of many issues of distributed mobile robots [Ref. 6]. Representative work addressing other issues of distributed mobile robots includes cellular robotics systems [Ref. 7, 8, 9], and dynamically reconfigurable robotics systems [Ref. 10]. These systems can change their overall shape depending on the task and the environment by autonomously detaching and combining cells.

B.   PROBLEM STATEMENT

Based on earlier work, this thesis studies the formation problem of distributed "physical" mobile robots.   The mobile robots considered in this study 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 the Nomad 200 mobile robots [Ref. 21, 22]. 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 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 simulation of this study.

This thesis also studies the development of behaviors for formation maintenance in a small group of physical mobile robots. In addition to maintaining their position in formation, robots must also move to a specified goal location while avoiding collisions with obstacles and other robots. Most work done in this area focused on the dynamics and stability of multi-robot formations. Ref. 11 developed a strategy for robot formations where individual robots are given specific locations relative to a leader or neighbor to maintain their positions. Ref. 12 demonstrated formation generation by distributed control. Both studies centered on the analysis of group dynamics and stability and did not

2

provide obstacle avoidance. Ref. 13 integrated motor schemas, for relative positional maintenance with existing navigational behaviors to help robots complete navigational tasks while in formation. The formation behaviors were implemented as motor schemas, a type of reactive navigational strategy [Ref. 14]. Each robot is assigned a unique identification number (ID), and a robot's designated position in a given formation depends upon its ID.

C.   IMPLEMENTATION OF COLLISION AVOIDANCE

Different schemes for collision avoidance were examined in References 4, 12, 14, 15, 16, 17, 18, 19. The method proposed in Ref. 4 is discussed in the following chapter. The strategy proposed in Ref. 16 is that if a robot detects another robot on its way, it stops and waits for some fixed period of time. If a robot is still present, the robot turns left and proceeds forward. The method proposed in Ref. 12 adds an initial step to the algorithms from References 1 and 5 to avoid collisions. Motor schemas [Ref. 14] is another method for navigation and collision avoidance.

Motion control and collision avoidance in this study are achieved by implementing a potential field algorithm [Ref. 20, 21]. 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.

D.   OUTLINE OF THE THESIS

Chapter II provides background information including the Nomad 200 mobile robot and the simulator as well as a brief description of the potential field algorithm. Chapter III discusses the implementation of the existing line algorithm with physical robots and presents a modified version of the existing algorithm which works well for physical robots. Chapter IV studies the existing circle algorithm and simulation results when it is implemented with physical robots. A modified algorithm is introduced which

3

is directed towards forming a better approximation of a circle. Chapter V discusses an algorithm based on "leader-follower" technique to help a small group of robots move in formation while avoiding collisions. Chapter VI presents the conclusion and recommendations for follow-on studies. The source codes for the modified line algorithm, modified circle algorithm and the moving in formation algorithm are presented in Appendix A, B, and C respectively.

II. BACKGROUND INFORMATION

This chapter provides background information so that the reader has an understanding of the NOMAD 200 mobile robot, as well as the potential field method which is implemented to realize the motion of robots.

A.   NOMAD 200 MOBILE ROBOT

The Nomad 200 is an integrated mobile robot system with four sensory modules including tactile, infrared, ultrasonic, and laser systems [Ref. 22]. The Nomad 200 has 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 includes a software package for the host computer with a graphic interface and a simulator.

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 the simulation of 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 infrared ranging system that provides 360 degrees coverage. Each of the 16 sensors is composed of two LED emitters and a photodiode detector enclosed in a delrin housing. 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 within 35 inches, but not reliable beyond 35 inches.

The Nomad 200 also has a 16 channel sonar ranging system which can give range information from 17 inches to 255 inches with 1 percent 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 16 sensors.

Although the user manuals [Ref. 21, 22] for the Nomad 200 robot state that the maximum sonar range is 255 inches, it is determined by two parameters (half-cone and overlap) which are stored in the *robot.setup* file. The sonar sensors have a non-zero beam-width, i.e., they can detect an object as long as it overlaps with this cone (and is within detectable distance). The half beam-width for sonar is specified by half-cone in the setup file. If this variable is set to 0, the sensor can detect an object only if it intersects with the ray drawn from the sensor along the direction where the sensor is pointing. When this variable is set to some positive values, the sensors can detect objects that fall within the cone specified by this variable assuming there is sufficient overlap between the object and the sonar cone (as specified by overlap in the setup file). Briefly, half-cone 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.

The sonar half-cones have to overlap in order to cover 360 degrees around the robot. If they overlap quite a distance away from the robot, then the triangular area that stays between two consecutive sonars and the intersection point of sonar beams cannot be covered at all. On the other hand, anything in the overlapped zone is seen by both sonars. This means that the position of that object cannot be calculated accurately. Worst of all, the sonars that see the same object which is in their overlapped zone, cannot sense

another object even if it is on the line of sight, unless it is closer than the one in overlapped zone. So, letting the sonar beams overlap very close to the robot creates a huge overlapped zone, which in turn affects the sight of the sonars negatively. The maximum sonar range in the area that is exactly in the middle of two consecutive sonars, is shorter than the one in line of sight area. The values of half-cone and overlap have to be set in such a way to best balance these factors. Table 1 illustrates the results of the simulations with different values of half-cone and overlap, in order to detect an object of a size of the Nomad 200 robot.

Table 1. Simulation Results to Determine the Values of Half-cone and Overlap.

Half-cone [1]	Overlap [2]	A [3]	B [4]	C [5]
125	0.05	51	53	62
125	0.07	208	210	212
125	0.10	128	144	145
125	0.15	74	75	85
105	0.07	145	154	236
105	0.08	131	139	206
105	0.09	119	130	180
105	0.10	111	115	162
110	0.08	159	175	194
110	0.09	143	152	174
115	0.07	185	187	235
115	0.08	166	168	188
115	0.10	129	130	146

[1] Half-cone (in tenths of degrees)
[2] Overlap (expressed as a ratio -- 0.1=10% -- to the angular range of the main lobe);
[3] A: The range that two consecutive sonars first start to sense the same robot in the overlapped zone (in inches)
[4] B: The maximum range in the overlapped zone to see a robot (in inches)
[5] C: The maximum line of sight range to see a robot (in inches)

The first set of values in Table 1 represent the default values which are set mainly for the detection of large objects such as a wall. The best values for half-cone and overlap to detect an object of the size of the Nomad 200 robot was determined to be $10.5°$ and 0.8 percent respectively, and used throughout all simulations.

3. 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 [Ref. 23]. It consists of two parts: *the server* and *the client*. The server performs four functions: *Host ←→ Robot Interface* which allows complete control of the robot from a host computer, *Robot Simulator* which simulates the Nomad 200 robot (including its basic motions: translation, steering and rotation, and its basic sensor systems: tactile, infrared, sonar and laser), *Graphic User Interface* which provides graphic display for various sensory information and convenient interface with the robot and the robot simulator, and *Client ←→ Server Language User Interface* which allows users' C or Lisp program (acts as a client process) to access the server. The client part provides the link (in terms of a set of interface functions) between the application program and the server.

The graphical user interface consists of several windows that give information and allow control of a robot. First, there is the world (map) window which gives an overall view of the environment (real or simulated) that the robots are in as well as the positions of each robot relative to the environment and each other. Also there is the robot window, (one copy for each robot), which contains information about each individual robot, such as current command executed, position, orientation, and sensor data history. Attached to each robot window are two windows that give more detailed information about the current sensor readings, including ray and half-cone data. Each time any of the functions that return sensor data is called, the sensory data returned as well as the current positions of robots are displayed graphically on these windows. The users are allowed to draw maps in the world window to simulate the environment. The figures that show the simulation results in the following chapters 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

8

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 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 UNIX operating system. This makes debugging very easy and provides the possibility to test each behavior independently as well as to add or remove some robots during the run.

B.   POTENTIAL FIELD METHOD

A robot in 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. 20]. 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 the 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 robot moves freely in $W=R^N$, with $N=2$ or 3, i.e. $C=R^N$. (W denotes Robot's workspace, R is the set of real numbers, C denotes the configuration space of a robot. An element of C is denoted by q.) A more detailed discussion can be found in [Ref. 20].

The field of artificial forces $\vec{F}(q)$ in C is produced by a differentiable potential function:

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

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:

$$\vec{\nabla} U = \begin{bmatrix} \partial U / \partial x \\ \partial U / \partial y \end{bmatrix} \quad \text{or} \quad \vec{\nabla} U = \begin{bmatrix} \partial U / \partial x \\ \partial U / \partial y \\ \partial U / \partial z \end{bmatrix}. \tag{2}$$

In order to make the robot attracted toward its goal configuration, while being repulsed from the obstacles, U is constructed as the sum of two elementary potential functions:

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

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:

$$\vec{F}_{att} = -\vec{\nabla} U_{att} \text{ and } \vec{F}_{rep} = -\vec{\nabla} U_{rep} , \tag{4}$$

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.:

$$U_{att}(q) = \frac{1}{2} \xi \rho_{goal}^2 (q) , \tag{5}$$

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

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

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

The parabolic well demonstrates good stabilizing characteristics when used for on-line collision avoidance [Ref. 10]. This is because 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 $\rho_{goal}(q) \to \infty$. Alternatively, U_{att} can be defined as a conic-well, i.e.:

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

Then, the attractive force is:

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

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 $q \to q_{goal}$, 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 1(a) when it is implemented by using the above conic-well definition. The attractive force must be made continuous at the transition point, which can be achieved simply by multiplying the conic-well attractive potential by "d".

11

Then, the resulting attractive force can be defined as follows, which is plotted in Figure 1(b).

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

2. Repulsive Potential

The main idea is to create a potential barrier around the C-obstacle region that cannot be traversed by the robot's configuration. 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:

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

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

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

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.

12

$F_{att}(q) = -\xi(q-q_{goal}) / \|q-q_{goal}\|$

$d\xi$

$F_{att}(q) = -\xi(q-q_{goal})$

ξ

-d

d

$(q-q_{goal})$

$-\xi$

$-d\xi$

$F_{att}(q) = -\xi(q-q_{goal}) / \|q-q_{goal}\|$

(a)

$F_{att}(q) = -d\xi(q-q_{goal}) / \|q-q_{goal}\|$

$d\xi$

$F_{att}(q) = -\xi(q-q_{goal})$

$(q-q_{goal})$

$-d\xi$

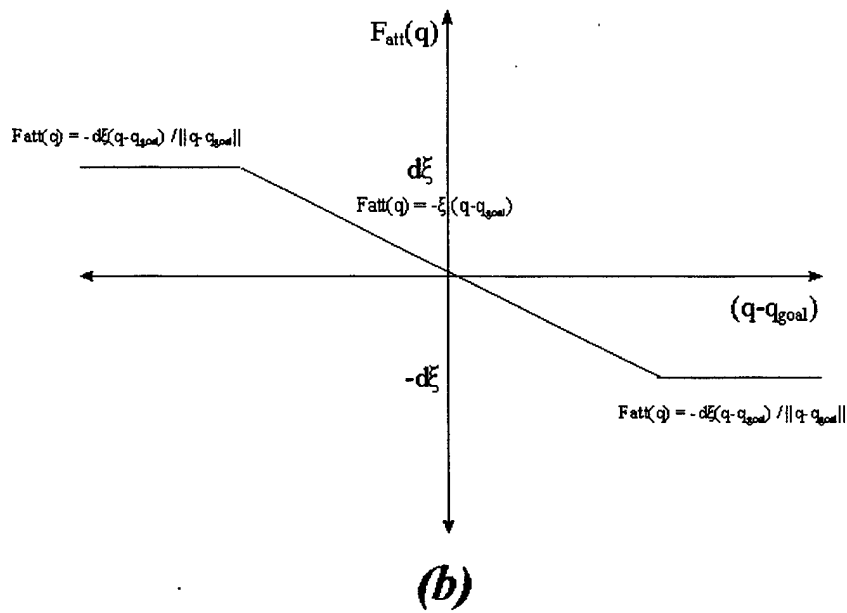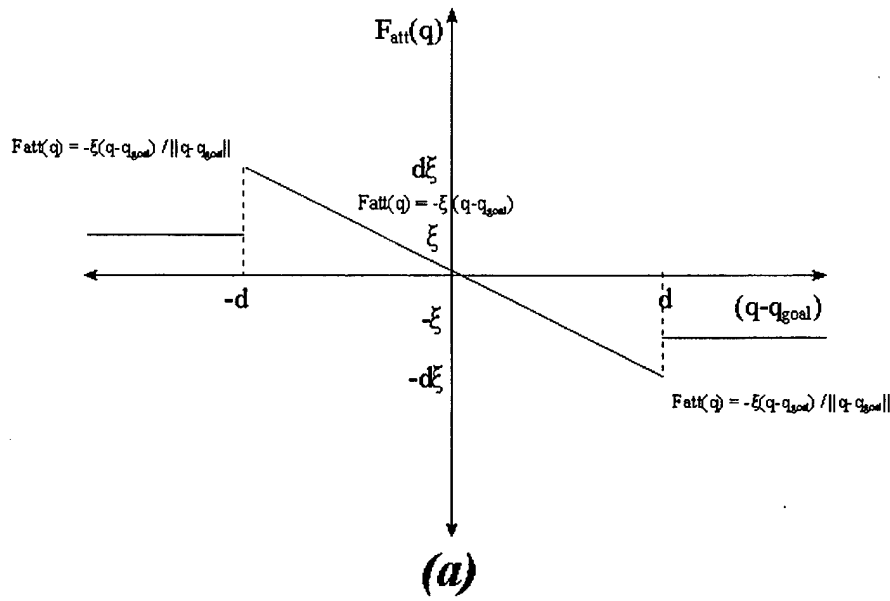$F_{att}(q) = -d\xi(q-q_{goal}) / \|q-q_{goal}\|$

(b)

Figure 1. Plot of an 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 1(a) illustrates the discontinuity problem, and (b) illustrates the provided continuity at the transition point.

13

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

$$\vec{F}_{rep}(q) = -\vec{\nabla} U_{rep}(q)$$

(12)

$$= \begin{cases} \eta \left(\dfrac{1}{\rho(q)} - \dfrac{1}{\rho_0} \right) \dfrac{1}{\rho^2(q)} \vec{\nabla} \rho(q) & \text{if } \rho(q) \le \rho_0 \\ \\ 0 & \text{if } \rho(q) > \rho_0 \end{cases}.$$

Let $\mathbf{q_c}$ be the unique configuration in CB that is closest to $\mathbf{q}$, i.e. that achieves $\|\mathbf{q} - \mathbf{q_c}\| = \rho(\mathbf{q})$. The gradient $\vec{\nabla} \rho(\mathbf{q})$ is a unit vector pointing away from *CB* and supported by the line passing through $\mathbf{q_c}$ and $\mathbf{q}$.

If we retract the (unrealistic) assumption that *CB* is convex, $\rho(\mathbf{q})$ remains differentiable everywhere in C_{free}, except at those configurations $\mathbf{q}$ for which there exist several $\mathbf{q_c} \in CB$ verifying $\|\mathbf{q} - \mathbf{q_c}\| = \rho(\mathbf{q})$. These configurations form a set of measure zero in C which is in general locally (N - 1)-dimensional. The force field $\vec{F}_{rep}$ is defined on both sides of this set, but with differently oriented vector values. This may result in producing paths that oscillate between the two sides of the set. At those configurations $\mathbf{q}$ for which there exists only one $\mathbf{q_c} \in CB$ verifying $\|\mathbf{q} - \mathbf{q_c}\| = \rho(\mathbf{q})$ at a time, there may be big jumps between the angular directions of consecutive $\mathbf{q_c}$ configurations during the motion of robots. This may cause serious oscillatory behaviors which in turn demolishes the robustness in motion.

One way to eliminate this difficulty is to find out the exact profile of the C-obstacle region *CB*, and then make the motion planning. This requires too much computation and it is not possible to obtain the exact profile with available sensors. Another way is to decompose *CB* into convex components CB_i, $i = 1, \dots r$, and to associate a repulsive potential U_{CBi} with each component. This is easy to implement in Nomad 200

14

mobile robot by using 16 sensors that have 22.5° between two consecutive sensors, which automatically decomposes *CB* into 16 components. Each obstacle detected by any sensor (whether it is a separate obstacle or a part of a big obstacle) produces a repulsive force. The resulting repulsive force is the sum of repulsive potential fields created by each individual sensor contact. Thus, the resulting artificial repulsive force is:

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

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 considered if the returned value is within maximum infrared sensor range, and the sonar information is considered otherwise.

III. FORMATION OF A LINE SEGMENT

This chapter discusses the results and associated problems of the existing algorithm when it is implemented with physical robots. Then a modified version of the existing algorithm which works well for physical robots is presented.

A. EXISTING LINE ALGORITHM

The following is the original line algorithm proposed in Ref. 5. It is assumed that each robot repeatedly becomes active and inactive (sleep mode) at unpredictable time instants. Each time a robot becomes active, it does the following:

- **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 point p that is the foot of the perpendicular drop from itself to the line passing through R_c and R_f.

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

Assuming that each robot is a dimensionless point, the algorithm enables all robots to form a line segment [Ref. 5]. In a revised version of the algorithm, the physical dimension of the robots was considered, and a simple collision avoidance strategy was implemented [Ref. 4]. The strategy works as follows:

- If a robot detects another robot within a certain distance in the direction of its move, it then swerves to the left minimally, provided that it successfully finds a direction that is clear of any robots.

- If no such left swerve is found possible, the robot decides not to move until either its path becomes clear or a suitable left swerve becomes possible.

First, this algorithm was simulated without any collision avoidance scheme, and an unacceptable number of collisions was experienced which prevented robots from forming a line. Then the algorithm was implemented with the simple left-swerve collision

strategy and 20 simulations was run, each time with a random initial distribution of the robots. A number of problems with the implementation of the algorithm were observed.

A problem occurs when four or more robots are very close to each other and try to avoid collisions. The robot with whom each one is trying to avoid collision changes frequently (so do R_c and R_f for each robot), which in turn changes the goal configuration of each one. In this case the robots jam each other. Another frequently encountered problem is illustrated in Figure 2. For the convenience of discussion, let us name the robots R_1 to R_6 from the upper-right corner to the lower-left corner in Figure 2(a), respectively. Figure 2(a) is a typical distribution where robots get closer to form a line segment. At that moment, R_3 swerves to its left to avoid R_4. R_4 moves downward to its goal location, which is the perpendicular drop to the line passing through the closest robot R_3 and furthest robot R_5. At the same time, the other robots go through a similar process. Figure 2(b) illustrates the distribution of the robots a few iterations later.

As the simulation continues, robots reconvene into a distribution similar to the one shown in Figure 2(a). Robots repeat the motion sequences and fail to form a line segment.

It is observed that robots form a line segment only when the initial distribution is very close to a line segment and little or no collision avoidance is required. Finally, it is noted that a uniform distribution of robots along the line segment cannot be accomplished using this method.

Aiming to overcome the collision avoidance problem encountered in the above implementation, the left-swerve strategy was replaced with the potential field method. It turns out that the potential field method does not help at all. The robots show various group behaviors other than forming a line segment. The result of a simulation is discussed below.

Figure 3(a) shows the initial, random distribution of six robots while Figure 3(b) to 3(e) illustrate their progressive movements. As soon as the simulation begins, robots start getting closer to each other as a natural result of the algorithm. A problem occurs when point p is within the physical dimensions or repulsive force range of a robot or

between two robots that do not have enough room in-between for another robot. Unfortunately this happens in most simulations, unless the initial distribution is very close to a line segment.
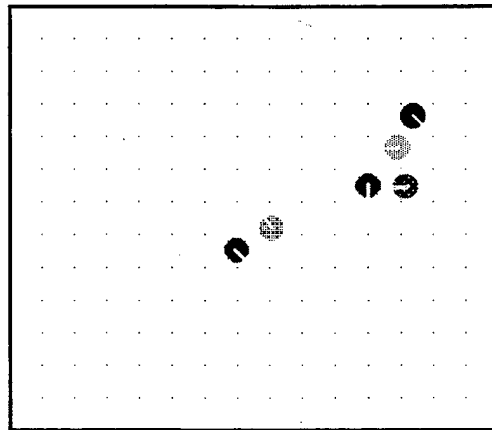
(a)
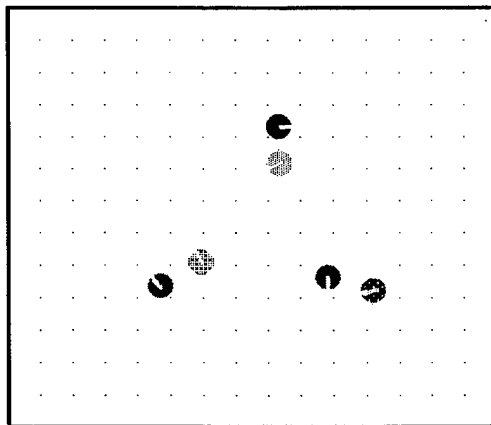
(b)

Figure 2. Implementation of the existing algorithm with the left-swerve collision avoidance strategy.

Figure 3. Selected images of the existing line algorithm simulation using the potential field method from an initial distribution to a final stage: (a) the initial distribution, (b)-(e) intermediate steps, and (f) the final distribution of the robots trying to form a line segment. The two robots on the lower-left side are approximately where they should be while the remaining ones are in deadlock.
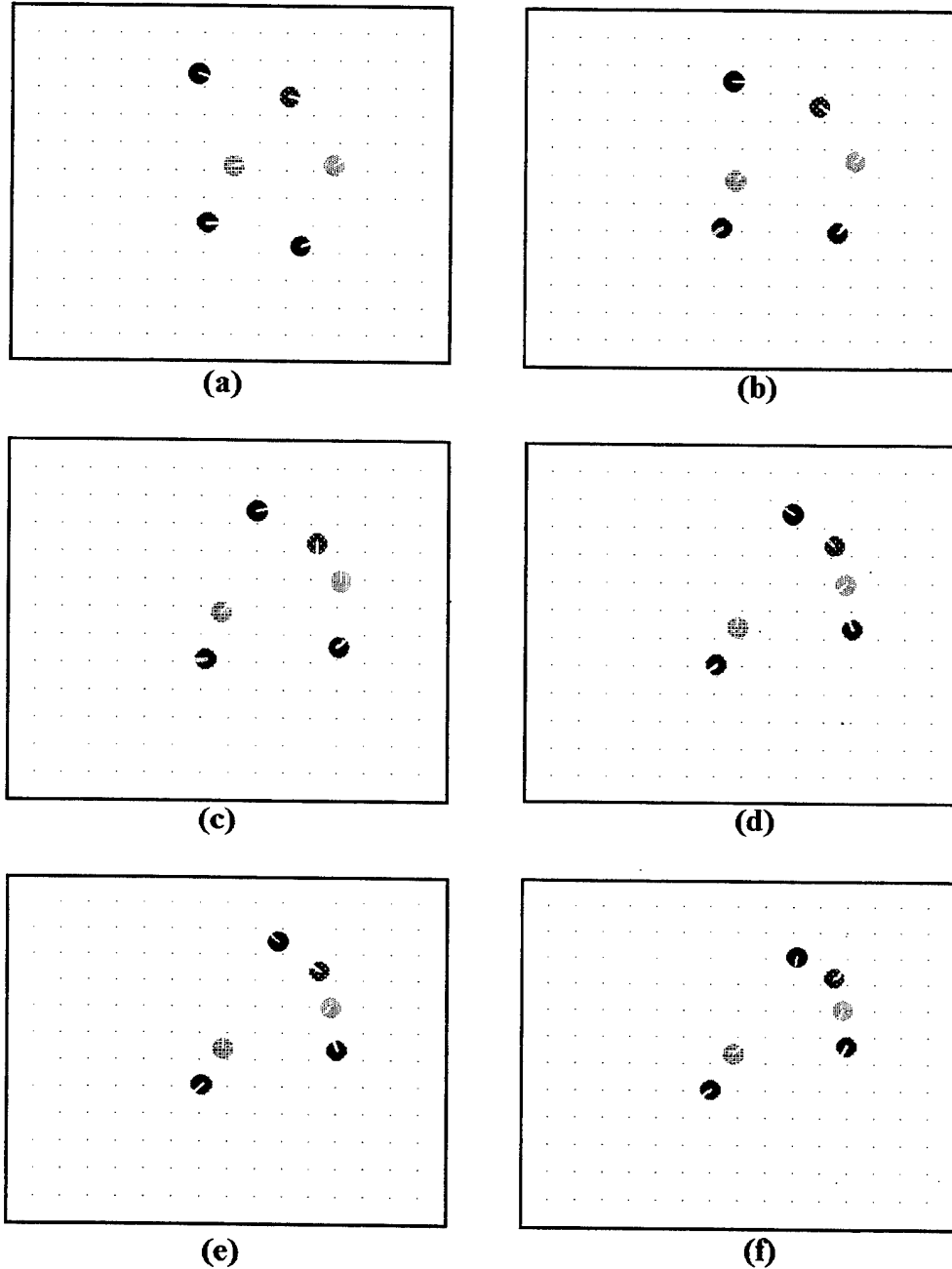
20

Eventually robots approach a deadlock configuration as shown in Figure 3(f). In this configuration, the attractive force generated by goal point p is negated by the repulsive force generated by surrounding robots. Consider the robot at the top in Figure 3(f), for instance. Its furthest robot R_f is the upper one in the two-robot group (it cannot see the lower one in the group), and its closest robot R_c is the closest one in the usual sense. Goal point p in this case corresponds to a point within the closest robot's repulsive force range. This is also true for all other robots in the four-robot group.

Another line algorithm is proposed in Ref. 1 and Ref. 4. The user selects two robots that will be the endpoints of the line segment and does the following, possibly concurrently:

- Move the two robots manually to their desired final positions.

- Let all other robots execute algorithm FILLPOLYGON

where FILLPOLYGON is a technique proposed in Ref. 4 to distribute robots nearly uniformly within a convex polygon, starting from an arbitrary initial distribution except anomalies.

This algorithm is not implemented in this study as the first task is clearly not distributed, and it does not coincide with the autonomous characteristics of distributed mobile robots.

B. MODIFIED LINE ALGORITHM

Although the existing algorithm does not work well for physical robots to form a line segment, its main idea is still valid. As discussed above, a problem occurs if the goal point p on the line passing through the closest (R_c) and furthest (R_f) robots is occupied by another robot, or if there is not enough room for another robot at the goal point. To circumvent the problem, the algorithm was modified so that the goal point p is still chosen to be on the same line, but at a location where there is room for another robot. Since each robot executes the same program, the discussion below is concerned with a

robot called R for convenience, which can be any one of the robots. At each iteration, robot R does the following:

- **Step 1.** Determines the closest robot R_c and furthest robot R_f based on its sensor readings.

- **Step 2.** Determines if point p, the foot of the perpendicular drop from its current position to the line l passing through R_c and R_f, is between R_c and R_f.

- **Step 3.** If yes, and if there is enough room for robot R to fit in-between R_c and R_f, it proceeds towards the mid-point, which is denoted by p_m.

- **Step 4.** If p is not between R_c and R_f, or if there is not enough room between R_c and R_f, it proceeds towards point p_d on the line l which is d distance away from R_c in the opposite direction of R_f, where d is the minimum distance that would prevent any repulsive force being applied to either robot.

Figure 4 shows the results of a simulation of this algorithm. Figure 4(a) is the same starting distribution as in Figure 3(a). Figures 4(b) through 4(e) show some selected intermediate distributions while Figure 4(f) illustrates the final distribution.

Some comments on the algorithm are in order. As mentioned earlier, the potential field algorithm is utilized to avoid collision. If d_o is used to denote the cut-off distance of repulsive forces in the potential field algorithm, any obstacles (in this case other robots) which are less than d_o distance away from robot R will generate repulsive forces to robot R. In Step 3, when determining if there is enough room to fit another robot between R_c and R_f, the distance from R_c and R_f must be at least:

$$\left| R_c R_f \right| = 2r_0 + 2d_0 , \tag{14}$$

where r_o is the radius of the robots. (A Nomad 200 robot is cylindric in shape. In the simulator, it is represented by a circle.) That is, if there is at least $2 r_o + 2 d_o$ distance between R_c and R_f, robot R will be able to ``squeeze" in. This is true even if there are other robots between R_c and R_f.
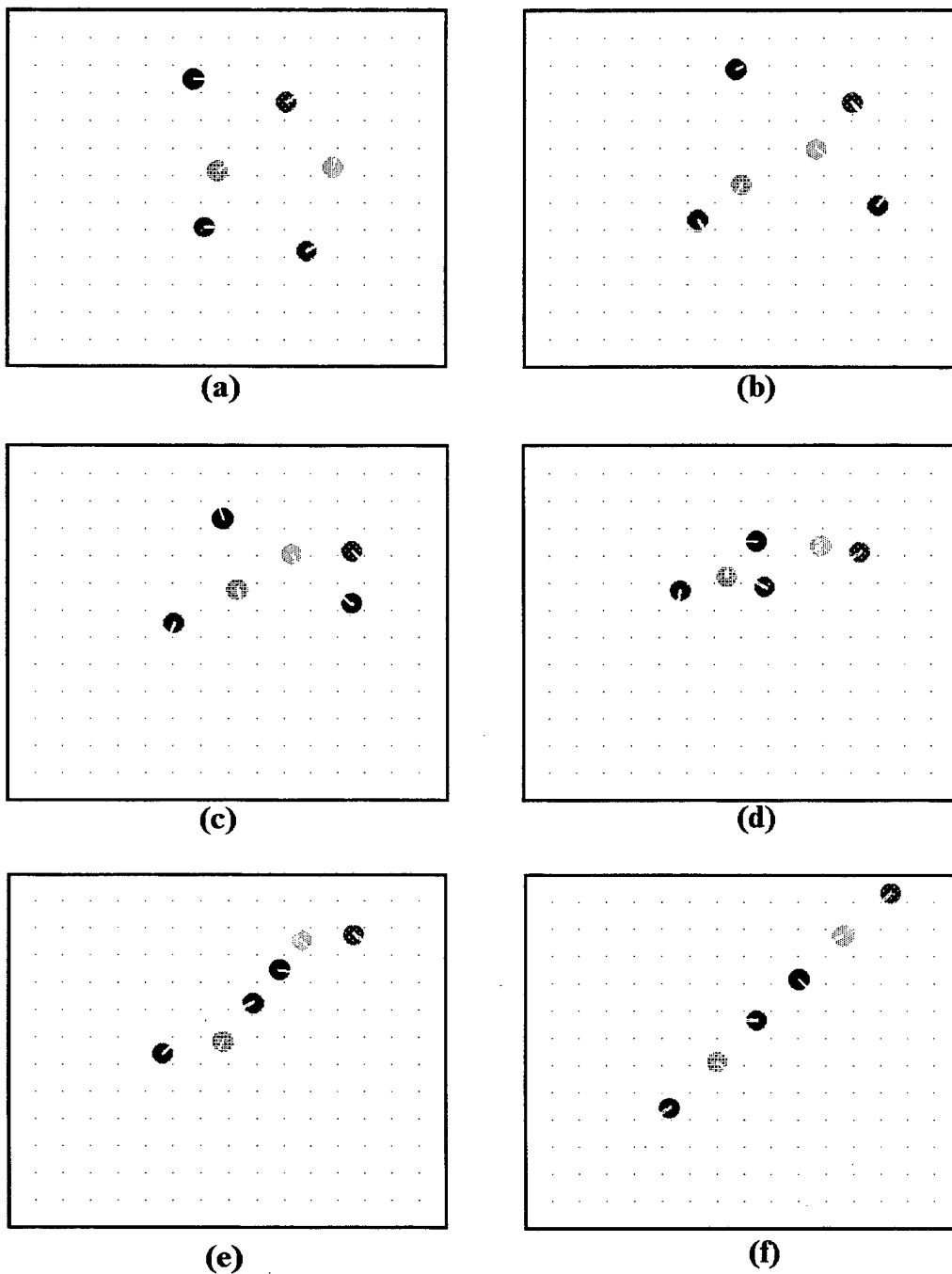
Figure 4. Selected images of the modified line algorithm simulation: (a) the initial distribution, (b)-(e) intermediate steps, and (f) the final distribution of the robots forming a line segment.

23

In Step 4, the distance d is given by

$$d = 2r_0 + d_0 \ . \tag{15}$$

There is, however, a problem in implementing Step 4 if point p is between R_c and R_f. Sending robot R directly to p_d mostly results in a local minimum while the robot tries to move to the other side of R_c. This is inevitable if the robot's direct route to p_d is within the repulsive force range of R_c. This problem is avoided by sending the robot to an intermediate point which is d distance away from R_c, on the line that is perpendicular to l at R_c. There are two points on this perpendicular line which are d_o distance away from R_c, but there is an obvious choice, the one which is closer to robot R. As soon as robot R reaches this intermediate point within a close proximity, its goal point is changed to p_d.

If robot R detects only one robot nearby (which is the case if it is at the endpoint of the line segment), it positions itself d_o distance away from the detected robot. If robot R does not detect any robots nearby, it can execute an algorithm to search for possible existence of other robots, which is not in the scope of this study.

Finally it is noted that all robots will uniformly distribute in the line segment because each robot tries to go to the mid-point between its neighbors until it gets into the repulsive force range of its neighbors.

IV.   FORMATION OF A CIRCLE

This chapter first discusses the existing circle formation algorithm, and simulation results from implementing the existing algorithm. Observing problems encountered with the existing algorithm, a sequence of modification and improvement is proposed. The improvement is directed towards forming a better approximation of a circle while uniformly distributing robots on a circle.

A.   EXISTING CIRCLE ALGORITHM

Let robot R be any one of the distributed robots participating in the task of circle formation. The existing circle algorithm works as follows [Ref. 5]. As before, robot R becomes active and inactive at random instants of time. Each time robot R becomes active, it:

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

- **Step 2.**   Calculates the distance d from its current position to the middle point p_m between R_f and R_c.

- **Step 3.**   Moves a distance of $min\{d\text{-}r, v\}$ towards p_m if $(d\text{-}r) >= 0$, or a distance of $min\{r\text{-}d, v\}$ away from p_m if $(d\text{-}r) < 0$, 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.

Figure 5 shows the results of a simulation of this algorithm. The desired radius of the circle is 28.0 inches. (The radius of the Nomad robot is 9.0 inches.) Figure 5(a) is the initial, random distribution of robots. Figure 5(b) to (e) show the intermediate positions of robots, and Figure 5(f) illustrates the final stage of the simulation. The final distribution of robots is a good approximation of a circle, and robots are fairly uniformly distributed. However, the degree of uniformity depends on the number of robots. Figure 6(a) shows the final distribution of five robots. The distribution is apparently less uniform.

25
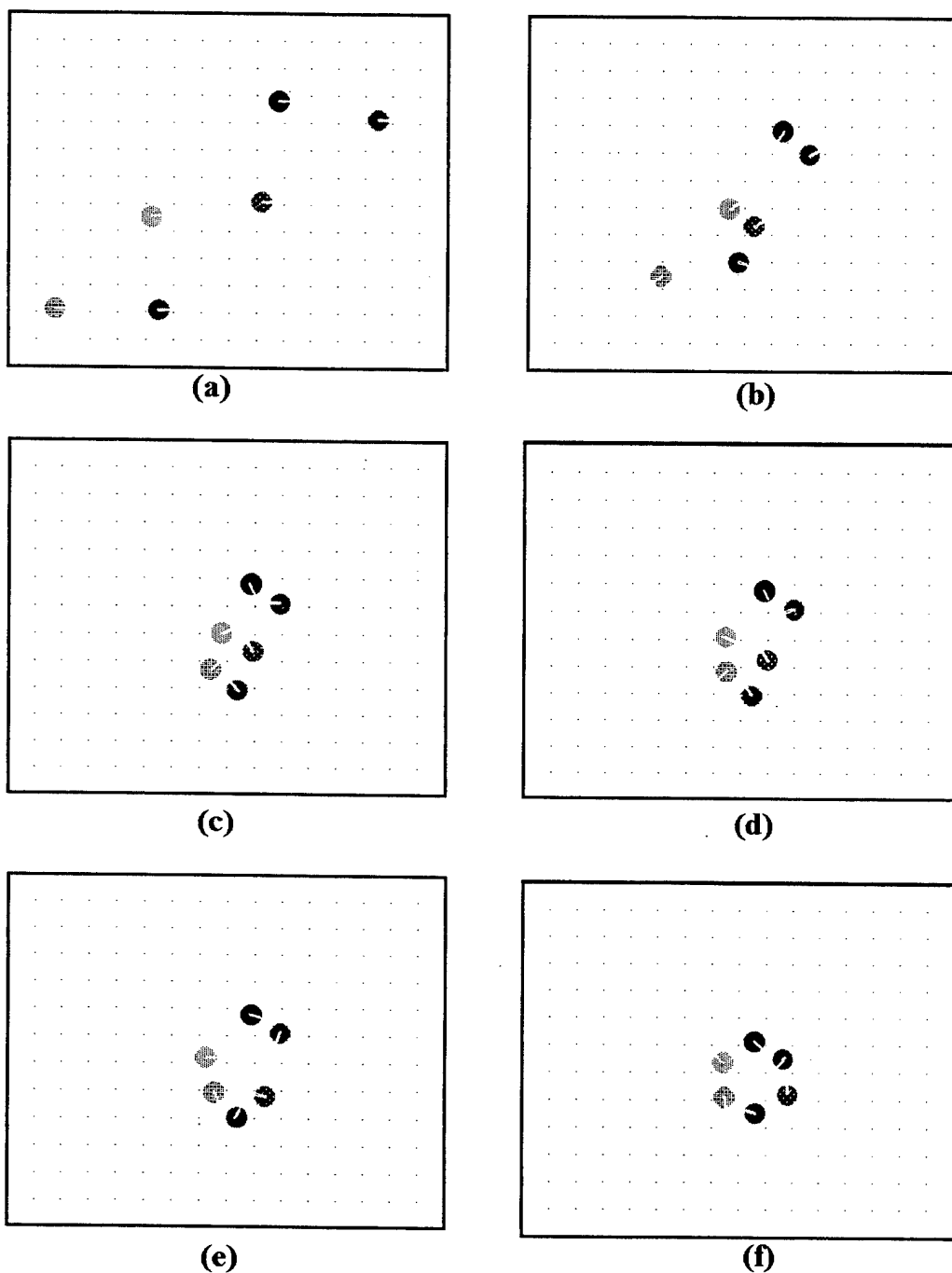
Figure 5. Selected images from a simulation of the existing circle algorithm implemented by using the potential field method: (a) the initial distribution, (b)-(e) intermediate steps, and (f) the final distribution of the robots forming a circle.
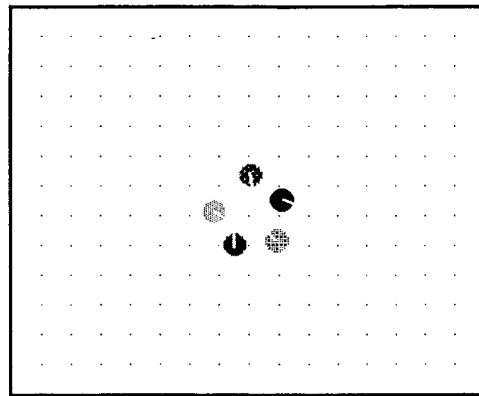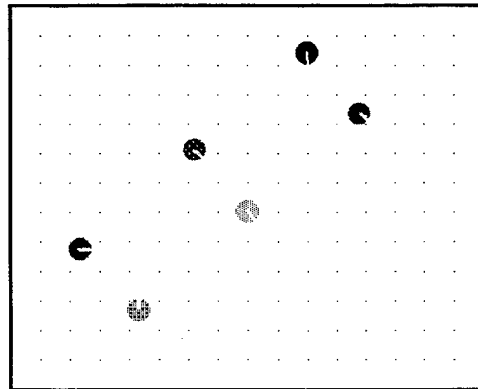
Figure 6. The final distributions of two more simulations of the existing circle algorithm started from the same initial distribution as the earlier simulation: (a)If a robot is missing, the remaining robots still form a circle, but are not uniformly distributed (r=28 inches). (b) The robots fail to form a circle for a relatively large desired radius (r=120 inches).

A minor problem is that the radius of the final circle is always smaller than the desired radius (20 inches versus 28 inches in this case). This is because p_m does not correspond to the origin of the circle. Consequently, the final formation appears as two half-circles put together. In Figure 5(f), the three lower-left robots form a half-circle as

27

do the three upper-right robots. This is the same source that causes robots to form *a Reuleaux's triangle* [Ref. 1, 4, 5].

Another problem occurs when the desired radius becomes relatively large. With limited sonar range, a robot is not able to see some robots as it moves outwards to form a large circle. With the same initial distribution as in Figure 5(a) and Figure 6(a), a simulation is carried out to form a circle with radius of 120 inches. The resulting distribution is shown in Figure 6(b). The pair of robots at the upper-right corner cannot see the two at the lower-left corner due to limited sensor range. In this case, the two robots at the upper-right corner and the two in the middle form a circle. Similarly, the two robots at the lower-left corner and the two in the middle form another circle.

Finally, it is noted that algorithm works the same way but much faster when it is implemented without using any sleep mode (making the robots active or inactive at unpredictable time instants).

B.   MODIFIED CIRCLE ALGORITHM

In this subsection, a modified circle algorithm is presented. The objective of the modified algorithm is to yield a better approximation of circles, and to uniformly distribute robots. The existing algorithm utilizes position information of the furthest and closest robots. It is attempted to improve it by utilizing position information of one more robot. More specifically, the steps of the modified algorithm are as follows. At each iteration, robot R:

- **Step 1.**   Determines the furthest robot R_f, the closest robot R_{c1}, and the second closest robot R_{c2}.

- **Step 2.**   Computes the coordinates of the centroid p_m of R_f, R_{c1}, and R_{c2}.

- **Step 3.**   Moves to the point p_r which is r distance away from p_m, on the line that passes through its current position and p_m, where r is the desired radius of a circle to be formed.

Figure 7 shows the results of a simulation of the modified algorithm with a desired circle of 28-inch radius.

(a)

(b)

(c)

(d)

(e)

(f)
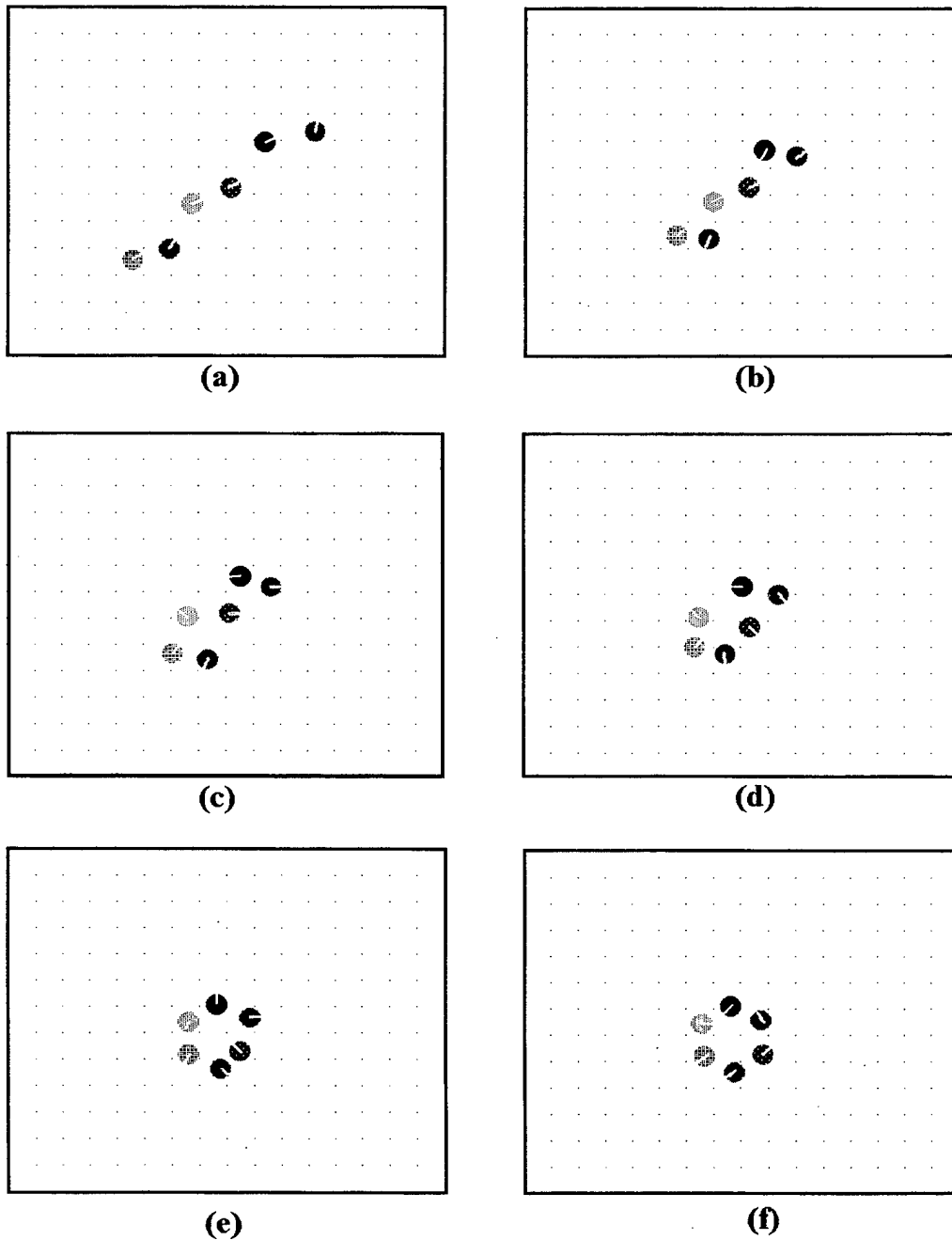
Figure 7. Selected images of a simulation of the modified circle algorithm: (a) the initial distribution, (b)-(e) intermediate steps, and (f) the final distribution of the robots forming a circle.

In step 3, while robot R tries to move towards or away from p_m in order to take its position on p_r, it is pushed by R_c and R_{c2}, which makes robot R move to the sides. An advantage of the modified algorithm is that p_m is closer to the origin of the desired circle, which makes the final formation a much better approximation of a circle. The radius of the resulting circle is still smaller than the given radius (21 inches versus 28 inches).

Another advantage is that robots will be uniformly distributed along a circle, independent of the number of robots. Nevertheless, it is observed that smaller number of robots tend to form a smaller circle. Figure 8 shows the final results with five and four robots.
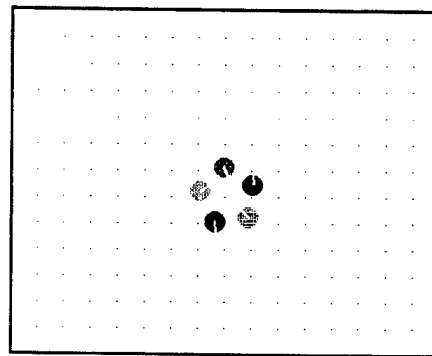
(a)
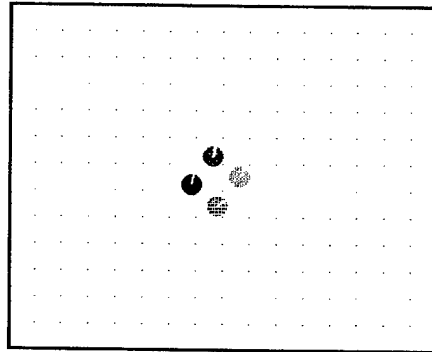
(b)

Figure 8. Final distributions of the modified circle algorithm simulations with five and four robots.

V.   MOVING IN FORMATION

In the previous chapters, the problem of formation was addressed. This chapter discusses a rather different problem: moving in formation. Starting from an initial formation, the problem of moving in formation is concerned with how to move robots from an initial location to a final goal location while maintaining the formation and avoiding obstacles. If the initial formation cannot be maintained due to obstacles, robots must stay in a formation as close as possible to the initial one.

A.   ASSUMPTIONS

Additional assumptions made for this section are :

- There is a global coordinate system, and the final goal location is passed to each robot.

- There is a bulletin board that can be reached by all the robots. Each robot is assigned a specific location in the bulletin board and each robot writes its position information always into its assigned location.

B.   IMPLEMENTATION

Three robots are used in implementation in this section as *Nserver* does not show 100 percent reliability with larger number of robots in implementation of the bulletin board communication scheme. *"Leader-Follower"* technique is chosen for formation position determination. In this technique, each robot determines its formation position relative to the leader robot. The leader simply moves towards the final goal location independently while other robots try to maintain their positions in the formation relative to the leader. The robots are all identical and the leader is not appointed, but it is selected by the robots at run time.

At their initial locations, the robots only know the final goal location. Each one writes its current position information into its assigned location in the bulletin board. Then it calculates the distances of all robots from the final goal location, based on the

position information it reads from the other robots' assigned locations in the bulletin board. The closest one to the final goal location is selected as the leader. The location number in the bulletin board which consists of the position information of this robot is memorized by all robots; they know where the leader's position information is available. Once the leader is selected, only the leader keeps writing its position to the bulletin board in every iteration. The other robots do not need to do it as their position information do not need to be known by any others.

Each robot determines its position relative to the leader and remembers it throughout the whole journey. The first movement for all the robots is to turn their faces towards the final goal location. This is to find out which side is the leader located. Each follower robot needs to know if it is on the left or on the right side of the formation as to the direction of motion. For instance, if a robot sees the leader on its right side initially, then it will move towards the formation's main body by leaving the obstacle on his left when avoiding collisions. If the leader is initially on its left, then it will leave the obstacle on its right and move towards the formation's main body when avoiding obstacles.

The leader moves towards the given final goal location independently, while avoiding any collision based on the potential field method. Each of the other robots determines a new goal location for itself at each iteration based on the current position of the leader and moves to this point while avoiding any collision, again based on the potential field method. The follower robots move slightly faster than the leader in order to catch it and maintain the formation. The further the follower robot is from its goal location at each iteration, the faster it moves. This is very crucial because the followers stay behind when avoiding collisions with obstacles while the leader keeps moving.

It can be explained better using Figure 9 which shows the simulation results from an initial to a final distribution. The final location for the robots in Figure 9(a) is given to be a point which is further on the right side of the area shown. The robots concur with the leader, they face towards the final goal point, the followers find out which side is the leader located, and determine their positions relative to the leader. Then, the leader starts moving towards the final goal point while the others start following it. Since the robot on

the very right in Figure 9(a) is the closest one to the final destination, it will be the leader. (Let's denote the leader as R_L, the robot on the top side as R_T, and the robot on the bottom side as R_B).

In Figure 9(b), R_T comes across an obstacle. After having a very slow speed (2 inches/sec. in simulations) two successive times although it is away from its goal location, it realizes that it is stuck and turns towards the main body of the formation and leaves the obstacle on its left. Then it keeps moving while keeping the obstacle on its left hand side. This makes the robot follow the edge of the obstacle and turn around the corner of the obstacle. As soon as R_T realizes that it moved towards the final goal location more than 10 inches in one iteration, it goes back to the normal flow of the program, determines a new goal position for itself relative to the current position of the leader, and starts moving. The new goal location may cause the robot to immediately experience another local minimum, especially if the avoided obstacle is a big one, which makes the robot go to the same process one more time.

The follower robots move faster when they are avoiding an obstacle. Otherwise they would stay very much behind the leader. R_L avoids obstacles always by leaving them on its right and moving left as to the direction of general motion of the formation, as appeared on Figure 9 (c) and (d). R_B avoids the obstacles by leaving them on its right and moving towards the main body of the formation as shown on Figure 9 (h) and (i). Moving to the opposite direction would cause the follower robots go further away from the formation, which in turn makes it difficult to catch.

The bulletin board scheme is implemented by using a built in function called *get_rpx()* which is not specified in the language manual but included in *Nclient.h* and *Nclient.o* files. This function uses the UNIX sockets to communicate with the server and get the position information of all the nearby robots. It has been a little bit difficult to figure out how exactly it worked as it was not stated in the language manual. Therefore, it is believed to be useful to include it here, in order to ease the understanding of the source code.

33
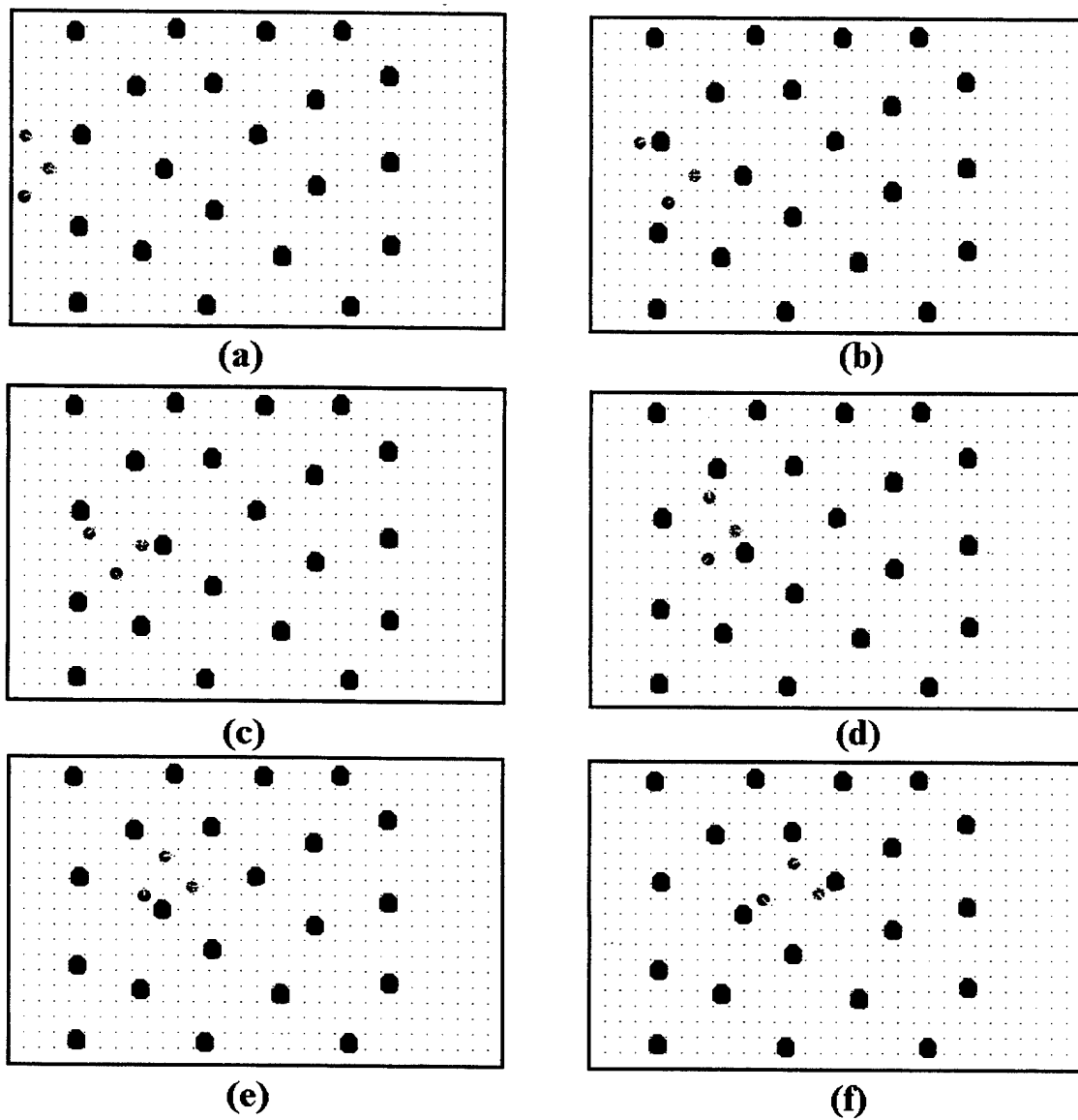
Figure 9. Selected images of the simulation of the moving in formation algorithm from an initial distribution to a final stage. (a) the initial distribution, the robot on the very right is selected to be the leader, (b)-(f) and (g)-(k) on the next page are intermediate steps,
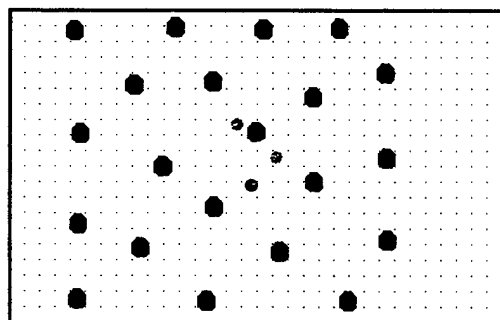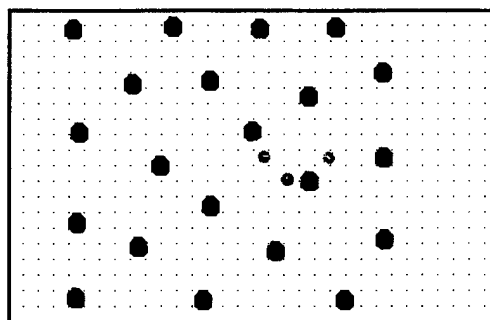
(g)   (h)
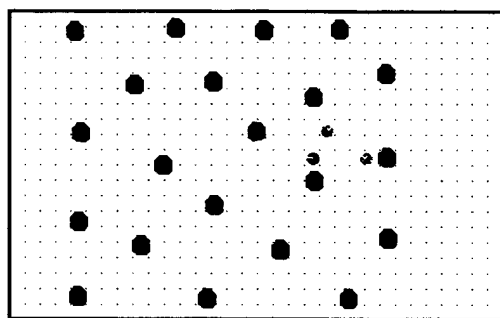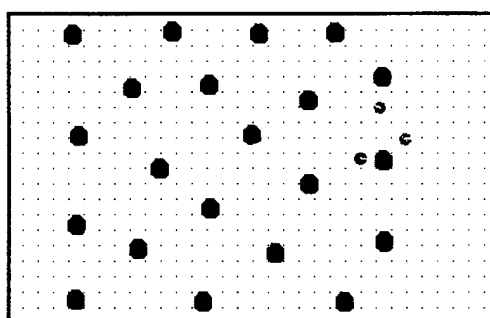
(i)   (j)
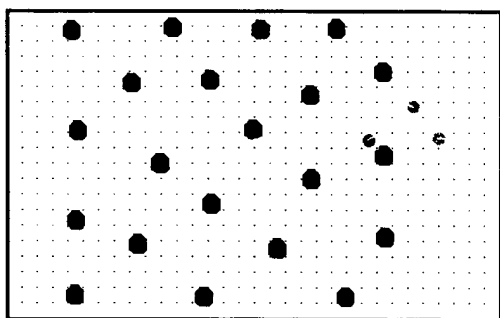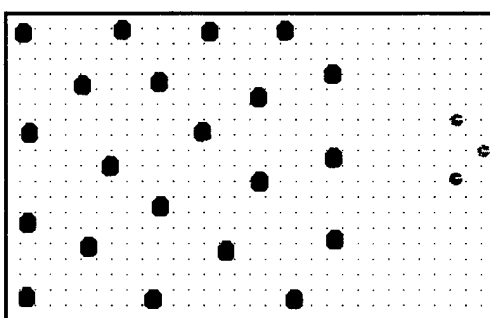
(k)   (l)

Figure 9. Continued. Leader moves independently towards the final goal location while others try to maintain their positions relative to the leader while avoiding obstacles, and (l) the distribution of the robots after avoiding all the obstacles and shortly before arriving the final goal location. Three identical small circles represent the robots, while big polygons represent obstacles.

A pointer is passed to the function which in return points to the position information of the nearby robots. The first value in the returned array (*pos_array[0]*) specifies the number of robots around. Each following group of three values in the array returns information about a robot. The first ones in each group of three (*pos_array[1]*, *pos_array[4]*, *pos_array[7]*, etc.) specifies the robot's ID number. The following two values in each group contain the x and y coordinates of that robot (i.e. *pos_array[2]*, *pos_array[5]*, *pos_array[8]* contain the x-coordinate, and *pos_array[3]*, *pos_array[6]*, *pos_array[9]* contain the y-coordinate of the robot whose number is specified in *pos_array[1]*, *pos_array[4]*, *pos_array[7]* respectively). The x and y coordinates of each robot are presented relative to the robot itself in the world coordinate system. For instance, if the robot who is calling this function is located at $x=100$, $y=50$ in the world coordinate system, and if there is only one robot around whose ID number is 5 and who is located at $x=350$, $y=200$ in the world coordinate system, then the function returns the following values: *rob_pos[0]=1, rob_pos[1]=5, rob_pos[2]=250, rob_pos[3]=150*.

The function *get_rpx()* does not return any information about the robots that are not in the sensor range or are completely behind an obstacle. If a follower robot cannot get the position information of the leader in such cases, which is encountered occasionally, it simply moves towards the final goal position with a higher speed until it gets the position information of the leader again.

Any robot can turn backwards while avoiding a collision or trying to maintain its position and carry on moving backwards later on as it happens to the leader on Figure 9 (g) and (h). This is a natural result of using physical robots, and it does not affect the flow of the motion at all. Robots repeat the motion sequences as appeared in the rest of the Figure 9 and maintain the initial formation as much as possible while avoiding obstacles on the way to the specified final goal configuration.

VI. CONCLUSIONS AND RECOMMENDATIONS

Line and circle formation of distributed mobile robots was studied. Motion control and collision avoidance were achieved by implementing a potential field algorithm. It was observed that existing line and circle formation algorithms do not work well when implemented with realistic robots. Modified line and circle algorithms were developed and verified through simulation. As demonstrated, the proposed algorithms not only achieve the goal of forming a line or a circle, but they also uniformly distribute robots on a line segment or circle. Behaviors for formation maintenance in a small group of physical mobile robots was also studied. An algorithm based on "Leader-Follower" technique was implemented for relative positional maintenance to help robots move to a specified goal location in formation while avoiding collisions with obstacles and other robots.

Follow-on work on the line and circle formation problems should focus on improving the convergence rate of formation and incorporating obstacles (other than robots themselves) in workspace. Studies on the moving-in-formation problem, on the other hand, should focus on implementing a means of implicit communication in which robots rely entirely upon their own perceptions of the world.

APPENDIX A. SOURCE CODE FOR MODIFIED LINE ALGORITHM

```
/* This is the source code of the modified line algorithm presented in
chapter III.  Some command line arguments must be passed to the program
in order to run it.  The first argument is the ID number of the robot
that will be used to connect to the server, which is compulsory. Second
and third arguments are the desired initial x and y coordinates of the
robot, and these are optional. If the current position of the robot is
desired to be the initial configuration, then no x-y coordinates need to
be entered. So, there are two ways to run the program.
        1. Filename ID x y
        2. Filename ID
One copy of this program must be run for each robot with a different ID
numbers.  Nserver allows to simulate up to six robots, hence the robot
ID number should be  between 1 and 6.  */

#include "Nclient.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define PI 3.1415926

/* Function Prototypes */
void GetSensorData(void);
void Movement(void);
int  sign(int);
void calculate() ;

/* Global Variables */
double fused_range[16]; /* Array to store the fused sensor readings */
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        */

double F_att[2], F_rep[2], F_tol[2] ; /*Arrays to store the attractive,
                                      repulsive and total forces. */
double preXgoal, preYgoal ;
int minreturn, maxreturn, iteration ;
long mindist, maxdist ;
int count = 0 ;

/*** Main Program ***/

main (unsigned int argc, char* argv[])
{
   int i,index;
   int order[16];

   int Robot_ID = atoi(argv[1]);
   int X,Y;
```

```c
/* Check the command line arguments. There should be either one or
three command line arguments */
if (argc!=4) {
  if (argc!=2) {
    printf("please enter 3 parameters besides the command\n");
    exit();
  }
}


/* Check the robot ID, IT should be between 1 and 6 */
if ( (Robot_ID<1) || ( Robot_ID>6) ) {
  printf("Robot ID must be between 1 and 6 ");
  exit();
}


/* Connect to Nserver.*/
SERV_TCP_PORT=7771 ;
strcpy(ROBOT_MACHINE_NAME, "nomad");
connect_robot(Robot_ID);


/* If the initial x-y coordinates are entered, store them into
variables*/
if (argc==4) {
    X = atoi(argv[2]);
    Y = atoi(argv[3]);
    place_robot(X,Y,0,0);
}

/* 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 it becomes disconnected. */
conf_tm(5);

/* 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 an element of value -1. 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);


/* Configure the Infrared Sensors */
for (i = 0; i < 16; i++)
  order[i] = i;

conf_ir(1,order);

/* Fortunately, the robot can talk... */
tk("Start to form a line");
```

```c
   iteration = 0 ;

   /* Main loop. Execute unless a collision occurs. */
   while (!BumperHit)
     {
       GetSensorData();
       Movement();
     }

   /* Disconnect. */
   disconnect_robot(Robot_ID);
}

/*****************************/

/* Movement(). This function is responsible for using the sensor data to
direct the robot's motion appropriately. */
void Movement (void)
{
   /* Variables */
   int i;
   int panic,flip ;
   int tvel, svel ;
   double gain_tvel = 0.07;     /* Translational velocity gain */
   double gain_svel = 100.0;    /* Rotational velocity gain    */

   /* Compute the attractive force and the repulsive force in the robot
   coordinates by using function calculate() */
   calculate() ;

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

   /*  Set the rotational velocity. If both minimum and maximum sensor
   readings are 255 which is the maximum sonar range, it means there is
   no contact around.  Then proceed by turning 5 degrees in each
   iteration, so that robot will be able to cover the area by making a
   big circle. Otherwise set the rotational velocity based on the total
   forces acting on robot. */

   if(mindist==255 && maxdist==255)
        svel = 50 ;
   else
        svel = (int)(gain_svel * sin(atan2(F_tol[1],F_tol[0])));

   /* Set the direction of rotation;left or right. */
   svel = svel * sign( (int) (F_tol[0]) );

   /* limit the translational  and rotational velocities. Maximum allowed
   translational velocity is 24 in/sec, and rotational velocity is 45
   degrees. */
   if (abs(tvel) > 240)
     tvel = 240 * sign(tvel);

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

41

```c
/* 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);

}


/*****************************/

/* This function reads sensor data and loads them into arrays. */
void GetSensorData (void)
{
    int i ;
    long SonarRange[16] ;       /* Array to store sonar readings */
    long IRRange[16] ;          /* Array to store infrared readings */
    double corrected_IR[16] ;   /* Array to store correlated infrared
                                   readings */
    double corrected_sonar[16] ;/* Array to store the corrected sonar
                                   readings */
    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 infrared 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]; /*From long to double*/

    /* Fuse the sonar and IR data, and store the final sensor reading into
    the global array fused_range[]. Infrared readings are not reliable
    beyond 14 inches. If the correlated value is smaller than 14, then
    normalize and fuse the infrared and sonar data. If it is more than
    14, then consider the sonar data. */
```

42

```c
    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
    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");
    }

    /* Find out the sensor ID number which detects the closest robot Rc */
    minreturn = 0;
    for (i = 1 ; i < 16 ; i++) {
      if (fused_range[i] < fused_range[minreturn])
          minreturn = i ;
    }

    /* The distance to the closest robot */
    mindist = fused_range[minreturn];

    /*Find out the sensor ID number which detects the furthest robot Rf*/
    maxreturn = minreturn ;
    for (i = 0 ; i < 16 ; i++) {
      if((fused_range[i]>=fused_range[maxreturn])&&(fused_range[i]<255.0))
        maxreturn = i;
    }

    /* The distance to the furthest robot */
    maxdist = fused_range[maxreturn];
}
/*******************************/

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

```
/*****************************/

/* This function computes the attractive force as to the goal point in
robot coordinate system and the repulsive force as to the obstacles in
robot coordinate system. Finally it calculates the x and y components of
the total forces on the robot in robot coordinate system. */

void calculate()
{
    double rho_0 = 40.0;   /* Cut-off distance of the repulsive force */
    double scale = 8.0 ;       /* Scaling factor for attractive force */
    double eta = 35000.0 ;     /* Scaling factor for repulsive force */
    double rho_att = 40.0 ;    /* Saturation distance for attractive
                                  force to switch between parabolic-well and
                                  conic-well */

    double rho_float ;
    double teta;
    double range = (2*rho_0) + (2*8.81) ; /* Minimum distance between
                                             closest and furthest robot to
                                             allow another robot to fit in.*/
    int i ;
    int signal = 0 ;
    int flag  = 0 ;
    int set   = 0 ;
    int alone = 0 ;

    double xx1,xx2,yy1,yy2 ;
    double length, slope, dist, distance ;
    double xgoal, ygoal ;
    double ya,yb,xa,xb ;

    /* The x and y coordinates of the closest robot */
    xx1 = ((double)mindist+17.62)*cos((double)minreturn*0.39) ;
    yy1 = ((double)mindist+17.62)*sin((double)minreturn*0.39) ;

    /* The x and y coordinates of the furthest robot */
    xx2 = ((double)maxdist+17.62)*cos((double)maxreturn*0.39) ;
    yy2 = ((double)maxdist+17.62)*sin((double)maxreturn*0.39) ;

    /* The distance between closest and furthest robots */
    length = hypot ( (double)(yy2-yy1),(double)(xx2-xx1) );

    /* Compute the coordinates of the point p, the foot of the perpendi-
    cular drop from robot's current position to the line l passing through
    Rc and Rf. Watch out the conditions that would lead to a division by
    zero error. */
    if ( xx1 == xx2 ) {
        if (yy1 == yy2) {        /*Means there is only one contact around*/
            dist = hypot((double)xx1,(double)yy1) ;
            printf("dist= %f\n", dist);
            xgoal = xx1 - (xx1*range/dist) ;
            ygoal = yy1 - (yy1*range/dist) ;
            alone = 1 ;
        }
        else {
            xgoal = xx1 ;
```

44

```
        ygoal = 0  ;
        signal = 1 ;
    }
}
else {
   if (yy1==yy2) {
       xgoal = 0 ;
       ygoal = yy1 ;
       signal = 2 ;
   }
   else {
       slope = (yy2-yy1)/(xx2-xx1) ;
       xgoal = ( yy1-(slope*xx1)) / ((-1/slope) - slope );
       ygoal = (-1/slope)*xgoal ;
       distance = hypot(xgoal,ygoal) ;
       signal = 3 ;
   }
}

/* Check to see if the point p is between Rc and Rf. If so, there is
enough room for a robot to fit in-between Rc and Rf. Then proceed
towards the mid-point. */
if (((xgoal>=xx1)&&(xgoal<=xx2))||((xgoal<=xx1)&&(xgoal>=xx2))) {
   if (((ygoal>=yy1)&&(ygoal<=yy2))||((ygoal<=yy1)&&(ygoal>=yy2))) {
       set = 1 ;
       if (length >= range) {
           xgoal = (xx1+xx2)/2.0 ;
           ygoal = (yy1+yy2)/2.0 ;
           flag = 1 ;
           set  = 0 ;
       }
   }
}

/* If  point p is not between Rc and Rf, or if there is not enough
room between Rc and Rf, then the final goal location is the point pd
on the line l which is d distance away from Rc in the opposite
direction of Rf, where d is the distance that would prevent any
repulsive force being applied to either robot. */
if (!flag) {
   switch(signal) {
       case 0 : break ;
       case 1 : ygoal = sign(yy1)*(abs(yy1)-sign(yy1)*sign(yy2)*range) ;
                break ;
       case 2 : xgoal = sign(xx1)*(abs(xx1)-sign(xx1)*sign(xx2)*range) ;
                break ;
       case 3 : xa = xx1 + (range-rho_0)/sqrt(1+slope*slope) ;
                xb = xx1 - (range-rho_0)/sqrt(1+slope*slope) ;
                ya = yy1 + slope*(xa-xx1) ;
                yb = yy1 + slope*(xb-xx1) ;

                if ( (hypot(ya,xa)-hypot(yb,xb))<=0.0  ) {
                    xgoal = xa ;
                    ygoal = ya ;
                }
                else {
                    xgoal = xb ;
                    ygoal = yb ;
```

```c
            }
            break ;
    default: break ;
  }
}
/* If point p is between Rc and Rf but there is not enough room for
between Rc and Rf, proceeding directly to pd mostly results in local
minima when the direct route to pd is within the repulsive force
range of Rc. To avoid this problem, send the robot to an intermediate
point which is d distance away from Rc, on the line that is
perpendicular to l at Rc. As soon as the robot passes this point, in
the next iteration pm will not be between Rc and Rf and robot will be
sent to pd. */
if (set && !alone) {
    ya = yy1 + (range-rho_0)/sqrt(1+slope*slope) ;
    yb = yy1 - (range-rho_0)/sqrt(1+slope*slope) ;
    xa = xx1 + slope*(yy1-ya) ;
    xb = xx1 + slope*(yy1-yb) ;
    printf("ya,yb, loop executed\n");
    if ( (hypot(ya,xa)-hypot(yb,xb))<=0 ) {
        xgoal = xa ;
        ygoal = ya ;
    }
    else {
        xgoal = xb ;
        ygoal = yb ;
    }
}


/* If Rc and Rf are seen by two consecutive sensors and the
difference between the sensor readings are less than 4 inches, then
there is only one contact around that is seen by two sensors. This
happens when the endpoint robots are moving to their positions,
after a certain point the robot cannot see the other robots. In this
case use the last goal values three times to send the robot to its
correct position, in order to form a straight line. Then simply send
the robot to the d distance away from the only detected robot. */
if ((abs(minreturn-maxreturn)==1 )||(abs(minreturn-maxreturn)==15)){
    if ( abs(mindist-maxdist)<=4 ) {
        if (count < 3 ) {
            xgoal = preXgoal ;
            ygoal = preYgoal ;
            count += 1 ;
        }
        else {
            dist = hypot((double)xx1,(double)yy1) ;
            xgoal = xx1 - (xx1*range/dist) ;
            ygoal = yy1 - (yy1*range/dist) ;
            count = 0 ;
        }
    }
}

distance = hypot(xgoal,ygoal); /*distance to the goal configuration*/

/* Store the goal configuration to remember in the next iteration */
preXgoal = xgoal ;
preYgoal = ygoal ;
```

```c
/* Compute the attractive force in the robot coordinate system */
if (distance <= rho_att) {          /* Parabolic Well */
    F_att[0] = scale*xgoal ;
    F_att[1] = scale*ygoal ;
}
else {                              /* Conic Well */
    F_att[0] = scale*rho_att*(xgoal/distance) ;
    F_att[1] = scale*rho_att*(ygoal/distance) ;
    printf("distance2= %f \n", 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) (fused_range[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);
    }
}

/* 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];

}
```

47

APPENDIX B. SOURCE CODE FOR MODIFIED CIRCLE ALGORITHM

```
/*  This is the source code of the modified circle algorithm presented
in  chapter  IV.    Some  command  line  arguments  must  be  passed  to  the
program in order to run it.  The first argument is the ID number of the
robot that will be used to connect  to the server, which is compulsory.
Second and third arguments are the desired  initial x and y coordinates
of the robot, and these are optional. If the current  position of the
robot  is  desired  to  be  the  initial  configuration,  then  no  x-y
coordinates  need  to  be  entered.  So,  there  are  two  ways  to  run  the
program.
        1. Filename ID x y                              .
        2. Filename ID
One copy of this program must be run for each robot with a different ID
numbers.   Nserver allows to. simulate up to six robots, hence the robot
ID number should be  between 1 and 6.  */


#include "Nclient.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define PI 3.1415926

/* Function Prototypes */
void GetSensorData(void);
void Movement(void);
int  sign(int);
void calculate() ;

/* Global Variables */
long SonarRange[16];     /* array of sonar readings (inches) */
long IRRange[16];        /* array of infrared readings (no units)*/
double fused_range[16];  /* Array to store the fused sensor readings */
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 */
double radius = 28.0 ;
double F_att[2], F_rep[2], F_tol[2]; /*Arrays to store the attractive,
                                     repulsive and total forces. */
int minreturn, maxreturn, minreturn2 ;
int count = 0 ;
long mindist, mindist2 ,maxdist;     /* Variables to store the distances
                                     of closest, second closest and
                                     furthest robots */

/*** Main Program ***/

main (unsigned int argc, char* argv[])
{
    int i,index;
    int order[16];
    int Robot_ID = atoi(argv[1]);
    int X,Y;
```

49

```c
/* Check the command line arguments. There should be either one or
three command line arguments */
if (argc!=4) {
   if (argc!=2) {
      printf("please enter 3 parameters besides the command\n");
      exit();
   }
}


/* Check the robot ID, It should be between 1 and 6 */
if ( (Robot_ID<1) || ( Robot_ID>6) ) {
   printf("Robot ID must be between 1 and 6 ");
   exit();
}


/* Connect to Nserver.*/
SERV_TCP_PORT=7771 ;
strcpy(ROBOT_MACHINE_NAME, "nomad");
connect_robot(Robot_ID);


/* If the initial x-y coordinates are entered, store them into
variables. */
if (argc==4) {
    X = atoi(argv[2]);
    Y = atoi(argv[3]);
    place_robot(X,Y,0,0);
}

/* 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 it becomes disconnected. */
conf_tm(5);

/* 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. The single integer value passed
controls the time delay between units in multiples of four msec. */
for (i = 0; i < 16; i++)
   order[i] = i;

conf_sn(1,order);

/* Configure the Infrared Sensors */
for (i = 0; i < 16; i++)
   order[i] = i;

conf_ir(1,order);

/* Fortunately, the robot can talk... */
tk("Let's form a circle guys");
```

```c
    /* Main loop. Execute unless a collision occurs. */
    while (!BumperHit)
      {
        GetSensorData();
        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)
{
    /* Variables  */
    int i;
    int panic;
    int tvel, svel;

    double gain_tvel = 0.07;    /* Translational velocity gain */
    double gain_svel = 100.0;   /* Rotational velocity gain    */

    /* Compute the attractive force and the repulsive force in the robot
    coordinates by using function calculate() */

    calculate() ;

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

    /* Set the rotational velocity. If both minimum and maximum sensor
    readings are 255 which is the maximum sonar range, it means there is
    no contact around. Then proceed by turning 5 degrees in each
    iteration, so that robot will be able to cover the area by making a
    big circle. Otherwise set the rotational velocity based on the total
    forces acting on robot.*/

    if(mindist==255 && maxdist==255){
        svel = 50 ;
        tvel = 100 ;
    }
    else {
        if ((F_tol[0]==0)&&(F_tol[1]==0))
            svel = 0 ;
        else if ( F_tol[0]==0 )
            svel = 450 ;
        else
            svel = (int) (gain_svel * sin(atan2(F_tol[1],F_tol[0])));
    }

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

51

```c
    /* limit the translational  and rotational velocities. Maximum allowed
    translational velocity is 24 in/sec, and rotational velocity is 45
    degrees/sec */
    if (abs(tvel) > 240)
       tvel = 240 * sign(tvel);

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

    /* 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);
}


/*****************************/

/* This function reads sensor data and loads them into arrays. */
void GetSensorData (void)
{
    int i, value , same1, same2 ;
    double corrected_IR[16] ;    /*Array to store correlated infrared
                                 readings. */
    double corrected_sonar[16]; /* Array to store the corrected sonar
                                 readings */
    double norm[16] ;
    int min_above, min_below ;

    /* 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 infrared 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];
```

```c
/* Fuse the sonar and IR data, and store the final sensor reading into
the global array fused_range[]. Infrared readings are not reliable
beyond 14 inches. If the correlated value is smaller than 14, then
normalize and fuse the infrared and sonar data. If it is more than
14, then consider the sonar 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 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");
   }

/* Find out the sensor ID number which detects the closest robot */
minreturn = 0;
for (i = 1 ; i < 16 ; i++) {
  if (fused_range[i] < fused_range[minreturn])
      minreturn = i ;
}

/* The distance to the closest robot */
mindist = fused_range[minreturn];

/* Find out the sensor ID number which detects the furthest robot */
maxreturn = minreturn ;
for (i = 0 ; i < 16 ; i++) {
  if((fused_range[i]>=fused_range[maxreturn])&&(fused_range[i]<255.0))
    maxreturn = i;
}

/* The distance to the furthest robot */
maxdist = fused_range[maxreturn];
```

```c
/* ID numbers of sensors that are next to the one which sees the
closest robot */
min_above = minreturn+1 ;
if ( min_above==16)
    min_above=0 ;

min_below = minreturn-1 ;
if ( min_below==-1)
    min_below==15;

/* Check if the neighbor sensors see the same robot. If so, set the
corresponding flag */
same1 = 0 ;
same2 = 0 ;
if ( abs(fused_range[minreturn] - fused_range[min_above]) <=4 )
    same1 = 1 ;

if ( abs(fused_range[minreturn] - fused_range[min_below]) <=4 )
    same2 = 1 ;

/*Find out the sensor ID number which sees the second closest robot*/
minreturn2 = maxreturn ;
for (i = 0 ; i < 16 ; i++) {
  if(i!=minreturn)
      if (fused_range[i] < fused_range[minreturn2]) {
          if( (same1==1) && (i==min_above) )
              printf("min_above == minreturn \n") ;
          else if( (same2==1) && (i==min_below) )
              printf("min_below == minreturn \n") ;
          else
              minreturn2 = i ;
      }
}

/* The distance of the second closest robot */
mindist2 = fused_range[minreturn2] ;
}

/*****************************/

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

/*****************************/

/* This function computes the attractive force as to the destination
point in Robot Coordinate system and the repulsive force as to the
obstacles in robot coordinate system. Finally it calculates the x and y
components of the total forces on the robot again in robot coordinate
system. */
void calculate() {
  double rho_0 = 15.0;  /* Cut-off distance of the repulsive force */
  double scale = 8.0 ;       /* Scaling factor for attractive force */
  double eta = 35000.0 ;     /* Scaling factor for repulsive force */
```

54

```c
double rho_att = 15.0 ;        /* Saturation distance for attractive
                                  force to switch between parabolic-well and
                                  conic-well */
double rho_float ;
double range = rho_0 + (2*8.81) ;
int i ;
double xx1,xx2,yy1,yy2, xxb, yyb ;
double length, distance ,min_max;
double xgoal, ygoal, xM, yM ;
double teta, alpha, Csquare ;
int signal = 0 ;

/* The coordinates of the closest robot */
xx1 = (double)(mindist+17.62)*cos((double)minreturn*0.39) ;
yy1 = (double)(mindist+17.62)*sin((double)minreturn*0.39) ;

/* The coordinates of the second closest robot */
xxb = (double)(mindist2+17.62)*cos((double)minreturn2*0.39) ;
yyb = (double)(mindist2+17.62)*sin((double)minreturn2*0.39) ;

/* The coordinates of the furthest robot */
xx2 = (double)(maxdist+17.62)*cos((double)maxreturn*0.39) ;
yy2 = (double)(maxdist+17.62)*sin((double)maxreturn*0.39) ;

/* The distance between the furthest and the closest robots */
min_max = hypot((yy2-yy1),(xx2-xx1)) ;

/* Check to see if there is only one contact around -Two consecutive
sensors can return two different range values, although they both see
the same robot- If there is only one robot detected, rotate around it.
*/
if ((abs(minreturn-maxreturn)==1) || (abs(minreturn-maxreturn)==15) ||
    (minreturn==maxreturn) ){
     if ( abs(mindist-maxdist)<=4 ) {

            Csquare = (mindist*mindist) + (2*radius*2*radius) -
                      (2*mindist*2*radius*cos(60.0*PI/180.0));

            alpha= acos( (double) ( (mindist*mindist) + Csquare -
                  (2*radius*2*radius) ) / (2.0*mindist*sqrt(Csquare)) );

            xgoal= sqrt(Csquare)*cos((double)(minreturn*0.39-alpha)) ;

            ygoal= sqrt(Csquare)*sin((double)(minreturn*0.39-alpha)) ;
            printf("There is only one contact around \n");

                 if ((ygoal==0)&&(xgoal==0))
                      teta = 0.0 ;
                 else if (xgoal==0)
                      teta = (90.0*PI/180.0) ;
                 else
                      teta  = atan2(fabs(ygoal),fabs(xgoal));

                 signal = 1 ;
     }
}
```

```
/* If there are more than one robot around */
if (!signal) {

    /* Compute the coordinates of the mid-point of the triangle that
    consists of the closest, second closest and the furthest robot at
    the corners. If there are only two robots around, then consider only
    the closest and furthest robot's coordinates. */

    if((minreturn2==maxreturn) || (((abs(minreturn2-maxreturn)==1) ||
        (abs(minreturn2-maxreturn)==15)) && (abs(mindist2-maxdist)==2))){
        xM=(xx1+xx2)/2.0 ;
        yM=(yy1+yy2)/2.0 ;
    }
    else {
        xM = (xx1+xx2+xxb)/3.0 ;
        yM = (yy1+yy2+yyb)/3.0 ;
    }

    /* The distance from robot's current position to the middle point
    between Rc, Rc2 and Rf */
    length = hypot(xM,yM) ;

    /*Compute the coordinates of the point which is r distance away from
    the mid-point, where r is the desired radius of a circled to be
    formed. */
    if ((yM==0)&&(xM==0))
        teta = 0.0 ;
    else if (xM==0)
        teta = (90.0*PI/180.0) ;
    else
        teta  = atan2(fabs(yM),fabs(xM));

    xgoal = sign((int)xM) * (length-radius) * cos(teta) ;
    ygoal = sign((int)yM) * (length-radius) * sin(teta) ;
}
distance = hypot(xgoal,ygoal); /*Distance to the goal configuration*/

/* Compute the attractive force in the robot coordinate system */
if (distance <= rho_att) {
    F_att[0] = scale*xgoal ;
    F_att[1] = scale*ygoal ;
}
else {
    F_att[0] = scale*rho_att*(xgoal/distance) ;
    F_att[1] = scale*rho_att*(ygoal/distance) ;
}

/* Compute the repulsive force in the robot coordinates */
F_rep[0] = 0.0;
F_rep[1] = 0.0;

for (i = 0; i <= 15; i++){
    rho_float = (double) (fused_range[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);
```

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

/* 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];
}
```

APPENDIX C. SOURCE CODE FOR MOVING IN FORMATION ALGORITHM

```
/* This is the source code of the moving in formation algorithm
presented in chapter V.  The ID number of the robot must be passed to
the program as the first command line argument.  The initial formation
should be formed manually. The x and y coordinates of the goal point
must be passed to each robot as the second and third command line
arguments respectively.  The same goal configuration must be passed to
all robots.  One copy of this program must be run for each robot with a
different ID number.  Nserver allows to simulate up to six robots, hence
the robot ID number should be between 1 and 6.  Only three robots are
used in this simulation, as the Nserver does not show 100% reliability
in the implementation of bulletin board communication scheme with more
than three robots. */

#include "Nclient.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define PI 3.1415926
#define TRUE 1
#define FALSE 0

/*** Function Prototypes ***/

void GetSensorData(void);
void Move_Leader(void) ;
void Move_Follower(void);
int  sign(int);

/*** Global Variables ***/

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 old_rob_config[2] ;
long x_coord[7], y_coord[7] ;
double xgoal_world, ygoal_world, m, d ;
double Dold_goal[2] ;
int Robot_ID, leader_ID, signal, stuck, count, count2, first ;
int left, minreturn  ;
int robots[6] ;
double old_dist ;

/*** Main Program ***/

main (unsigned int argc, char* argv[])
{
   int j,i,leader,svel ;
   int order[16];
   double dist_array[7] ;
   double x_leader_rob, y_leader_rob, xgoal_rob, ygoal_rob, alpha ;
```

59

```c
double xg_1w,yg_1w,xg_2w,yg_2w ;
double gain_svel = 150.0 ;

/*Store the x-y coordinates of the goal point into global variables*/
Robot_ID = atoi(argv[1]);
xgoal_world = (double) atoi(argv[2]);
ygoal_world = (double) atoi(argv[3]);

/* Check the command line arguments. There should be three command
line argument */
if (argc!=2) {
    printf("please enter one parameters besides the command\n");
    exit();
}

/* Check the robot ID, It should be between 1 and 6 */
if ( (Robot_ID<1) || ( Robot_ID>6) ) {
  printf("Robot ID must be between 1 and 6 ");
  exit();
}

/* Connect to Nserver.*/
SERV_TCP_PORT=7771 ;
strcpy(ROBOT_MACHINE_NAME, "nomad");
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 it becomes disconnected. */
conf_tm(5);

/* 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 (j = 0; j < 16; j++)
  order[j] = j;

conf_sn(1,order);

/* Configure the Infrared Sensors */
for (j = 0; j < 16; j++)
  order[j] = j;

conf_ir(1,order);

GetSensorData();

/* Find the distance of each robot from the goal point, and put them
in dist_array[] */
```

```c
for(i=0 ; i<signal ; i++)
    dist_array[i] = hypot((xgoal_world - (double) x_coord[robots[i]]),
                          (ygoal_world - (double)y_coord[robots[i]])) ;

/* Find the closest robot to the goal point. If two or more robot's
are exactly the same distance away from the goal, then choose the one
with smallest ID number.  This will be the leader of formation. */

leader = 0 ;
for(i=1 ; i<signal ; i++) {
   if(dist_array[i] < dist_array[leader])
       leader = i ;
}
leader_ID = robots[leader] ;

svel = 10 ;

/* Turn robot's face towards the goal point. The robot faces towards
the goal point if the rotational velocity is less than or equal to 2
degrees/sec. */

while(abs(svel) > 2){
   GetSensorData() ;
   alpha = (double)robot_config[2]*PI/1800.0 ;

   /*Coordinates of the goal point in robot's local coordinate system*/
   xgoal_rob =  xgoal_world*cos(alpha) + ygoal_world*sin(alpha) -
                (double) robot_config[0]*cos(alpha)-
                (double) robot_config[1]*sin(alpha) ;

   ygoal_rob = -xgoal_world*sin(alpha) + ygoal_world* cos(alpha) +
                (double)robot_config[0]*sin(alpha)-
                (double)robot_config[1]*cos(alpha) ;

   svel = (int)(gain_svel*sin(atan2(ygoal_rob,xgoal_rob))) ;

   if(abs(svel)>450)
        svel=450*sign(svel) ;
   vm(0,svel,svel) ;
}

stuck = 0 ;
count = 0 ;
first = 0 ;
left  = 1 ;
old_rob_config[0] = 0 ;
old_rob_config[1] = 0 ;

if(leader_ID==Robot_ID) {   /* If this robot is the leader */
    st() ;
    sleep(2) ;          /* Wait for two seconds for the other robots */

    /* Execute the Move_Leader() function as long as there is no
    collision */
    while(!BumperHit) {
        Move_Leader();
    }
}
```

```
else {                        /* If this robot is not the leader */

    /* Compute the coordinates of the leader in robot's local
    coordinate system */
    x_leader_rob =  x_coord[leader_ID]*cos(alpha) +
                    y_coord[leader_ID]*sin(alpha) -
                    (double)robot_config[0]*cos(alpha) -
                    (double)robot_config[1]*sin(alpha);

    y_leader_rob = -x_coord[leader_ID]*sin(alpha) +
                    y_coord[leader_ID]*cos(alpha)+
                    (double)robot_config[0]*sin(alpha)-
                    (double)robot_config[1]*cos(alpha);

    /* Determine which side is the leader located : left or right. */
    if((sign(x_leader_rob)*sign(y_leader_rob))<0)
    left = -1 ;        /* Leader is on the right side as to the
                        direction of motion in formation*/

    /* Slope of the line that passes through the leader and the robot
    itself in world coordinate system. */
    m = (double)(y_coord[leader_ID] -
        robot_config[1])/(double)(x_coord[leader_ID]-robot_config[0]);

    /* Distance from the leader */
    d = hypot((double)(y_coord[leader_ID] - robot_config[1]),
              (double)(x_coord[leader_ID] - robot_config[0]));

    /* The coordinates of the goal point which is "d" distance away
    from the leader's goal point on the line that has a slope of "m" -
    in world coordinate system */
    xg_1w = xgoal_world + d/sqrt(1.0+m*m) ;
    yg_1w = ygoal_world + m*d/sqrt(1.0+m*m) ;

    xg_2w = xgoal_world - d/sqrt(1.0+m*m) ;
    yg_2w = ygoal_world - m*d/sqrt(1.0+m*m) ;

    /* Chose the closer one between these two points */
    if  ( hypot((xg_1w-robot_config[0]),(yg_1w-robot_config[1])) <
          hypot((xg_2w-robot_config[0]),(yg_2w-robot_config[1])) )
    {
        xgoal_world = xg_1w ;
        ygoal_world = yg_1w ;
    }
    else
    {
        xgoal_world = xg_2w ;
        ygoal_world = yg_2w ;
    }

/* Execute the Move_Follower() function as long as there is no
collision */

while(!BumperHit) {
    Move_Follower();
}
}
```

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

/*****************************/

/* This function determines the movements of the leader. The leader
simply moves towards the given goal location independently, while
avoiding any collision based on the potential field method */

void Move_Leader(void)
{
    int i;
    int tvel, svel;
    double F_att[2], F_rep[2], F_tol[2];
    double gain_tvel = 0.09;    /* Translational velocity gain */
    double gain_svel = 150.0;   /* Rotational velocity gain */

    /*  Compute the attractive force as to the destination point and the
    repulsive force as to the obstacles in Robot Coordinate system .
    Finally calculate the x and y components of the total forces on the
    robot, again in robot coordinate system. */

    double rho_0 = 15.0;   /* Cut-off distance of the repulsive force */
    double scale = 7.0 ;          /* Scaling factor for attractive force */
    double eta = 35000.0 ;        /* Scaling factor for repulsive force */
    double rho_att = 15.0;        /* Saturation distance for attractive
                                   force to switch between parabolic-well and
                                   conic-well */
    double rho_float ;
    double D_goal[2],X_comp1,X_comp2,X_comp3 ;
    double Y_comp1,Y_comp2,Y_comp3 ;
    int dirdiff,panic ;
    double distance,alpha ;
    double xgoal, ygoal ;

    GetSensorData() ;

    /* Set the panic flag if there is any contact within 8 inches on the
    direction of motion */

    panic = FALSE;
    for (i = 12; i <= 15; i++)
        if (fused_range[i] < 8 ) panic = TRUE;
    for (i = 0; i <= 4; i++)
        if (fused_range[i] < 8 ) panic = TRUE;

    if(!stuck) {      /* If you are not stuck */

        /* If the robot moves less than 2 inches two successive times
        although it is away from its goal location, then set the stuck
        flag */
        if (hypot((double)(robot_config[0]old_rob_config[0]),
                  (double)(robot_config[1]-old_rob_config[1]))<= 2.0){

            count++ ;
            if(count==2){
```

63

```
        if(old_dist > 5.0) {
                stuck = 1 ;
                first = 1 ;
        }
        count = 0 ;
    }
}


/* Store the robot's position to remember in the next iteration */
old_rob_config[0] = robot_config[0] ;
old_rob_config[1] = robot_config[1] ;

alpha = (double)robot_config[2]*PI/1800.0 ;

/* Coordinates of the goal point in robot's local coordinate
system */
xgoal =  xgoal_world*cos(alpha) + ygoal_world*sin(alpha) -
         (double)robot_config[0]*cos(alpha)-
         (double)robot_config[1]*sin(alpha);

ygoal = -xgoal_world*sin(alpha) + ygoal_world*cos(alpha) +
         (double)robot_config[0]*sin(alpha)-
         (double)robot_config[1]*cos(alpha) ;

distance = hypot(xgoal,ygoal) ; /* Distance to the goal point */
old_dist = distance ;

/* Compute the attractive force in the robot coordinate system */
if (distance <= rho_att) {
    F_att[0] = scale*xgoal ;
    F_att[1] = scale*ygoal ;
}
else {
    F_att[0] = scale*rho_att*(xgoal/distance) ;
    F_att[1] = scale*rho_att*(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) (fused_range[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);
    }
}
/* compute the total force in the robot coordinate system */
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]);
```

```c
        /* set the rotational velocity */
        if ((F_tol[0]==0)&&(F_tol[1]==0))
            svel = 0 ;
        else if ( F_tol[0]==0 )
            svel = 450 ;
        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) > 240)
          tvel = 240 * sign(tvel);

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

        /* 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. */

        printf("tvel=%d svel=%d \n",tvel,svel);
        vm(tvel,svel,svel);
}
else                /* If you are stuck */
{
    D_goal[0] = (double)(xgoal_world-robot_config[0]); /*x component */
    D_goal[1] = (double)(ygoal_world-robot_config[1]); /*y component */

    distance = hypot(D_goal[0],D_goal[1]) ;

    /* If you entered this block for the first time, meaning that you
    have just realized that you were stuck, make a 90 degree turn to
    left as to the direction of motion, and leave the obstacle on your
    right */

    if ( first ){
        dirdiff=200;
        printf("first\n");

        while(!(abs(dirdiff)<100))
        {
            tvel=0;
            svel=dirdiff/5;
            vm(tvel,svel,svel);
            GetSensorData() ;

            alpha = ((double)robot_config[2])*PI/(1800.0);
            xgoal =  cos(alpha)*D_goal[0] + sin(alpha)*D_goal[1];
            ygoal = -sin(alpha)*D_goal[0] + cos(alpha)*D_goal[1];
            dirdiff = (int)(atan2(ygoal,xgoal)*1800.0/PI) + 1100 ;
        }

        first=0;   /* Reset the first flag */
```

```
      Dold_goal[0] = 0.0;
      Dold_goal[1] = 0.0;
   }


   if (!panic)
      tvel = 10;
   else
      tvel = 0;


   /* If the robot moves towards the final goal point more than 16
   inches for 5 successive times, it means, it turned the corner of
   the obstacle and started to move towards the direction of motion of
   the formation. In this case reset the stuck flag. */
   if ( distance < (hypot(Dold_goal[0],Dold_goal[1])-16.0)) {
      count2++;
      if (count2==5) {
         stuck  = 0 ;
         count2 = 0 ;
      }
   }


   Dold_goal[0]=D_goal[0];
   Dold_goal[1]=D_goal[1];


   /* Watch the empty space on the direction of motion while following
   the edge of the obstacle, and adjust the rotational velocity in
   order to follow the edge and turn around the corner of the
   obstacle. */

   X_comp1 = (double)fused_range[15]  * cos(2.0*PI*15.0/16.0);
   X_comp2 = (double)fused_range[0]   * cos(2.0*PI*0.0 /16.0);
   X_comp3 = (double)fused_range[1]   * cos(2.0*PI*1.0 /16.0);

   if ((X_comp1<15.0)||(X_comp2<15.0)||(X_comp3<15.0)){
      svel = 65*sign(left);
      tvel = 5;
      stuck = 0 ;
      count2 = 0 ;
   }
   else
      svel=0;


/* Also watch out the distance from the obstacle while following
the edges. Try to keep the same distance from the obstacle by
adjusting the rotational velocity */
Y_comp1 = (double)fused_range[14]  * sin(2.0*PI*2.0/16.0);
Y_comp2 = (double)fused_range[13]  * sin(2.0*PI*3.0/16.0);
Y_comp3 = (double)fused_range[12]  * sin(2.0*PI*4.0/16.0);

svel = svel - sign(left)*(int)(Y_comp1 + Y_comp2 + Y_comp3 - 200.0);

if (abs(svel)<100)
    svel=svel;
else
    svel=svel*20/36;

if(stuck==0) {
    tvel = 0 ;
```

```
        svel = 0 ;
   }
   if(panic==1)
        svel = sign(left)*50 ;

   vm(tvel,svel,svel);
}    /* End else */
}


/*************************/

/* This function determines the movements of the follower robots. Each
follower robot determines a new goal location for itself at each
iteration based on the current position of the leader, and moves to this
point while avoiding any collision, based on the potential field method.
*/

void Move_Follower(void)
{
   int I, tvel, svel;
   double F_att[2], F_rep[2], F_tol[2];

   double gain_tvel = 0.1;      /* Translational velocity gain */
   double gain_svel = 150.0;    /* Rotational velocity gain */

   /* Compute the attractive force as to the destination point and the
   repulsive force as to the obstacles in Robot Coordinate system.
   Finally calculate the x and y components of the total forces on the
   robot again in robot coordinate system. */

   double rho_0 = 15.0;        /* Cut-off distance of the repulsive force */
   double scale = 7.0 ;        /* Scaling factor for attractive force */
   double eta = 35000.0 ;      /* Scaling factor for repulsive force */
   double rho_att = 15.0 ;     /* Saturation distance for attractive force
                                  to switch between parabolic-well and
                                  conic-well */
   double rho_float ;
   double D_goal[2],X_comp1,X_comp2,X_comp3 ;
   double Y_comp1,Y_comp2,Y_comp3 ;
   int dirdiff,son1,son2,son3,panic,panic1 ;

   int see_leader = 0 ;
   double distance ;
   double xgoal, ygoal;
   double xg_1,yg_1, xg_2,yg_2 ;
   double xg_1w,yg_1w, xg_2w,yg_2w ;
   double teta ;

   GetSensorData() ;

   /* Set the panic flag if there is any contact within 8 inches on the
   direction of motion */

   panic  = FALSE;
   panic1 = FALSE ;

   for (i = 12; i <= 15; i++){
        if (fused_range[i] < 8 )
```

67

```
                panic = TRUE;
        else if (fused_range[i] < 20)
                panic1 = TRUE ;
}
for (i = 0; i <= 4; i++) {
    if (fused_range[i] < 8 )
            panic = TRUE ;
        else if (fused_range[i] < 20)
            panic1 = TRUE ;
}

if(!stuck) {      /* If you are not stuck */

    /* If the robot moves less than 3 inches two successive times
    although it is away from its goal location, then set the stuck
    flag */

    if (hypot((double)(robot_config[0]-old_rob_config[0]),
            (double)(robot_config[1]-old_rob_config[1]))<=3.0 ){
        count++ ;
        if(count==2){
                if(old_dist > 8.0) {
                    if(fused_range[minreturn]<(rho_0+1.0)){
                        stuck = 1 ;
                        first = 1 ;
                    }
                }
            count = 0 ;
            }
    }

    old_rob_config[0] = robot_config[0] ;
    old_rob_config[1] = robot_config[1] ;

    teta = (double)robot_config[2]*PI/1800.0 ;

    /* Check to see if you can get the position information of the
    leader. Depending on the distance and the location of the leader,
    sometimes it is not possible to reach the position information of
    the leader, because of the implementation of get_rpx() function
    in Nserver */

    for(i=1 ; i<signal ; i++) {
        if(robots[i]==leader_ID)
            see_leader=1;
    }

    if(see_leader) {      /* If you can get the position information of
                            the leader */

        /* The coordinates of the point which is "d" distance away from
        the leader on the line that has a slope of "m" - in world
        coordinate system. */

        xg_1w = x_coord[leader_ID] + d/sqrt(1.0+m*m) ;
        yg_1w = y_coord[leader_ID] + m*d/sqrt(1.0+m*m) ;
```

68

```
xg_2w = x_coord[leader_ID] - d/sqrt(1.0+m*m) ;
yg_2w = y_coord[leader_ID] - m*d/sqrt(1.0+m*m) ;

/* Convert the coordinates of these points into robot's local
coordinate system. */

xg_1 = xg_1w*cos(teta) + yg_1w*sin(teta)-
        (double)robot_config[0]*cos(teta)-
        (double)robot_config[1]*sin(teta) ;

yg_1 = -xg_1w*sin(teta) + yg_1w*cos(teta)+
        (double)robot_config[0]*sin(teta)-
        (double)robot_config[1]*cos(teta) ;

xg_2 =  xg_2w*cos(teta) + yg_2w*sin(teta)-
        (double)robot_config[0]*cos(teta)-
        (double)robot_config[1]*sin(teta) ;

yg_2 = -xg_2w*sin(teta) + yg_2w*cos(teta)+
        (double)robot_config[0]*sin(teta)-
        (double)robot_config[1]*cos(teta) ;

/* Chose the closer one among these two points as the goal
configuration */
if ( hypot(xg_1,yg_1) < hypot(xg_2,yg_2) ) {
    xgoal = xg_1 ;
    ygoal = yg_1 ;
}
else {
    xgoal = xg_2 ;
    ygoal = yg_2 ;
}
}
else {    /* If you cannot get the position information of the
            leader, move towards the final goal location faster
            than the formation speed. */
    if(!panic1)
        gain_tvel = 0.15 ;

    gain_svel = 250 ;

    /* Coordinates of the final goal point in robot's local
    coordinate system */

    xgoal =  xgoal_world*cos(teta) + ygoal_world*sin(teta) -
            (double)robot_config[0]*cos(teta)-
           .(double)robot_config[1]*sin(teta) ;

    ygoal = -xgoal_world*sin(teta) + ygoal_world*cos(teta) +
            (double)robot_config[0]*sin(teta)-
            (double)robot_config[1]*cos(teta) ;
}

distance = hypot(xgoal,ygoal) ;   /* Distance to the goal point */
old_dist = distance ;

/* Compute the attractive force in the robot coordinate system */
if (distance <= rho_att) {
```

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


/*Increase the translational velcity gain to speed up the robot*/
if (see_leader)
    if(distance > (3*rho_0))
        if (!panic1)
            gain_tvel = 0.15 ;

/* Compute the repulsive force in the robot coordinates */
F_rep[0] = 0.0;
F_rep[1] = 0.0;

for (i = 0; i <= 15; i++){
    rho_float = (double) (fused_range[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);
    }
}

/* 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)&&(F_tol[1]==0))
    svel = 0 ;
else if ( F_tol[0]==0 )
    svel = 450 ;
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) > 240)
    tvel = 240 * sign(tvel);

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

/* 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

70

```
          that the turret is always facing the direction of motion. */

          vm(tvel,svel,svel);
}
else {      /* If you are stuck */

   D_goal[0] = (double)(xgoal_world-robot_config[0]); /* x component */
   D_goal[1] = (double)(ygoal_world-robot_config[1]); /* y component */

   distance = hypot(D_goal[0],D_goal[1]) ;

   /* If you entered this block for the first time, meaning that you
   have just realized that you were stuck, make a 90 degree turn
   towards the main body of the formation */

   if ( first ){
        dirdiff=200;

        while(!(abs(dirdiff)<100)) {
             tvel=0;
             svel=dirdiff/5;
             vm(tvel,svel,svel);
             GetSensorData() ;

             teta = ((double)robot_config[2])*PI/(1800.0);
             xgoal =  cos(teta)*D_goal[0] + sin(teta)*D_goal[1];
             ygoal = -sin(teta)*D_goal[0] + cos(teta)*D_goal[1];

             dirdiff = (int)(atan2(ygoal,xgoal)*1800.0/PI) +
                       sign(left)*1100 ;
        }
        first=0;
        Dold_goal[0] = 0.0;
        Dold_goal[1] = 0.0;
   }

   /* Set the translational velocity according to panic flag */
   if (!panic)
        tvel = 20;
   else
        tvel = 0;

   /* If the robot moves towards the final goal point more than 30
   inches for 4 successive times, it means, it turned the corner of the
   obstacle and started to move towards the direction of motion of the
   formation. In this case reset the stuck flag. */

   if ( distance < (hypot(Dold_goal[0],Dold_goal[1])-30.0)) {
        count2++ ;
        if(count2==4) {
            stuck  = 0;
            count2 = 0 ;
        }
   }

   Dold_goal[0]=D_goal[0];
   Dold_goal[1]=D_goal[1];
```

```c
/* Watch the empty space on the direction of motion while following
the edge of the obstacle, and adjust the rotational velocity in
order to follow the edge and turn around the corner of the obstacle.
*/
X_comp1 = (double)fused_range[15] * cos(2.0*PI*15.0/16.0);
X_comp2 = (double)fused_range[0]  * cos(2.0*PI*0.0 /16.0);
X_comp3 = (double)fused_range[1]  * cos(2.0*PI*1.0 /16.0);

if ((X_comp1<15.0)||(X_comp2<15.0)||(X_comp3<15.0)){
    svel = 65*sign(left);
    tvel = 5;
    stuck = 0 ;
    count2 = 0 ;
}
else
    svel=0;

/* Also watch out the distance from the obstacle while following the
edges. Try to keep the same distance from the obstacle by adjusting
the rotational velocity. Sensor numbers are determined as to which
side is the obstacle. */

if (left==1) {
   son1 = 14 ;
   son2 = 13    ;
   son3 = 12 ;
}
else {
   son1 = 2 ;
   son2 = 3 ;
         son3 = 4 ;
}

Y_comp1 = (double)fused_range[son1] * sin(2.0*PI*2.0/16.0);
Y_comp2 = (double)fused_range[son2] * sin(2.0*PI*3.0/16.0);
Y_comp3 = (double)fused_range[son3] * sin(2.0*PI*4.0/16.0);

if(left==1)
   svel = svel - (int)(Y_comp1 + Y_comp2 + Y_comp3 - 90.0);
else
        svel = svel + (int)(Y_comp1 + Y_comp2 + Y_comp3 - 100.0);

if (abs(svel)<100)
        svel=svel;
else
        svel=svel*20/36;

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

vm(tvel,svel,svel);
  }
}

/***************************/
```

72

```c
/* Read in sensor data and load into arrays. */
void GetSensorData ()
{
    int i,index;
    double corrected_IR[16] ; /* Correlate infrared reading to distance */
    long SonarRange[16];       /* Array of sonar readings (inches) */
    long IRRange[16];          /* Array of infrared readings (no units)*/
    double corrected_sonar[16] ;      /* Array to store the corrected
                                         sonar readings */
    double norm[16] ;
    long rob_pos[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 infrared 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];

    /* Fuse the sonar and IR data, and store the final sensor reading into
    the global array fused_range[]. Infrared readings are not reliable
    beyond 14 inches. If the correlated value is smaller than 14, then
    normalize and fuse the infrared and sonar data. If it is more than
    14, then consider the sonar 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];
```

73

```c
        if (fused_range[i] <= 5.0)
            fused_range[i] = 0.0;
    }
    else
       fused_range[i] = corrected_sonar[i];
}
/* Find out the sensor ID number which detects the closest robot */

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

/* Get the position information of the robots around in robot's local
coordinate system */

get_rpx(rob_pos) ;

/* Put the x - y coordinates of each robot in array, into the cell
that corresponds to each robot's ID number */

x_coord[Robot_ID] = robot_config[0] ;
y_coord[Robot_ID] = robot_config[1] ;

for(i=1 ; i<(3*rob_pos[0]) ; i+=3 ) {
   x_coord[rob_pos[i]] = rob_pos[i+1] + robot_config[0] ;
   y_coord[rob_pos[i]] = rob_pos[i+2] + robot_config[1] ;
}

robots[0]=Robot_ID ;
index = 1 ;

for(i=1 ; i<=rob_pos[0] ; i++){
   robots[i]=rob_pos[index] ;
   index+=3 ;
}
signal = i ;
}
```

```
/**********************/

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

LIST OF REFERENCES

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

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

5. Suzuki I. and Yamashita M., "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. Arai T., Asama H., Fukuda T., and Endo I., *Distributed Autonomous Robotic Systems*, Springer Verlag, New York City, NY, 1994.

7. Ueyama T., Fukuda T., Iritani G., and Arai F., "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.

8. Inaba M., Kawauchi Y., and Fukuda T., "A Principle 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.

9. Wang J. and Beni G., "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.

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

11. Wang P., "Navigation Strategies for Multiple Autonomous Robots Moving in Formation," *Journal of Robotic Systems*, vol.8, no.2, 1991, pp.177-195.

12. Chen Qin and Luh J.Y.S., "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.

13. Balch T. and Arkin R.C., "Motor Schema Based Formation Control for Multi-Agent Robot Teams," *First International Conference on Multi-Agent Systems*, San Francisco, CA, June, 12-14, 1995.

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

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

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

17. Tournassoud P., "A Strategy for Obstacle Avoidance and its Application to Multi-robot Systems," *Proceedings of IEEE International Conference on Robotics and Automation*, San Francisco, CA, 1986, pp. 1224-1229.

18. Borenstein J. and Koren Y., "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.

19. Parker L., "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.

20. Latombe Jean-Claude, *Robot Motion Planning,* Norwell, MA: Kluver Academic Publisher, 1991.

21. Koren Y. and Borenstein J., "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.

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. of 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, Code EC Department of Electrical 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 93943-5121	4
5.	Professor Murali Tummala Code EC/Tu Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	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 Bsk.ligi Bakanliklar - Ankara 06100 Turkey	1
8.	Deniz Harp Okulu Kutuphanesi Tuzla, Istanbul, 81704 - Turkey	1

9.   Golcuk Tersanesi K.ligi   1
      Golcuk, 41650
      Kocaeli - Turkey

10.   LTjg. Gokhan Alptekin   2
      Ayvali Cad. Murat Apt.
      No: 9/3 Etlik, 06010
      Ankara, Turkey