2-1-2007

# On the applicability of random mobility models for swarm robot movements

Siddharth Sail

# On the applicability of Random Mobility Models for Swarm Robot Movements

by

Siddharth Subhash Sail

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Supervised by

Assistant Professor Dr. Shanchieh J. Yang
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
February 2007

**Approved By:**

---

Dr. Shanchieh J. Yang
Assistant Professor
Primary Adviser

---

Dr. Nirmala Shenoy
Assistant Professor - Department of Network System Security & Administration

---

Dr. Dhireesha Kudithipudi
Assistant Professor - Department of Computer Engineering

# Thesis Release Permission Form

Rochester Institute of Technology

Kate Gleason College of Engineering

Title: On the applicability of Random Mobility Models for Swarm Robot Movements

I, Siddharth Subhash Sail, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Siddharth Subhash Sail

Date

# Dedication

I wish to thank my parents, Subhash D. Sail and Sheela Sail. Inspite of being away from home, they supported me, taught me, and loved me. To them I dedicate this thesis.

# Acknowledgments

It is a pleasure to thank the many people who made this thesis possible. It is difficult to overstate my gratitude to my adviser, Dr. Shanchieh J. Yang. With his efforts to explain things clearly and simply, he helped to make my work fun for me. Throughout my thesis-writing period, he provided encouragement, sound advice, good teaching, good company, and lots of good ideas.

I would also like to thank my secondary advisers Dr. Nirmala Shenoy, Dr. Pratapa V. Reddy and Dr. Dhireesha Kudithipudi for their invaluable suggestions and constructive criticism.

I am indebted to the many people working on related topics for their evaluations and suggestions of my work. They deserve a special mention indeed.

# Abstract

Random mobility models have been traditionally used to represent cellular or ad hoc movement patterns for PDA and laptop users. These models are critical for network traffic and protocol analysis. As robotics move to a paradigm where wireless network protocols are being utilized, it is unclear whether the same mobility models are applicable for analyzing these wireless network protocols. This thesis examines the similarity and differences between the traditional random mobility models and the mobility patterns exhibited by a team of randomly moving robots. Though the movements are driven by the same random functions, the robots need to detect and avoid collisions. Results obtained over different scenarios help establish a better understanding of how to develop mobility models representative of robot movements.

# Contents

# List of Figures

# List of Tables

# Glossary

## G

**GloMoSim** Global Mobile System Simulator, p. 1.

## N

**ns-2** Netowork Simulator - 2, p. 1.

# Chapter 1

# Introduction

For more than a decade, much attention has been focused on the development and evaluation of protocols for cellular and wireless ad hoc networks with mobile devices [1]. Most of this evaluation has been performed with the aid of network simulators such as ns-2 [2], GloMoSim [3] and others, assuming random or arbitrary mobility patterns.

Random mobility models have been investigated and analyzed to determine their validity and impact on network protocol performance. In cellular networks, for example, a users mobility behavior directly affects the signaling traffic needed for hand-over, location management and channel holding time [4]. Many of the mobility models used in simulation of cellular networks are also used in the simulation of ad hoc networks. Yoon, Liu and Noble [5] suggest that the average speed of the nodes, in the case of the Random Waypoint model will diminish gradually over time. It has been a consensus that the modeling of a users movement is an essential building block in analytical and simulation based studies of networked mobile systems.

An emerging networked system is the use of autonomous and cooperative robots. Research on robotics has been primarily focused on enabling self-capable robots [6] and a number of algorithms utilizing these robots to accomplish a common task [7]. These robot teams undertake search and rescue missions [8] and environmental studies [9], especially in wide spread regions. One approach towards enabling cooperation is to instill wireless communication and networking capability on the robots. As evidenced from the work presented for cellular and ad hoc networks in the past decades [10, 11], it will also be essential

to understand and utilize representative mobility models for analyzing wireless networking protocols for autonomous and cooperative robot teams. A starting point of this effort is to examine the applicability of existing random mobility models for autonomous robot movements. Analysis presented in this work aims to enhance the understanding of how we may leverage and modify existing mobility models for the new problems ahead.

The remainder of this thesis is organized as follows. Chapter 2 discusses the classification of different random mobility models. It analyzes the pros and cons of all the models. Chapter 3 helps gain insight into a few swarm navigation algorithms. It talks about the features of each algorithm and the way they help disperse the robots into the different parts of the region. This forms the basis for selection of key robot movement features to be implemented as a part of this thesis. Chapter 4 discusses the simulator design and the conceptual models supported by that simulator. Chapter 5 presents the simulation results. With discussions on the behavior of robots under the presence or absence of collision avoidance and collision detection amongst themselves. The conclusion sums up the work and its implications in Chapter 6.

# Chapter 2

# Random Mobility Models - A taxonomy

Various random mobility models have been proposed for cellular and wireless ad hoc networks. Currently there are two types of mobility models used in the simulation of networks: Traces and Synthetic Models [12]. Traces are mobility patterns that are observed in real life, over a sufficiently long period of time. Traces are representative and convincing when used for testing networking protocols. Sufficiently long traces, however, are hard to obtain, especially for new network environment or applications with diverse behavior. Under these circumstances, synthetic models are typically used as a part of the study for designing wireless network protocols. This chapter summarizes several synthetic mobility models used widely for performance analysis and the design of wireless network protocols. Figure 2.1 shows a few mobility models in use. A brief summary of the most commonly used ones is given below.

The synthetic mobility models can be further classified into memoryless mobility models and memory mobility models. Memory mobility models take into account values gained by the different simulation parameters during a particular iteration and use them to deduce new values for those parameters during the next iteration. On the contrary, memoryless mobility models use random values to populate the parameters during every iteration in the simulation. The Random Walk Mobility Model, Random Waypoint Mobility Model and the Random Direction Mobility Model are all examples of memoryless mobility models while the Gauss Markov Mobility Model is a prime example of the latter category. Before presenting the details about all the above-mentioned models a general outline is provided

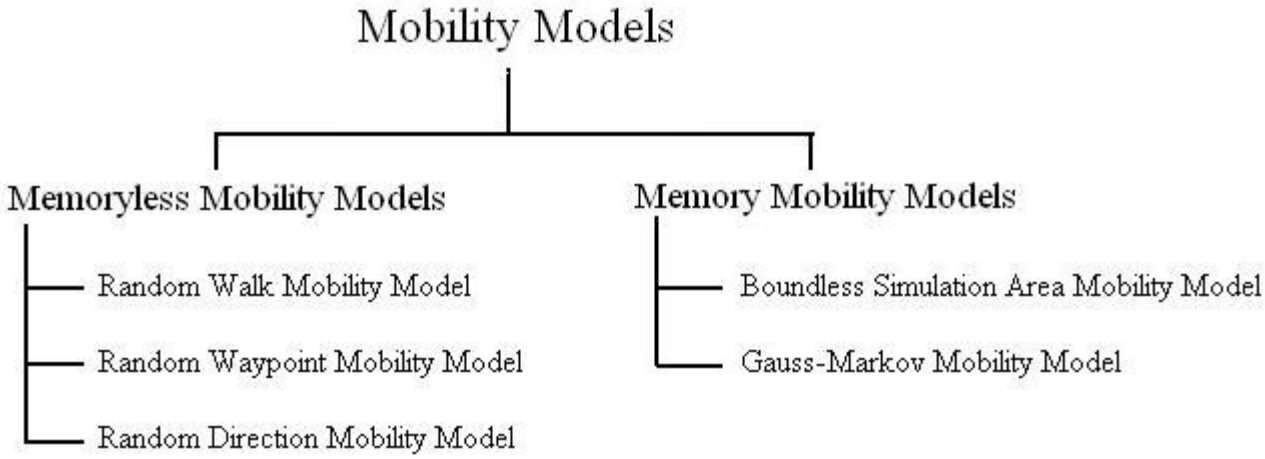



Figure 2.1: A classification of Mobility Models

about the existing environment which is applicable to all the models mentioned in this chapter. The environment under study consists of a group of robots scattered uniformly across a bounded region. Each robot is capable of moving independently of the others within the region. When a robot reaches the boundary of the region it takes measures to stay within the region.

## 2.1 Memoryless Mobility Models

The random mobility models in this category are as the name suggests, truly random in nature. New values for any parameter used in these models are never based on the previous values of the same parameter. A uniform random number generator is used to generate new values for all the parameters used in these models.

### 2.1.1 Random Walk Mobility Model

Random Walk mobility model [12, 13] is one of the basic mobility models in existence and is also famous as the Brownian model [12]. It was first mathematically described [14] by Einstein in 1926. It was developed to mimic erratic movements not governed by the laws

of physics. In the *Random Walk Mobility Model* each node chooses a random speed and direction to move within a bounded region. Given the the minimum speed $V_{min}$ and the maximum speed $V_{max}$, the random speed ($v(t)$) and the random direction ($\theta(t)$) are chosen based on the Uniform distribution $U_{rand}$.

$$v(t) = U_{rand}(V_{min},\ V_{max}) - (1)$$

$$\theta(t) = U_{rand}(0,\ 2\pi) - (2)$$

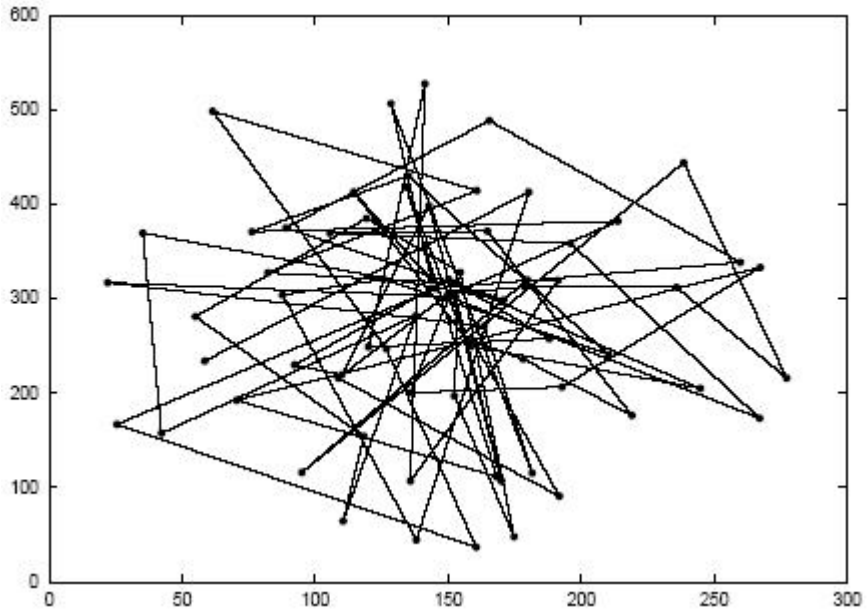



Figure 2.2: Traveling pattern of a mobile node using Random Walk Mobility Model(time) [12]

Each node will choose a new value if one of the following three conditions are met - a pre-selected timer has expired or a pre-selected travel distance has been reached or the node reaches the boundary of the simulated region. In the timer based model, every node assumes a new set of random speed and direction values at the end of a fixed time interval $\tau$. Thus the nodes update their values synchronously. Figure 2.2 [12] shows an example of the movement observed from this time model. The node begins its movement in the center of the 300mx600m simulation area or position (150, 300). At each point, the node

randomly chooses a direction between 0 and $2\pi$ and a speed between 0 and 10 m/s. The node is allowed to travel for 60 seconds before changing direction and speed.

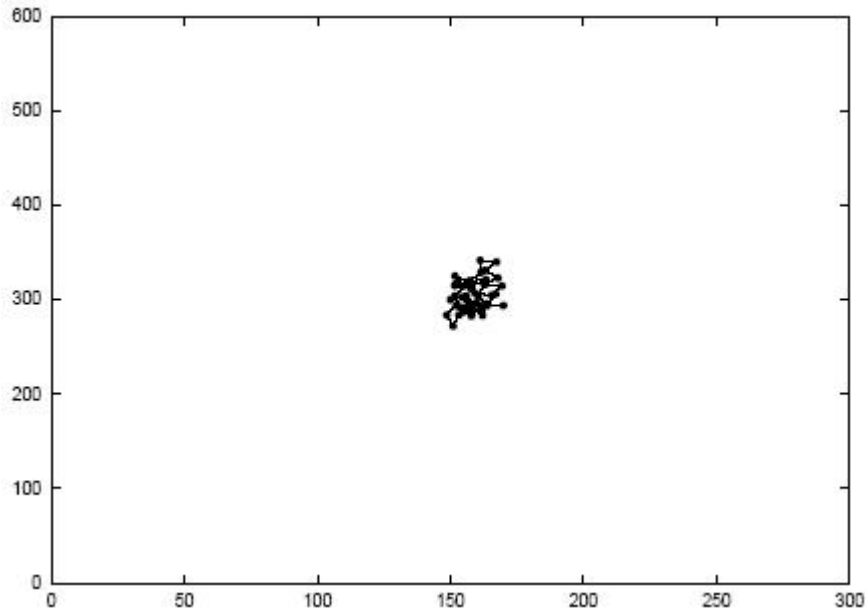



Figure 2.3: Traveling pattern of a mobile node using Random Walk Mobility Model(distance) [12]

In the distance based Random Walk Mobility Model, a node may change direction after traveling a specified distance $\zeta$ instead of a specified time. This variation is illustrated in Figure 2.3. In this example, the node travels for a total of 10 steps (instead of 60 seconds) before changing its direction and speed. Unlike Figure 2.2, each movement of the node in Figure 2.3 travels the exact same distance.

At any instance of time *t* the movement of the node can be represented with the velocity vector ($v(t)cos\theta(t)$, $v(t)sin\theta(t)$). If the node moves according to the above rules and reaches the boundary of simulation field, the leaving node bounces back into the simulation field with the same speed but with a different angle $(\pi - \theta)$, where $\theta \leq \pi/2$ is the adjusted incoming angle. This is referred to as the border effect.

The Random Walk mobility model has its nodes continuously in motion based on the principle of Brownian motion [15] - a physical phenomenon exhibited by minute particles that are immersed in a fluid and move about randomly. Figure 2.4 shows the path traversed

by one such particle.

Figure 2.4: Pattern generated by a particle in suspension

If a number of particles subject to Brownian motion [16] are present in a given medium and there is no preferred direction for the random oscillations, then over a period of time the particles will tend to be spread evenly throughout the medium. Brownian motion makes them attain random oscillations much like those attained by particles immersed in a fluid. A suspended particle is constantly and randomly bombarded from all sides by molecules of the liquid. If the particle is very small, the hits it takes from one side will be stronger than the bumps from other side, causing it to jump. These small random jumps are what make up Brownian motion.

### 2.1.2 Random Waypoint Mobility Model

The Random Waypoint Mobility Model is a widely used mobility model for testing and simulation of wireless network protocols. It was originally proposed in [17] and has since been studied actively especially within the context of wireless ad hoc networks. Broch [18], Johnson [17] and Perkins [19], [20] performed many comparisons of routing protocols using this model. It is an extension of the Random Walk Mobility Model. In that it eliminates the continuous movement inspired by Brownian motion which is a key feature of the Random Walk Mobility Model. It includes a fixed pause time $T_p$ between two successive movements. Note that the use of direction dependent $T_p$ values may be used to model the rotation times needed for the nodes to turn towards a new direction. Typical Random Waypoint mobility model however, uses fixed or random $T_p$s.

The duration of $T_p$ however gives rise to other problems. Yoon, Liu, and Noble [5] have run a number of simulations of the Random Waypoint Mobility Model, varying $T_p$

during each simulation. In their experiments, each node in a large region chooses a random destination and moves there with a random speed chosen from (0 - $V_max$], where $V_{max}$ is the maximum attainable speed by the robots. Intuitively, the model expects to give an average speed of $V_{max}/2$. Such averages should remain the same once the simulation reaches a steady state of the robots, especially during the initial part of the simulation. Unfortunately, Yoon, Lin and Noble found that this is not the case.

An intuitive explanation for this is as follows. The Random Waypoint model chooses a destination and a speed for a node at random. The node keeps moving at that speed until it reaches its destination. Given such a node with a slow speed and a far-away destination, it may take that node a long time to finish its trip. If the nodes do reach the destination they will be assigned another possibly higher random speed, but nodes that do not can be trapped onto the slow journeys for significant amount of time dominating the average nodal speed. As the simulation time goes on, an increasing number of nodes will be trapped onto slower trips resulting in the speed decay observed in Figure 2.5 [5].

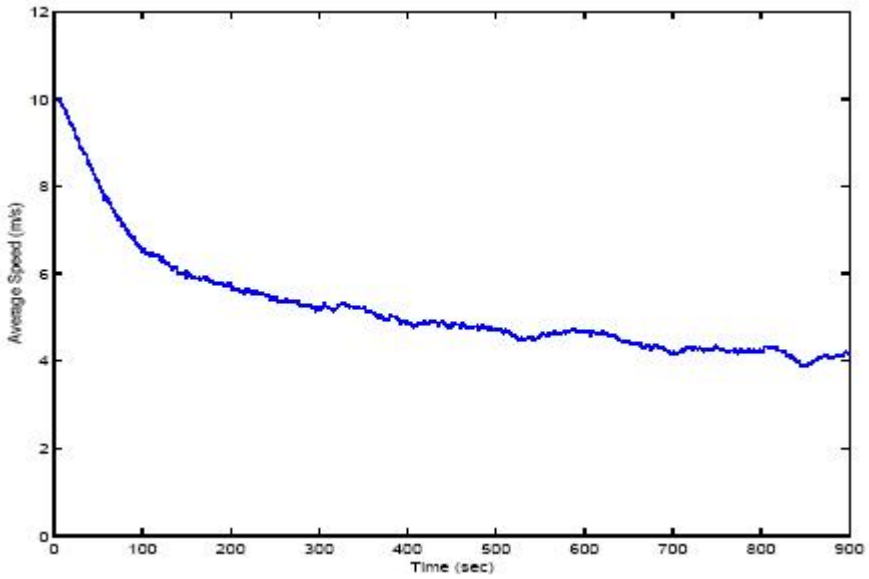



Figure 2.5: Average speed of the robots decreasing over time [5]

Thus the model fails to provide a steady state in terms of average speed. The overheads and performance of mobile systems usually depend strongly on node mobility. In light of

this, random waypoint can generate misleading or incorrect results. In particular, time-average results change drastically over time; the longer the simulation is run, the further the results deviate. This phenomenon is more severe when a longer pause time is used.

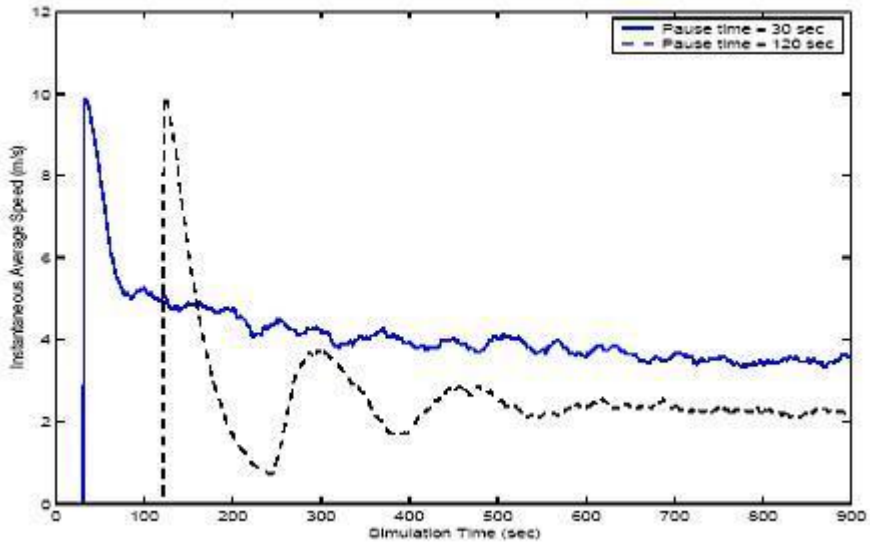



Figure 2.6: Fluctuating average speed of the robots over time [5]

Yoon, Liu and Noble's experiments show that longer pause times lead to more fluctuations in the average speed of the robots, especially during the initial part of the simulation as shown in Figure 2.6.

An intuitive solution to this problem is to follow the Random Waypoint model whereby all node movements follow a fixed time interval. This is known as the synchronized Random Waypoint model. The nodes in this model choose a Uniformly distributed random speed and direction at the same time, *i.e.*, at the commencement of an iteration. All the nodes travel through the region with this velocity only for that iteration. A new velocity is chosen during the next iteration. This guarantees that no node travels for an excessively long time at slow speeds, and maintains a relatively constant average speed for all the robots throughout the runtime of the simulation. It would at most travel with that slow speed for a time equivalent to the duration of the iteration. This helps maintain a relatively constant average speed for all the robots throughout the runtime of the simulation.

### 2.1.3 Random Direction Mobility Model

Both Random Walk and Random Waypoint models suffer from the problem of node clustering [5, 21, 22]. When the simulation is run for longer periods it can be observed that most of the nodes tend to accumulate near the center of the *bounded*region. This forms a cluster at the center with very little nodes near the boundaries as seen in Figure 2.7.

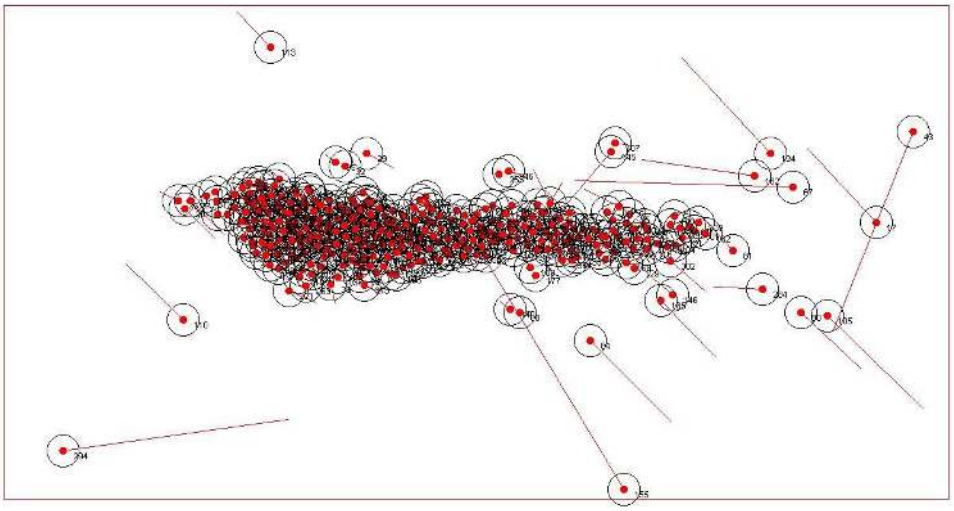


Figure 2.7: Clustering of nodes in the center of the region

This is because the probability of a node choosing a destination in the center of the region or a destination which requires travel through the center of the region is higher than the probability of it choosing to move towards the boundary. In order to alleviate this type of behavior and have the nodes dispersed throughout the region, the Random Direction Mobility Model was developed.

The Random Direction Mobility Model [12, 13] is similar to the Random Waypoint Mobility Model, in that, the robots calculate the speed and direction using the same equations mentioned in the earlier models. But instead of changing the node direction and speed after a constant time interval $\tau$ or after covering the predetermined distance $\zeta$ the nodes are forced to continue traveling with the same speed $v$ and direction $\theta$ until they encounter a boundary. This makes all the nodes travel to a boundary of the region and thus alleviates the problem of node clustering.

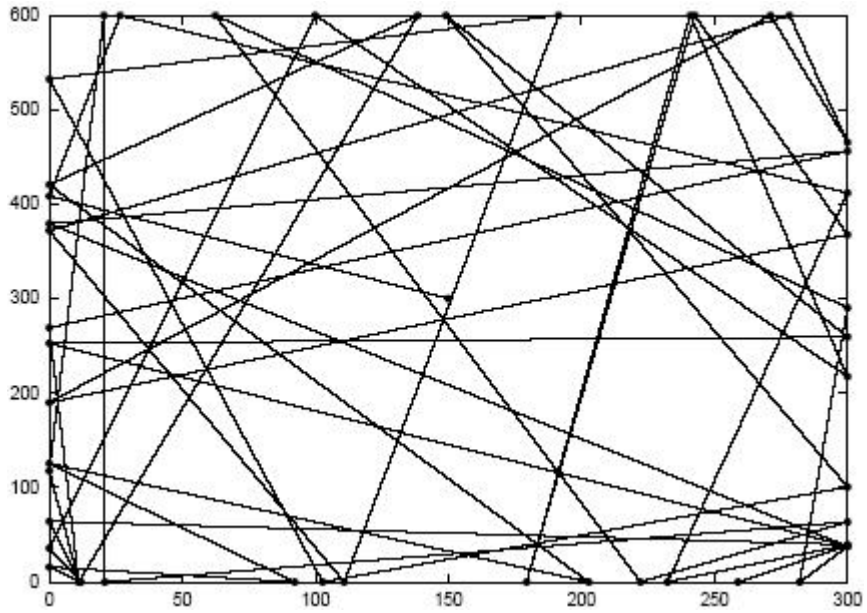



Figure 2.8: Path traversed by nodes following the Random Direction Mobility Model [12]

Figure 2.8 shows an example path of an node, which begins in the center of the simulation area or position (150, 300), using the Random Direction Mobility Model. The dots at the boundary of the figure illustrate when the node has reached a border, paused, and then chosen a new direction.

## 2.2 Memory Mobility Models

Models in this category are characterized by their distinctive feature to utilize memories of the past to determine future movements. Some of the most widely used memory mobility models are summarized below.

### 2.2.1 Boundless Simulation Area Mobility Model

In the Boundless Simulation Area Mobility Model [12], there is a relationship between the direction of travel and the velocity of an mobile node at time $t - 1$ and time $t$. A velocity vector $\vec{v}$ = (v, $\theta$) is used to describe a node's velocity $v$ as well as its direction $\theta$. The node

position is represented as $(x, y)$. Both the velocity vector and the position are updated at every $\Delta$t time steps as per the following equations:

$$v(t + \Delta t) = min[max[v(t) + \Delta v, 0], V_{max}]$$

$$\theta(t + \Delta t) = \theta(t) + \Delta(\theta)$$

$$x(t + \Delta t) = x(t) + v(t) * cos(\theta)t$$

$$y(t + \Delta t) = y(t) + v(t) * sin(\theta)t$$

where $V_{max}$ is the maximum velocity defined in the simulation, $\Delta v$ is the change in velocity uniformly distributed between $[-A_{max} * \Delta t, A_{max} * \Delta t]$, $A_{max}$ being the maximum acceleration of a given node, $\Delta\theta$ is the change in direction which is uniformly distributed between $[-\alpha * \Delta t, \alpha * \Delta t]$ and $\alpha$ being the maximum angular change in the direction of travel for a node.

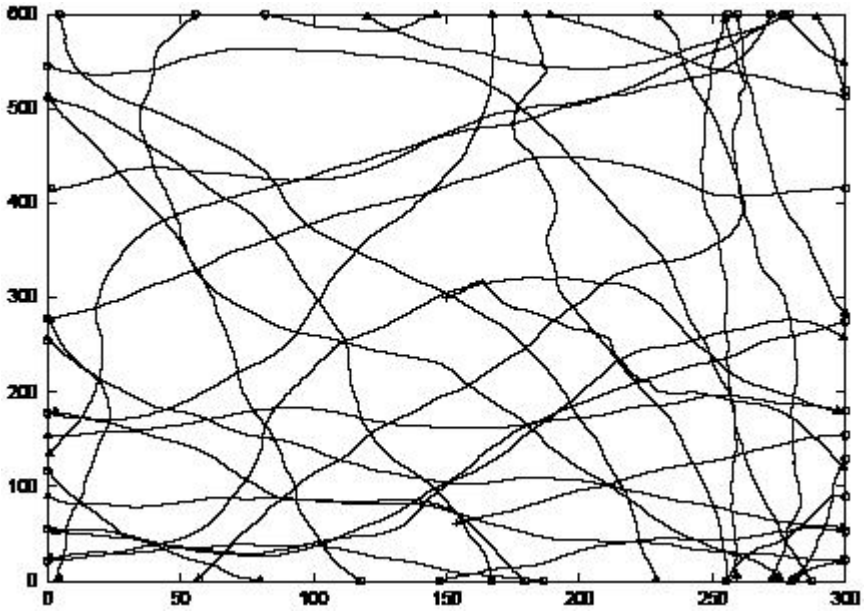



Figure 2.9: Path traversed by a node using the Boundless Simulation Area Mobility Model [12]

Recall that, for the memoryless mobility models, nodes reflect off or stop moving once they reach a simulation boundary. In the case of Boundless Simulation Area Mobility

Model, nodes that reach one side of the simulation area continue traveling and resurface on the opposite side of the simulation area. Figure 2.9 illustrates this concept. This technique creates a torus-shaped simulation area allowing nodes to travel unobstructed. Yet it is unclear whether such assumption is valid in representing real-life mobility patterns.

### 2.2.2 Gauss - Markov Mobility Model

The Gauss-Markov Mobility Model was originally proposed for the simulation of a Personal Communication System [23]. It was later adopted to analyze wireless ad hoc network protocols [24].

The Gauss-Markov Mobility Model [12] was designed to adapt to different levels of randomness via one tuning parameter. Initially each node is assigned a random speed and direction. After regular intervals the speed $v(t)$ and the direction $\theta(t)$ at the $t^{th}$ iteration is calculated based upon the value of speed and direction at the $(t-1)^{th}$ iteration and a random variable $\alpha$ using the following equations:

$$v(t) = \alpha v(t-1) + (1-\alpha)V_{mean} + (1-\alpha^2)\sqrt{\gamma}$$

$$\theta(t) = \alpha\theta(t-1) + (1-\alpha)\theta_{mean} + (1-\alpha^2)\sqrt{\gamma}$$

$V_{mean}$ is the average speed of the robots while $\theta_{mean}$ is the average direction rotated by the robots. $\gamma$ represents the Gaussian Random Variable while $\alpha$ forms tuning parameter. $v(t-1)$ and $\theta(t-1)$ represent the speed and direction respectively of the robot at time $(t-1)$.

Random movements with no memory can be obtained by setting $\alpha$ = 0 and linear motion is obtained by setting $\alpha$ = 1. Intermediate levels of randomness are obtained by varying the value of $\alpha$ between 0 and 1.

To ensure that a node does not remain near the edge of the grid for a long period of time, the nodes are forced away from the edge when they move within a certain distance of the edge. This is done by modifying the mean direction variable $\theta_{mean}$ in the above

direction equation, *e.g.*, when a node is near the right edge of the simulation grid, the value $\theta_{mean}$ is changed to $180°$. Thus, the nodes new direction is away from the right edge of the simulation grid. The values of mean direction for different locations in the simulation grid are shown in Figure 2.10.

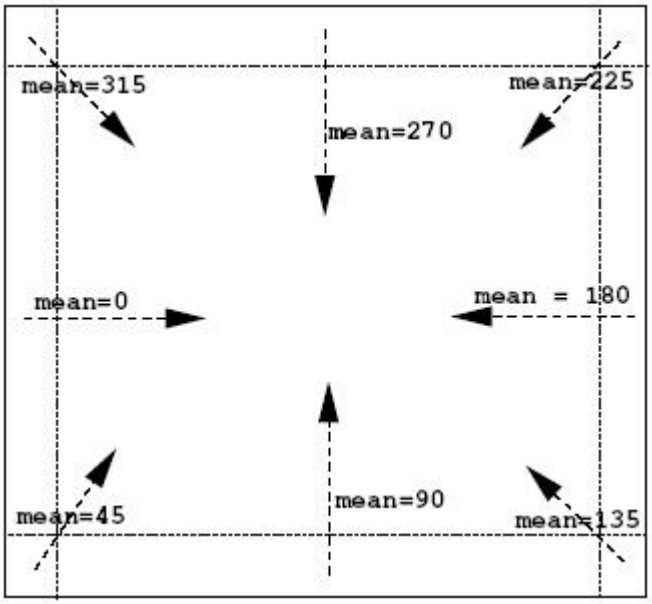



Figure 2.10: The distribution of the mean direction values across a rectangular region [12]

Figure 2.11 illustrates an example traveling pattern of a node using the Gauss-Markov Mobility Model. The node begins its movement in the center of the simulation area or position (150, 300) and moves for 1000 seconds. In Figure 2.11, $t$ is 1 second, $\alpha$ is 0.75, $\gamma$ is chosen from a random Gaussian distribution with mean equal to 0 and standard deviation equal to 1. The value of $V_{mean}$ is fixed at 10 m/s; the value of $\theta_{mean}$ is initially $90°$ but changes over time according to the edge proximity of the node.

As shown in Figure 2.11, the Gauss-Markov Mobility Model can eliminate the sudden stops and sharp turns encountered in the Random Walk Mobility Model (see Section 2.1) by allowing past velocities (and directions) to influence future velocities (and directions).

The above description is how the Gauss-Markov Mobility Model was implemented in [24]. Other implementations of the model also exist - see [25] for example.

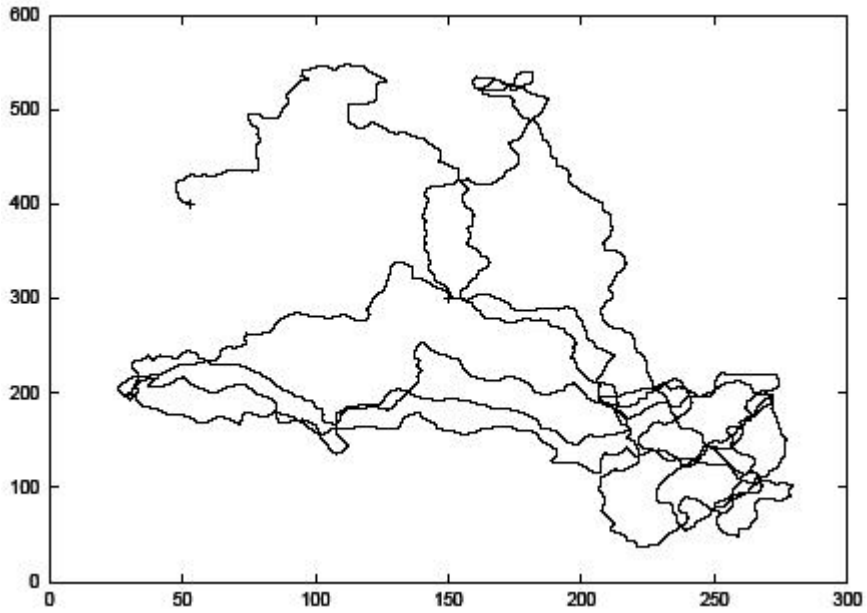



Figure 2.11: Travelling pattern of a node using the Gauss - Markov mobility model [12]

# Chapter 3

# Robot Swarm Algorithms

Exploration, surveillance and security applications all require swarms of robots to disperse throughout their environments [26]. Many autonomous robot team algorithms have been proposed for similar or distinct tasks [27]. This section analyzes a few such algorithms used to disperse a large group of autonomous mobile robots efficiently throughout a bounded environment using local inter-robot sensing or communication. Certainly, moving toward targets or regions will create movement patterns dissimilar to those created by any of the afore-mentioned random mobility models. Because this work aims to shed light on how the random robot movements differ from (or similar to) the random mobility models that were designed for mobile device users, the focus is on the movements exhibited by the robots when they roam randomly in a confined region.

## 3.1 Directed Dispersion

The goal of the Directed Dispersion algorithm [26] is to disperse the robots within the region efficiently and uniformly, all the while keeping them within the communication range of other robots. The dispersion is accomplished by using two algorithms that alternate controlling the swarm movement: Uniform Dispersion and Frontier Guided Dispersion. The former spreads robots evenly, using boundary conditions to limit the dispersion while the latter directs robots towards unexplored areas, and is designed to perform well both in open environments and in environments with obstructions.

### 3.1.1 Uniform Dispersion

The Uniform Dispersion algorithm disperses robots uniformly throughout their environment. Physical walls and a maximum dispersion distance of $r_{safe}$ between the robots are used to help prevent the swarm of robots from wandering away too far thereby fracturing the network and forming multiple disconnected components.

The algorithm works by moving each robot away from the vector sum of the positions $p = p_1,\ p_2,\ .....\ p_c$ of their *c* closest neighbors. The velocity with which robot$_i$ moves away from the rest is given by

$$v = \begin{cases} -\frac{V_{max}}{c \cdot r_{safe}} \sum_{i=0}^{c} p_i & |p_i| \leq r_{safe} \\ 0 & |p_i| \geq r_{safe} \end{cases} \tag{3.1}$$

In the equation above, $V_{max}$ is the maximum allowable velocity for any given robot. $r_{safe}$ is the minimum allowable distance that has to be maintained during and after robot dispersion. $c$ represents the number of neighboring robots while the position of the $i^{th}$ robot is represented by the notation $p_i$.

Thus the swarm of robots disperse themselves across the region always having a neighbor at a distance no more than $r_{safe}$.

### 3.1.2 Frontier Guided Dispersion

The goal of this algorithm is to guide robots towards areas which have not been explored. This coupled with the fact that the swarm is not allowed to disperse into multiple components helps a chain of robots occupy even crevices within the enclosed boundary. This algorithm makes use of multiple *frontier* robots to achieve its objective.

Initially, the *frontier* robots are separated from the non frontier robots. Robots identify themselves as occupying one of three positions in the region: Wall, Frontier, or Interior. "Wall" robots are those that detect an obstacle. An obstacle can be a neighboring robot itself or the boundary wall within which the robots are enclosed. "Frontier" robots are

those that have no neighbors or walls in their view on one or more sides; *i.e.*, they are on the edge of an open space. The remainder are "interior" robots, as illustrated in Figure 3.1. Tight hallways require robots to become frontiers even when they detect walls. Figure 3.1 shows how including the wall in the calculation of unoccupied space can correct this problem.

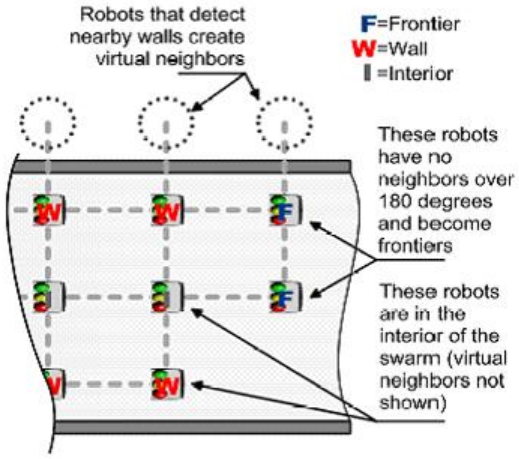



Figure 3.1: Classifying a robot as being "Wall", "Frontier" or "Interior" robot [26]

All robots that detect a wall create a Virtual Neighbor which as the name suggests, is an imaginary robot just outside the wall exactly opposite to the robot creating it. All the non virtual robots compute the largest angle between itself and all its adjacent robots (virtual robots included) within a distance of $r_{safe}$ from that robot. The robots which obtain an angle of 180° or greater as its largest angle between itself and any of its neighbors form *frontier* robots. These robots guide other robots through the region.

The strategy discussed above that classifies robots as *frontier* robots is slightly ambiguous. Consider a circular formation of the swarm. In this case with the boundary wall no where to be seen, all the robots along the circumference of the circle are *frontier* robots. Therefore all these robots will pull the swarm in their own direction leading to immobility or fragmentation. In order to avoid this situation the above rule is revised by stating that robots always move away from the children robots in the frontier tree as shown in Figure

3.2. Also nodes do not move until and unless they have two children. This gives the swarm a direction. Slowly and steadily the children themselves become anchors and in turn guide their children in the direction shown by the *frontier* robots.

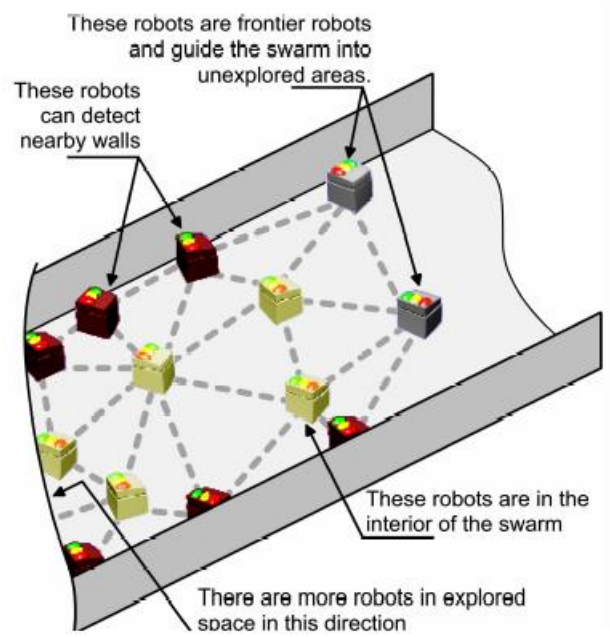



Figure 3.2: Robots moving along a thin passage using the Directed Dispersion algorithm[26]

As explained above the *Directed Dispersion* algorithm makes use of the Uniform Dispersion and Frontier Guided Dispersion alternately to uniformly spread around and at the same time assume a direction to move towards at the abstract level. The algorithm finally terminates when all the robots remain inactive for some predetermined time.

## 3.2 Payton's Gas Expansion and Guided Growth models

There are a number of techniques for coordinating the actions of large numbers of small-scale robots to achieve useful large-scale results in surveillance, reconnaissance, hazard detection, and path finding [28]. The biologically inspired notion of a "virtual pheromone" is implemented to achieve the above mentioned tasks. Unlike the chemical markers used

by insect colonies for communication and coordination, the virtual pheromones are symbolic messages tied to the robots themselves rather than to fixed locations in the environment. This enables the robots collectively to form a distributed computing mesh embedded within the environment. This leads to notions of world-embedded computation and world-embedded displays that provide different ways to think about robot colonies and the types of distributed computations that such colonies might perform.

Virtual pheromones are locally transmitted without specifying a recipient. They possess pheromone diffusion gradients which provide navigation information within the region. These pheromones decay over time which makes any obsolete information void.

The robots make use of the attraction and repulsion forces to help maintain a reasonable distance between itself and each of its neighbors. Obstacles are sensed by the robots when the encoded messages bounce off them. Robots close to each other repel themselves to avoid collisions while at the same time staying within the communication range of the repelling robot. Figure 3.3 illustrates the repulsion and attraction and repulsion zones surrounding each robot.

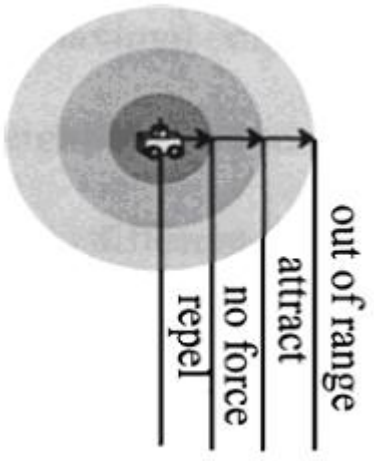



Figure 3.3: Zones surrounding a robot implementing Pheromone robotics [28]

### 3.2.1 Gas Expansion Model

Payton's gas expansion model [28] emulates the way gas particles fill a vacuum. The robots expand from an initial compact state, based on a competition between attraction and repulsion behaviors that depend on distances to obstacles and other robots. This is much like the Uniform Dispersion mentioned above which also spreads robots uniformly throughout the region. The major difference is not in the work they do but the way they do it. Whereas the Uniform Dispersion algorithm uses positional vectors of the neighbors to repel a robot, the Gas Expansion Model makes use of the attraction and repulsion forces between the robots to maintain a reasonable distance between them. These simple behaviors allow a robot swarm to expand from a tight grouping into a maximal dispersion while maintaining nearest-neighbor communications.

### 3.2.2 Guided Growth Model

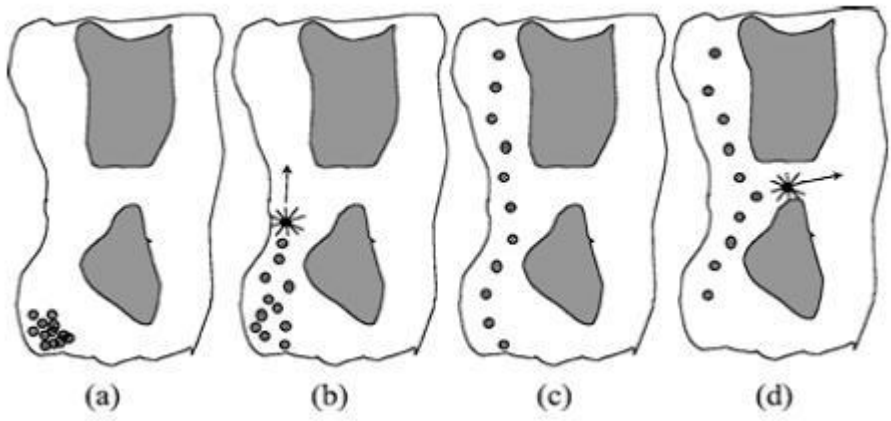



Figure 3.4: A *barrier* acts as a single growth point until it hits a dead end, then a new barrier takes over [28]

The Gas Expansion Model has certain limitations. It works well only when there is a sufficient number of robots to cover the entire region, *i.e.*, the density of the robots in the region is relatively high. If this is not the case then it is impossible for the robot swarm to attain its objective. It is under such circumstances that the Guided Growth Model [28] is

used. This model uses the Gas Expansion Model in conjunction with *barrier* robots to help spread the robots across the entire region even when the density of the robots is low.

A *barrier* robot is the one which emits the "barrier" pheromone. This prevents other robots from coming too close to this robot. This is implemented by making the repulsion force of this *barrier* robot stronger than the repulsion force of the other robots. As the *barrier* robot moves away from the others, they expand to fill the space (Figure 3.4(b)) to maintain communications connectivity. The *barrier* emits a growth inhibitor pheromone, which inhibits the formation of other *barriers*, so that the robots tend to string out in a column that stretches away from the user. When a *barrier* determines it can make no more forward progress, it stops transmitting the growth inhibitor (Figure 3.4(c)). In the absence of growth inhibitor, other robots that detect open space can become *barriers*.

Although the gas expansion model refrains from using any leader - follower strategy it makes use of an attraction force to have robots close to each other stick together. This is a behavior not seen in the case of laptop and PDA users. These mobile device users move randomly without following one another.

## 3.3 SHAPEBUGS

Algorithms for spatial organization of robots have steadily increased in sophistication. SHAPEBUGS [7] is a decentralized algorithm for coordinating a swarm of identically-programmed mobile agents to spatially self-aggregate into arbitrary shapes using only local interactions.

In the SHAPEBUGS algorithm, each agent has a map of the shape to be constructed that is later overlaid on the agent's learned coordinate system. Initially, agents are scattered randomly in the world in a lost state, oblivious of their own positions. When turned on, each agent begins to execute its program using only the data gathered by its proximity sensor and its communication using the wireless link with nearby neighbors.

SHAPEBUGS algorithm consists of two parts namely, trilateration and movement. Trilateration is the process of calculating the positions of a robot based on the positions of its neighbors. This requires the position of at least three robots in the swarm to be known. Once the position of a robot is known it moves into the Movement phase. Since this thesis concentrates mainly on the movement patterns of robot swarms, explanation of the detailed trilateration process is skipped. Interested readers can refer to [7]. A detailed explanation of the movement phase of the SHAPEBUGS algorithm is provided below.

The objective of the swarm of robots in SHAPEBUGS is to form any random shape which is known to them before hand. Initially all the robots are scattered randomly into the region. They then use the trilateration process to calculate their coordinates. At any instant of time, robots who don't know their coordinates are considered lost. These lost robots assume that they are outside the shape and move with a random speed and direction. If the robot knows it position it hopes to move inside the shape using random movements.

Robots who think they are inside the shape have a more complex objective, since they are part of the intelligent swarm that comprises it. First, these agents should not take any steps that will put them outside the shape. This helps keep the shape intact once formed. Second, the swarm should be capable of executing desired tasks such as self-repair and graceful absorption of new agents.

The SHAPEBUGS algorithm achieves these goals by modeling agents in the shape as gas particles in a closed container. Agents react to different densities of neighbors around them, moving away from areas of high density towards low density. Over time, they settle into an equilibrium of constant pressure throughout the shape. When agents die, surrounding agents quickly flood the resulting area of low pressure until equilibrium is restored. Thus, the swarm can respond to any loss as long as there are enough agents left to generate a sensible equilibrium pressure. If new agents are injected into the swarm at any point, the resulting area of high pressure will quickly dissipate. Therefore, many agents entering the swarm at a single location will not be a barrier to subsequent agents. This behavior is inspired by Payton's Gas Expansion Model [28], but is also similar in nature to the flocking

rules proposed by Reynolds [29].

The Random Mobility models discussed in Chapter 3 do not possess the ability to detect and avoid collisions while moving randomly in a bounded region. However robots moving across a field in most of the situations have to avoid collisions. The exact details of this collision avoidance process are mentioned in the upcoming sections. SHAPEBUGS algorithm possesses this ability to avoid collisions between two or more robots. The SHAPEBUGS movement model has two goals namely, equalizing the pressure at any agent density and superimposing the notion of a container within the bounded region.

The first goal is achieved by giving each robot a repulsive force that has a maximum value adjacent to the robot and decays at a constant rate until it reaches a neutral zone. The robot's movement vector, calculated at every time step, is the sum of the vectors away from repulsing neighbors, weighted inversely by distance. This allows robots to disperse evenly at any density.

Let $R$ be the repulsive radius of the robot and $d_i$ be the distance separating itself from robot $i$. Given the robot's own velocity $\vec{v}$ and the velocity of each of the neighbors *i* within its repulsion zone $\vec{v}$, the movement velocity $\vec{m}$ is given by

$$\vec{m} = \sum_{i=1}^{N} \frac{\vec{v_i} - \vec{v}}{d_i(R - d_i)}$$

The second goal is achieved by superimposing a container on the robots coordinate systems. Robot movement is restricted to only steps that keep them inside the container. If the calculated movement vector would take an agent outside the shape, it is discarded in exchange for either staying still or making a random movement within the shape with some small probability. This keeps agents on borders from getting stuck. The work in this thesis will base on these concepts and investigate how movements induced due to collision avoidance may differ from the traditional random mobility models.

# Chapter 4

# Simulator Design and Simulation Models

The following discussion provides an in depth functional analysis of the different components of the simulator. It has been designed and implemented in C# using the Microsoft Visual Studio .NET ®editor and Microsoft Excel ®tool. This is followed by a description of the simulation models capable of being represented by the simulator.

## 4.1 Simulator Architecture

Figure 4.1 shows the simulator architecture. It comprises of the *Graphical User Interface(GUI), Graph Plotter, Robot Movement Calculator* and *Dynamic robot handler*.

The robot parameters which may be updated every time interval are passed onto the *Robot Movement Calculator*. This component parses and processes those parameters subjecting it to repulsion, boundary check and imminent collision avoidance phases. These phases are discussed in greater detail in the next section. These processed robot parameters are then sent to the *Graphical User Interface (GUI)* where they help update the graphical view of the robots scattered all over the bounded region. These processed parameters also make their way into the *Graph Plotter* component where they are transformed into 2D and 3D plots. These plots help the user compare different parameters with one another and study their evolution over the runtime of the simulation. The *Dynamic Robot Handler* helps

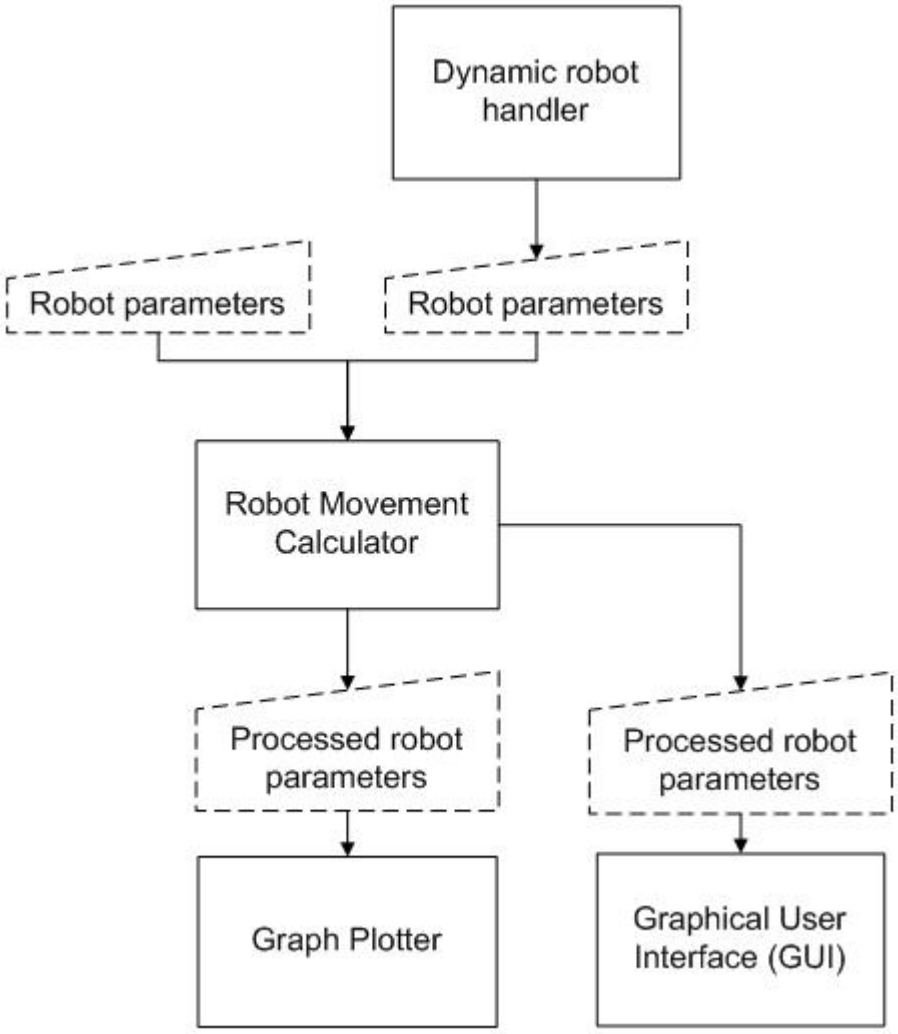



Figure 4.1: Communication between different Simulator components

add or remove robots during any point of the runtime of the simulation. If it adds robots into the simulation, they are passed on to the *Robot Movement Calculator* where they are processed with the existing robots within the simulation.

### 4.1.1 Graphical User Interface (GUI)

The GUI not only gives the user a pictorial view of the motion of the robots but also provides the user the option to interact dynamically with an ongoing simulation. Of all the methods the `OnPaint()` method in the code forms the heart of the GUI. This method

acts as the GUI refresher, updating it on any change in parameters of the robots or the environment surrounding it. It also refreshes the GUI under certain situations when the simulator generates intermediary dialogue boxes to interact with the user for updating certain parameters within the code. The `OnPaint()` method takes care of such cases and refreshes the GUI with the most recently updated values of different parameters within the simulation environment.

### 4.1.2 Graph Plotter

The Graph Plotter plots different robot parameters against one another. It helps the users by providing them with a visual comparison of the statistics collected up to that point in the simulation. It is capable of creating 2D as well as 3D plots. It plots the graphs using Microsoft Office Excel ®. The simulator makes use of the functions in the Microsoft.Office.Core library provided specifically for integrating C# with Excel.

Plots can be requested at regular time intervals (programmed internally) or they can be produced when desired (triggered dynamically) by the user. While using the former technique a check is kept on the time elapsed. As soon as the simulation run time reaches a predetermined mark the graph plotting function is triggered to generate the plots. If the plots have to be generated dynamically, the function is called on the detection of the appropriate key press from the user. This helps the user to create the plots during any specific time slot during the lifetime of the simulation.

### 4.1.3 Robot Movement Calculator

The Robot Movement Calculator is probably the most component of the simulator. It includes all the decision making capabilities of the simulator. A common denominator of the *Robot Movement Calculator* is the capability of collision avoidance. Collision avoidance may be further classified as imminent collision avoidance, which many robots today are capable of, and preventive collision avoidance. Through imminent collision avoidance,

the robots detect collision and stop just before it is about to happen. Preventive collision avoidance may be accomplished by various means. Typical approaches are based on the concepts of artificial repulsion forces between robots [30] or gas expansion models [28]. The *Robot Movement Calculator* consists of various modules to help robots navigate through the region while avoiding imminent collisions and using repulsions to implement preventive collision avoidance.

As shown in Figure 4.2 the components of the *Robot Movement Calculator* can be broadly classified into *Repeller, Boundary Checker* and *Imminent Collision Avoider*.

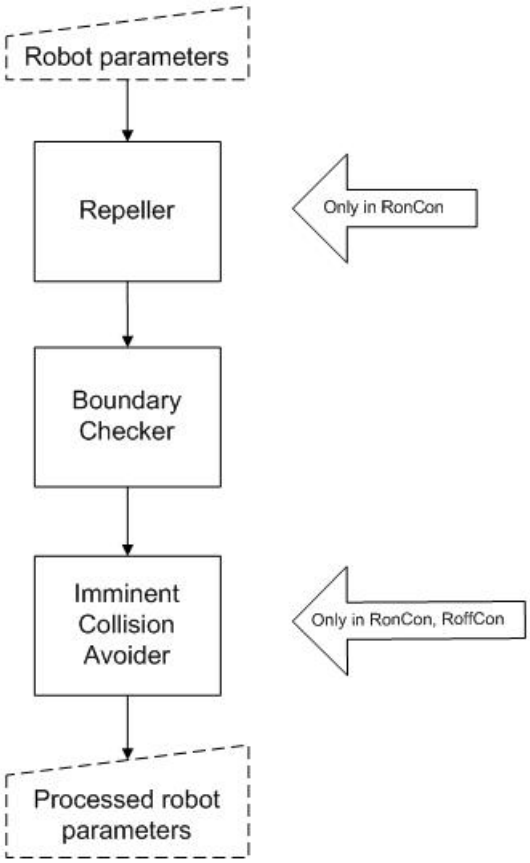



Figure 4.2: Components comprising the Robot Movement Calculator

**Repeller**

The source code for the *Repeller* is included in Appendix A.1. The speed and direction to be attained by a robot at the start of a time iteration depends on the position attained by the robots at the end of the previous time iteration (except during the first iteration of the simulation, in which case, the robots are placed randomly throughout the region using Uniform distribution).

Each robot is circular in shape and has a circular repulsion zone surrounding it. A robot attains random speed and direction if it not present in the repulsion zone of any other robot. In this case the random number generator imparts new speed and direction to the robot. As seen in the Figure 4.3 robots $178, 179$ and $252$ are some of the robots which do not lie in the repulsion zones of any robots other than itself. They would assume a random speed and direction for the ongoing iteration.

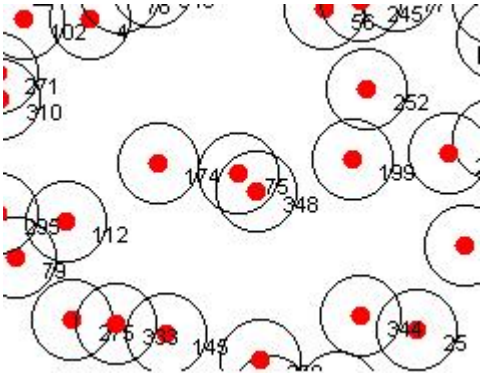



Figure 4.3: Robots being affected by repulsion zones

However this is not true for robots $75$ and $348$. They lie within each others repulsion zones. In such cases the new speed and direction for each of these robots has to be computed instead of being assigned randomly. Let $\vec{f}(i,j,t)$ be the repulsion force induced from $i$ to $j$ at iteration $t$, and $\vec{m}_j(t-1)$ be the movement vector of robot $j$ at time $t-1$, regardless whether it is due to random movement or repulsion forces. The movement vector of $j$ at iteration $t$ is

$$\vec{m}_j(t) \;=\; \vec{m}_j(t-1) \;+\; \sum_{i \in I/j} \vec{f}(i,j,t)$$

where $I$ is the set of all robots, and with $d_{i,j}$ being the distance between $i:\ (x_i, y_i)$ and $j:\ (x_j, y_j)$,

$$|\vec{f}(i,j,t)| = (r_r - d_{i,j}),$$
$$\angle\vec{f}(i,j,t) = \arctan(\frac{y_i - y_j}{x_i - x_j})$$

As it can be observed from the above equation, the new direction after repulsion depends on the direction of travel of the robot to be repelled as well as the directional vector acting on that robot from the repelling robot. In accordance with the Figure 4.3, the repulsion vector for robot $75$ would be calculated on the basis of its own vector and the directional vector acting on $75$ from robot $348$.

### Boundary Checker (Edge Effect)

The source code for the *Boundary Checker* is included in Appendix A.2. The robots in the simulation follow a set of predefined rules to calculate the new vector for every robot in the simulation for a time interval $\tau$. But the simulation is run within a bounded region as shown in Figure 4.4. This may result in the robots potentially going outside the bounded region. A mechanism has to be in place to help the robots stay and move within the region irrespective of the movement vectors they assume during any given iteration.

This ensures that the robots never crossover outside the boundary. The edge detection strategy being implemented by the simulator is described in greater detail in the next section.

### Imminent Collision Avoider

The source code for the *Imminent Collision Avoider* is included in Appendix A.3. Imminent collision avoidance is the process of detecting collisions occurring between any two or more

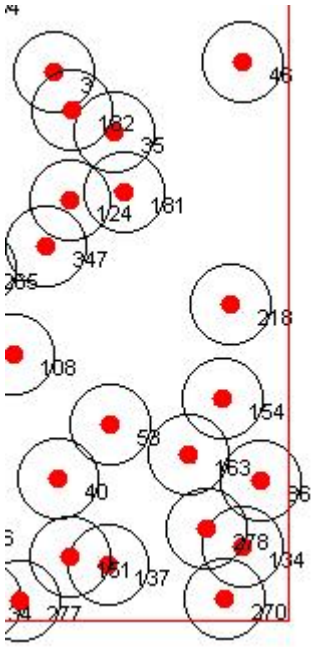



Figure 4.4: Different boundary conditions encountered during a simulation run

sets of robots and then taking evasive actions to prevent it just before its occurrence. The collision detection mechanism used in this simulation is a quadratic equation which guides the robots through the region helping them avoid imminent collisions. Implementation details of the *Imminent Collision Avoider* are deferred to the next section.

### 4.1.4 Dynamic Robot Handler

The *Dynamic Robot Handler* helps the user create hot spots within the simulation environment. It is capable of adding the robots into or removing the robots from a certain constrained section within the rectangular bounded region. It helps analyze the behavior of the robots when they are subjected to sudden changes within their environment. This study helps make improvements within the robot swarm algorithms so that the robots are able to handle such emergency situations better.

At the discretion of the user, a certain number of robots are added into or removed from a circular region creating a hot spot. The details of this procedure are deferred to the next

section. The robots after this environmental change are passed on to the *Robot Movement Calculator* to help them travel through the region as described above.

## 4.2 Simulation Mobility Models

Today, different robot navigation algorithms find themselves stored in robot memories guiding them through an unknown terrain. Some are purely random in nature, others deterministic while algorithms having both these qualities form the third category. The simulator in this category is capable of representing three simulation models. Because the rotation time for the robots in the simulation is not taken into account the first model emulates the Random Walk model and is referred to as the *Random Mobility with no repulsion and no collision avoidance*. However, if the robot rotation time is included within the simulation it will more likely represent the Random Waypoint mobility model with its pause times $T_p$ used to rotate the robots. The second model extends the first one by including an imminent collision avoidance model. The third furthers the extension by including a repulsion force based preventive collision avoidance.

### 4.2.1 Random Mobility with no repulsion and no imminent collision avoidance

The *Random Mobility with no repulsion and no imminent collision avoidance* simulation model closely resembles the Random Walk Mobility model as discussed above. Without any repulsion or collision the robots move around in the region using random speeds and directions, updated every fixed time interval $\tau$

Figure 4.5 shows the working of this simulation model. The simulation starts with the set up of the GUI. This involves setting up a rectangular bounded region and populating it with a fixed number of robots.

Once the environment is set the robots within that environment are imparted random speed and directions. This speed is a random number between $V_{min}$ and $V_{max}$. The direction

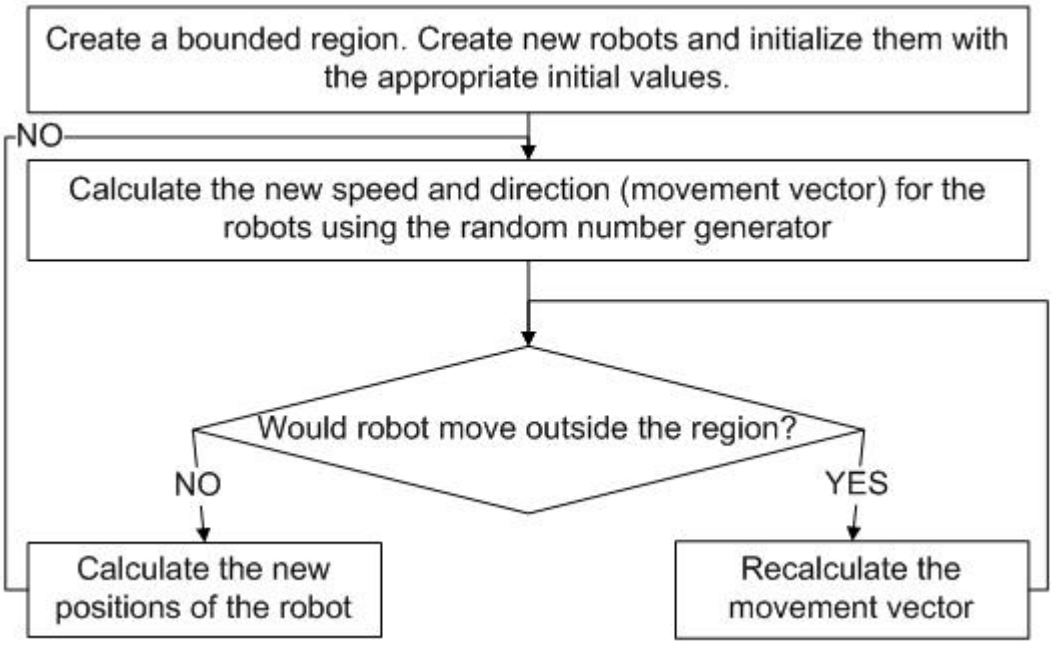



Figure 4.5: Simulation without any repulsion or collision between the robots

is a number between $0°$ and $360°$. The choices of the speed $v(t)$ and the direction $\theta(t)$, and $\tau$ together determines the movement vector $\vec{m}(t)$ for each robot in iteration $t$.

$$\vec{m}(t) \;=\; (v(t) \cdot \cos(\theta(t)) \cdot \tau,\; v(t) \cdot \cos(\theta(t)) \cdot \tau)$$

The simulation models use a different approach for implementing the border effect. If the movement vector of a given iteration will have the robot move outside the pre-defined boundary, a new set of $v(t)$ and $\theta(t)$ will be calculated until the end point of the movement vector ensures that the robot will stay in the confined region.

### 4.2.2 Random Mobility with imminent collision avoidance but no repulsion

As the name suggests, robots in the *Random Mobility with imminent collision avoidance but no repulsion* model possess an imminent collision detection mechanism which they use to avoid running into other robots. In the previous model, robots are modeled as points and

do not occupy a physical space. Once a robot detect collision it stops in its path towards its destination. The point where it stops forms its new destination for that iteration as shown in Figure 4.6.

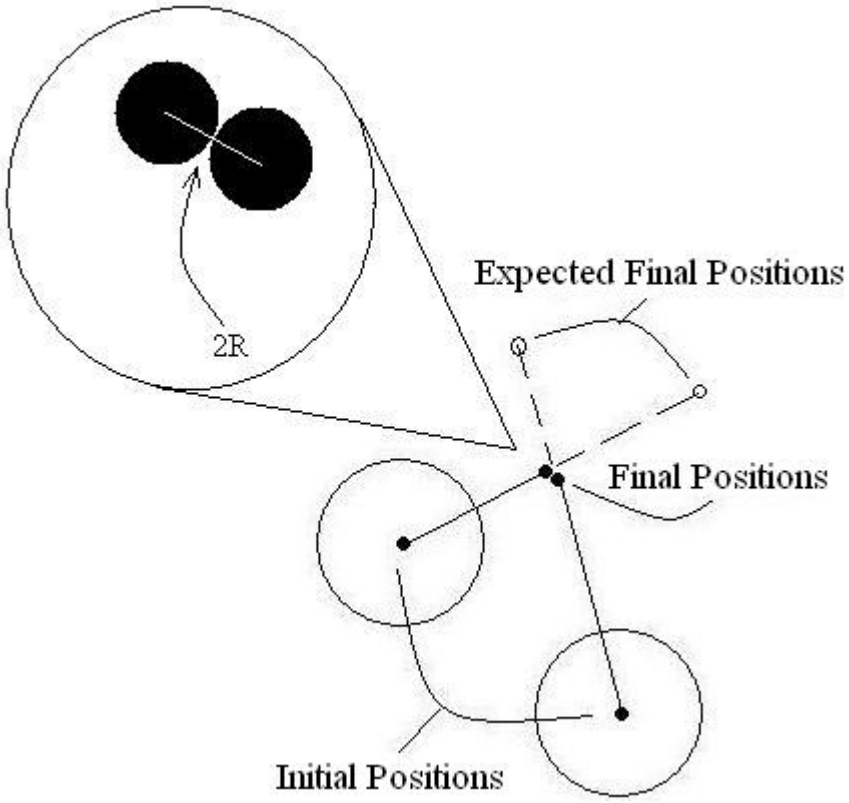



Figure 4.6: Two potentially colliding robots stop in their path to avoid collision

The functioning of this model is quite similar to the previous one with a few changes. As seen in the Figure 4.7, after the setup of the GUI the robots attain a new movement vector with the help of the random number generator. The boundary check phase ensures all the robots stay within the bounded region just as in case of the previous model. If necessary it assigns them new movement vectors to keep them within that region.

However things change after the robots attain new movement vectors guaranteed to keep them within the region. These robots execute an imminent collision avoidance phase. In this, the two robots stop in their path towards their destination as soon as their bodies touch each other as seen in Figure 4.6. At this point they are separated by a distance of

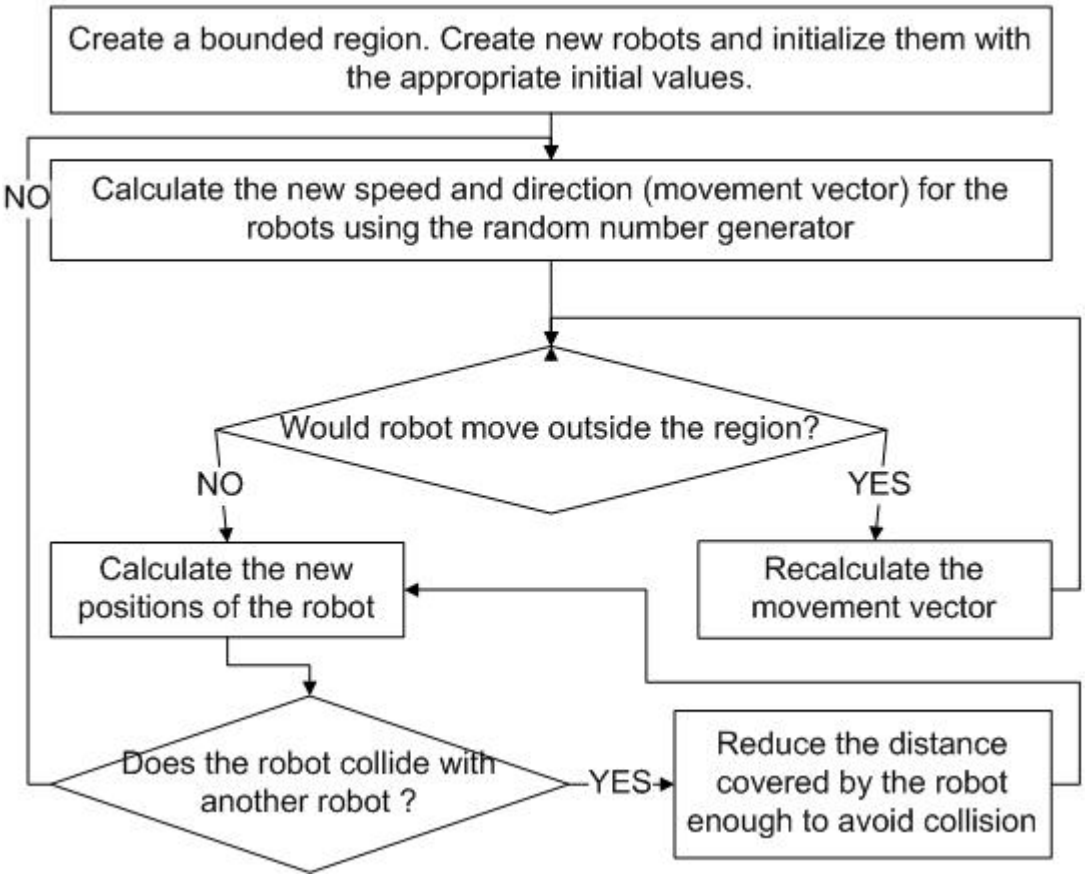



Figure 4.7: Simulation model with collision detection but no repulsion

'$2R$' where '$R$' is the radius of each robot. This new destination is determined by finding a solution to the quadratic equation shown below.

$$A = x_0 + v_0 cos\theta_0 t$$
$$B = y_0 + v_0 sin\theta_0 t$$
$$C = x_1 + v_1 cos\theta_1 t$$
$$D = y_1 + v_1 sin\theta_1 t$$
$$2R = \sqrt{(A-C)^2 + (B-D)^2} - (1)$$

where $(x_0, y_0)$ and $\theta_0$ are the position and direction of travel of robot $0$, $(x_1, y_1)$ and $\theta_1$ are the position and direction of travel of robot $1$, $R$ represents the radius of a robot and $t$ represents the time for which a robots travel.

In the imminent collision avoidance phase the robots are aware of their positions as well as their movement vectors. This information coupled with the knowledge of time for which they travel is used to determine the distance they travel during that iteration. This distance helps them obtain new positions. However, these new destinations are guaranteed to be at their final destinations if and only if at any point of time during the entire iteration any one of those robots does not find an obstruction in its path towards its destination. If it does encounter an obstruction then its path is cut short and this new place of imminent collision becomes the final position for this robot and the entire process repeats.

The solution to a quadratic equation is used to determine whether there is collision amongst any of the robots in the bounded region. If two robots encounter a collision then the solution to equation (1) helps determine and reduce the time for which those robots travel in order to avoid collision. The Left Hand Side (LHS) of equation (1) represents the desired distance to be maintained between the two robots. The Right Hand Side (RHS) of the equation is the existing knowledge about the movement vectors of the two robots and their positions. The solution of this equations produces two results $t'$ and $t''$. If either $t'$ or $t''$ is not between $0$ and the duration of a single iteration $\tau$, the two robots do not collide. However if $t'$ or $t''$ lie between $0$ and $\tau$ there is a potential collision between those two robots and the duration of travel for these two robots is then modified to this value.

As mentioned above, the collision detection process is carried out between all possible robot combinations. The presence of a even a single collision between any of these couples in an iteration results in the entire collision detection process being reiterated amongst all the robots during that iteration. This process is terminated once no collisions are detected between all the robot couples.

### 4.2.3 Random Mobility with repulsion and imminent collision avoidance

The last and the most versatile of all the models is the *Random Mobility with repulsion and imminent collision avoidance*. It aims to represent robots that attempt to avoid imminent

collisions and use artificial repulsion forces to prevent them.

Each robot possesses an imaginary repulsion force for any other robot within the distance of $r_r$. In other words, a circular region centered on a robot with a radius of $r_r$ is considered as the repulsion region of that robot. If a robot is within one or more repulsion regions of other robots, then it will calculate its movement vector based on the repulsion force(s), instead of based on the random movement vector. Let $\vec{f}(i,j,t)$ be the repulsion force induced from $i$ to $j$ at iteration $t$, and $\vec{m}_j(t-1)$ be the movement vector of robot $j$ at time $t-1$, regardless whether it is due to random movement or repulsion forces. The movement vector of $j$ at iteration $t$ is

$$\vec{m}_j(t) \;=\; \vec{m}_j(t-1) \;+\; \sum_{i \in I/j} \vec{f}(i,j,t)$$

where $I$ is the set of all robots, and with $d_{i,j}$ being the distance between $i:\ (x_i, y_i)$ and $j:\ (x_i, y_i)$,

$$|\vec{f}(i,j,t)| = (r_r - d_{i,j}),$$

$$\angle\vec{f}(i,j,t) = \arctan((y_i - y_j)/(x_i - x_j))$$

As seen in 4.8 just as in the previous two models, the simulation for this model starts with the setup of the GUI followed by the robot initialization. Unlike the previous model, however, the new speed and direction are not necessarily calculated using a random number generator. Initially computations are performed to check whether a robot lies in the repulsion region of other robot(s). If it does not then the random number generator assigns a new movement vector to this robot. If it does, the movement vector for this robot is calculated as shown in Figure 4.9

Due to the presence of repulsion between the robots exhibiting the behavior represented by this model the movement vector to be computed for each robot at the start of a new iteration depends on the presence or absence of that robot within another robot's repulsion region.

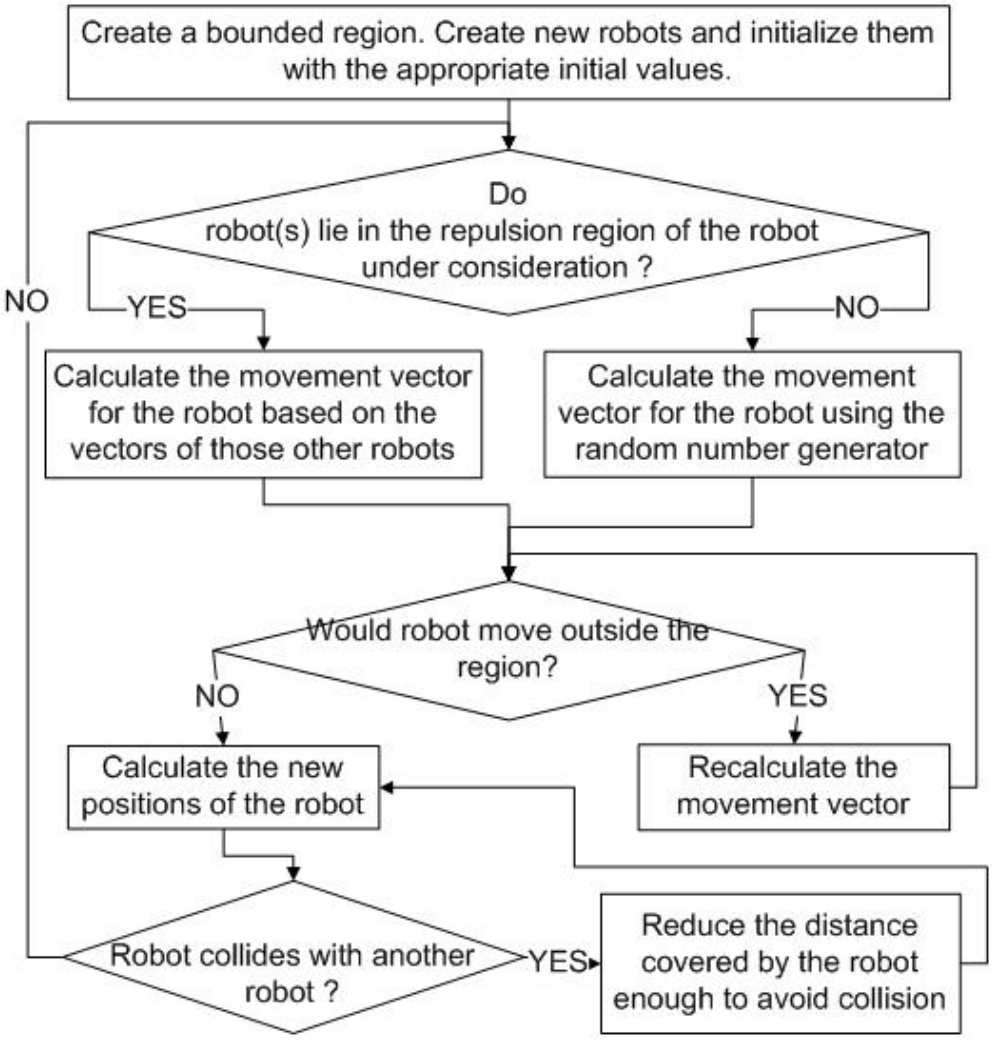



Figure 4.8: Simulation with both collision and repulsion

As shown in Figure 4.9 initially a directional vector is computed pointing from the repelling robot towards the robot to be repelled. The magnitude of this vector depends on the distance between the two robots. Greater the distance greater is the magnitude of the directional vector and vice versa. Once the directional vector is known it is added to the vector of the robot to be repelled. The resulting vector forms the new movement vector for the robot to be repelled. This vector is a derivative of both the distance and position of the two robots as well as the incoming speed and direction of the robot to be repelled.

If this robot lies in the repulsion region of multiple robots then the above process is

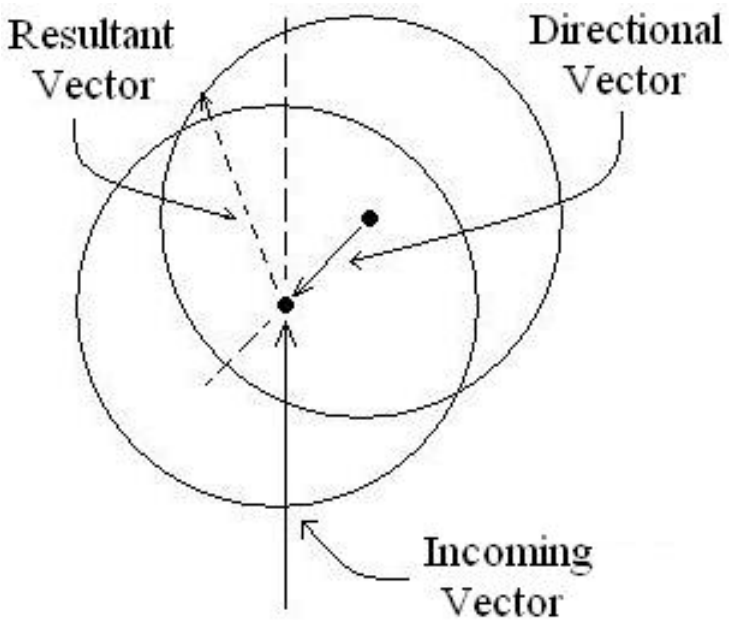



Figure 4.9: Repulsion between two robots

repeated as many times as the number of those multiple robots, each time using the newly calculated resultant movement vector as the new incoming vector for that robot.

Once each robot knows its movement vector for that iteration, it moves over to the boundary check phase. This is similar to the model discussed above. There is a restriction as to where the robots can and cannot move. They have to stay within the bounded rectangular region. So a check is made to see if the new speed and direction calculated by each robot make them move outside the region. If a robot goes outside the region then it is identified as to which boundary or boundaries it can cross. Based on this information a new direction is chosen by the random number generator for this robot. This random direction is a number between limits which ensure that the robot never crosses that boundary or those boundaries. This marks the end of the current time iteration. The robots possess the speed and direction information needed by the simulator during the next time iteration.

The boundary detection phase is followed by the collision detection phase. This too is similar to the one in the *Random Mobility with imminent collision avoidance but no repulsion* model. The completion of the collision detection process marks the end of an

iteration in the *Random Mobility with repulsion and imminent collision avoidance* model. As in the other models, the robots carry over their new speed and direction for calculations to be performed during the next iteration.

### 4.2.4 Dynamic addition and removal of robots

The *Dynamic addition and removal of robots* is a manual event which is used to dynamically add or remove a cluster of robots within a constrained area of the bounded region during the running of the simulation. This is used to study the changes experienced by the existing robots when they encounter this dynamic change. In the simulation, robots are always added into or removed from a circular region, the area of which is controlled by the user. The larger the area of the circle the more the number of robots that can be added or removed from that region.

The addition or removal of robots involves the use of mouse events which help the user in designating a circular area within the bounded region. A *Mouse Down* event records the $(x_1, y_1)$ coordinates of the mouse at that instance. The subsequent *Mouse Up* event again logs the $(x_2, y_2)$ coordinates of the mouse position. If any or both of these points lie outside the region boundary, then the mouse is assumed to be lying on the boundary. Once the coordinates of the mouse are detected a circle is sketched on the GUI possibly encompassing one or more robots in the bounded rectangular region. The diameter of this circle is equivalent to the Euclidean distance between $(x_1, y_1)$ and $(x_2, y_2)$. Once the circle is known the simulation proceeds with addition or removal of robots within the circle.

#### Addition of robots

Before adding the robots into the circular region the simulator determines the maximum possible robots that can be added into the region. This depends on the area of the circular region, the number of robots present in the region and the radius of each robot. Once this number is determined, the user adds a number of robots less than or equivalent to this calculated number. This prevents the simulator from crashing while trying to do an

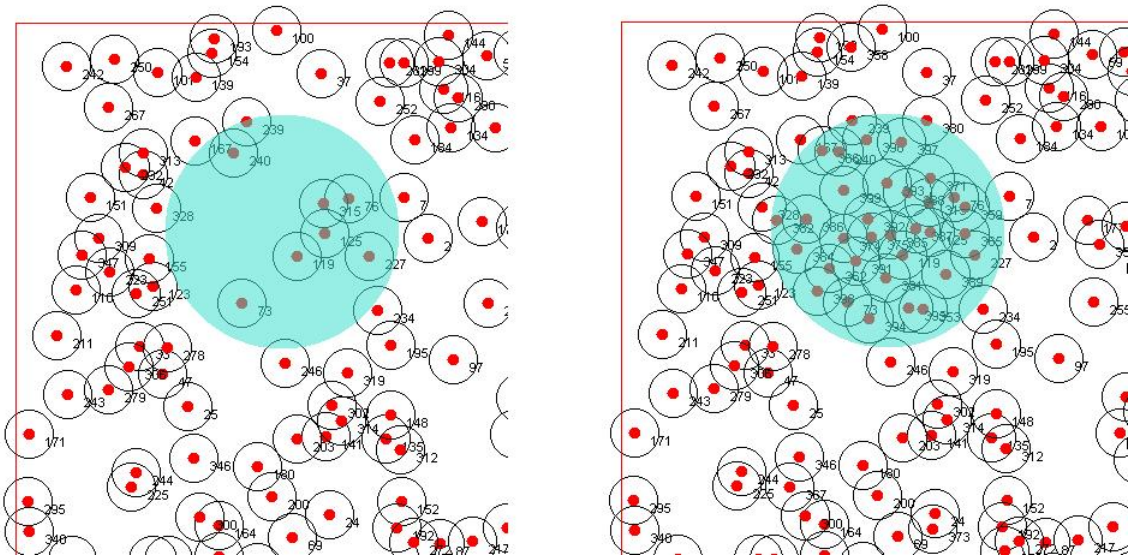


Figure 4.10: Dynamic addition of robots

impossible task of incorporating more robots than the capacity of the circle.

Figure 4.10 has the simulation running on the left and the effect of dynamic addition of the robots on the right. As can be seen the density of robots within the circular region in the figure on the right is more than that on the left.

**Removal of robots**

Removal of robots is an easier task as compared to addition. This is because there is limit and as such no computation necessary while removing a set of robots from a circular region. As can be seen from Figure 4.11 removal of robots creates a void within the bounded rectangular region.

### 4.2.5 Edge Effect (Boundary Conditions)

The robot movement during every time interval is based on the movement vector attained by those robots during that time interval. But the simulation is run within a bounded region. This may result in the robots potentially going outside the bounded region. A mechanism

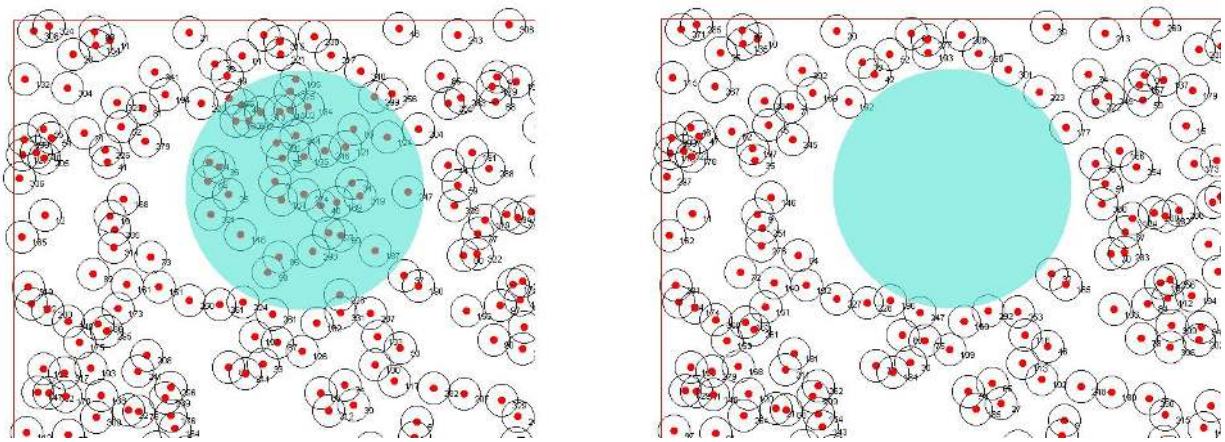


Figure 4.11: Dynamic removal of robots

has to be devised to help the robots stay and move within the region irrespective of the movement vectors they assume during any given iteration.

As seen in the Figure 4.12 two types of boundary conditions can arise during any given simulation run. The first one is the case where a robot can potentially go outside the region via one of the four directions namely, *North, East, South* and *West*. The robot 46 seen in the figure is one such robot. Another case of boundary crossing can occur when a robot poses a threat to go outside the region via either of any two adjacent boundaries. Robots 270 and 134 form examples of such robots.

Initially new positions are calculated for all the robots using the new movement vectors. If at all a robot finds itself outside the region then calculations are performed to find out the boundary which it has crossed and other boundaries which it can potentially cross. Using this information a new direction is randomly chosen for that robot which confines it within that bounded region.

A probabilistic approach can also be used for checking the boundary conditions. In that a robot which can potentially go outside the boundary is "probabilistically" restricted to make a movement during that iteration thus keeping it at the same position as it was at the end of the previous iteration. This effectively keeps it within the region.

The simulator being used uses the former approach.

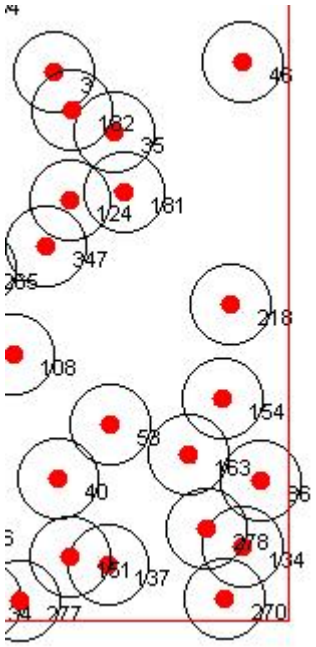



Figure 4.12: Different boundary conditions encountered during a simulation run

# Chapter 5

# Simulation Results and Discussion

The motion patterns of wireless devices used in the day to day life such as PDAs, laptops and cellular phones have been captured by the Random Mobility Models. The way robots navigate throughout the region however, is slightly different. In that they bring into picture two vital aspects unseen in the former case namely, Repulsion and Collision. This chapter analyzes the effects of these differing characteristics on the movement patterns of the simulated swarm algorithm.

The statistics being collected were Average Speed and Average Distance. Variances in these statistics were analyzed while subjecting the robots to changes in their number, repulsion radius, radius while maintaining a constant repulsion radius, radius over increasing repulsion radius, maximum attainable speed and changes in the update interval $\tau$.

## 5.1 Simulation Setup

The three mobility models are developed and tested in a simulated environment. By default, a total of 200 robots are originally uniformly placed in a $1086 \times 514$ cm$^2$ 2D region. Each robot occupies a circular space of radius 5 cm, has a maximum velocity of 10 cm/s, and can detect accurately other robots within a distance of 20 cm from its center. In the case of robots detecting imminent collisions, the colliding robots will stop with their center separated by 10 cm. In the case of robots using a repulsion algorithm to avoid collisions, each robot will have a circular repulsion zone of radius 20 cm. The three models are labeled

with *RoffCoff* for *Random Mobility without collision avoidance and repulsion*, *RoffCon* for *Random Mobility with collision avoidance but no repulsion*, and *RonCon* for *Random Mobility with collision avoidance and repulsion*. Each data point is an average of what the robots perceived over 1000 seconds with 10 independent runs. The standard deviations of the data are insignificant and thus omitted in the plot for clarity. Different factors are varied to exhibit the differences and similarities between the commonly used mobility model *RoffCoff* and the mobility pattern exhibited by randomly moving robots *RoffCon* and *RonCon*. Table 5.1 shows the parameter values used during the running of the simulations.

Table 5.1: Simulation Setup

| Sr.No. | Parameter | Value |
|---|---|---|
| 1 | Number of robots | 200 |
| 2 | Robot radius | 5 cm |
| 3 | Repulsion radius | 20 cm |
| 4 | Maximum attainable speed | 10 cm/sec |
| 5 | Length of a time interval $\tau$ | 1 sec |
| 6 | Area of the region | $1086 \times 514$ cm$^2$ |

## 5.2 Simulation Results

Increasing or decreasing the above mentioned parameters may cause a gradual or dramatic change in the statistics being collected. This section provides an in depth analysis about such changes in these statistics, their causes and their implications.

### 5.2.1 Number of Robots

Figure 5.1 shows the effect of increasing number of robots on the average speed and average distance perceived by the robots with the three simulation models. The average speed is not affected by the number of robots, as long as there is no repulsion. This is observed in the *RoffCoff* model. This behavior is also observed in the *RoffCon* model inspite of the presence of collision detection mechanism present amongst the robots. Since the robot

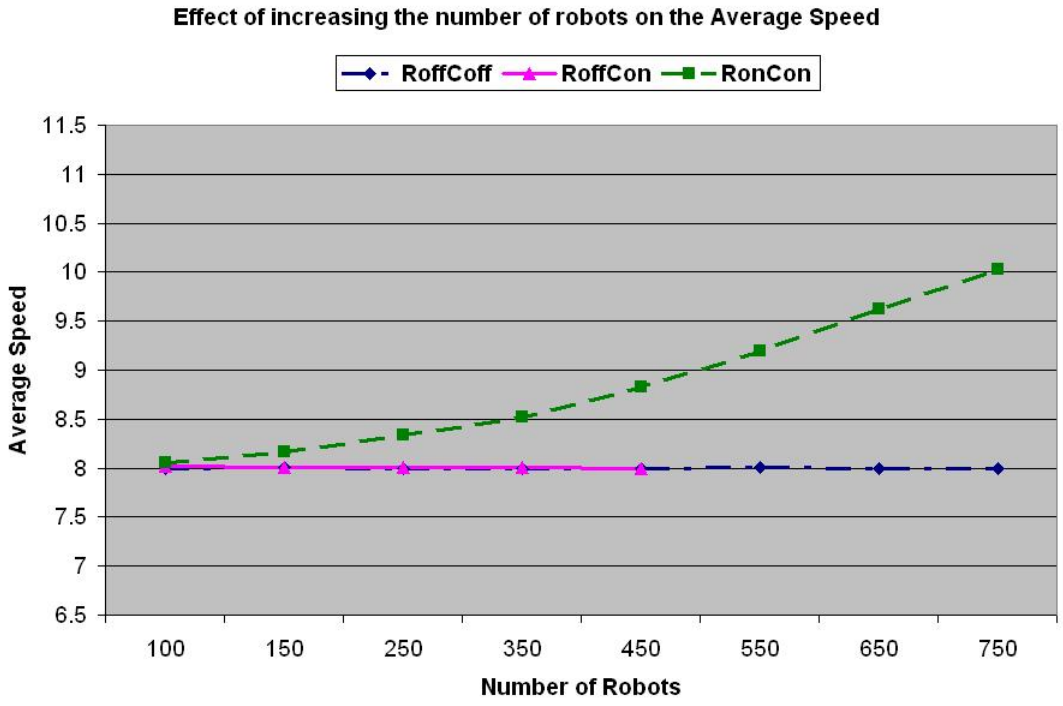
collisions can only cut short on the travel distance (as can be seen from Figure 5.1(b)) but not the speed, the collision detection process does not affect the average speed of the robots. With repulsion (*RonCon*), the average speed increases and the average distance declines as the number of robots increases. The average speed increases because the repulsion force is stronger with more robots. The increase in speed compensates the decrease in travel distance due to collisions, hence the smaller decline in distance for *RonCon* than that for *RoffCon*.

### 5.2.2 Robot Radius (repulsion radius being held constant)

In this set of simulation, the repulsion radii of all the robots is held constant at 50 cm while inducing an increase in the robot radius from 5 cm to 35 cm in steps of 5 cm increments. Figure 5.2(a) shows that the increasing robot radius helps hike the average speed in case of the *RonCon* case until the robot radius reaches 25 cm. After this point though there would just be imminent collisions detections without any repulsions as the repulsion radius of a robot would be smaller than its own radius. This pulls down the *RonCon* curve. Ideally the *RonCon* and the *RoffCon* curves should overlap each other. But due to precision errors within the simulation this is not the case. The increase in robot radius also elevates the chance of collisions, hence the smaller average travel distance as exhibited in Figure 5.2(b). As the radii of the robots increase beyond half the repulsion radius, repulsions between robots cease to exist. Thus the average distance in the cases of *RoffCon* and *RonCon* curves, from this point onwards, show similar behavior.

### 5.2.3 Robot Radius (Varying repulsion radius)

In this set of simulation, the repulsion radii of all the robots is varied with the radii of the robots. Figure 5.3(a) shows that the average speed goes on increaseing for the *RonCon* case due to rise in the probability of repulsions. The average speed is limited only by the maximum allowable limit $V_{max}$. This, however, leads to a rise in the number of collisions



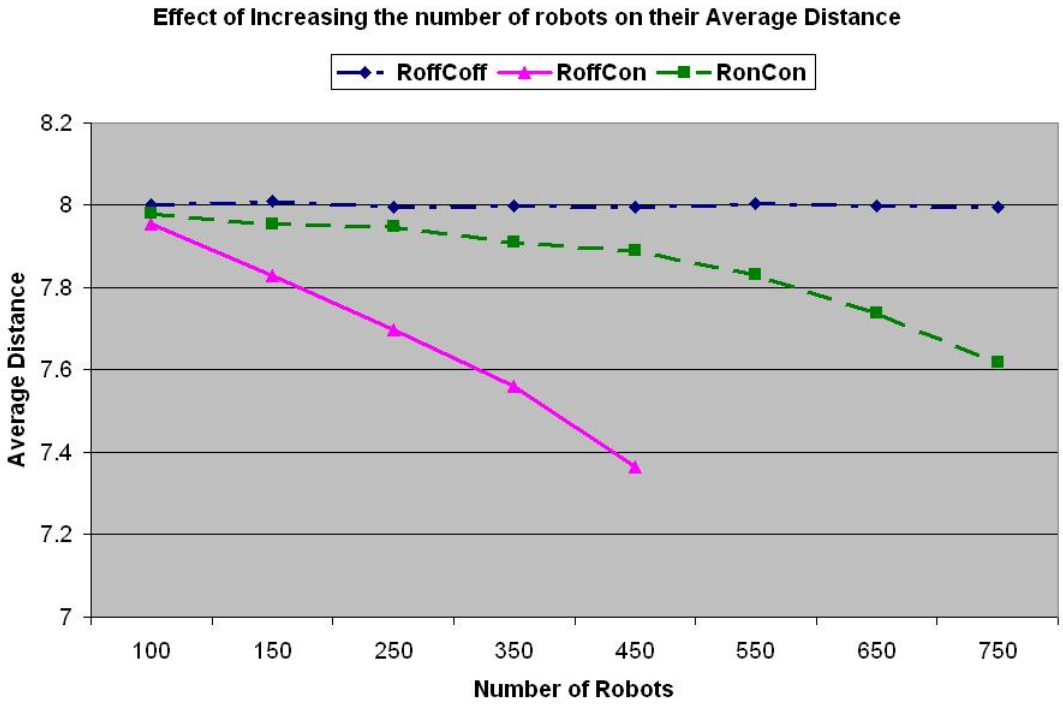
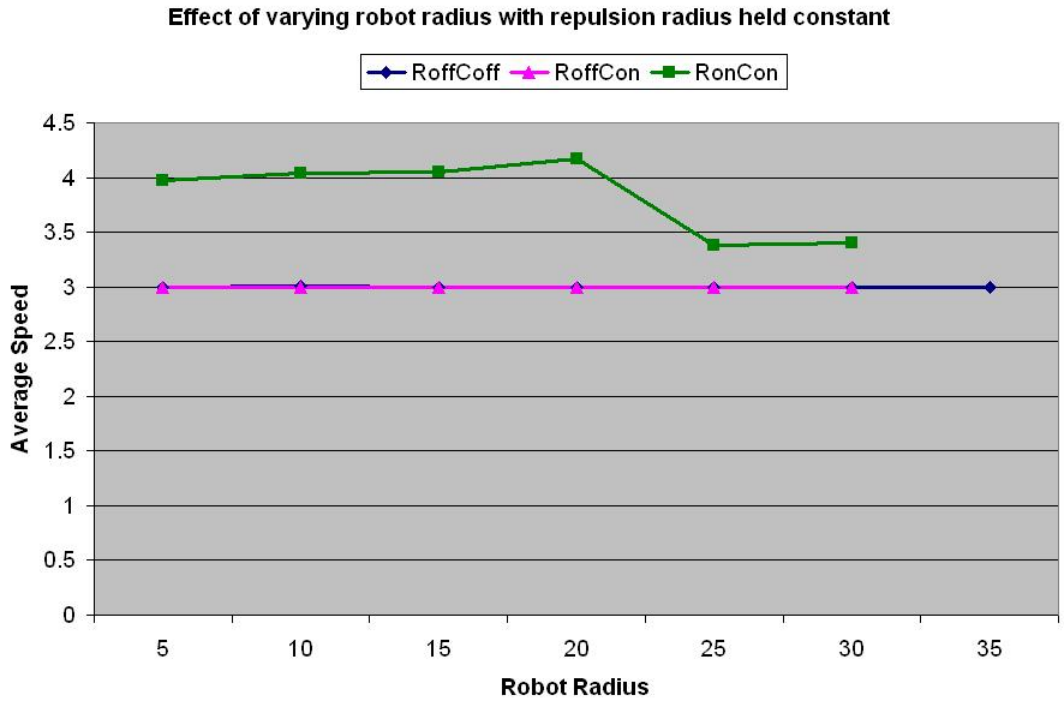




Figure 5.1: The average speed (5.1(a)) and the average distance (5.1(b)) perceived by the robots over 1000 seconds with 10 independent runs as the number of robots increases.



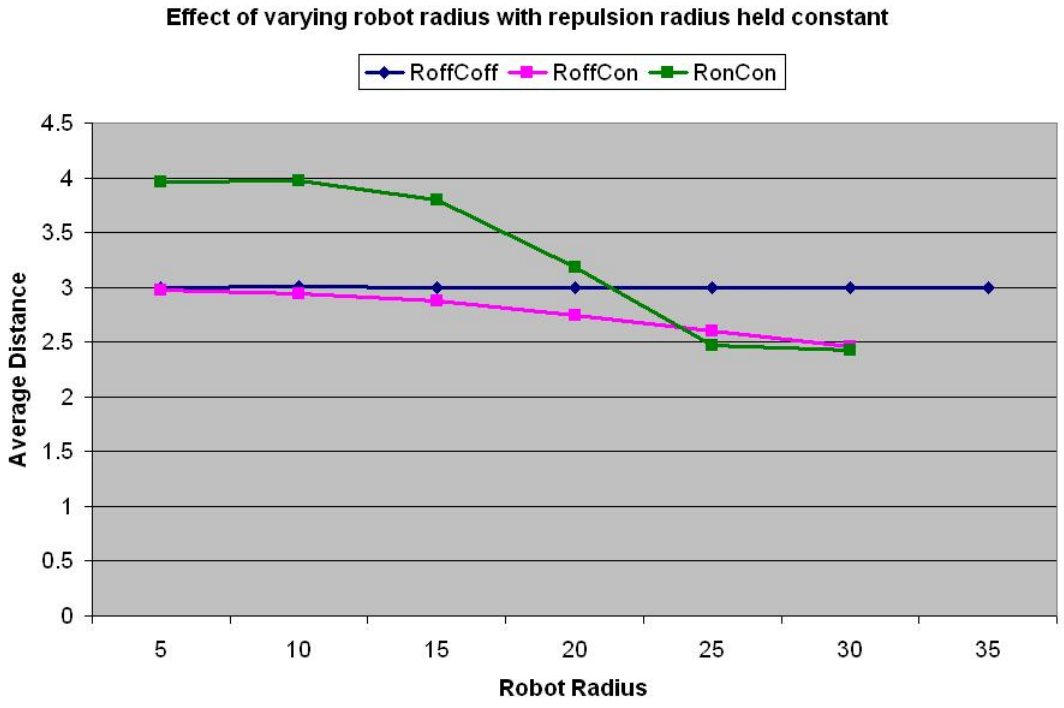




Figure 5.2: The average speed (5.2(a)) and the average distance (5.2(b)) perceived by the robots over 1000 seconds with 10 independent runs as the radius of the robots increases (the repulsion radius = 50 cm).

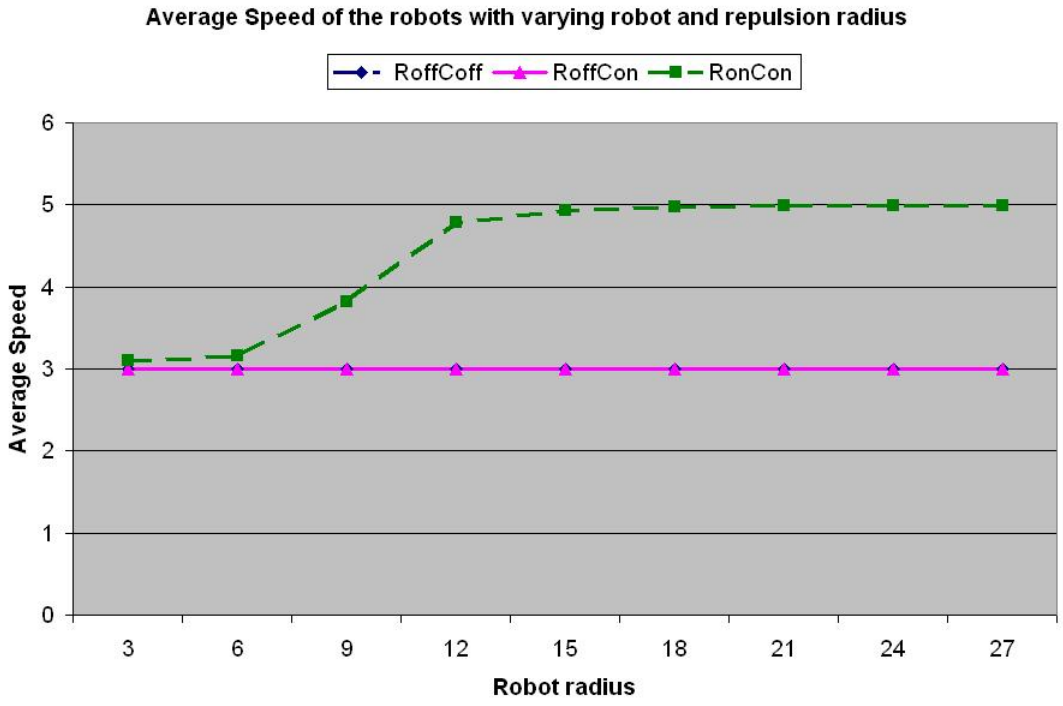
which pulls down the average distance in case of the *RonCon* curve after a certain point in Figure 5.3(b).

### 5.2.4 Robot Maximum Speed

Increasing the maximum robot speed clearly increases linearly the average speed induced by the Uniform random number generation. This is shown by the *RoffCoff* and *RoffCon* curves in Figure 5.4(a), where once again the collision detection alone does not make any difference. With repulsion forces, however, the robots will have a faster linear increase in the average speed. Meanwhile, increasing maximum speed leads to a rise in the number of collisions except for the *RoffCoff* case where robots are modeled as points with no physical size. These collisions reduce the distance traveled by the robots and thus result in the sub linear trend for the cases of *RoffCon* and *RonCon*, as shown in Figure 5.4(b). With repulsion forces, the induced average distance is initially larger than that for the *RoffCon* case because of the higher speed but later on will be restricted severely by the excessive collisions. This is why the *RonCon* curve dives below the *RoffCon* curve as the maximum speed goes beyond 65 cm/sec.

### 5.2.5 Update Interval

In our simulation model, the robots synchronously derive their movement vectors every $\tau$ interval. As $\tau$ increases, the travel distance per iteration increases, but could lead to more collisions and higher repulsions forces. As expected, Figure 5.5(b) shows an increasing trend in case of the *RoffCoff* curve in terms of the average distance traveled. For *RoffCon* and *RoffCoff*, however, the increases taper off because the excessive collisions detected. In other words, the robots will not be able to travel much farther even with a longer update interval. The average speed, on the other hand, does not change for the *RoffCoff* and *RoffCon* as shown in Figure 5.5(a). This is quite intuitive though as that the collision does not change the uniformly derived random speeds. For *RonCon*, longer $\tau$ leads to robots



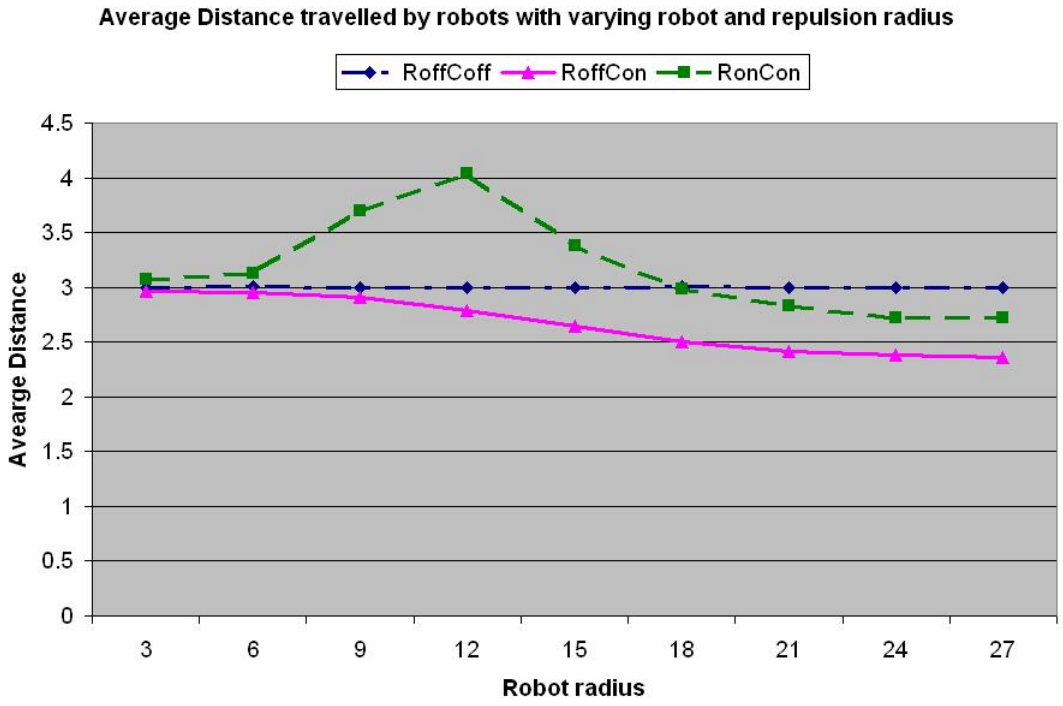
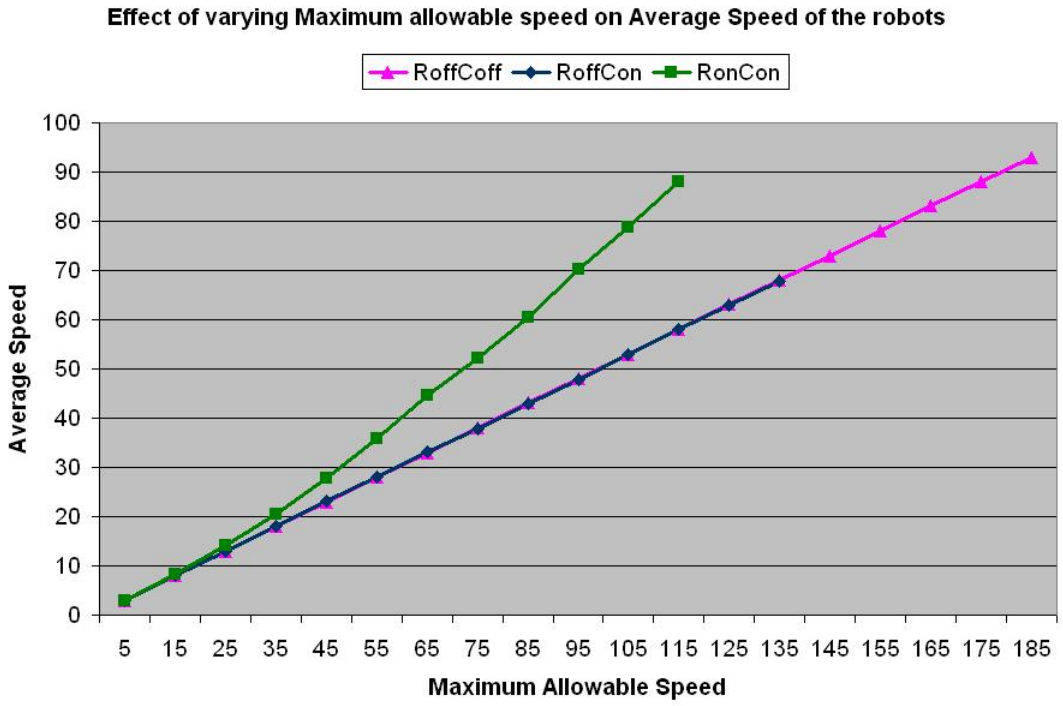




Figure 5.3: The average speed (5.3(a)) and the average distance (5.3(b)) perceived by the robots over 1000 seconds with 10 independent runs as the radius of the robots increases with increasing repulsion radius.



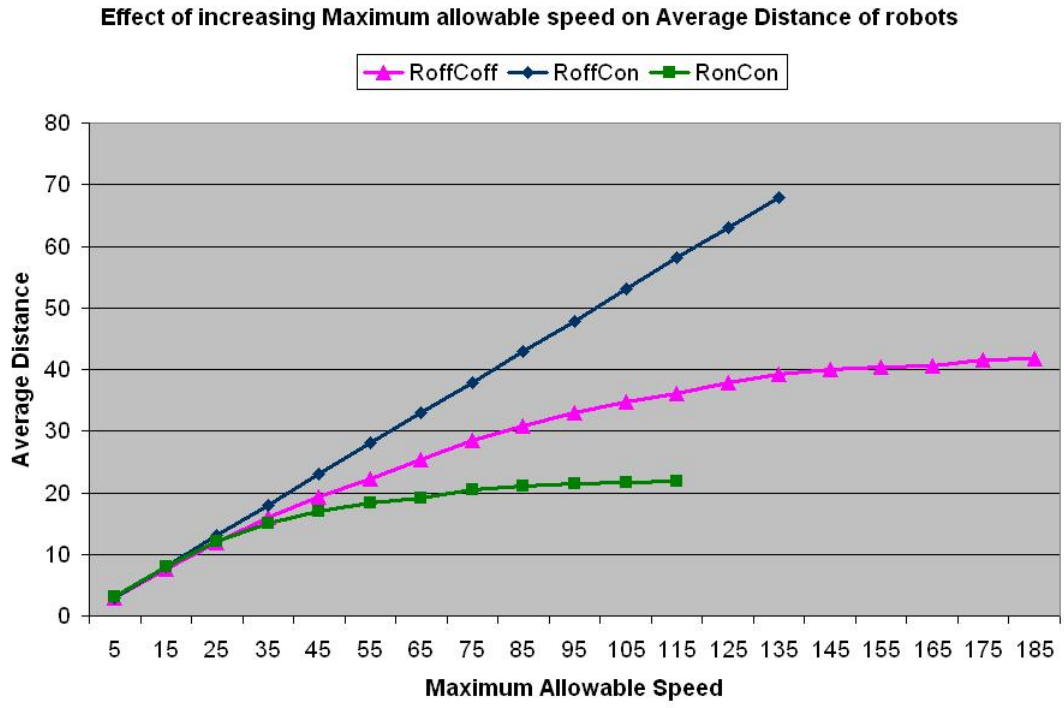




Figure 5.4: The average speed (5.4(a)) and the average distance (5.4(b)) perceived by the robots over 1000 seconds with 10 independent runs as the maximum allowable speed increases.

moving closer to other robots and thus higher repulsion forces. In turn, this increases the average speed induced by the repulsion forces. It tapers off due to the maximum speed limitation.

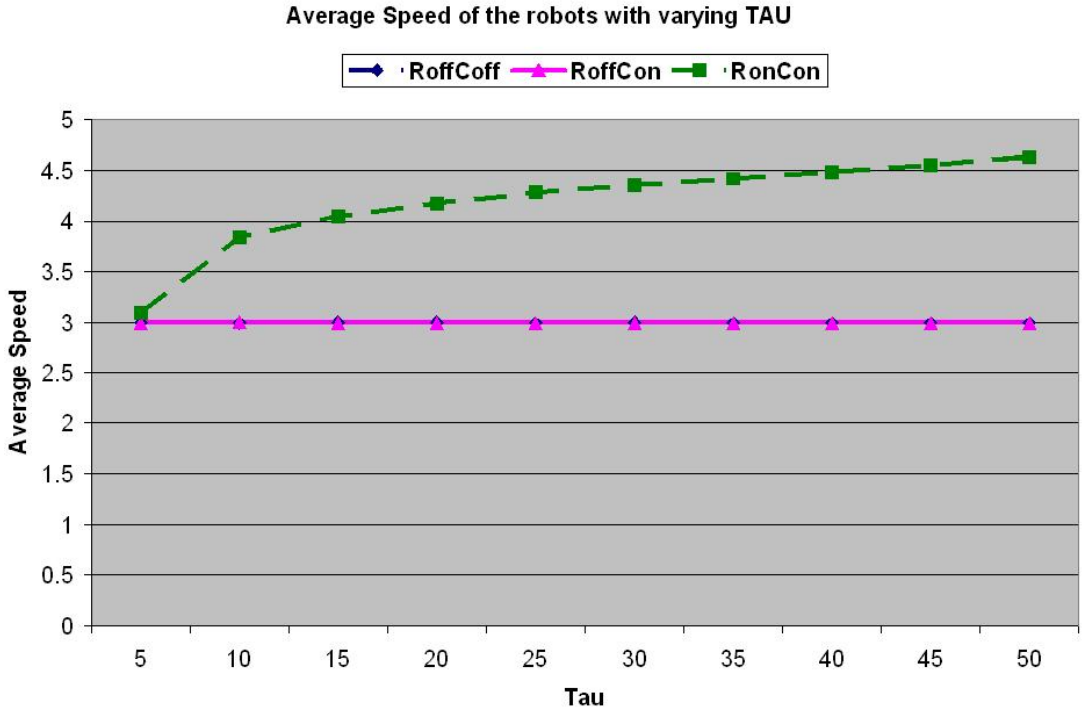
### 5.2.6 Dynamic Event

A dynamic event is a change brought about in the simulation environment during the run-time of the simulation. The objective of this study is to understand the effects of a sudden removal of a large number of robots on the average speed and the average distance of the existing robots. Figure 5.6 shows the transitions in speed and distance traveled occurring before and after such an event. Figure 5.6(a) shows no change for the *RoffCoff* and the *RoffCon* cases, but a slight decline for the *RoffCoff* case. The slight decline is due to the reduction of repulsion forces when there are fewer robots. The removal of some robots gives room for the other robots to move more freely, thus increasing the average distance traveled by them as shown in Figure 5.6(b). The decrease in speed is outweighed by the increase of room to move and thus the significant increase in the average distance traveled for the *RonCon* case.

### 5.2.7 3D analysis

The plots discussed in this chapter until now talked about the effects of varying several parameters like radius of the robots, their repulsion radius, the time duration of a single iteration '$\tau$' and the maximum attainable speed by the robots. The discussion has been limited to comparisons between two parameters. This type of analysis is of interest while analyzing the behavior of a parameter with respect to another over a period of time.

Sometimes though parameters of interest are so interrelated with each other that a variance in a parameter may be a result of two or more other parameters. In such cases it makes more sense to have a multi dimensional representation highlighting the effective behavior of the model with respect to all the concerned parameters whereby each dimension



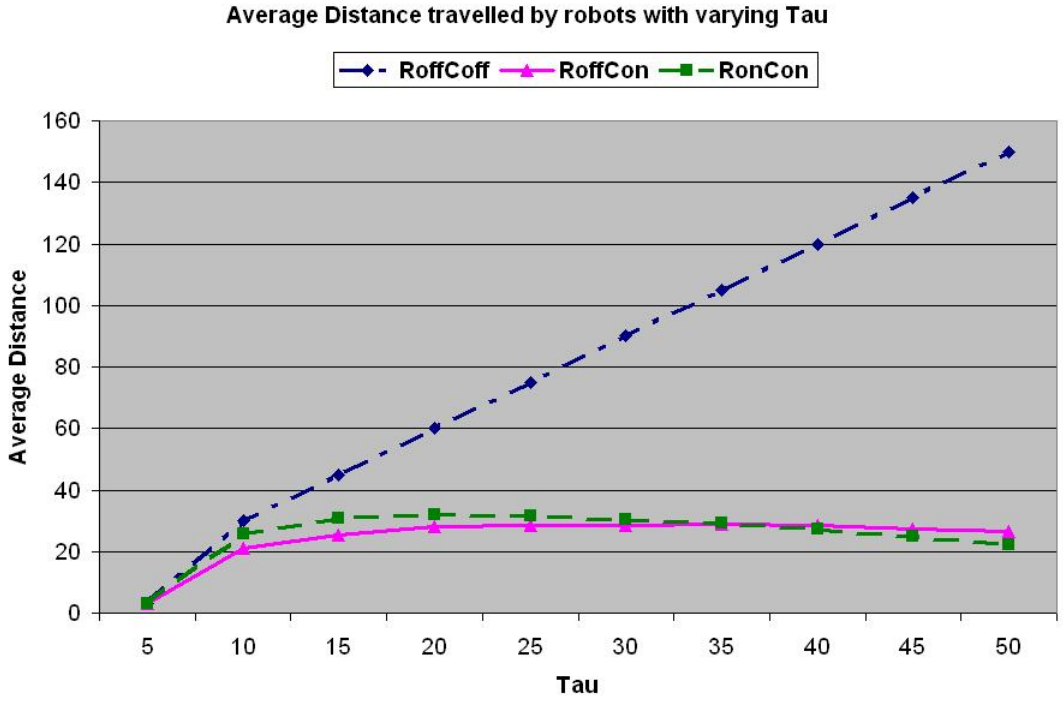
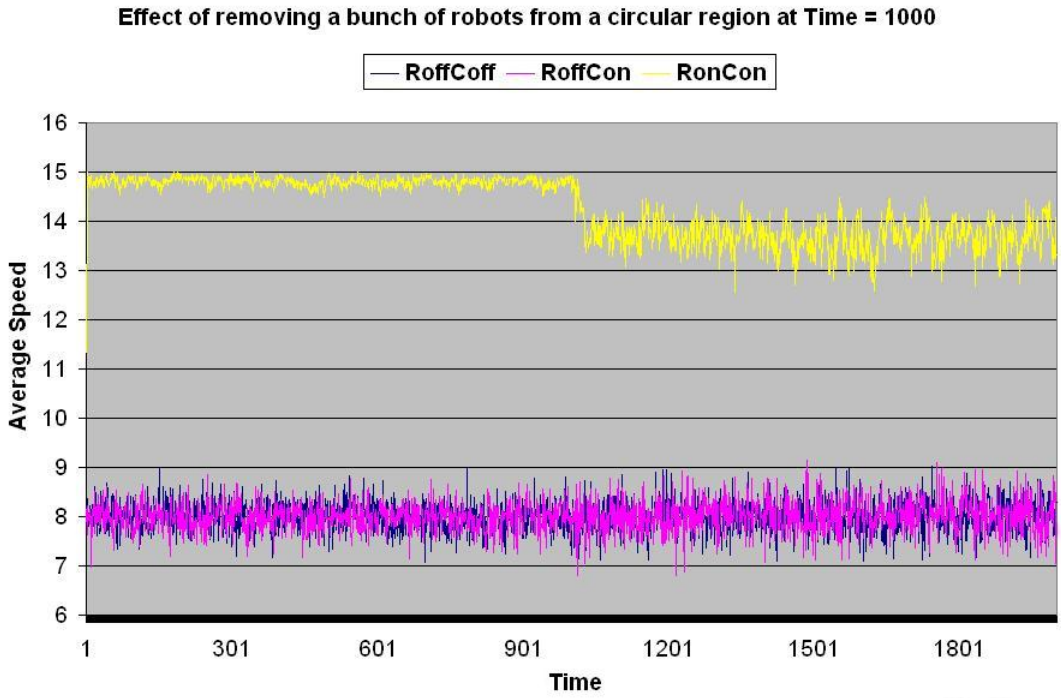




Figure 5.5: The average speed (5.5(a)) and the average distance (5.5(b)) perceived by the robots over 1000 seconds with 10 independent runs with increasing the update interval $\tau$.



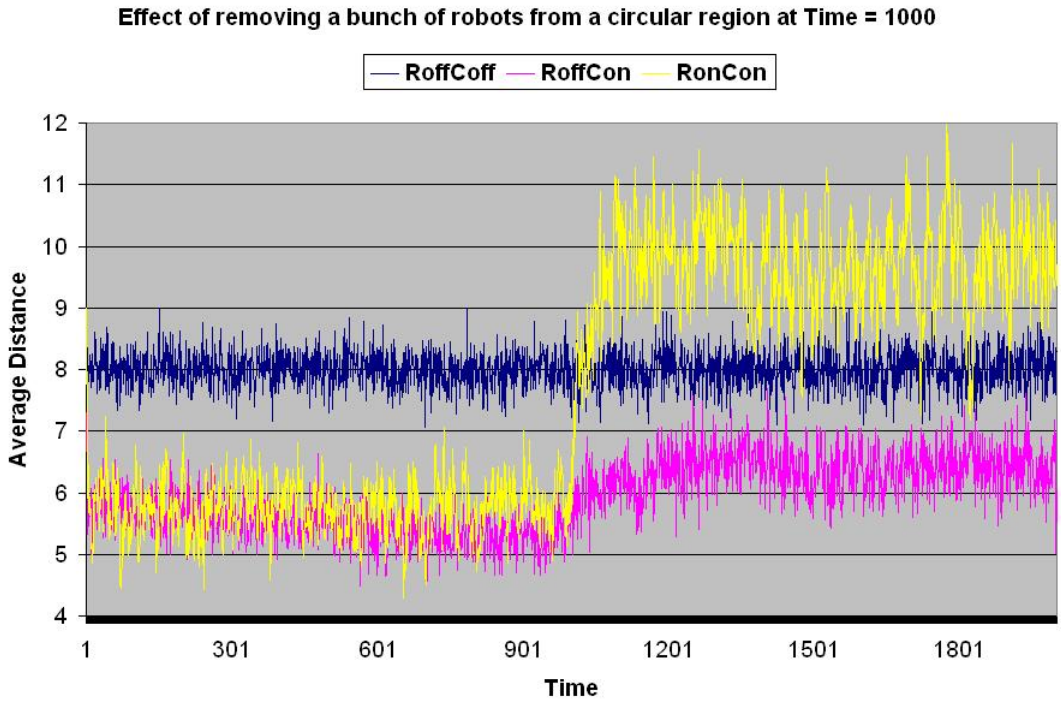




Figure 5.6: The average speed (5.6(a)) and the average distance (5.6(b)) perceived by the robots over time. Fifty out of 200 robots are removed at around time 900 seconds.

represents a parameter. In this discussion the focus is on four three dimensional (3D) representations or plots. They study the effect of maximum attainable robot speed and *Density* on the average speed attained by the robots and the average distance traveled by them. The curve in consideration represents the *RonCon* model. Since it has been observed on previous occasions that repulsion is a major contributor towards significant changes in the 2D curves seen above, the 3D analysis deals only with the effects of repulsion and hence the *RonCon* model.

The term *Density* mentioned above is percentage of the bounded region covered by the robot bodies. It is observed that both, a variance in the robot radius and the number of robots in the simulation affects the average speed and average density of the robots. Both these factors contribute towards an increase in the density of the robots within the bounded region in the simulation.

**Varying repulsion radius and average speed and distance**

Figure 5.7(a) shows the effect of increasing $V_{max}$ on the average speed with increasing density. The curve tends to get steeper with an increase in $V_{max}$ as well as an increase in density. This is because as $V_{max}$ increases it increases the ability of the robots to attain greater speeds resulting in a sharp increase in the curve. Also, as the density of the robots increases due to an increase in the repulsion radius of the robots, the chances of robots lying within each others repulsion region increase. This further bumps up their average speed as seen by the steeper curves at the back of the surface plane than those at the front.

Increasing density constantly helps increase the Average Distance covered by the robots. This is because increasing repulsion radii result in higher robot speeds. This in turn increases the average distance traveled by the robots. This however is not true for increasing values of $V_{max}$. Average Distance initially shoots up for an increase in $V_{max}$ because of the presence of reasonably sufficient number of repulsions and marginal collisions. The more the density the greater is its rate of increase. This situation however starts changing with increasing values of $Vmax$. This results in more collisions which overshadow the presence

of repulsions. This in turn brings down the average distance traveled by the robots. This makes all the curves settle down after some time as seen in Figure 5.7(b).

**Varying number of robots and average speed and distance**

Figure 5.8(a) shows the effect of increasing $V_{max}$ on the Average Speed with increasing density. The curve tends to get steeper with an increase in $V_{max}$ as well as an increase in density. This is because as $V_{max}$ increases it increases the ability of the robots to attain greater speeds resulting in a sharp increase in the curve. Also, as the density of the robots increases due to an increase in the number of robots, the chances of robots lying within another robot's repulsion region increase. This further bumps up their average speed.

The surface curve in Fig 5.8(b) looks like a half pipe. It is a clear indication of the steadiness with which parameters increase or decrease. An initial rise in $V_{max}$ leads to a sudden increase in the average distance traveled by the robots. After a certain time though increasing values of $V_{max}$ have no effect on the average distance. This is called the saturation point due to reasons mentioned in the case above. It is beyond this point that the Average Distance tends to stay constant throughout the run time of the simulation.

This point is different for different densities of robots in the simulation. As can be seen, for a lower density of robots in the simulation the Average Distance shoots up much higher than in the case of a densely populated region. So the saturation point is higher for a region with low robot density and vice versa.

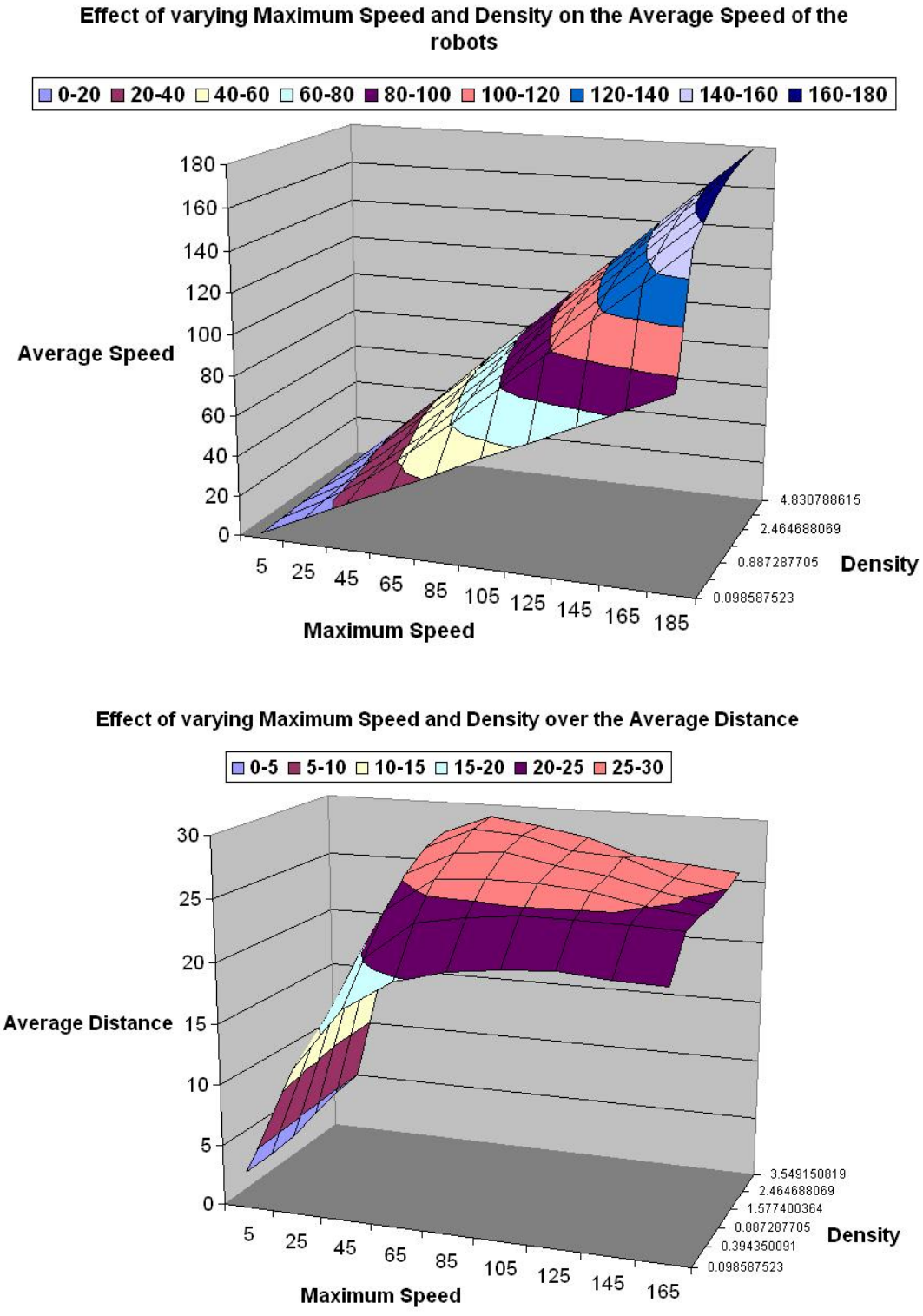



Figure 5.7: The average speed (5.7(a)) and the average distance (5.7(b)) plotted across varying values of repulsion radii of the robots as well as their density within the bounded region

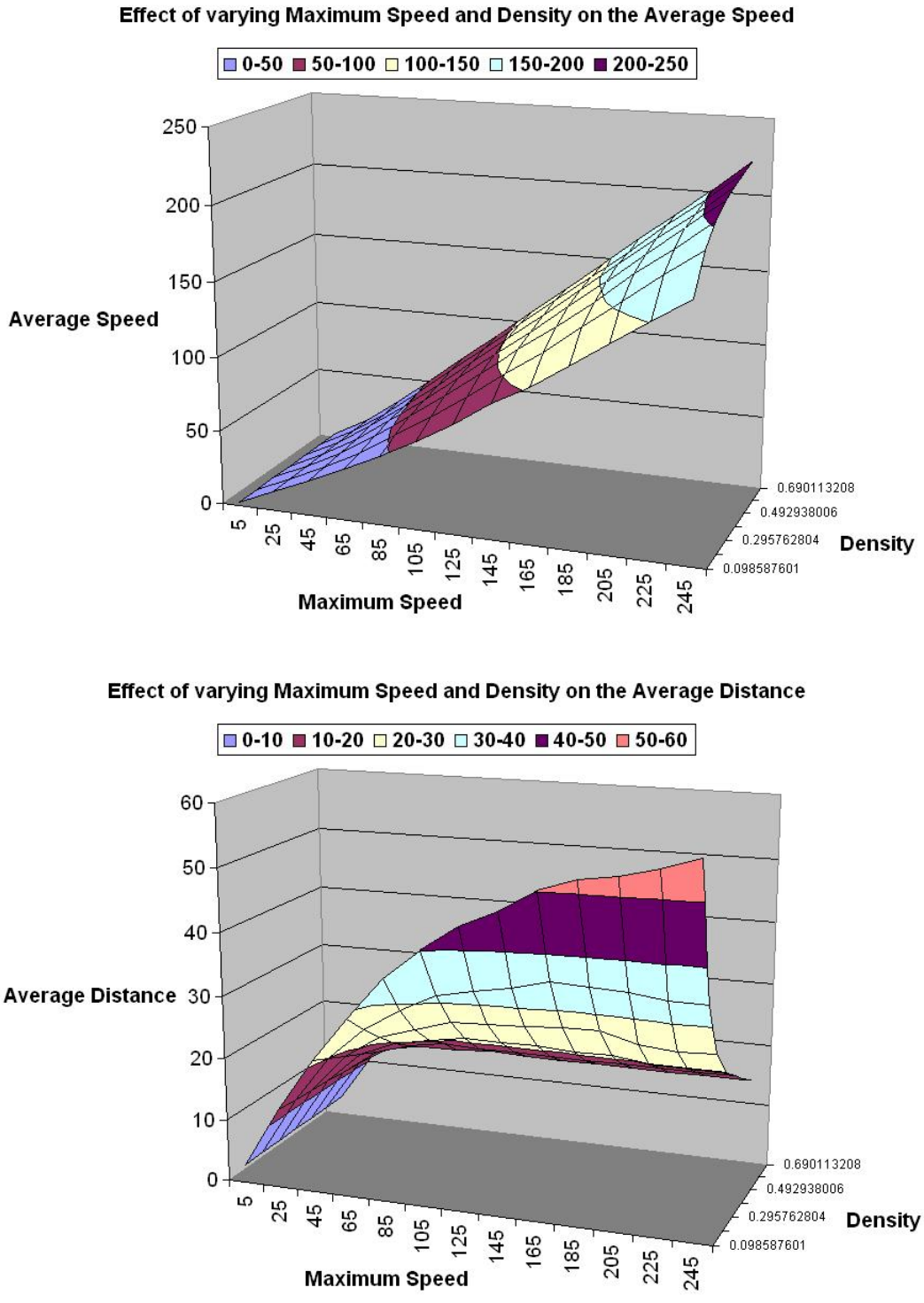



Figure 5.8: The average speed (5.8(a)) and the average distance (5.8(b)) plotted across varying values of repulsion radii of the robots as well as their density within the bounded region

# Chapter 6

# Conclusions

This work compares the mobility patterns exhibited by autonomous robots roaming in a confined region to random mobility models traditionally used for representing mobile device user behavior. Simulation results presented in this work suggest that the average speed exhibited by robots with only imminent collision avoidance is by and large similar to that exhibited by the random walk mobility model (random waypoint mobility model if the robot rotation times are taken into account). However, equipped with preventive collision avoidance (using artificial repulsion forces), the robots exhibit significant changes in both average speed and average distance traveled by them per iteration. In particular, a larger repulsion region, or higher repulsion force, results in higher speed (up to the robot maximum speed) but typically shorter travel distance. Factors, such as the maximum speed and the number of robots, have a different impact on the random mobility models as compared to the mobility pattern exhibited by the robots. When a significant number of robots are destroyed or deployed, the autonomous robots (with and without repulsion) will have noticeable changes to their movement pattern while no significant changes may be observed for the traditional random mobility models. These observations suggest the need for a more carefully designed mobility model for autonomous robots. We demonstrate that even in the simplest case, where robot's primary task is to roam randomly in a region, they still exhibit mobility patterns dissimilar to those represented by a traditional random mobility model.

While developing the simulator it was observed that there were some fallacies with the SHAPEBUGS [7] robot swarm algorithm. In particular, it was not able to repel two or

more robots moving away from each other. On the contrary it made them move towards each other. This problem coerced us to come up with a new equation which was better adept at handling repulsions. Work can be carried out to come up with an even better robot swarm algorithm capable of handling repulsions abiding by the laws of physics. This algorithm would then truly be capable of representing repulsions the way they occur in real life amongst circular bodies.

Some of the results in the previous chapter suggest that the density of robots depends not only on the number of robots within the simulation but also their radius. 3D representations can be created with the $x$ and $y$ axis represented by changing density due to the increasing number of robots and increasing robot radius respectively. The $z$ axis would represent the effect of these changes on either the average speed or the average density of the robots.

Work can also be carried out towards developing a random mobility model capable of representing autonomous robots. This would lead to further studies and developments in the field of ad hoc wireless protocols for communication between autonomous robots. Investigations could be carried out towards modeling random motions for individual robots while at the same time modeling their team behavior by a common factor. This could help represent cooperative robot behavior and such other applications in the future.

# Bibliography


[1] Amir Reza Momen, Jahangir Dadkhah Chime and Rey Branch. A New Analytical Method in User Mobility Modeling in Wireless Network. In *Proceedings of the International conference on Information and Communication Technologies: From theory to applications*, 2004.

[2] Network Simulator - ns-2. *http://www.isi.edu/nsnam/ns/*

[3] Global Model Simulator. *http://pcl.cs.ucla.edu/projects/glomosim/*

[4] Yuguang Fang, Imrich Chlamtac and Yi-Bing Lin. Channel Occupancy Times and Handoff Rate for Mobile Computing and Personal Communication System Networks. In *Proceedings of the IEEE Transactions on Computers*, 47(6): 679-692, 1998.

[5] Jungkeun Yoon, Mingyan Liu and Brian Noble. Random Waypoint Considered Harmful. In *Proceedings of the* $21^{st}$ *annual IEEE Infocom*, April 2003.

[6] Akiya Kamimura, Satoshi Murata, Eiichi Yoshida, Haruhisa Kurokawa, Kohji Tomita, and Shigeru Kokaji. Self-reconfigurable modular robot experiments on reconfiguration and locomotion. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 606-612, 2001.

[7] Jimming Cheng, Winston Cheng, Radhika Nagpal. Robust and self-repairing formation control for swarms of mobile agents. In *Proceedings of the* $20^{th}$ *national conference on Artificial Intelligence*, 2005.

[8] George Kantor, Sanjiv Singh, Ronald Peterson, Daniela Rus, Aveek Das, Vijay Kumar, Guilherme Pereira and John Spletzer. Distributed Search and Rescue with Robot and Sensor Teams. In *Proceedings of the* $4^{th}$ *International Conference on Field and Service Robotics*, July, 2003.

[9] Mohammad Rahimi, Richard Pon, William Kaiser, Gaurav Sukhatme, Deborah Estrin and Mani Srivastava. Adaptive sampling for environmental robotics. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2003.

[10] Haiyun Luo, Ramachandran Ramjee, Prasun Sinha, Li (Erran) Li and Songwu Lu. UCAN: A Unified Cellular and Ad-Hoc Network Architecture. In *Proceedings of the $9^{th}$ Mobicom, ACM International Conference on Mobile Computing and Networking*, pages 353-367, ACM Press, 2003.

[11] Ioannis Ioannidis and Bogdan Carbunar. Scalable Routing in Hybrid Cellular and Ad-Hoc Networks. In *Proceedings of the $1^{s}t$ IEEE Conference on Mobile Ad hoc and Sensor Systems*, Fort Lauderdale, FL, October 24-27, 2004.

[12] Tracy Camp, Jeff Boleng and Vanessa Davis. A Survey of Mobility Models for Ad Hoc Network Research. In *Wireless Communications and Mobile Computing: Special Issue on Mobile Ad Hoc Networking: research trends and applications, vol 2 no. 5 483-502*, 2002.

[13] Fan Bai and Ahmed Helmy. A Survey of Mobility Models for Wireless Ad Hoc Networks. In *Wireless Ad Hoc and Sensor Networks, Kluwer Academic Publishers*, 2004.

[14] Miguel Sanchez and Pietro Manzoni. A java based simulator for ad-hoc networks. *http://www.scs.org/confernc/wmc99/errata/websim/w408/w408.html*, May 2001.

[15] M. Kak. Random Walk and Theory of Brownian Motion. *American Math monthly, 369-391 volume 54*, 1947.

[16] Paul Langevin. On the theory of Brownian motion. In *Sur la theorie du mouvement brownien, Conference on Academic Science (Paris), 146, 530533*, 1908.

[17] David Johnson and David Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In *Tomasz Imielinski and Henry Korth, editors, Mobile Computing, volume 353, pages 152181*, Kluwer Academic Publishers, 1996.

[18] Josh Broch, David Maltz, david Johnson, Yih-Chun Hu and Jorjeta Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proceedings of MobiCom, ACM International Conference on Mobile Computing and Networking*, Oct 1998.

[19] Charles Perkins and Elizabeth Royer. Ad-hoc on demand distance vector routing. In *Proceedings of the $2^{nd}$ IEEE Workshop on Mobile Computing Systems and Applications, 90-100*, Feb 1999.

[20] Charles Perkins, Elizabeth Royer, Samir Das and Mahesh Marina. Performance comparison of two on-demand routing protocols for ad hoc networks. In *IEEE Personal Communications, volume 8, 16-28*, Feb 2001

[21] Christian Bettstetter and Christian Wagner. The Spatial Node Distribution of the Random Waypoint Mobility Model. In *Proceedings of the 1$^{st}$ German Workshop on Mobile Ad Hoc Networking WMAN*, 2002.

[22] Elizabeth Royer, Michael Melliar-Smith and Louise Moser. An Analysis of the Optimum Node Density for Ad hoc Mobile Networks In *Proceedings of the IEEE International Conference on Communications*, 857861, 2001.

[23] Ben Liang and Zygmunt Haas. Predictive Distance-Based Mobility Management for Multi-Dimensional PCS Networks. In *IEEE/ACM Transactions on Networking (TON)* Volume 11, Issue 5, Oct 2003.

[24] Viswanath Tolety. Load reduction in ad hoc networks using mobile servers. In *Masters thesis, Colorado School of Mines*, 1999.

[25] The Scenario Generator: A Tool To Generate Manet Mobility Scenarios for ns-2. *http://www.comp.nus.edu.sg/ liqiming/fyp/scengen/*

[26] James McLurkin and Jennifer Smith. Distributed Algorithms for Dispersion in Indoor Environments using a Swarm of Autonomous Mobile Robots. In *Proceedings of the 4$^{th}$ International Joint Conference on Autonomous agents and multiagent systems, The Netherlands*, 2005.

[27] Tien-Ruey Hsiang, Esther Arkin, Michael Bender, Sandor Fekete, and Joseph Mitchell. Algorithms for Rapidly dispersing Swarms in unknown environments. *Workshop on Algorithmic Foundations of Robotics*, arXiv.org $\geq$ cs $\geq$ cs/0212022, 10 Dec, 2002.

[28] David Payton, Mike Daily, Regina Estowski, Mike Howard and Craig Lee. Pheromone Robotics. In *Autonomous Robots 11, 319324, 2001*, ©2001 Kluwer Academic Publishers.

[29] Craig Reynolds. Flocks, Herds, and Schools: A Distributed Behavioral Model. *SIGGRAPH*, 1987.

[30] Tucker Balch and Maria Hybinette. Social potentials for scalable multi-robot formations. In *Proceedings of the IEEE International Conference on Robotics and Automation*, April 2000.

# Appendix A

# Source Code Modules

## A.1 Repulsion

```
/*****************************************************************************
* This function calculate the repulsion speed and direction for the two robots
* It calls the 'rep_force' module to calculate the 3rd component force
*****************************************************************************/
public void repulsion(ref ArrayList erb, robots robot)
{
   int i;
   bool tempflag = false;
   float distance, rx, ry, t_speed = t_direction = m_speed = m_direction = (float)0.0;
   StreamWriter SW;

   for(i=0;i<num_of_robots;i++)
   {
      if(((robots)erb[i]) != robot)
      {
         distance = (float)Math.Pow((float)Math.Pow((((robots)erb[i]).x - robot.x), 2) +
                    (float)Math.Pow((((robots)erb[i]).y - robot.y), 2) , 0.5);
         if(distance < robot.repulsion_radius - 1)
         {
            robot.repel_flag = true;

            // Call this module if you want to have the repulsion force in addition to the
            // velocities of the robots
            rep_force(((robots)erb[i]), robot); // Calculate repulsion force between two robots

            rx = robot.speed * (float)Math.Cos(robot.direction) + rep_robot.speed *
                 (float)Math.Cos(rep_robot.direction);
```

```
ry = robot.speed * (float)Math.Sin(robot.direction) + rep_robot.speed *
     (float)Math.Sin(rep_robot.direction);

t_speed = (float)Math.Pow( ((float)Math.Pow(rx, 2) + (float)Math.Pow(ry, 2)), 0.5);

tempflag = true;

if(rx==(float)0.0)
{
   if(ry==(float)0.0)
      t_direction = (float)0.0;
   else
      t_direction = (float)Math.PI / 2;
}
else
{
   t_direction = (float)Math.Atan(ry/rx);
   if(ry<(float)0.0)
   {
      if(rx>(float)0.0)
         t_direction+=(2*(float)Math.PI);
      else
         t_direction+=(float)Math.PI;
   }
   else
   {
      if(rx<(float)0.0)
         t_direction+=(float)Math.PI;
   }
}
rx = t_speed * (float)Math.Cos(t_direction) + m_speed * (float)Math.Cos(m_direction);
ry = t_speed * (float)Math.Sin(t_direction) + m_speed * (float)Math.Sin(m_direction);

m_speed = (float)Math.Pow( ((float)Math.Pow(rx, 2) + (float)Math.Pow(ry, 2)), 0.5);
if(rx==0.0)
{
   if(ry==0.0)
      m_direction = (float)0.0;
   else
      m_direction = (float)Math.PI / 2;
}
else
{
```

```
                m_direction = (float)Math.Atan(ry/rx);
                if(ry<0.0)
                {
                    if(rx>0.0)
                        m_direction+=(2*(float)Math.PI);
                    else
                        m_direction+=(float)Math.PI;
                }
                else
                {
                    if(rx<0.0)
                        m_direction+=(float)Math.PI;
                }
            }
        }//if
    }//if
    if(tempflag)
    {
        tempflag = false;
    }
}//for

// Store the new repelled speed and direction
robot.temp_speed = m_speed;
robot.temp_direction = m_direction;
}

/*****************************************************************************
* This routine calculates the repulsion force (3rd component force) between the two robots
*****************************************************************************/
public void rep_force(robots repeller, robots repellee)
{
    try
    {
        float distance = (float)0.0;

        rep_robot = new robots();

        if(repeller.x != repellee.x)
            rep_robot.direction = (float)Math.Atan(((float)repeller.y - repellee.y) /
            (repeller.x - repellee.x));
        else
            rep_robot.direction = (float)Math.PI / 2;
```

```
        if(rep_robot.direction < 0)
            rep_robot.direction = 2 * (float)Math.PI - rep_robot.direction;

        if(repellee.x < repeller.x)
        {
            rep_robot.direction += (float)Math.PI;
            rep_robot.direction %= ((float)Math.PI * 2);
        }

        distance = (float)Math.Pow( (float)Math.Pow(repeller.y - repellee.y, 2) +
                    (float)Math.Pow(repeller.x - repellee.x, 2) , 0.5 );

        if(distance > 1.0)
            rep_robot.speed = repeller.repulsion_radius - distance;
        else
            rep_robot.speed = robots.max_speed;
    }
    catch(Exception e)
    {
        MessageBox.Show(e.Message);
    }
}
```

## A.2 Edge Effect

```
/*****************************************************************************
* This function helps calculate the new direction so as to curtail the movement
* of the robots to the rectangular region
*****************************************************************************/
public void boundary_robots(robots bdry)
{
    int tw, te, tn, ts;
    int l1=0, l2=0;
    tw = (int)(bdry.x + bdry.speed * Math.Cos(180 * (Math.PI / 180)) * bdry.time);
    te = (int)(bdry.x + bdry.speed * Math.Cos(0 * (Math.PI / 180)) * bdry.time);
    tn = (int)(bdry.y + bdry.speed * Math.Sin(270 * (Math.PI / 180)) * bdry.time);
    ts = (int)(bdry.y + bdry.speed * Math.Sin(90 * (Math.PI / 180)) * bdry.time);
    if(ts >= SBimplement.ymax - robots.radius_robot)
    {
        if(tw <= SBimplement.xmin + robots.radius_robot)
        {
```

```
            l1 = 270;
            l2 = 360;
        }
        else if(te >= SBimplement.xmax - robots.radius_robot)
        {
            l1 = 180;
            l2 = 270;
        }
        else
        {
            l1 = 180;
            l2 = 360;
        }
    }
    else if(tw <= SBimplement.xmin + robots.radius_robot)
    {
        if(tn <= SBimplement.ymin + robots.radius_robot)
        {
            l1 = 0;
            l2 = 90;
        }
        else
        {
            l1 = 270;
            l2 = 450;
        }
    }
    else if(tn <= SBimplement.ymin + robots.radius_robot)
    {
        if(te >= SBimplement.xmax - robots.radius_robot)
        {
            l1 = 90;
            l2 = 180;
        }
        else
        {
            l1 = 0;
            l2 = 180;
        }
    }
    else if(te >= SBimplement.xmax - robots.radius_robot)
    {
        l1 = 90;
```

```
        l2 = 270;
    }

    if(l1==0 && l2==0)
        MessageBox.Show("Both l1 and l2 are 0", "Error",
        MessageBoxButtons.OK, MessageBoxIcon.Warning);

    bdry.direction = (float)(rand.crt_random(l1, l2) * (Math.PI / 180));
}
```

## A.3 Collision detection and avoidance

```
/*****************************************************************************
* This function computes the new position in case of a collision
*****************************************************************************/
public void solve_quadratic(ref ArrayList qrb)
{
    try
    {
        StreamWriter SW;
        StreamWriter SA;

        int i=0, j=0;
        float a, b, c, d;
        float ca, cb, cc;
        float p_root=(float)-1.0, n_root=(float)-1.0, root=(float)-1.0;
        float pnrt;

        bool global_flag = true;

        while(global_flag)
        {
            global_flag = false;

            for(i=0;i<num_of_robots;i++)
                if(((robots)qrb[i]).intr_flag == false)
                    ((robots)qrb[i]).temp_time = ((robots)qrb[i]).time;

            for(i=0;i<num_of_robots;i++)
            {
                for(j=0;j<num_of_robots;j++)
                {
```

```
if(i!=j && ((robots)qrb[j]).intr_flag==false)
{
   // Solving a quadratic equation
   a = ((robots)qrb[i]).x - ((robots)qrb[j]).x;
   b = (float)(((robots)qrb[i]).speed * Math.Cos(((robots)qrb[i]).direction)
       - ((robots)qrb[j]).speed * Math.Cos(((robots)qrb[j]).direction));

   ((robots)qrb[i]).newpos_x.a = (float)Math.Pow( a, 2 );
   ((robots)qrb[i]).newpos_x.b = 2 * a * b;
   ((robots)qrb[i]).newpos_x.c = (float)Math.Pow( b, 2 );

   c = ((robots)qrb[i]).y - ((robots)qrb[j]).y;
   d = (float)(((robots)qrb[i]).speed * Math.Sin(((robots)qrb[i]).direction)
       - ((robots)qrb[j]).speed * Math.Sin(((robots)qrb[j]).direction));

   ((robots)qrb[i]).newpos_y.a = (float)Math.Pow( c, 2 );
   ((robots)qrb[i]).newpos_y.b = 2 * c * d;
   ((robots)qrb[i]).newpos_y.c = (float)Math.Pow( d, 2 );

   ca = ((robots)qrb[i]).newpos_x.a + ((robots)qrb[i]).newpos_y.a;
   cb = ((robots)qrb[i]).newpos_x.b + ((robots)qrb[i]).newpos_y.b;
   cc = ((robots)qrb[i]).newpos_x.c + ((robots)qrb[i]).newpos_y.c;

   //Detect the collision of circumferences and not the centers of the robots.
   ca -= (float)Math.Pow((2*robots.radius_robot), 2);

   pnrt = cb*cb - 4*cc*ca;

   try
   {
      if(pnrt >= 0)
      {
         p_root = ( -cb + (float)Math.Pow( pnrt, 0.5 ) ) / (2*cc);
         n_root = ( -cb - (float)Math.Pow( pnrt, 0.5 ) ) / (2*cc);

         if(p_root>=0)
         {
            if(n_root>=0)
            {
               if(p_root<n_root)
                  root=p_root;
               else
                  root=n_root;
```

```
                }
                else
                    root = p_root;
            }
            else if(n_root>=0)
            {
                if(p_root>=0)
                {
                    if(n_root<p_root)
                        root=n_root;
                    else
                        root=p_root;
                }
                else
                    root=n_root;
            }
            else
                root=(float)-1.0;
            if(root >= 0 && root < ((robots)qrb[i]).time )
            {
                global_flag = true;
                ((robots)qrb[i]).temp_flag = true;
                ((robots)qrb[i]).time = root;

                ((robots)qrb[i]).c = (int)(
                                        ((robots)qrb[i]).x
                                      + ((robots)qrb[i]).speed
                                      * Math.Cos(((robots)qrb[i]).direction)
                                      * ((robots)qrb[i]).time
                                        );
                ((robots)qrb[i]).d = (int)(
                                        ((robots)qrb[i]).y
                                      + ((robots)qrb[i]).speed
                                      * Math.Sin(((robots)qrb[i]).direction)
                                      * ((robots)qrb[i]).time);
            }
        }
    }
    catch(Exception abc)
    {
        MessageBox.Show(abc.Message);
    }
}
```

```
            }//for loop (j)

        }//for loop (i)

        for(i=0;i<num_of_robots;i++)
        {
            if(((robots)qrb[i]).temp_flag)
            {
                ((robots)qrb[i]).intr_flag = false;
                ((robots)qrb[i]).temp_flag = false;
                ((robots)qrb[i]).speed = (((robots)qrb[i]).speed * ((robots)qrb[i]).time)
                                        / ((robots)qrb[i]).temp_time;
            }
            else
                ((robots)qrb[i]).intr_flag = true;
        }
    }//while loop
  }
  catch(Exception e)
  {
    MessageBox.Show(e.Message);
  }
}
```