RICE UNIVERSITY

Art and Engineering Inspired by Swarm Robotics

by

Yu Zhou

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Ronald Goldman, Chair
Professor of Computer Science

Joe Warren
Professor of Computer Science

Marcia O'Malley
Professor of Mechanical Engineering

Houston, Texas

April, 2017

RICE UNIVERSITY

Art and Engineering Inspired by Swarm Robotics

by

Yu Zhou

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Ronald Goldman, Chair
Professor of Computer Science

Joe Warren
Professor of Computer Science

Marcia O'Malley
Professor of Mechanical Engineering

Houston, Texas

April, 2017

ABSTRACT

Art and Engineering Inspired by Swarm Robotics

by

Yu Zhou

Swarm robotics has the potential to combine the power of the hive with the sensibility of the individual to solve non-traditional problems in mechanical, industrial, and architectural engineering and to develop exquisite art beyond the ken of most contemporary painters, sculptors, and architects. The goal of this thesis is to apply swarm robotics to the sublime and the quotidian to achieve this synergy between art and engineering.

The potential applications of collective behaviors, manipulation, and self-assembly are quite extensive. We will concentrate our research on three topics: fractals, stability analysis, and building an enhanced multi-robot simulator. Self-assembly of swarm robots into fractal shapes can be used both for artistic purposes (fractal sculptures) and in engineering applications (fractal antennas). Stability analysis studies whether distributed swarm algorithms are stable and robust either to sensing or to numerical errors, and tries to provide solutions to avoid unstable robot configurations. Our enhanced multi-robot simulator supports this research by providing real-time simulations with customized parameters, and can become as well a platform for educating a new generation of artists and engineers.

The goal of this thesis is to use techniques inspired by swarm robotics to develop a computational framework accessible to and suitable for both artists and engineers.

The scope we have in mind for art and engineering is unlimited. Modern museums, stadium roofs, dams, solar power plants, radio telescopes, star networks, fractal sculptures, fractal antennas, fractal floral arrangements, smooth metallic railroad tracks, temporary utilitarian enclosures, permanent modern architectural designs, guard structures, op art, and communication networks can all be built from the bodies of the swarm.

Acknowledgments

I would like to express my special appreciation and thanks to my advisor Professor Ronald Goldman for years of care and guidance. You helped me overcome the toughest problems during my PhD studies. Your deep insights in mathematics motivate me to think in-depth for the essential reasons and to think broad for the general solutions. Without your concerns and encouragements this thesis would not have been possible.

I would like to sincerely thank my thesis committee members, Professor Ronald Goldman, Professor Joe Warren, and Professor Marcia O'Malley for serving as my committee members despite busy schedules. Your insightful comments and helpful suggestions inspire me to keep improving.

I would like to thank my department coordinators, Beth Rivera and Belle Martinez, for helping me through all the required administrative processes and for providing me with important information through my graduate studies.

I would like to thank my Master's advisor Dr. James McLurkin. Thank you for leading me to the exciting research area of swarm robotics and thank you for letting me access the frontier of industrial applications.

I would like to thank my previous committee members and collaborators, Professor Lydia Kavraki, Professor Swarat Chaudhuri, Dr. Aaron Becker, Dr. Sándor P. Fekete, Dr. Golnaz Habibi, and Dr. Seoungkyou Lee. Your broad knowledge in the different areas provided me essential resources that I could expand my research.

I would like to thank my parents for decades of support towards my education. My achievements would not have been possible without your strict discipline and abundant resources.

Contents

Illustrations

Tables

Chapter 1

Introduction

1.1 Benefits of Multi-Robot Systems

A large population of low-cost robots is more suitable than a single expensive robot for solving problems in large environments. Swarm robots are also effective in microscopic environments, where large robots are not viable. Multi-robot systems provide these benefits: lower cost, greater robustness, and better physical coverage.

Lower cost. High-precision sensors and high-power actuators are difficult to manufacture and are typically highly overpriced. The cost of sensors and actuators is often related exponentially to precision and power. A large set of low-cost sensors with good software can usually achieve the same high resolution as a single expensive sensor. Swarm cooperation is often cheaper and more flexible than a single expensive multi-axle actuator, which is why modern quadcopters are more popular and less expensive than traditional helicopters with the same capacity.

Greater robustness. Carefully designed distributed algorithms for multi-robot systems can automatically bypass or average out inaccurate sensor readings, which is typically not possible for a single large robot with only a small number of sensors. Multi-robot systems may contain many similar robots, which can take over each other's tasks when a small proportion of the robots fail due to wear and tear. In comparison, if a single large robot such as a Mars rover fails, the whole project fails at the potential loss of much valuable research and billions of dollars.

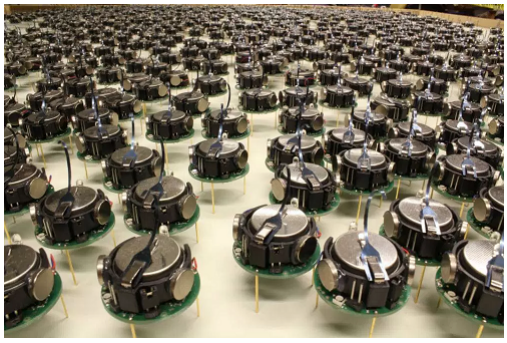
Better physical coverage. Even if a single powerful robot can be equipped with a high-resolution camera or long-range radar, a large population of low-cost robots can provide much larger physical coverage, continuous surveillance, and multiple perspectives. Swarm robots move locally after they are fully dispersed, in comparison to a single robot actively patrolling the whole environment. Multi-robot systems can even be dropped from an airplane to cover a large area, saving money by reducing wear and tear on motors and gears.

1.2 Existing Applications of Multi-Robot Systems

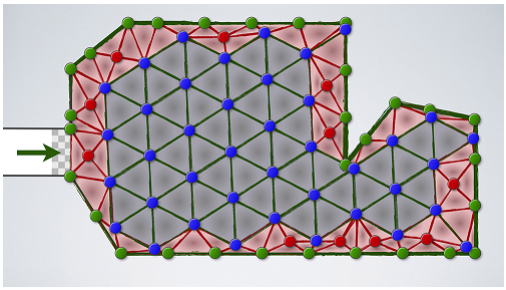
Multi-robot systems have numerous engineering applications. Collective behaviors mimic common behaviors of social animals and provide the foundation for controlling swarms of robots. Triangulation is a topological artifice that expands robots into unknown environments and builds a communication backbone. Manipulation allows robots to sense and actuate collaboratively and to transport large objects through complicated environments. Self-assembly utilizes the bodies of swarm robots to build temporary or permanent structures that are difficult or inefficient to manufacture in traditional ways.

1.2.1 Collective Behaviors

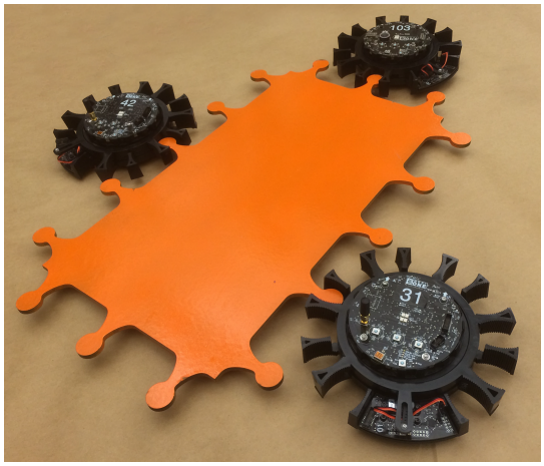
Social animals like ants and bees have complicated behaviors: individuals can collaborate to build structures orders of magnitude larger than themselves. Multiple ants can cooperate to transport large pieces of food, and even build bridges with their bodies [6]. Birds and fish fly and swim in formation and share the same direction. Researchers in swarm robotics would like to mimic these animal behaviors with swarms of robots.



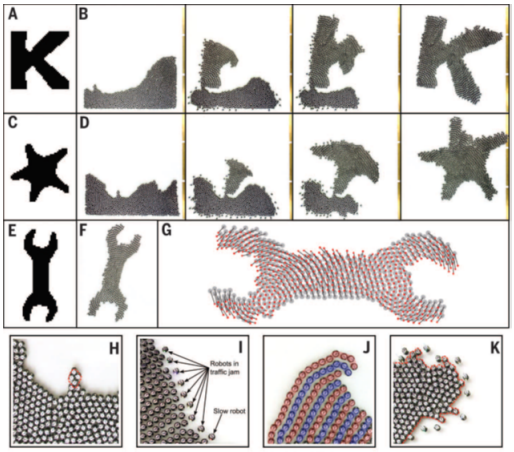
(a) Collective behavior: flocking [2].



(b) Triangulation [3].



(c) Swarm manipulation [4].



(d) Multi-robot self-assembly [5].

Figure 1.1 : Images from existing applications of multi-robot systems.

Flocking is the most common swarm behavior (Figure 1.1(a)), which occurs when all the individuals move in the same direction [7]. The first step is to turn the robots towards a common heading. Each robot turns in the direction that is the average heading of all its neighbors, and after some time to reach consensus, all the robots will share the same heading [8]. If the robots move at slightly different speeds, error accumulates and eventually tears the swarm apart. Therefore speed must be managed and feedback must be provided. These techniques are still not enough: if the swarm starts with some locally weak connections (articulation points), these connections may be accidentally broken and the robots may become separated. Some researchers insist on concentration so that weak connections get protected and strengthened, and avoid generating new weak connections [9].

When the swarm moves towards an obstacle, the first few robots change direction to avoid the obstacle, and other robots adjust headings according to the swarm rules. If all the robots try to avoid an obstacle by moving in the same direction, the swarm can successfully dodge the obstacle. But if robots avoid the obstacle by moving in different directions, the swarm may separate. Some researchers require robots to surround the obstacle and remain connected [10] [11], and some others allow temporary separation but try to reunite the robots at the other side of the obstacle [12]. There are also some other behaviors, such as following, clustering, and dispersing, which are demonstrated in some multi-robot systems. Some of these behaviors are learned from social animals, and some are invented by humans to control the robot swarm. Joysticks can be used by humans to interact with these robots, in order to change to a different behavior, manually control the leader robot, and globally control all the robots [13] [14] [15].

1.2.2 Triangulation

Triangulation is the process wherein robots build a triangular mesh [16]. Since a triangle requires only three non-collinear points and is stable with fixed edge lengths, a triangular network is easy to build with multiple robots and is a reliable way to relay communication.

The first two robots are manually placed to form a gate. Other robots go through the gate and locate their destinations. Once a robot passes through an external edge between two robots (initially the gate), the robot tries to stop at a position where the three robots form an equilateral triangle, and this robot becomes the owner of the triangle. Two external edges are generated, and the previous external edge becomes internal. The robot may also discover some adjacent triangles and become their owners. These generated triangles and discovered triangles form a triangle mesh, and expansion in one direction stops if there is obstacle in this direction. Other robots are guided to different external edges by the triangular network and repeat this process [17] (Figure 1.1(b)). Since robots might not form equilateral triangles due to sensor error, motion error, and obstacles, the triangular mesh may contain lattice defects.

Building a triangular mesh is often a good method to expand over an unknown environment and remain connected. If each robot has some sensing ability such as metal detectors, the robots will be able to find areas that are likely to contain minerals. The triangular network can efficiently pass these kinds of information back to a base station, and guide another set of robots to the target areas for mining [17]. Triangulation can also provide a communication backbone and area division for cooperative surveillance [18].

1.2.3 Manipulation

Multiple ants can transport large pieces of food by cooperatively holding the food along all the sides and moving in the same direction. When humans move furniture into a new house, workers need to make sure that they have enough force to lift the furniture, and rotate the furniture to pass through doors and stairs. Humans want robots to carry large objects to save man power, so researchers try to endow swarm robots with the ability to cooperatively transport large objects.

To deal with large objects, the first issue to consider is how robots interact with objects. Some researchers assume that the object is manually mounted on the robots so that the object can never fall to the ground [19]. When the robots move, their relative positions are fixed due to the rigid object. Some researchers assume that the object has handles and the robots have grippers, and once gripped robots can push or pull the object [4] (Figure 1.1(c)). Since there is friction between the object and the ground, and grippers can pivot at the attach point, robots need to call for reinforcements if they do not have enough force. Some researchers assume that the object can slide on the floor, and robots have no special manipulators but simply push the object with their bodies [15]. Robots may slide away from the object, so robots need to compact their swarm and come back for another push. In some more realistic applications, robots lift the object with their arms like humans, and the robots need to take care of the forces to prevent the object from falling [20].

The second issue is to transport an object along a desired path. Robots need to face in the same direction and move at the same speed. The most common solution is to select a leader distributively or manually. The leader robot moves along the desired path, and the other robots move in the same direction as the leader at the same speed. Slight difference in the direction or speed can accidentally rotate the object or push

the object sideways, so feedback is needed to keep robots on the desired path.

When the environment contains obstacles, or especially narrow passages, the robots need to rotate the object to avoid collision. The object can be irregular, and thus the robots need to figure out where the mass center of the object is, what the largest dimension is, and what the smallest dimension is. The environment can be partitioned into different safety levels: the safe zone is where the object does not collide with obstacles in any directions; the risk zone is where the object collides in some directions and does not collide in some other directions; the collision zone is where the object collides in all directions. Robots can move freely in the safe zone, but needs to carefully rotate the obstacle before entering a risk zone. The motion planner needs to balance between the length of the transportation path and the additional work needed to rotate the object [19].

1.2.4 Self-assembly

Self-assembly is a procedure wherein several agents change positions and connections in order to automatically build an organized structure. Self-assembly is discovered in a large range of scales in nature. Atoms and molecules automatically assemble into crystals [21]. DNA and protein self-assemble from nucleotides and amino acids to build live cells [22]. Some species of ants can create a bridge with their bodies so that coworkers can cross the gap quickly to bring back food [6]. Researchers are actively developing algorithms and mechanisms to implement self-assembly on multi-robot systems, and utilize self-assembly in artistic and scientific projects.

Imagine swarm robots constructing temporary buildings, such as galleries in a World Expo, with the robots own bodies. A set of brick-shaped robots, smart bricks, are transported to the construction site. Smart bricks have coils so they can generate

magnetic forces and relay electric energy. A smart brick climbs a wall of smart bricks with magnetic interaction, moves to the desired area and interlocks with other smart bricks. Multiple smart bricks can join the construction in parallel to save time. After the temporary building completes its mission, smart bricks automatically change into a pile of bricks for transportation. Construction and destruction of temporary buildings with smart bricks can greatly save building materials, reduce non-renewable energy consumed by construction vehicles, and protect the environment.

The most recent research on self-assembly with high reputation is by Harvard University Self-Organizing Systems Research Group, which is published in Science Magazine about a multi-robot controlling method to deploy a large number of robots to form a given shape [5] (Figure 1.1(d)). Initially, four robots are specially programmed and manually deployed into special positions. The other robots are placed in a solid group adjacent to the special robots, and are given the desired shape as a bitmap. When the algorithm starts, a spanning tree is built and rooted at one of the special robots. The spanning tree classifies robots into different layers with different gradient values, and only those robots in the outer-most layer (with the largest gradient) start to follow the edge of the swarm. These robots calculate their positions in the bitmap through a distributed inter-robot positioning system, and stop upon exiting the desired shape or filling in the next position in the current layer. Repeating this procedure converts the initial swarm into the desired shape.

The Harvard researchers had great success because self-assembly is a very popular research area, and they provide a strong background from bioscience and potential applications in bioscience. They also set up the worlds largest robot swarm: 1,024 Kilobots were used to implement their algorithm in a single experiment. Their Kilobot design [23] with compact size, extremely low cost, reflective infrared communication,

vibration motors, and charging connectors enables them to implement this algorithm with real robots and study high-density swarm behaviors.

Professor Daniela Rus and her team at MIT built a swarm of cubic robots that can flip, jump, and self-assemble [24]. Each robot looks like a cube with no external moving parts, but can magically move and jump. This magic is powered by flywheels. When a flywheel suddenly brakes, the angular momentum transforms to torque on the object and forces the object to move. With the help of edge magnets and face magnets, robots can assemble into complicated structures, and can build, change, or decompose the structure via individual movements. ETH Zurich in Switzerland has a similar design where a robot cube can balance on its edge or vertex [25].

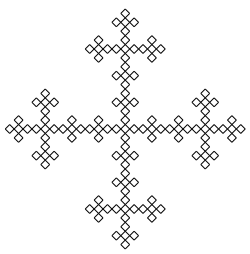
Chapter 2

Building Fractals with a Robot Swarm

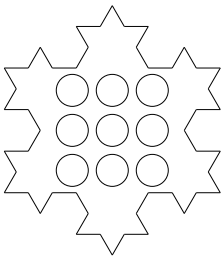
2.1 Motivation and Related Work

Fractals are self-similar shapes [26]. For example, a minor branch of a tree is similar to a major branch at a smaller scale, and a snowflake has complicated details that have small structures similar to the structure of the snowflake as a whole. Fractals are common in nature, and include such diverse objects as plants, crystals, mountain surfaces, lightning bolts, and tracheobronchial trees.

We want robots to form fractals because fractals are useful in engineering – certain antennas have fractal shapes [27] [28] [29] [30] [31] (Fig. 2.1); civil utility structures [32] and the Internet [33] demonstrate self-similar behaviors. Fractal formations can be useful as well for generating aesthetic shapes [34] [35] [36], such as artificial trees, or beautiful patterns of flowers in a botanical garden.



(a) Vicsek fractal (redrawn from [27]).



(b) Koch snowflake with round holes (redrawn from [28]).

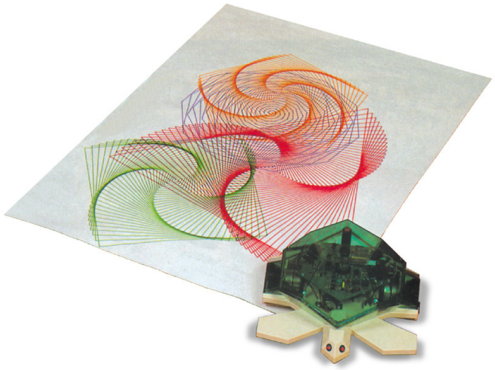
Figure 2.1 : Fractal antennas with different designs.

Two traditional methods for generating fractals are Recursive Turtle Programs [37] and Iterated Function Systems [38]. Recursive Turtle Programs require a pre-set depth, and a single turtle draws the fractal in a single thread, like a Depth-First Search. Iterated Function Systems require keeping track of a detailed description of a large number of objects so that at each iteration each object can be replaced with a self-similar structure.

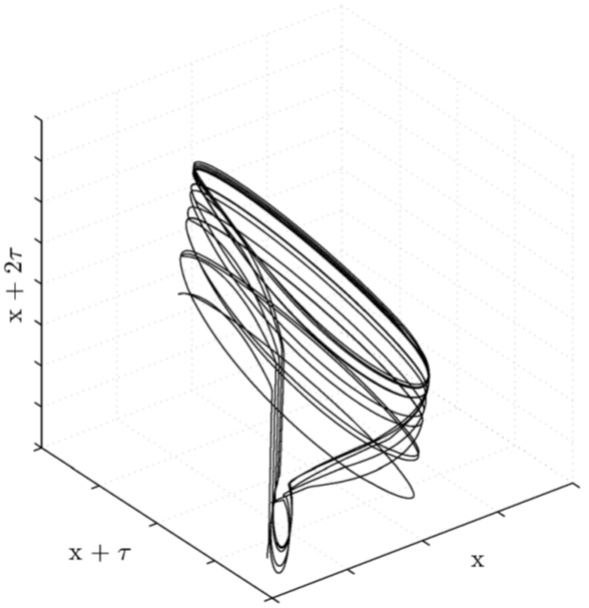
In the real world people manufacture fractals with several different techniques. One approach is to draw the fractal with a pen or to stitch the fractal with a sewing machine, following the trajectory generated by a recursive turtle program. The Valiant Turtle [39] (Figure 2.2(a)) was introduced in 1983 as a robot toy to draw fractals on paper. Another approach is to build the fractal as a whole by printing or casting; this method is typically used to manufacture fractal antennas. Fractals can also be assembled with pre-made building blocks, such as cellular base stations disguised as trees. The assembly process is usually single-threaded either by a human or by a robot arm.

We are interested in generating fractals with multi-robot systems, because robot swarms are a cheap, effective, and reconfigurable tool for generating complex shapes. With the help of multiple robots, we can generate fractals in parallel, and the detailed description can be distributed to different robots so that each robot needs to remember only a small number of states.

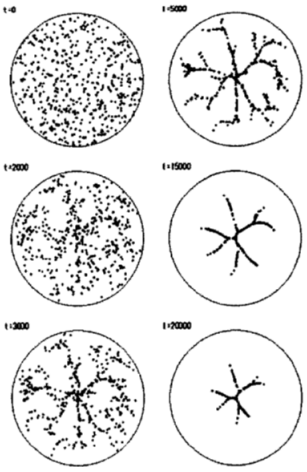
There are several articles related to robots forming randomized and deterministic fractals. Rold [41] discovers that multiple robots can exhibit fractal dimensions by programming the robots to simulate attractor behaviors (Figure 2.2(b)). Sugawara and Watanabe [42] gather robots towards the center of the environment, and the clustering process builds a fractal tree (Figure 2.2(c)). Aznar, Pujol, and Rizo [43]



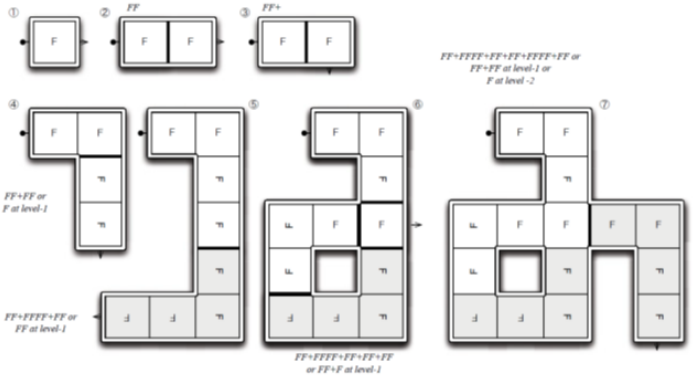
(a) The Valiant Turtle [39] (image from Wikipedia [40]).



(b) Robot simulates attractor behaviors [41].



(c) Recovery tree [42].



(d) Multi-robot self-assembly [43].

Figure 2.2 : Figures from related work.

describe a multi-robot self-assembly procedure with L-Systems (Figure 2.2(d)).

We are also inspired by algorithms that place robots into other formations. Lee, Fekete, and McLurkin [44] develop an algorithm to build a triangular mesh with a robot swarm. This algorithm can be used to explore unknown spaces and guide other robots for security patrols. Alonso-Mora et. al. [45] move multiple robots into solid geometric shapes in order to generate artistic patterns. Guo, Meng, and Jin [46] deploy swarm robots on a NURBS curve to approximate the boundary of an arbitrary shape.

2.2 Model and Assumptions

Fractals have complicated structures, but these structures can be generated from simple growth rules. Based on how to apply these rules with multi-robot systems, we classify fractals that can be approximated by line segments into four major types:

- Tree-based fractals: fractals with branching growth rules and a tree-like skeleton.
- Curve-based fractals: fractals formed by polygonal chains, or represented by subsets of polygonal chains.
- Space-filling fractals: similar to curve-based fractals, these fractals are continuous and dense in the unit square.
- Shape-based fractals: fractals that are often drawn with colored fills.

There are many fractals outside these categories, especially those fractals that cannot be effectively approximated by line segments or polygons, like the Mandelbrot set

[47] and the Lorenz attractors [48]. This chapter focuses on 2-dimensional fractals for which robots can be used to outline their skeletons.

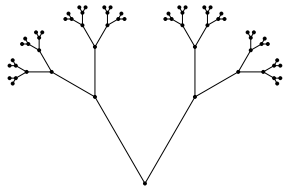
Fractals are defined as a limit of an infinite number of iterations, but fractals can only be printed or manufactured with some finite number of vertices and edges. Therefore, we are more interested in approximating a fractal with a limited number of iterations. We define the *level* of a fractal as the depth of the iterations or recursions used to approximate the fractal. We also define the *level* of a vertex as the lowest level at which the vertex appears in the fractal. A lower level is a level with fewer iterations; a higher level is a level with more iterations.

2.3 Algorithms

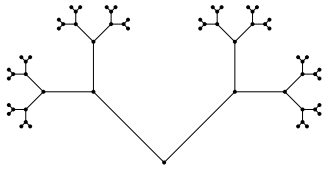
2.3.1 Tree-based Fractals

Fractal trees are fractals with a branching growth rule. The growth rule should have two or more branches in non-trivial directions with a scale factor < 1 so that the fractal converges to a tree-like shape. Fractal trees differ from each other by a set of parameters (Fig. 2.3): the number of branches, the angle between branches, the scale factor, symmetric or asymmetric rules, and the activeness of each node. We need to define a set of rules so that the robots generate the desired fractal.

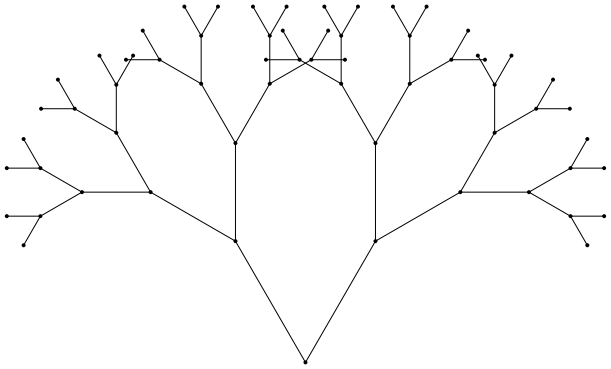
For iterated function systems the base case is usually represented by a line segment, and this line segment is replaced by a set of line segments described in the growth rule. To implement these growth rules using robots, we denote the base case by a single vertex and an expansion direction. We manually place a stationary initial robot into the workspace, heading towards a fixed direction. We define a polar coordinate system whose pole O lies on the center of the initial robot and whose polar



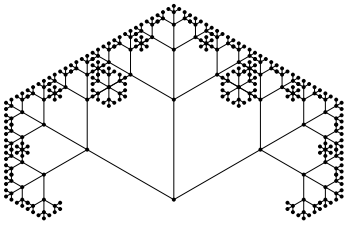
(a) A fractal tree with two branches.



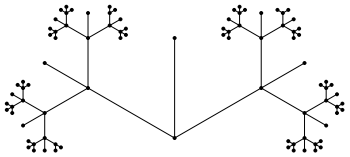
(b) With a different angle.



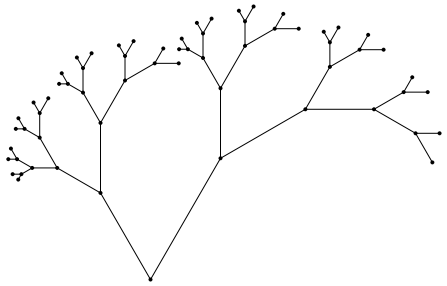
(c) With a different scale factor.



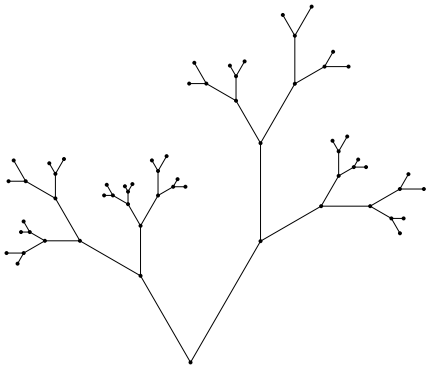
(d) With three branches.



(e) With inactive branches.



(f) Asymmetric.



(g) Asymmetric alternative.

Figure 2.3 : Fractal trees with different parameters (shown at level 5).

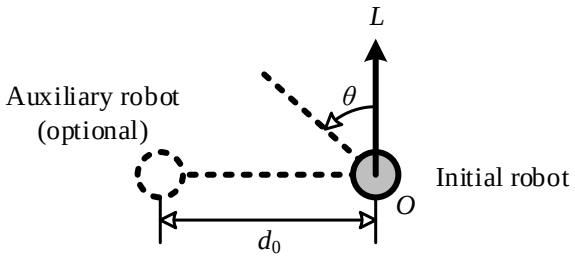


Figure 2.4 : Polar coordinate system. We always assume that the polar axis is in the upward direction so we will omit the polar axis in all subsequent drawings.

axis L points in the direction of the heading of the initial robot (Fig. 2.4). We also set up an initial distance d_0 either manually or by placing another auxiliary robot whose distance to the initial robot is d_0.

For the simplest fractal tree, a k-branch symmetric fractal tree, we describe the growth rule with k vectors extending from the pole O, embodied as k robot children forming branches from their parent robot. Each vector v_i can be described by its angle θ_i from the polar axis L, in addition to its scale factor s_i. For a symmetric fractal tree, the angles $\{\theta_i\}$ form an arithmetic sequence symmetric about zero, and all the scale factors $\{s_i\}$ are equal (Fig. 2.6(a)).

To simplify the description of our algorithm, we shall initially assume that the robots have a sensing range larger than the length of any edge in the fractal tree. In addition, we will assume that a robot can always move accurately a given distance in a given direction without colliding with other robots. We will handle limited sensing range, errors in accuracy, and collisions later in this section.

When a new robot joins the building process, the robot starts near the initial robot with the same heading as the initial robot. The new robot treats the initial robot as its parent. Now there are k directions in which to grow the fractal tree. One strategy is to pick a branch randomly from $1, ..., k$; another strategy is to ask

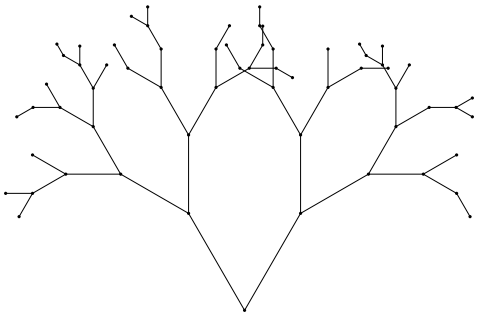


Figure 2.5 : Fractal tree generated with random branching.

the parent robot to select a branch, and have the parent robot reply sequentially with $1, ..., k$ to each request. These different strategies determine whether the fractal tree grows probabilistically (Fig. 2.5) or deterministically (Fig. 2.3(c)). Once the new robot knows in which branch i to grow, the robot rotates by θ_i to face towards that branch, where $\theta_i = \frac{k+1-2i}{2}\alpha$ if the angle between two branches is given by α, or $\theta_i = \frac{k+1-2i}{2k-2}\beta$ if the angle between the leftmost and the rightmost branches is given by β.

The robot moves a distance d in that direction. When the robot arrives at its target, the robot becomes a vertex of the fractal tree and remains stationary during the remainder of this algorithm. However, if another robot has already occupied this position, the new robot selects that robot as its parent, repeats this process as growth in a sub-tree, and shortens the next distance by a given scale factor s.

When we add more and more robots at the location of the initial robot and all the robots follow this same protocol, these robots build a symmetric fractal tree.

We present our main algorithm in Algorithm 1, along with several customizable helper functions in Algorithms 2 to 5.

Next we enhance our algorithm to deal with more complicated fractal trees:

Asymmetric angles. Instead of calculating θ_i from the adjacent branch angle α or

Algorithm 1 FractalTree(u)

```
1: InitRobot(u)
2: while u.state ≠ STOP do
3:    if u.state = EXPAND then
4:       if ∃v such that v.parent = u.parent and v.branch = u.branch and v.level = u.level and
         v.state = STOP then
5:          u.state ← FOLLOW
6:          if u.branch is active branch then
7:             u.parent ← v
8:             u.level ← v.level + 1
9:             u moves towards v
10:         else
11:            u backs to parent and selects another branch
12:         end if
13:      else if u reaches expansion destination then
14:         u.state ← STOP
15:      else
16:         u continues current expansion motion
17:      end if
18:   else if u.state = FOLLOW then
19:      if u is close enough to v then
20:         u.branch ← GetBranch(u)
21:         if u.branch is active or inactive branch then
22:            u.state ← EXPAND
23:            u.length ← u.length×GetScale(u)
24:            u rotates GetAngle(u.branch)
25:            u moves forward for a distance up to u.length
26:         else
27:            u.length ← u.length×GetScale(u)
28:            u.level ← u.level + 1
29:         end if
30:      else
31:         u continues current following motion
32:      end if
33:   end if
34: end while
```

Algorithm 2 InitRobot(u)

```
1: if ∃v such that v.level = 0 then
2:    u.parent ← v
3:    u.level ← 1
4:    u.state ← FOLLOW
5: else
6:    u.parent ← ∅
7:    u.level ← 0
8:    u.state ← STOP
9: end if
10: u.length ← an initial length
```

Algorithm 3 GetScale(u)

1: *scale* ← a constant value less than 1
2: **return** *scale*

Algorithm 4 GetBranch(u)

1: **if** random branch mode **then**
2: *branch* ← a random branch among all active, inactive, and head branches
3: **else**
4: *branch* ← ask *u.parent* about the next branch at *u.level*
5: **end if**
6: **return** *branch*

the total branch angle β and the number of branches k, the user can specify an array of angles to describe the directions of the branches.

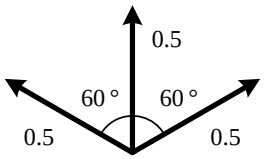
Asymmetric scale factors. Instead of all the robot children moving the same distance, the user can specify an array of scale factors. Individual scale factors apply to each branch, and the size difference of sub-trees amplifies in higher levels. All of the scale factors should be less than 1 to ensure convergence. We can combine the angles and the scale factors into vectors (Fig. 2.6(b)).

Inactive branches. Some branches can be inactive, i.e. no fractal sub-trees grow from inactive branches. The user can specify an array of Boolean values to indicate whether each branch is active or inactive (Fig. 2.6(c)). If a robot selects a branch randomly and a robot is already present in this inactive branch, the robot must return to its parent and reselect another branch. If the parent assigns branches for robot children, an inactive branch can be assigned only once.

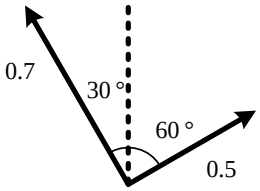
Complicated local structure. There are some tree-based fractals with special local

Algorithm 5 GetAngle(u)

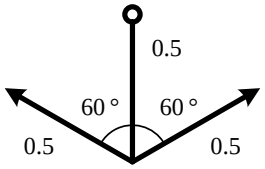
1: *angle* ← $(k/2 + 1/2 - u.branch)\alpha$ where k is the number of active and inactive branches
2: **return** *angle*



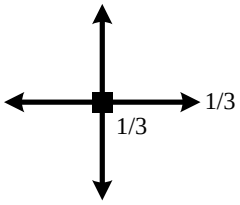
(a) Growth rule for 3 symmetric branches.



(b) Growth rule with arbitrary vectors.

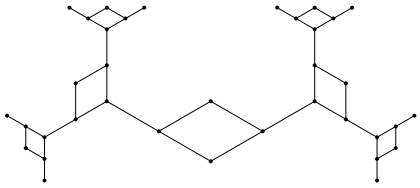


(c) No iteration at inactive branch.

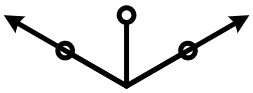


(d) Growth rule for Vicsek fractal with an additional iteration at the root.

Figure 2.6 : Fractal tree growth rules.

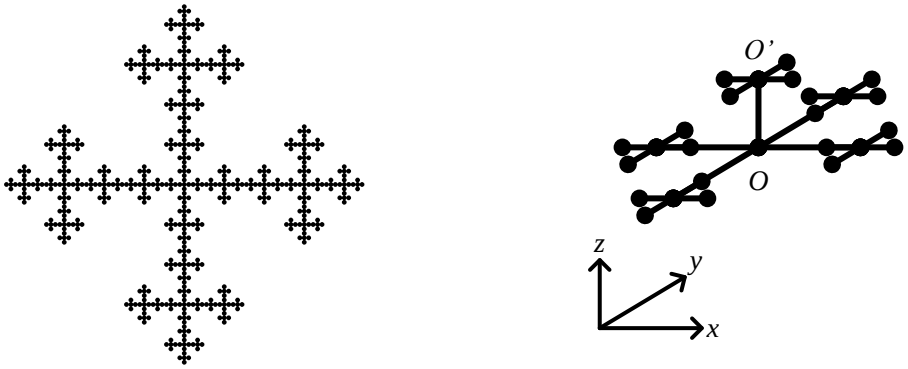


(a) A fractal tree with local cycles (shown at level 3).



(b) Growth rule without the loop.

Figure 2.7 : Fractal tree with complicated local structure.



(a) Vicsek fractal shown at level 4.

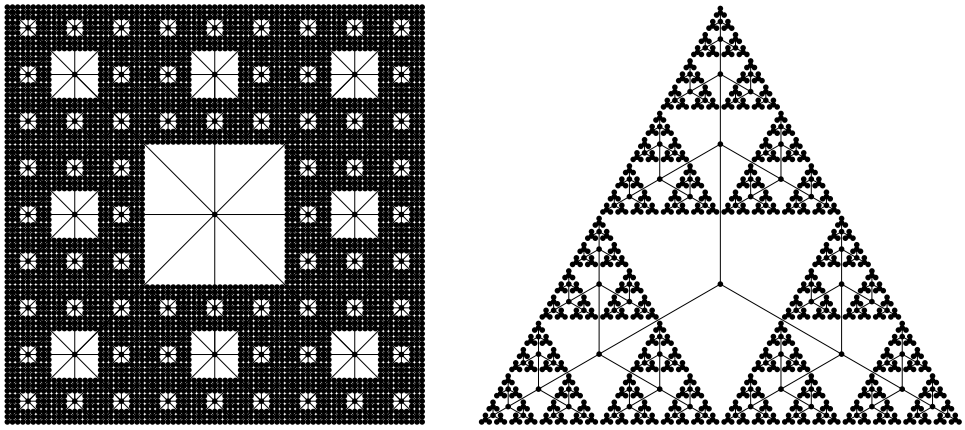
(b) An illustration of growing into three-dimensional space. O and O' should be occupied by the same robot in the plane.

Figure 2.8 : Vicsek fractal can be divided into 5 similar parts.

structures including cycles (Fig 2.7(a)). We can simply convert these structures to vectors extending directly from the parent robot, and mark some of the branches as inactive (Fig. 2.7(b)).

Vertices with multiple degrees. Fractal trees usually grow only at the vertices on the highest level, but some fractals such as Vicsek fractals (Fig. 2.8) grow at all the vertices. These fractals can be treated as if they have additional branches above the plane extending into three-dimensional space, and the original fractal is the parallel projection onto the plane of a three-dimensional fractal. Thus a parent robot can have children robots at various levels. When a moving robot approaches a stationary robot and selects this stationary robot as its parent, the moving robot can either branch immediately or fall into the higher level by applying the scale factor to its future motion and choosing a branch again at the same parent (Fig. 2.6(d)). Since any stationary robot is considered as multiple instances in the different levels, if a child robot asks the parent robot to assign a branch, the parent robot needs to maintain multiple states for different levels in order to answer at the child's level.

Complicated growth rules. Some shape-based fractals, such as the Sierpiński carpet



(a) Sierpiński carpet shown at level 4. (b) Sierpiński triangle shown at level 6.

Figure 2.9 : Shape-based fractals approximated with fractal trees.

and the Sierpiński triangle, can be approximated with fractal trees (Fig. 2.9). For the Sierpiński carpet, we use one robot to represent each square (and the robot may have the ability to paint the square). Each node has 8 branches, but these branches have different scale factors from the scale factor in the parent node. Furthermore, scale factors alternate in every level and all the squares must be oriented along the same directions. We also allow the user to override some helper functions and attach an event handler to state change in order to build more complicated fractals.

For a Sierpiński carpet, we define even branches as the branches in the forward direction, and these branches increment at right angles (0°, 90°, 180°, 270°). Odd branches are those branches with an angle of 45° to even branches (45°, 135°, 225°, 315°). We set the number of branches to 8 and override GetAngle to return these angles. Each robot u needs to count how many times u enters an odd branch in $u.counter$, and this counter increments whenever $u.state$ changes to $EXPAND$ and $u.branch$ is odd. Finally, we override GetScale as in Algorithm 6.

Limited sensing range and motion errors. Even when the robots have a limited

Algorithm 6 SierpinskiCarpet::GetScale(u)

```
1: scale ← 1/3
2: if u.branch is odd then
3:     if u.counter is odd then
4:         scale ← scale × √2
5:     else
6:         scale ← scale/√2
7:     end if
8: end if
9: return scale
```

sensing range and even when the robots have motion errors, the robots can still form the correct fractal topology if the maximum length of edge d satisfies the following condition (in any Cartesian coordinate system):

$$\begin{aligned} \forall a, b \in \{(x,y) | (d-\Delta d)^2 < x^2 + y^2 < (d+\Delta d)^2, \\ \tan(\theta - \Delta\theta) < \frac{y}{x} < \tan(\theta + \Delta\theta)\} \Rightarrow \|a, b\| < r. \end{aligned} \tag{2.1}$$

This condition means that if two robots try to reach the same destination (d, θ) but separate due to motion error (Δd, $\Delta\theta$), these robots will fall within each other's sensing range r when both believe that they have reached their destination. In this case, if a robot moves into a branch that is already established, the robot can always discover the next vertex robot and become its child.

Collision avoidance. To implement this algorithm on real robots, we need to keep the robots at a safe distance from each other in order to avoid collisions. We pack the robot swarm into a dense formation near the initial robot, and we run the algorithm in [5] so that the robots enter the workspace one after another without becoming disconnected.

Once a robot moves along edges in the fractal tree, we define a set of routes around the edges so that robots only move sequentially on these routes (Fig. 2.10). There

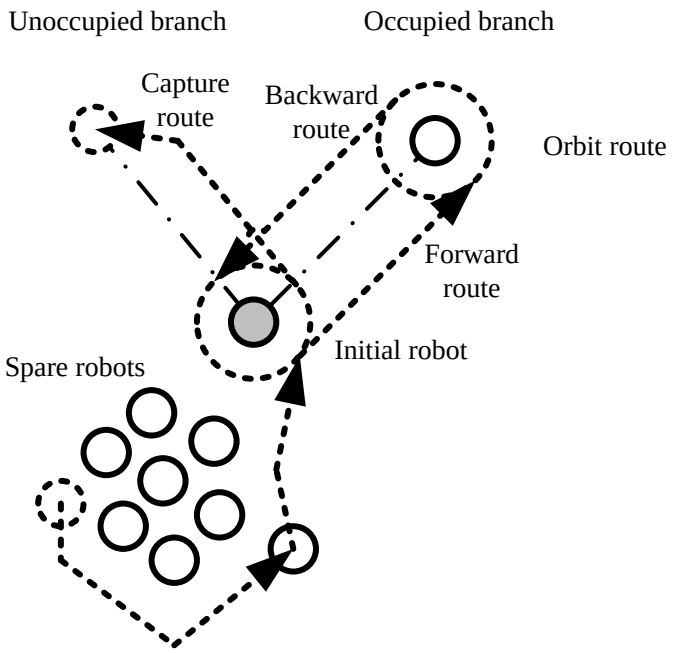
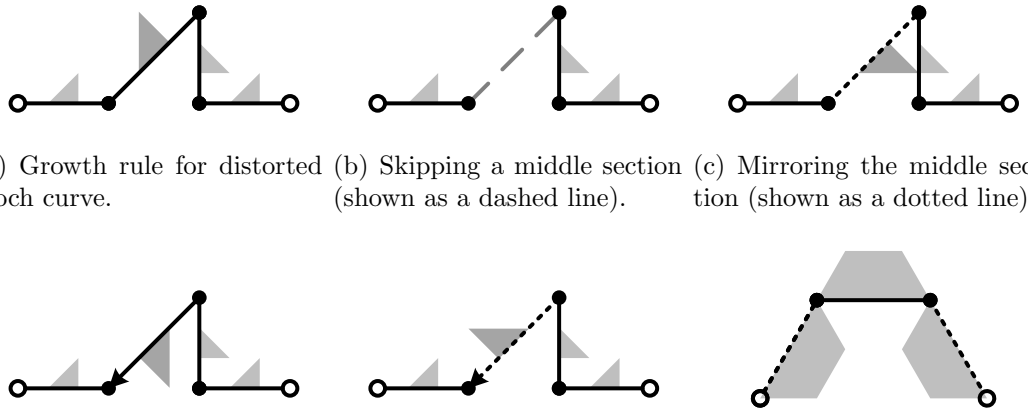


Figure 2.10 : Routes for collision avoidance.

are four types of routes with descending priority:

- Capture route (robot becomes a new vertex),
- Backward route (robot returns to its parent due to an inactive branch or for other reasons),
- Orbit route (circular connection between other routes),
- Forward route (to reach a higher level).

Robots in a route with a lower priority must yield to robots with a higher priority, as well as to any robot in front of the robot. If a robot encounters an oncoming robot or is about to collide with a vertex robot due to motion errors, the robot must yield to the right to avoid a collision.



(a) Growth rule for distorted Koch curve.

(b) Skipping a middle section (shown as a dashed line).

(c) Mirroring the middle section (shown as a dotted line).

(d) Reversing the middle section (shown as an arrow).

(e) Flipping (mirroring and reversing) the middle section.

(f) Growth rule for Sierpiński arrowhead curve.

Figure 2.11 : Growth rules for curve-based fractals.

If the fractal to be built has enough clearance between branches at the desired level, the robots will be collision-free and can organize themselves automatically into the fractal.

2.3.2 Curve-based Fractals

Curve-based fractals are those fractals formed by a polygonal chain, such as the Koch Curve [49]; in particular, those fractals that are continuous and do not have branches. This category can be expanded to fractals that self-intersect, such as the Lévy C curve [50], but can still be represented as an Eulerian path. This category can be further expanded to fractals that are not continuous, such as the Cantor set [51], but are subsets of a continuous curve. Space-filling curves will be discussed in a later section because the iteration rule for space-filling curves requires interfaces between components.

We describe the base case by a line segment with two vertices, and for the growth

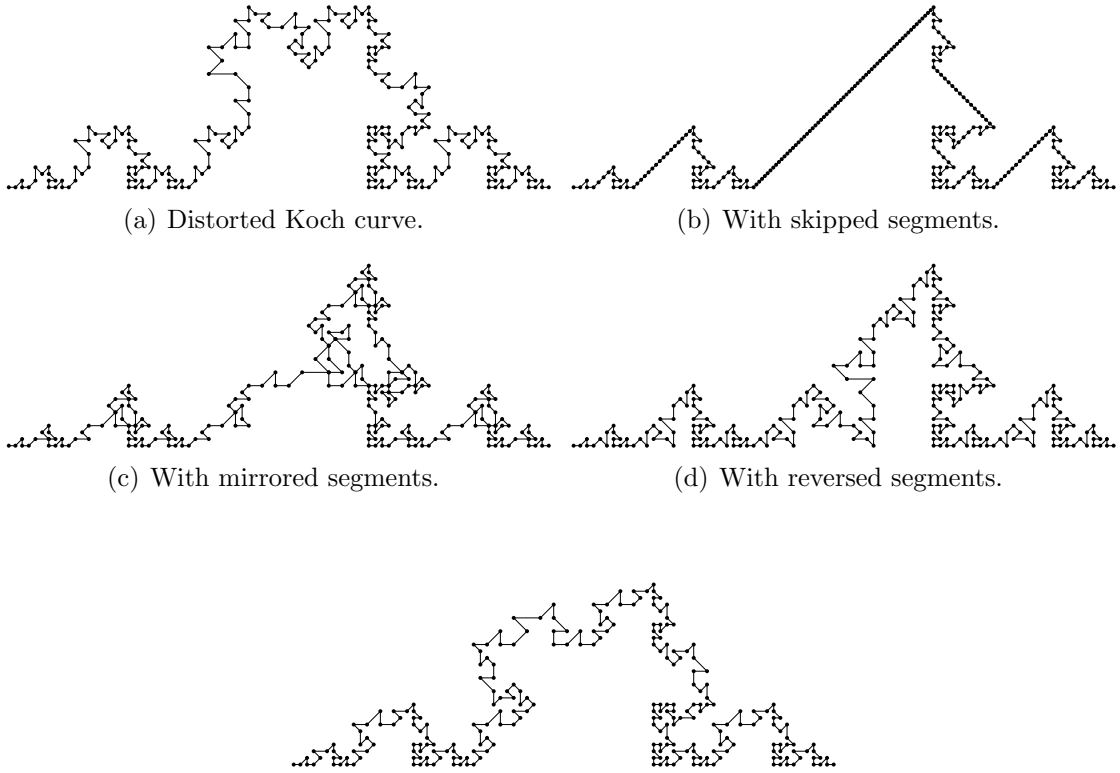
rule we replace the line segment with a polygonal chain (Fig. 2.11(a)). We denote the number of vertices in the growth rule (including both endpoints) by $t+1$, and the level of iterations by k. For a perfect fractal with n vertices, we should have $n = t^k + 1$. The base shape with two vertices and one line segment has $k = 0$ and $n = 2$.

Suppose we already have n robots placed along a line, and each robot knows both of its adjacent neighbors in opposite directions (except for the two robots at each end which have only one adjacent neighbor). The linear ordering can be achieved with chain-formation algorithms [52] [53] [54] [55], or with physical sorting algorithms [56] [57] [58] [59]. From some of these algorithms each robot also knows its topological distances to one end h and to the other end h'; thus all the robots know the total number of robots $n = h + h' + 1$.

Algorithm 7 GetSequence$(h, begin, end, k)$

```
1: if end − begin ≥ k then
2:     step ← ⌊(end − begin)/k⌋
3:     if h − begin ≡ 0 (mod step) then
4:         return (h − begin)/step
5:     else
6:         return GetSequence(h, ⌊(h − begin)/step⌋, ⌈(h − begin)/step⌉, k)
7:     end if
8: else
9:     Not enough robots to form a new level. The robot simply moves to the midpoint of the closest
       two neighbors.
10:    return ∅
11: end if
```

To generate curve-based fractals with a robot swarm, we first describe the iterating shape as a set of vectors. The iterating shape maps to $t+1$ vectors, where we denote the first vector by the zero vector $(0, 0)$ and the last vector by the unit vector $(1, 0)$. Other vectors have their own scales of the unit vector; for example, the three middle vectors describing the Koch curve are $(\frac{1}{3}, 0)$, $(\frac{1}{2}, \frac{\sqrt{3}}{2})$, and $(\frac{2}{3}, 0)$. Then each robot is assigned its level and sequence in the fractal. Since each robot knows the total



(a) Distorted Koch curve.

(b) With skipped segments.

(c) With mirrored segments.

(d) With reversed segments.

(e) With flipped (mirrored and reversed) segments.

Figure 2.12 : Curve-based fractals (shown at level 4).

number of robots n and its relative position h along the line, the robots can run an iterated function (Algorithm 7) locally to determine its sequence as well as the closest two robots on the previous level.

If each robot has a sensing range large enough to cover at any time the two closest robots on the previous level, each robot can move to the relative position defined by its corresponding vector. When all the robots move in the same manner (except the two robots on each end that remain stationary), the line of robots transforms progressively into a curve-based fractal.

In order to build more complicated curve-based fractals, we introduce some prop-

erties attached to each line segment in the growth rule. *Skip* determines whether a line segment is skipped instead of replaced (Fig. 2.11(b)), and can be used for erasing the line segments in a Cantor set. A customized GetSequence may be provided to avoid assigning robots into such void zones. *Mirror* determines whether the replacement rule is mirrored over the original line segment (Fig. 2.11(c)). *Reverse* determines whether the replacement rule is rotated 180 degrees (Fig. 2.11(d)). Mirror and reverse can be combined into a *flip* for the same line segment (Fig. 2.11(e)). The corresponding fractals are shown in Fig. 2.12.

Some shape-based fractals can be approximated with curve-based fractals. For example, the Sierpiński triangle can be constructed using a Sierpiński arrowhead curve. The vectors for constructing a Sierpiński arrowhead curve are:

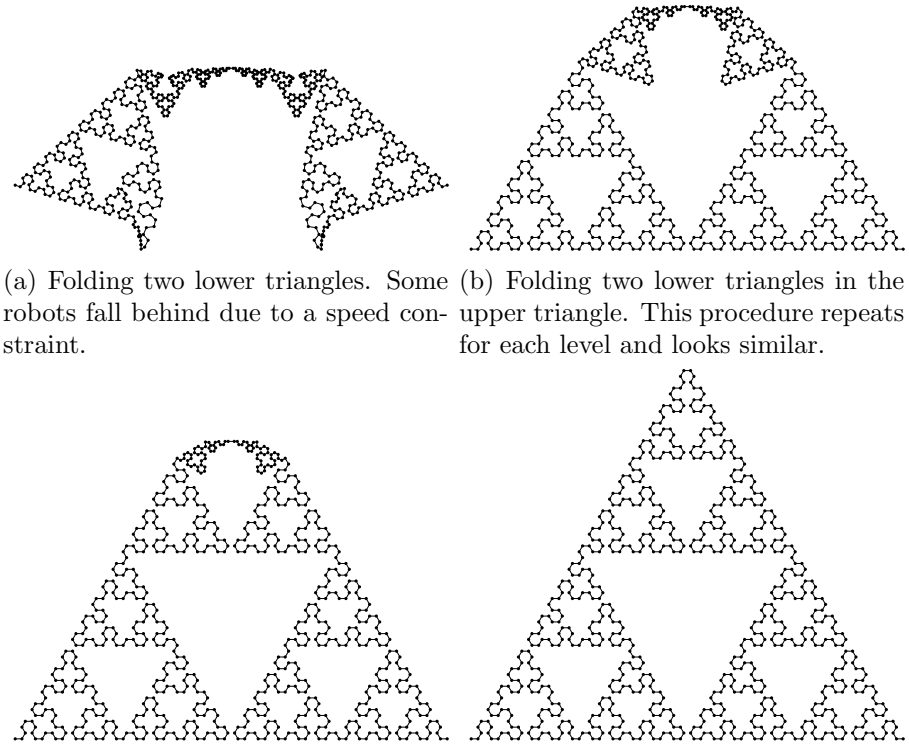
$v_0 = (0, 0)$, $v_1 = (\frac{1}{4}, \frac{\sqrt{3}}{4})$, $v_2 = (\frac{3}{4}, \frac{\sqrt{3}}{4})$, $v_3 = (1, 0)$.

The line segments (v_0 to v_1) and (v_2 to v_3) are mirrored (Fig. 2.11(f)). When the program starts, the lower two triangles fold inward until their destination, while the other robots for the upper triangle move above. Then the upper triangle gets constructed with a similar procedure. The process for building this fractal (Fig. 2.13) is a fractal!

This algorithm requires some robots to have a longer sensing range than other robots. This assumption is still practical since the swarm can have different robots with different costs, and still keep the total cost low. Removing the requirement of long sensing range is possible, but may introduce accumulated errors.

2.3.3 Space-filling Curves

Space-filling curves are more difficult to generate than curve-based fractals because of the existence of interfaces (Fig. 2.14). *Interfaces* are the line segments that connect



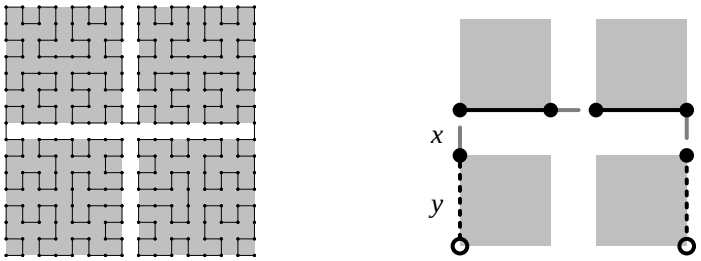
(a) Folding two lower triangles. Some robots fall behind due to a speed constraint.

(b) Folding two lower triangles in the upper triangle. This procedure repeats for each level and looks similar.

(c) Folding in a higher level.

(d) Final result.

Figure 2.13 : Procedure for building the Sierpiński arrowhead curve (shown at level 6).



(a) The Hilbert space filling curve shown at level 3.

(b) Growth rule.

Figure 2.14 : The Hilbert space filling curve contains 4 similar squares and 3 interfaces.

adjacent sub-fractals. At a given level these interfaces usually have the same size as the shortest line segments, which depend on the number of levels and the number of robots used to build the space-filling curve. Therefore, the number of levels must be determined in advance, and the ratio of interfaces to sub-fractals must be predetermined.

For the Hilbert curve, we use an algorithm similar to the algorithm for a curve-based fractal, but we define the vectors based on the highest level $l_{\max}$ and the current level l. The ratio of the interface to the size of the fractal at level l is $x = \frac{1}{2^{l_{\max}-l+2}-1}$, and we denote by $y = \frac{1-x}{2}$ the ratio of the sub-fractal to the fractal. Then vectors are calculated at each level by setting:

$v_1 = (0, y)$, $v_2 = (0, y + x)$, $v_3 = (y, y + x)$, $v_4 = (y + x, y + x)$, $v_5 = (1, y + x)$, $v_6 = (1, y)$.

To avoid assigning robots inside the interfaces, we also mark the interface line segments as skipped, using a customized assignment function. Then we mirror the first and last sub-fractals.

This method applies to space filling curves that have start and end points distinct from each other. An alternative method is to pack the robots into a square first [60], and then assign the virtual edges between robots.

2.4 Simulation Results

All the algorithms discussed in this chapter are implemented with our Multi-Robot Simulator written in C#. Running on a Windows laptop (1.8GHz CPU) and using 25MB memory, this simulator can generate the Sierpiński arrowhead curve up to level 8 (with 6,562 robots) in a two-minute animation with smooth motion. All the fractal figures in this chapter are exported directly from this simulator in SVG format.

2.5 Conclusion

We presented a distributed method that allows swarm robots to self-assemble into fractals. Starting from a dense swarm, robots move into fractal trees and fractal curves, and our algorithm adapts to different fractals by setting a small set of parameters. We validated our algorithm with simulations using thousands of robots.

Chapter 3

Stability Analysis of Distributed Algorithms

3.1 Introduction

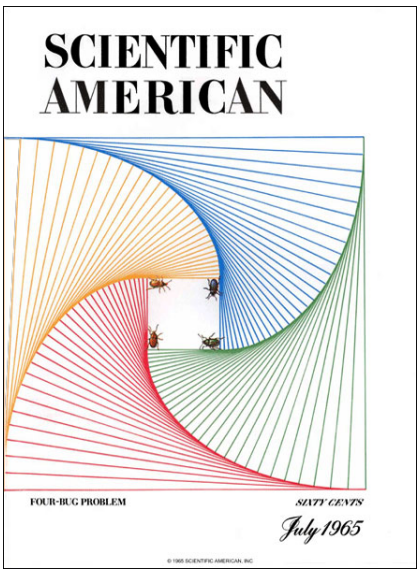
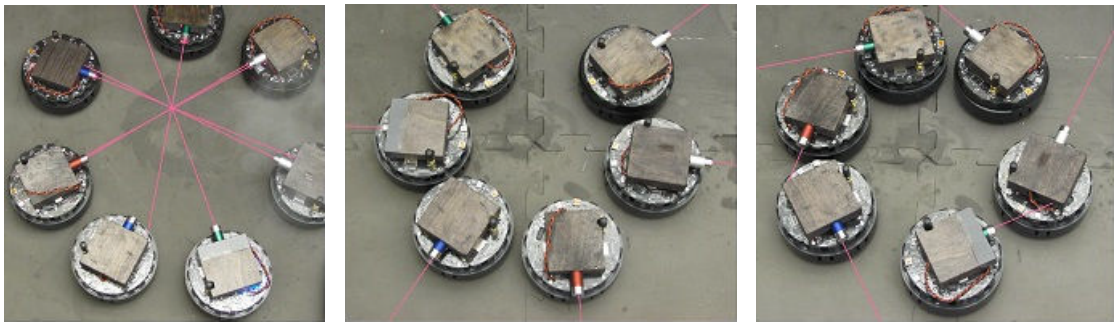


Figure 3.1 : The cover of Scientific American July 1965.

Scientific American [61] introduced the four-bug problem in July 1965 (Figure 3.1). Four bugs start at the corners of a square and chase each other moving clockwise at a speed proportional to their distance from the bug they are chasing. The trajectory of each bug forms a spiral. How far does each bug walk before all the bugs gather at the center?

In this problem, we observe that these bugs get closer to each other instead of moving along a circle. In addition, by symmetry these bugs always lie on the vertices of a square until they meet at the center; the square rotates clockwise and shrinks



(a) Attack. (b) Defend. (c) Form a perimeter.

Figure 3.2 : Robots attack, defend, and form a perimeter [1].

while the bugs move, and the sequence of these squares forms a whirl [62].

We can implement this behavior of shrinking a perimeter using robots, which already have the ability of following in a queue [63], clustering [64], dispersing [65], flocking [66], and sorting [58]. Shrinking the perimeter provides the robots the ability to guard an area and to narrow down the area while keeping the border free of penetration, in order to encircle and suppress enemies (Figure 3.2) [14].

When we generalize this problem for various directions of robot motion, including in the reverse direction, some unexpected strange behaviors occur. The reverse motion does not exactly follow the reverse extension of the forward motion, because moving in reverse is unstable for some configurations. For a similar reason, tractor-trailers can jackknife if not backed up properly or if uneven forces are applied to the wheels [67] [68], and jackknife becomes severe if these conditions continue to occur. With multiple robots interacting with each other, these behaviors sometimes end up in chaos.

This chapter focuses on why chaotic behaviors appear in some configurations and not in other configurations. We provide multiple perspectives on how errors accumulate and interfere with each other. We focus our analysis on several special

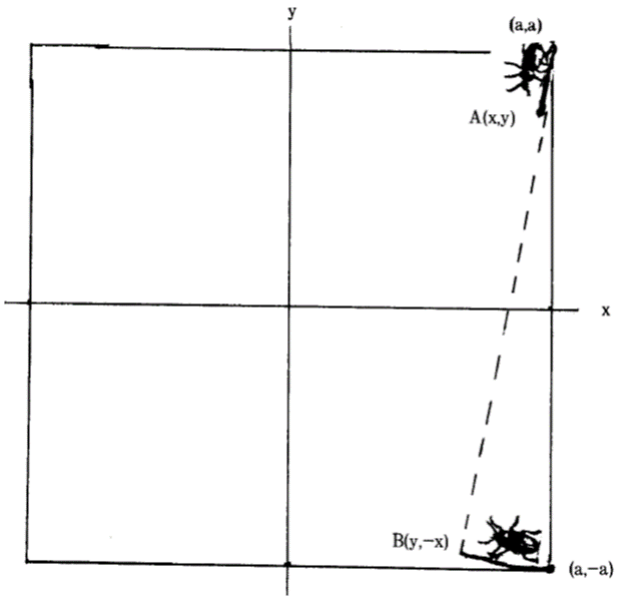
directions, and then reach a general description to classify all the directions into three categories: counterclockwise/clockwise, inward/outward, and stable/unstable. We also provide solutions to convert unstable trajectories into similar, but stable trajectories. With our simulation software, we draw the trajectories of the robots to demonstrate stable and unstable motions, and verify our solution to maintain stability while arranging robots in desired patterns.

We are also interested in robots expanding the perimeter at a constant rate during each revolution, like following the Archimedean spiral [69], which allows robots to seamlessly examine the ground while expanding the perimeter. There is some related work about generating Archimedean spirals using a single robot so that the robot can vacuum the floor [70] or detect landmines [71]. Our work enables a new kind of swarm behavior to generate approximate Archimedean spirals, where multiple robots move along different spiral arms that provides higher efficiency of collaboration and prevents single point of failure.

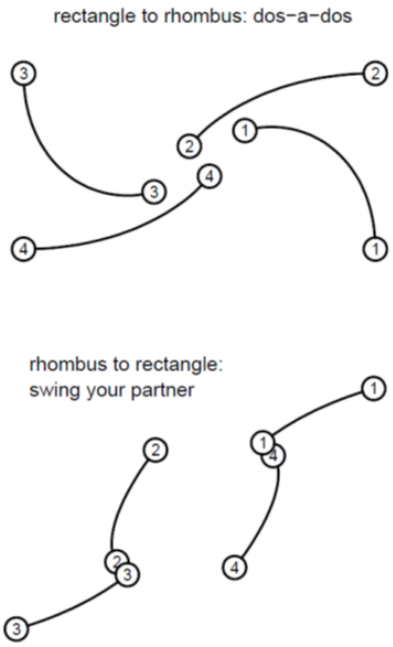
3.2 Related Work

Four-bug problem. There are many different versions of the four-bug problem in geometry books and mathematical puzzles. Other animals (such as mice, beetles, dogs) and vehicles are also used to name the moving entities. The first known publication of this problem is by Gardner [76] in 1957. Walfram MathWorld has a collection of these problems under the topic Mice Problem [77].

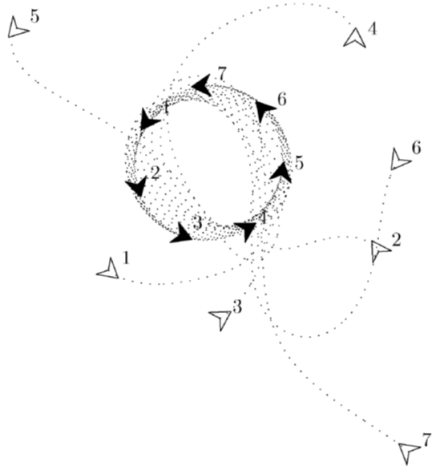
Peters [72] solves the four-bug problem with the geometric constraint that the tangents of adjacent bugs are always perpendicular to each other (Figure 3.3(a)). Thus, a differential equation describes the derivative of the slope so that the time parameter can be avoided. This method can be extended to any number of bugs starting at the



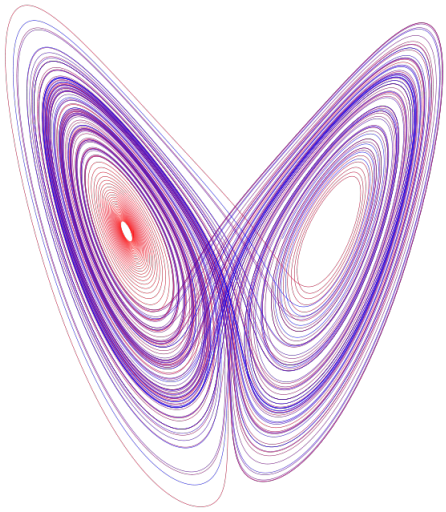
(a) Image from Peters's solution [72].



(b) Robots change formation between rectangles and rhombuses [73].



(c) Pursuit of nonholonomic vehicles [74].



(d) Lorenz attractor [48] (image from Wikipedia [75]).

Figure 3.3 : Images from related work.

vertices of a regular polygon, although the differential equation becomes complicated due to a non-perpendicular but fixed angle between tangents.

Klamkin and Newman [78] proved that three bugs starting from the vertices of any arbitrary non-degenerate triangle meet at the same time at a point uniquely determined by the initial positions of the bugs. Behroozi and Gagnon [79] later proved that mutual capture also happens when more bugs are present. Nester [80] provides a good demonstration of how the different parameters - height, side lengths, and angles - change the position of this center point.

Bruckstein, Cohen, and Efrat [81] investigated differential equations and matrices with both continuous and discrete solutions, and discussed how constant and varying speeds affect their solutions.

There are also some articles about the chasing problem in general. Bernhart studies pursuit curves in many scenarios [82] [83] [84]. Good [85] generates mathematical art with pursuit curves. Nahin [86] provides several interesting puzzles in his book about the pursuit and evasion problems.

Stability of swarm robotics and formation control. There have been several articles about the stability of swarm algorithms on different problems or with different methods.

Chapman, Lottes, and Trefethen [73] study a generalized version of this problem in which four bugs start from the vertices of a rectangle or a parallelogram (Figure 3.3(b)), and mention that instability over some parameters could lead to chaos.

Marshall, Broucke, and Francis [74] [87] solve for the trajectories of nonholonomic vehicles (using limited steering angle) from arbitrary configurations with matrices and complex numbers, and analyze the local stability of these trajectories (Figure 3.3(c)).

Liu, Passino, and Polycarpou [88] expand stability analysis into higher dimensions

with a fixed communication topology, enabling stable control of robotic aircraft. Gazi and Passino [89] provide a similar analysis for multi-dimensional swarm aggregations. Gazi also develops stability analysis for robot swarms in his Ph.D. dissertation [90].

Stability of dynamical systems. We are inspired by the stability analysis of dynamical systems. In particular, nonlinear dynamical systems can display unpredictable behaviors called *chaos*, which seem random but are intrinsically deterministic.

The Lorenz attractor [48] is a well-known chaotic dynamical system. The solution of the Lorenz equations describes motion in a butterfly pattern around two attractors (Figure 3.3(d)), but which attractor is subsequently surrounded is unpredictable and highly sensitive to the initial configuration.

There are many books discussing stability of dynamical systems and differential equations, such as [91] [92] [93] [94] [95].

Formation control. In formation control people try to maintain the stability of a swarm against errors and obstacles by applying control theory. Here is some related work that I studied in my Master's thesis [59].

Desai, Ostrowski, and Kumar [96] created a feedback model so that multiple robots can maintain their relative positions while moving and avoiding obstacles.

Poduri and Sukhatme [97] develop an algorithm to deploy a mobile sensor network with maximum area coverage, and robots are connected with any user-defined degree of connection. This algorithm can generate a connected uniform distribution when all the robots stop.

Zavlanos and Pappas [98] Williams and Sukhatme [99] use a potential field to keep robots connected and follow the leader, and maintain a similar organized structure in motion. Zavlanos, Jadbabaie, and Pappas [100] also keep robots connected when flocking, using a fully distributed algorithm without a leader.

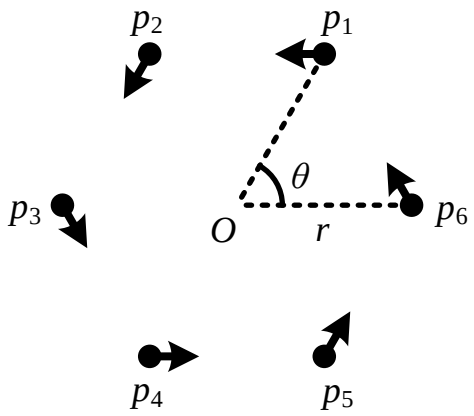


Figure 3.4 : An initial configuration of the robot pursuit problem with 6 robots in the counterclockwise arrangement.

Hsieh et. al. [101] and Zhang et. al. [102] keep robots connected to each other while moving in an environment with obstacles. The robots maintain appropriate distances and relative angles to minimize the risk of disconnection caused by obstacles.

Egerstedt and Hu [103] provide a mathematical model for multi-agent formation control, and apply their model to rigid body constrained motions.

Ji and Egerstedt [104] keep robots connected during rendezvous maneuvers by adding appropriate weights to the connections.

3.3 The Robot Pursuit Problem

We define the robot pursuit problem as follows. Given n robots whose positions are $p_1(t), p_2(t), ..., p_n(t)$ at time t, lying initially on the vertices of a regular polygon oriented in the counterclockwise direction at time $t = 0$, each robot p_k moves towards p_{k+1} (and robot p_n moves towards p_1) at a speed proportional to their distance. Find the trajectories of all the robots (Figure 3.4).

To solve this problem, first we write linear differential equations to describe each robot's velocity vector (ignoring constants):

$$\frac{\mathrm{d}p_k}{\mathrm{d}t} = p_{k+1} - p_k, \tag{3.1}$$

with these initial conditions:

$$p_{kx}(0) = r_0 \cos\left(\frac{2\pi k}{n}\right),\ p_{ky}(0) = r_0 \sin\left(\frac{2\pi k}{n}\right). \tag{3.2}$$

In these equations, robots move at a common but non-constant speed proportional to the distance between adjacent robots. Solving these differential equations analytically yields the following solutions:

$$p_{kx}(t) = r_0 \mathrm{e}^{-bt} \cos\left(\frac{2\pi k}{n} + t\right), \tag{3.3}$$

$$p_{ky}(t) = r_0 \mathrm{e}^{-bt} \sin\left(\frac{2\pi k}{n} + t\right). \tag{3.4}$$

b is a constant to be determined. Introducing polar coordinate (r_k, θ_k), we find that the functions p_{kx} and p_{ky} satisfy $r_k = r_0 \mathrm{e}^{-b\theta_k}$, $\theta_k \geq 0$ for every $k \in \{0, ..., n-1\}$. Thus the trajectory of each robot is a counterclockwise inward logarithmic spiral [105]. From the logarithmic spiral it can also be verified that each robot always maintains a constant angle $\phi = \frac{n-2}{2n}\pi$ (half of the internal angle of regular n-gon) between its heading and the direction to the origin [105].

Let

$$\arctan\frac{1}{b} = \phi = \frac{n-2}{2n}\pi. \tag{3.5}$$

Solving Equation (3.5), we find the constant

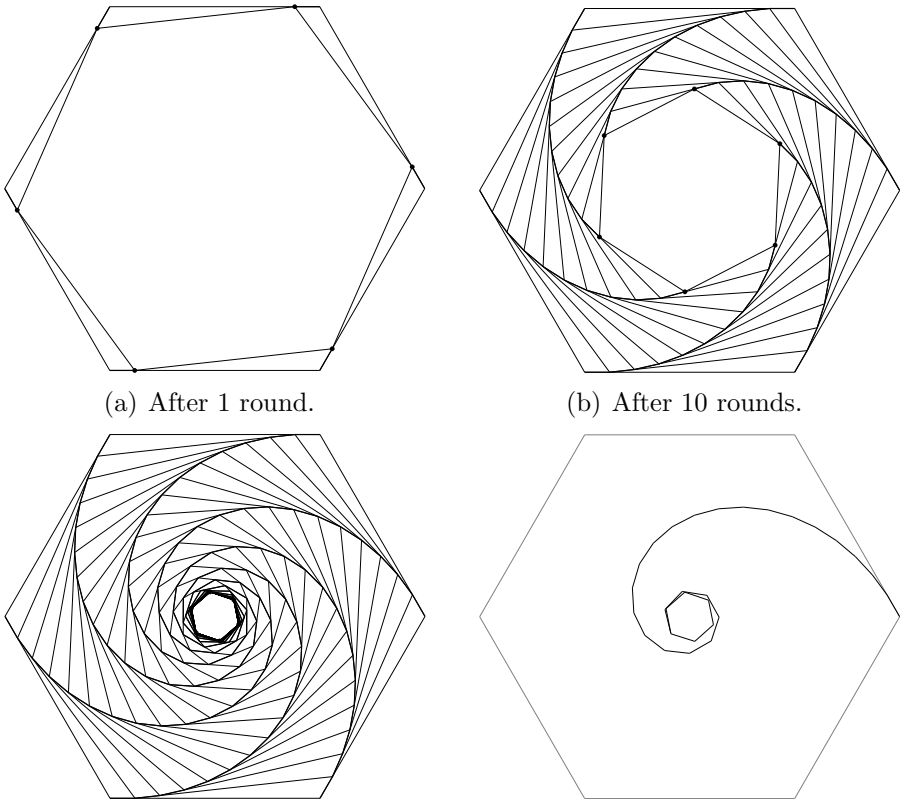
$$b = \cot\left(\frac{n-2}{2n}\pi\right). \tag{3.6}$$

If, instead, the robots move at a constant speed, the robot pursuit problem can be described by the following differential equations (ignoring constants):

$$\frac{\mathrm{d}p_k}{\mathrm{d}t} = \frac{p_{k+1} - p_k}{\|p_{k+1} - p_k\|}. \tag{3.7}$$

Although the speed is constant, these differential equations are nonlinear: the x and y coordinates of the robots interact with each other, making an analytic solution impractical. However, due to the symmetry of this problem and restrictions between the x and y coordinates, the trajectories of the robots are exactly the same as those with proportional speeds. Lacking analytic solutions, we focus this chapter on developing numerical algorithms to solve these problems in swarm robotics, especially with simulation software.

On actual robot platforms, robots can take measurements of the other robots' positions and communicate periodically with other robots. The movement between communication rounds is independent of the position change of any other robots during these rounds, because the robot cannot be aware of the changes until taking new measurements at the next round. Software simulations are designed to generate frames of animation; thus the movement of robots is discrete. We treat these frames as rounds, and model the motion between rounds as line segments. We also define a step size s that each robot can move between two consecutive rounds. If a robot intends to move to a destination with distance smaller than s, the robot reaches its destination at the next round; otherwise the robot moves towards its destination for



(a) After 1 round. (b) After 10 rounds.

(c) After 30 rounds. (d) The trajectory of a single robot.

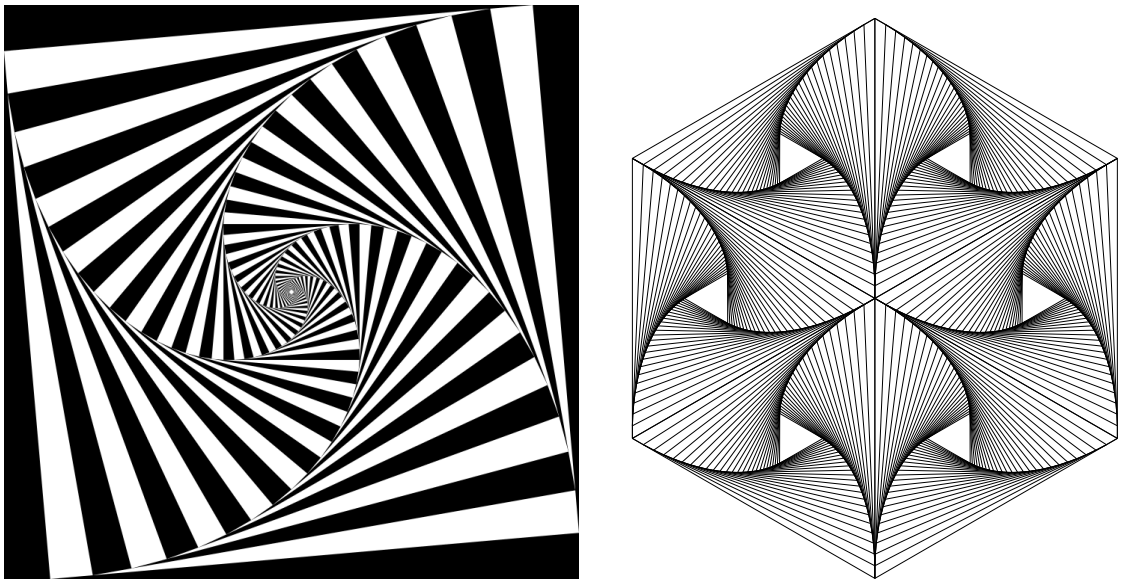
Figure 3.5 : Simulation of the robot pursuit problem with 6 robots.

a distance of s along a straight line.

In the discrete motion model, we describe robot motion by the following difference equation:

$$p_k(t+1) - p_k(t) = \min\left(\frac{p_{k+1}(t) - p_k(t)}{\|p_{k+1}(t) - p_k(t)\|}s, p_{k+1}(t) - p_k(t)\right). \tag{3.8}$$

Solving these difference equations numerically with software simulation generates a set of line segments as in Figure 3.5. Note that the robots cannot reach the center of the polygon. Instead, the robots end up on the vertices of a smaller polygon with edge length less than or equal to s. When the edge length of the polygon is less



(a) Op art from the whirl of 4 robots filled with black and white (image from Wikipedia [106]).

(b) Six whirls form an artistic pattern (redrawn from [72]).

Figure 3.6 : Artistic patterns with whirls.

than or equal to s, each robot can jump to the next vertex in a single round and thus the spiral generated is not complete. Also note that Figure 3.5(d) is different from a recursively generated logarithmic spiral [26]: a recursively generated spiral has edges with decreasing lengths and constant angle, while the robot's trajectory has line segments with (almost) constant length and various angles.

The whirls formed by the robot pursuit problem can be used for artistic design (Figure 3.6(a)). Combining 6 instances of the robot pursuit whirls with 3 robots generates a beautiful artistic pattern (Figure 3.6(b)) [72].

3.4 The Robot Evasion Problem

We can reverse the robot pursuit problem by reversing the direction of each robot. Each robot p_k now moves in the opposite direction away from p_{k+1} (and robot p_n

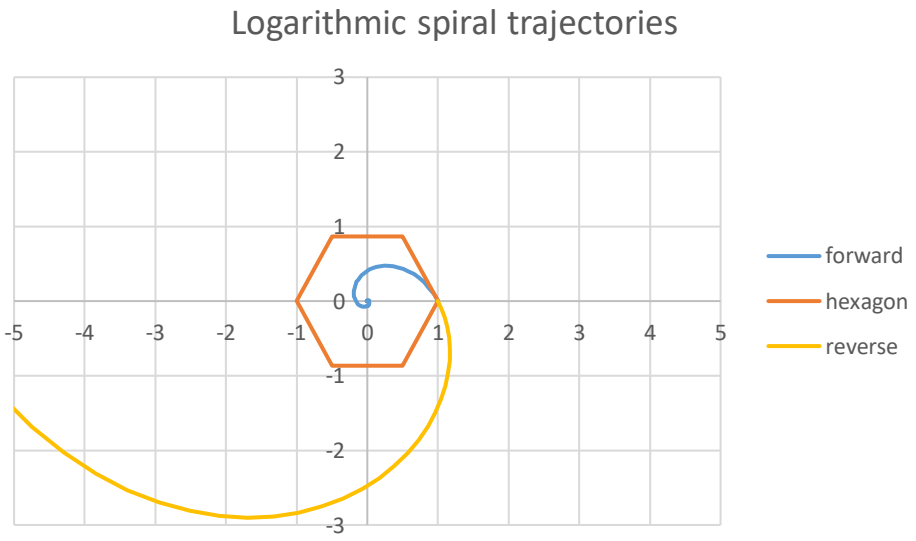


Figure 3.7 : The inward spiral and the outward spiral connect at a vertex and are tangent to an edge at that vertex.

moves away from p_1). Thus the velocity of each robot is given by the differential equation

$$\frac{\mathrm{d}p_k}{\mathrm{d}t} = -(p_{k+1} - p_k). \tag{3.9}$$

These differential equations look similar to the previous ones in Equation 3.1, but with a negative sign. The solution is also similar except for the negative signs

$$p_{kx}(t) = r_0 \mathrm{e}^{bt} \cos\left(\frac{2\pi k}{n} - t\right), \tag{3.10}$$

$$p_{ky}(t) = r_0 \mathrm{e}^{bt} \sin\left(\frac{2\pi k}{n} - t\right), \tag{3.11}$$

$$b = \cot\left(\frac{n-2}{2n}\pi\right). \tag{3.12}$$

The solution is an outward clockwise logarithmic spiral that expands to infinity.

Drawing the forward spiral and the reverse spiral in Figure 3.7, we observe that

these spirals connect at a vertex of the polygon (the robot), and both spirals are tangent to the edge pointing to another robot which the robot intends to chase or to avoid. If we allow $t \in (-\infty, \infty)$, the robot moves smoothly in both directions and the motion is reversible.

The iterations for the corresponding discrete motion model are defined by the following difference equation:

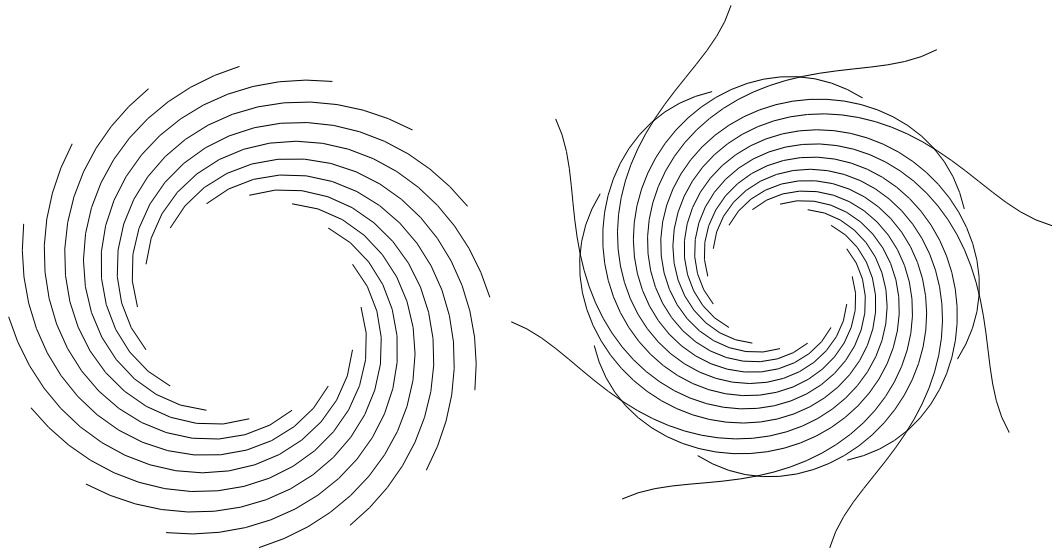
$$p_k(t+1) - p_k(t) = -\frac{p_{k+1}(t) - p_k(t)}{\|p_{k+1}(t) - p_k(t)\|} s. \tag{3.13}$$

Simulating the robot evasion problem in software, we see at first that the robots expand the perimeter of the polygon. However, after some time, the robots start to move in chaotic directions. In Figure 3.8 with 16 robots, the **g16** symmetry is lost but an approximate **g8** symmetry is preserved [107]. In Figure 3.9 with 13 robots, no symmetry is preserved from the less symmetric initial configuration.

Our experiments show that any of the following factors speed up the descent into chaos:

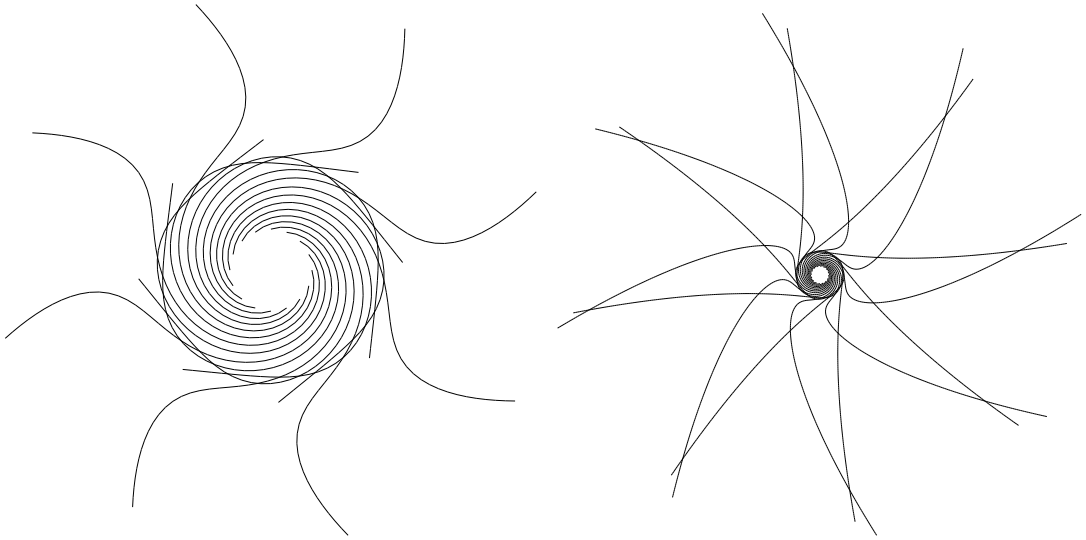
- Place more robots on the vertices of a regular polygon.
- Increase the speed or step size of the robots.
- Introduce error in the speed or direction of the robots.
- Place the robots on the vertices of a non-regular polygon.

Chaos results because errors accumulate and amplify in the reverse problem. Our investigations reveal that chaos comes from uncontrollable increments in the angle between neighbors (Figure 3.10(a)). If this angle increases during iterations and



(a) After 18 rounds, the trajectories are spirals with **g16** symmetry.

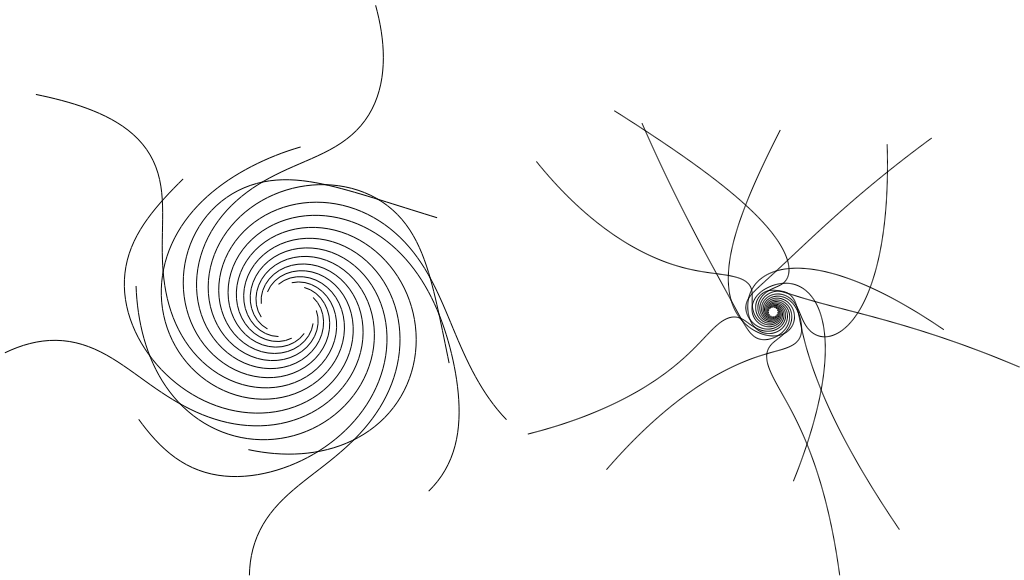
(b) After 36 rounds, the differences in the density between spiral arms become obvious. Some trajectories bend towards the inside, while other trajectories escape from the center.



(c) After 50 rounds, those trajectories bent inside leave at tangent directions.

(d) After 160 rounds, trajectories form a flower shape with an approximate **g8** symmetry.

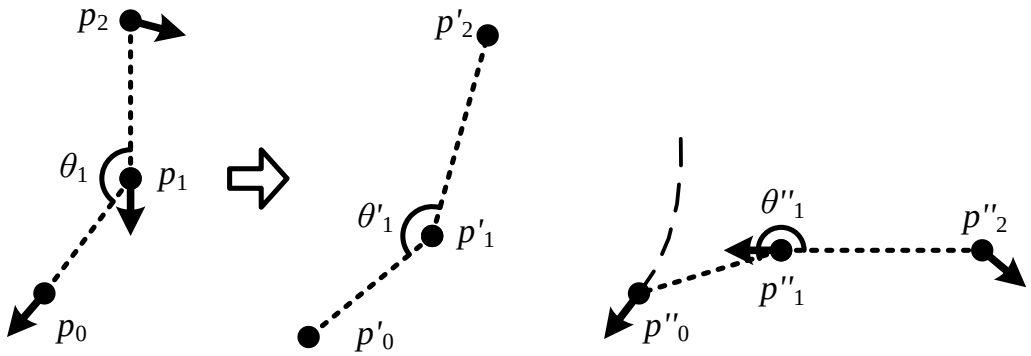
Figure 3.8 : Simulation of the robot evasion problem with 16 robots. (trajectories are drawn to different scales)



(a) After 90 rounds.

(b) After 300 rounds.

Figure 3.9 : Simulation of the robot evasion problem with 13 robots. (trajectories are drawn to different scales)



(a) The angle θ_1 increases during iterations ($\theta'_1 > \theta_1$ until θ'_1 becomes a reflex angle.

(b) Once the angle θ''_1 is above the straight angle, p''_1 moves in a direction where the trajectories intersect.

Figure 3.10 : Chaos results from trajectory intersections which are driven by uncontrollable increments in the angles.

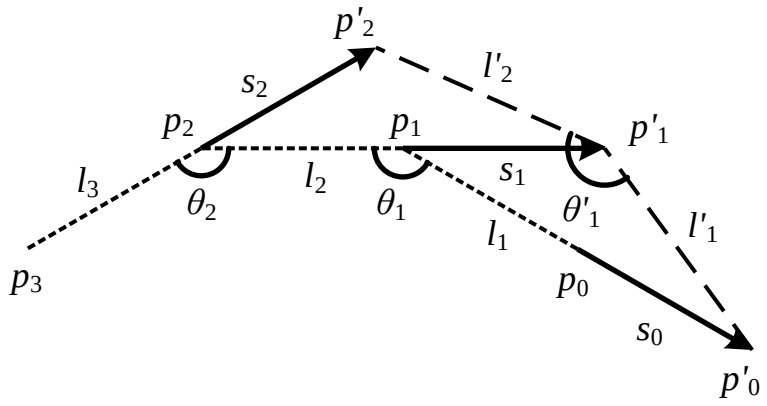


Figure 3.11 : The geometry in the robot evasion problem.

becomes a reflex angle, the vertex robot moves in a direction towards another robot's trajectory (Figure 3.10(b)), causing trajectories to intersect.

Here we explain the influence of accumulated errors by introducing two kinds of controllable errors: direction error and distance error.

In Figure 3.11, several adjacent robots in counterclockwise order are denoted by p_0, p_1, p_2, p_3. Distances between adjacent robots are denoted by l_1, l_2, l_3, and angles between neighbors are denoted by θ_1, θ_2. Robots p_0, p_1, p_2 move toward p'_0, p'_1, p'_2 in the next round by moving at step sizes s_0, s_1, s_2. The distances between adjacent robots after these steps are denoted by l'_1, l'_2, and the angle between neighbors of p_1 after these steps is denoted by θ'_1.

From the law of cosines

$$l'_1 = \sqrt{s_1^2 + (l_1 + s_0)^2 - 2s_1(l_1 + s_0)\cos(\pi - \theta_1)}, \tag{3.14}$$

$$l'_2 = \sqrt{s_2^2 + (l_2 + s_1)^2 - 2s_2(l_2 + s_1)\cos(\pi - \theta_2)}, \tag{3.15}$$

and from the law of sines

$$\theta_1' = \arcsin\left(\frac{l_1 + s_0}{l_1'}\sin(\pi - \theta_1)\right) + \arcsin\left(\frac{s_2}{l_2'}\sin(\pi - \theta_2)\right). \tag{3.16}$$

Note that the arcsines in Equation (3.16) may need to be resolved from an obtuse angle by subtracting from π, and whether the angle is acute or obtuse can be determined by applying the law of cosines to the corresponding angle.

If the robots are on the vertices of a regular polygon, and there are no errors in the directions or distances, then $l_1 = l_2$, $\theta_1 = \theta_2$, and $s_0 = s_1 = s_2$. Therefore the two triangles are congruent, $\triangle p_0' p_1' p_1 \cong \triangle p_1' p_2' p_2$, so $l_1' = l_2'$ and $\theta_1' = \theta_1$.

Now we introduce some error in the direction by setting $\theta_1 = \theta_2 + \Delta\theta$ while keeping $l_1 = l_2$ and $s_0 = s_1 = s_2$. If $\theta_1' > \theta_1$, a slightly larger angle generates an even larger angle in the next iteration. During subsequent iterations this angle may rise above a straight angle. Then the polygon is no longer convex and robot trajectories may intersect.

Calculating the derivative $\frac{\mathrm{d}\theta_1'}{\mathrm{d}\theta_1}$ analytically from Equation (3.16) results in a very complicated result. Therefore, we solve for this derivative numerically by enumerating θ_2 over a series of angles and setting $\Delta\theta = 1^\circ$.

In Figure 3.12, we measure the instability by the instability index $\frac{\theta_1' - \theta_1}{\Delta\theta}$, which measures whether a positive error in θ_1 is amplified or reduced after an iteration. To perform our simulations, we set the step size ratio $\frac{s}{l}$ to five different values. Our calculations show that:

- The crossover point (θ_2 where $\frac{\theta_1' - \theta_1}{\Delta\theta} = 0$) is smaller when the step size ratio $\frac{s}{l}$ is smaller, meaning that a smaller step size leads to instability starting from a smaller angle.

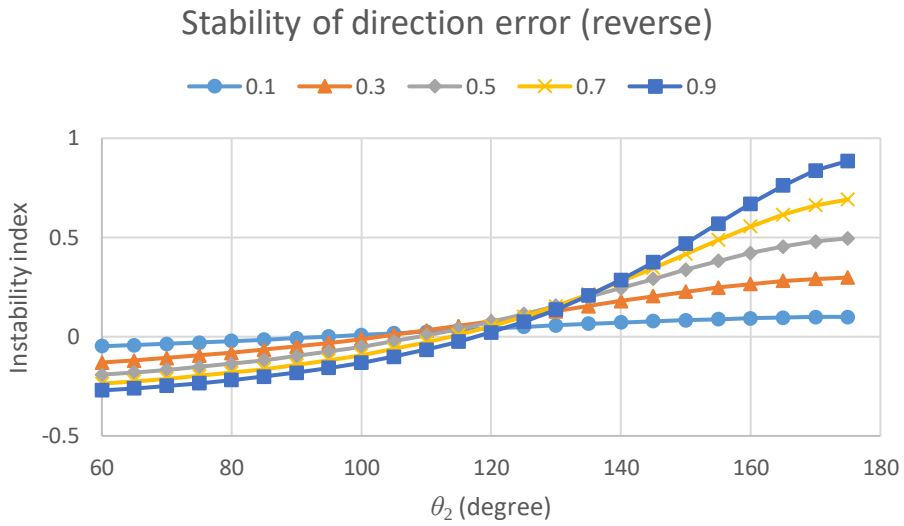


Figure 3.12 : Instability of the robot evasion problem due to direction error. A positive instability index means that direction error is amplified and the configuration is unstable. Five data series reflect different step size ratios $\frac{s}{l}$. Since we care about the largest internal angle of a convex polygon, we limit $\theta_2 \in [60°, 180°)$.

- With a larger step size ratio and an unstable configuration, chaos is fiercer since the direction error is amplified by a larger ratio.
- The crossover point has a limit near 90° when $\frac{s}{l} \to 0$, meaning that an angle smaller than 90° (such as those angles in a regular triangle) is always stable.
- If the step size s is fixed and is independent of the distance l, the step size ratio $\frac{s}{l}$ becomes smaller after each iteration; thus all angles above 90° are unstable. However, a smaller step size ratio amplifies the distance error at a smaller ratio, so the system may take a very large number of iterations to display chaotic behavior.

Also keep in mind that if $\theta_1 > \theta_2$, we always get $l'_1 < l'_2$, as in Figure 3.13. The distance in the next iteration is reduced because of a positive direction error in the

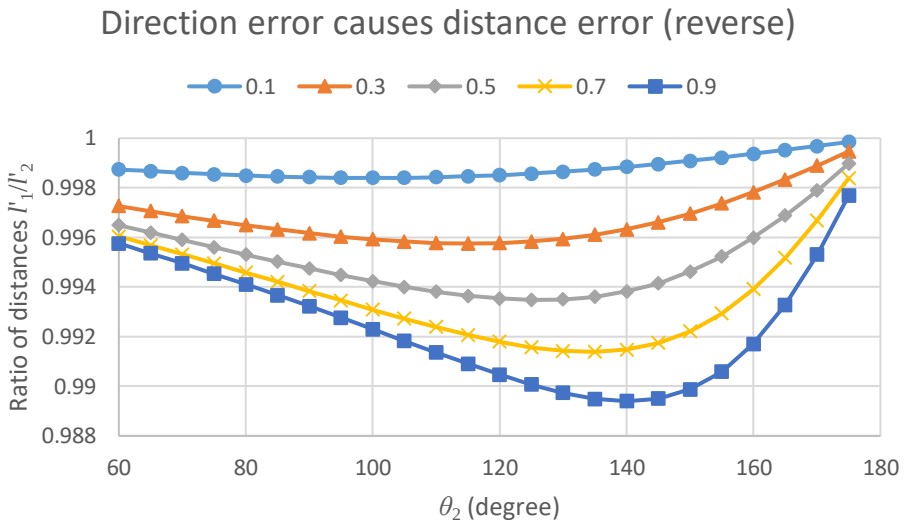


Figure 3.13 : Direction error generates distance error. Five data series reflect different step size ratios $\frac{s}{l}$, and in all cases $l'_1 < l'_2$.

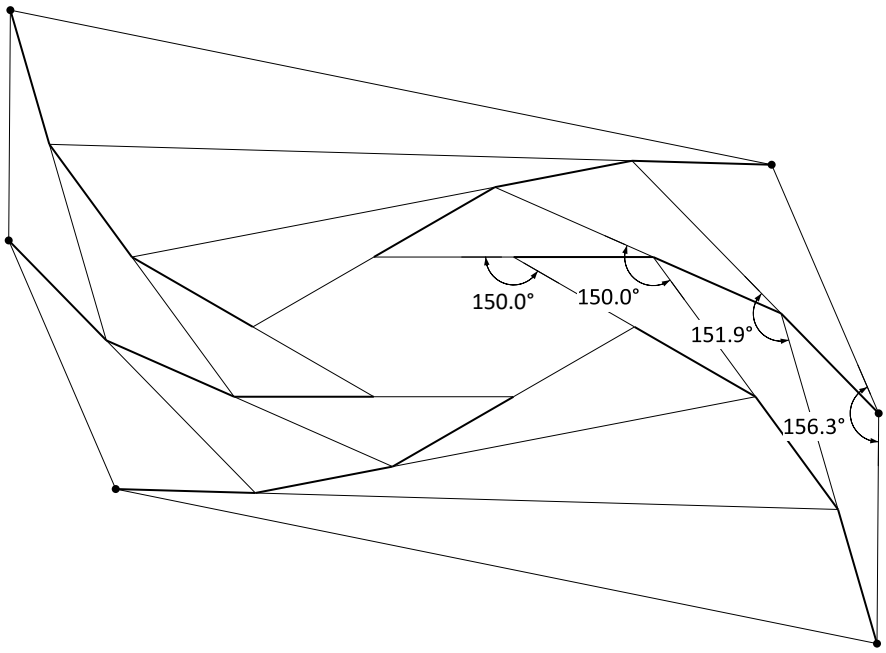
current iteration and the law of sines.

Figure 3.14 shows an example of a non-regular hexagon configured with the same initial lengths but different initial angles. Simulation shows that one of the largest internal angles increments above 180° causing trajectories to intersect. All the edge lengths of the hexagon increase before intersection but at different speeds.

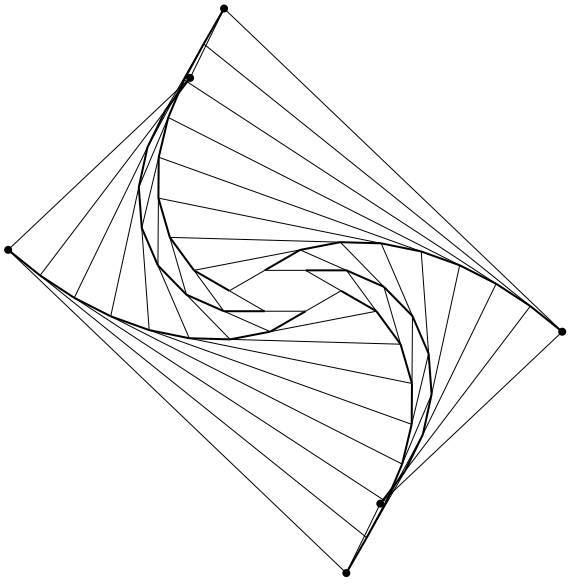
Next we discuss distance error. Figure 3.15 illustrates another example of a non-regular hexagon with the same initial angles but different initial lengths.

By invoking Equation (3.16) with $\theta_1 = \theta_2$ and $s_0 = s_1 = s_2$ but $l_1 \neq l_2$, we find that the distance error always decreases, and the distances between neighboring robots converge to a single distance. Figure 3.16 shows that the difference in the distances is always reduced. A smaller initial angle θ has a larger impact on reducing the difference of distances.

However, decrementing distance error introduces direction error at a relatively

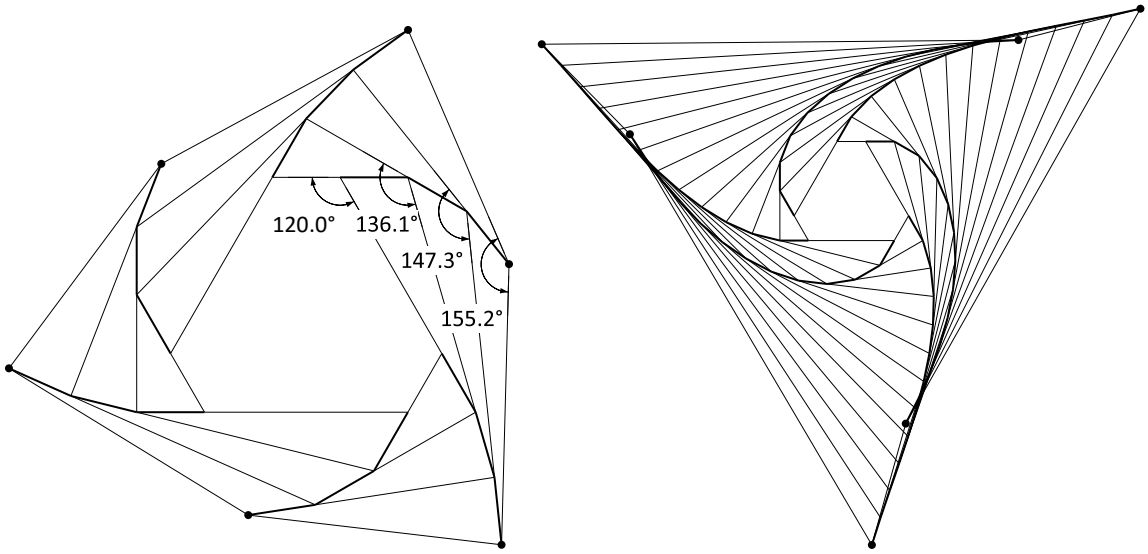


(a) The marked angle increases after each iteration, and the differences between distances increase.



(b) The trajectories of the robots intersect.

Figure 3.14 : The robot evasion problem with a flattened hexagon (same initial lengths but different initial angles).



(a) The marked angle increases after each iteration, and the differences between distances decrease.

(b) The trajectories of the robots intersect.

Figure 3.15 : The robot evasion problem with a deformed hexagon (same initial angles but different initial lengths).

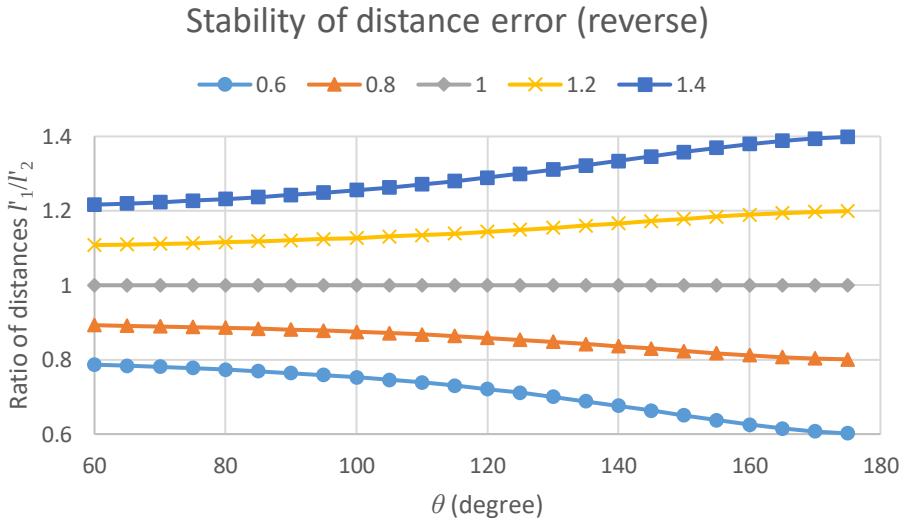


Figure 3.16 : The stability of distance error. Five data series reflect different distance ratios $\frac{l_1}{l_2}$, and the step size is set to $s = \frac{l_2}{2}$. The distance ratio in the next round $\frac{l'_1}{l'_2}$ is always closer to 1 than the initial ratio $\frac{l_1}{l_2}$ (except for the control group where $\frac{l_1}{l_2} = \frac{l'_1}{l'_2} = 1$).

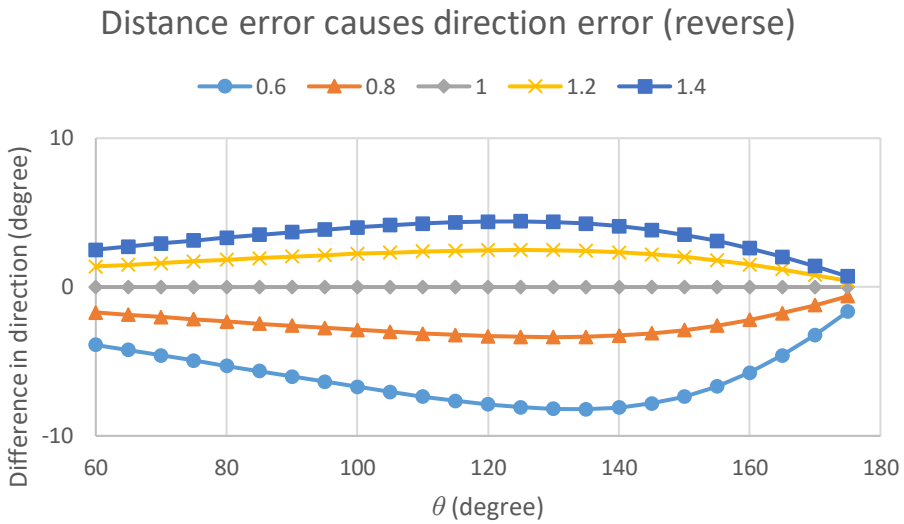


Figure 3.17 : Although distance error decreases, direction error is introduced. Five data series reflect different distance ratios $\frac{l_1}{l_2}$, and the step size is set to $s = \frac{l_2}{2}$.

large scale (Figure 3.17). θ_1' can either increase or decrease based on $\frac{l_1}{l_2}$, so different distances between neighboring robots change the angle between adjacent robots, resulting in the previous problem of direction error.

The final effect is the accumulation of the combination of these two geometric errors, which may either strengthen or cancel each other. Motion errors, such as deviation from the direction of travel or different step sizes, also result in these geometric errors. In computer systems, floating point errors are somewhat unpredictable but usually deterministic. Euclidean space becomes anisotropic, since each axis is represented by a different variable and subject to a separate floating point error. For example, if a robot moves only along the x-axis, no floating point error can apply to the y-axis since the variable for the y coordinate never changes; if the robot moves along the line $y = x$, the same error always applies to both the x and y coordinates because floating point error is deterministic for the same value. But if a robot pushes

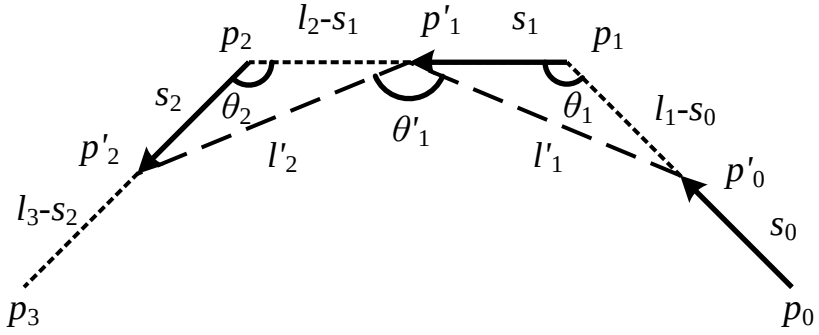


Figure 3.18 : The geometry in the robot pursuit problem.

another robot in a generic direction, floating point error may cause the robots to sway away from the line. Therefore the symmetry of the initial configuration may or may not be preserved during this chaotic procedure.

3.5 Stability of the Robot Pursuit Problem

In the previous section we analyzed why the robot evasion problem is unstable in most cases. But why does the chaotic behavior not appear in the original pursuit problem? Let's analyze the original problem with the same method to show why the robots are stable in the forward direction.

In Figure 3.18, from the law of cosines

$$l'_1 = \sqrt{s_1^2 + (l_1 - s_0)^2 - 2s_1(l_1 - s_0)\cos\theta_1}, \tag{3.17}$$

$$l'_2 = \sqrt{s_2^2 + (l_2 - s_1)^2 - 2s_2(l_2 - s_1)\cos\theta_2}, \tag{3.18}$$

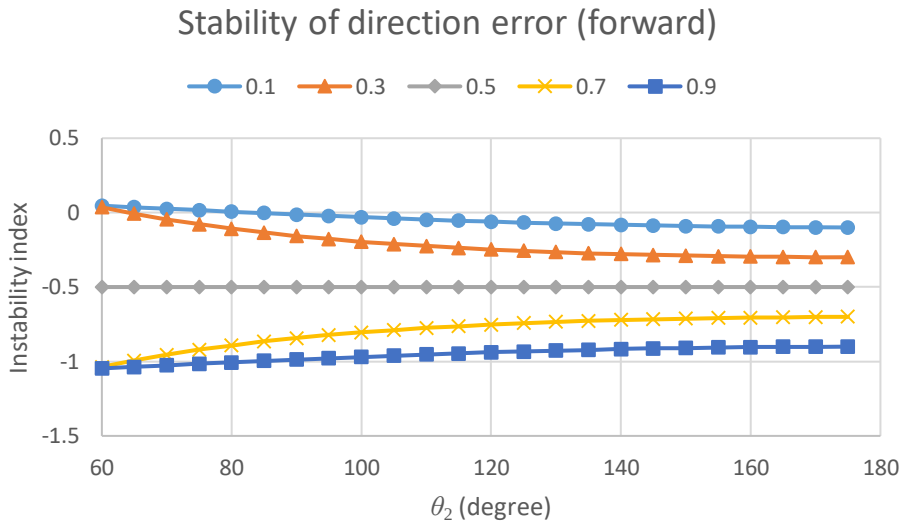


Figure 3.19 : The robot pursuit problem is mostly stable (negative instability index) except for $\theta_2 < 90°$ with small step size ratio. Five data series reflect different step size ratios $\frac{s}{l}$.

and from the law of sines

$$\theta_1' = \pi - \arcsin\left(\frac{l_1 - s_0}{l_1'}\sin\theta_1\right) - \arcsin\left(\frac{s_2}{l_2'}\sin\theta_2\right). \tag{3.19}$$

Note that the arcsines in Equation (3.19) may need to be resolved from an obtuse angle by subtracting from π, and whether the angle is acute or obtuse can be determined by applying the law of cosines to the corresponding angle.

By invoking Equation (3.19) with $\theta_1 = \theta_2 + 1°$, $s_0 = s_1 = s_2$, and $l_1 = l_2$, we get the instability index of the robot pursuit problem against direction error in Figure 3.19. Unlike the robot evasion problem, most configurations are stable to direction error. The only exception is that for a small step size ratio $\frac{s}{l} < \frac{1}{3}$, there exist some $\theta_2 < 90°$ with positive instability index. But their instability indexes $\frac{\theta_1' - \theta_1}{\Delta\theta}$ are very small (< 0.075) so that noticeable instability is never observed in simulations. Also

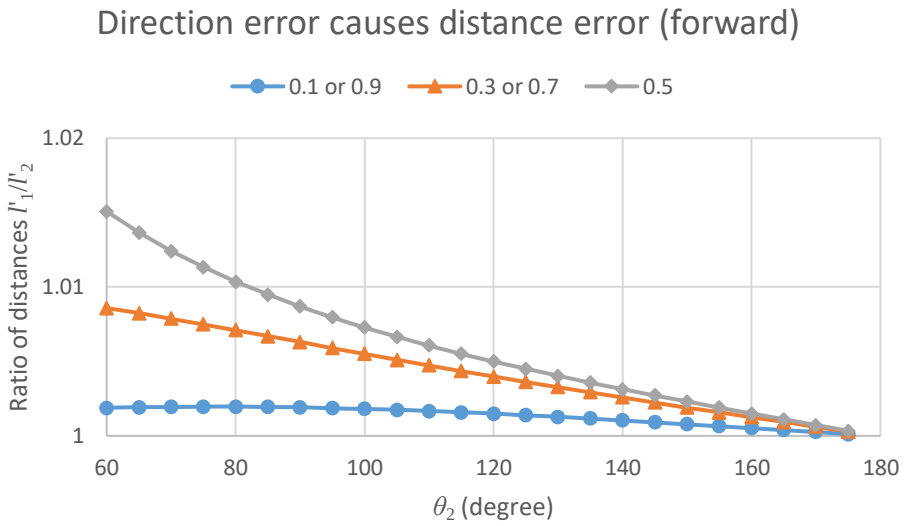


Figure 3.20 : Direction error generates distance error in the robot pursuit problem. Three data series reflect different step size ratios $\frac{s}{l}$, and show that series with step size ratio $(1-\frac{s}{l})$ have the same data points as series with step size ratio $\frac{s}{l}$.

note that when $\frac{s}{l} = \frac{1}{2}$, we always have $\theta'_1 = \frac{\theta_1+\theta_2}{2}$ regardless of θ_1 and θ_2; other pairwise step size ratios summing up to 1 (such as 0.1 and 0.9) have data points symmetric to $\frac{\theta'_1-\theta_1}{\Delta\theta} = \frac{1}{2}$ due to geometric symmetry.

Direction error generates distance error (Figure 3.20). We have $\frac{l'_1}{l'_2} > \frac{l_1}{l_2}$ under all circumstances.

For the distance error, we invoke Equation (3.19) with $\theta_1 = \theta_2$ and $s_0 = s_1 = s_2$ but $l_1 \neq l_2$. In Figure 3.21, $\frac{l'_1}{l'_2}$ is closer to 1 when $\frac{l_1}{l_2} < 1$, which is stable to distance error; but for $\frac{l_1}{l_2} > 1$, we have $\frac{l'_1}{l'_2} > \frac{l_1}{l_2} > 1$ which is an unstable configuration. The instability in the distance explains why the transitions between rectangles and rhombuses are difficult to predict in [73].

However, decrementing distance error introduces direction error at a relatively large scale (Figure 3.22). θ'_1 can either increase or decrease based on $\frac{l_1}{l_2}$, so different

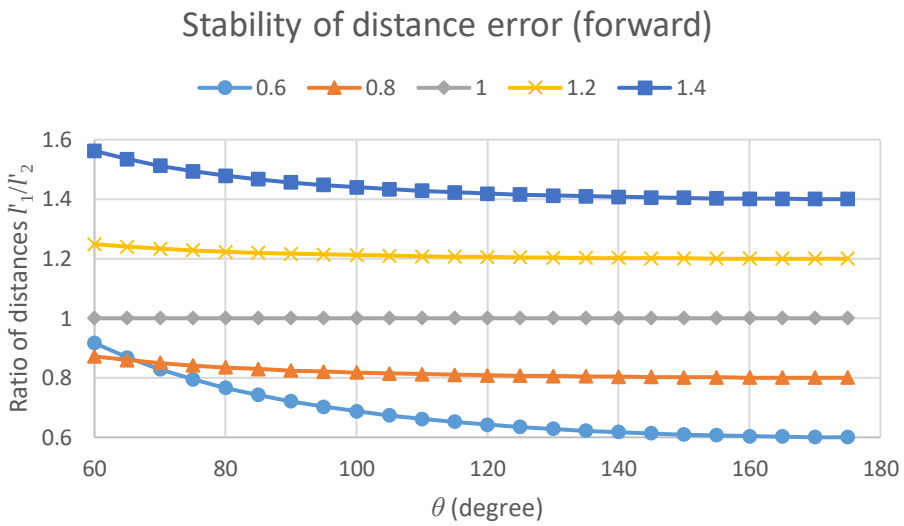


Figure 3.21 : The stability of distance error. Five data series reflect different distance ratios $\frac{l_1}{l_2}$, and the step size is set to $s = \frac{l_2}{2}$. Data series with $\frac{l_1}{l_2} < 1$ have $\frac{l'_1}{l'_2}$ closer to 1 (stable), and data series with $\frac{l_1}{l_2} > 1$ have $\frac{l'_1}{l'_2}$ farther away from 1 (unstable).

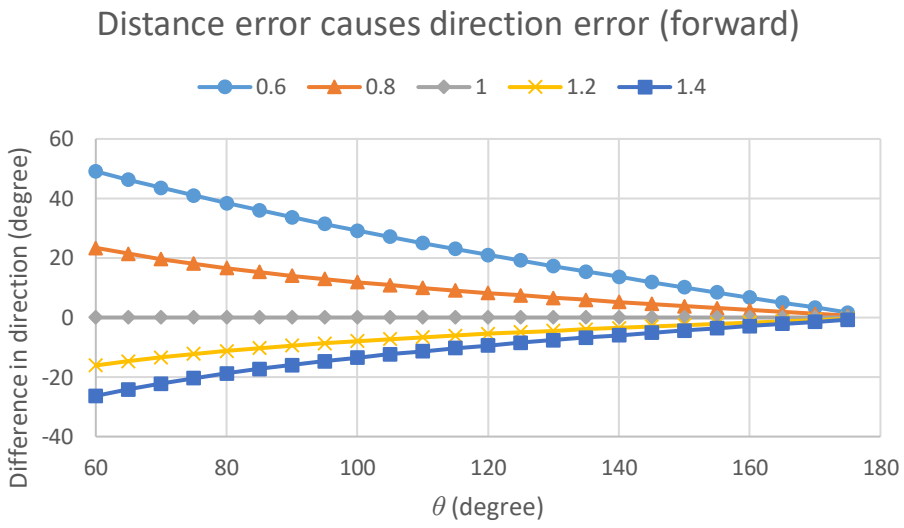


Figure 3.22 : Direction error is introduced by distance error (except for $l_1 = l_2$). Five data series reflect different distance ratios $\frac{l_1}{l_2}$, and the step size is set to $s = \frac{l_2}{2}$.

distances between neighboring robots change the angle between robots, resulting in the previous problem of direction error.

In the robot pursuit problem, we observe two types of feedback:

- Even though negative distance error ($l_1 < l_2$) generates positive direction error ($\theta_1' > \theta_1$), this phenomenon is very weak when $\theta \to 180°$ (since $\theta_1' - \theta_1 \to 0$), and cannot cause the angle to rise above 180°. On the other hand, direction error is stable so that large angles decrease during iterations.

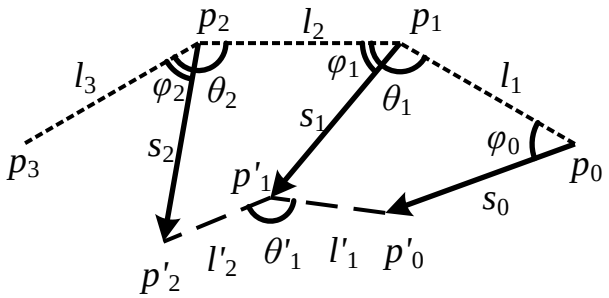
- The distance ratio $\frac{l_1}{l_2}$ tends to increase, either because of the distance error itself or due to errors introduced by direction error. But since the robots on the vertices of a polygon form a circular chain, $\prod_i \frac{l_i}{l_{i+1}} = 1$, so increasing $\frac{l_1}{l_2}$ must cause another ratio $\frac{l_i}{l_{i+1}}$ to decrease, and, as observed in Figure 3.21, a small ratio $\frac{l_i}{l_{i+1}}$ has higher stability preventing this ratio from further decrement.

Therefore, although instability exists in the robot pursuit problem and errors interfere with each other, feedback keeps the errors bounded and prevents chaotic behavior.

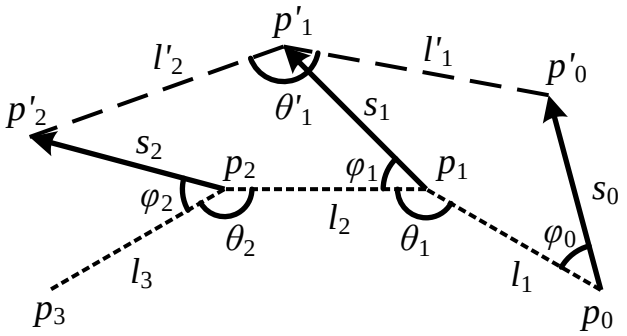
3.6 The General Problem of Stability

Here we introduce a general stability problem from the robot pursuit paradigm. Given n robots whose positions are $p_1(t), p_2(t), ..., p_n(t)$ at time t, initially lying on the vertices of a regular polygon oriented in the counterclockwise direction at time $t = 0$, where each robot p_k moves in the direction towards p_{k+1} plus an angle φ, determine whether this dynamic system is stable for the given parameters (n, φ).

For the discrete version of this problem, two general positions are shown in Figure 3.23. The previous sections discussed two special cases: the robot pursuit problem has $\varphi = 0$, and the robot evasion problem has $\varphi = \pi$.



(a) With positive φ.



(b) With negative φ.

Figure 3.23 : The geometry in the general problem.

For $\varphi > 0$, from the law of sines and the law of cosines

$$x_2 = \overline{p_1' p_2} = \sqrt{l_2^2 + s_1^2 - 2 l_2 s_1 \cos\varphi_1}, \tag{3.20}$$

$$\angle p_1 p_2 p_1' = \arcsin\left(\frac{s_1}{x_2}\sin|\varphi_1|\right), \tag{3.21}$$

$$\angle p_1' p_2 p_2' = \theta_2 - \varphi_2 - \angle p_1 p_2 p_1', \tag{3.22}$$

$$\angle p_1 p_1' p_2 = \arcsin\left(\frac{l_2}{x_2}\sin|\varphi_1|\right), \tag{3.23}$$

$$l_2' = \sqrt{x_2^2 + s_2^2 - 2 x_2 s_2 \cos\angle p_1' p_2 p_2'}, \tag{3.24}$$

$$\angle p_2 p_1' p_2' = \arcsin\left(\frac{s_2}{l_2'}\sin\angle p_1' p_2 p_2'\right), \tag{3.25}$$

$$x_1 = \overline{p_0' p_1} = \sqrt{l_1^2 + s_0^2 - 2 l_1 s_0 \cos\varphi_0}, \tag{3.26}$$

$$\angle p_0 p_1 p_0' = \arcsin\left(\frac{s_0}{x_1}\sin|\varphi_0|\right), \tag{3.27}$$

$$\angle p_0' p_1 p_1' = \theta_1 - \varphi_1 - \angle p_0 p_1 p_0', \tag{3.28}$$

$$l_1' = \sqrt{x_1^2 + s_1^2 - 2 x_1 s_1 \cos\angle p_0' p_1 p_1'}, \tag{3.29}$$

$$\angle p_0' p_1' p_1 = \arcsin\left(\frac{x_1}{l_1'}\sin\angle p_0' p_1 p_1'\right), \tag{3.30}$$

$$\theta_1' = 2\pi - \angle p_1 p_1' p_2 - \angle p_2 p_1' p_2' - \angle p_0' p_1' p_1. \tag{3.31}$$

All the angles mentioned above are within the range $(0, \pi)$ except for φ, and the arcsines may need to be resolved from an obtuse angle by subtracting from π (whether the angle is acute or obtuse can be determined by applying the law of cosines to the corresponding angle).

These equations rely on the fact that $p_0 p_1 p_1' p_0'$ and $p_1 p_2 p_2' p_1'$ are non-degenerate

convex quadrilaterals. If any angle is equal to 0 or π, or any line segment has zero length, or any two line segments are collinear, some triangles may become degenerate and need to be handled specially, since division by zero may occur or some angles may become undefined. Concave and complex quadrilaterals also need special handling and will be discussed later.

For $\varphi < 0$, absolute values need to be taken when φ participate in the sines to avoid negative angles. We also need to subtract Equations (3.22), (3.28), and (3.31) from 2π to resolve these angles correctly.

Now we enumerate a few special values of φ.

Forward (pursuit counterclockwise). We have discussed the original robot pursuit problem in Section 3.5 when $\varphi = 0$.

Reverse (evasion clockwise). We have discussed the robot evasion problem in Section 3.4 when $\varphi = 180°$.

Radial motion. If the robots select an appropriate direction φ, the robots can enlarge or shrink the polygon perimeter without clockwise or counterclockwise rotation (Figure 3.24). From the geometry of the polygon, we have

$$\varphi_{\mathbf{i}} = \frac{\theta}{2} = \frac{n-2}{2n}\pi, \tag{3.32}$$

$$\varphi_{\mathbf{o}} = \varphi_{\mathbf{i}} - \pi = -\frac{n+2}{2n}\pi, \tag{3.33}$$

where n is the number of the vertices.

From Figure 3.25, we find that the outward direction is stable (negative instability index), and the inward direction is unstable (positive instability index).

The stability of outward motion is verified with simulation in Figure 3.26, where the initial polygon configuration has the same edge lengths but different angles. The

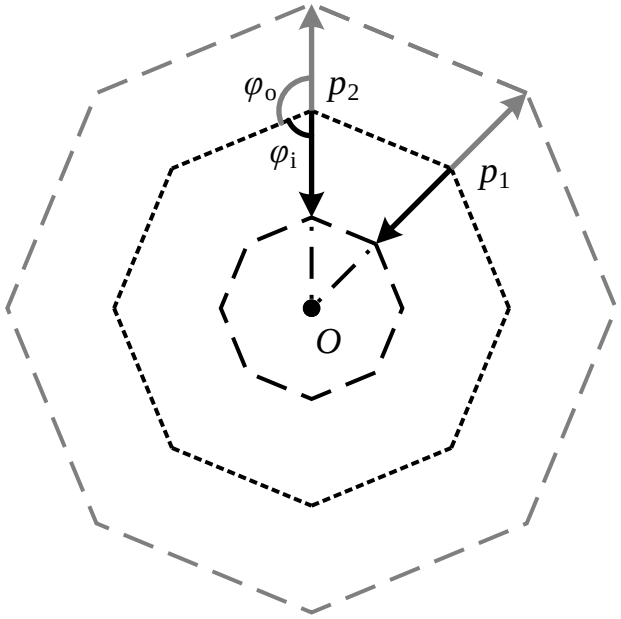


Figure 3.24 : The motion angle $\varphi_{\mathbf{i}}$ allows the robots to move radially inward, and the angle $\varphi_{\mathbf{o}}$ allows the robots to move radially outward.

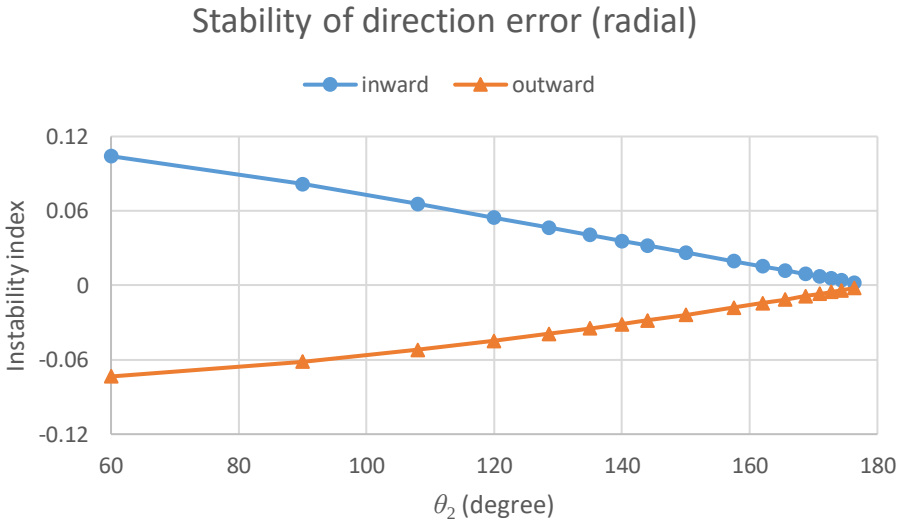
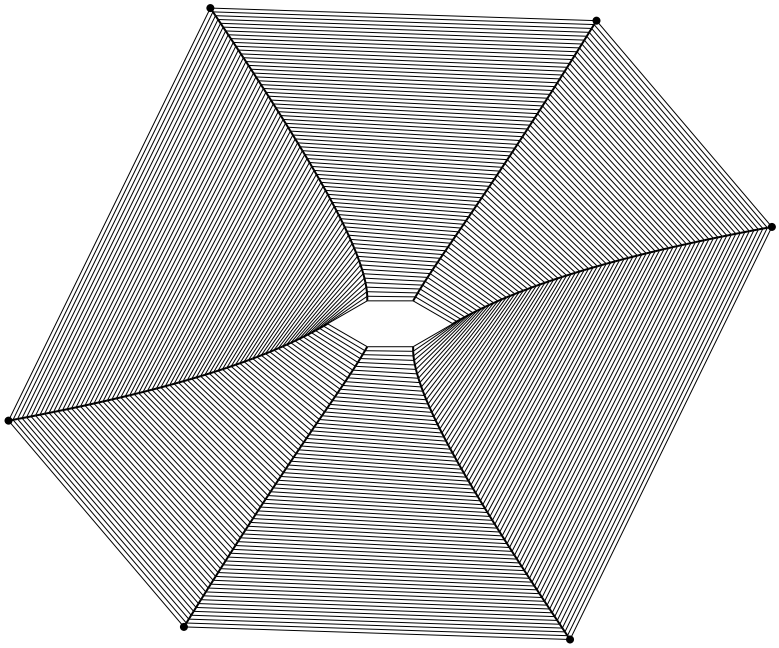
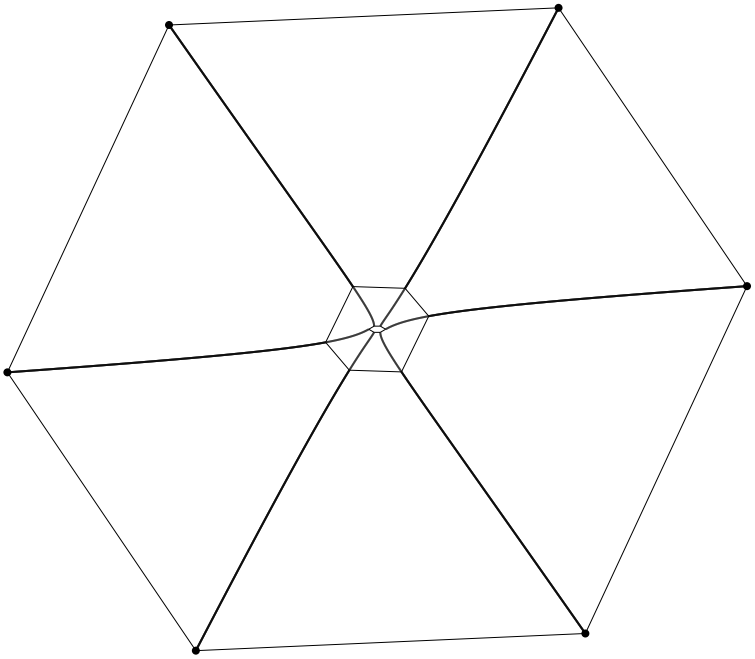


Figure 3.25 : The inward direction has positive instability index (unstable), and the outward direction has negative instability index (stable). Step size ratio $\frac{s}{l}$ is set to 0.1.

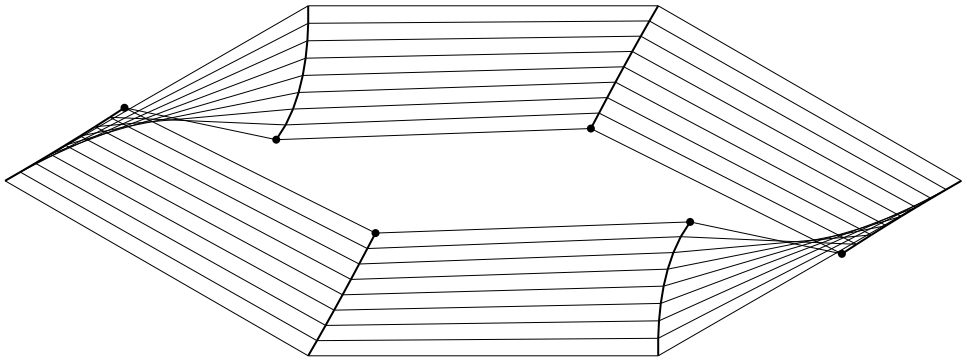


(a) In the first stage, some distance error is introduced while fixing the direction error.

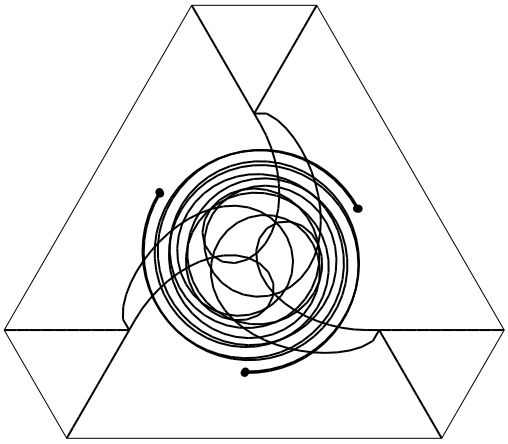


(b) In the second stage, both distance error and direction error decrease.

Figure 3.26 : Outward radial motion is stable, even if the initial configuration is distorted.



(a) By keeping $\varphi = 60°$, two robots move out of the initial polygon, and change the convex polygon into a concave polygon.

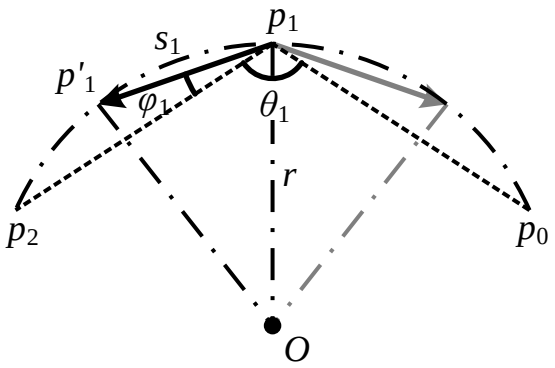


(b) Some robots meet before reaching the center, causing their trajectories to intersect and the polygon becomes complex.

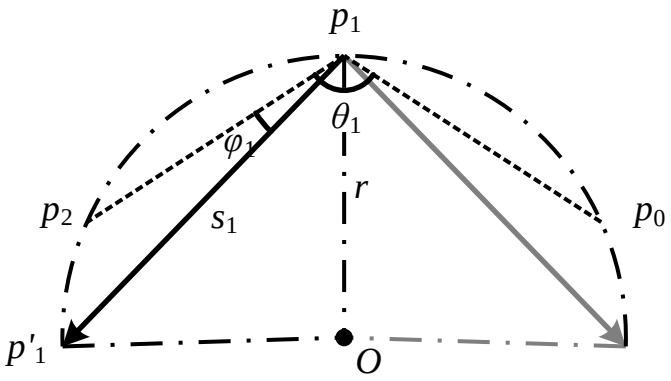
Figure 3.27 : Inward radial motion is unstable in these two extreme cases.

robots are given the information that $n = 6$, so the robots keep $\varphi = -120°$. The largest angle $\max_i \theta_i$ decreases, and the largest ratio of adjacent edges $\max_i \frac{l_i}{l_{i+1}}$ first increases and later decreases.

The instability of inward motion is not obvious for regular polygons, since the instability grows slowly and robots soon meet in the center. Figure 3.27 shows two extreme cases: two robots move out of the convex polygon or meet before reaching the center, changing the convex polygon into a concave or complex polygon.



(a) With $s < l$.



(b) With $s > l$ and $\varphi > 0$.

Figure 3.28 : Robots move onto the circumscribed circle so that the perimeter remains unchanged.

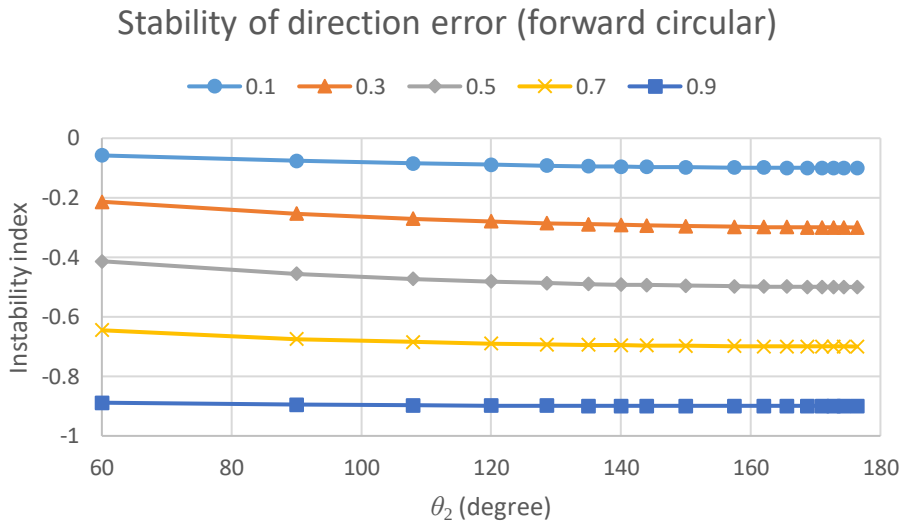


Figure 3.29 : Five data series with different step size ratios $\frac{s}{l}$ all have negative instability index, meaning that forward circular motion is stable relative to direction error.

Circular motion. If the robots start on a regular polygon and are given some specially chosen s and φ, the robots end up on another concentric regular polygon of the same size. In Figure 3.28, any point on the circumscribed circle can be chosen for the next move, including $s < l$ and $s > l$. Since we are more interested in the problem at a larger scale, we continue limiting $s < l$.

From the law of cosines

$$r^2 = r^2 + l^2 - 2rl\cos\frac{\theta}{2}, \tag{3.34}$$

$$r^2 = r^2 + s^2 - 2rs\cos\left(\frac{\theta}{2} - \varphi\right), \tag{3.35}$$

we get

$$\varphi = \frac{\theta}{2} - \arccos\frac{s}{2r},\ \varphi < 0. \tag{3.36}$$

Thus counterclockwise circular motion is stable (Figure 3.29), and is verified with the simulator for some extreme cases. Robots can reconstruct a circle from a line if they are given the corresponding parameters (n, s, φ). The perimeter automatically enlarges or shrinks until

$$r = \frac{s}{2\cos\left(\frac{n-2}{2n}\pi - \varphi\right)} \tag{3.37}$$

is satisfied.

There is another direction where robots keep the perimeter and rotate clockwise:

$$\varphi = \frac{\theta}{2} + \arccos\frac{s}{2r}. \tag{3.38}$$

This clockwise circular motion is unstable as observed in Figure 3.30.

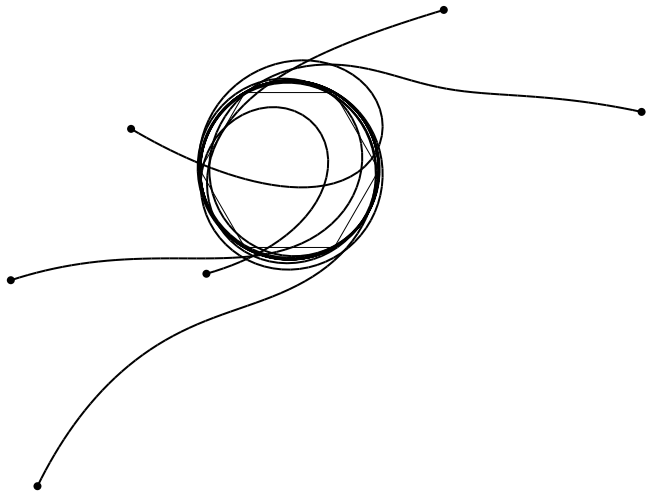
The most and the least stable directions. We can find the boundaries between stable and unstable configurations by finding φ that satisfies

$$\frac{\mathrm{d}\theta_1'(\varphi)}{\mathrm{d}\delta} = 1 \tag{3.39}$$

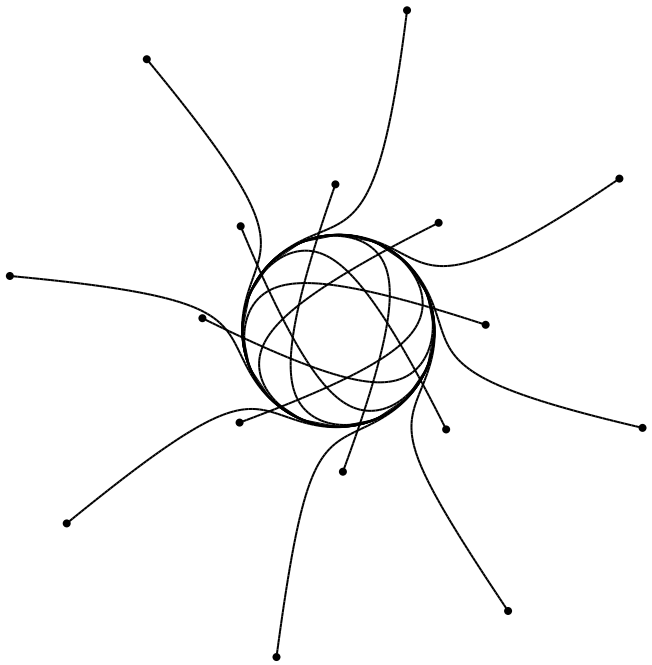
from Equation (3.31) where $\theta_1 = \theta_2 + \mathrm{d}\delta$ and θ_1' is influenced by φ. We can also find the most and the least stable directions by satisfying

$$\frac{\mathrm{d}^2\theta_1'}{\mathrm{d}\delta^2} = 0. \tag{3.40}$$

The analytical expressions for the derivatives are very complicated and depend on the step size ratio $\frac{s}{l}$. Using a numerical approach at a small step size ratio (Figure 3.31), we observe that the most stable direction is $\varphi_\mathrm{s} \approx \theta_2 - \pi$ which is $-\frac{2\pi}{n}$ for a regular polygon, and the stable range spans about 180°. A large step size ratio can cause these directions to depart from those observed values (Figure 3.31).



(a) With 6 robots.



(b) With 16 robots.

Figure 3.30 : Robots first rotate clockwise on the same circle, then error accumulates and robot motion becomes chaotic.

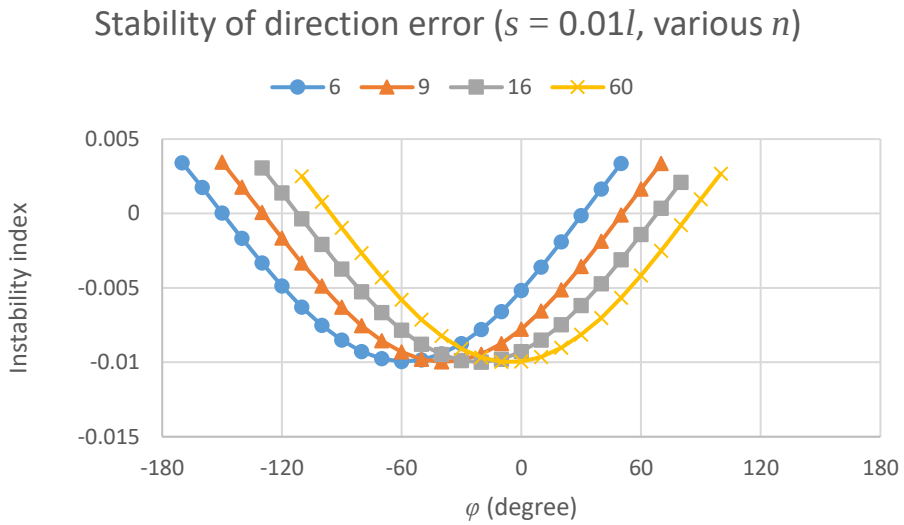


Figure 3.31 : At a small step size ratio $\frac{s}{l} = 0.01$, we observe that the most stable direction is about $-\frac{2\pi}{n}$. The stable range and the unstable range each occupies a half-plane. Four data series show the calculation for a hexagon, nonagon, hexadecagon, and hexacontagon.

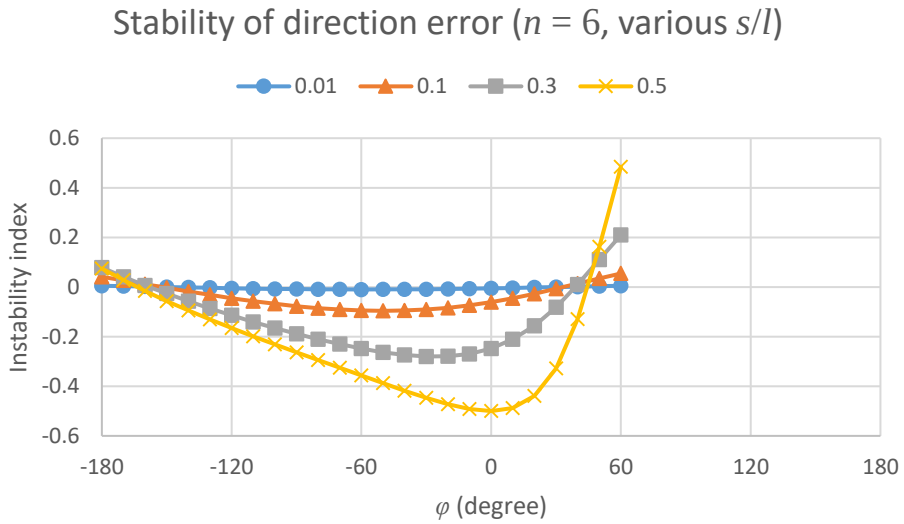


Figure 3.32 : With larger step size ratios in the legend, we observe that the curves of stability become distorted and the most stable directions drift counterclockwise towards the forward direction. The boundaries between stable and unstable remain almost unchanged.

Name	φ	ccw/cw	in/out	stable?
Unstable/stable boundary	$-\frac{n+4}{2n}\pi$	cw	out	-
Outward radial	$-\frac{n+2}{2n}\pi$	-	out	y
Most Stable	$-\frac{2\pi}{n}$	ccw	out	y
Forward circular	$\frac{n-2}{2n}\pi - \arccos\frac{s}{2r}$	ccw	-	y
Forward	0	ccw	in	y
Stable/unstable boundary	$\frac{n-4}{2n}\pi$	ccw	in	-
Inward radial	$\frac{n-2}{2n}\pi$	-	in	n
Least stable	$\frac{n-2}{n}\pi$	cw	in	n
Reverse circular	$\frac{n-2}{2n}\pi + \arccos\frac{s}{2r}$	cw	-	n
Reverse	π	cw	out	n

Table 3.1 : List of special directions in the general problem.

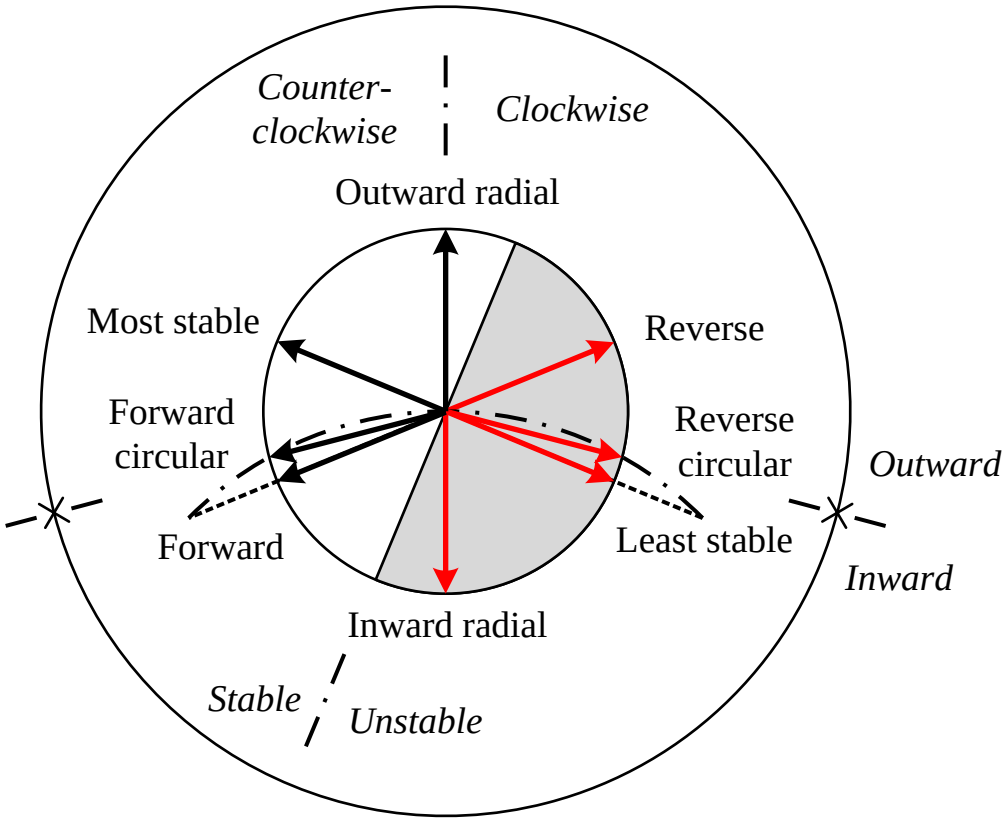
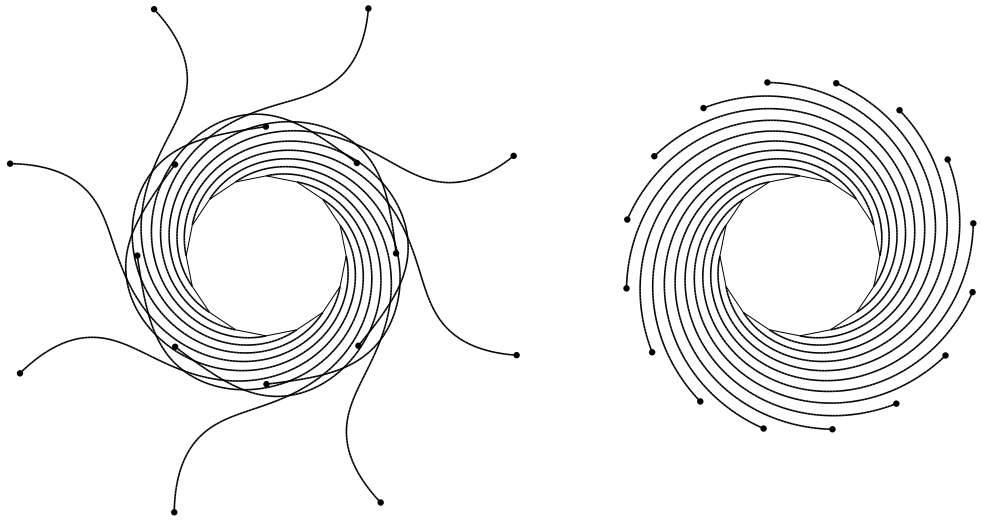


Figure 3.33 : The directions can be classified into three pairs of categories: counter-clockwise/clockwise, inward/outward, and stable/unstable.



(a) Unstable with $\varphi = 180°$. (b) Stable with $\varphi = -22.5°$.

Figure 3.34 : With a different φ, an unstable problem has a stable version in a mirror image. Both figures have 16 robots and the snapshot is taken after 150 rounds.

Finally, we have a list of all the special directions. Table 3.1 lists all these special directions (at a small step size ratio), and Figure 3.33 shows all these directions in a unit circle.

3.7 Methods to Maintain Stability

Now we understand why the robot evasion problem is unstable and which general configurations are unstable. How then can we control the robots in these problems to maintain stability?

One solution is that since Figure 3.33 has some reflective symmetry, we may be able to choose another angle φ to obtain a stable configuration in the mirror image. For the robot evasion problem, if we change the direction from $\varphi = 180°$ to $\varphi = -\frac{360°}{n}$, the robots move outward clockwise as if in a mirror image (Figure 3.34), and this configuration is the most stable one among all the configurations.

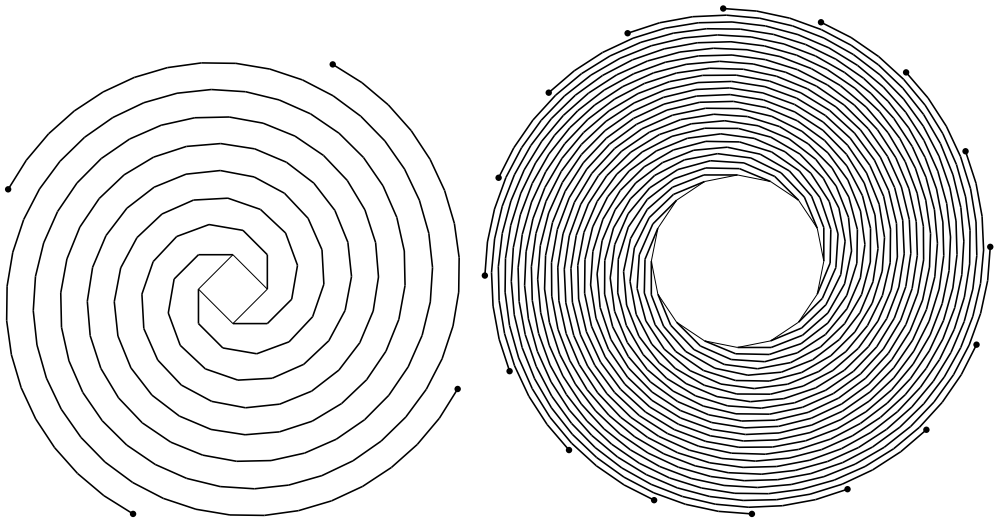
If we initially deploy the robots in the clockwise orientation and use a left-handed coordinate system, the trajectories will be exactly the spirals as expected. This method does not apply to the directions near the inward radial direction since the mirror image is still unstable, but since these directions have very little instability (near the boundary) and the robots are gathering at the center, noticeable instability is not observed from configurations with regular polygons.

We are also interested in moving robots along some other types of spirals. The Archimedean spiral has a nice geometric property: *any ray from the origin intersects successive turnings of the spiral in points with a constant separation distance* [108]. The involute of a circle also has a similar geometric property: *its successive turns are parallel curves with constant separation distance* [109]. If each robot has an area of reach greater than this separation distance, the union of the area of reach over time can efficiently cover a continuous piece of the ground. Robots can paint the floor, mow the grass, or detect underground objects progressively from a given origin. The iRobot Roomba robotic vacuum cleaner [70] starts in this behavior when placed on the floor away from its charger.

When we set $\varphi = \frac{\theta-\pi}{2} = -\frac{\pi}{n}$, we observe that the robots move along outward counterclockwise spirals that have almost constant separation distance. Figure 3.35 shows two examples (compare with Figure 3.34(b) to see the difference).

Figure 3.36 provides an explanation of this spiral behavior. With $\varphi = -\frac{\pi}{n}$, we have $\frac{\theta}{2} - \varphi = \frac{\pi}{2}$, and the robot always move perpendicular to the radial direction. From the Pythagorean theorem

$$r_1 = \sqrt{r_0^2 + s^2},\ r_2 = \sqrt{r_1^2 + s^2} = \sqrt{r_0^2 + 2s^2}, \tag{3.41}$$



(a) With 4 robots after 50 rounds, $\varphi = -45°$.

(b) With 16 robots after 50 rounds, $\varphi = -11.25°$.

Figure 3.35 : Robots move along spirals with almost constant separation distance.

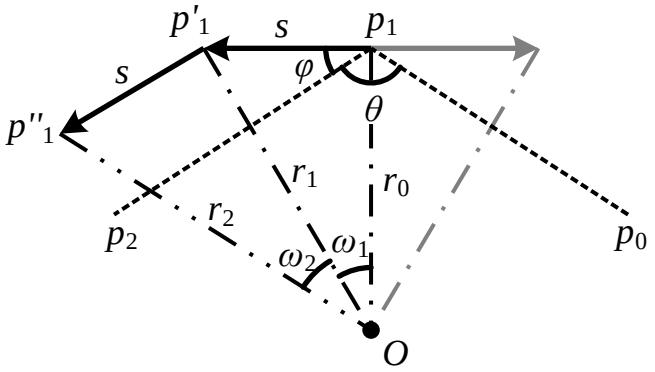


Figure 3.36 : The robot moves in the direction perpendicular to the radial direction, and its distance the the origin increases in each round.

$$\omega_1 = \arcsin\frac{s}{r_1},\ \omega_2 = \arcsin\frac{s}{r_2}. \tag{3.42}$$

Triangle $\triangle p_1'' O p_1'$ is not similar to $\triangle p_1' O p_1$, since we keep the same step size s, thus $\omega_2 < \omega_1$. In general, after k rounds of iteration

$$\frac{r_k - r_{k-1}}{\omega_k} = \frac{\sqrt{r_0^2 + ks^2} - \sqrt{r_0^2 + (k-1)s^2}}{\arcsin\frac{s}{\sqrt{r_0^2+ks^2}}}. \tag{3.43}$$

During the kth round, the distance between the robot and the origin increases by $r_k - r_{k-1}$, and the robot rotates by the angle ω_k around the origin. $\frac{r_k - r_{k-1}}{\omega_k}$ describes how much distance is gained from one radian of revolution.

Since $x < \arcsin x < \frac{\pi}{2}x$ for $0 < x < \pi$, we can find an upper bound

$$\frac{r_k - r_{k-1}}{\omega_k} = \frac{r_k - r_{k-1}}{\arcsin\frac{s}{r_k}} < \frac{r_k - r_{k-1}}{\frac{s}{r_k}} < \frac{r_k^2 - r_{k-1}^2}{s} = s \tag{3.44}$$

and a lower bound

$$\frac{r_k - r_{k-1}}{\omega_k} = \frac{r_k - r_{k-1}}{\arcsin\frac{s}{r_k}} > \frac{r_k - r_{k-1}}{\frac{\pi}{2}\frac{s}{r_k}} > \frac{2}{\pi}\frac{r_k^2 - r_{k-\frac{1}{2}}^2}{s} = \frac{s}{\pi}. \tag{3.45}$$

We have both an upper bound and a lower bound for the distance gained per radian of revolution, so per revolution $2\pi s < 2\pi\frac{r_k - r_{k-1}}{\omega_k} < 2s$. This spiral has bounded separation distance. In fact, $\frac{r_k - r_{k-1}}{\omega_k}$ monotonically decreases with the limit

$$\lim_{k\to\infty}\frac{r_k - r_{k-1}}{\omega_k} = \frac{s}{2}. \tag{3.46}$$

This limit is independent of r_0. The step size s controls the tightness of the spiral. Figure 3.37 shows the actual distance gained per radian of revolution. This spiral is not exactly an Archimedean spiral or the involute of a circle, but is a very good

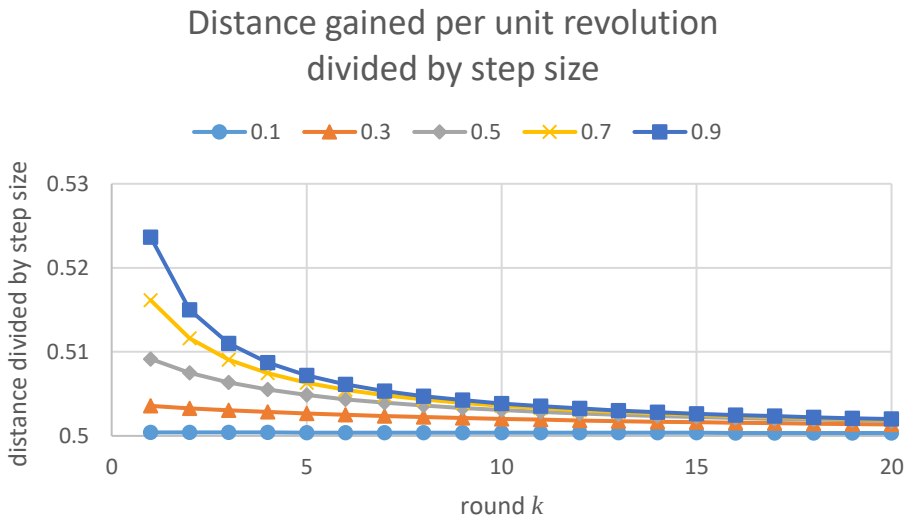


Figure 3.37 : The separation distance is proportional to step size when $k \to \infty$ and the limit is independent of the initial circumradius which is set to $r_0 = 1$. Five data series have different step sizes valued in the legend.

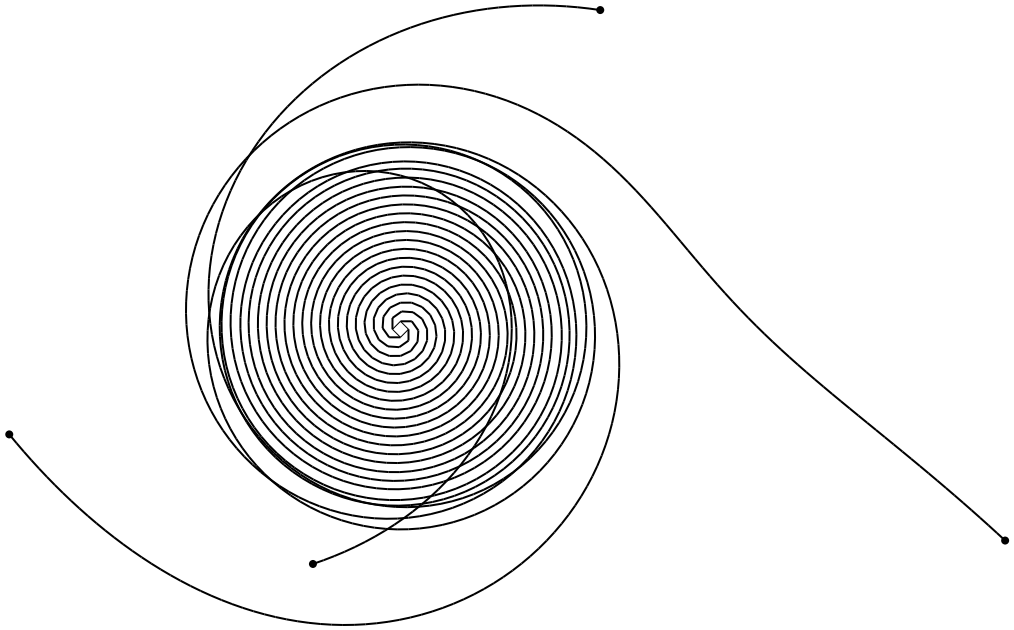
approximation.

Since $-\frac{2\pi}{n} < \varphi < 0$, φ falls into the stable range, and the stability is observed in simulations. There is another $\varphi = \theta + \frac{\pi}{n} = \frac{n-1}{n}\pi$ for a clockwise version which is unstable (Figure 3.38).

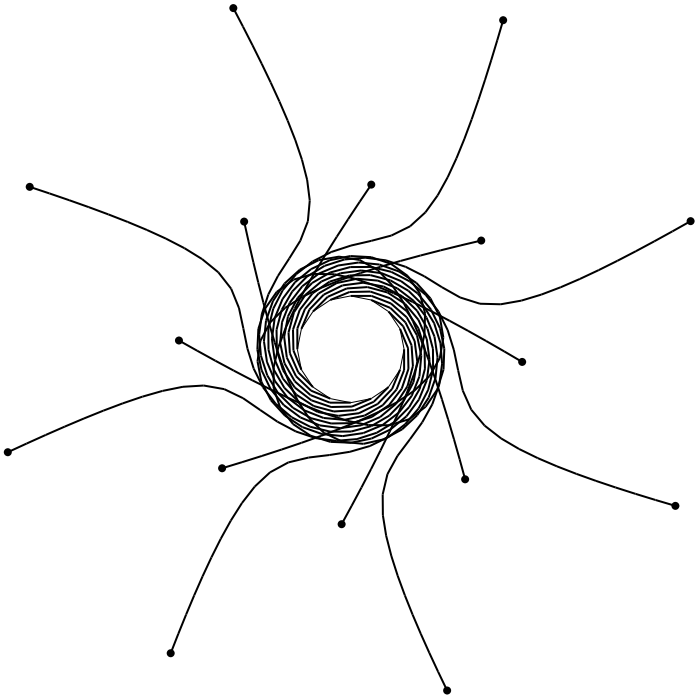
With multiple robots, we can adjust the parameters to ensure that each piece of ground is examined by two or more different robots ensuring that our algorithm is robust to the failure of a single sensor.

3.8 Conclusion

Starting from the four-bug problem, we generalize to an arbitrary number of robots and arbitrary motion angles. We find that some configurations are unstable to direction error and distance error, and verify in simulations that robots move in chaotic



(a) With 4 robots after 400 rounds, $\varphi = 135°$.



(b) With 16 robots after 30 rounds, $\varphi = 168.75°$.

Figure 3.38 : The clockwise spirals intended to have constant separation distance are unstable.

trajectories. We analyze the stability with both analytical and numerical methods, and classify which range of angles lead to instability. Finally, we provide solutions that maintain stability so that multiple robots can efficiently patrol along a perimeter, shrink the perimeter, expand the perimeter, and seamlessly search over the ground with controllable redundancy. The results from this algorithm not only enhance engineering applications with better parallelization and robustness but also can be used to generate artistic drawings.

Chapter 4

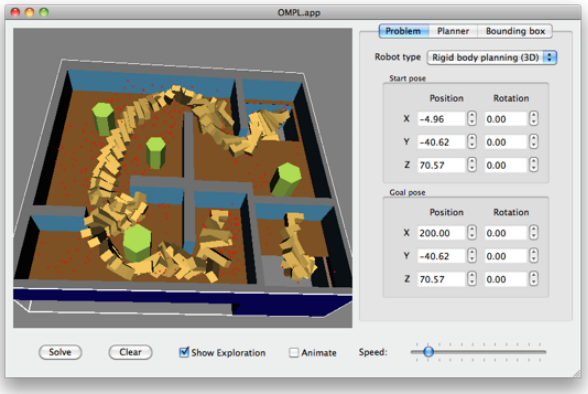
Design and Implementation of Multi-Robot Simulator

4.1 Introduction and Related Work

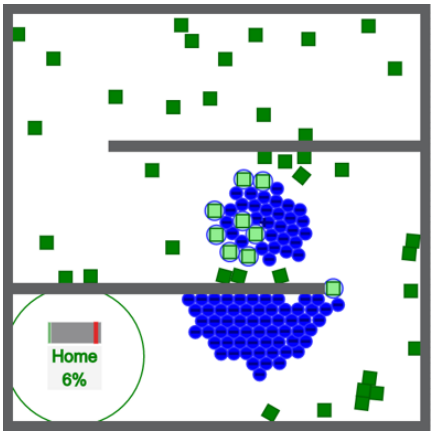
When I joined the Multi-Robot Systems Lab led by James McLurkin to start my research on swarm robotics, we programmed directly on the r-one robots [114]. Programming was slow before wireless broadcast programming was implemented: USB cable connections were required and only one robot can be programmed at a time. Robots have sensing and motion errors, and sometimes robots fail due to hardware defects. Wireless bandwidth was limited so that real-time data collection often interfered with robot-to-robot communication. In order to efficiently verify and debug new algorithms, a multi-robot simulator became a necessity.

There are many robot simulators available to the public for free, and each simulator has its own applicable scope. Some simulators focus on the physics of connected rigid bodies with degrees of freedom, such as robot arms and robot hands [115] [116] [117]. Some simulators build realistic 3D environments for humanoid robots and robot soccer [112] [113] [118].

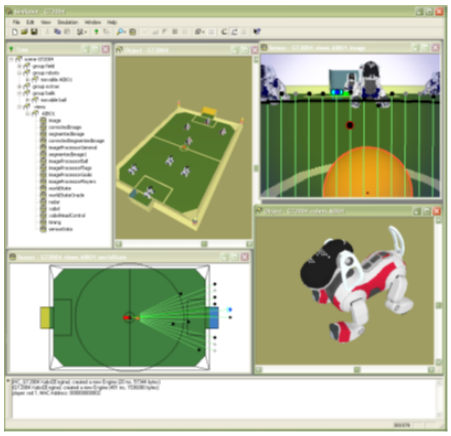
The Open Motion Planning Library (OMPL) is a robotic software library developed by the Kavraki Lab at Rice University [110]. This software is written in C++ and supports motion planning for multiple agents, using sampling-based algorithms including PRM [119], RRT [120], KPIECE [121], and STRIDE [122]. A user interface



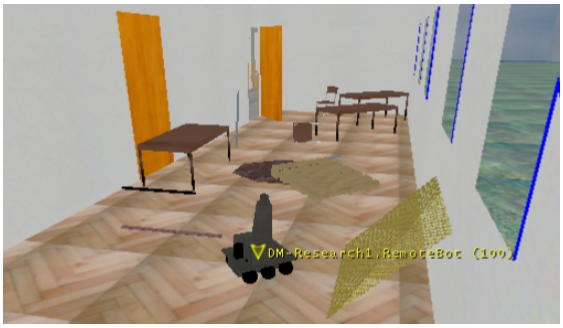
(a) OMPL [110].



(b) SwarmControl.net by Aaron Becker [111].



(c) SimRobot [112].



(d) USARSim [113].

Figure 4.1 : Images from related work.

is provided to view animations of computed results (Figure 4.1(a)).

Aaron Becker studied controlling a robot swarm with a common signal [123] and built a series of online games with HTML5 [111]. In these games a user can use a mouse to attract robots or use the keyboard to drive the robots in the same direction (Figure 4.1(b)). Robots are modeled as solid objects and physical forces are simulated so that objects can be pushed by the swarm and robots can push against each other.

There are also many other simulators written by researchers to simulate the swarm robot platforms they use [124] [125] [126]. These simulators often have fancy animations, but the details of individual robots and the construction of the background environment become a large overhead when a large number of robots need to be simulated. My simulator focuses on the geometry and topology of robot networks, and communication between robots is more important to my research than physics. Existing simulators are not good solutions for me to quickly evolve my algorithms. So I wrote my own simulator for my own research.

4.2 System Design

My fifth generation multi-robot simulator is developed as a C# Windows Application with Microsoft Visual Studio 2015. The simulator is optimized to run on Microsoft Windows 7 with .NET Framework 4.0, but some other platforms can also be supported. The simulator consists of one framework and six modules: robot, physics, program, control, graphics, and data (Figure 4.2).

Figure 4.3 shows a screenshot of the simulator with Windows Forms graphics. All the figures describing robot swarms in the previous chapters are also exported from the simulator.

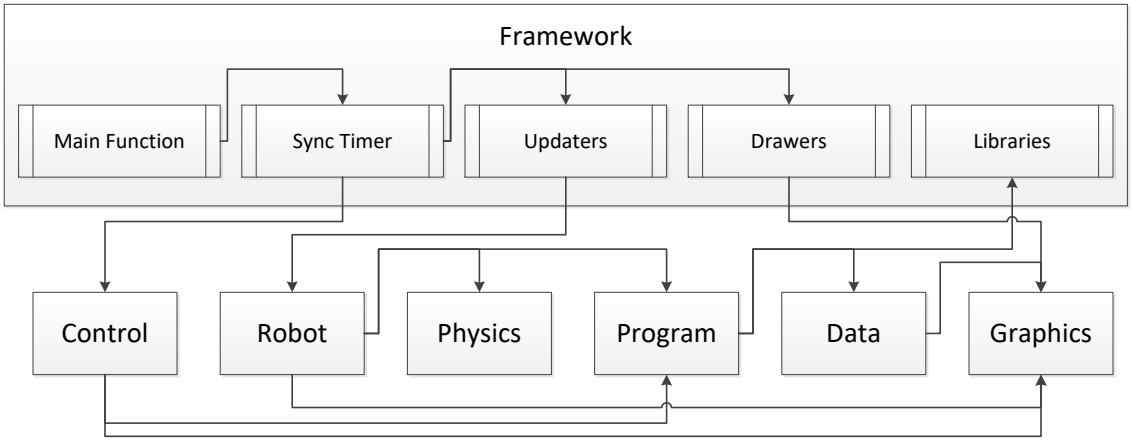


Figure 4.2 : The system structure of my simulator.

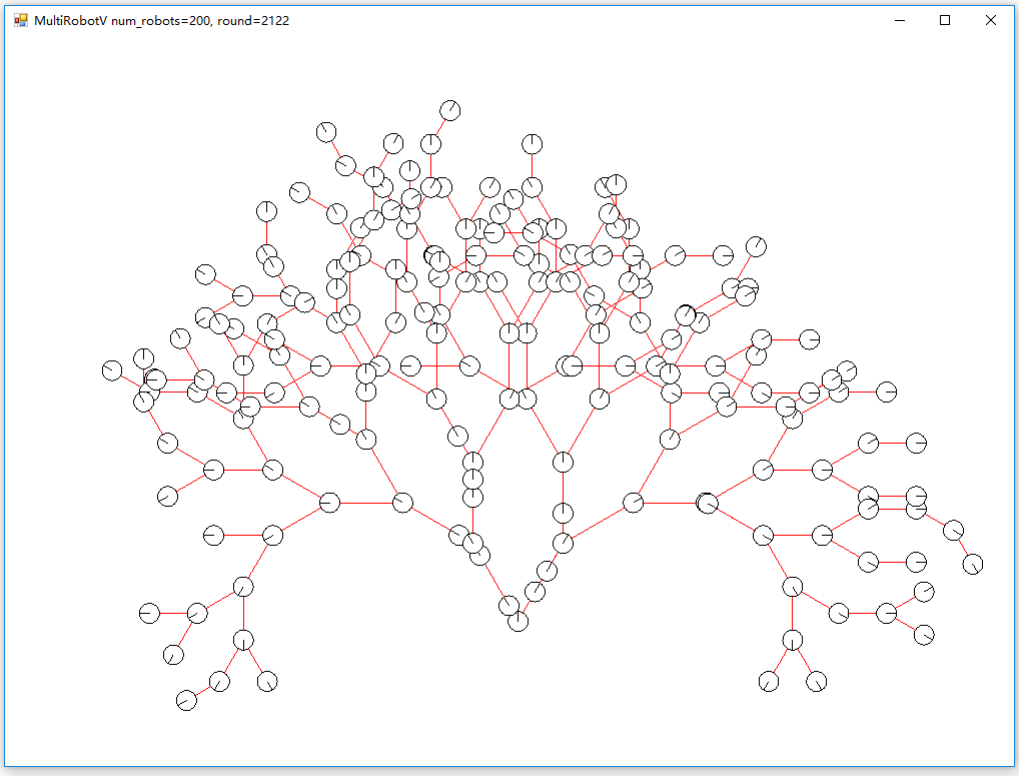


Figure 4.3 : The simulator with 200 robots building a fractal tree. Some robots are still in motion and have not yet arrived at their final destinations.

4.2.1 The Simulator Framework

The simulator framework is a set of classes that provides these functionalities:

- Serve as the static Main function for program entry point.
- Manage the threads for parallel execution.
- Synchronize the timer for a stable refresh rate and for critical sections that require single-thread execution.
- Provide shared data structures, default parameters, and mathematical libraries.

When the simulator starts to run, the simulator framework first loads several parameters, including how many threads to use and how many frames to display per second. Each module is then initialized. Several threads are created in a paused state with different payloads, such as the updaters and the drawers. Finally, the sync timer is created which is used to synchronize the threads.

The updaters and the drawers are inspired by XNA, in which one updater and one drawer execute alternatively sharing a single thread. To enable multi-threading, my simulator allows a configurable number of updaters and drawers, and manages them with minimal thread-safe constraints.

An *updater* is a worker thread for updating states related to the robots. A list of the robots is stored by the robot module, and the list is fragmented so that each robot is updated by one of these updaters, unless a user pauses the simulator. The updater calls these modules in order:

- Physics module: determines which robots fall in each other's sensing range, and calculates their relative positions.

- Program module: executes user-written program to process sensor readings, updates robot internal states, and generates the commands to actuators.
- Physics module: simulates actuator behaviors including robot motion.
- Data module: generates output data and prepares data for real-time plots.

The robot states before and after one round of update are stored into different copies. This storage structure ensures that if a robot reads information from another robot, the information is always from the previous round regardless of in what order the robots are updated. Since only the information written during updating is accessible by the corresponding updater, we do not have to introduce any locking mechanism on the time-consuming updating process.

A *drawer* is a worker thread for converting robots' states into a graphical drawing. The drawer may be single-threaded if multi-threading is not supported by the graphics engine. The drawer calls the program module to gather the contents needed for drawing, including the robots, the caption of each robot, robot-to-robot connections, and any other information customized by the user program. The drawer may also gather data from the data module for real-time plots. The coordinate systems are converted from ground coordinates and plot coordinates into the screen coordinates. Finally the drawer invokes the graphics module to render the user interface.

The *sync timer* has two modes: user interaction mode and video rendering mode. In the user interaction mode, the sync timer is invoked once every 20 ms (50 Hz) or 16.7 ms (60 Hz) to maintain a stable refresh rate. In the video rendering mode, the sync timer is invoked once all the updaters and drawers finish their tasks.

The sync timer activates the updaters and drawers so that all the tasks are aligned by rounds. Before activating the multi-thread tasks, the sync timer needs to execute

some single-threaded tasks or those tasks requiring locking mechanism:

- Exchange the two sets of robot states: previous and current. This exchange also moves all the robots from their previous positions to their calculated destinations.
- Add and/or remove robots, since the number of robots cannot change when iterating over the list of robots.
- Initialize the updaters and drawers if a single-thread initialization is required before multi-thread tasks.
- Process user input (can also be handled with a dedicated updater).

This framework provides several common *data structures*. The types of angles are packed into an Angle class with subclasses like Degree and Radian. When the user creates a variable for an angle, the user calls different static methods of the Angle class to identify whether the angle unit is in degrees or in radians, and the static methods call the private constructors to create a subclass instance according to the factory pattern [127]. Operators are provided so that the user can simply sum up a degree angle and a radian angle with a plus sign, and all the conversions are performed automatically behind the scene. The Vector class is handled with a similar set of subclasses to represent vectors written in a Cartesian coordinate system (x, y) and in a polar coordinate system (r, θ). Integer-only calculation is also provided as subclasses to simulate a robot lacking floating point calculation.

The framework also provides a *robotic math library* for some popular mathematical operations such as calculating the distance between two neighbors. Most of these functions are bound to the data structures to support the diversity of data units.

Integer-only approximations of trigonometric functions automatically apply to those integer-only units.

4.2.2 The Robot Module

The robot module maintains a list of the robots, and each robot is represented by an instance of a dynamic class.

Each robot is uniquely identified by a Globally Unique Identifier (GUID) [128]. GUIDs are 128-bit randomly generated numbers that are unlikely to be duplicated, and are efficient for use in hash maps. The list of the robots is a hash map that maps GUIDs to robot objects.

The robot object contains a set of properties, external states, and internal states. The *properties* are the parameters of robot physics, such as the radius, the communication range, and the maximum speed. The *external states* describe the robot's position, heading, current speed, and current angle speed. The *internal states* are those states defined by the user, such as topological information used by a distributed spanning tree [58]. The robot class is now designed as a dynamic class to improve the compatibility between programs, and the internal states are stored as a hash map from string to object.

The robot object also provides several *methods* that can easily be used to manipulate the robot, such as moving towards another robot or along some specific direction. These functions only set the motion targets: the actual movements are handled by the physics module.

4.2.3 The Physics Module

The physics module is a set of classes that provides these functionalities:

- Models for sensing (neighbor recognition and obstacle detection) with simulated sensing errors.
- Models of measurement (distance and angle) with simulated measurement errors.
- Executes communication commands with simulated communication errors.
- Executes motion commands with simulated motion errors and collision events.

Researchers have tried to solve swarm problems with low-cost sensors or even allowing some sensors to be absent. The Kilobot [129] designed at Harvard University can measure distance with infrared reflection, but cannot measure direction with a single infrared receiver. The r-one robot [114] designed at Rice University can measure heading and orientation with multiple infrared transmitters and receivers, but does not support distance measurement before range bits are introduced [4] [59]. Kroller et. al. are even able to extract topology from robots with no continuous measurements but can tell only whether or not another robot is within sensing range [130]. My simulator provides sensing and measurement models that can intentionally hide some information from the robots. The measurement error can be modeled as a uniform distribution, a Gaussian distribution, or some user-defined discrete distribution. Systematic errors and random loss of measurement are also supported.

The motion commands which specify motion targets are converted into motion destinations that adhere to speed limits, avoid collisions, and respect nonholonomic restrictions. Robots are given feedback concerning whether the motion targets are achieved.

4.2.4 The Program Module

The program module contains a base class called Program, and holds user-defined robot programs derived from this base class.

A user program contains user-defined code that sets up the environment, modifies parameters, and updates the robots. Users can also override some functions such as how the robot is drawn on the screen, for example, draw a robot in some specific color with a customized caption.

Several template programs are provided, such as robots building fractals. Users can subclass a program and change some parameters to build a different fractal.

4.2.5 The Control Module

The control module provides a bidirectional user interface with these functionalities:

- A control panel with buttons for input and indicators for output.
- Capture mouse events that select/add/delete/move a robot or adjust the view.
- Capture keyboard events that start/stop/change the current program, load/save data, adjust the view, move robots.
- Handle input and output for peripherals such as a joystick, keyboard illumination, and real robots.

A default set of keyboard and mouse commands are provided, and users can also assign keys to functions in the user program (including overriding the existing ones). Users can use customized keys to manually control some specific robots.

4.2.6 The Graphics Module

The graphics module is a set of classes that provides these functionalities:

- Converts robot coordinates into graphics coordinates and vice versa.
- Maintains a list of items to be drawn and/or graphics buffers, depending on the type of graphics.
- Draws the items on the screen or through third-party graphics engines.

Different graphics engines can be chosen by the user. Drawing on a Windows Form is now supported, and third-party graphics engines such as DirectX and OpenGL will be introduced soon. Third-party graphics engines usually have their own threading models, but several tricks are required to enable multi-threading on Windows Forms.

Instead of drawing objects sequentially based on their layers, a z-buffer can be used to describe the depth of each pixel. The z-buffer can be rendered in parallel if painting on a pixel can be done atomically, or if the canvas can be split into distinct parts. In the past, drawing robots along with their historical trajectories was often slow because the line segments in the trajectories increase rapidly and need to be rendered in every frame. A solution to this problem is to render the trajectories and the robots separately; the trajectories are cached in a bitmap and only recent line segments are added. The full history of trajectories is then still preserved in case the user changes the view and a full repaint is needed.

4.2.7 The Data Module

The data module is a set of classes that provides these functionalities:

- Loads and saves robot configurations.

- Saves and exports any data generated by user programs.
- Generates real-time plots.
- Handles program logs and exceptions.

Exporting the positions of the robots and their trajectories as Scalable Vector Graphics (SVG) files is a new feature. SVG files use XML to describe line segments, circles, rectangles, and curves, which can be formatted from existing data structures. SVG files can later be converted to PDF files by Inkscape command-line, and embedded at high definition in academic articles.

4.3 Discussion

I started to develop my first-generation multi-robot simulator in mid-2013. I chose Microsoft XNA because I had just learned XNA for game development in a game design course, and XNA provides texture-based drawings and frame management. In addition, C# is an object-oriented programming language that is easy to write and to maintain. My first simulator focused on the transition between the Cartesian coordinate system of the ground and the polar coordinate system of each robot. A robot converts its destination in the global coordinate system into the angles and distances relative to some stationary robots, and becomes stationary itself when the robot thinks that its destination has been reached. Simulated sensing and motion errors can be adjusted, and the ground-truth destinations are shown as references to compare between perceived and actual positions.

The second version of my simulator was developed when I needed to simulate swarm behaviors. I implemented a distributed spanning tree for my topology-based

sorting algorithm [131]. Realistic scenarios such as integer-based computing, section-based bearing measurement, and collision avoidance were also introduced. A data collection module was written to gather real-time data from the robots and the AprilTag [132] positioning system so that experiments could be monitored and controlled by the computer. These improvements provided more realistic simulations, but because at that time I lacked sufficient knowledge of software engineering, the program soon became too complicated and difficult to use.

I had to write the third version of my simulator from scratch because I needed to implement topological information which is difficult to debug in a over-complicated simulator. For my topology-based sorting algorithm [58], I introduced a switch between global topology mode and local topology mode. The global topology mode provides collection logic over a set of states so that I can always generate the correct topology, and develop my algorithm without worrying about topological errors. Later I switch back to local topology mode to verify that my algorithm is robust to topological errors such as a robot being added or removed from the swarm. I also created a tool bar to control the program by clicking a mouse on some buttons, but this panel seems not to work well with XNA.

In the fourth generation of my multi-robot simulator, I introduced a better object-oriented design so that different programs with different parameters – sorting, fractal formation, and stability analysis – could be integrated within the same framework. The user-defined internal states of the robots were packed into a separate class, and I intended to support dynamic switching between programs, but this attempt failed due to different definitions of the states in different programs.

Then I realized that the XNA framework is becoming a bottleneck. XNA was deprecated by Microsoft, and XNA is no longer supported by Visual Studio 2012 and

newer versions of Visual Studio. XNA handles updating and drawing automatically, but runs only in a single thread so many CPU cores remain idle. Texture-based drawing is redundant as I usually draw lines and circles, and bitmap textures do not scale to high-definition. XNA is based on single-precision floating point and 32-bit integers, which are insufficient for scientific computing. The classes of vector and color provided by XNA lack some efficient methods, but Microsoft marks those classes as final so I cannot inherent them to make my improvements. Therefore, I decided to get rid of XNA and replace XNA with my own framework that solves these problems.

4.4 Conclusion

I have built a multi-robot simulator that can simulate swarm algorithms in real time. This simulator is specially crafted to my research and provides realistic modeling and high definition output with minimal latency. Multi-threading and 64-bit architecture increase the performance over previous simulators based on Microsoft XNA, customized data structures eliminate the pains of unit conversions, and software engineering techniques make the program easier to maintain and to use to develop new functionalities.

GPU computing is an oncoming trend. GPUs are manufactured with more cores than CPUs and with more powerful floating point calculation abilities. These advantages are especially suitable for multi-robot simulation. So in the future I plan to learn GPU programming and to build a faster GPU-based multi-robot simulator.

Chapter 5

Conclusion

I developed an algorithm that allows a swarm of robots to self-assemble into fractals. Since there are various types of fractals – tree-based, curve-based, and space-filling fractals – with different intrinsic properties, I designed methods corresponding to these properties to generate fractals efficiently. A collision-free solution is provided for building tree-based fractals. The process of building fractals can be used in both art and engineering, for generating aesthetic shapes and for building fractal antennas.

I analyzed the stability of the general robot chasing problem, which is an extension of the classical four-bug problem. I observed both spiral motions and chaotic behaviors with robot swarms, which inspired me to figure out the reason for instability and whether a general configuration is stable. I provided a solution to convert an unstable problem into a stable one, and found a concise method to approximate the Archimedean spiral which allows multiple robots to traverse the environment efficiently. The results can help engineers build swarm systems with more stability, while the trajectories of unstable robot swarms can be drawn as artistic patterns.

This work and the work in my Master's thesis are supported by my multi-robot simulator. I developed my own multi-robot simulator so that I can verify my algorithms quickly and gather data in real time. The simulator is optimized for my area of research and enhanced with parallel computing. The artistic drawings and some figures for scientific analysis are exported by the simulator in high definition formats.

In my future work, I need to make a better connection between art and engineering,

and generate more practical solutions for art and engineering problems. The fractal generating algorithm can build only a subset of all the fractals and needs to be expanded, and more specific algorithms need to be designed for different space-filling fractals. In addition to geometric stability, topological stability is another area of study that deals with inconsistency. To increase the number of robots supported by the multi-robot simulator, GPU programming is an oncoming trend and adopting the new technologies will benefit researches in both art and engineering.

Bibliography

[1] A. Becker, C. Onyuksel, T. Bretl, and J. McLurkin, "Controlling many differential-drive robots with uniform control inputs," *The international journal of Robotics Research*, vol. 33, no. 13, pp. 1626–1644, 2014.

[2] "The kilobot project." Self-organizing Systems Research Group, Harvard University. http://www.eecs.harvard.edu/ssr/projects/progSA/kilobot.html.

[3] A. Becker, S. P. Fekete, A. Krller, S. K. Lee, J. McLurkin, and C. Schmidt, "Triangulating Unknown Environments Using Robot Swarms," in *Proceedings of the Twenty-ninth Annual Symposium on Computational Geometry*, SoCG '13, (New York, NY, USA), pp. 345–346, ACM, 2013.

[4] J. McLurkin, A. McMullen, N. Robbins, G. Habibi, A. Becker, A. Chou, H. Li, M. John, N. Okeke, J. Rykowski, S. Kim, W. Xie, T. Vaughn, Y. Zhou, J. Shen, N. Chen, Q. Kaseman, L. Langford, J. Hunt, A. Boone, and K. Koch, "A robot system design for low-cost multi-robot manipulation," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2014)*, pp. 912–918, Sept. 2014.

[5] M. Rubenstein, A. Cornejo, and R. Nagpal, "Programmable self-assembly in a thousand-robot swarm," *Science*, vol. 345, no. 6198, pp. 795–799, 2014.

[6] C. R. Reid, M. J. Lutz, S. Powell, A. B. Kao, I. D. Couzin, and S. Garnier, "Army ants dynamically adjust living bridges in response to a cost–benefit

trade-off," *Proceedings of the National Academy of Sciences*, vol. 112, no. 49, pp. 15113–15118, 2015.

[7] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *ACM SIGGRAPH computer graphics*, vol. 21, no. 4, pp. 25–34, 1987.

[8] J. McLurkin, *Analysis and Implementation of Distributed Algorithms for Multi-Robot Systems*. Ph.D. thesis, Massachusetts Institute of Technology, 2008.

[9] S. K. Lee and J. McLurkin, "Distributed cohesive configuration control for swarm robots with boundary information and network sensing," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1161–1167, Sept 2014.

[10] D. Sakai, H. Fukushima, and F. Matsuno, "Flocking for multirobots without distinguishing robots and obstacles," *IEEE Transactions on Control Systems Technology*, 2016.

[11] M. S. R. Mousavi, M. Khaghani, and G. Vossoughi, "Collision avoidance with obstacles in flocking for multi agent systems," in *Industrial Electronics, Control & Robotics (IECR), 2010 International Conference on*, pp. 1–5, IEEE, 2010.

[12] X. Luo and D. Liu, "Flocking and obstacle avoidance for multi-agent based on potential function," in *Control Conference (CCC), 2010 29th Chinese*, pp. 4613–4618, IEEE, 2010.

[13] A. Becker, E. D. Demaine, S. P. Fekete, and J. McLurkin, "Particle computation: Designing worlds to control robot swarms with only global signals," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pp. 6751–6756, IEEE, 2014.

[14] A. Becker and J. McLurkin, "Exact range and bearing control of many differential-drive robots with uniform control inputs," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pp. 3338–3343, IEEE, 2013.

[15] A. Becker, G. Habibi, J. Werfel, M. Rubenstein, and J. McLurkin, "Massive uniform manipulation: Controlling large populations of simple robots with a common input signal," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pp. 520–527, IEEE, 2013.

[16] J. Tan, N. Xi, W. Sheng, and J. Xiao, "Modeling multiple robot systems for area coverage and cooperation," in *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, vol. 3, pp. 2568–2573, IEEE, 2004.

[17] S. K. Lee, A. Becker, S. P. Fekete, A. Kröller, and J. McLurkin, "Exploration via structured triangulation by a multi-robot system with bearing-only low-resolution sensors," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pp. 2150–2157, IEEE, 2014.

[18] D. Maftuleac, S. K. Lee, S. P. Fekete, A. K. Akash, A. López-Ortiz, and J. McLurkin, "Local policies for efficiently patrolling a triangulated region by a robot swarm," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pp. 1809–1815, IEEE, 2015.

[19] G. Habibi, Z. Kingston, W. Xie, M. Jellins, and J. McLurkin, "Distributed centroid estimation and motion controllers for collective transport by multi-robot systems," in *Robotics and Automation (ICRA), 2015 IEEE International*

Conference on, pp. 1282–1288, IEEE, 2015.

[20] E. Campbell, Z. C. Kong, W. Hered, A. J. Lynch, M. K. O'Malley, and J. McLurkin, "Design of a low-cost series elastic actuator for multi-robot manipulation," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 5395–5400, IEEE, 2011.

[21] S. Ahirwar, S. Saxena, and S. Shukla, "Self assembly of large area 3d photonic crystals," in *International Conference on Fibre Optics and Photonics*, pp. M3C–3, Optical Society of America, 2012.

[22] D.-C. Wang, G.-Y. Chen, K.-Y. Chen, and C.-H. Tsai, "Dna as a template in self-assembly of au nano-structure," *IET nanobiotechnology*, vol. 5, no. 4, pp. 132–135, 2011.

[23] M. Rubenstein, C. Ahler, N. Hoff, A. Cabrera, and R. Nagpal, "Kilobot: A low cost robot with scalable operations designed for collective behaviors," *Robotics and Autonomous Systems*, vol. 62, no. 7, pp. 966–975, 2014.

[24] J. W. Romanishin, K. Gilpin, and D. Rus, "M-blocks: Momentum-driven, magnetic modular robots," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pp. 4288–4295, IEEE, 2013.

[25] M. Gajamohan, M. Merz, I. Thommen, and R. D'Andrea, "The cubli: A cube that can jump up and balance," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 3722–3727, IEEE, 2012.

[26] R. Goldman, *An Integrated Introduction to Computer Graphics and Geometric Modeling*. Boca Raton, FL, USA: CRC Press, Inc., 1st ed., 2009.

[27] N. Cohen, "Fractal antennas and fractal resonators," Sept. 17 2002. US Patent 6,452,553.

[28] D. Sankaranarayanan, D. Venkatakiran, and B. Mukherjee, "Koch snowflake dielectric resonator antenna with periodic circular slots for high gain and wideband applications," in *2016 URSI Asia-Pacific Radio Science Conference (URSI AP-RASC)*, pp. 1418–1421, Aug 2016.

[29] C. Puente-Baliarda, J. Romeu, R. Pous, and A. Cardama, "On the behavior of the sierpinski multiband fractal antenna," *IEEE Transactions on Antennas and Propagation*, vol. 46, pp. 517–524, Apr 1998.

[30] D. Kumar, Manmohan, and S. Ahmed, "Modified ring shaped sierpinski triangle fractal antenna for c-band and x-band applications," in *2014 International Conference on Computational Intelligence and Communication Networks*, pp. 78–82, Nov 2014.

[31] R. Choudhary, S. Yadav, P. Jain, and M. M. Sharma, "Full composite fractal antenna with dual band used for wireless applications," in *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 2517–2520, Sept 2014.

[32] F. Peiravian, A. Kermanshah, and S. Derrible, "Spatial data analysis of complex urban systems," in *2014 IEEE International Conference on Big Data (Big Data)*, pp. 1–6, Oct 2014.

[33] J. Zhang, H. Zhao, G. Luo, and Y. Zhou, "The study on fractals of internet router-level topology," in *2008 The 9th International Conference for Young Computer Scientists*, pp. 2743–2747, Nov 2008.

[34] D. Ashlock and J. Tsang, "Evolving fractal art with a directed acyclic graph genetic programming representation," in *2015 IEEE Congress on Evolutionary Computation (CEC)*, pp. 2137–2144, May 2015.

[35] M. Kharbanda and N. Bajaj, "An exploration of fractal art in fashion design," in *2013 International Conference on Communication and Signal Processing*, pp. 226–230, April 2013.

[36] S. Xu, J. Yang, Y. Wang, H. Liu, and J. Gao, "Application of fractal art for the package decoration design," in *2009 IEEE 10th International Conference on Computer-Aided Industrial Design Conceptual Design*, pp. 705–709, Nov 2009.

[37] H. Abelson and A. DiSessa, *Turtle Geometry*. Cambridge, MA, USA: MIT Press, 1981.

[38] T. Ju, S. Schaefer, and R. Goldman, "Recursive turtle programs and iterated affine transformations," *Computers & Graphics*, vol. 28, no. 6, pp. 991 – 1004, 2004.

[39] Valiant Designs Limited, "The valiant turtle," 1983.

[40] "Turtle (robot)." From Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Turtle_(robot).

[41] F. D. Rold, "Deterministic chaos in mobile robots," in *2015 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–7, July 2015.

[42] K. Sugawara and T. Watanabe, "A study on foraging behavior of simple multi-robot system," in *IEEE 2002 28th Annual Conference of the Industrial Electronics Society. IECON 02*, vol. 4, pp. 3085–3090 vol.4, Nov 2002.

[43] F. Aznar, M. Pujol, and R. Rizo, *L-System-Driven Self-assembly for Swarm Robotics*, pp. 303–312. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[44] S. K. Lee, S. P. Fekete, and J. McLurkin, "Geodesic topological voronoi tessellations in triangulated environments with multi-robot systems," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3858–3865, Sept 2014.

[45] J. Alonso-Mora, A. Breitenmoser, M. Rufli, R. Siegwart, and P. Beardsley, "Multi-robot system for artistic pattern formation," in *2011 IEEE International Conference on Robotics and Automation*, pp. 4512–4517, May 2011.

[46] H. Guo, Y. Meng, and Y. Jin, "Swarm robot pattern formation using a morphogenetic multi-cellular based self-organizing algorithm," in *2011 IEEE International Conference on Robotics and Automation*, pp. 3205–3210, May 2011.

[47] A. Douady and J. Hubbard, *Étude dynamique des polynômes complexes*. No. v. 1-2 in Publications mathématiques d'Orsay, Université de Paris-Sud, Département de Mathématique, 1984.

[48] E. N. Lorenz, "Deterministic nonperiodic flow," *Journal of the Atmospheric Sciences*, vol. 20, no. 2, pp. 130–141, 1963.

[49] P. Addison, *Fractals and Chaos: An illustrated course*. Taylor & Francis, 1997.

[50] P. Lévy, *Les Courbes Planes Ou Gauches Et Les Surfaces Composées de Parties Semblables Au Tout*. J. École Poly., 1938.

[51] Cantor, "Ueber unendliche, lineare punktmannichfaltigkeiten," *Mathematische Annalen*, vol. 15, pp. 1–7, 1879.

[52] J. Kutyowski and F. Meyer auf der Heide, "Optimal strategies for maintaining a chain of relays between an explorer and a base camp," *Theoretical Computer Science*, vol. 410, pp. 3391–3405, Aug. 2009.

[53] B. Degener, B. Kempkes, P. Kling, and F. M. a. d. Heide, "A continuous, local strategy for constructing a short chain of mobile robots," in *Structural Information and Communication Complexity* (B. Patt-Shamir and T. Ekim, eds.), no. 6058 in Lecture Notes in Computer Science, pp. 168–182, Springer Berlin Heidelberg, Jan. 2010.

[54] F. M. a. d. Heide and B. Schneider, "Local strategies for connecting stations by small robotic networks," in *Biologically-Inspired Collaborative Computing* (M. Hinchey, A. Pagnoni, F. J. Rammig, and H. Schmeck, eds.), no. 268 in IFIP The International Federation for Information Processing, pp. 95–104, Springer US, Jan. 2008.

[55] M. Dynia, J. Kutylowski, F. Meyer auf der Heide, and J. Schrieb, "Local strategies for maintaining a chain of relay stations between an explorer and a base station," in *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, (New York, NY, USA), pp. 260–269, ACM, 2007.

[56] Y. Litus and R. Vaughan, "Fall in! sorting a group of robots with a continuous controller," in *2010 Canadian Conference on Computer and Robot Vision (CRV)*, pp. 269–276, May 2010.

[57] D. Krupke, M. Hemmer, J. McLurkin, Y. Zhou, and S. Fekete, "A parallel distributed strategy for arraying a scattered robot swarm," in *Intelligent Robots*

and Systems (IROS), 2015 IEEE/RSJ International Conference on, pp. 2795–2802, Sept 2015.

[58] Y. Zhou, R. Goldman, and J. McLurkin, "An asymmetric distributed method for sorting a robot swarm," *IEEE Robotics and Automation Letters*, vol. 2, pp. 261–268, Jan. 2017.

[59] Y. Zhou, "Swarm Robotics: Measurement and Sorting," Master's thesis, Rice University, Houston, TX, USA, 2005.

[60] W. M. Spears, D. F. Spears, J. C. Hamann, and R. Heil, "Distributed, physics-based control of swarms of vehicles," *Autonomous Robots*, vol. 17, no. 2, pp. 137–162, 2004.

[61] M. Gardner, "Four-bug problem," *Scientific American*, vol. 213, July 1965.

[62] E. W. Weisstein, "Whirl." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Whirl.html.

[63] J. McLurkin, *Stupid Robot Tricks: A Behavior-Based Distributed Algorithm Library for Programming Swarms of Robots*. S.M. thesis, Massachusetts Institute of Technology, 2004.

[64] S. Poduri and G. S. Sukhatme, "Achieving connectivity through coalescence in mobile robot networks," in *International Conference on Robot Communication and Coordination*, Oct. 2007.

[65] J. McLurkin, J. Smith, J. Frankel, D. Sotkowitz, D. Blau, and B. Schmidt, "Speaking swarmish: Human-Robot interface design for large swarms of autonomous mobile robots," in *Proceedings of the AAAI Spring Symposium*, 2006.

[66] R. Olfati-Saber, "Flocking for multi-agent dynamic systems: algorithms and theory," *IEEE Transactions on Automatic Control*, vol. 51, no. 3, pp. 401–420, 2006.

[67] S. Zhou and S. Zhang, "Lateral stability control on tractor semi-trailer based on anti-jackknife apparatus," in *2014 IEEE Conference and Expo Transportation Electrification Asia-Pacific (ITEC Asia-Pacific)*, pp. 1–6, Aug 2014.

[68] M. Bouteldja, A. Koita, V. Dolcemascolo, and J. C. Cadiou, "Prediction and detection of jackknifing problems for tractor semi-trailer," in *2006 IEEE Vehicle Power and Propulsion Conference*, pp. 1–6, Sept 2006.

[69] E. W. Weisstein, "Archimedes' spiral." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/ArchimedesSpiral.html.

[70] J. L. Jones, "Robots at the tipping point: the road to iRobot Roomba," *IEEE Robotics Automation Magazine*, vol. 13, pp. 76–78, March 2006.

[71] Z. Najdovski, C. Mawson, H. Trinh, and S. Nahavandi, "Solution to robotic landmine detection through use of path planning and motor control," in *2006 World Automation Congress*, pp. 1–6, July 2006.

[72] C. Peters, "An interesting problem to 'bug' your students with," *Alabama Journal of Mathematics*, vol. 3, no. 2, pp. 25–33, 1979.

[73] S. J. Chapman, J. Lottes, and L. N. Trefethen, "Four bugs on a rectangle," *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 467, no. 2127, pp. 881–896, 2011.

[74] J. A. Marshall, M. E. Broucke, and B. A. Francis, "Formations of vehicles in cyclic pursuit," *IEEE Transactions on Automatic Control*, vol. 49, pp. 1963–1974, Nov 2004.

[75] "Lorenz system." From Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Lorenz_system.

[76] M. Gardner, "Mathematical games. nine titillating puzzles, the answers to which will be given next month," *Sci. Am*, vol. 197, pp. 140–146, 1957.

[77] E. W. Weisstein, "Mice problem." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/MiceProblem.html.

[78] M. Klamkin and D. Newman, "Cyclic pursuit or "the three bugs problems"," *American Mathematical Monthly*, pp. 631–639, 1971.

[79] F. Behroozi and R. Gagnon, "Cyclic pursuit in a plane," *Journal of Mathematical Physics*, vol. 20, no. 11, pp. 2212–2216, 1979.

[80] D. Nester, "Beetle centers of triangles." Bluffton University. http://www.bluffton.edu/math/dept/seminar_docs/BeetleCenters/.

[81] A. M. Bruckstein, N. Cohen, and A. Efrat, *Ants, crickets and frogs in cyclic pursuit.* Technion-Israel Institute of Technology. Center for Intelligent Systems, 1991.

[82] A. Bernhart, "Curves of pursuit," *Scripta Math*, vol. 20, pp. 125–141, 1954.

[83] A. Bernhart, "Curves of pursuit-II," *Scripta Math*, vol. 23, pp. 49–65, 1957.

[84] A. Bernhart, "Curves of general pursuit," *Scripta Math*, vol. 24, pp. 189–206, 1959.

[85] I. J. Good, "Pursuit curves and mathematical art," *The Mathematical Gazette*, vol. 43, no. 343, p. 3435, 1959.

[86] P. J. Nahin, *Chases and escapes: the mathematics of pursuit and evasion*. Princeton University Press, 2012.

[87] J. A. Marshall, M. E. Brouke, and B. A. Francis, "A pursuit strategy for wheeled-vehicle formations," in *42nd IEEE International Conference on Decision and Control (IEEE Cat. No.03CH37475)*, vol. 3, pp. 2555–2560 Vol.3, Dec 2003.

[88] Y. Liu, K. M. Passino, and M. M. Polycarpou, "Stability analysis of m-dimensional asynchronous swarms with a fixed communication topology," *IEEE Transactions on Automatic Control*, vol. 48, pp. 76–95, Jan 2003.

[89] V. Gazi and K. M. Passino, "Stability analysis of swarms," *IEEE Transactions on Automatic Control*, vol. 48, pp. 692–697, April 2003.

[90] V. Gazi, *Stability analysis of swarms*. PhD thesis, The Ohio State University, Columbus, OH, 2002.

[91] J. P. La Salle, *The stability of dynamical systems*. SIAM, 1976.

[92] M. W. Hirsch, S. Smale, and R. L. Devaney, *Differential equations, dynamical systems, and an introduction to chaos*. Academic press, 2012.

[93] L. Perko, *Differential equations and dynamical systems*, vol. 7. Springer Science & Business Media, 2013.

[94] F. Verhulst, *Nonlinear differential equations and dynamical systems*. Springer Science & Business Media, 2006.

[95] S. Wiggins, *Introduction to applied nonlinear dynamical systems and chaos*, vol. 2. Springer Science & Business Media, 2003.

[96] J. P. Desai, J. Ostrowski, and V. Kumar, "Controlling formations of multiple mobile robots," in *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, vol. 4, pp. 2864–2869 vol.4, May 1998.

[97] S. Poduri and G. S. Sukhatme, "Constrained coverage for mobile sensor networks," in *IEEE International Conference on Robotics and Automation*, (New Orleans, LA), pp. 165–172, May 2004.

[98] M. Zavlanos and G. Pappas, "Potential Fields for Maintaining Connectivity of Mobile Networks," *IEEE Transactions on Robotics*, vol. 23, pp. 812–816, Aug. 2007.

[99] R. Williams and G. Sukhatme, "Locally constrained connectivity control in mobile robot networks," in *2013 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 901–906, May 2013.

[100] M. Zavlanos, A. Jadbabaie, and G. Pappas, "Flocking while preserving network connectivity," in *2007 46th IEEE Conference on Decision and Control*, pp. 2919–2924, Dec. 2007.

[101] M. A. Hsieh, A. Cowley, V. Kumar, and C. J. Taylor, "Maintaining network connectivity and performance in robot teams," *Journal of Field Robotics*, vol. 25, pp. 111–131, Jan. 2008.

[102] D. Zhang, L. Mao, Z. Li, and J. Chen, "Infrared communication link maintaining method for multiple mobile microrobots," in *2013 IEEE International*

Conference on Robotics and Biomimetics (ROBIO), pp. 2269–2273, Dec. 2013.

[103] M. Egerstedt and X. Hu, "Formation constrained multi-agent control," *IEEE Transactions on Robotics and Automation*, vol. 17, pp. 947–951, Dec. 2001.

[104] M. Ji and M. Egerstedt, "Distributed Coordination Control of Multiagent Systems While Preserving Connectedness," *IEEE Transactions on Robotics*, vol. 23, pp. 693–703, Aug. 2007.

[105] E. W. Weisstein, "Logarithmic spiral." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/LogarithmicSpiral.html.

[106] "Rotational symmetry." From Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Rotational_symmetry.

[107] J. H. Conway, H. Burgiel, and C. Goodman-Strauss, *The Symmetries of Things*. CRC Press, Apr. 2016.

[108] "Archimedean spiral." From Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Archimedean_spiral.

[109] "Involute." From Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Involute#Examples.

[110] I. A. Sucan, M. Moll, and L. E. Kavraki, "The open motion planning library," *IEEE Robotics Automation Magazine*, vol. 19, pp. 72–82, Dec 2012.

[111] A. Becker, "Swarmcontrol: Massive manipulation." http://www.swarmcontrol.net.

[112] T. Laue, K. Spiess, and T. Röfer, "Simrobot–a general physical robot simulator and its application in robocup," in *Robot Soccer World Cup*, pp. 173–183, Springer, 2005.

[113] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "Usarsim: a robot simulator for research and education," in *Robotics and Automation, 2007 IEEE International Conference on*, pp. 1400–1405, IEEE, 2007.

[114] J. McLurkin, A. J. Lynch, S. Rixner, T. W. Barr, A. Chou, K. Foster, and S. Bilstein, "A low-cost multi-robot system for research, teaching, and outreach," *Proc. of the Tenth Int. Symp. on Distributed Autonomous Robotic Systems DARS-10, October*, 2010.

[115] N. G. Chalhoub and A. G. Ulsoy, "Dynamic simulation of a flexible robot arm and controller," in *1984 American Control Conference*, pp. 631–637, June 1984.

[116] S. M. Megahed and S. M. Megahed, *Principles of robot modelling and simulation.* J. Wiley, 1993.

[117] A. T. Miller and P. K. Allen, "Graspit! a versatile simulator for robotic grasping," *IEEE Robotics & Automation Magazine*, vol. 11, no. 4, pp. 110–122, 2004.

[118] V. Tikhanoff, A. Cangelosi, P. Fitzpatrick, G. Metta, L. Natale, and F. Nori, "An open-source simulator for cognitive robotics research: the prototype of the icub humanoid robot simulator," in *Proceedings of the 8th workshop on performance metrics for intelligent systems*, pp. 57–61, ACM, 2008.

[119] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[120] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, vol. 2, pp. 995–1001, IEEE, 2000.

[121] I. A. Şucan and L. E. Kavraki, "Kinodynamic motion planning by interior-exterior cell exploration," in *Algorithmic Foundation of Robotics VIII*, pp. 449–464, Springer, 2009.

[122] B. Gipson, M. Moll, and L. E. Kavraki, "Resolution independent density estimation for motion planning in high-dimensional spaces," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pp. 2437–2443, IEEE, 2013.

[123] A. Becker, *Ensemble control of robotic systems*. PhD thesis, University of Illinois at Urbana-Champaign, 2012.

[124] O. Michel, "Cyberbotics ltd. webots: professional mobile robot simulation," *International Journal of Advanced Robotic Systems*, vol. 1, no. 1, p. 5, 2004.

[125] R. Vaughan, "Massively multi-robot simulation in stage," *Swarm intelligence*, vol. 2, no. 2, pp. 189–208, 2008.

[126] M. Dorigo, D. Floreano, L. M. Gambardella, F. Mondada, S. Nolfi, T. Baaboura, M. Birattari, M. Bonani, M. Brambilla, A. Brutschy, *et al.*, "Swarmanoid: a novel concept for the study of heterogeneous robotic swarms," *IEEE Robotics & Automation Magazine*, vol. 20, no. 4, pp. 60–71, 2013.

[127] P. Wolfgang, *Design patterns for object-oriented software development*. Reading, Mass.: Addison-Wesley, 1994.

[128] P. J. Leach, M. Mealling, and R. Salz, "A universally unique identifier (uuid) urn namespace," RFC 4122, RFC Editor, July 2005.

[129] M. Rubenstein, C. Ahler, and R. Nagpal, "Kilobot: A low cost scalable robot system for collective behaviors," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pp. 3293–3298, IEEE, 2012.

[130] A. Kröeller, S. P. Fekete, D. Pfisterer, and S. Fischer, "Deterministic boundary recognition and topology extraction for large sensor networks," in *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, 2006.

[131] Y. Zhou, H. Li, and J. McLurkin, "Physical bubblesort," 2014. Presented at International Symposium on Distributed Autonomous Robotic Systems (DARS), November 2–5, Daejeon Convention Center, Daejeon, Korea.

[132] E. Olson, "AprilTag: A robust and flexible visual fiducial system," in *2011 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3400–3407, May 2011.